Small Byzantine Quorum Systems

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

Abstract

In this paper we present two protocols for asynchronous Byzantine Quorum Systems (BQS) built on top of reliable channels—one for self-verifying data and the other for any data. Our protocols tolerate f Byzantine failures with f fewer servers than existing solutions by eliminating nonessential work in the write protocol and by using read and write quorums of different sizes. Since engineering a reliable network layer on an unreliable network is difficult, two other possibilities must be explored. The first is to strengthen the model by allowing synchronous *networks that use time-outs to identify failed links or machines. We consider running synchronous and asynchronous Byzantine Quorum protocols over synchronous networks and conclude that, surprisingly, "self-timing" asynchronous Byzantine protocols may offer significant advantages for many synchronous networks when network time-outs are long. We show how to extend an existing Byzantine Quorum protocol to eliminate its dependency on reliable networking and to handle message loss and retransmission explicitly.*

1. Introduction

Quorum systems are valuable tools for implementing highly available distributed shared memory. The principle behind their use is that if a shared variable is stored at a set of servers, then read and write operations need only be performed at some set of servers (a *quorum*). The intersection property of quorums ensures that each read has access to the most recently written value of the variable. Any practical use of quorum systems must account for the possibility that some of the servers may be faulty; hence, quorum systems must enforce the intersection property even in the presence of failures. Malkhi and Reiter introduce quorum systems, called *masking quorum systems*, that guarantee data availability in the presence of arbitrary (*Byzantine*) failures [19]. They also introduce a special class of quorum systems, *dissemination quorum systems*, which 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. To tolerate f Byzantine failures, masking quorum systems must include at least $4f + 1$ servers, while dissemination quorum systems need only $3f + 1$ servers to provide the same guarantee.

In this paper, we present two new quorum systems, one for generic data and the other for self-verifying data, that need only $3f + 1$ servers and $2f + 1$ servers, respectively, to tolerate f Byzantine failures. These results apply in the same system model used by Malkhi and Reiter, i.e. one in which communication is authenticated and reliable, but asynchronous.

Our quorums thus use fewer servers to tolerate a given number of failures than previously possible. Reducing the required number of servers is particularly important where Byzantine protocols protect against security breaches of servers [7, 8, 20]. Note that using Byzantine protocols to tolerate security breaches is sound only if server failures are independent, i.e. if breaking into one server does not increase the probability of successfully breaking into others. Achieving such failure independence may require developing and maintaining multiple independent implementations of the server and underlying operating system [26]. Because implementing these multiple variations is expensive, the number of different implementations is, in practice, limited. It is therefore essential to minimize the number of servers needed to tolerate a given number of failures.

We call our new quorum systems *a-masking* and *a-dissemination*, where the leading "a" indicates the distinguishing characteristic of these quorums, namely, that they are asymmetric with respect to the operations they support: reads and writes use quorum of different sizes.

The key insight that allows us to exploit asymmetric quorums is the recognition that assuming reliable communication has different implications for read and write operations. Although reads need a response from a *read quorum* of servers in order to return a reliable value, writes do

This work was supported in part by DARPA/SPAWAR grant N66001-98-8911, the Texas Advanced Technology Program and Tivoli. Alvisi was also supported by an NSF CAREER award (CCR-9734185) and an Alfred P. Sloan Fellowship. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Fellowship.

not need to be explicitly acknowledged by a corresponding *write quorum*: a reliable communication abstraction already guarantees that every value written by a correct client will eventually be stored by every correct server in the write quorum, and the writer itself has no use for the knowledge that the write completed. We call read and write protocols that exploit this insight Small Byzantine Quorum (*SBQ*) protocols.

Reliable asynchronous communication is a common model for Byzantine quorum algorithms [19, 20], and our protocol aggressively exploits that model's properties to improve efficiency. In an asynchronous system, unfortunately, if the underlying network is unreliable then the presence of even crash failures can pose significant challenges to engineering a reliable messaging layer because a message sender cannot distinguish a crashed receiver from a slow one. For example, if an asynchronous reliable messaging layer requires senders to buffer and retransmit unacknowledged messages, a failed receiver can force the system to consume unbounded amounts of buffer memory.

To understand such practical concerns, we explore the trade-offs for building Byzantine quorum systems (BQS) as we vary the properties of the underlying communication infrastructure. In this analysis, we consider not just the SBQ protocols but also existing protocols [4, 19].

We begin by strengthening the reliable and asynchronous communication model to consider systems that implement *reliable and synchronous* communication. Under these assumptions, read and write protocols that tolerate f Byzantine failures require just $2f + 1$ servers for generic data ($f + 1$ for self-verifying data) [4]. However, these protocols are vulnerable to *slow reads*: even a single faulty server can delay each read until a timeout occurs. Unfortunately, for some systems of practical interest, the natural timeout at which network transmission should be abandoned is long compared to the desired performance of read operations. Unexpectedly, our analysis suggests that some systems that assume a reliable and synchronous networks may still choose to use an *asynchronous* BQS protocol such as SBQ. Such systems may use timeouts in the networking layer to bound network retransmission buffers, but they may choose an asynchronous BQS protocol to allow reads to proceed at a rate governed by the speed of the fastest quorum of servers rather than at a rate governed by communication timeouts to failed servers. To address these tradeoffs more generally, we develop a new class of synchronous SBQ protocols, which we call S-SBQ. S-SBQ protocols can be tuned with respect to two parameters: f, the maximum number of faulty servers for which the protocol is safe and live, and t ($t \leq f$), the maximum number of faulty servers for which the protocol is free from slow reads. When $t = 0$, S-SBQ uses the same number of servers as the synchronous protocol described in [4], and when $t = f$, S-SBQ is identical to the asynchronous SBQ protocol.

We then explore the implications of weakening the assumption of asynchronous reliable communication. We consider *authenticated unreliable asynchronous networks*, in which protocols must explicitly manage both server faults and network faults, and show that the quorum systems and protocols introduced by Malkhi and Reiter for reliable asynchronous networks can be easily extended to operate in this weaker model.

In summary, our analysis results in a series of Byzantine quorum systems and protocols over a range of system models, with increasing numbers of servers required to tolerate progressively weaker system models. For generic data, $2f + 1$ servers are needed for synchronous reliable network systems where timeouts are short, $2f + 1$ to $3f + 1$ for synchronous reliable network systems where timeouts are long, $3f + 1$ for asynchronous reliable network systems, and $4f + 1$ for asynchronous unreliable network systems. Self-verifying-data allows systems to be built for each of these scenarios using f fewer servers.

The rest of this paper is organized as follows: Section 2 presents the system model. Section 3 presents the new a-masking and a-dissemination quorum systems. Section 4 discusses the design space of BQS protocols under different system models. Section 5 puts our results in perspective with related work and Section 6 summarizes our conclusions.

2. System Model

We assume the system model commonly adopted by previous works [2, 4, 19, 20, 21] that have applied quorum systems in the Byzantine failure model. In particular, our system 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* $\mathcal{Q} \subseteq 2^U$ is a non empty set of subsets of U, each of which is called a *quorum*. We denote with $\mathcal{Q}_r$ the set of quorums used by read operations (*read quorums*) and with $\mathcal{Q}_w$ and the set of quorums used by write operations(*write quorums*). Any pair of read and write quorums intersect, and $\mathcal{Q} = \mathcal{Q}_r \cup \mathcal{Q}_w$.

Servers can be either *correct* or *faulty*. A correct server follows its specification; a faulty server can arbitrarily deviate from its specification. Following [19], we define a *fail-prone system* $\mathcal{B} \subseteq 2^U$ as a nonempty 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 express 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. We restrict our attention in this work to server failures; clients are

assumed to be correct. Clients communicate with servers over point-to-point channels. In this paper, we consider Byzantine quorum systems for the following models of communication:

Reliable Synchronous A correct process q receives a message from another correct process p if and only if p sent it; furthermore, q can determine that p was the sender of the message. Also, there exists a bound on message delivery time that can be used to timeout failed processes that do not respond to requests [4].

Reliable Asynchronous A correct process q receives a message from another correct process p if and only if p sent it; furthermore, q can determine that p was the sender of the message. However, no bound is assumed on message transmission times [19].

Authenticated Unreliable Asynchronous If a correct process p sends a message infinitely often to another correct process q, then q will eventually receive the message and know that it came from p; a correct process q receives a message only if a correct process p sent the message; and no bound is assumed on message transmission times.

We explicitly state which model is assumed at each point of our discussion.

3. Small Byzantine Quorums

Figures 1 and 2 show our Small Byzantine Quorum (SBQ) protocols for generic and self-verifying data, respectively, under the assumption of reliable asynchronous communication. To write data Δ to a variable v in either protocol, a client first queries a read quorum of servers to choose a timestamp that is larger than the timestamp for any completed write (steps 1-4) and then sends the data and the new timestamp to a write quorum of servers (step 5). To read data, a client queries a read quorum of servers for their most recent values (steps 1-2) and then chooses and returns the valid answer with highest timestamp (step 3-4). Each correct server updates its local variable and timestamp to the values $\langle ts, \Delta \rangle$ received by a client only if ts is larger than the timestamp currently associated with Δ.

A noteworthy aspect of the protocol is that unlike operations on read quorums, an operation on a write quorum does not wait for replies from the servers it contacts. For reliable asynchronous communication, the eventual delivery of all messages sent by a correct client to correct servers is assured, and the write operation can complete at the client without gathering information from the servers to which the write messages have been sent. Note, however, that this means that a client's local write operation may *return* before the global write *completes*. In order to define an order among reads and writes, we say that a global write operation completes when all correct servers in some write quorum have finished processing the STORE messages sent in step 5 of the write() operation defined in Figures 1 and 2. Furthermore, we say that a write operation w_1 *happens before* a write operation w_2 if w_1 ends (according to the above definition) before w_2 starts. A disadvantage of this definition of write completion is that a client issuing a write may not know when the write completes. This is not a problem from a theoretical standpoint, since this knowledge is required by neither safe semantics (provided by the SBQ protocol for generic data) nor regular semantics (provided by the SBQ protocol for self-verifying data) [16]. Furthermore, completion of write operations is both well defined from the point of view of an observer external to the system, and *timely*, in the sense that completion cannot be delayed by faulty servers because it only depends on actions taken by correct processes. Nonetheless, SBQ protocols do carry a price: they do not support the implicit synchronization that can be obtained through write operations that block until the write completes. Fortunately, there are several interesting applications that do not require this implicit synchronization, either because they don't need any synchronization (e.g., in networked sensors [14], nodes producing data often do not need to receive acknowledgments, implicit or explicit, from consumers) or because they only require end-to-end explicit acknowledgments in which clients synchronize by reading values written to various memory locations by other clients. For instance, two clients can communicate using an SBQ protocol in the same way as two pen-pals communicate through regular mail: in both cases, the writer relies on the fact that its message will be eventually received, even if it does not know when. Its counterpart can assure the writer of the receipt of his message by acknowledging it in his next message.

The rest of this section explains this protocol in more detail. We first describe how quorums are constructed and why the SBQ protocols' quorums are small, needing only $3f + 1$ servers in the *f*-threshold case for generic data and $2f+1$ for self-verifying data. We then compare our protocol to existing protocols to identify the differences and explain why these differences allow quorums based on SBQ protocols to be smaller than those of existing protocols for reliable asynchronous communications systems. Finally, we step through the details of the SBQ protocol and provide a proof of its correctness.

3.1. Quorum definition

The key advantage of SBQ protocols over existing Byzantine quorum systems protocols is their reduction in the number of servers required by the system. This reduction stems from the different constraints SBQ places on read and write quorums. Because the protocol places

Write(Δ)

1. send (GET-TS) to all servers.
2. **wait until** received timestamp ts_i from each server s_i in a read quorum.
3. let *last_ts* be the largest received timestamp.
4. choose a new timestamp new_ts that is larger than both *last_ts* and any timestamp previously chosen by this server.
5. send (STORE,Δ, new_ts) to a write quorum of servers.

Δ =**Read()**

1. send (GET) to all servers.
2. **wait until** received pairs $\langle \Delta_i, ts_i \rangle$ from each server s_i in a read quorum Q_r.
3. { *Build a set A' containing all pairs returned by a voucher set of servers* }
   compute $A' = \{\langle \Delta, ts \rangle \mid (\exists B^+ \subseteq Q_r :: (\forall B \in \mathcal{B} : B^+ \not\subseteq B : (\forall s_u \in B^+ :: \Delta_u = \Delta \wedge ts_u = ts)))\}$
4. **if** $A' \neq \emptyset$ **then**
   select the pair $\langle \Delta, ts \rangle$ with the highest timestamp *ts*
   return Δ
   else
   return $\perp$

Figure 1. *SBQ protocol for generic (non-self-verifying) data*

Write(Δ)

1. send (GET-TS) to all servers.
2. **wait until** received timestamp ts_i from each server s_i in a read quorum.
3. let *last_ts* be the largest received timestamp.
4. choose a new timestamp new_ts that is larger than both *last_ts* and any timestamp previously chosen by this server.
5. send (STORE, Δ, new_ts) to a write quorum of servers.

Δ =**Read()**

1. send (GET) to all servers.
2. **wait until** received pairs $\langle \Delta_i, ts_i \rangle$ from each server s_i in a read quorum Q_r.
3. discard all pairs that are not verifiable.
4. select among the remaining pairs the pair $\langle \Delta, ts \rangle$ with the highest timestamp return Δ

Figure 2. *SBQ protocol for self-verifying data*

asymmetric constraints on read and write quorums, it can use asymmetric masking quorums (*a-masking quorums*) for generic data and asymmetric dissemination quorums (*a-dissemination quorums*) for self-verifying data in place of the traditional (symmetric) masking and dissemination quorums [19].

To understand how the protocol's constraints on quorum construction influence the minimum number of services required by a system, consider the simple case of *f*-threshold quorums for self-verifying data under the SBQ protocol and let $|Q_r|$ and $|Q_w|$ denote, respectively, the size of read and write quorums. In order to guarantee safety and liveness for this protocol, there are effectively three constraints that must be met:

SBQ1. $|Q_r| \leq n - f$ (*Availability*)

This constraint is required for step 2 of Read() and step 2 of Write() to be live.

SBQ2. $|Q_r| + |Q_w| - n \geq f + 1$ (*Consistency*)

This constraint is required for the intersection of reads (in step 2 of Read() and step 2 of Write()) and writes (in step 5 of Write()) to be large enough to ensure that each read intersects with each completed write in at least one correct server. This constraint is essential for the safety of the protocol.

SBQ3. $|Q_w| \leq n$ (*Realism*)

The following values meet these constraints: $|Q_w| = \lceil \frac{n+1}{2} \rceil + f$ and $|Q_r| = \lceil \frac{n+1}{2} \rceil$. Substituting this value for $|Q_r|$ into SBQ1 gives $n \geq 2f + 1$.

Similar reasoning applies for non-self-verifying data, where the consistency constraint requires that read and write quorums intersect in a majority of correct processes. SBQ2 then becomes:

SBQ2′. $|Q_r| + |Q_w| - n \geq 2f + 1$ (*Consistency*)

The corresponding bound for n is $n \geq 3f + 1$.

The above arguments capture the intuition behind a-masking and a-dissemination quorums. We now define them formally.

3.1.1 Asymmetric quorum systems

We say that a set V of servers is a *voucher set*, if, under all possible failure scenarios, it is guaranteed to contain at least one correct server, i.e. $\forall B \in \mathcal{B} : V \not\subseteq B$.

We define asymmetric quorum system for generic (non-self-verifying) data and self verifying data as follows.

Definition 1 *A quorum system is an* a-masking quorum system *if the sets of read and write quorums $\mathcal{Q}_r$ and $\mathcal{Q}_w$ have the following properties.*

AM-Consistency The intersection of any pair of read and write quorums always contains a voucher set consisting entirely of correct servers.
$\forall Q_r \in \mathcal{Q}_r \forall Q_w \in \mathcal{Q}_w \forall B_1, B_2 \in \mathcal{B} : Q_r \cap Q_w \setminus B_1 \not\subseteq B_2$

AM-Availability One read quorum is always available.
$\forall B \in \mathcal{B}\ \exists Q_r \in \mathcal{Q}_r : B \cap Q_r = \emptyset$

Definition 2 *A quorum system is an* a-dissemination quorum system *if the sets of read and write quorums $\mathcal{Q}_r$ and $\mathcal{Q}_w$ have the following properties.*

AD-Consistency The intersection of any pair of read of read and write quorums is a voucher set.
$\forall Q_r \in \mathcal{Q}_r \forall Q_w \in \mathcal{Q}_w \forall B \in \mathcal{B} : Q_r \cap Q_w \not\subseteq B$

AD-Availability One read quorum is always available.
$\forall B \in \mathcal{B}\ \exists Q_r \in \mathcal{Q}_r : B \cap Q_r = \emptyset$

Note that the consistency requirement is easier to discharge when the data is self-verifying. As a result, in the *f*-threshold case, a-masking quorums require $n \geq 3f+1$, $|Q_r| = \lceil \frac{n+f+1}{2} \rceil$, and $|Q_w| = \lceil \frac{n+f+1}{2} \rceil + f$, while a-dissemination quorum systems only need $n \geq 2f+1$, $|Q_r| = \lceil \frac{n+1}{2} \rceil$, and $|Q_w| = \lceil \frac{n+1}{2} \rceil + f$.

3.2. Comparison with existing protocols

The SBQ protocols for generic and self-verifying data are similar to the protocols introduced by Malkhi and Reiter for masking and dissemination quorum systems [19]. There are two differences between these protocols and SBQ protocols. First, in the Write(Δ) operation, in place of the SBQ protocol's step 5, which just sends data to a write quorum, earlier protocols for masking and dissemination quorum systems first send the data and then wait for acknowledgments from a quorum of servers. In essence, these protocols send writes to a quorum of *responsive* servers while SBQ sends writes to a quorum of servers that may or may not be responsive. Second, earlier protocols use same-sized quorums for both reads and writes, while the SBQ protocols allow asymmetric read and write quorums.

To illustrate these differences, consider the *f*-threshold case. In addition to the constraints SBQ1, SBQ2, and SBQ3 listed above, Malkhi and Reiter protocols (MR protocols for short) add two more constraints.

First, MR protocols require that writes wait for a write quorum of acknowledgments.

MR1. $|Q|_w \leq n - f$ (*Availability*)

Second, MR protocols use symmetric quorums.

MR2. $|Q|_r = |Q|_w = |Q|$ (*Symmetry*)

Note that because MR1 and SBQ1 impose symmetric constraints on read and write quorums the use of symmetric quorums is a natural design decision for MR protocols. Note also that either of MR1 and MR2, when combined with constraints SBQ1 to SBQ3, is sufficient in the *f*-threshold case to increase by f the number of servers required to tolerate f failures: generic data now requires $n \geq 4f+1$ servers, with minimum quorum size $|Q| = \frac{n+2f+1}{2}$ ($n \geq 3f+1$ and $|Q| = \frac{n+f+1}{2}$ for self-verifying data). The following table compares the minimum quorum sizes in the *f*-threshold case for the MR protocols and the SBQ protocols.

For generic data:

	MR	SBQ
Server count	$4f+1$	$3f+1$
Write quorum	$\lceil \frac{n+2f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil + f$
Read quorum	$\lceil \frac{n+2f+1}{2} \rceil$	$\lceil \frac{n+f+1}{2} \rceil$

For self-verifying data:

	MR	SBQ
Server count	$3f+1$	$2f+1$
Write quorum	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil + f$
Read quorum	$\lceil \frac{n+f+1}{2} \rceil$	$\lceil \frac{n+1}{2} \rceil$

Because SBQ quorums are formed under strictly weaker constraints than the dissemination and masking quorums used in the MR protocols, the SBQ quorums never need to be larger than the MR quorums. The formulas above confirm this observation.

Conversely, for a given number of servers, the SBQ protocols can tolerate more failures than the MR protocols. For example in the case of self-verifying data on 13 servers, MR can tolerate 4 failures and SBQ can tolerate 6. The quorum sizes are 9 for MR and 13/7 for SBQ (for the write/read quorum, respectively).

Finally, we note that SBQ protocols use the same reliable asynchronous messaging system model as MR protocols, and, as we show in the next section, they provide the same consistency guarantees: regular semantics in the case of self-verifying data and safe semantics otherwise.

Although SBQ protocols can reach the same level of fault-tolerance with fewer servers, they sacrifice something in order to get these improvements: a writer that uses SBQ can not determine when a write operation ends. A mitigating factor is that all write operations are guaranteed to end eventually.

Section 4 shows that as a result, our protocol cannot be adapted to unreliable networks. Instead, we adapt the original protocols of Malkhi and Reiter to this more general model.

Because of space constraints, we refer the reader to our extended technical report [22] for the correctness proof of the SBQ protocols.

4. Network models

Both the MR and the SBQ protocols assume a *reliable asynchronous network*, that is for any pair of correct machines A and B, if A sends a message, then B is guaranteed to eventually receive it. In some systems, reliable communication is provided by the underlying network subsystem. In other cases, however, the network provides weaker guarantees such as *unreliable asynchronous communication*, in which each message sent has a non-zero probability of arriving at its destination but there are no bounds on message delivery time.

In that case, communicating machines commonly attempt to construct a network layer that provides a reliable network abstraction over unreliable network hardware. Unfortunately, Byzantine machine failures can make this difficult because faulty servers can flush their network buffers or refuse to send acknowledgments. In particular, we are

concerned about bounding memory consumption of message buffers. Commonly, a system achieves reliable message delivery by requiring a sender to buffer and occasionally retransmit each message it sends until it receives an acknowledgment from the receiver [1, 12, 23]. In an asynchronous system, such an approach can consume unbounded buffer memory even if failures are restricted to crash failures [24]. This danger arises because a correct but slow machine cannot be distinguished from a faulty (crashed) machine. Therefore, a sender can never safely delete an unacknowledged message from its buffer and a faulty server can easily force clients to consume infinite memory by never acknowledging messages.

Table 1 summarizes the key results discussed in this section. Our analysis results in a series of Byzantine quorum systems and protocols over a range of system models, with increasing numbers of servers required to tolerate progressively weaker system models. For generic data, $2f + 1$ servers are needed for synchronous reliable network systems where timeouts are short, $2f + 1$ to $3f + 1$ for synchronous reliable network systems where timeouts are long, $3f + 1$ for asynchronous reliable network systems, and $4f + 1$ for asynchronous unreliable network systems. Self-verifying-data allows systems to be built for each of these scenarios using f fewer servers.

In Subsection 4.1 we explain the circumstances under which the reliable asynchronous network abstraction can be implemented on top of an unreliable network. If this construction is not possible then other network models must be considered, for example a synchronous model or one that makes no assumption of reliability. These models are discussed in Subsections 4.2 and 4.3 respectively.

4.1. Engineering an asynchronous reliable network

If the network layer is subject to arbitrary Byzantine failures then a faulty receiver can prevent a sender from ever deleting buffered messages. Nonetheless, one can engineer a reasonable approximation of an asynchronous reliable network abstraction when one can (1) restrict the failures to which the system or the network layer is vulnerable or (2) restrict the workload so that infinite buffering is not a concern. To illustrate when a reliable network layer can be built, we provide a few examples of both types of restriction below.

Restricting failures. For a fail-stop system model, this problem may not be a large concern because there exist reasonable engineering approaches to avoid the need for infinite memory while providing a reasonable approximation of reliable asynchronous messaging. For example, several reliable messaging systems [1, 15] store unacknowledged messages on in an on-disk log. It may be safe in practice to assume that it is extremely unlikely that the log will overflow by assuming (1) a large log, (2) a reasonable bound on crash or partition durations, and (3) that a machine will acknowledge received messages after the repair of a crash or partition. Although such an approach may be theoretically unsatisfying (it implicitly assumes a bound on the duration of failures and therefore is no longer, strictly speaking, an asynchronous system), this approach seems common in practice.

Restricting network failures. In some systems, the Byzantine quorum protocol layer is vulnerable to arbitrary Byzantine failures, but the network layer is less vulnerable. Examples include "System/Storage Area Networks" (SANs) (such as Fibre Channel [27]), networks for Massively Parallel Processors (MPPs) (such as the Thinking Machines CM5 and Cray T3D), networks with built-in redundancy and automatic fail-over such as Autonet [28], and networks with automatic link-level retransmission [25]. A second, related, approach to bounding memory consumption by assuming a restricted model of network failures is to construct a network protocol without relying on acknowledgments to free network retransmission buffers. For example, consider the case where the primary cause of message loss is bit errors from transient electronic interference, where each packet has a probability p of arriving at its destination. A sender that retransmits a message a constant number of times or with sufficient forward error control redundancy [6] may in this case regard the packet as successfully sent, even if no acknowledgments are received; such a system may still use acknowledgments to reduce the number of retransmissions in the common case of a responsive sender. A third approach that insulates the network layer from some failures is to rely on protection across software modules. For example, in some systems the network layer may be a protected kernel subsystem and may be considered less vulnerable to Byzantine failures than higher-level protocols.

Restricting the workload. Rather than restricting the network failure model, some systems may approximate reliable asynchronous messaging with finite buffers by assuming a restricted workload. If the request rate is low and the retransmission buffer large (e.g., on disk as in MQS [1] for example), then a system may reasonably buffer all sent messages regardless of whether they have been acknowledged. An example of a system where such an assumption is natural is a system that already maintains a persistent log of all transactions for another purpose such as auditing.

4.2. Synchronous network

Given the challenges to engineering a reliable asynchronous network, it may not be much more difficult to en-

Network Model	Protocol	servers for generic data	servers for self-verifying data
reliable synchronous (fast timeouts)	Bazzi [4]	2f+1	f+1
reliable synchronous (slow timeouts)	S-SBQ SBQ	2f+1 to 3f+1 3f+1	f+1 to 2f+1 2f+1
reliable asynchronous	SBQ	3f+1	2f+1
unreliable asynchronous	U-masking/U-dissemination	4f+1	3f+1

Table 1. *Summary of protocols tolerating f Byzantine failures for different network models.*

gineer a reliable *synchronous* network that allows network buffers to be bounded by placing an upper bound on delivery time. In effect, such a system declares that a server has failed if it fails to acknowledge a message within a prescribed time.

An obvious strategy to constructing Byzantine storage in a synchronous system is to use time-outs not only to garbage collect network buffers but also to detect server failures at the BQS-protocol level. This additional information can improve the efficiency of the BQS protocol. In particular, Bazzi [4] describes a synchronous BQS protocol for generic (or self-verifying) data that requires just $2f + 1$ (or $f + 1$) servers to provide storage with safe (or regular) semantics. Bazzi's read protocol for self-verifying data, for example, sends read requests to all $f + 1$ servers, waits for $f + 1$ replies or time-outs, and then returns the correct value with the highest timestamp from the set of replies.

The disadvantage of such an approach is that a single faulty server can force each read request to wait for a timeout. Unfortunately, for many systems the natural network timeout may be long or it may be difficult to estimate precisely. For example, empirical measurements of network failures show a heavy-tailed distribution for the duration of Internet connectivity failures, with significant numbers of failures lasting several minutes and some network failures lasting hours [10]. As another example, TCP's protocol for establishing an initial connection attempts retransmissions at increasing intervals that can exceed one minute if several packet losses occur in a row [3]. Therefore, it may often be desirable to conservatively set such timeouts to be as long as possible in order to avoid introducing spurious server failures. When messages can be buffered on disks, timeouts of minutes, hours, or longer may be desirable.

Unfortunately, if a synchronous BQS protocol is used, such timeouts could result in unacceptable read performance for many applications. In some cases, the impact of long timeouts can be mitigated by having clients track which servers have timed out in the past so that clients can avoid sending messages to or waiting for servers known to have failed. Unfortunately, this solution is not always appropriate. For example, for some applications or environments such an approach can (1) increase the complexity of a client, (2) increase the complexity of server recovery [8], (3) inflict a timeout that is too long (e.g., minutes or hours) to be accepted for even a single operation per client, or (4) remain vulnerable to a server that consistently responds a few moments before a series of timeouts.

An alternative approach is to use an asynchronous Byzantine quorum protocol over a synchronous network. In this approach, a server that fails to acknowledge a message within a timeout is defined to have failed, and the network layer uses timeouts to bound buffer consumption by deleting messages to failed servers. The Byzantine quorum protocol, however, is asynchronous and does not make use of timeouts. In effect, we are deploying an asychronous protocol on top of a synchronous network. Although the concept may be surprising, this approach has a number of advantages. First, this approach allows for a clean separation of concerns. Second, it is "self-timing": reads and writes proceed at the rate of the correct servers rather than the rate imposed by failed servers and timeouts. The price for this speed is that the SBQ protocol requires f more servers than Bazzi's synchronous protocol.

This naturally raises the question of how much performance can be achieved using fewer additional servers. In fact, a continuum exists between (a) the option of synchronous protocols such as Bazzi's that use $2f + 1$ servers for generic data but that can suffer slow reads if even one server is faulty and (b) the option of asynchronous protocols that use $3f + 1$ servers for generic data servers but that can keep all failed servers off the critical path of read and write operations. We cover this complete continuum by adapting the SBQ protocol to the reliable synchronous network model. The resulting protocol, S-SBQ, provides two different guarantees: it can still tolerate f failures, and in addition it is guaranteed to complete operations without waiting for time-outs as long as the number of actual failures stays below some threshold t ($t \leq f$). We say that S-SBQ is *f-safe, t-fast*. By comparison, the Bazzi protocol is f-safe, 0-fast and the asynchronous BQS protocols are f-safe, f-fast. The quorum construction used by S-SBQ allows it to be f-safe, t-fast using $f+t+1$ servers ($2f+t+1$ for non-self-verifying data). Because the choice of the value of t is left to the implementor, S-SBQ can either use as few servers as Bazzi's protocol or always be self-timing like SBQ. More interestingly, its performance can be adjusted to any intermediate

scenario.

Due to space constraints, we refer the reader to [22] for the complete description of the S-SBQ protocol. Note that even though the discussion of the previous paragraph was limited to the threshold case, S-SBQ uses a more general failure model that includes not only a fail-prone system but also a new *delay-prone system* to describe the conditions under which the protocol must be fast.

The following theorems describe the key behaviors of the S-SBQ protocol.

Theorem 1 *The S-SBQ protocol for self-verifying data follows regular semantics and the S-SBQ protocol for non-self-verifying data follows safe semantics.* (Safety)

This theorem expresses the safety of the protocol. Its proof derives from the intersection property of our quorum construction.

Theorem 2 *The S-SBQ protocols are live (i.e. all requests eventually terminate).* (Liveness)

It is easy to show by inspection that all protocol operations terminate at most after a time-out delay. The next theorem expresses the conditions under which the protocol does not need to wait for this delay.

Theorem 3 *The S-SBQ protocols are self-timed as long as the failure set is covered by some delay scenario.* (Performance)

This derives from the availability property of the quorums.

It is also straightforward to adapt Bazzi's protocol to construct an f-safe, t-fast version by adding more servers. However, because Bazzi's protocol includes synchronous acknowledgments of writes, the natural definition of such an "S-Bazzi" protocol retains symmetric read and write quorums and therefore requires $2f + 2t + 1$ servers for generic data ($f + 2t + 1$ servers for self-verifying data).

4.3. Unreliable asynchronous network

In this section we describe a U-masking and U-dissemination Byzantine quorum protocol for *authenticated unreliable networks* as defined in Section 2 in which the protocol deals with network-layer failures, retransmission, and buffering. We also show how variations of this protocol can bound network retransmission buffer consumption. This protocol is a straightforward extension of Malkhi and Reiter's protocol for asynchronous reliable networks [19]. Due to space constraints, we summarize the protocol and its properties in this section. We refer the reader to [22] for a full statement of the protocol as well as proofs for the theorems and lemmas stated in this section.

Although the model used by Malkhi and Reiter's original protocol ensures that all correct servers receive all transmitted messages, the protocol itself only relies on a quorum of servers receiving each message. Thus, once a sender receives responses to a request from a quorum of machines, it may safely stop retransmitting that request. Because the protocol requires explicit responses to all requests, including writes, it is simple to adapt it to manage retransmission buffers. In particular, we modify the protocol to replace each step that waits for a quorum of replies to instead repeatedly resend the message sent in the previous step to all servers that have not responded until a quorum of servers has responded. Note that a sender can space the repeated resends arbitrarily far apart in time as long as it follows an algorithm that ensures an infinite number of retries to a receiver if no response from that receiver is ever received and if the send to that receiver is not cancelled. Also note that these application-level retransmissions provide weaker guarantees than the reliable asynchronous networking abstraction because some correct servers may not receive messages transmitted to them.

The resulting U-masking (or U-dissemination) protocol provides safe (or regular) semantics for generic (or self-verifying) data. The protocol is live because the availability property guarantees that it must always eventually stop resending messages: under an unreliable asynchronous network as defined here, a message sent repeatedly must eventually reach its destination. Given that, we show that each send/receive/wait step is equivalent to a reliable asynchronous send to a responsive quorum of servers. Then, the proof of safety and liveness follows Malkhi and Reiter's original proof.

The advantage of managing message retransmission in the Byzantine quorum protocol as opposed to abstracting it into the communications layer is that doing so makes it easy to bound buffer consumption even if a server's network protocol software is considered vulnerable to Byzantine failures of the server. In particular, under these protocols, a read or write request may consume client buffer memory proportional to n, the number of servers. If a client issues c concurrent operations, then the client's total memory consumption is O(nc). Unfortunately, in an asynchronous system, each request may take arbitrary time to complete, so c may, in general, be unbounded. Fortunately, this protocol is amenable to several techniques for bounding the number of outstanding requests from each client. For example, if a client application using the BQS system is single-threaded and blocks for reads and writes, then system buffer consumption is naturally bounded to O(n) buffers per client.

A more general solution is for the protocol itself to manage allocation and deallocation from a finite set of buffer and to block incoming requests when insufficient buffers are available to complete a request. In particular, in the

Finite Buffer U-masking or U-dissemination protocol, we assume L local buffers and add a step FIRST before and a step LAST after both the read and the write function.

FIRST) Wait for n local buffers to be available then lock n local buffers.

LAST) Unlock the n local buffers claimed in step FIRST.

We provide the complete proofs for the following three theorems in our technical report [22].

Theorem 4 *The Finite Buffer U-masking protocol for generic data follows safe semantics and the Finite Buffer U-dissemination protocol for self-verifying data follows regular semantics.* (Safety)

The safety of the Finite Buffer protocol follows from the fact that each send/wait/repeat step is equivalent to a reliable asynchronous send and the safety properties of Malkhi and Reiter's original protocol.

Theorem 5 *The Finite Buffer U-masking and U-dissemination protocols are live (i.e. all requests eventually terminate).* (Liveness)

This follows from three facts: (1) step FIRST terminates because the rest of the protocol is live, (2) each network send/wait/resend step terminates because it must eventually reach a responsive set of servers, and (3) the original Malkhi and Reiter protocols terminate.

Theorem 6 *The Finite Buffer U-masking and U-dissemination protocol consumes at most L buffers.* (Finite Buffering)

This follows from the locking of step FIRST.

5. Related Work

There is a significant body of work on quorum systems [11, 13, 18, 29] but Byzantine failures were first considered by Malkhi and Reiter [19]. They have extended this work in other directions, for example by distinguishing between crash and Byzantine failures [21]. In the same work, Malkhi and Reiter show how to use smaller quorums (as opposed to smaller quorum *systems*, as examined here), of size $O(\sqrt{n})$. These constructions however require as many total servers as their previous work. Investigating whether our SBQ protocols can be adapted to these smaller quorums remains future work. Malkhi and Reiter also explore the load of the quorum system and present a quorum construction which does not require the clients to know about the failure scenarios [19]. Exploring these concepts in the context of SBQ is future work.

The idea of distinct read and write quorums has been explored before [11] but not in the context of Byzantine failures.

Bazzi [4] explored Byzantine quorums in a synchronous environment with reliable channels. In this context it is possible to require fewer servers ($f + 1$ for self-verifying data, $2f + 1$ otherwise). Our work shows an alternative asynchronous algorithm that can efficiently utilize additional servers to avoid slow reads.

Bazzi [5] argues that an important metric of a quorum system is the asynchronous access cost – the number of servers that are contacted during an operation. In a sense, SBQ has already optimized its use of messages by using asymmetric quorums. As a result, the asynchronous cost cannot be optimized further.

Triantafillou and Taylor [30] have extended work in quorums under a fail-stop assumption by reasoning about the location of the replicas. They present results which provide similar availability to quorum systems but with improved latency. Extending these results to Byzantine environments remains future work.

Phalanx [20] builds shared data abstractions and provides a locking service, both of which can tolerate Byzantine failure of servers or clients. It uses dissemination and masking quorums. Asymmetric quorums would not be appropriate in this case because to implement locks, one must be able to determine when the write operation completes.

Castro and Liskov [9] also attacked the problem of reliable storage under Byzantine failures. They implement a Byzantine-fault-tolerant NFS service using a technique different from quorum systems. They use self-verifying data only for the (relatively infrequent) view-change and new-view messages and can tolerate f Byzantine failures using $3f + 1$ servers.

When using non-self-verifying data, faulty servers can force new timestamps to take arbitrarily large values. This is a problem because in practice timestamps can only take values from a finite range and therefore faulty servers can compromise the safety of the protocol. All the quorum protocols discussed in this paper are vulnerable to this problem, but it can be solved by applying known techniques [17].

6. Conclusion

We present two Small Byzantine Quorum (SBQ) protocols for shared variables, one that provides safe semantics for generic data using $3f + 1$ servers and the other that provides regular semantics for self-verifying data using $2f + 1$ servers. This reduces by f the number of servers needed by previous protocols in the reliable asynchronous communication model. Our protocols use the novel a-masking and a-dissemination quorums. They differ from existing quorums for Byzantine systems in that they make a distinction between read and write quorums.

The reliable channels required by our protocols can be difficult to engineer, particularly when Byzantine failures are a concern. We therefore consider Byzantine quorum protocols with different system models.

In the case of reliable synchronous networks, protocols that rely on synchrony can be forced to wait for a time-out if faulty servers do not reply. It can therefore be advantageous to use asynchronous protocols and to use the synchrony assumption only in the network layer. We propose an intermediate protocol for the synchronous model which tolerates f Byzantine failures but also provides the guarantee of self-timed operation as long as the number of actual failures does not exceed a threshold t ($t \leq f$).

For the case of unreliable asynchronous networks we show how to adapt Malkhi and Reiter's protocol to this environment to provide safe semantics using $4f+1$ servers or, if the data is self-verifying, regular semantics using $3f+1$ servers.

A limitation of the asymmetric quorums used by the SBQ protocols is that the implicit synchronization provided by blocking writes is lost. We are exploring the benefits and limitations of solutions that combine SBQ protocols with explicit end-to-end acknowledgments of writes that have been successfully read.

References

[1] MQSeries, IBM, `http://www-4.ibm.com/software/ts/mqseries`.

[2] 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.

[3] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Usenix Symposium on Internet Technologies and Systems*, Oct. 1997.

[4] R. A. Bazzi. Synchronous Byzantine quorum systems. *Distributed Computing Journal Volume 13, Issue 1*, pages 45–52, January 2000.

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

[6] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

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

[8] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00), San Diego, USA*, pages 273–287, October 2000.

[9] M. Castro and N. 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.

[10] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN service availability. In *Third Usenix Symposium on Internet Technologies and Systems (USITS01)*, March 2001.

[11] 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.

[12] J. Gray and A. Reuter. Transaction processing: Concepts and techniques, 1993.

[13] M. Herlihy. A quorum-consensus replication method for abstract data types. In *ACM Transactions on Computer Systems (TOCS) Volume 4 , Issue 1*, pages 32–53, 1986.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '00), Cambridge, USA*, pages 93–104, October 2000.

[15] A. D. Joseph, F. A. deLespinasse, J. A. Tauber, D. K. Gifford, and F. M. Kaashoek. Rover: A Toolkit for Mobile Information Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 156–171, Copper Mountain, Co., 1995.

[16] L. Lamport. On interprocess communications. *Distributed Computing*, pages 77–101, 1986.

[17] M. Li, . Tromp, and P. M. B. Vitányi. How to share concurrent wait-free variables. *Journal of the ACM*, 43(4):723–746, 1996.

[18] N. A. Lynch and A. A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Symposium on Fault-Tolerant Computing*, pages 272–281, 1997.

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

[20] 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.

[21] D. Malkhi, M. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal on Computing 29(6)*, pages 1889–1906, 2000.

[22] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences, December 2001.

[23] J. Postel. Transmission control protocol. Technical Report RFC-793, Internet Engineering Task Force Network Working Group, Sept. 1981.

[24] A. Ricciardi. personal communication, Nov. 2001.

[25] J. Robinson. Reliable link layer protocols. Technical Report RFC-935, Internet Engineering Task Force Network Working Group, Jan. 1985.

[26] 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.

[27] M. Sachs and A. Varma. Fibre channel. *IEEE Communications*, pages 40–49, August 1996.

[28] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8), October 1991.

[29] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *Database Systems*, 4(2):180–209, 1979.

[30] P. Triantafillou and D. J. Taylor. The location-based paradigm for replication: Achieving efficiency and availability in distributed systems. In *IEEE Transactions on Software Engineering, 21/1*, pages 1–18, January 1995.