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. We also examine protocols that store self-verifying data (i.e. data that cannot be undetectably altered). Existing protocols can use self-verifying data to reduce the number of servers required to tolerate faults but because SBQ-L already uses the minimum possible number of servers for its semantics, self-verifying data provides no advantage. 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 [17]. 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

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

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 [16] 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 those 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 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 [14]. 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.

	Existing Protocols	SBQ-L	Bound on server count
confirmable, generic	$4f+1$, safe [11, 12][2],[14][1] $4f+1$, partial-atomic [16][2]	**$3f+1$, atomic[2]**	**$\geq 3f+1$ for safe or stronger semantics**
confirmable, self-verifying	$3f+1$, regular [11],[14][1]; $3f+1$, atomic [12],[5][1,2]	$3f+1$, atomic[2]	
non-confirmable, generic	$3f+1$, safe [14]	**$2f+1$, regular[2]**	**$\geq 2f+1$ for safe or stronger semantics**
non-confirmable, self-verifying	$2f+1$, regular [14]	$2f+1$, regular[2]	

(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 bold*

The SBQ-L protocol uses additional communication compared to previous protocols to reduce the number of servers and improve consistency semantics. These messages are not a performance bottleneck however: they are limited to one message per server for each read when no write is concurrent with the read; otherwise the number of additional messages per server is proportional to the number of concurrent writes. Section 7.3 presents measurements of the latency increase due to concurrent writes.

Like other quorum protocols, SBQ-L guarantees correctness by ensuring that reads and writes intersect in a sufficient number of servers. But 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

[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.

protocols access a subset of servers on each operation for two reasons: to tolerate server faults and to 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. SBQ-L contacts a large fraction of servers because it uses the minimal 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 proves bounds on the number of servers required to implement these semantics. We discuss self-verifying data in Section 5 and then examine non-confirmable semantics. In Section 6, we present a minimal protocol for these semantics and prove it correct. In Section 7, we show how to tolerate faulty clients, discuss the trade-offs between bandwidth and concurrency and show how to avoid live-lock or memory problems during concurrent execution. Section 8 discusses related work and we conclude in Section 9.

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 [14]. Initially, we restrict our attention to server failures and assume that clients are correct. We relax this assumption in Section 7.1.

2.2 Consistency Semantics

Consistency semantics define system behavior in the presence of concurrency. We first review Lamport's definitions of safe, regular, and atomic semantics. 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 [16].

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 6 and show that the resulting protocols with non-confirmable writes require fewer servers.

```
W1  Write(D) {
W2          send (QUERY_TS) to all servers
W3          loop {
W4                  receive answer (TS, ts) from server s
W5                  current[s] := ts
W6          } until the current[] array contains q_w answers.
W7          max_ts := max{current[]}
W8          my_ts := min{t ∈ C_ts : max_ts < t ∧ last_ts < t}
            // my_ts ∈ C_ts is larger than all answers and previous timestamp
W9          last_ts := my_ts
W10         send (STORE, D, my_ts) to all servers.
W11         loop {
W12                 receive answer (ACK,my_ts) from server s
W13                 S := S ∪ {s}
W14         } until |S| ≥ q_w // q_w servers answered
W15 }

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 > largest[s].ts then largest[s] := (D, ts)
R6                  if s ∉ S then // we call this event an "entrance"
R7                          S := S ∪ {s}
R8                          T := the f + 1 largest timestamps in largest[]
R9                          for all isvr, for all jtime ∉ T, delete answer[isvr, jtime]
R10                         for all isvr,
R11                                 if largest[isvr].ts ∈ T  then answer[isvr, largest[isvr].ts] := largest[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*

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 W8, 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 W10 and waits for acknowledgments at lines W11 to W14. The Read() function queries a read quorum 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

[3] We describe the more general quorum SBQ-L protocols in the Appendix.

variable	initial value	notes
q_w	$\left\lceil \frac{n+f+1}{2} \right\rceil$	Size of the write quorum
q_r	$\left\lceil \frac{n+3f+1}{2} \right\rceil$	Size of the read quorum
C_{ts}	Set of timestamps for client c	The sets used by different clients are disjoint
$last_ts$	0	Largest timestamp written by a particular server. This is the only variable that is maintained between function calls ("static" in C).
$largest[]$	$\emptyset$	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 client has received an answer

Fig. 2. *Client variables*

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 $f+2$ answers from each server. We show in Section 3.1 that the protocol is correct.

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.

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 sound, timely answers for the reader to identify a value that meets the consistency guarantee of the system (e.g., using a majority rule). 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. 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 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 7.1 we show how to adapt this protocol for faulty clients.

3.1 Correctness

To better understand the intuition behind the correctness proofs, it helps to think about the constraints on the read protocol. The read protocol decides on a value that is proposed by a *voucher set* of q_w servers, so the first constraint is that after a write has completed all voucher sets much contain at least one correct server. The second constraint pertains to liveness: the protocol must ensure that the reader eventually receives replies from q_w correct servers so it can complete.

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. Hence no later read can return an earlier timestamp.

(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. It follows that any read that starts after t_1 has to return a timestamp of at least ts_1. □

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

Write. The Write() function is trivially live because its waits (in steps W6 and W14) expect $q_w = \lceil (n+f+1)/2 \rceil$ answers and $q_w \leq n - f$ so these answers are guaranteed to eventually arrive.
Read. Even though it only tracks $f + 2$ different timestamps simultaneously (lines R11 and R12), the Read() function is live. Note that there are at most n entrances. Consider the last one, i.e., the last time line R7 of Read() is executed. Consider the largest `largest[].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 (because these data items are kept in `largest[]`). ts_{max} is in T because T contains the $f + 1$ largest timestamps in `largest[]`. 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.

The reader will receive replies from $q_r = \lceil \frac{n+3f+1}{2} \rceil$ servers. Because $q_w \leq q_r - f$, there are enough correct servers to guarantee that the read for that timestamp will eventually complete.
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. □

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

This lemma 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.

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 (i.e. protocols that use random coin flips in order to guarantee safe semantics). 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. Since SBQ-L does not use cryptography, we know that self-verifying data is not necessary for protocols that use the minimal number of servers.

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. □

5 Self-Verifying Data

In previous work, protocols using self-verifying data often required f fewer servers than otherwise [11, 14]. It is easy to understand why protocols using self-verifying data might be easier: the signatures make it possible to detect faulty servers that lie—as long as they don't lie by replaying some older value.

It was not known until now whether self-verifying data really made a fundamental difference or if protocols using only generic (i.e. non-self-verifying) data could be made to perform as well. In this paper, we show that self-verifying data has no effect on the number of servers required to solve the problem. To do this we show a protocol, SBQ-L, that provides atomic semantics using $3f + 1$ servers. We then show that no fault-tolerant storage protocol can match these semantics using fewer servers—even if using self-verifying data.

Although self-verifying data has no impact on the minimal number of servers, it may be useful for other properties of these protocols such as the number of messages exchanged or the ability to restrict access to the shared variables.

It is also interesting to note that this equality in power between generic and self-verifying data carries over to the non-confirmable protocols described in the next section. Using that weaker guarantee, it is still the case that no program using self-verifying data can satisfy safe semantics using fewer servers than the non-confirmable version of SBQ-L.

6 Non-Confirmable Protocols

In Sections 3 and 4 we have limited ourselves to protocols in which the writer can determine when its writes complete. We now explore *non-confirmable* writes in which the writer cannot locally determine when its writes complete, even though write completion is still a well-defined event.

6.1 Definition

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. Note that confirmable 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 although all writes still eventually complete, then we call the protocol *non-confirmable*. Non-confirmable protocols cannot provide blocking writes. The SBQ protocol [14], 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.

As an example of a system where a non-confirmable 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-confirmable semantics are appropriate.

6.2 Protocol

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.

Since in a non-confirmable protocol the writer is not required to know when the write completes, we can remove lines W11 to W14 of the `Write()` function in which the writer waits for acknowledgments. The STORE messages sent earlier (at line W10) 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 q_w (now used only in line R13) to $\lceil \frac{n+1}{2} \rceil$ instead of $\lceil \frac{n+f+1}{2} \rceil$ previously. We also change q_r to $\lceil \frac{n+2f+1}{2} \rceil$ instead of $\lceil \frac{n+3f+1}{2} \rceil$. These changes are possible because eliminating the acknowledgments eliminates a constraint on the overlap of read and write quorums [14].

Recall that in non-confirmable 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 $\lceil \frac{n+1}{2} \rceil$ *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. As shown in Theorem 4, $2f + 1$ is the optimal number of servers for non-confirmable protocols.

Pierce [16] presents a general technique to transform any regular protocol into one that satisfies atomic semantics. This technique, however, only works for confirmable protocols and therefore does not apply to this case.

Due to space constraints, we state the main theorems here. Proofs are included in the Appendix.

6.3 Correctness

Theorem 3. *The non-confirmable threshold SBQ-L protocol is live and provides regular semantics.*

6.4 Lower Bounds

We prove lower bounds for non-confirmable protocols. The minimum number of servers for safe semantics is $2f+1$, as opposed to $3f + 1$ for confirmable protocols.

Theorem 4. *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can satisfy the safe semantics for distributed shared memory using* $2f$ *servers.*

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 that use integrity checks in their messages since faulty servers have all the necessary information to create the messages they send.

7 Practical Considerations

Our basic Listener protocol allows the SBQ-L protocols to use the optimal number of servers but (1) it does not handle faulty clients, (2) the communication pattern it requires causes more messages to be exchanged than in other protocols, (3) supporting a large number of clients is costly and (4) the reader stores messages in memory before deciding. In the next subsections we show how to handle faulty clients, quantify the number of additional messages, experimentally measure the effect of additional messages, scale up to large numbers of clients, discuss the protocol latency, and show an upper bound on memory usage.

7.1 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, violating liveness, or (2) faulty clients can violate the read protocol to waste server resources.

Liveness. Faulty writers can prevent future read attempts from terminating by preventing any quorum of servers from having 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.

To avoid poisonous writes, we introduce a "writer" private key, shared by all the writers but not accessible to the servers – servers only have the corresponding public key. Clients sign their write requests with this key. The

modified server protocol is shown in Figure 3. The written values are stored at the servers in the variables D_0 and ts_0. As before, these variables are updated whenever a server receives a newer value and the servers implement their part of the listeners protocol. To handle poisonous writes, we modify servers to forward all valid requests to all other servers in addition to the normal processing. If a server receives two valid requests with an identical timestamp, it considers the one whose data comes first lexicographically to have a lower timestamp (this allows servers to agree even when the faulty writer is trying to poison the write). The variables D_0 and ts_0 represent the server's storage. The variable *start-listening*[c] stores the value that ts_0 had when the server received a read request from a particular client c. We use the notation $\{x\}_{writer}$ for "the message x, signed with the private key *writer*". The protocol shown is appropriate for confirmable writes; for non-confirmable writes line S3 can be omitted.

This new protocol guarantees that the system recovers once poisonous writes end. Let (ts', D') be the highest-valued message sent in a write by a faulty client. The system recovers when $\lceil \frac{n+1}{2} \rceil$ correct servers have finished processing the (ts', D') message (either from the client directly or from an echo message). Note that in the case of correct clients, recovery unsurprisingly coincides with the completion of the write. With this modification, both confirmable and non-confirmable SBQ-L provide atomic (respectively regular) semantics despite faulty clients.

A related win is that slow writers–even writers that crash—will not delay nor prevent the completion of the write since the servers themselves propagate the write information. With this modification all operations from correct clients will eventually complete even if some servers are faulty and regardless of the speed of the other clients, even if some or all other clients crash.

```
S1  Store({D, ts}_writer) {
S2          if the signature is invalid then ignore this message.
S3          send (ACK,ts) back to the sender, unless it is a server.
S4          for each listening client c
S5                  if ( start-listening[c] ≤ ts ) send (VALUE,D,ts) to c
S7          if ((ts, D) > (ts_0, D_0)) then
S8                  (ts_0, D_0) := (ts, D) // store the new value
S9                  send (STORE,{D, ts}_writer) to all servers.
S10 }
```

Fig. 3. *Server protocol for confirmable SBQ-L, adapted to handle faulty clients*

Resource Exhaustion. A faulty reader can neglect to notify the servers that the read has completed and therefore force the server to continue that read operation forever. One way to mitigate this problem is to restrict each reader to a single read at a time. Similarly in a "real" implementation, a reader would only be allowed to access a certain number of variables at a time. A faulty reader can still cause significant amounts of unnecessary traffic by sending the read request for some variable that is often written.

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 imbalance makes the denial of service attack possible.

A natural solution therefore is to allow the servers to unilaterally stop the read when they feel that they have performed "enough" work, forcing the clients to send messages if they want to continue the read. The imbalance has now disappeared, and instead we introduce a new parameter: the system designer can define how much server work is "enough" to make the system more or less resilient to resource exhaustion attacks.

In the new protocol, when a server decides that some read has worked "enough", it sends a NAK message to the reader and stops forwarding write notices. If the reader has not completed the read by the time it receives the NAK, it sends a READ message to that server to continue the read (other servers are unaffected).

The servers can decide they have worked "enough" after a fixed number of messages have been forwarded to the reader, if no message was received from the reader in some time, or a combination of the two. The servers may also accept READ messages from an already-reading server as a "heartbeat" message to keep the read connection open.

Because of the asynchronous nature of the network, it is possible that correct readers, too, receive NAK messages. These messages do not affect the safety of the protocol, but the liveness is now only guaranteed if there is a "good period" during which messages flow fast enough for the read to complete before the reader receives a NAK. The designer's choice of when servers should interrupt reads influences how long that period has to last.

7.2 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-confirmable case and $4n$ messages in the confirmable case, where n is the number of servers. This communication is identical to previous results: the non-confirmable SBQ protocol [14] 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.

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 the next section.

7.3 Experimental evaluation of overhead

We construct a simple prototype to study the overhead of the extra messages used to deal with concurrency in SBQ-L, described in the Appendix. We find, as expected, that increasing concurrency has a measurable but modest effect on the latency of the reads.

7.4 Maximum Throughput

A goal of a BQS architecture is to support a high throughput for a low system cost. The maximum throughput of a BQS architecture is proportional to the inverse of its *load factor* [15], which is the minimal access probability of the busiest server, minimizing over the strategies. 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.

There are two ways for a given quorum system to increase its throughput: the system can use more powerful servers or it can add additional servers.

First, the use of more powerful servers for a system with a given load factor can provide a linear improvement in the system's maximum throughput. In systems that attempt to satisfy the assumption of failure independence by constructing different servers with different implementations of software, this approach has the significant advantage of minimizing the number of software implementations. Although increasing server size linearly increases maximum throughput, server cost may increase more than linearly with server size; furthermore, there is a finite maximum practical server size.

Second, the use of additional servers of a given power can reduce the load factor of a BQS system. For example, a SBQ-L system that tolerates one fault and that supports confirmable writes has a minimum size of 4 servers; this minimum-size system has a load factor of 1.0. If 16 server implementations are available, SBQ-L's load factor can be reduced to 0.8125, and in the limit, if a large number of servers can be constructed, SBQ-L's load factor approaches 0.75. Compared to the f-masking protocol, which can also improve its load factor by adding servers, these load factors are 25%, 31%, 50% higher than the 0.8, 0.62, and 0.5 load factors for $n = 5$ (f-masking's minimum configuration), $n = 16$, and $n = \infty$. The Grid construction [11] provides a potentially better load factor of $\frac{(2f+2)\sqrt{n}-(2f+1)}{n}$. At its minimum configuration size of $n = 16$, its load factor of 0.8125 matches that of 16-node SBQ-L, but for very large n its load factor can approach $\frac{2f+2}{\sqrt{n}}$. Although increasing n increases maximum throughput, this increase is much less than linear for all BQS algorithms. Thus, cost considerations may limit the

extent to which this technique can be exploited in practical systems. When comparing across algorithms, comparing load factors at small to medium n may be more relevant than comparing asymptotic load factors.

In evaluating a system's throughput versus cost trade-offs, both its load factor and total number of machines n must be considered. If system A has a higher load factor but a smaller n than system B, it may requires more expensive servers but require fewer of them and may have a lower total system cost. These trade-offs will tend to favor system A when software costs are high relative to hardware costs. They will tend to favor system B when hardware sufficient to support the needed throughput under system A's load factor is much more expensive than the hardware needed for the target throughput under system B's load factor.

7.5 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 second term is due to the fact that there are at most n entrances.

7.6 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 $n(f+2)$ units of memory: the set T contains at most $f+1$ elements (line R8) and the `answer[][]` matrix therefore never contains more than n columns and $f+1$ rows (lines R9, R11 and R12). An additional n elements are stored in `largest[]`. 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 $n(f+2)$.

8 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) [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 [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, 15] 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 failure scenarios).

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.

9 Conclusion

We present two protocols for shared variables, one that provides regular 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, 14, 17] 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, M. Reiter, 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. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.
15. M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.
16. 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.
17. 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 Non-Confirmable Semantics

We show the complete proof of the non-confirmable SBQ-L protocol and the lower bound of $2f+1$ servers for safe protocols with non-confirmable writes. This proof is similar to that of Section 3.1.

A.1 Correctness

Theorem 5. *The non-confirmable threshold SBQ-L protocol is live and provides regular semantics.*

Lemma 6 (Regularity). *The non-confirmable f-threshold SBQ-L satisfies regular semantics, assuming it is live.*

When a read r completes, the reader decides on a value that has been vouched for by $q_w = \lceil \frac{n+1}{2} \rceil$ servers (line R13). By definition, q_w correct servers have seen the latest completed write with respect to r. Since $2q_w > n$, 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 sends the value of the completed write, the value of a write with a higher timestamp, or both. As a result, subsequent reads will never return a value older than the latest completed write. □

Lemma 7 (Liveness). *The non-confirmable f-threshold SBQ-L is live.*

The beginning of the proof for liveness is identical to the proof for the confirmable case in Section 3.1, showing that all operations eventually terminate. The proof of the read operation needs to be adapted slightly. In the last step, showing that the reader will eventually receive sufficiently many echos, the quorum size must be modified as follows. The write protocol eventually reaches all $n-f$ correct servers. The read operation contacts $\lceil \frac{n+2f+1}{2} \rceil$ servers and the intersection of the two quorums contains the $\lceil \frac{n+1}{2} \rceil$ correct answers required for completion because $\lceil \frac{n+2f+1}{2} \rceil + n - f \leq n + \lceil \frac{n+1}{2} \rceil$. □

A.2 Bounds

We prove lower bounds for non-confirmable protocols. The minimum number of servers for safe semantics is $2f+1$, as opposed to $3f+1$ for confirmable protocols.

Theorem 6. *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.

Lemma 8. *For all live read protocols using $2f$ servers, for all sets S of f servers and for all reachable quiet system states, there exists at least one execution in which a read is only influenced by all servers in a set S' such that $S' \subseteq S$.*

By contradiction: suppose there exists a live protocol P using $2f$ servers, a set S of f servers and a reachable quiet system state in which all executions of the read protocol are not only influenced by the servers in any S' : $S' \subseteq S$, but instead are also influenced by some other server $x \notin S$. Since $|U - S| = f$, we can suppose that all servers in $U - S$ are faulty. It follows that x is faulty and may crash, therefore the protocol P is not live. □

Lemma 9. *Consider a live read protocol using $2f$ servers. There exist executions for which this protocol does not satisfy safe semantics.*

Intuitively, whenever the reader relies on only f servers it will be fooled if all these servers are faulty. We show this through a more formal explanation below.

Consider the initial state of the system in which the individual servers have states $a_0 \ldots a_{2f-1}$ and the shared variable has value A. We call this "state A". Consider now an execution e of the read protocol in state "A" that

is only influenced by a subset of the servers $0 \ldots f-1$ (Lemma 8 proves that e exists). This execution correctly returns the value A for the shared variable.

Imagine a later snapshot of the same system, when no operation is in progress. The individual servers now have states $b_0 \ldots b_{2f-1}$ and the shared variable has value B. We call this "state B". A correct read should return the value B. Suppose that servers 0 through $f-1$ are faulty and behave as if they were in states $a_0 \ldots a_{f-1}$, and suppose that a new read starts, only influenced by servers $0 \ldots f-1$ (again, Lemma 8 proves that this read exists). The reader will receive the exact same answers in state B as the previous reader did in state A. Because the two executions are indistinguishable, the new read will return the incorrect value A. □

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

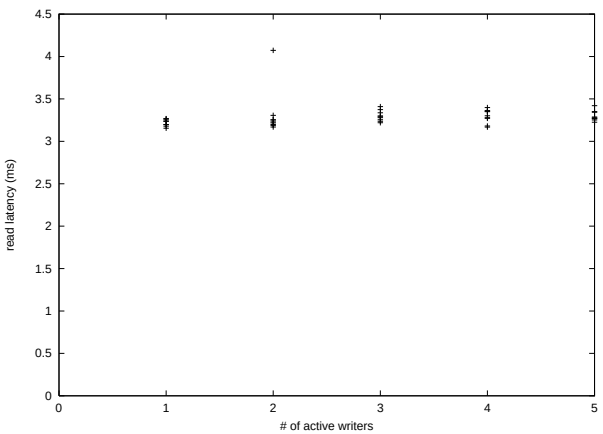
Note that the proof is not limited to the f-threshold model and makes no assumption of deterministic behavior from the protocol. The proof also covers protocols that use integrity checks in their messages since faulty servers have all the necessary information to create the messages they send.

B Experimental evaluation of overhead

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. We implemented the confirmable f-threshold version of SBQ-L.

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-confirmable 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.

C Generalized Confirmable Protocol

The confirmable protocol can be generalized to a fail-prone system instead of the simpler f-threshold case presented in Section 3. For any given fail-prone system $\mathcal{B}$ (defined in Section 2.1), our quorums $Q \in \mathcal{Q}$ and access sets $A \in \mathcal{A}$ 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 through an access set.

$$\forall A \in \mathcal{A}\ \forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : Q \subseteq A - B$$

The Write() function is modified to return once it receives an acknowledgment from a quorum. The modified Read() is presented in Figure 4. 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.

```
W1  Write(D) {
W2          send (QUERY_TS) to all servers
W3          loop {
W4                  receive answer (TS, ts) from server s
W5                  current[s] := ts
W6          } until the ts[] array covers a quorum of servers.
W7          max_ts := max{current[]}
W8          my_ts := min{t ∈ C_ts : max_ts < t ∧ last_ts < t}
            // my_ts is larger than all answers and previous timestamp
W9          last_ts := my_ts
W10         send (STORE, D, my_ts) to all servers.
W11         loop {
W12                 receive answer (ACK,my_ts) from server s
W13                 S := S ∪ {s}
W14         } until ∃Q_w ∈ Q :: Q_w ⊆ S // a quorum servers have sent an ACK
W15 }

R1  (D,ts) = Read() {
R2          send (READ) to all servers in some A ∈ 𝒜.
R3          loop {
R4                  receive answer (VALUE,D, ts) from server s // (possibly more than one answer per server)
R5                  if ts > largest[s].ts then largest[s] := (D, ts)
R6                  if s ∉ S then // we call this event an "entrance"
R7                          S := S ∪ {s}
R8                          T := the f + 1 largest timestamps in largest[]
R9                          for all isvr, for all jtime ∉ T, delete answer[isvr, jtime]
R10                         for all isvr,
R11                                 if largest[isvr].ts ∈ T then answer[isvr, largest[isvr].ts] := largest[isvr]
R12                 if ts ∈ T then answer[s, ts] := (D, ts)
R13         } until ∃D', ts', Q_r :: Q_r ∈ Q ∧ (∀i : i ∈ Q_r : 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 }
```

Fig. 4. *Generalized confirmable SBQ-L protocol*

D Generalized Non-Confirmable Protocol

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

In the non-confirmable 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 through an access set.

$$\forall A \in \mathcal{A}\ \forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : Q \subseteq A - B$$

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. 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.

E Correctness

E.1 Generalized Confirmable SBQ-L

Theorem 7. *The confirmable generalized SBQ-L protocol provides atomic semantics.*

Lemma 10 (Regularity). *The confirmable 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. □

This confirmable protocol, similar to the threshold version, 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 11 (Atomicity). *The confirmable 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:

$$\forall Q_1, Q2 \in \mathcal{Q}\ \forall B \in \mathcal{B} : Q_1 \cap Q_2 \not\subseteq B$$

Similarly, suppose that at some global time t_1, some client c reads timestamp ts_1. A quorum $Q_1 \in \mathcal{Q}$ of servers agree on this timestamp. Since the faulty and remaining machines cannot form a quorum, it follows that any read that starts after t_1 has to return a timestamp of at least ts_1. □

Lemma 12 (Liveness). *All functions of the confirmable 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 `largest[]` associated with each server. Consider the largest `largest[].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). $ts_{max} \in T$ because T contains the $f+1$ largest timestamps in `largest[]`. 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. □

E.2 Generalized Non-confirmable SBQ-L

Theorem 8. *The non-confirmable generalized SBQ-L protocol provides regular semantics.*

Lemma 13 (Regularity). *The non-confirmable 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. Since writes complete when a quorum of correct servers have received them, the weaker consistency requirement for non-confirmable is sufficient. This holds because witness quality guarantees that no quorum can consist entirely of faulty servers.

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 confirmable case. □