The Dissertation Committee for Jean-Philippe Etienne Martin
certifies that this is the approved version of the following dissertation:

Byzantine Fault-Tolerance and Beyond

Committee:

Lorenzo Alvisi, Supervisor

Michael Dahlin

Gregory Plaxton

Fred B. Schneider

Harrick Vin

Byzantine Fault-Tolerance and Beyond

by

Jean-Philippe Etienne Martin, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

Acknowledgments

I would like to thank my thesis advisor, Prof. Lorenzo Alvisi, for his constant mentoring and support. I am fortunate to have been able to work with him. I also want to thank Prof. Mike Dahlin for his feedback on the thesis and help with several papers and Prof. Peter Stone for insightful discussions and help with thesis writing. Thanks also to the other committee members, Prof. Greg Paxton, Prof. Fred Schneider, and Prof. Harrick Vin, for their careful reading of my draft. I also really appreciate the effort that the faculty at UT makes to be available to their students. These efforts are very much appreciated; in particular I would would like to thank Prof. Don Batory and Prof. J Moore.

I owe a great deal to all of the LASR group and the 3rd floor lunch group for their support and many conversations, insightful or relaxing as needed. I could not have asked for a better group of people to work with. In particular I would like to thank Allen, Amit, Arun, Ed, Jian, and Roberto—it was great working with you. Many thanks to Maria and Ted as well as Stefano and Chiara for their generous help in times of need. An especially heartfelt thanks to Eunjin for her support, encouragement and understanding through long years of graduate school.

Last but not least, I would like to thank my family for their support, love, and encouragement starting long before graduate school.

Jean-Philippe Etienne Martin

The University of Texas at Austin

December 2006

Byzantine Fault-Tolerance and Beyond

Publication No. ____________________

Jean-Philippe Etienne Martin, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Lorenzo Alvisi

Byzantine fault-tolerance techniques are useful because they tolerate arbitrary faults regardless of cause: bugs, hardware glitches, even hackers. These techniques have recently gained popularity after it was shown that they could be made practical.

Most of the dissertation builds on Byzantine fault-tolerance (BFT) and extends it with new results for Byzantine fault-tolerance for both quorum systems and state machine replication. Our contributions include proving new lower bounds, finding new protocols that meet these bounds, and providing new functionality at lower cost through a new architecture for state machine replication.

The second part of the dissertation goes beyond Byzantine fault-tolerance.

We show that BFT techniques are not sufficient for networks that span multiple administrative domains, propose the new BAR model to describe these environments, and show how to build BAR-Tolerant protocols through our example of a BAR-Tolerant terminating reliable broadcast protocol.

Contents

Chapter 1

Introduction

In an ideal world, computer systems would work as designed. We do not live in an ideal world. Computer systems fail because of software or hardware defects [79, 94, 148], either spontaneously or because of malicious attacks [65].

The reliability of a distributed system (defined as continuity of correct service) can be improved through replication. Starting from a set of assumptions covering the reliability of the components (nodes) that constitute the system, the reliability and timeliness of inter-node communication, and the behavior of faulty nodes, replication in space or time can be used to design distributed systems that provably continue to function correctly despite the failure of one or more of their components.

Such guarantees, however, have a flip side: if the system ever operates in an environment that violates any of the initial assumptions, then its behavior can become unpredictable. Malicious attackers, for instance, could bring a system down by forcing it to operate outside of the boundaries defined by the assumptions under which it was designed.

We believe that, if assumptions can be the cause of failures, then they should be treated as faults [14] and that consequently, in designing systems, the sound principle of fault prevention should be applied to remove all unnecessary assumptions.

In this dissertation, we investigate systems designed under very weak assumptions. We model communication between nodes as *asynchronous*: in the message-passing style of communication that we consider in the rest of this document, this means that nodes do not have access to a synchronized clock, there exist no upper bound on message delivery time, and there is no bound on the relative processing speed of nodes. We model the behavior of faulty nodes according to the *Byzantine* failure model [90, 124], in which faulty nodes behave arbitrarily.

The first six chapters of the dissertation explore what systems can be built, and at which cost, under this weak set of assumptions, focusing on two fundamental constructs used to build reliable distributed systems: the *register* [83]—the unit of reliable distributed storage—and the *replicated state machine* [82, 84, 139]—a general methodology for building fault-tolerant services.

Our findings improve the capabilities [107, 108, 109, 158] and reduce the cost [108, 110, 111, 158] of asynchronous Byzantine fault-tolerant techniques by revisiting Byzantine replication protocols from first principles. In particular, we have focused on reducing replication costs in terms of the number of nodes to tolerate f Byzantine nodes. This cost could otherwise become prohibitive when, to reduce the risk of correlated failures, the different nodes are built from different software stacks (as in n-version programming [13] or opportunistic n-version programming [130]).

Our contributions include proving new lower bounds, finding new protocols that meet these bounds, providing new functionality at lower cost through a new architecture for state machine replication, and, with the introduction of the Privacy

Firewall, resolving for the first time the tension between replication and confidentiality in state machine replication.

In the last chapter of this dissertation we consider instead an increasingly important class of systems: cooperative services. These systems have no central administrator and as a result, Byzantine fault-tolerance is no longer sufficient. In cooperative services, nodes can deviate from their assigned protocol not only because they are broken, but also because they (or, rather , their administrators) are selfishly intent on maximizing their own utility. It is always possible to model all such behaviors as Byzantine, but it is impossible to design reliable distributed systems under the assumption that *all* nodes may be Byzantine.

We address this challenge by introducing a new system model, called BAR, that treats selfish deviations separately from arbitrary ones. We provide a formal framework for reasoning about protocols in the BAR model by introducing the notion of Byzantine Nash Equilibrium that bridges Byzantine fault-tolerance and Game Theory [57]. To show that it is possible to design interesting protocols in the BAR model, we derive a new terminating reliable broadcast protocol and prove that it is a Byzantine Nash equilibrium. The new protocol, in addition the customary number of Byzantine nodes allowed in solutions based on traditional Byzantine fault-tolerance, tolerates also an arbitrary number of selfish nodes.

1.1 Overview of our Contributions

The simplest way to list our contributions is to group them by the primitive that is made Byzantine fault-tolerant.

BFT Quorums and Registers Registers are a unit of distributed storage. They offer two operations: read and write. *Quorums* [59] are a tool we use to build registers. Our contributions to BFT registers are:

- A safe [83] register with $3f+1$ nodes that does not require digital signatures. Previous signature-free protocols required $4f+1$ nodes instead, where f is the number of tolerated Byzantine failures.
- A proof that this protocol is optimal in the number of nodes.
- An atomic register with $3f+1$ nodes that does not require digital signatures. A safe register offers only weak guarantees if several nodes execute read or write at the same time—an atomic register is much stronger. Previous work offered no BFT signature-free atomic register.
- A protocol for dynamic quorums and registers. The resulting atomic register has the property that an administrator can change the set of nodes implementing the register while the system is running, without interrupting it. The system still only requires the minimal number of nodes: $3f+1$.
- A new trade-off: if it is not necessary to be able to determine when writes complete, we can build a *non-confirmable* register using only $2f+1$ nodes instead of $3f+1$.
- A new regular register for situations where some (but not all) nodes may be asynchronous. This protocol only needs $f+d+1$ nodes when d nodes may violate the synchrony assumptions.

The contribution that most surprised us relates to the distinction between, on the one hand, protocols that rely on the adversary not being able to forge digital signatures or read encrypted messages (more generally an adversary that is *computationally bound*) and, on the other hand, protocols that impose no such restriction on the adversary. Previous results [100] require more nodes to tolerate a given number of Byzantine failures when the adversary is not computationally bound. We show that when links are reliable (meaning no message is lost in transit), atomic registers can be implemented for an adversary that is not computationally bound without requiring any more nodes than for a computationally bound adversary.

BFT State Machine Replication Our contributions are:

- A new architecture for BFT state machine replication that reduces the cost from $3f + 1$ to $2f + 1$ replicas of the state machine.
- The Privacy Firewall, the first replicated state machine[1] with confidentiality guarantees. An adversary who takes control of any of the nodes in the traditional state machine replication approach may access information that should have been confidential. Our new Privacy Firewall ensures that even if the adversary can control some number of nodes, the system as a whole will prevent the adversary from extracting confidential information from them. This result is achieved by requiring all communication to go through the Privacy Firewall.

These two contributions come from the same principle: physically separating agreement from execution. Agreement is the procedure through which the nodes determine the order in which to execute operations. The agreement nodes require little storage and computational power compared to execution nodes, but most importantly the software they are running is simple. It is therefore easy to write

[1]Earlier systems could store encrypted data on behalf of clients [4, 80, 100, 105, 113, 134], our mechanism instead applies to any state machine.

several implementations of the agreement node, so they are more likely to fail independently. Since these nodes are cheap, we then investigate the benefits of using additional agreement nodes and find the following.

- A new consensus protocol that completes in two communication steps in the common case instead of three previously. We show that the minimal number of nodes required to complete in two steps despite t failures is $3f+2t+1$. Our consensus protocol matches this lower bound.

Cooperative Services

- A new model, BAR, that describes cooperative services. This model assumes that the nodes in the system may be Byzantine, Altruistic, or Rational. The Byzantine nodes may deviate arbitrarily from the protocol. *Altruistic nodes* follow the protocol unconditionally. *Rational nodes* seek to maximize their benefit, so they will only follow the protocol if no other course of action benefits them more. The rational nodes must be further described by indicating what they consider a benefit or a cost. Nodes do not initially know which other nodes are Byzantine, altruistic or rational.

- A new terminating reliable broadcast protocol, TRB+, that ensures that rational nodes maximize their utility by participating in the protocol truthfully. This is the first TRB protocol for the BAR model: it allows us to ensure that the altruistic and rational nodes will correctly participate in the agreement protocol. The TRB+ protocol demonstrates how to build BAR-Tolerant protocols.

The rest of this thesis is organized as follows: Chapter 2 presents quorums, registers, and their semantics. Chapter 3 and 4 show new results related to the minimal number of nodes required for various kind of registers. Chapter 5 presents registers with a dynamic membership. Chapter 6 presents work related to state machine replication, and chapter 7 explores the BAR model.

Chapter 2

Registers and Quorums

2.1 Introduction

In this chapter we introduce quorums and registers, as these concepts are used in the next few chapters. We define the safe, regular and atomic semantics and present masking and dissemination quorums. Readers already familiar with these notions may choose to skip this chapter.

2.2 Registers

The *register* [83] is an abstraction that provides two operations: **read**() and **write**(Δ).

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 there exists a total order $\rightarrow$ of all operations (*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.

Lamport defines three different kinds of registers [83]: safe, regular and atomic. His original definitions exclude concurrent writes, so we present extended definitions that allow for concurrent writes [145]. A safe register is a register that provides safe semantics (similarly for regular and atomic).

- *safe* semantics guarantees 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 provides safe semantics and guarantees 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 provides regular semantics and guarantees that the sequence of values read by any given client is consistent with the global serialization order ($\rightarrow$). The global serialization order provides a total order on all writes that is consistent with the *happens before* relation.

Read and writes are defined as starting when the **read**() (respectively **write**(Δ)) operation is called. 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 say that a protocol is *live* if all operations eventually complete. Quorums are a used to build registers: we introduce them in the following section, and then show two fault-tolerant registers that can be built using quorums. The node that sends requests to the register is called the *client*.

2.3 Quorums

A *quorum system* [59, 153] $\mathcal{Q}$ is a collection of subsets of servers, each pair of which intersect. Quorum systems provide two benefits: fault-tolerance and improved performance. If a protocol is designed so that it can make progress even when clients can communicate with only one quorum, then the quorum system will continue to function despite some number of failed nodes as long as a quorum is available. Quorum systems can also improve performance because the work from executing operations is spread across server nodes.

Quorum systems can be used for a variety of applications. We focus on using quorum systems for registers [59], but quorum systems have also been used to protected confidentiality of data [68], replicate objects [66], or control access [118], for example.

The failures that quorum systems can tolerate can be described in two different ways. The simplest is the *threshold model* that puts a limit f to the number of nodes that might fail (f is called the *resilience threshold*) . This pattern is common but it is not expressive enough when nodes do not fail uniformly. Some nodes could be more likely to fail, or some nodes may fail in a correlated manner because they have some parts in common. There is a model that can express these dependencies: the *fail-prone system model* [100]. We specify sets of servers, called *failure scenarios*. The set of failure scenarios is the *fail-prone system* $\mathcal{B}$. Fault-tolerant protocols designed for the fail-prone model give guarantees as long as at least one of the failure scenarios contains all of the faulty nodes.

Quorum systems are a powerful mechanism, and part of their power comes from the fact that their properties hold regardless of how the client determines which set of nodes it communicates with when forming a quorum. There is a trade-off between the number of messages sent and the time it may take to locate a responsive quorum: contacting all the nodes at once minimizes the time until a quorum responds, at the expense of possibly sending more messages than necessary. Sending only to one quorum and only contacting more nodes if there is no immediate answer reduces the number of extraneous messages but may slow down the application.

Protocols built on top of quorums use the **Q-RPC**(...) communication primitive [100]. **Q-RPC**(...) takes as input a message and the set of all nodes. It sends the message to at least a quorum of responsive nodes and returns their answers. This primitive maintains the ability for the administrator to trade between speed or number of messages.

2.3.1 Byzantine Fault-Tolerant Registers using Quorums

Malkhi and Reiter were the first to propose to use quorums to build Byzantine fault-tolerant registers. The protocol they use is shown in Figure 2.1. The server side is not shown, but it is very simple: servers store a value and a timestamp, and they are updated when the server receives a STORE with a higher timestamp. The pseudocode shows the client-side of write and read. The **write**(...) operation first queries a quorum of servers for their timestamp, and then stores the value with a new, higher, timestamp on a quorum. The **read**() operation queries a quorum of servers for their (timestamp, value) pair and selects one pair using the **select**(...) function. We specify the quorum construction and the **select**(...) function below.

Quorums must intersect to guarantee that if a client A writes some value to a quorum and then another client B reads from another quorum, B will see the value written by A. In order for this property to hold despite failures, their first construction requires the intersection to contain a *voucher set* of correct nodes (M-

```
write(Δ) :
1.  Timestamps := Q-RPC("GET-TS") // response from server $i$ is $ts_i$
2.  last_ts := max( Timestamps )
3.  choose a new timestamp new_ts that is larger than both
    last_ts and any timestamp previously chosen by this server.
4.  Q-RPC("STORE",new_ts,Δ)

$\langle ts, \Delta\rangle$=read() :
1.  Values := Q-RPC("GET") // response from server $i$ is $(ts_i, \Delta_i)$
2.  answer := select(Values)
3.  return answer
```

Figure 2.1: Write and read protocols for Malkhi and Reiter's constructions

Consistency). A voucher set is a set that is large enough to not be covered by any of the failure scenarios (in the threshold case, $f+1$ nodes). It is also necessary that the client always be able to find some quorum to communicate with (M-Availability). These two requirements are precisely captured in the definition below.

Definition 1. *A quorum system $\mathcal{Q}$ is a* masking quorum system *for a fail-prone system $\mathcal{B}$ if the following properties are satisfied.*

M-Consistency $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B_1, B_2 \in \mathcal{B} : ((Q_1 \cap Q_2) \setminus B_1) \not\subseteq B_2$

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

To select the correct value from a read operation, the reader discards values that are not contained in a voucher set and chooses among the remaining values the one with the highest timestamp. Since all voucher sets contain a correct node, the value it returns was indeed written by a client. M-Consistency guarantees that this procedure will select a value at least as recent as the last completed write. M-Availability guarantees that all reads terminate (although they may fail to read a value if a write is concurrent with the read; in this case they return the special value $\perp$). A safe register can be built using this version of **select**(...), with $4f + 1$ nodes [100].

If the model is changed so that digital signatures are available, and if clients are not Byzantine, then the same protocol can be used to build a regular register, using a *dissemination quorum system.* Here, quorums are only required to intersect in a single correct server. Clients must sign their value before passing them to the **write**(Δ) function. When reading, nodes then select, among the values returned by **Q-RPC**(...), the highest timestamped value that has a valid client signature. Since servers cannot fake digital signatures, this guarantees that the value was written by a client. This construction implements a regular register

Definition 2. *A quorum system $\mathcal{Q}$ is a* dissemination quorum system *for a fail-prone system $\mathcal{B}$ if the following properties are satisfied.*

D-Consistency $\forall Q_1, Q_2 \in \mathcal{Q}\ \forall B \in \mathcal{B} : (Q_1 \cap Q_2) \not\subseteq B$

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

Chapter 3

Minimal Cost Quorums and Registers

3.1 Introduction

Using replication to provide fault-tolerance can be costly. In this section we establish the minimal replication for Byzantine fault-tolerant registers and design a protocol that matches the bound. We then adapt this protocol to tolerate Byzantine clients.

A replicated system is hardly any more reliable than an unreplicated system if a single failure can cause the whole system to fail. There are several possible causes for a single point of failure. If the machines are all connected to the same power source, for example, then loss of power can bring down the whole service. Single points of failure can occur in software as well: if all the machines are running an operating system that has a bug causing it to crash when a certain malformed network packet arrives, then an adversary can bring down all the machines simultaneously. To avoid software being a single point of failure, the software on each node should ideally fail independently, meaning that the probability $p_i(x)$ of machine i failing for an input x is not correlated with the probability that another machine j fails on the same input. Formally, if p_i is the probability that machine i will fail when fed the next input, then if machines i and j fail independently then the probability that both will fail should be $p_i p_j$. Independence of failure is extremely difficult to achieve in practice: different operating systems sometimes have code in common, and even different teams implementing the same program tend to make the same sort of mistakes [77]. Luckily, independence of failures is not required: even with some correlation between the failures, a replicated system can be more reliable than a single implementation. Systems should be designed in such a way to minimize the correlation between failures, by using components that are as diverse as possible. This diversity is costly, so it is useful to design replicated systems that need as few machines as possible to tolerate a given number of failures (or, equivalently, design systems that can tolerate as many failures as possible for a given number of machines).

In this chapter we show two results that pertain to reducing the number of nodes in a Byzantine storage system: (i) we show the minimal number of nodes that are necessary for implementing safe, regular or atomic registers, and (ii) we show protocols matching this lower bound. The two key metrics that we consider are (i) the number of nodes necessary and (ii) whether digital signatures are available (we say that the data is self-verifying) or not (generic data). We describe the model in more detail in Section 3.2.1.

Tables 3.1 and 3.2 summarize our findings concerning the minimal number of nodes required to build an asynchronous Byzantine fault-tolerant register that can provide safe or stronger semantics. The bounds hold even for protocols that

Semantics	Generic data	Self-verifying data
safe	$4f+1$ [100]	$3f+1$ [100]
regular	-	$3f+1$ [100]
atomic	-	$3f+1$ [31]

Table 3.1: Best known asynchronous register protocols before our research.

Semantics	Generic data	Self-verifying data
safe	$3f+1$	$3f+1$
regular	$3f+1$	$3f+1$
atomic	$3f+1$	$3f+1$

Table 3.2: Tight lower bound on the number of nodes required for safe or stronger asynchronous registers.

assume clients are correct. Our results show that the lower bound is $3f+1$ even for the simplest case we consider (safe semantics with self-verifying data) and we show that the bound can be matched even for the most complex case we consider (atomic semantics with generic data).

Our new protocol that meets the lower bound is called Listeners.[1] The Listeners protocol[2] reduces the number of nodes and improves consistency semantics compared to previous protocols.

Like other quorum protocols, Listeners guarantees correctness by ensuring that reads and writes intersect in a sufficient number of nodes. Most existing quorum protocols access a subset of nodes on each operation for two reasons: to tolerate node faults and to reduce load. Listeners' fault-tolerance and load properties are similar to those of existing protocols. In particular, Listeners can tolerate f faults, including f non-responsive nodes. In its minimal-node configuration it sends read and write requests to $3f+1$ nodes, just like most existing protocols that contact $3f+1$ (out of $4f+1$) nodes.

[1]In [106], we introduced the Listeners protocol under the name Generalized SBQ-L)

[2]We sometimes use "Listeners" instead of "The Listeners protocol" for brevity. Similarly, we sometimes call other protocols simply by their name.

3.2 Preliminaries

3.2.1 Model

Following the literature [9, 22, 100, 101, 103], we assume a system consisting of an arbitrary number of clients and a set U of data nodes such that the number $n = |U|$ of nodes is fixed. We defined quorums systems in Section 2.3. Recall that a quorum system is a non-empty set of subsets of nodes, each of which is called a *quorum*. Nodes can be either *correct* or *faulty*. A correct node follows its specification; a faulty node can arbitrarily deviate from its specification (a *Byzantine failure*). We use a fail-prone system $\mathcal{B} \subseteq 2^U$, as described in Section 2.3.

The set of clients of the service is disjoint from U and clients communicate with nodes over point-to-point channels that are authenticated,[3] reliable, and asynchronous. In addition to asynchronous links, the system is asynchronous i.e. there is no bound on computation time and there is no synchronized clock. We discuss the implications of assuming reliable communication under a Byzantine failure model in detail in Section 3.6.6. Initially, we restrict our attention to node failures and assume that clients are correct. We relax this assumption in Section 3.5.

We use the definitions of Section 2.2 for safe, regular and atomic semantics.

3.3 Lower Bound for Registers

In this section, we prove lower bounds on the number of nodes required to implement safe registers (the weakest registers defined by Lamport). The bound is $3f + 1$ nodes and it applies to any fault-tolerant storage protocol, regardless of whether it uses non-determinism, cryptography, or non-quorum communication patterns.

[3]Note that authenticated channels can be implemented without using digital signatures, for example if every pair of nodes share a secret key.

This bound is tight because it is matched by our Listeners protocol, presented in Section 3.4. It is natural to wonder whether more nodes may be needed to provide stronger guarantees than safe semantics (such as regular or atomic). We show that this is not the case because our Listeners protocol provides atomic semantics, the strongest semantics defined by Lamport.

We prove that $3f + 1$ nodes are necessary to implement safe registers that tolerate f Byzantine failures. We show that if only $3f$ nodes are available, then any protocol must violate either safety or liveness. If a protocol always waits for $2f + 1$ or more nodes to answer for all read operations, it is not live because f crashed nodes will cause the reader to wait forever. But if a live protocol ever relies on $2f$ or fewer nodes 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 node.

Definition 3. *A message m is* influenced by *a node s iff the sending of m causally depends [82] on some message sent by s.*

We extend the definition of influence to operations: an operation o is influenced by a node s iff the end event of o causally depends on some message sent by s.

Definition 4. *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 1. *For all live write protocols using $3f$ nodes, for all sets S of $2f$ nodes, for all reachable quiet system states, there exists at least one execution in which a write is influenced only by nodes in a set S' such that $S' \subseteq S$.*

Proof. Suppose that the f nodes $\bar{S}$ outside of S crash. Since the protocol is live, it must not rely on a reply from $\bar{S}$, even indirectly, so by definition the write is not influenced by $\bar{S}$. □

Note that executions that is not influenced by $\bar{S}$ can also take place if the nodes in $\bar{S}$ do not crash since crashed nodes cannot be distinguished from slow nodes. Note also that Lemma 1 can easily be extended to the read protocol.

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

Thus, if there are $3f$ nodes, all read and write operations must at some point depend on $2f$ or fewer nodes 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 does not satisfy the safe semantics.

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

Proof. Informally, this read protocol sometimes decides on a value after consulting only with $2f$ nodes. 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 nodes (by Lemma 1). Without loss of generality, we number the influencing nodes 0 to $2f-1$. Immediately before the write starts in e_w, the nodes have states $a_0 \ldots a_{3f-1}$ ("state α") and immediately afterwards they have states $b_0 \ldots b_{2f-1}, a_{2f} \ldots a_{3f-1}$ ("state β"). 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 α then reads must return the value A; in particular this holds for the reads that influence fewer than $2f+1$ nodes. Lemma 2 guarantees such reads exist since the read protocol is live by assumption. Consider such a read whose execution we call e. Execution e receives messages that are influenced by nodes f to $3f-1$ and returns a value for the read based on messages that are influenced by these $2f$ or fewer nodes; in this case, it returns A.

Now consider what happens if execution e were to occur when the system is in state β. Suppose also that nodes 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. Note that servers $2f \ldots 3f+1$ remain in states $a_{2f} \ldots a_{3f+1}$. In this situation, states α and β are indistinguishable for execution e and therefore the read will return A even though the correct answer is B. □

Theorem 1. *No Byzantine-tolerant safe register implemented using $3f$ or fewer nodes is live.*

Proof. Lemmas 2 and 3 show that in the conditions given, no read protocol with $3f$ nodes can be live and safe. This includes protocols that do not communicate with all the nodes, so protocols with fewer than $3f$ nodes cannot be live and safe, either. □

3.4 Optimal Protocols and Listeners

3.4.1 Overview of Results

Section 3.3 proves that the minimal number of nodes for safe registers is $3f+1$. In the case of self-verifying data, this bound is achieved by Malkhi and Reiter [101] and by Castro and Liskov [31]. The latter protocol does not require reliable links and can tolerate faulty clients, but it makes some synchrony assumptions.

Our Listeners protocol [106] achieves this bound, even in the case of generic data and in an asynchronous environment. Figure 3.1 presents the client side of the Listeners protocol for generic data. Table 3.3 lists the variables' initial values. Our pseudocode sometimes uses the *current*[ ] array as a set; when accessed that way it naturally represents the set of values stored in the *current*[ ] array. We show the protocol for the fail-prone model. For the threshold model, $\mathcal{Q}$ is all subsets of q nodes, where $q = \lceil \frac{n+f+1}{2} \rceil$ and $\mathcal{A}$ is all subsets of $\lceil \frac{n+f+q}{2} \rceil$ nodes. To tolerate f Byzantine faults in this model, Listeners needs only $3f + 1$ nodes. Listeners does not tolerate Byzantine clients; we extend it in Section 3.5 to a version that does.

3.4.2 Gateway Quorum Systems

The key behind Listeners is a new quorum construction that takes into account not only which nodes answer the initial query, but also the set of nodes to which the query was sent.

Let $\mathcal{Q}$ be a dissemination quorum. Let $\mathcal{A}$ be a *gateway quorum system*, defined as a quorum system that has the property below.

G-Consistency $\forall A_1, A_2 \in \mathcal{A}\ \forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : (Q \subseteq A_1 \cap A_2) \wedge (Q \cap B = \emptyset)$

Informally, this property means that any two elements of $\mathcal{A}$ intersect in a correct quorum (e.g. a quorum consisting entirely of correct nodes) from $\mathcal{Q}$. Gateway quorums get their name from the fact that they describe the sets of nodes to which we communicate to reach an underlying quorum $Q \in \mathcal{Q}$. It follows from G-Consistency that every element of $\mathcal{A}$ contains a correct quorum from $\mathcal{Q}$, a property we call *G-Availability*.

```
W1   write($D$) :
W2       send (“QUERY_TS”) to all nodes in some $A \in \mathcal{A}$
W3       loop :
W4           receive answer (“TS”, $ts$) from node $s$
W5           $current[s] := ts$
W6       until $\exists Q \in \mathcal{Q} : Q \subseteq current[]$ // a quorum answered
W7       $max\_ts := max\{current[]\}$
W8       $my\_ts := min\{t \in C_{ts} : max\_ts < t \wedge last\_ts < t\}$
         // $my\_ts \in C_{ts}$ is larger than all answers and previous timestamp
W9       $last\_ts := my\_ts$
W10      send (“STORE”, $D, my\_ts$) to all nodes in some $A \in \mathcal{A}$
W11      loop :
W12          receive answer (“ACK”,$my\_ts$) from node $s \in A$
W13          $S := S \cup \{s\}$
W14      until $\exists Q_w \in \mathcal{Q} : Q_w \subseteq S$ // a quorum answered

R1   $(D, ts) =$ read() :
R2       send (“READ”) to all nodes in some $A \in \mathcal{A}$.
R3       loop :
R4           receive answer (“VALUE”,$D, ts$) from node $s$ // (possibly several answers per node)
R5           if $ts > largest[s].ts$ : $largest[s] := (ts, D)$
R6           if $s \notin S$ : // we call this event an “entrance”
R7               $S := S \cup \{s\}$
R8               $T :=$ the $f + 1$ largest timestamps in $largest[]$
R9               for all $isvr \in U$, for all $jtime \notin T$ : delete $answer[isvr, jtime]$
R10              for all $isvr \in U$ :
R11                  if $largest[isvr].ts \in T$ : $answer[isvr, largest[isvr].ts] := largest[isvr]$
R12          if $ts \in T$ : $answer[s, ts] := (ts, D)$
R13      until $\exists D', ts', Q_r : Q_r \in \mathcal{Q} \wedge (\forall i : i \in Q_r : \text{answer}[i, ts'] = (ts', D'))$
         // i.e., loop until a quorum of nodes agree on a (ts,D) value
R14      send (“READ_COMPLETE”) to all nodes in $A$
R15      return $(D', ts')$
```

Figure 3.1: Listeners, client protocol

G-Availability $\forall A \in \mathcal{A}\ \forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : (Q \subseteq A) \wedge (Q \cap B = \emptyset)$

3.4.3 The Protocol

In lines W1 through W8, the **write**(D) function queries a quorum of nodes in order to determine the new timestamp. The writer then sends its timestamped data to all nodes at line W10 and waits for acknowledgments at lines W11 to W14. The **read**() function queries a gateway quorum $A \in \mathcal{A}$ of nodes in line R2 and waits for messages in lines R3 to R13. An unusual feature of this protocol is that nodes send more than one reply if writes are in progress. For each read in progress, a reader maintains a

variable	initial value	notes
f	Size of the largest failure scenario	
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 node. 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 node and the associated data
answer[ ][ ]	$\emptyset$	Sparse matrix storing at most $f + 1$ data and timestamps received from each node
S	$\emptyset$	The set of nodes from which the client has received an answer

Table 3.3: Client variables

matrix of the different answers and timestamps from the nodes (*answers*[ ][ ]). The read decides on a value at line R13 once the reader can determine that a quorum $Q_r \in \mathcal{Q}$ of nodes vouch for the same data item and timestamp, and a notification is sent to the nodes at line R14 to indicate the completion of the read. A naïve implementation of this technique could result in the client requiring an unbounded amount of memory; instead, as we see in Theorem 3, the protocol only retains at most $f+2$ answers from each node, where f is the size of the largest failure scenario.

This protocol differs from previous Byzantine quorum system protocols because of the communication pattern it uses to ensure that a reader receives a sufficient number of sound and timely values. A *sound* value was written by a client. A *timely* value is recent enough to match the required semantics. A reader receives different values from different nodes for two reasons. First, a node may be faulty and supply incorrect or old values to a client. Second, correct nodes 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 and 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 nodes to disambiguate concurrency, Listeners uses extra rounds of messages when nodes and clients detect writes concurrent with reads. Intuitively, other protocols take a "snapshot" of the situation—the Listeners protocol looks at the evolution of the situation in time: it views a "movie", and so it has more information available to disambiguate concurrent writes.

As we mentioned before, Listeners uses more messages than some other protocols. Other than the single additional "READ_COMPLETE" message sent to each node at line R14, however, the additional messages are only sent when writes are concurrent with a read.

Figure 3.1 shows the protocol for clients. Server nodes follow simpler rules: they only store a single timestamped data version, replacing it whenever they receive a "STORE" message with a newer timestamp from a client (channels are authenticated, so nodes can determine the sender). When receiving a read request, they send their timestamp and data. Nodes in Listeners differ from previous protocols in what we call the Listeners communication pattern: after sending the first message, nodes keep a set of clients who have a read in progress. Later, if they receives 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 node receives a "READ_COMPLETE" message from the client indicating that the read has completed. For simplicity we create a conceptual initial write operation that puts the initial value on the nodes.

This protocol requires a minimum of $3f + 1$ nodes and provides atomic semantics with writes. We prove its correctness in the next section.

3.4.4 Correctness

Unlike previous quorum protocols, Listeners' read protocol does not just read a snapshot, but instead remains in communication with nodes until it has gathered enough information. That is why our quorum constraints contain not only a quorum system $\mathcal{Q}$ describing the nodes that answered the first read or write message, but also the set $\mathcal{A}$ describing the set of nodes to which information was sent. These nodes may receive this information later and forward it to the reader that is still listening.

Theorem 2. *The Listeners protocol provides atomic semantics.*

Lemma 4. *The Listeners protocol never violates regular semantics.*

Proof. Consider a read. Let Q_w be the quorum of nodes (not necessarily all correct) that have seen the latest completed write.[4] The read completes only after it gathers a quorum Q_r of identical responses (line R13). The D-Consistency property of $\mathcal{Q}$ ensures that the two quorums intersect in a correct node. Since correct nodes only replace their value with higher-timestamped ones, the read will return a value with a timestamp at least as large as that of the latest completed write. In other words, the read will be correctly ordered after the latest completed write. The value returned from the read was written by a client (it is sound) since correct nodes only accept sound data and the reader gets the same value from a quorum of nodes, at least one of which must be correct. $\square$

Having shown that Listeners satisfies regular semantics, we now prove atomicity of the Listeners protocol. The serialized order of the writes is that of the timestamps (this is a total order since no two clients' C_{ts} overlap). To prove this,

[4]Q_w exists since there is at least one completed write and each completed write affects a quorum.

we show that after a write for a given timestamp ts_1 completes (meaning, of course, that the **write**(...) function executes to completion), no read can decide on a value with an earlier timestamp.

Lemma 5 (Atomicity). *The Listeners protocol never violates atomic semantics.*

Proof. Suppose a write with timestamp ts_1 has completed: a quorum $Q_r \in \mathcal{Q}$ of nodes agree on this timestamp (line R13). Even if the faulty and untimely nodes send the same older reply ts_0, they cannot form a quorum (more formally: $(U-Q_r)\cup B \notin \mathcal{Q}$). We prove this by contradiction: the D-Consistency property must hold between all pairs of quorums, but if $O = (U - Q_r) \cup B$ were a quorum, then D-Consistency would not hold for O and Q_r. The D-Consistency property is shown below.

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

Suppose that O were a quorum. We compute the intersection of O and Q_r.

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

Since O is not a quorum, no correct client will accept the older reply ts_0. Similarly, suppose that at some global time t_1 some client c reads timestamp ts_1 and therefore (line R13) a quorum $Q_r \in \mathcal{Q}$ of nodes 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. Therefore, writes are ordered by their timestamp (which is consistent with real-time): Listeners never violates atomic semantics. □

Lemma 6 (Liveness). *Both* **read**() *and* **write**(...) *eventually terminate.*

Proof. **Write**. All calls to **write**(...) eventually return because the G-Availability property guarantees that a quorum of correct nodes will answer the "QUERY_TS" and "STORE" messages (lines W6 and W14).

Read. We say that an *entrance* happens every time the reader receives the first reply from a node (line R6). Note that there are at most n entrances. Nodes that answer are listed in the set S and the *largest*[ ] array contains the largest-timestamped answer from each node in S. The f earlier answers are stored in *answer*[ ].

Consider the last entrance. Let ts_{max} be the largest *largest*[ ]$.ts$ associated with a correct node. *largest*[ ] contains the largest timestamps received, so the client calling **read**() has not received nor discarded any data item with timestamp larger than ts_{max} from a correct node. $ts_{max} \in T$ because T contains the $f + 1$ largest timestamps in *largest*[ ] (line R8). Since all clients are correct, all correct nodes in some $A \in \mathcal{A}$ will eventually see the ts_{max} write and echo it back to the reader. None of these messages were discarded by the reader, and none will be (since T does not change after the last entrance). The G-Consistency property of A guarantees that there are enough correct nodes for the echoes to eventually form a quorum and the read will be able to complete (line R13).

STORE, QUERY_TS. The node's **store**(...) and **query_ts**(...) functions terminate because they have no loops and do not call any blocking functions.

READ. The node's **read**() function terminates because the client's **read**() terminates: since the client is correct, at this point it sends "READ_COMPLETE" to all nodes involved. Links are reliable, and when nodes receive this message their **read**() terminates. □

Theorem 3 (Finite memory)**.** *The reader protocol uses only a finite amount of space.*

Proof. All structures except **answer**[ ] have finite size. The **answer**[ ] array is indexed by node and timestamp. It only contains answers from nodes in S and timestamps in T (lines R9-R12). S has size at most n and T has size at most $f+1$, so **answer**[ ] has finite size: it contains at most $n(f+1)$ elements. Since readers may also keep a copy of the value from each node in the **latest**[ ] structure, it follows that each reader keeps at most $f+2$ answers per node. □

3.4.5 Listeners Protocol Summary

The Listeners protocol provides an asynchronous atomic register using the optimal number of nodes, without requiring digital signatures. It demonstrates that registers built without digital signatures do not necessarily need to use more nodes than those that require digital signatures.

3.5 Optimal Protocol for Byzantine Clients

The Listener protocol can tolerate Byzantine nodes but it is susceptible to Byzantine clients. In many environments, client machines are more vulnerable to failures because they typically run more software than nodes and are maintained by end users instead of professional staff. A protocol that does not take Byzantine clients into account does not bound the amount of damage that a single Byzantine client can inflict: the client might for example put the service into a "poisoned" state that prevents correct clients from accessing the service [110]. The Listeners protocol suffers from this problem: if a Byzantine client writes a different value to every node (a "poisonous write"), then read operations from correct clients will be unable to terminate because they cannot gather a quorum of identical answers.

We therefore now present the *Byzantine Listeners* protocol, a variant of the Listeners protocol that can tolerate Byzantine clients.

3.5.1 The Byzantine Listeners Protocol

The Byzantine Listeners protocol, just like Listeners, provides an asynchronous atomic register with only the minimal number of nodes ($3f + 1$) and does not require digital signatures. Byzantine Listeners is wait-free [67], meaning that clients are guaranteed an answer even if other clients are slow or crash. Byzantine Listeners can also handle Byzantine clients, in the following sense. The protocol associates operations with clients. It allows an administrator to remove authorizations from clients, for example after some client has been observed behaving improperly. The protocol guarantees that if a client c is removed, then eventually no operation associated with c will take place, even if other Byzantine clients and nodes remain.

The intuition behind the Byzantine Listeners protocol is that the value written comes with the authenticator (K, J) that proves that the value was suggested by a client. During writes, nodes will forward the value along with (K, J) to other nodes, so that (i) the write is guaranteed to complete eventually even if the client stops before finishing and (ii) the protocol's handling of (K, J) ensures that Byzantine nodes cannot fabricate writes unilaterally: all written values need to be generated by some client. For the authenticator (K, J), we avoid expensive digital signatures and instead use message authentication codes. But MACs are less powerful than signatures since a Byzantine node can craft what looks like a correct MAC to one node but looks incorrect to another node. The protocol uses additional forwarding to transcend this limitation: if two correct nodes communicate directly without going through a Byzantine node, then they will be able to recognize each

variable	contents
$MAXQ$	Size of the largest quorum (constant)
$mac_i(x)$	Shorthand for the vector: $hash(x, key_{i,j})$ for each node j
$mac_i^2(x)$	Shorthand for the tuple: $(mac_i(x), mac_i(mac_i(x)))$
m_i	$mac_i(ts_i, hash(D), last\_ts, c)$
M	A set: (ts_i, m_i) for each node i in some quorum
K	A set: $mac_i^2(ts, hash(D), c)$ for each node i in some quorum
J	A sparse array: if present, $J[i] = mac_i(ts, hash(D), c, K)$
$start\text{-}listening[c]$	The value of ts_0 when the node received a read request from client c.
T_c	The set of timestamps assigned to client c
$MAXM$	Maximum number of messages waiting to be forwarded (constant)
T_{wait}	An estimate for the message delivery time (constant)
$sendQ\{msg\}$	An associative array of at most $MAXM$ messages matched to tuples $(ttl, J, sent)$.

Table 3.4: Variables in the Byzantine listeners protocol

other's MAC as valid.

Figures 3.2, 3.3 and 3.4 show pseudocode for Byzantine Listeners. The **read**() code similar to that of the Listeners protocol: the only difference is that **reads**() return not only the data and timestamp but also the identity of the client who wrote the data. We include the reader code in Figure 3.2 for convenience. Our pseudocode uses the notation $sendQ\{msg\}$ for the associative array $sendQ$.

An earlier version of the Byzantine Listeners appears in [110]. This earlier version differs in requiring digital signatures to tolerate Byzantine clients.

3.5.2 Correctness

When tolerating Byzantine clients, we provide the *Byznearizable* [102] semantics defined by Malkhi, Reiter, and Lynch. To explain it, we first introduce a few terms that they use.

A *history* is a possibly infinite sequence of invocations and response events, each assigned to a single client (an invocation means that an operation (in our case, **read**() or **write**(D)) was called, a response means that the operation returned). A history is *sequential* if it is a sequence of alternating invocations and matching responses. A client subhistory $H|p$ is the subsequence of H that only includes

```
W1   **write**($D$) on client $c$ :
W2       send (“QUERY_TS”,$hash(D), last\_ts$) to all nodes in some $A \in \mathcal{A}$
W3       **loop** :
W4           receive $(TS, ts, m_s)$ from node $s$
W5           **if** $(m_s.c == c \wedge m_s.last\_ts == last\_ts)$ : $current[s] := (ts, m_s)$
W6       **until** $\exists Q \in \mathcal{Q} : Q \subseteq current[]$ // a quorum answered
W7       $M := \{m_s : s \in Q\}$
W8       $max\_ts := max\{current[s].ts : s \in Q\}$
W9       $my\_ts := min\{t \in T_c : max\_ts < t \wedge last\_ts < t\}$
         // $my\_ts \in C_{ts}$ is larger than all answers and previous timestamps
W10      $last\_ts := my\_ts$
W11      send (“PROPOSE_TS”, $my\_ts, hash(D), last\_ts, M$) to all nodes in $A$
W12      **loop** :
W13          receive (“TS_OK”,$my\_ts, k_s$) from node $s$
W14      **until** $\exists Q' \in \mathcal{Q} : Q \subseteq \{k_s\}$ // a quorum answered
W13      $K := \{k_s : s \in Q'\}$
W14      send (“STORE”, $my\_ts, D, c, K, \emptyset, MAXQ$) to all nodes in some $A \in \mathcal{A}$
W15      **loop** :
W16          receive answer (“ACK”,$my\_ts$) from node $s \in A$
W17          $S := S \cup \{s\}$
W18      **until** $\exists Q_w \in \mathcal{Q} : Q_w \subseteq S$ // a quorum answered

R1   (D,ts,c) = **read**() :
R2       send (“READ”) to all nodes in some $A \in \mathcal{A}$.
R3       **loop** :
R4           receive answer (“VALUE”,$D, ts$) from node $s$ // (possibly several answers per node)
R5           **if** $ts > largest[s].ts$ : $largest[s] := (ts, D, c)$
R6           **if** $s \notin S$ : // we call this event an “entrance”
R7               $S := S \cup \{s\}$
R8               $T$ := the $f+1$ largest timestamps in $largest[\,]$
R9               **for each** $isvr \in U$, **for each** $jtime \notin T$ : delete $answer[isvr, jtime]$
R10              **for each** $isvr \in U$ :
R11                  **if** $largest[isvr].ts \in T$ : $answer[isvr, largest[isvr].ts] := largest[isvr]$
R12          **if** $ts \in T$ : $answer[s, ts] := (ts, D, c)$
R13      **until** $\exists D', ts', c', Q_r : Q_r \in \mathcal{Q} \wedge (\forall i : i \in Q_r : \text{answer}[i, ts'] = (ts', D', c'))$
         // i.e., loop until a quorum of nodes agree on a (ts,D,c) value
R14      send (“READ_COMPLETE”) to all nodes in $A$
R15      **return** $(D', ts', c)$
```

Figure 3.2: Byzantine Listeners client protocol

```
S1   store($ts, D, c, K, J, ttl$) on node $s$, from process $p$ :
S2       if quorumVouches($ts, D, c, K, J$) : // write $D$
S3           send (“ACK”,$ts$) to $c$
S4           for each listening client $j$
S5               if ( $start\text{-}listening[j] \leq ts$ ) then send (“VALUE”, $ts, c, D$) to $j$
S6           if $((ts, c, D) > (ts_0, c_0, D_0))$ :
S7               $(ts_0, c_0, D_0) := (ts, c, D)$ // store the new value
S8               $ttl := MAXQ$
S9       if vouchable($ts, D, c, K, J$) $\wedge\, ttl > 0$ : // forward $D$
S10          $msg := (ts, D, c, K)$
S11          $sendNow := (sendQ\{msg\} = \emptyset)$
S12          $sendQ\{msg\}.ttl := max(ttl, sendQ\{msg\}.ttl)$
S13          $sendQ\{msg\}.J[s] := mac_s(ts, hash(D), c, K)$
S14          $sendQ\{msg\}.sent :=$ not $sendNow$
S15          for each node $i \neq s$ // merge $J$ with buffered message
S16              if $sendQ\{msg\}.J[i] = \emptyset$ : $sendQ\{msg\} := J[i]$
S17          if ($sendNow$) :
S18              send (“STORE”, $msg, sendQ\{msg\}.J, sendQ\{msg\}.ttl - 1$) to all nodes

D1   dequeue() on node $s$ :
         // called automatically whenever $sendQ$ is full or it has been non-empty for time $T_{wait}$
D2       for each $msg$ in $sendQ$ :
D3           if not $sendQ\{msg\}.sent$ :
D4               send (“STORE”, $msg, sendQ\{msg\}.J, sendQ\{msg\}.ttl - 1$) to all nodes
D5       $sendQ := \emptyset$

T1   query_ts($hD, last\_ts$) on node $s$ from client $c$ :
T2       if not $authorized(c)$ : return
T3       send (“TS”, $ts_0, mac_s^2(ts_0, hD, last\_ts, c)$) to $c$

P1   propose_ts($ts, hD, last\_ts, M$) on node $s$ from client $c$ :
P2       if not $authorized(c)$ : return
P3       if not $consistent(ts, last\_ts, c, M)$ : return
P4       let $M'$ be the set of elements $M.m_q \in M$ that satisfy :
             $M.m_q[s] = mac_q(M.ts_q, hD, last\_ts, c)[s]$
P5       if $\exists B \in \mathcal{B} : M' \subseteq B$ : return
P6       send (“TS_OK”, $ts, mac_s^2(ts, hD, c)$) to $c$
```

Figure 3.3: Byzantine Listeners server store protocol

```
Q1  quorumVouches($ts, D, c, K, J$) on node $s$ :
    // true if a quorum vouches that $(ts, D)$ comes from client $c$
Q2      if $\exists Q \in \mathcal{Q} : \forall q \in Q$, either
            (i) $K_q[s] == mac_q^2(ts, hash(D), c)[s]$
            or (ii) $J[q][s] == mac_q(ts, hash(D), c, K)[s]$
        then return true
Q3      return false

V1  vouchable($ts, D, c, K, J$) on node $s$ : // true if $D$ comes from a client
V2      if either
            (i) $K_s[s][1] == mac_s(ts, hash(D), c)[s]$
            or (ii) $J[s][s] == mac_s(ts, hash(D), c, K)[s]$
        then return true
V3      $V := \{v : J[v][s] = mac_v(ts, hash(D), c, K)[s]\}$ // set of valid MACs
V4      if $\forall B \in \mathcal{B} : V \not\subseteq B$ : return true
V5      return false

C1  consistent($ts, last\_ts, c, M$) :
C2      if $\nexists Q \in \mathcal{Q} : Q \subseteq M$ : return false
C3      $max\_ts := max\{m_i.ts : m_i \in M\}$
C4      $his\_ts := min\{t \in T_C : max\_ts < t \wedge last\_ts < t\}$
C5      return $(ts == his\_ts)$
```

Figure 3.4: Helper functions for Byzantine Listeners, server store protocol

invocations and responses for client p. A history is *well-formed* if for each client p, $H|p$ is sequential. We use $\mathcal{H}_C$ to denote the set of well-formed histories that can be induced when C is the set of correct nodes. N is the set of all nodes. A history H induces an irreflexive partial order $<_H$ on the operations in H as follows: $a <_H b \iff$ the response to operation a precedes the invocation of operation b in H (i.e. a happens before b). We say that history H *legal* if it obeys the specification of an atomic register.

Definition 5. *Let C be the set of correct nodes. A history $H \in \mathcal{H}_C$ is* Byznearizable *if there exists some legal sequential history $H' \in \mathcal{H}_N$ such that*

1. *$H|p = H'|p$ for all $p \in C$,*

2. *$<_H \subseteq <_{H'}$, and*

3. *if every $p \notin C$ eventually stops, then for each $p \notin C$, $H'|p$ is finite.*

Let $X.ts$ denote the timestamp associated with operation X: either the timestamp returned by X (if it is a **read**()), or the timestamp written by X (if it is a **write**)...). We define $X.D$ and $X.c$ likewise.

We now prove that the Byzantine Listeners protocol provides Byznearizable atomic semantics. To show this we construct the linearized order $\rightarrow$ of the operations and show that this order is consistent with real time (Theorem 4). We construct this order as follows: each operation $op \in \{\textbf{write}(D), \textbf{read}()\}$ from client c corresponds to some timestamp ts and some data D. In the case of **read**() they are the return value. In the case of **write**(D), ts is the timestamp my_ts used when writing value D.

Let $ord(X) = (X.ts, X.D, X.c, X.op)$ for any operation X. We say that for operations A and B on correct nodes, $A \rightarrow B \iff ord(A) < ord(B)$. We use lexicographical ordering in the comparison, meaning that two operations with the same timestamp are ordered by the data; if the data are also the same then reads are ordered before writes, and, failing that, operations are sorted based on the identity of the client. Since correct clients never reuse timestamps, $\rightarrow$ defines a total order on the operations from correct clients (this is order $<_H$ from the Byznearizability definition).

After showing that $\rightarrow$ is consistent with real-time, we show that if there are no Byzantine clients then Byzantine listeners implements an atomic register (Lemma 13) and that its write and read operations are live (Lemma 15 and Lemma 17), even in the presence of Byzantine clients.

Finally, we show that Byzantine Listeners has what we call *cleanup properties*: all values are tagged with the identity of the client who wrote the value (Lemma 18) and after a Byzantine client c is removed from the system, eventually no value tagged with c is written (Lemma 20). So just as Byznearizable semantics requires, if all the Byzantine clients are removed, then eventually only operations from correct clients are executed. Naturally, values tagged with c that were written in the past might remain if they are not overwritten, and values written by correct nodes might be influenced by incorrect values from c. The cleanup properties only guarantee that the removed nodes are eventually prevented from executing operations on the register.

Correct behavior for operations from correct clients

The proof that ordering $\rightarrow$ is consistent with real-time for operations from correct clients involves four Lemmas, which show that ordering is maintained across all combinations of reads and writes. Recall that each node p stores what it believes is the most recent (timestamp,data) pair in variables $p.ts_0$ and $p.D_0$.

Definition 6. *We say that a timestamp-value pair (ts, D) is* stable *once there is quorum $Q \in \mathcal{Q}$ of nodes such that each correct node p in Q has $(p.ts_0, p.D_0) \geq (ts, D)$.*

Lemma 7. *Once a pair is stable, it remains stable forever.*

Proof. This follows directly from the fact that correct nodes only increase their timestamp-value pair, never decrease it (lines S6-S7). □

Lemma 8. *If write W from a correct client ends with writing (ts, D), then (ts, D) is stable.*

Proof. The protocol allows writes to end only after the client receives an acknowledgment from some quorum $Q_w \in \mathcal{Q}$ (line W18). Since W ended, all the correct nodes in Q_w have send the acknowledgement. These correct nodes store (ts, D) or a newer value before answering any other request (lines S6-S7), so (ts, D) is stable by definition. □

Lemma 9. *If a read R from a correct client ends, reading (ts, D), then (ts, D) is stable.*

Proof. The read operation returns a value (D, ts, c) after it receives matching answers from a quorum $Q_r \in \mathcal{Q}$ (line R13). □

Lemma 10. *If pair (ts, D) is stable before a write W from a correct client starts, then $(W.ts, W.D) > (ts, D)$.*

Proof. The writer picks a timestamp that is larger than all timestamps from the quorum Q of answers it got for its "QUERY_TS" message. Since (ts, D) is stable, all correct nodes in some quorum Q' have values of at least (ts, D). By D-consistency, quorums Q and Q' intersect in at least one correct node. Therefore, $W.ts > ts$. □

Lemma 11. *If pair (ts, D) is stable before a read R from a correct client starts, then $(R.ts, R.D) \geq (ts, D)$.*

Proof. Since (ts, D) is stable, all correct nodes in some quorum Q' have values of at least (ts, D). The read operation returns a value (D, ts, c) after it receives matching answers from a quorum $Q_r \in \mathcal{Q}$. By D-Consistency, quorums Q' and Q_r intersect in a correct node. Correct nodes' values only increase, so $(R.ts, R.D) \geq (ts, D)$. □

Combining the previous lemmas, we see that ordering $\rightarrow$ is consistent with real-time for operations from correct clients.

Theorem 4. *Ordering $\rightarrow$ is consistent with real-time for operations from correct clients.*

Proof. Consider two operations by correct nodes O and O' such that O happens before O'. After O returns, $(O.ts, O.D)$ is stable (Lemma 8 if O is a write, Lemma 9 if it is a read). Since $(O.ts, O.D)$ is stable, $(O.ts, O.D) \leq (O'.ts, O'.d)$ (Lemma 10 if O' is a write, Lemma 11 otherwise). Since in addition, correct clients do not reuse timestamps, $ord(O) < ord(O')$. Therefore, $O \rightarrow O'$. $\square$

The next step is to show that if there are no Byzantine clients, then the values returned by the operations correspond to atomic semantics.

Lemma 12. *If correct client's call to* **read**() *returns* (D, ts, c) *and* c *is a correct client, then* c *has sent a* "PROPOSE_TS" *message with timestamp* ts *and* $hash(D)$.

Proof. We show that a call to **read**() can only return (D, ts, c) if client c sent a message "PROPOSE_TS" with ts and $hash(D)$. In other words, faulty servers cannot change the value of the register, even if they collude and share their secret keys.

In order for a read to return (D, ts, c), a quorum of nodes must have stored (D, ts, c) in line S7 (as that is the only instruction that changes what nodes store). All quorums contain at least a correct node (D-Consistency). Correct nodes only store a value if **quorumVouches**$(\ldots)$ returns *true* (line S2), which in turn implies that a quorum of nodes (including at least one correct node) put a MAC indicating that it vouches that the value, timestamp pair comes from client c. Client c therefore participated in the write by sending a "PROPOSE_TS" message with both $hash(D)$ and ts. $\square$

Lemma 13. *If there are no Byzantine clients then Byzantine Listeners implements an atomic register, assuming it is live.*

Proof. This lemma follows from the previous lemmas: timestamps are consistent with real-time (Theorem 4), and if several **write**(...) completed before a call to **read**(), then the returned timestamp will be at least as large as the latest completed **write**(...)'s timestamp (since timestamps from completed writes are stable, by Lemma 8, and **read**() cannot return a smaller timestamp than any stable timestamp, by Lemma 11).

Atomic semantics indicates that if a **read**() R returns data that was written by **write**(...) W, then there should not exist any **write**(D') W' s.t. $W \rightarrow W' \rightarrow R$ and $D \neq D'$. This property follows directly from our construction of *ord*: if R returned the data written by W then $ord(R) = (W.ts, W.D, \mathbf{write}(), W.c)$. If $ord(W) < ord(W') < ord(R)$ then W' must have written data $W.D$ with timestamp $W.ts$.

Finally, Lemma 12 shows that faulty nodes cannot initiate **write**(...) operations: a client must be involved. If there is no Byzantine client, then every value D returned by **read**() comes from a preceding **write**(D). □

Next we show that all operations eventually complete, regardless of whether Byzantine clients are present.

Lemma 14. *All dissemination quorums have the double cover property.*

Double cover $\forall Q \in \mathcal{Q}\ \forall B_1, B_2 \in \mathcal{B} : (Q - B_1) \not\subseteq B_2$

Proof. Consider a dissemination quorum $\mathcal{Q}$ with a matching fail-prone system $\mathcal{B}$. Fix $Q \in \mathcal{Q}, B_1, B_2 \in \mathcal{B}$. By D-Availability, $\exists Q_1 : Q_1 \cap B_1 = \emptyset$. By D-Consistency, $Q \cap Q_1 \not\subseteq B_2$. In other words, $\exists x \in Q \cap Q_1 : x \notin B_2$.

Since Q_1 and B_1 are disjoint, $Q \cap Q_1 \subseteq Q - B_1$. So $x \in Q - B_1$ but $x \notin B_2$. In other words, $Q - B_1 \nsubseteq B_2$. □

Lemma 15. *The Byzantine Listeners write protocol eventually terminates, even in the presence of Byzantine clients.*

Proof. We consider only operations from correct clients. The first loop in **write**(D) on correct clients completes by the G-Availability property: it sends a message to a set $A \in \mathcal{A}$ of nodes, so there will be a correct quorum of nodes $Q_0 \subset A$ that can answer. The second loop in **write**(D) completes for the same reason: it sends a message to a set $A \in \mathcal{A}$ and waits for a quorum of answers. All correct nodes will answer because the test for *consistent* and M will pass as the nodes reproduce the client's computation. Let Q' be the quorum of nodes whose answers are gathered by the writer to create K.

The third loop sends the "STORE" message to all nodes in A (line W14). The correct nodes in A will answer as soon as **quorumVouches**$(\ldots)$ returns *true* (line S3). This may hold initially, in which case we are done because A contains a quorum of correct nodes. It may also be the case that **quorumVouches**$(\ldots)$ does not return *true* right away. However, the function **vouchable**$(\ldots)$ will return true for every correct node in Q' since it will recognize its message authentication code in K (line V2). Consider one such correct node, a_0. This node adds its message authentication code to J and then forwards the message to all nodes (line S18 or D4). Links are reliable so all nodes will receive the message, including correct node $a_1 \in Q'$ $(a_1 \neq a_0)$. The **vouchable**$(\ldots)$ function will return *true* for node a_1 because condition (i) of line V2 is *true*: the message was not modified in transit so a_1 will recognize its own message authentication code in K. Node a_1 will then add its MAC to J and forward it to all nodes, including correct node $a_2 \in Q'$ $(a_2 \neq a_1, a_0)$. This process continues until a_q, the last correct node in Q'. All these messages have a positive value for *ttl* since $MAXQ \geq |Q'|$. Node a_q also adds its MAC to J. At this point, all the correct nodes in the quorum Q' have added their MAC to J. Node a_q then forwards the message to all nodes, including correct node $a_r \in Q_0$. By the double cover property (Lemma 14), **vouchable**(...) is *true* for a_r because the set of correct nodes in Q' is not a subset of any failure scenario, so line V4 returns *true*. The message is forwarded again, until it has been received by a quorum of correct nodes (e.g. Q_0). At this point, the message is forwarded again to all nodes, in particular to the nodes in Q_0. For these nodes, **quorumVouches**(...) will return *true* because test (ii) in line Q2 passes. They will therefore send an ACK to the client (line S3). The client is therefore guaranteed to leave the loop of lines W15-W18. □

Before we can show that all read operations complete, we show that if a correct node stores some value, then a quorum of correct nodes will store it as well.

Lemma 16. *If a correct node s executes line S7 for (ts, c, D), then eventually all correct nodes execute line S7 for (ts, c, D).*

Proof. If a correct node s executes line S7, storing (ts, c, D), then **quorumVouches**(...) returned *true* on line S2. That means that there is a quorum Q of nodes that vouch for the fact that data (ts, D) comes from client c, either because their MAC is in K, or because their MAC is in J. If **quorumVouches**(...) returns *true*, then **vouchable**(...) returns *true* as well (since the condition in V2 is implied by the one in Q2). Since the *ttl* is positive, node s will forward the "STORE" message to all nodes (line S18 or D4).

Consider a correct node $a_1 \in Q$ that receives the "STORE" message from s. Function **vouchable**(...) will return *true* for a_1 because it will recognize its own message authentication code in K or J (line V2). Node a_1 will add its signature to

J and forward the message to all nodes, including correct node $a_2 \in Q$. This process repeats until all correct nodes in Q have added their MAC to J. The message is not dropped since $ttl = MAXQ \geq |Q|$. Afterwards, J contains enough MACs to not be covered by any failure scenario (by the double cover property), so the test in line V4 will return *true*. So every correct node that receives this message (unmodified since we are considering only the forwarding between correct nodes) will add its MAC to J. The process continues until J contains MACs from a quorum of correct nodes so that **quorumVouches**$(\ldots)$ holds. That message is forwarded to all, and all correct nodes then execute line S7 for (ts, c, D). $\square$

Lemma 17. *The Byzantine Listeners protocol satisfies liveness, even in the presence of Byzantine clients.*

Proof. Lemma 15 shows that writes satisfy liveness, so only reads remain. There is a single loop in **read**(), where the reader waits for a quorum of answers with the same timestamp and data. We say that an *entrance* happens every time the reader receives the first reply from a node (line R6). Note that there are at most n entrances because there are only n nodes. Nodes that answer are listed in the set S and the *largest*[ ] array contains the largest-timestamped answer from each node in S. The f earlier answers are stored in *answer*[ ]. Consider the last entrance. Let ts_{max} be the largest *largest*[ ]$.ts$ associated with a correct node s. *largest*[ ] contains the largest timestamps received, so the client has not received nor discarded any data item with timestamp larger than ts_{max} from a correct node. $ts_{max} \in T$ because T contains the $f + 1$ largest timestamps in *largest*[ ] (line R8).

The reader waits until it receives a quorum of answers matching ts_{max}. By Lemma 16 we know that, since a correct node wrote ts_{max}, all correct nodes eventually will. When they do, the correct nodes in A will forward their value to the reader (lines S4-S5). By G-Availability we know that the correct nodes in A form at least one quorum, so the read will complete. □

We have shown that operations from correct clients are safe and live despite Byzantine nodes, and that Byzantine clients cannot compromise liveness. Byznearizable semantics allows ghost operations to be inserted. A *ghost operation* is one whose effects can be observed (so it appears in the Byznearized order) even though the operation was not explicitly requested by a client. In the case of a register, all ghost operations are writes (since unrequested reads cannot be observed). In our protocol, operations are associated with a client. We show that Byzantine clients cannot trigger ghost operations from correct clients, and we show that if Byzantine clients are stopped then eventually no ghost operation is invoked at all.

Correct behavior for operations from Byzantine clients

Definition 7. *We say that a write W is* associated with client c *if a read R such that W is the latest completed write relative to R would return (D, ts, c).*

Lemma 18. *If there is a write with timestamp ts associated with client c, then client c has sent a* "PROPOSE_TS" *message with timestamp ts. Similarly, if there is a write with data D associated with client c then client c has sent a* "PROPOSE_TS" *message with $hash(D)$.*

Proof. Suppose that some write W with timestamp ts and data D is associated with client c. By definition, there is some operation R that would return (D, ts, c). The conclusion follows by Lemma 12. □

Lemma 19. *Byzantine clients cannot trigger ghost writes from correct clients.*

Proof. By definition, a write for D from a correct client c is a ghost write if client c did not call **write**(D). Correct clients do not send a "PROPOSE_TS" message with $hash(D)$ if **write**(D) is not called, so by Lemma 18 there will be no write for D associated with the correct client. $\square$

Nodes maintain a set of authorized clients. The intent is that this set will be modified by an administrator: removing an element from the set will prevent that client from interacting with the system. The following lemma shows how this mechanism helps the administrator fight Byzantine nodes after they have been identified (the mechanism for identifying Byzantine behavior is out of the scope of this protocol).

Lemma 20. *If client c is removed from the authorized set at some point then there is a point in time from which no write associated with c occurs.*

Proof. If c is not authorized then correct nodes will not answer its "PROPOSE_TS" messages (line T2), so c cannot initiate new **write**$(\ldots)$ operations. If c is removed from the authorized set, then (since in a finite amount of time clients can only send a finite number of messages) c has sent a finite number of "PROPOSE_TS" messages before being removed from the set, so it can only initiate a finite number of writes. Each of these writes eventually complete, and then no other write associated with c can occur. $\square$

Theorem 5. *The Byzantine Listeners protocol provides Byznearizable atomic semantics.*

Proof. The relation $\rightarrow$ totally orders the writes and it is consistent with real-time for operations from correct clients (Theorem 4). Byzantine Listeners, if there is no Byzantine client, satisfies the safety properties of an atomic register (Lemma 13).

This register behaves as if all writes happened in the order described by $\rightarrow$, so if **read**() R returns value D then that value was written by the latest **write**(...) W such that $W \rightarrow R$. Byzantine Listeners also satisfies the liveness properties (Lemma 15). Since the register is atomic without Byzantine clients and since the clean-up property holds (Lemmas 18–20), Byzantine Listeners provides Byznearizable atomic semantics. $\square$

Common-case performance

Byzantine Listener's write operation is optimized so that in the common case, writes complete in $3\frac{1}{2}$ round-trips despite Byzantine clients and nodes.

Lemma 21. *If the* sendQ *buffer never fills up completely and all messages are delivered within time T_{wait}, then all writes from correct clients complete in $3\frac{1}{2}$ round-trips despite Byzantine clients and nodes.*

Proof. The first two messages, "QUERY_TS" and "PROPOSE_TS", are answered immediately. The third message, "STORE", is sent to all nodes in some $A \in \mathcal{A}$, which includes (by G-Availability) a quorum Q of correct nodes. These nodes recognize their own message authentication code in K, add a MAC to J, and forward the message to all nodes (including nodes in Q). Nodes in Q receive this forwarded message. Since *sendQ* contains an earlier version of the message, they do not send it immediately but instead wait as they receive more messages. Since messages are delivered within time T_{wait}, nodes will receive the messages from all other nodes in Q. At this point they forward the message to all nodes (including themselves). **quorumVouches**(...) returns *true* as J contains a quorum of valid MACs, so the nodes store the data and send an acknowledgment to the client. The client therefore receives the necessary quorum of answers to “STORE” after one round-trip and one forwarding of the message between nodes. □

3.6 Practical Considerations

We now turn to concerns that may arise when using Listeners or Byzantine Listeners in practice.

3.6.1 Resource Exhaustion

Byzantine clients can call **write**(...) or send messages to the nodes, reducing the capacity that is available for correct clients. This limitation is fundamental, because we cannot always distinguish Byzantine clients from correct ones. The Byzantine Listeners protocol handles this problem to a point because if a client is identified as Byzantine, it can be removed from the system. Also, the *ttl* field on all “STORE” messages is decremented every time the message is forwarded, ensuring that these messages are forwarded a finite number of times. However, the protocol can suffer greatly before the client is identified as Byzantine, because a faulty reader might not notify nodes that the read has completed, thereby forcing these nodes to continue that read operation forever. The root of the problem is that readers can cause potentially unbounded work at the nodes (the processing of a nonterminating **read**() request) at the cost of only constant work (a single faulty **read**() request). This imbalance makes this form of Byzantine behavior particularly damaging. Two things can be done to address resource exhaustion: (i) limit the amount of time nodes allocate to a given request or (ii) limit the number of messages that nodes send in response to a given request.

The first approach is to limit the amount of time nodes allocate to a given request, based on a simple time-out. Because our model is purely asynchronous, this solution in theory risks violating liveness. In practice, this liveness issue is tolerable because links are timely most of the time, so it is reasonable for the nodes to terminate **read**() unilaterally after some time-out. In the new protocol, clients also use a time-out, and if the **read**() has not completed when the time-out fires, the reader aborts the **read**() and starts anew. Because of the asynchronous nature of the network, it is possible that correct readers, too, are cut off. This does not affect the safety of the protocol, but the liveness is now only guaranteed if there is a an interval during which messages are timely enough for the read to complete. The designer's choice of when nodes should interrupt reads influences how long that interval has to last. A longer time-out means that aborted reads are less likely, and a shorter time-out reduces the load that Byzantine clients might impose on the nodes.

Instead of a time-out on nodes, it is possible for nodes to stop a **read**() after having forwarded some constant number of messages. This measure is similar to the time-out in that it reduces the impact of Byzantine clients but at the risk of aborting legitimate reads in periods where many writes are concurrent. A potential benefit of this variant is that the maximal amount of work incurred answering read requests is known ahead of time.

When several instances of the protocol are used in parallel to provide several registers, it is useful to restrict each reader to a constant number of parallel reads (this can be checked unilaterally by the nodes). This solution reduces the damage a Byzantine client can inflict but time-outs (or a bound in number of answers) should also be in place; otherwise, a faulty reader can still cause unnecessary traffic by sending a **read**() request for some variable that is often written.

3.6.2 Additional Messages

The write protocol always requires the same number of messages, regardless of the level of concurrency. Listeners' write operation requires $4q$ messages, where q is the number of nodes in a gateway quorum $A \in \mathcal{A}$. This is the same number of messages as used by Malkhi and Reiter's quorum protocols [100].

The behavior of Listeners' read operation depends on the number of concurrent writes. The MR read protocol [100] exchanges a maximum of $2q$ messages for each read. Listeners requires up to $3q$ messages when there is no concurrency. In particular, line R14 adds a new round of messages. Additional messages are exchanged when there is concurrency because the nodes echo all concurrent write messages to the reader. If c writes are concurrent with a particular read then that read will use $3q + cq$ messages.

For some systems, there is little or no concurrency in the common case [18, 56]. 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 unidirectional. Thus, the Listeners protocol will not wait for $3q + cq$ message roundtrips. This is apparent in the experimental results of the next section.

3.6.3 Experimental Evaluation of Overhead

Listeners uses additional messages compared to previous protocols to reduce the number of nodes and improve consistency semantics. These messages are not a performance bottleneck, however: the overhead is limited to one message per node for each read when no write is concurrent with the read; otherwise, the number of additional messages per node is proportional to the number of concurrent writes.

We construct a simple prototype to measure the overhead of the extra messages used to deal with concurrency in Listeners. The prototype is written in C++, stores data in main memory, and communicates using TCP. We implemented the f-threshold version of Listeners.

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. Writers repeatedly write 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.

Figure 3.5 shows the read latency in milliseconds 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.

3.6.4 Load and Throughput

The throughput benefits of using quorums can be measured using the *load factor* [119]. We must explain a few terms before we can define the load factor.

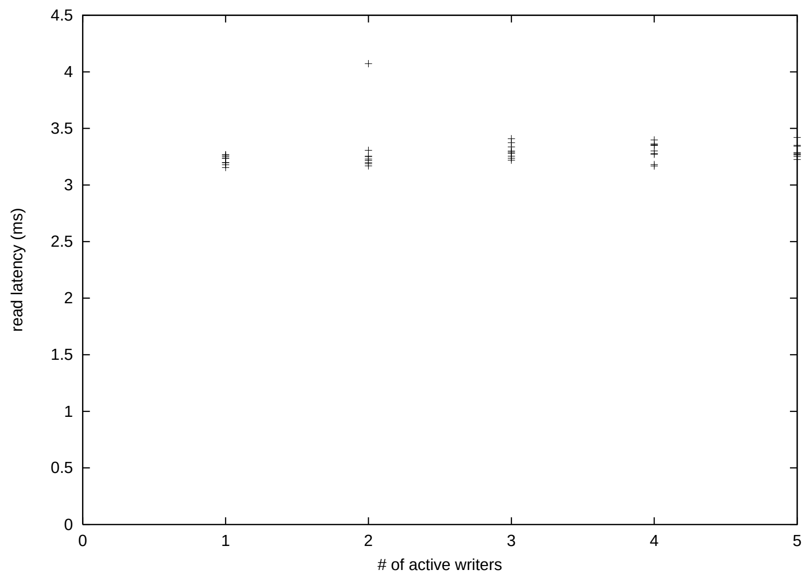

Figure 3.5: Read latency (ms)

Given a quorum system $\mathcal{Q}$, an *access strategy* w is a probability distribution on the elements of $\mathcal{Q}$: i.e., $\sum_{Q \in \mathcal{Q}} w(Q) = 1$. $w(Q)$ is the probability that quorum Q will be chosen when the service is accessed. Load is then defined as follows:

Definition 8. *Let an access strategy w be given for a quorum system $\mathcal{Q} = \{Q_1, ..., Q_m\}$ over a universe U of nodes. For a node $u \in U$, the load induced by w on u is $l_w(u) = \sum_{\{Q_i : u \in Q_i\}} w(Q_i)$. The load induced by a strategy w on a quorum system $\mathcal{Q}$ is*

$$L_w(\mathcal{Q}) = \max_{u \in U} \{l_w(u)\}.$$

The load factor *(or just* load*) on a quorum system $\mathcal{Q}$ is*

$$L(\mathcal{Q}) = \min_{w} \{L_w(\mathcal{Q})\},$$

where the minimum is taken over all strategies.

protocol	read load	write load	n	semantics	Byz. clients	signatures
f-masking [100]	$\frac{1}{n}\lceil\frac{n+2f+1}{2}\rceil$		$4f+1$	safe	no	no
Grid [100]	$\frac{(2f+2)\sqrt{n}-(2f+1)}{n}$		$(3f+1)^2$	safe	no	no
f-dissemination [100]	$\frac{1}{n}\lceil\frac{n+f+1}{2}\rceil$		$3f+1$	regular	yes	yes
Listeners	$\frac{1}{n}\lceil\frac{3n+3f+1}{4}\rceil$		$3f+1$	atomic	no	no
Byzantine Listeners	$\frac{1}{n}\lceil\frac{3n+3f+1}{4}\rceil$	1	$3f+1$	atomic	yes	no

Table 3.5: Comparison of register protocols

For example, in a universe U of three nodes, where $\mathcal{Q}$ is the set of all subsets of U of size 2, the load factor on U is $\frac{2}{3}$ (the random access strategy induces this load factor). Adding servers to reduce the load of a system from 1 to x multiplies the throughput by $1/x$.

Table 3.5 shows the load factor and other information for Listeners, Byzantine Listeners, and previous work. The third column indicates the number of nodes needed to tolerate f Byzantine nodes, the fourth gives the semantics of the register that the protocol implements, the fifth indicates whether the protocol tolerates Byzantine clients, and the sixth indicates whether the protocol requires digital signatures. In order to tolerate Byzantine clients, Byzantine Listeners instructs nodes to forward "STORE" messages to each others. The message is forwarded to every node, resulting in a write load factor of 1.

Listeners and Byzantine Listeners have a higher load than the other protocols, but in return offer stronger semantics and use the minimal number of nodes $(3f+1)$ to tolerate f Byzantine nodes. They do not need digital signatures and Byzantine Listeners can tolerate Byzantine clients.

Even though a low load means that the register's throughput can increase if servers are added, adding servers is rarely an effective method for increasing throughput. Consider for example the f-dissemination protocol: to go from a load of 0.75 to 0.6, the number of nodes must grow from $n = 4$ to $n = 10$. We must more than double the number of nodes to get a 25% increase in throughput ($0.75 * (1/1.25) = 0.6$)! The Grid protocol has a better load, but requires at least 16 nodes to tolerate a single Byzantine node. Purchasing nodes that are more powerful (i.e. have a faster processor and more memory) is a more practical approach to increasing throughput. Another option, if the system comprises several registers, is to run each register on a different set of nodes.

3.6.5 Live Lock

In a system such as Listeners or Byzantine Listeners, calls to **read**() and **write**(...) must complete even if the system is under a heavy load. Writes cannot starve in either protocol, because their execution is independent of concurrent reads. Reads, however, can be starved if an infinite number of writes are in progress and if the nodes always choose to service the writes before sending the "VALUE" messages.

There is an easy way to guarantee read starvation does not happen. When serving a write request while a read is in progress, nodes queue a "VALUE" message (Figure 3.3, line S5). Liveness of both **read**() and **write**(...) is guaranteed, provided nodes send these "VALUE" messages before processing the next "QUERY_TS" message (the order in which nodes go through messages was previously unspecified). The client running **read**() will therefore eventually receive the "VALUE" messages it needs to complete even if an arbitrary number of writes are concurrent with **read**().

Another related concern is that of latency: can reads become arbitrarily delayed? In the asynchronous model, there is no bound on the duration of reads. However, if we assume writes never last longer than w units of time, there are z concurrent writes, and there are n nodes, then we can show a bound on latency.

In the worst case (taking failures into account) reads will be delayed by no more than $min(z * w, n * w)$. This result follows because in the worst case f nodes are faulty and return very high timestamps so that only one row of $answer[\,][\,]$ contains answers from correct nodes. Also, in the worst case each entrance (line R6) occurs just before the monitored write can be read. The second term follows from the maximal number of entrances, n.

3.6.6 Engineering an Asynchronous Reliable Network

The standard model for Byzantine quorum systems has been reliable asynchronous links, and we have followed that model. There are difficulties in implementing this model. If physical links are unreliable, then the abstraction of reliable links can be created by buffering messages that are sent and using a retransmission protocol. If the recipient is subject to Byzantine failures, then a faulty receiver can omit acknowledgments and thus prevent the sender from ever deleting buffered messages. This is clearly a problem since memory is finite. Nonetheless, one can engineer a reasonable approximation of an asynchronous reliable network abstraction when one can (i) restrict the failures to which the system or the network layer is vulnerable or (ii) restrict the workload so that infinite buffering is not a concern. To illustrate when a reliable network abstraction can be built, we provide a few examples of both types of restriction below.

Restricting failures. If nodes fail only by crashing and restart afterwards, then implementing a reliable network abstraction 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[5] [117, 75] store unacknowledged messages on an on-disk log. It may be safe in practice to assume that it is extremely unlikely that the log will overflow by assuming (i) a large log, (ii) a reasonable bound on crash or partition durations, and (iii) 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, even though the software running on nodes can fail arbitrarily, the hardware in the network components ensures that acknowledgments are sent. This hardware may be less vulnerable to failures. Examples include "System/Storage Area Networks" (SANs) (such as Fibre Channel [137]), 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 [141]), and networks with automatic link-level retransmission [129]. 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 [29] may in this case regard the packet as successfully sent, even if no acknowledgments are received; such a sys-

[5]Even though these are not technically a network abstraction, they provide **send**(...) and **receive**(...) operations that behave as if the underlying network were reliable.

tem 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 from some failures is to rely on protection across software modules. For example, in some systems the network drivers 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 network failures, some systems 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 [117] 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.

3.7 Related Work

Although both Byzantine failures [54] and quorums systems [59] 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 [100, 101]. They showed how to use quorums to build a regular register with $3f+1$ nodes (with digital signatures) or a safe register with $4f+1$ nodes (without).

Bazzi [22] explored Byzantine quorums in the synchronous model with reliable channels. In that model it is possible to build atomic registers with fewer nodes ($f+1$ for self-verifying data, $2f+1$ otherwise). This result is not directly comparable to ours since it uses a different model.

Bazzi [23] 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 nodes without concern with whether these nodes are responsive. According to this definition, all the protocols presented here use non-blocking quorum systems.

Several papers [23, 103, 119] study the load of Byzantine quorum systems, a measure of how increasing the number of nodes influences the amount of work of each individual node. 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 nodes necessary to tolerate a given fault threshold (or failure scenarios).

Phalanx [101] builds shared data abstractions and provides a mutual exclusion service, both of which can tolerate Byzantine failure of nodes. Phalanx can provide safe semantics despite Byzantine clients and unreliable links, using digital signatures and $4f + 1$ nodes.

Castro and Liskov [31] present a replication algorithm that requires $3f + 1$ nodes and, unlike most of the work presented above, can tolerate unreliable network links and faulty clients. Their protocol uses synchrony assumptions and cryptography to produce self-verifying data, and it provides linearizability. It is fast in the common case (in the sense that in the absence of failures, their NFS implementation completed the Andrew benchmark within 3% of an unreplicated system). Our work shows that safe semantics cannot be provided using fewer nodes.

Attiya, Bar-Noy and Dolev [11] implement an atomic single-writer multi-reader register over asynchronous links, 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.

Our idea of using information from concurrent writes through the listeners pattern has inspired other researcher Goodson et al. [61] to provide an atomic register without signatures. Their protocol uses erasure codes to write large data efficiently and can tolerate crash and Byzantine failures (the hybrid failure model [152]), but when tolerating only Byzantine failures it requires $4f+1$ nodes instead of $3f+1$ in our case.

Bazzi and Ding [21] use the same technique to provide an atomic register without signatures. Their variant has a lower load than our protocol (in the sense that a smaller fraction of nodes need to service requests) but requires $4f+1$ nodes.

Both protocols use information from concurrent writes, but instead of forwarding information from concurrent operations they keep a history of all current and past writes. While in theory this approach requires infinite storage, in practice this appears not to be an issue [149].

3.8 Conclusion

This chapter examines Byzantine fault-tolerant registers in an asynchronous setting. The first contribution is the proof of a tight bound on the number of nodes: we show that $3f+1$ nodes are necessary to provide even safe register semantics.

The other two contributions are protocols that match the lower bound of $3f+1$ nodes and provide atomic semantics. The first, Listeners, assumes correct clients and the second, Byzantine Listeners, tolerates Byzantine clients but sends more messages than the first. Neither protocol requires self-verifying data. The protocols derive from quorum systems but use an original communication pattern, the Listeners. The protocols use the fail-prone error model but they can be adapted to the f-threshold model. Previous protocols for the same model [100] required $4f + 1$ nodes, did not tolerate Byzantine clients, and provided only regular semantics. Several protocols [31, 100, 101, 111, 130] use digital signatures to reduce the number of nodes. It was therefore surprising to us that we were able to employ this same minimum number of servers without using digital signatures. Instead, our protocols send additional messages if concurrent writes are in progress, and Byzantine Listeners uses MACs.

Chapter 4

Non-Confirmable Semantics for Cheaper Registers

4.1 Introduction

In the previous chapter we have seen that the minimal number of nodes to build a safe asynchronous register that can tolerate f Byzantine failures is $3f + 1$. This is a good deal amount of replication, so it is natural to wonder under what conditions it is possible to use fewer replicas.

In this chapter we show that it is possible to build fault-tolerant asynchronous registers using only $2f + 1$ nodes instead of $3f + 1$ (a reduction of up to 33%). To create these cheaper registers, we remove the requirement that a writer know when its **write**(...) completes. Of course, the **write**(...) function still returns and our writes are still guaranteed to complete eventually—the only change is that these two events are distinct, and the writer is not necessarily informed when the **write**(...) completes. We call the resulting semantics *non-confirmable*.

The benefit of the new semantics is a reduced cost: we show that the minimal number of nodes to achieve non-confirmable safe registers in the asynchronous model that can tolerate f Byzantine failures is $2f + 1$. We show that this bound is tight by presenting a new variant of Listeners that matches our lower bound. In addition, the new protocol can provide stronger semantics than just non-confirmable safe: it implements a non-confirmable regular register using the optimal number of nodes $(2f + 1)$.

Non-confirmable semantics are sufficient when either (i) the writer does not need to know when the write completes, or (ii) an application-level mechanism lets the writer know that its value was read. An example of the first case is a sensor that periodically writes some data to shared memory (it does not need to know when the write ends); an example of the second case is several nodes communicating through shared memory: receiving a response means that a prior communication has been received—there is no need for a separate confirmation mechanism built into the shared memory. These acknowledgments cannot always be added: in the case of two nodes communicating through a shared memory, for example, if the first node were to wait for this acknowledgment, then a crash of the second node would cause the first node to violate liveness.

4.2 Non-Confirmable Semantics Defined

If a protocol defines write completion 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 do not support blocking writes. The SBQ protocol [111], 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 always determine when it happens.

4.3 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.

To prove that it is impossible to implement a non-confirmable safe register using $2f$ nodes in the asynchronous model, we show that under these assumptions any protocol must violate either safety or liveness. Recall that U is the set of nodes.

Lemma 22. *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 $S' \subseteq S$.*

Proof. 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 are also influenced by some other server $x \notin S$. Since $|U - S| = f$, it is possible for all servers in $U - S$ are faulty. In that case, x is faulty and may crash (since x is in $U - S$). Since P is influenced by x and x crashed, that execution of P is not live. □

Lemma 23. *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.

Proof. 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 22 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 22 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. □

Theorem 6. *In the reliable authenticated asynchronous model with Byzantine failures, no live protocol can implement a shared register using $2f$ servers.*

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

Note that the proof of Lemma 23 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

```
W1   $\textbf{write}(D)$ :
W2       send (“QUERY_TS”) to all servers in some $A \in \mathcal{A}$
W3       loop :
W4           receive answer (“TS”, $ts$) from server $s$
W5           $current[s] := ts$
W6       until $\exists Q \in \mathcal{Q} : Q \subseteq current[\,]$ // a quorum answered
W7       $max\_ts := max\{current[\,]\}$
W8       $my\_ts := min\{t \in C_{ts} : max\_ts < t \wedge last\_ts < t\}$
         // $my\_ts \in C_{ts}$ is larger than all answers and previous timestamp
W9       $last\_ts := my\_ts$
W10      send (“STORE”, $D, my\_ts$) to all servers in some $A \in \mathcal{A}$

R1   $(D, ts) = \textbf{read}()$ :
R2       send (“READ”) to all servers in some $A \in \mathcal{A}$
R3       loop :
R4           receive answer (“VALUE”,$D, ts$) from server $s$ // (possibly several answers per server)
R5           if $ts > largest[s].ts$ : $largest[s] := (ts, D)$
R6           if $s \notin S$ : // we call this event an “entrance”
R7               $S := S \cup \{s\}$
R8               $T :=$ the $f + 1$ largest timestamps in $largest[\,]$
R9               for each $isvr$, for each $jtime \notin T$ : delete $answer[isvr, jtime]$
R10              for each $isvr$ :
R11                  if $largest[isvr].ts \in T$ : $answer[isvr, largest[isvr].ts] := largest[isvr]$
R12          if $ts \in T$ : $answer[s, ts] := (ts, D)$
R13      until $\exists D', ts', Q_r : Q_r \in \mathcal{Q} \wedge (\forall i : i \in Q_r : answer[i, ts'] = (ts', D'))$
         // i.e., loop until a quorum of servers agree on a (ts,D) value
R14      send (“READ_COMPLETE”) to all servers in $A$
R15      return $(D', ts')$
```

Figure 4.1: Non-Confimable Listeners, client protocol

have all the necessary information to create the messages they send (e.g. signatures from other nodes).

4.4 Non-Confirmable Listeners Protocol

The confirmable Listeners protocol of Chapter 3 requires at least $3f + 1$ servers. In this section we show how this number can be reduced to $2f + 1$ if the protocol is made non-confirmable. Figure 4.1 shows the pseudocode for the Non-Confirmable Listeners protocol. This protocol is based on the Listeners protocol of Figure 3.1.

Since in a non-confirmable protocol the writer is not required to know when the write completes, we can delete lines W11 to W14 of the **write**(...) function from the Listeners protocol (Figure 3.1), 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.

Weakening the semantics to non-confirmable allows us to modify not only the protocol, but also the quorum construction. Using this different quorum construction will allow us to reduce the number of nodes needed to tolerate f Byzantine nodes. The quorum construction still still contains a gateway quorum system for $\mathcal{A}$, but $\mathcal{Q}$ does not need to be a dissemination quorum anymore: instead it can be a regular quorum system (i.e. quorums can intersect in a single node). Recall that a gateway quorum system satisfies:

G-Consistency $\forall A_1, A_2 \in \mathcal{A}\ \forall B \in \mathcal{B}\ \exists Q \in \mathcal{Q} : (Q \subseteq A_1) \cap (A_2 \wedge Q \cap B = \emptyset)$

This quorum construction is sufficient because eliminating the acknowledgments eliminates a constraint on the overlap of quorums. The Listeners protocol requires the quorum intersection because readers must be able to identify the correct value as soon as **write**(...) (i.e. as soon as a quorum acknowledges the "STORE" message). In non-confirmable Listeners, instead, writes can complete after the **write**(...) method returns, so we are only constrained by the intersection between the quorum of nodes queried by the reader and the quorum of *correct* nodes that are guaranteed eventually to receive the "STORE" message. As a result, we can reduce the number of nodes needed to tolerate f Byzantine failures: if $\mathcal{B}$ is all subsets of f servers, then $\mathcal{Q}$ is all subsets of size q for $q = \lceil \frac{n+1}{2} \rceil$ servers and $\mathcal{A}$ all subsets of $\lceil \frac{n+f+q}{2} \rceil$ servers. To tolerate f Byzantine faults in this model, Non-Confirmable Listeners needs only $2f + 1$ nodes.

Recall that in non-confirmable protocols, the **write**(...) function does not determine when the write has completed: instead, the completion must be specified separately. We therefore specify that the write completes when some quorum $Q \in \mathcal{Q}$ of *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: they cannot delay writes any more than crashed servers would. The G-Availability property guarantees that such a quorum will be found eventually.

This protocol requires only $2f + 1$ servers and provides non-confirmable regular semantics. As shown in Theorem 6, fewer servers do not suffice, so we conclude that $2f + 1$ is the optimal number of servers for non-confirmable protocols.

We do not have a non-confirmable protocol for atomic semantics. Pierce [125] presents a general technique to transform regular protocols into ones that satisfies atomic semantics, but unfortunately this technique does not apply to non-confirmable protocols.

4.5 Correctness

Theorem 7. *The non-confirmable Listeners protocol is live and provides non-confirmable regular semantics.*

Lemma 24. *The non-confirmable Listeners protocol never violates regular semantics.*

Proof. When a read R completes, the reader decides on a value that has been vouched for by a quorum $Q_r \in \mathcal{Q}$ of servers (line R13). Since we defined writes to complete when a quorum of correct servers have received the "STORE" message (Section 4.4), by definition there is a quorum Q of correct servers that have seen the latest completed write with respect to R. By definition of quorums, Q and Q_r intersect in at least one server s. This server is correct and it has seen the latest completed write.

Since s is correct, it follows the protocol and therefore sends to R 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 25. *The Non-Confirmable Listeners protocol is live.*

Proof. The beginning of the proof for liveness is identical to the proof for the confirmable case in Section 3.4.4, 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 echoes, the quorum size must be modified as follows. The write protocol eventually reaches all correct servers in some $A \in \mathcal{A}$. The read operation contacts a quorum A_2 of servers and the intersection of the two quorums contains (by G-Availability) a quorum Q of correct servers. These will send the correct answers required for completion. □

Chapter 5

Dynamic Quorums

5.1 Introduction

Most Byzantine quorum system (BQS) protocols set two parameters—N, the set of servers in the quorum system, and f, the resilience threshold denoting the maximum number of servers that can be faulty[1]—and treat them as constants throughout the life of the system. The rigidity of these static protocols is clearly undesirable.

Fixing f forces the administrator to select a conservative value for this resilience threshold: one that can tolerate the worst case-failure scenario. Usually, this worst-case scenario will be relatively rare; however, since the value of f determines the size of the quorums, in the common case quorum operations will be forced to access unnecessarily large sets, with obvious negative effects on performance.

Fixing N not only prevents the system administrator from retiring faulty or obsolete servers and substituting them with correct or new ones, but also greatly reduces the advantages of any technique designed to change f dynamically. For a

[1]The previous chapters consider generalized fault structures, offering a more general way of characterizing fault-tolerance than a threshold. However, such structures remain static.

given Byzantine quorum protocol, N must be chosen to accommodate the maximum value f_{max} of the resilience threshold, independent of the value of f that the system uses at a given point in time. Hence, in the common case the degree of replication required to tolerate f_{max} failures is wasted.

In this chapter we propose a methodology for transforming static Byzantine quorum protocols into dynamic ones where both N and f can change, growing and shrinking as appropriate[2] during the life of the system. We have successfully applied our methodology to several Byzantine quorum protocols [61, 100, 101, 111, 126]. The common characteristic of these protocols is that they only use the Q-RPC primitive [100] for communication. A Q-RPC contacts a responsive quorum of servers and collects their answers, making it a natural building block for implementing quorum-based read and write operations. Our methodology is simple and non-intrusive: all that it requires to make a protocol dynamic is to substitute each call to Q-RPC with a call to a new primitive, called DQ-RPC for *dynamic* Q-RPC. DQ-RPC maintains the properties of Q-RPC that are critical for the correctness of Byzantine quorum protocols, even when N and f can change.

The main difficulty in defining DQ-RPC to minimize changes to existing protocols comes from proving that read and write operations performed on the dynamic version of a protocol maintain the same consistency semantics of the operations performed on the static version of the same protocol. In the static case, these proofs rely on the intersection properties of the responsive quorums contacted by Q-RPCs. Unfortunately, these proofs do not carry easily to DQ-RPC. When N changes, it is no longer possible to guarantee quorum intersection: given any two distinct times t_1 and t_2, the set of machines in N at t_1 and t_2 may be completely disjoint. We address

[2]We focus on the mechanisms necessary for supporting dynamic quorums systems (i.e. a quorum system that can change at runtime). A discussion of the policies used to determine when to adjust N and f is outside the scope of this chapter. Some examples of such policies are given in [8, 78].

this problem by taking a fresh look at what makes Q-RPC-based static protocols work.

Traditionally, the correctness of these protocols relies on properties of the quorums themselves, such as intersection. Instead, we focus our attention on the properties of the *data* that is retrieved by quorum operations such as Q-RPC. In particular, we identify two such properties, *soundness* and *timeliness*. Informally, soundness states that the data that clients gather from the servers was previously written; timeliness requires this data to be as recent as the last written value. We call these properties *transquorum* properties, because they do not explicitly depend on quorum intersection. We prove that transquorum properties are sufficient to guarantee the consistency semantics provided by each of the protocols that we consider. Now, all that is needed to complete our transition from static to dynamic protocols is to show an instance of a quorum operation that satisfies the transquorum properties even when f and N are allowed to change: we conclude the chapter by showing that DQ-RPC is such an operation.

The rest of the chapter is organized as follows. We cover related work and system model, respectively, in Section 5.2 and Section 5.3. We specify the transquorum properties in Section 5.4 and show in Section 5.5 that our DQ-RPC satisfies the transquorum properties before concluding. Sections with additional detail have been placed between the conclusion and the end of this chapter.

5.2 Related Work

Alvisi et al. [9] are the first to propose a dynamic BQS protocol. They let quorums grow and shrink depending on the value of f, which is allowed to range dynamically

[3]Partial-atomic semantics guarantees that reads either satisfy atomic semantics or abort [126].

name	tolerates (crash,Byz)	signatures	client failures	semantics
crash	$(f, 0)$	no	crash	atomic
U-dissemination [111]	$(0, b)$	yes	crash	atomic
hybrid-d [61]	(f, b)	yes	crash	atomic
U-masking [126]	$(0, b)$	no	correct	partial-atomic[3]
hybrid-m [61]	(f, b)	no	correct	partial-atomic[3]
Phalanx [101]	$(0, b)$	server only	Byzantine	partial-atomic[3]
hybrid Phalanx	(f, b)	server only	Byzantine	partial-atomic[3]

Table 5.1: List of quorum protocols that can be made dynamic using DQ-RPC

within an interval $[f_{min}, ..., f_{max}]$. This flexibility, however, comes at a cost: because their protocol does not allow N to change, it requires $2(f_{max} - f_{min})$ more servers than an equivalent static protocol to tolerate a maximum of f_{max} failures.

The Agile store [78] modifies the above protocol by introducing a special, fault-free node that monitors the set of servers in the quorum system. The monitor tries to determine which are faulty and to inform the clients, so that they can find a responsive quorum more quickly. In the Agile store, servers can be removed from N, but not added. Therefore, if the monitor mistakenly identifies a node as faulty and removes it from N, the system's resilience is reduced: the system tolerates f_{max} Byzantine faulty servers only as long as the monitor never makes such mistakes.

The Rosebud project [132] shares several of our goals. Rosebud envisions a dynamic peer to peer system, where servers can fail arbitrarily, the set of servers can be modified at run-time, and clients use quorum operations to read and write variables. When we originally published our dynamic quorum system, only a preliminary design document for Rosebud was available [133]. The authors have since published a full description [132]. Rosebud differs from out work in that it is a specific dynamic system rather than a framework for transforming static protocols into dynamic ones. Rosebud allows N to change only at pre-set intervals. In contrast, we allow operations to continue even as N is changing, and we allow N (and f) to change at any time. Perhaps the most important difference lies in the timing assumptions: our protocol is purely asynchronous whereas Rosebud relies on weak synchrony assumptions not only for the liveness of its embedded replicated state machine, but also for safety ([131], Section 3.6, second paragraph).

Several quorum-based protocols allow to change N and f but only tolerate crash failures. Rambo and Rambo II [98, 135] provide the same interface as our protocols: read, write and reconfigure. They guarantee atomic semantics in an unreliable asynchronous network despite crash failures.

In GeoQuorums [46] the world is split into n focal points and servers are assigned to the nearest (geographically) focal point. The system provides atomic semantics as long as no more than f focal points have no servers assigned to them. Servers can join and leave; however, neither n nor f can change with time.

Abraham et al. [2] target large systems, such as peer-to-peer, where it is important for clients to issue reads and writes without having to know the set of all servers, and it is important for servers to join and leave without having to contact all servers. Their *probabilistic quorums* meet these goals (for example, clients only need to know $O(\sqrt{n})$ servers), provide atomic semantics with high probability, and can tolerate crash failures of the servers.

View-oriented group communication systems provide a membership service whose task is to maintain a list of the currently active and connected members of a group [36]. The output of the membership service is called a *view*. If we consider the set of servers in the quorum system as a group, then in our protocol the membership service is trivially implemented by an administrator, who is solely responsible for steering the system from view to view (see Section 5.5.1).

An interesting property of our protocol is that it allows processes that are outside the quorum systems —i.e. the clients in our protocol—to query servers within the quorum system to learn the current view. Note that our clients do not learn about views from the membership service, but rather indirectly, through the servers. Nonetheless, our protocol guarantees that, despite Byzantine failures of some of the servers, a correct client will only accept views created by the administrator and will never accept as current a view that is obsolete (see Section 5.5.1).

5.3 System Model

Our system consists of a universe U of servers. Only a subset N of these servers are *commissioned* at any point in time. We use n to denote the number of servers in N. Servers can join or leave N at any point in time, i.e. both N and n can change during execution. To prevent Sybil attacks [47], we require strong identities: each server has an identity that cannot be forged and there is an external mechanism for specifying which identities can participate (for example, the identity must bear a signature from the administrator). Servers can be either correct or faulty. A correct server follows its specification; a faulty server can arbitrarily deviate from its specification. We do not assume that decommissioned servers ($U-N$) are correct: they may all be faulty. The set of clients of the service is disjoint from U. Clients perform *read* and *write* operations on the variables stored in the quorum system. We assume that these operations return only when they complete (i.e. we consider confirmable operations [111]).

Our dynamic quorum protocols maintain the same assumptions about client failures of their static counterparts. Clients communicate with servers over point-to-point, authenticated, asynchronous fair channels. A fair channel guarantees that a message sent an infinite number of times will reach its destination an infinite number of times. We allow channels to drop, reorder, and duplicate messages.

5.4 A New Basis for Determining Correctness

The first step in our transition to dynamic quorum protocols is to establish the correctness of the static protocols we consider (shown in Table 5.1) on a basis that does not rely on quorum intersection. To do so, we observe that at the heart of all these protocols lies the Q-RPC primitive [100]. Our approach to extend quorum protocols to the case where servers are added and removed (and thus quorums may not intersect anymore) is to define correctness in terms of the properties of the data returned by quorum-based operations such as Q-RPC. In this section, we first specify two properties that apply to the data returned by Q-RPC; then, we prove that these properties are sufficient to ensure correctness. In Section 5.5 we will show that it is possible to implement Q-RPC-like operations that guarantee these properties even when quorums do not intersect.

5.4.1 The Transquorum Properties

In the protocols listed in Figure 5.1, quorum-based operations such as Q-RPC are the fundamental primitives on top of which read and write operations are built. Not all Q-RPCs are created equal, however. Some Q-RPC operations change the state of the servers (e.g. when the message passed as an argument contains information that the servers should store), others do not. Some Q-RPCs need to return the latest data actually written in the system, others only need to return data that is not obsolete, whether it was written or not. To capture this diversity, we use two properties, timeliness and soundness. We call them *transquorum* properties

```
read()
1.  $Q$ := Q-RPC("READ")
    // Q: set of $\langle ts, writer\_id, data\rangle_{writer}$
2.  $r := \phi(Q)$
    // returns largest valid value
3.  $Q$ := Q-RPC("WRITE",$r$)
4.  return $r.data$

write($D$)
1.  $Q$ := Q-RPC("GET_TS")
2.  $ts := max\{Q.ts\} + 1$
3.  $m := \langle ts, writer\_id, D\rangle_{writer}$
4.  $Q$ := Q-RPC("WRITE",$m$)
```

```
read()
1.  $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")
    // Q: set of $\langle ts, writer\_id, data\rangle_{writer}$
2.  $r := \phi(Q)$
    // returns largest valid value
3.  $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$r$)
4.  return $r.data$

write($D$)
1.  $Q$ := TRANS-Q$_{\mathcal{T}}$("GET_TS")
2.  $ts := max\{Q.ts\} + 1$
3.  $m := \langle ts, writer\_id, D\rangle_{writer}$
4.  $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$m$)
```

Figure 5.1: U-dissemination protocol (fail-stop clients). On the left: Q-RPC. On the right: TRANS-Q.

because, as we will see in Section 5.5, they do not require quorum intersection to hold. Intuitively, timeliness says that any read value must be as recent as the last written value, while soundness says that any read value must have been written before. Q-RPC operations return data that reflect the state of the servers at a given point in time, so the concepts of timeliness and soundness apply to them. Note that not all Q-RPCs need to be both timely and sound. For example, Q-RPCs used to gather the current timestamps associated with the value stored by a quorum of servers need not be sound—all that is required is that the returned timestamps be no smaller than the timestamp of the last write.

We then define three sets $\mathcal{W}$, $\mathcal{R}$, and $\mathcal{T}$ of Q-RPC-like quorum operations. Each Q-RPC-like operation in a protocol belongs to zero or more of these sets.

Let $w \rightarrow r$ (w "happens before" r) indicate that quorum operation w ended (returned) before the quorum operation r started (in real time). Further, let o be an ordering function that maps each quorum operation to an element of an ordered set $\mathcal{M}$. We define the transquorum properties as follows:

$$
\begin{aligned}
&\text{(timeliness)} \quad && \forall w \in \mathcal{W}, \forall r \in \mathcal{R} \cup \mathcal{T}, o(r) \neq \bot \ : \\
& && \quad w \rightarrow r \Longrightarrow o(w) \leq o(r) \\
&\text{(soundness)} \quad && \forall r \in \mathcal{R}, o(r) \neq \bot : \\
& && \quad \exists w \in \mathcal{W} \text{ s.t. } r \not\rightarrow w \ \wedge \ o(w) = o(r)
\end{aligned}
$$

In this chapter we always choose o so that when applied to a Q-RPC-like operation x, it returns both a timestamp and the data that is associated with x (i.e. either read or written). This allows us to use the timeliness property to ensure that readers get recent timestamps and the soundness property to ensure that reads get data that has been written.

5.4.2 Proving Correctness with Transquorums

Transquorum properties suffice to prove that the protocols listed in Figure 5.1 correctly provide the consistency semantics that they advertise. We present the complete set of proofs in the Appendix. For conciseness, in this chapter we limit ourselves to the first three protocols in the figure. All three protocols have the same client code, shown on the left in Figure 5.1 and all three guarantee atomic semantics. The server code in all these is also identical: servers simply store the highest timestamped data they receive and send back to the client the data or its timestamp (in reply to READ or GET_TS requests, respectively). The protocols differ in the size of the quorums they use and in the degree of fault-tolerance they provide: U-dissemination protocols [110] (a variant for fair channels of the dissemination protocol presented in [100]) can tolerate b Byzantine faulty servers, crash can tolerate f fail-stop faulty servers, and hybrid-d can tolerate both b Byzantine failures and f fail-stop failures ($f + b$ failures in total). To simplify our discussion, since the three client protocols are identical, we will only discuss the U-dissemination protocol here; all we say also applies to the crash and hybrid-d protocols as well, except that the crash protocol does not use any signatures. Another simplification is that we show the transformation on the non-optimized version of the U-dissemination protocol. Read operations can in fact be shortened to a single message round-trip in the common case by skipping the write-back when it is not necessary (see Section 5.5.4).

Dissemination protocols with transquorums

To illustrate that we only rely on the transquorum properties and not on the specific implementation of Q-RPC, we replace all Q-RPC calls in the protocol (Figure 5.1) with an "abstract" function TRANS-Q that we postulate has the transquorum properties. TRANS-Q takes the same arguments and returns the same type of values as Q-RPC.

The U-dissemination protocol on the right of Figure 5.1 uses TRANS-Q as its low-level quorum communication primitive. We have annotated each call to indicate which set it belongs to ($\mathcal{R}, \mathcal{W}$, or $\mathcal{T}$).

Operations of this form	are assigned this order,	set
r : TRANS-Q("$READ$")	$o(r) = \phi(r_{ret})$	$\mathcal{R}$
w : TRANS-Q("$WRITE$", ts, w_id, D)	$o(w) = (w_{arg}.ts, w_{arg}.w\_id, w_{arg}.D)$	$\mathcal{W}$
t : TRANS-Q("GET_TS")	$o(t) = (max(t_{ret}) + 1, \perp, \perp)$[4]	$\mathcal{T}$

Figure 5.2: The o mapping

[4]We do not explicitly require this value to be larger than any timestamp previously sent by this client because we do not allow clients to issue multiple concurrent writes.

Notation $\langle a \rangle_b$ denotes that a is signed by b. Note that data is signed before being written and verified before being read. The function $\phi(Q)$ returns the largest value in the set Q that has a valid client signature using lexicographical ordering. Values are triples $(ts, writer\_id, D)$, so ϕ selects the largest valid timestamp, using *writer_id* and then D to break ties.

We assign each TRANS-Q quorum operation to one of the sets $(\mathcal{R}, \mathcal{W}$ or $\mathcal{T})$ and define the ordering $o(x)$ for each quorum operation x. This assignment is shown in Figure 5.2 and is fairly intuitive: operations that change the server state have been assigned to $\mathcal{W}$ and the ordering function consists either of what is being written, or of what the caller extracts from the set of responses to its query. Operations that determine the value to be read are assigned to $\mathcal{R}$ and the remaining operations are assigned to $\mathcal{T}$. More precisely, to define $o(x)$ we observe that any quorum operation x has two parts: the arguments passed to x and the value that x returns. We use the notation x_{arg} to refer to the arguments passed to the x operation, and x_{ret} to indicate the value returned by x (always a set).

We want to show that the U-dissemination protocol with TRANS-Q operations offers atomic semantics. Atomic semantics requires all readers to see the same ordering of the writes and furthermore that this order be consistent with the order in which writes were made. Note that atomic semantics is concerned with *user-level* (or, simply, *user*) reads and writes, not to be confused with the *quorum-level operations* (or, simply, *quorum operations*) such as Q-RPC and TRANS-Q. We use lowercase to denote quorum-level operations, and uppercase to denote user-level operations (e.g. R or W). Similarly, we use mapping o to denote the ordering constraint that the transquorum properties impose on quorum operations, and mapping O to denote the ordering constraints imposed by the definition of atomic semantics on user read and write operations.

Atomic semantics can be defined precisely as follows.

Definition 9. *Every user read R returns the value that was written by the last user write W preceding R in the ordering "$<$". "$<$" is a total order on user writes, and $W \to X \implies W < X$ and $X \to W \implies X < W$ for any user write W and user read or user write X.*

To show that U-Dissemination is atomic we construct the ordering "$<$" with O, which maps every user read and write operation to an element of some ordered set $\mathcal{M}'$: $X < X' \iff O(X) < O(X')$.

We are now ready to prove our first theorem, showing that we can replace Q-RPC with any operation that satisfies the transquorum properties without compromising the semantics of the U-dissemination protocol of Figure 5.1. For simplicity, the initial state of the servers corresponds to a write for the empty value $\emptyset$, so every read is preceded by some write.

Lemma 26. *Ordering relation "$<$" has the following properties: (i) $W \to X \implies W < X$ and (ii) $X \to W \implies X < W$ for any user write W and user read or user write X.*

Proof. We build the ordering O for user reads and writes from the ordering o of the quorum operations that are invoked within these user operations.

The simplest choice would be to set $O(X) = o(x)$ for some quorum operation x that is called as part of the user-level operation X. Unfortunately, we need to take one extra step to make sure that user reads get ordered after the user write whose value they read.

Let $last\_o(X)$ be the value assigned through the o mapping to the last quorum operation in the (read or write) user-level operation X. For a user write operation

W, we define $O(W)$ to be the pair $(last\_o(W), 0)$. For a user read operation R, we define $O(R)$ to be the pair $(last\_o(R), 1)$. The second element in these pairs ensures that user read operations are ordered after the user write whose value they return.

We now show that condition (i) holds. Suppose first that $W \to X$. The U-dissemination protocol is confirmable, so $W \to X$ means that W returns before X starts, and $w \to x$ for all quorum operations w of W and x of X. W ends with a $\mathcal{W}$ operation (see Figure 5.1) and X starts with a quorum operation $t \in \mathcal{R} \cup \mathcal{T}$, so, by timeliness, $last\_o(W) \leq o(t)$. If X is a user read then $O(X)$'s second element is 1, so $O(W) < O(X)$. If X instead is a user write then $last\_o(X) > o(t)$ and therefore $O(W) < O(X)$ still holds.

To show (ii), suppose now that $X \to W$. The last quorum operation of X (regardless of whether it is a user read or write) is a $\mathcal{W}$ operation. The first quorum operation of W, t, is a $\mathcal{T}$ operation. By timeliness, $last\_o(X) \leq o(t)$. The write W then fills in the last two elements of $o(t)$ with its *writer_id* and data D, resulting in a value that is strictly larger than $o(t)$. This value is then passed to the last quorum operation in W, ensuring that $last\_o(W) > o(t)$. Since $last\_o(X) \leq o(t)$ and $o(t) < last\_o(W)$, it follows that $O(X) < O(W)$. $\square$

Lemma 27. *Ordering relation "$<$" is a total order on user writes.*

Proof. By construction of "$<$", any two writes W_1 and W_2 can be compared. To show that $<$ is a total order, we need to also show that $O(W_1) = O(W_2) \implies W_1 = W_2$.

Assume $O(W_1) = O(W_2)$. W_1 and W_2 cannot be performed by different writers, because the o value includes the writer_id, which is different for each writer. And if W_1 and W_2 are performed by the same writer they cannot be distinct. If they were, because user writes are confirmable it would either follow that $W_1 \to W_2$ (and, by (1) $O(W_1) < O(W_2)$, or, symmetrically, that $W_2 \to W_1$ and $O(W_2) < O(W_1)$. $\square$

Lemma 28. *All user reads R return a value written by some user write W, and $R \not\to W$.*

Proof. The value that is returned by read R is the third element in the first quorum operation $r \in \mathcal{R}$ in R. By soundness, this value was passed to some quorum operation in $\mathcal{W}$. Both user-level write and user-level read operations call a $\mathcal{W}$ quorum operation, but we observe that (Figure 5.1) the user-level read calls it with a value that it first gets from a $\mathcal{R}$ quorum operation. We can therefore use soundness repeatedly to show that there must have been some quorum operation $w \in \mathcal{W}$ inside a user-level write operation W such that $o(r) = o(w)$ and $r \not\to w$. This proves the first part of the lemma. Since r did not happen before w, it is also impossible that R happened before W (since the latter would imply the former). This proves the second part of the lemma. $\square$

Lemma 29. *All reads R return the value that was written by the last write W preceding R in the "$<$" ordering.*

Proof. Lemma 28 shows that the value returned by R was written by some write W such that $R \not\to W$. By construction of O, we have $O(W) = (last\_o(W), 0) = (last\_o(R), 0) < (last\_o(R), 1) = O(R)$, so W precedes R in the "$<$" ordering. By definition, for any write W' that is ordered after W in "$<$" we know that $O(W') > O(W)$. Since the pairs for $O(W)$ and $O(W'$ have the same second element, the first element in $O(W')$ must be larger than the first in $O(W)$. Hence, since $O(R)$ and $O(W)$ have the same first element, it follows that W' is ordered in "$<$" after R. Therefore W is the last write preceding R in the "$<$" ordering. $\square$

Theorem 8. *The U-dissemination protocol provides atomic semantics if (i) the TRANS-Q operations have the transquorums properties for the function o defined in Figure 5.2, and (ii) for all $r \in \mathcal{R} : o(r) \neq \bot$.*

Proof. The four lemmas together prove that U-dissemination with transquorums is atomic (Definition 9). The three requirements of the definition are: (i) Every user read R returns the value that was written by the last user write W preceding R in the ordering "<" (shown by Lemma 29), (ii) "<" is a total order on user writes (shown by Lemma 27), and (iii) $W \rightarrow X \implies W < X$ and $X \rightarrow W \implies X < W$ for any user write W and user read or user write X (shown by Lemma 26). □

5.5 Dynamic Quorums

The transquorum properties allow us to reason about quorum protocols without being forced to use quorums that physically intersect. In this section, we leverage this result to build DQ-RPC, a quorum-level operation that satisfies the transquorum properties but also allows both the set of servers and the resilience threshold to be adjusted.

We must first introduce some way to describe how our system evolves over time, as N and f change.

5.5.1 Introducing Views

We use the well-established term *view* to denote the set N that defines the quorum system at each point in time. Views are totally ordered and each view is characterized by a set of attributes. For now we consider the following attributes for view number t: the set of servers $N(t)$ and the resilience threshold $f(t)$. No two views have the same number. In general, view attributes include enough information to compute the quorum size $q(t)$. The responsibility to steer the system from view to view is left with an administrator, who can begin a view change by invoking the **newView**$(\ldots)$ command.

The *administrator* is an abstraction that can be implemented in several ways. It can be a person, a replicated state machine, or a person interacting with the system through a replicated state machine. Having a person involved has advantages because some tasks are difficult to automate, such as deciding to increase f when anticipating future threats or changing N by buying and installing new machines.

When the administrator calls **newView**$(\ldots)$, the view information stored at the servers is updated. We say that a view t *starts* when any server receives a view change message for view t (for example because the administrator called **newView**$(t, \ldots)$). A view t *ends* when a quorum $q(t)$ of servers have processed a message indicating that some later view u is starting. Between starting and ending, the view is *active*. A view may start before the previous view has ended, i.e. there may exist multiple active views at the same time; our protocol makes sure that the register semantics (e.g. atomic) are maintained despite view changes, even if client operations happen concurrently with them.

The **newView**$(\ldots)$ function has the property that after **newView**(t) returns, all views older than t have ended and view t has started. At this point the administrator can safely turn off server machines that are not in view t.

Obviously, we must restrict who can call the **newView**$\ldots)$ function. In our system, this is solely the privilege of the administrator. If the administrator is malicious then we cannot provide any guarantee (for example, a malicious administrator could start a view containing no server to deny service to all clients). However, the system can tolerate crash failures of the administrator (or, when the administrator is a person, the system can tolerate a sudden stop of activity from the administrator).

We say that a server is *correct* in some view t if it follows the protocol from the beginning of time until view t ends. Otherwise, it is faulty in view t. Note that a server may be correct in some view t and faulty in a later view u. However, faulty servers will never be considered correct again. If some server recovers from a failure (for example by reinstalling the operating system after a disk corruption), it takes on a new name before joining the system. The notion of resilience threshold is also parameterized using view numbers. For example, a static U-dissemination protocol requires a minimum of $n \geq 3f + 1$ servers: this requirement now becomes $|N(t)| \geq 3f(t) + 1$ for each view t. Our system assumes that, between the start and the end of view t, at most $f(t)$ of the servers in $N(t)$ are faulty. Since views can overlap, sometimes a conjunction of such conditions must hold at the same time.

5.5.2 A Simplified DQ-RPC

We begin with a simplified version of DQ-RPC that, while suffering from serious limitations, allows us to present more easily several of the key features of DQ-RPC—the full implementation of DQ-RPC is presented in Section 5.5.3.

The easiest way to implement DQ-RPC is to ensure that different views never overlap, i.e. that at any point in time there exists at most one active view. If the administrator copies the data to the new view during a change, we can maintain the transquorum properties. Since we know that the protocols in Figure 5.1 are correct for a static quorum system, we can simply make sure to evolve the system through, as it were, a sequence of static quorum systems. We can do so as follows.

- Replies from servers are tagged with a view number

```
simplified-DQ-RPC($D$) :
1. repeat :
2.     send $D$ to activeServers()
3.     gather responses in $replies_0$
4.     $replies := \{r \in replies_0 : r.sender \in \text{activeServers}()\}$
5. until $\exists t, mt : mt \in replies \wedge mt.tag.t = t$
     $\wedge |\{r \in replies : r.tag.t = t\}| \geq q(|mt.tag.N|, mt.tag.f)$
6. return $\{r \in replies : r.tag.t = t\}$
     // t is the current view associated with this operation
```

Figure 5.3: Simplified Dynamic quorum RPC

- Once a client accumulates $q(t)$ responses tagged with view t, the DQ-RPC returns these responses.

Our simplified DQ-RPC computes two things: a view t (that we call DQ-RPC's *current* view) and a quorum of $q(t)$ responses. It returns the set of responses, and t can be determined by looking at $r.tag.t$ from any response r in the set.

Pseudocode for the simplified DQ-RPC is shown in Figure 5.3. The function $q(n, f)$ computes the quorum size based on the number of servers n and the resilience threshold f. The **activeServers**() function gives the list of servers in active views (views that have started but not ended—there is only one such view at a time in the Simplified DQ-RPC but we will lift this restriction later). Variable *replies* keeps track of all replies from active servers. Simplified DQ-RPC loops until it gets $q(|N(t)|, f(t))$ messages tagged with the same view t (message mt is one of these messages). We write $m.tag$ for the view meta-information tagged onto message m. These tags contain three fields: the set of servers N, the resilience threshold f and the view number t. These tags are attached to every message, servers forward them along with the data that was written. If we assume that clients have some external, infallible way of knowing which servers are in an active view (the **activeServers**()

function) then the above simple scheme is sufficient: DQ-RPC sends its messages to servers in an active view and it makes sure that it only picks active views as its current view[5].

Showing how DQ-RPC can determine which views are active is the subject of the rest of this section.

View changes

To determine whether a view is active, it is important to specify how the system starts (and ends) views.

To initiate a view change, the administrator's computer first tells a quorum of machines in the old view that their view has ended. These machines immediately stop accepting client requests. Clients can thus no longer read from the old view since they will not be able to gather a quorum of responses; they will continue retransmitting until they get answers from the new view). The administrator then performs a user-level read on the machines from the old view to obtain some value v. Finally, the administrator tells all the machines in the new view that the new view is starting, and provides them with the initial value v. At this point, the machines in the new view start accepting client requests.

Naturally, it is not always possible for the administrator to make sure it has contacted all the new machines: if some server is faulty then it could choose not to acknowledge, causing the administrator to block forever. In our simplified DQ-RPC we remove this problem by simply assuming that the administrator has some way to contact all the servers. We will see in Section 5.5.3 how the full DQ-RPC ensures that all view changes terminate.

[5]It is necessary to pick an active view: after some DQ-RPC writes data to the latest view, reads to a view that has ended would return old data since different views may have no servers in common.

A delicate point to consider when performing a view change is that, after view t ends, so does the constraint that at most $f(t)$ of the machines in view t can be faulty. For example, if the view was changed to remove some decommissioned servers, it is natural to expect that the semantics of the system from then on does not depend on the behavior of the decommissioned servers.

And yet, the decommissioned machines know something about the previous state of the system. If they all became faulty (as it may happen, since they are no longer under the administrator's watchful eye) they would be able to respond to queries from clients that are not yet aware of the new servers and fool them into accepting stale data, violating atomic semantics. An analogous situation is depicted in Figure 5.4: horizontal lines represent servers, and time flows to the right. The Figure shows a view change where servers are added and f is increased to 4, the size of the original view. If the servers in the original view all fail, they could fool clients into believing that the view change never took place. To prevent faulty servers from being able to fool clients into accessing a view that has ended,the view change protocol must ensure that no client can read or write to a view after that view has ended. Our *forgetting* protocol implements this property.

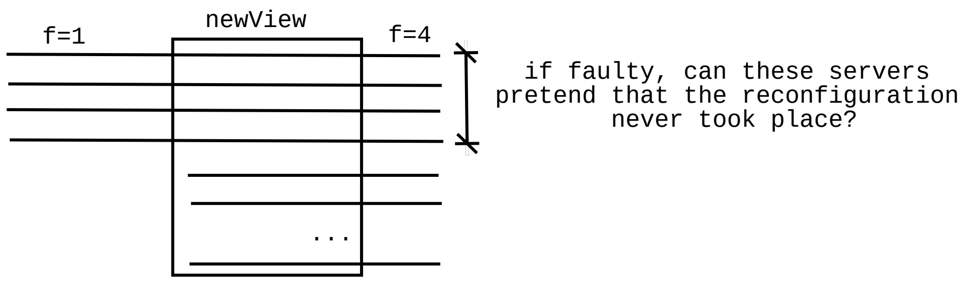
f=1
newView
f=4
if faulty, can these servers pretend that the reconfiguration never took place?
...

Figure 5.4: Example of view change

Safe View Certification through "Forgetting" The simplified DQ-RPC requires the client to receive a quorum of responses with view t's tag before it returns that value and considers view t current. If the servers are correct, then this ensures that no DQ-RPC chooses t as current after t ends (recall that views end once a quorum of their servers have left the view).

The forgetting protocol ensures that this property holds despite Byzantine failure of the servers. The key insight behind the forgetting protocol is that the bound $f(t)$ still holds on view t when the forgetting protocol is run, so all then-correct servers will correctly forget as requested. Clients tag their queries with a nonce e. Server i tags its response with two pieces of information: 1) server i's view certificate $\langle i, meta, pub\rangle_{admin}$, signed by the administrator and 2) a signature for the nonce $\langle e\rangle_{priv}$, proving that server i possesses the private key associated with the public key in the view certificate. The key pair $pub, priv$ is picked by the administrator. In the certificate, $meta$ contains the meta information for the view, namely the view number t, the set of servers N and the resilience threshold f. The quorum size q can be computed from these parameters.

When servers leave view t, they discard the view certificate and private key that they associated with that view. The challenge is to ensure that even if they become faulty later, they cannot recover that private key and thus cannot vouch for a view that they left. We now discuss how our protocol addresses this issue.

The private key is only transmitted when the administrator informs the server of the new view. Our network model allows the channel to duplicate and delay this message, which may therefore be received after the server has left the view. To prevent a decommissioned server that was correct when leaving view t from recovering the private key we encrypt the message using a secret key that changes for every view.

The administrator's view change message for view t to server i has the following form:

$$(\text{NEW\_VIEW}, t, \mathit{oldN},\ \mathit{encrypt}\ \big((\langle i, \mathit{meta}, \mathit{pub}\rangle_{\mathit{admin}}, \mathit{priv}), k_i^t)\big)$$

We use the notation $\mathit{encrypt}(x, k)$ for the result of encrypting data x using the secret key k. The view secret key k_i^t is shared by the administrator and server i for view t. It is computed from the previous view's key using a one-way hash function: $k_i^t := h(k_i^{t-1})$. The administrator and server i are given k_i^0 at system initialization.

When correct servers leave a view t, they discard view t's certificate, private key *priv* and view key k_i^t. As a result they will be unable to vouch for view t later even if they become faulty and gather information from duplicated network messages. This ensures that clients following the simplified DQ-RPC protocol will not pick view t as its current view after t ends.

Finding the current view

In the previous section we have seen how clients can identify old views. We now need to make sure that the clients will be able to find the current view, too.

If the set of servers that the client contacts to perform its DQ-RPC intersects with the current view in one correct server i, then the client will receive up to date view information from i and will be able to find the current view.

If that is not the case, then the client can consult all of the servers in U, looking for a list of the servers in the current view that is published by the administrator (of course, the client would start from some well-known sites in U that have been assigned this task). These sites provide hints, in the sense that they cannot induce a client in error because our certified tags ensure safety: even if the information the client retrieves from one of these sites is obsolete, the client will never pick as current a view that has ended. Therefore it suffices that the client eventually learn of an active view from one of the sites in U.

In the case of a local network, clients could also broadcast a query to find the servers currently in N. This solution has the advantage of simplicity but it can be expensive if the servers are not all in the same subnet.

Summary

Clients only accept responses if they all have valid tags for the same view. Until they accept a response, clients keep re-sending their request (for read or write) to the servers. Clients use the information in the tags to locate the most recent servers, and periodically check well-known servers if the servers do not respond or do not have valid tags. Tags are valid if their view certificate has a valid signature from the administrator and the tag includes a signature of the client-supplied nonce that matches the public key in the certificate.

Replacing Q-RPC with this simplified DQ-RPC in a dissemination quorum protocol from Figure 5.1 results in a dynamic protocol that maintains all the properties listed in the figure.

However, simplified DQ-RPC has two significant limitations. First, it requires the administrator's **newView**(...) command to wait for a reply from all the servers in the new view, which may never happen if some servers in the new view are faulty. Second, it does not let DQ-RPCs (and, implicitly, user-level read and write operations issued by clients) complete during a view change: instead the operations are delayed until the view change has completed. We address both limitations in the next section.

5.5.3 The Full DQ-RPC for Dissemination Quorums

```
DQ-RPC($msg$) :
1. Sender $sender$ := new Sender($msg$)
2. static ViewTracker $g\_vt$ := new ViewTracker()
3. repeat :
4.     $sender$.sendTo($g\_vt$.get().$N$)
5.     $(Q, t)$ := $g\_vt$.consistentQuorum($sender$.getReplies())
6.     if running for too long : $g\_vt$.consult()
7. until $Q \neq \emptyset$
   // t is the current view associated with this operation
8. return $Q$          // sender stops sending at this point
```

Figure 5.5: Dynamic quorum RPC

The full DQ-RPC for dissemination quorums follows the same pattern as its simplified version: it sends the message repeatedly until it gets a consistent set of answers, and it picks a current view in addition to returning the quorum of responses. DQ-RPC uses the technique described in the previous section to determine whom to send to, but it can decide on a response sooner than the simplified DQ-RPC because it can identify consistent answers without requiring all the responses to be tagged with the same view. The full DQ-RPC also runs a different view change protocol that terminates despite faulty servers.

We split the implementation of DQ-RPC into three parts. The main DQ-RPC body (Figure 5.5) takes a message and sends it repeatedly (lines 3–7) to the servers believed to constitute the current view. The client's current view changes with the responses that it gets; if no responses are received for a while, then DQ-RPC consults well-known sources for a list of possible servers (line 6). The repetitive sending is handled by the *sender* object, and the determination of the current view is done by the **ViewTracker** instance (Figure 5.9). The client exits when it receives a quorum of consistent answers. In the simplified protocol, answers were consistent if they all had the same tag. In this section we develop a more efficient notion of consistent responses.

The *sender* is given a message and a destination and it repeatedly sends the message to the destination. The destination can be changed using the **sendTo**(...) method and the replies are accessed through **getReplies**(...) (the code for the **Sender** class is shown in Figure 5.6).

The **ViewTracker** class (Figure 5.9) acts like a filter: The client only reads the replies from *sender* that **ViewTracker** considers consistent (Figure 5.5, line 5). The **ViewTracker** looks at the messages and keeps track of the most recent view certificate it sees. As we saw in the forgetting protocol, messages are tagged with a signed view certificate and a signed nonce. Messages that do not have a correct signature for the nonce are not considered as vouching for the view (line 3 of **ViewTracker**.**consistentQuorum**(...)). However, even if the nonce signature is invalid, **ViewTracker** will use valid view certificates to learn which servers are part of the latest view (line 5 of **ViewTracker**.**receive**(...)). The most recent view certificate can be accessed through the **get**() method. The **ViewTracker** can also get new candidates from U with the **consult**(...) method. Finally, the **ViewTracker** has the responsibility of deciding when a set of answers is consistent through the **consistentQuorum**(...) method.

Sender variables

m_message	the message that is to be transmitted
m_destinations	the set of destination addresses
m_thread	a resend thread (initially not running)
m_replies	set of (sender, reply, meta) triples
m_recentReply	set of senders who sent a reply in the most recent view
m_delay	delay until the next retransmission (initially 1 second)

```
constructor Sender(msg) :
1.  m_message := msg

Sender.sendTo(Dests) :
1.  m_destinations := Dests
2.  if m_thread is not running :
3.      create m_thread          // the worker thread m_thread will call run()

Sender.run() :
1.  m_message.nonce = a new nonce
2.  while (true) :
3.      m_recentReply = senders in m_replies with meta that has the same
            view number as g_vt.get()
4.      asynchronously send m_message to each element of
            m_destinations - m_recentReply
5.      wait m_delay seconds
6.      m_delay := m_delay * 2
7.      while ((j,r,s):=g_vt.receive(m_message.nonce))
            // received valid reply (r,s) from server j
8.          if $j \in$ m_destinations :
9.              m_replies := m_replies $\cup$ $(j, r, s)$

Sender.getReplies() :
1.  return m_replies
```

Figure 5.6: **Sender** class for dynamic quorum RPC

Introducing generations

Our dynamic protocols only require the minimal number of servers to tolerate f faults: $3f + 1$ (as established in Chapter 3). The price for this minimal replication is that every time new servers are added, the data must be copied to them.

When more machines are available, it is possible to use the additional replicas to speed up view changes. We offer this capability by introducing a new parameter, *spread*. When the spread parameter m is non-zero, quorum operations involve more servers than strictly necessary. This margin still allows quorums to intersect when a few new servers are added, allowing these view changes to proceed quickly. As a result, there are now two different kinds of view changes: one in which data must be copied, and one in which no copy is necessary. In the second case, we say that the old and new views belong to the same *generation*. Each view is tagged with a generation number g that is incremented at each generation change.

These two parameters, m and g, are stored in the view meta-data alongside with N, f and t.

The additional servers do not necessarily need to be used to speed up view changes. Using a smaller m with a given n makes the quorums smaller and reduces the load on the system. The parameter m therefore allows the administrator to trade-off low load against quick view changes.

Intra-Generation: When Quorums Still Intersect When clients write using the DQ-RPC operation, their message is received by a quorum of responsive servers. The size of the quorum depends on the parameters of the current view t (recall that t is also determined in the course of a DQ-RPC). The quorum size depends on the failure assumptions made by the protocol. For a U-dissemination Byzantine protocol that tolerates b faulty servers, the quorum size is $q(n, b, m) = \lceil (n + b + 1)/2 + m/4 \rceil$.

In the absence of view changes, our quorums intersect in $b+1+m/2$ servers. If m new (blank) servers are added to the system, then our quorums intersect in $b+1$ servers ($q(n, b, m)+q(n+m, b, m)-(n+m) = b+1$), which is still sufficient for correctness: one of the servers is correct and the reader will recognize the signature on the correct data. Thus, up to m servers can be added to the system before data must be copied to any of the new servers.

Similarly, if m of the servers that were part of a write quorum are removed, new quorums will still intersect in $b+1$ servers and the system will behave correctly: $q(n, b, 0) + q(n - m, b, m) - n = n + b + 1 + m/4 - m/2 + m/4 - n = b + 1)$ Finally, if b is increased or reduced by up to m (causing the quorums to grow or shrink accordingly), new quorums will still intersect the old ones in $b + 1$ servers: $q(n, b, m) + q(n, b + m, m) - n = n + b + 1 + m/2 + 2m/4 - n = b + 1 + m$ and $q(n, b, m)+q(n, b-m, m)-n = n+b+1-m/2+2m/4-n = b+1$. More generally, if after a write a servers are added, d servers are removed, b is changed to b', and m is changed to m', then the quorums will still intersect in a correct servers as long as $a+d+c \leq (m+m')/2$, where $c = |b'-b|$. To establish this inequality, we compute the intersection: $q(n, b, m)+q(n+a-d, b', m')-(n+a) = (n+b+1)/2+m/4+(n+a-d+b')/2+m'/4-(n+a) = (b+b')/2+1+(m+m')/4-(a+d)/2$. If $a+d \leq (m+m')/2-c$, then the intersection has size at least $(b + b')/2 + 1 + |b - b'|/2 = max(b, b') + 1$.

If a view change were to break this inequality, then the value must be copied to some of the new servers before the view change completes: we say that the old and new views are in different generations.

View changes: closing the generation gap

The copying of data across generations is done as part of the view change protocol. Unlike the view change protocol that is associated with simplified DQ-RPC, the full view change protocol terminates.

Recall, view changes are initiated by the administrator when some machines need to be added, removed, or moved, or when the resilience f or the spread m have to be changed. The **newView**(...) method first determines whether the new view will be in the same generation as the previous one, using the relation in Section 5.5.3. It then computes the key pairs and certificates for the new view. Finally the administrator encodes the certificates using the appropriate secret key and sends them to all servers in t, re-sending when appropriate and waiting for a quorum of responses.

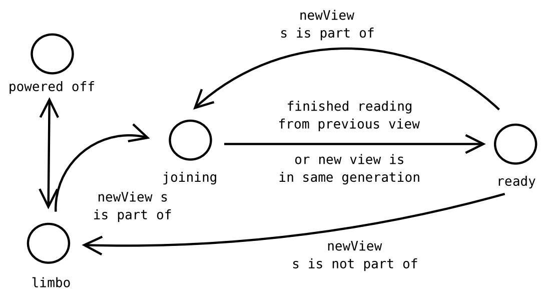

Figure 5.7: Server s, transitions for the dissemination protocol

Every server switches state according to the diagram in Figure 5.7. Figure 5.8 shows the matching pseudocode for servers in dissemination quorums. What server s does when receiving a “NEWVIEW” message depends on whether the new view belongs to the same generation as s’s current view. In case of generation change (and if s belongs to the new view), s piggybacks that message on top of a read it performs on a quorum from the old view. Server s then updates its value to what it

```
**mainLoop**() :
1.  receive $message$ from machine $i$
2.  $response$ := result of calling the non-private function with the same name
    as the first element of $message$, if any ($\perp$ otherwise).
3.  reply to $i$ with $(m\_viewCert, \langle message.nonce\rangle_{m\_priv}, response)$

write($ts$,$D$) :
1.  if ($m\_ts < ts$) then ($m\_ts$,$m\_D$) := ($ts$,$D$)
2.  return "WRITE-ACK"

read() :
1.  return ($m\_ts$, $m\_D$)

newView($t$, $oldN$, $encryptedBody$) :
    // encryptedBody is of the form encrypt $((\langle i, meta, pub\rangle_{admin}, priv), k_i^t)$
1.  $newK := h^{t-m\_meta.t}(m\_k)$
2.  $(cert, priv) := decrypt(encryptedBody, newK)$
3.  if ($cert.meta.N$ does not include this server) :
4.      // limbo
5.      $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$
6.      return "OK"
7.  if ($cert.meta.g == m\_cert.meta.g$) :
8.      // intra-generation view change (ready state)
9.      $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$
10.     return "OK"
11. // inter-generation view change (joining state)
12. $(newTS, newD) := \phi$(Q-RPC("READ+NEWVIEW", $cert$)) to the servers in $oldN$
13. if $m\_ts < newTS$ : ($m\_ts$,$m\_D$) := ($ts$,$D$)
14. $(m\_cert, m\_priv, m\_k) := (cert, priv, newK)$ // ready state now
15. return "OK"

read+NewView($cert$) :
1.  if ($m\_cert.meta.t < cert.meta.t \wedge cert$ has a valid signature) :
2.      $(m\_cert, m\_priv) := (cert, \perp)$
3.  return ($m\_ts$, $m\_D$)
```

Figure 5.8: Server protocol for dissemination quorums

read (if it is newer than the value s currently stores) and updates its view certificate. If s is part of the new view but there is no generation change, then the s just updates its view information as per the forgetting protocol. If s is not part of the new view, then it updates its view certificate, too. In that case s will not be able to vouch for the new view, since it has no valid view certificate for it, but it will still be able to direct clients to the current servers.

Servers are in the *limbo* state initially and after leaving the view. They are in the *joining* state while they copy information from the older view, and they are in the *ready* state otherwise. Servers process client requests in all three states. Servers in the *joining* state use the view certificate for the old view (if they have it) until they are *ready*.

The administrator's **newView**(...) waits for a quorum of new servers to acknowledge the view change and then it posts the new view to at least one server in U and returns. At this point, the administrator knows that the data stored in the machines that were removed from the view are not needed anymore and therefore the old machines can be powered off safely.

There may still be some machines in the *joining* stage at this point. These machines do not prevent operations from completing because DQ-RPC operations only need $f + 1$ servers in the new generation to complete, and any dissemination quorum contains at least $f + 1$ correct servers.

When **newView**(...) returns, the old view has ended and the new view has started and *matured*, meaning that at least one correct server is done processing the view change message for it. This means that reads and writes to the new view will succeed and reads and writes to the old view will be redirected to the new view (either by the old servers or after consultation of the well-known locations).

```
(meta) ViewTracker.get() :
     // returns the latest view meta-data
1.   return $m\_maxMeta$

ViewTracker.consult() :
     // ask well-known servers for the latest meta-data
1.   Choose a server $j$ at random from $U$, biased to favor
     the well-known view publishers
2.   send (“CONSULT”, $m\_maxMeta$) to $j$

ViewTracker.receive(nonce) :
     // used by the Sender object when gathering replies
1.   if there is no message waiting : return false
2.   receive ($msg, meta$) from $sender$
3.   if not validCertificate($meta$) : goto 1 // faulty server
4.   if $meta.t > m\_maxMeta.t$ :
5.        $m\_maxMeta := meta$
6.   if not validTag($meta, nonce$) : goto 1 // faulty or limbo server
7.   if $msg ==$ “CONSULT-ACK” : goto 1
8.   return ($sender, msg, meta$)

(messages, view) ViewTracker.consistentQuorum(messageTriples) :
     // returns a consistent quorum of messages (if any) and the current view
1.   $msgInQuorun := \{m \in messageTriples : m.sender \in m\_maxMeta.N\}$
2.   if $|msgInQuorun| < q(|m\_maxMeta.N|, m\_maxMeta.f, m\_maxMeta.m)$ :
3.        return $(\emptyset, \perp)$ // fail if there is no consistent quorum of messages
4.   $recentMessages := \{m \in msgInQuorum : m.meta.g == m\_maxMeta.g\}$
5.   if $|recentMessages| < m\_maxMeta.f + 1$ : return $(\emptyset, \perp)$
     // fail if the view is not mature
6.   return ($msgInQuorun, m\_maxMeta$)
```

Figure 5.9: **ViewTracker** class

The protocol as presented here requires the administrator to be correct. If the administrator crashes after sending the new view message to a single faulty new server, the new server can cause the servers in the old view to join the limbo state without informing the new servers that they are supposed to start serving. In

Section A.2 we show a variant that tolerates crashes in the sense that if the administrator machine crashes at any point during the view change and never recovers then read and write operations will still succeed even though it is not possible to change views anymore. If the administrator crashes and then recovers, then view changes can proceed normally. The basic idea is that the servers in the old view make sure that a quorum of servers in the new view is informed of the view change before they let the old view end.

DQ-RPC satisfies transquorums for dissemination quorums

We can now prove that DQ-RPC satisfies the transquorum properties, so Dynamic U-Dissemination provides atomic semantics.

Lemma 30. *The view t chosen by a DQ-RPC operation is concurrent with the DQ-RPC operation.*

Proof. The view being concurrent means that it started and has not ended. We show each in turn. The view t is chosen in ViewTracker's **findConsistentQuorum**(...) method, which takes as input messages returned by ViewTracker's **receive**(...) method (Figure 5.9). The **receive**(...) method only returns messages that have a valid certificate signed by the administrator (**receive**(...), line 3). These signatures cannot be faked. If view t is chosen, then a message was received that bore a valid certificate for view t, signed by the administrator. By definition, then, view t has started.

If view t has ended, then the correct servers in a quorum Q_0 of $q(N(f), f(t), m(t))$ servers have processed the view change message and discarded the view meta-information associated with view t. The forgetting protocol ensures that the servers in Q_0 that discarded the meta-information are not able to recover it. By quorum intersection[6], the quorum of $q(t)$ replies in t selected by DQ-RPC to vouch for view t intersects the quorum Q_0 in at least one server s that was correct in t. Only messages with a valid tag are accepted (**receive**(...), line 6, Figure 5.9) and a server can only generate a valid tag for view t if it has the meta-information for view t. If DQ-RPC chooses view t, then s must have been able to generate a valid tag. Therefore, view t has not ended. □

Lemma 31. *All reads succeed. That is, there is no DQ-RPC$_{\mathcal{T}}$ or DQ-RPC$_{\mathcal{R}}$ operation x such that $\phi(x) = \perp$.*

Proof. The $\phi(Q)$ function returns the largest valid element of Q. DQ-RPC replies contain at least $f + 1$ responses tagged with the latest generation (line 5 of **consistentQuorum**(...), Figure 5.9). Servers always forward the highest-timestamped data they have received (**write**(...), line 1, Figure 5.8), so all DQ-RPC replies contains at least one non-$\perp$ reply from a correct server that can be chosen by ϕ. □

We now show that DQ-RPC provides the transquorum properties.

Lemma 32. *All $\mathcal{T}$ operations in the dissemination DQ-RPC are timely.*

Proof. Lemma 31 shows that for any DQ-RPC$_{\mathcal{R}}$ operation r, $\phi(r) \neq \perp$. Since $o(r) = \phi(r)$ (Figure 5.2), it follows directly that $o(r) \neq \perp$. So, according to the definition of timeliness, we must prove: $\forall w \in \mathcal{W}, \forall r \in \mathcal{R} \cup \mathcal{T} \;:\; w \rightarrow r \implies o(w) \leq o(r)$. Our proof proceeds by case analysis on the views associated with operations r and w. The three possible cases are: (i) w and r belong to the same generation, (ii) w belongs to the generation preceding r's, or (iii) w belongs to a generation older than r's. It is not possible for w to belong to a generation that comes after r if $w \rightarrow r$:

[6]It is safe to use quorum intersection here, since Q_0 and $q(t)$ are quorums of the same view t.

DQ-RPCs decide on a view whose generation is vouched for by at least $f+1$ servers with data (Figure 5.9, **consistentQuorum**(...), line 5). That means that one of these servers is correct. Correct servers only install a generation after all previous generation have ended (Figure 5.8, **newView**(...), line 14), so r cannot pick an earlier generation.

If w and r picked views that are in the same generation then the two quorums intersect in at least one correct server. Since $w \rightarrow r$, servers never decrease the timestamp they store, and the first element of $o(r)$ is at least the timestamp of the data written by w (for both $\mathcal{R}$ and $\mathcal{T}$ operations), it follows that $o(w) \leq o(r)$.

If w picked a view t that is in the generation that immediately precedes the generation v to which r's view belongs, then we consider the last view u in t's generation. As shown in the previous paragraph, reads from a quorum $q(u)$ in u will result in a timestamp that is at least as large as $o(w)$. Such a read occurs when a server transitions to the ready state in v's generation, and all correct servers that enter v's generation therefore have a timestamp at least as recent as $o(w)$. The read r waits until it knows that view v is mature, so at least one correct server answered the DQ-RPC r after installing view v. Since that server is correct, its reply is valid. Since that reply has a timestamp at least as recent as $o(r)$ and dissemination quorums pick the largest valid reply, it follows that $o(w) \leq o(r)$.

If w picked a view that happens several generations before r then we can use induction, using the previous paragraph's reasoning several times to show that $o(w) \leq o(r)$.

This covers all the possible ordering for generations, so $\forall w \in \mathcal{W}, \forall r \in \mathcal{T}$: $w \rightarrow r \Longrightarrow o(w) \leq o(r)$. □

Lemma 33. *All $\mathcal{R}$ operations in the dissemination DQ-RPC are sound.*

Proof. No dissemination operation returns $\perp$ (Lemma 31), so we show $\forall r \in \mathcal{R} : \exists w \in \mathcal{W}$ s.t. $r \not\rightarrow w \wedge o(w) = o(r)$. Since $o(r)$ picks the largest valid value and digital signatures cannot be faked, $o(r)$ can only return a value that was written previously, so one for which a w operation exists. Since the value was written previously, the write must happen before the read or concurrently with it. □

Lemma 34. *The DQ-RPC protocol in Figure 5.5 provides the transquorum properties for the ordering function o of Figure 5.2.*

Proof. This lemma follows directly from Lemmas 32 and 33. □

Theorem 9. *U-dissemination based on DQ-RPC provide atomic semantics.*

Proof. This follows directly from the fact that U-dissemination is atomic if the transquorum properties hold (Theorem transquorumimpliesatomic-redux) and that DQ-RPC for dissemination quorums provides the transquorum properties (Lemma 34). □

5.5.4 Optimizations

Single-Roundtrip Reads

Both the U-dissemination and U-masking protocols (and their hybrid counterparts) can be sped up by skipping the writeback (Figure 5.1, **read**(), line 3) in the case of *unanimous reads*, i.e. reads in which responses in the quorum agree. This idea is not new but it is interesting since it leads to single-roundtrip reads in the common case where no operation is concurrent with the read.

For single-roundtrip reads the TRANS-Q$_{\mathcal{R}}$ operations must have the property that $\forall r_1, r_2 \in \mathcal{R}$: unanimous$(r_1) \Rightarrow o(r_1) \leq o(r_2)$. This property holds for quorum intersection (since an unanimous read means that the service state is similar to what it would be after writing that value), and it also holds for both our dissemination and masking DQ-RPC implementations.

Reducing Transmissions

The protocol, as described in Section 5.5.2, piggybacks view information onto each message sent by the servers. Also, clients verify all of these messages. Since in most cases the view will be the same as it was in the previous exchange, optimizations can be used to decrease both the amount of data that needs to be transmitted and the computations necessary to verify that information.

Servers send the view information along with the administrator certificate and signed nonce. To optimize for the common case while retaining the forgetting property described in Section 5.5.2, servers could omit the view information and instead just send the view number t. If the client knows about that view then it has all the necessary data to verify the signature. If it does not, then the client sends a request to the server to retrieve the complete view information.

Another opportunity arises in the choice of quorums. Instead of using the same quorums for the view's meta-information as for the data, we can use quorums of different size for read and write operations (we call them *asymmetric quorums*). This is beneficial because view meta-data is read more often than it is written. These asymmetric quorums use the smallest possible read quorums ($2f + 1$ since the view meta-information is self-verifying) and the largest possible write quorums ($n - f$). The current approach in ViewTracker is to verify that (1) there is a dissemination (resp. masking) quorum of responses such that no member claims that the current view has ended and (2) enough responses are in the same generation as the current view. The first point can be changed to (1) there is a read quorum of responses such that no member claims that the current view has ended. Reads naturally still need to gather a dissemination (resp. masking) quorum of responses in order to read the variable, but with this change it is not necessary anymore that all the responses be tagged with view information. The client can then indicate in its queries whether the server should include a view certificate in its reply.

Another natural optimization is that in the few cases where servers still need to send the view meta-information to clients, the servers can send the difference between their information and the one the client knows instead of sending the whole thing.

Proactive Recovery

Proactive recovery is a technique in which machines are periodically refreshed to a known good state. This reduces the number of machines that are faulty at the same time and thus reduces the risk that the system will fail because more than f servers are faulty.

Proactive recovery requires us to remove a server, refresh it (for example by rebooting it), and then bring it back in the system under a different name. Our dynamic quorums are particularly well suited for proactive recovery because we can add and remove servers with little overhead, and client operations can complete even if they span several different views.

Faster Reads and Writes

It is possible to speed up quorum operations by slightly weakening the conditions under which the DQ-RPC function returns. The correctness of the protocol only requires that DQ-RPC selects as its current view t one that is concurrent with

```
$(messages, view)$ **ViewTracker.consistentQuorum**$(messageTriples)$ :
1. find $qrm \subseteq messageTriples$ such that
2.     let $mt :=$ largest-timestamped element of $qrm$
3.     $\forall m \in qrm : m.sender \in mt.meta.N$, and
4.     $|qrm| = q(|mt.meta.N|, mt.meta.f, mt.meta.m)$,
5.     and $|\{m \in qrm : \textbf{validTag}(m) \wedge m.meta.g = mt.meta.g\}| \geq mt.meta.f + 1$
6. **if** no such $qrm$ exists : **return** $(\emptyset, \perp)$
7. **return** $(qrm, mt.meta)$
```

Figure 5.10: Optimized **consistentQuorum**(...)

the DQ-RPC. The DQ-RPC we show in this chapter always picks the most recent concurrent view that is knows of. This sometimes causes DQ-RPC to wait for messages unnecessarily. Consider the case where $q + 1$ responses are received. The first response is in view $t + 1$ and all the others are in view t. In that situation it would be perfectly reasonable to pick the last q responses as the result of the DQ-RPC operation, but our simplified operation will wait until it gets a q responses in view $t + 1$.

The change impacts **ViewTracker** (Figure 5.9). Instead of keeping track of the most recent view certificate it sees (*m_maxS*), ViewTracker must now inspect each set of responses to see if there exists some view that can be considered current. **consistentQuorum**(...) is replaced with the code of Figure 5.10.

This code is slightly harder to read than the original, but it still picks a current view that is concurrent with the DQ-RPC and it allows DQ-RPC to complete sooner in the case outlined above. The termination condition here is strictly weaker than before, so there is no situation where this DQ-RPC would be slower than the original one.

The **get**() method should also be modified to include more servers than just the last view, for example the union of the two most recent views (safety and liveness hold as long as servers from the last view are included, but allowing more servers increases the chances of speeding up a read or write operation).

Faster Generation Changes

Our protocols' ability to add servers when necessary relies on the fact that the data will be copied to the new servers. The DQ-RPC and view change protocols make sure that the protocol semantics are maintained despite the copying. However, copying data takes time. There are some cases where we can speed up generation changes.

Consider first the case where some servers of the new view are also part of the old view. It would be unwise for them to just keep whatever data they have, for the data they are storing could be untimely and the new view may require them to hold timely data (for example if the quorum size changes). In most cases, however, the data on the server is timely and we can avoid the copy by using *conditional reads*. In a conditional read, the server issuing the read indicates the timestamp of the data that it has. If the respondent does not have data that is newer than the indicated timestamp then it sends a response with empty data (but the timestamp and view certificates are still included when appropriate). If the respondent has newer data then it sends it as usual. As a result, servers that are already timely do not need to transfer the data across the network and they can join the new view much more quickly.

Conditional reads yield their full power when used in combination with our second optimization. Recall that the spread parameter allows new servers to join the system without having to receive a copy of the data first (intra-generation view changes). These servers normally participate in the protocol and refresh their data in the next generation change. We can choose, instead, to have the new (blank) servers read the current value of the data in the background (perhaps using TCP Nice [155]). When it comes time for the generation change, the servers can use conditional reads: if they already have the right data then they can move to the new view instantly.

5.6 Conclusions

We present a methodology that transforms several existing Byzantine protocols for static quorum systems [61, 100, 101, 111, 126] into corresponding protocols that operate correctly when the administrator is allowed to add or remove servers from the quorum system, as well as to change its resilience threshold. Performing the transformation does not require extensive changes to the protocols: all that is required is to replace calls to the Q-RPC primitive used in static protocols with calls to DQ-RPC, a new primitive that in the static case behaves like Q-RPC but can handle operations across quorums that may not intersect while still guaranteeing consistency. Our methodology is based on a novel approach for proving the correctness of Byzantine quorum protocols: through our transquorum properties, we specify the characteristics of quorum-level primitives (such as Q-RPC) that are crucial to the correctness of Byzantine quorum protocols and proceed to show that it is possible to design primitives, such as DQ-RPC, that implement these properties even when quorums don't intersect. We hope that designers of new quorum protocols will be able to leverage this insight to easily make their own protocols dynamic.

Chapter 6

Separating Agreement from Execution for State Machine Replication

6.1 Introduction

In previous chapters we focused on quorums and registers and saw how useful they can be. However, there are limits to what can be achieved with quorums and registers when clients can fail. Consider for example the simple program below which attempts to use a shared register R to implement a counter.

```
1.    increment() {
2.        $x := R$.read()
3.        $R$.write( $x + 1$ )
4.    }
```

Figure 6.1: An illustration of the limitations of registers

There are situations were **increment**() does not behave as one might expect. Specifically, if the initial value of the register is 0 and two users run **increment**() concurrently, then one would expect the final value of R to be 2. Unfortunately it is possible for the final value to be 1 (if the second client runs **increment**() to completion in between lines 2 and 3 from the first client). Although read and write operations on registers are atomic, sequences of these operations are not—so the two clients' calls to **increment**() can be interleaved, causing unintended results.

In this chapter we turn our attention to a more powerful primitive, the replicated state machine [82, 84, 139], a universal construction that can make any operation (including **increment**()) atomic and fault-tolerant.

Recent work has demonstrated that Byzantine fault-tolerant (BFT) state machine replication is a promising technique for using redundancy to improve integrity and availability [127], and that it is practical in that it adds modest latency [31], can proactively recover from faults [32], and can make use of existing software diversity to exploit opportunistic n-version programming [130].

Unfortunately, two barriers limit widespread use of BFT state machine replication to improve reliability. First, asynchronous BFT replicated state machines require more than three-fold replication to work correctly (see [27] for the proof in the case of a fair scheduler and [49] for the proof in the case of eventual synchrony); tolerating just a single faulty replica requires at least four replicas. Even with opportunistic n-version programming and falling hardware costs, this replication cost is significant, so reducing the replication required would significantly increase the practicality of BFT replication.

Second, even though BFT state machine replication allows correct clients to access the service despite some failures (thus improving reliability), it also makes it easier for an unauthorized client to access the service, hurting confidentiality. In particular, although either increasing the number of replicas or making the implementations of replicas more diverse reduces the chance that an attacker can compromise enough replicas to bring down a service, each replica also increases the chance that at least one replica has an exploitable bug—and in state machine replication all data are stored on all replicas so an attacker need only compromise the weakest replica in order to endanger the confidentiality of all data stored on the service.

In this chapter, we explore a new BFT replication architecture that addresses both limitations. The key principle of this architecture is to separate agreement from execution. State machine replication relies on first agreeing on a linearizable order of all requests [82, 84, 139] and then executing these requests on all state machine replicas. Existing BFT state machine systems tightly couple these two functions, but cleanly separating agreement and execution yields three fundamental advantages.

First, our architecture reduces replication costs because the new architecture can tolerate faults in up to half of the state machine replicas responsible for executing requests. While our system still requires $3f+1$ *agreement replicas* to order requests using f-resilient Byzantine agreement, it only requires a simple majority of correct *execution replicas* to process the ordered requests. This distinction is crucial, because execution replicas are likely to be much more expensive than agreement replicas both in terms of hardware—because of increased processing, storage, and I/O—and, especially, in terms of software. By reducing the number of execution replicas, we reduce the hardware cost by reducing the number of times the same request is processed, the number of I/O requests, and the number of application state copies that must be kept. We also reduce the software and maintenance cost, because fewer versions of the application are needed. When n-version (or opportunistic n-

version) programming is used to eliminate common-mode failures across replicas, the agreement nodes are part of a generic library that may be reused across applications, further reducing software cost. Finally, since agreement nodes are simpler, it may be possible to verify their correctness formally so that only a single version is needed.

Second, decoupling agreement from execution leads to agreement replicas that are cheaper than execution replicas and can therefore be used more liberally. In this chapter, we show that additional replicas allow agreement to complete in two communication steps in the common case instead of three (we say the protocol is *two-step*). More precisely, we prove that the minimal number of agreement replicas that must be added for agreement to be two-step despite t actual failures is $2t$ (for a total of $3f + 2t + 1$). We show a new consensus protocol (FaB Paxos) that is optimal in that it uses the minimal number of agreement nodes to reach two-step agreement.

Third, separating agreement from execution leads to a practical and general *Privacy Firewall* architecture to protect confidentiality through Byzantine replication. In existing state machine replication architectures, a voter co-located with each client selects from among replica replies. Voters are co-located with clients to avoid introducing a new single point of failure when integrity and availability are goals [139], but when confidentiality is also a goal, existing architectures allow a malicious client to observe confidential information leaked by faulty servers. We do not assume servers that cannot be compromised so voting must take place, and we aim to ensure that clients can only see data they are allowed to see. In our system, therefore, we delegate the voting role to a redundant set of Privacy Firewall nodes. These nodes filter out incorrect replies before they reach the the agreement nodes, so that even compromised execution nodes (that have access to confidential information) cannot leak it to an attacking client. We restrict the physical communication links to prevent faulty execution nodes from bypassing the Privacy Firewall and communicating directly with a client.

A challenge of separating agreement and execution is that some communication formerly implicit because of the co-location of agreement and execution nodes must now be made explicit. Moreover, these messages can now be lost by unreliable links where, formerly, communication was reliable since source and destination were colocated, so special care needs to be taken for retransmissions. We face significant challenges to guaranteeing confidentiality with the Privacy Firewall because of nondeterminism in some applications and in the network, because an adversary can potentially influence nondeterministic outputs to achieve a covert channel for leaking information. For the applications we examine, agreement nodes can resolve application nondeterminism obliviously, without knowledge of the inputs or the application state, and therefore without leaking confidential information. However, more work is needed to determine how broadly this approach can be applied. For nondeterminism that stems from the asynchrony of our system model and unreliability of our network model (e.g., reply timing, message order, and retransmission counts), we show that our system provides *output symbol set confidentiality*, which is similar to possibilistic non-interference [114] from the programming languages literature. We also outline ways to restrict network nondeterminism, but future work is needed to understand the vulnerability of systems to attacks exploiting network nondeterminism and the effectiveness of techniques to reduce this vulnerability.

We prototype our Privacy Firewall to evaluate experimentally its performance. Overall, we find it competitive with previous systems [31, 32, 130] even though these systems offer no fault-tolerant privacy guarantee. For example with respect to latency, our system is 16% slower than BASE [130] for the modified Andrew-500 benchmark with the Privacy Firewall. The comparison is less clear-cut with respect to processing overhead and overall system cost, however. On one hand, our architecture allows reduction in the number of execution servers and the resources consumed to execute requests. On the other hand, when the Privacy Firewall is used, the system must pay for the extra firewall nodes and for a relatively expensive threshold signature operation (though the latter cost can be amortized across multiple replies by signing bundles containing multiple replies). Overall, applications with little processing and small aggregate load (thus small bundle size) make the relative overhead of the Privacy Firewall higher, and in the opposite situation the Privacy Firewall's relative overhead is smaller.

The main contribution of this chapter is to present the first study to apply systematically the principle of separation of agreement and execution to (1) reduce the replication cost, (2) reduce the number of communication steps for agreement in the common case, and (3) enhance confidentiality properties for general Byzantine replicated services. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, have paid unneeded replication costs, and have increased system vulnerability to confidentiality breaches.

In Section 6.2, we describe our system model and assumptions. Then, Section 6.3 describes how we separate agreement from execution, and Section 6.4 shows how to modify agreement so it completes in fewer communication steps in the common case. In Section 6.5, we determine the minimal number of communication steps for a given number of agreement nodes. We construct an optimal protocol matching this lower bound in Sections 6.6 and 6.7. We show in Section 6.8 how to build a replicated state machine with the two-step agreement cluster, and Section 6.9 presents performance optimizations. Section 6.10 describes the Privacy Firewall architecture. Section 6.11 describes and evaluates our Privacy Firewall prototype. Finally, Section 6.12 puts related work in perspective, and Section 6.13 summarizes our conclusions.

6.2 System Model and Assumptions

Consider a distributed asynchronous system [82]. Our protocols maintains the safety properties of the replicated state machine regardless of timing, crash failures, message omission, message reordering, and message alterations that do not subvert our cryptographic assumptions defined below. It is well known [54] that consensus cannot be solved in the asynchronous model with an adversarial scheduler. Therefore our system makes the relatively weak *bounded fair links* assumption for progress. Safety is maintained even if this assumption does not hold. Define a bounded fair links network as a network that provides two guarantees. First, it provides the *fair links* guarantee: a message sent infinitely often to a correct receiver is received infinitely often. Second, there exists some delay T such that if a message is retransmitted infinitely often to a correct receiver according to some schedule from time t_0, then the receiver receives the message at least once before time $t_0 + T$; note that the participants in the protocol need not know the value of T. This assumption appears reasonable in practice assuming that network partitions are eventually repaired.

We assume a Byzantine fault model for machines, and a strong adversary that can coordinate faulty nodes in arbitrary ways. However, we restrict this weak assumption about machine faults with two strong assumptions. First, we assume that some bound on the number of faulty servers is known; for example a given configuration might assume that at most f of n servers are faulty. Second, we assume that no machine can subvert the assumptions about cryptographic primitives described in the following paragraphs.

Our protocol assumes cryptographic primitives with several important properties. We assume a cryptographic *authentication certificate* that allows a subset containing k nodes out of a set $\mathcal{S}$ of nodes to operate on message X to produce an authentication certificate $\langle X \rangle_{\mathcal{S},\mathcal{D},k}$ that any node in some set of destination nodes $\mathcal{D}$ can regard as proof that k distinct nodes in $\mathcal{S}$ *said* [28] X. To provide a range of performance and privacy trade-offs, our protocol supports three alternative implementations of such authentication certificates that are conjectured to have the desired properties if a bound is assumed on the adversary's computational power: public key signatures [128], message authentication code (MAC) authenticators [33, 154], and threshold signatures [43].

In order to support the required cryptographic primitives, we assume that correct nodes know their private keys (under signature and threshold signature schemes) or shared secret keys (under MAC authenticator schemes) and that if a node is correct then no other node knows its private keys. Further, we assume that if both nodes sharing a secret key are correct, then no other node knows their shared secret key. We assume public keys are distributed so that all intended recipients of messages know the public keys needed to verify messages they receive. We make the standard cryptographic assumption that only the holder of the private key can sign a message in a way that the signatures matches the public key, that only the holder of the private key can decrypt a message encrypted with the public key, and that messages encrypted with a shared key can only be decrypted with that key.

Note that in practice, public key and threshold signatures are typically implemented by computing a cryptographic digest of a message and signing the digest, and MACs are implemented by computing a cryptographic digest of a message and a secret. We assume that a cryptographic digest of a message X produces a fixed-length collection of bits $D(X)$ such that it is computationally infeasible to generate a second message X' with the same digest $D(X') = D(X)$ (collision-resistance) and such that it is computationally infeasible to calculate X given $D(X)$ (D is one-way). Because digest values are of fixed length, it is possible for multiple messages to have the same digest value, but the length is typically chosen to be large enough to make this probability negligible over an execution of the protocol. Several existing digest functions such as SHA256 [142] are believed to have these properties, assuming a computationally bounded adversary.

To allow servers to buffer information regarding each client's most recent request, we assume a finite universe of *authorized clients* that send authenticated requests to the system. For signature-based authenticators, we assume that each server knows the public keys of all authorized clients. For MAC-based authenticators, we assume each client/server pair shares a secret key and that if both machines are correct, no other node knows the secret key. For simplicity, our description assumes that a correct client sends a request, waits for the reply, and sends its next request, but it is straightforward to extend the protocol to allow each client to have k outstanding requests. The system tolerates an arbitrary number of Byzantine-faulty clients in that non-faulty clients observe a consistent system state regardless of the actions of faulty clients. Note that a faulty client can issue disruptive requests that the system executes (in a consistent way). To limit such damage, applications typically implement access control algorithms that restrict which actions can be taken by which clients.

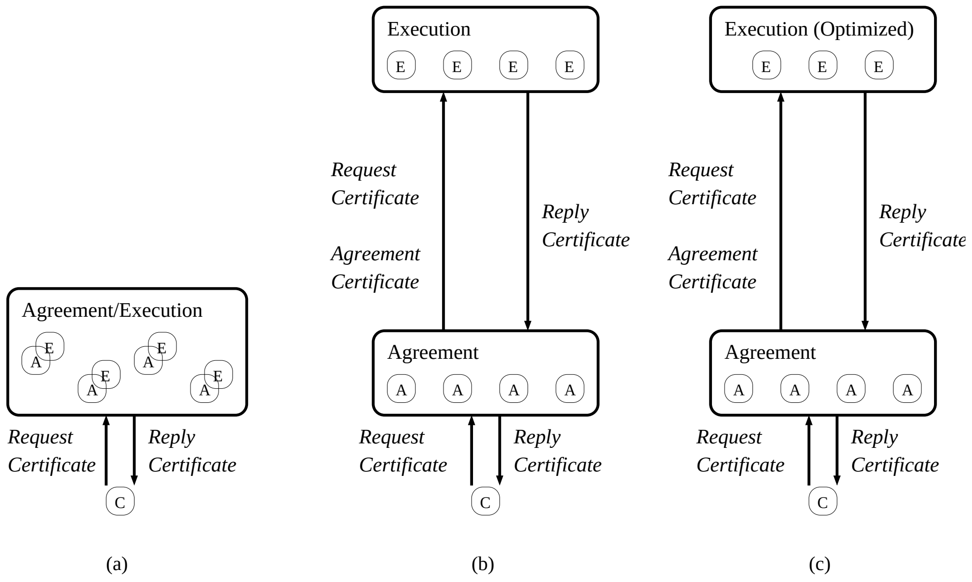

Figure 6.2: High level architecture of (a) traditional Byzantine fault-tolerant state machine replication, (b) separate Byzantine fault-tolerant agreement and execution, and (c) separate optimization of the execution cluster.

The basic system replicates applications that behave as *deterministic* state machines. Given a current state C of the state machine and an input I, all non-faulty copies of the state machine transition to the same subsequent state C'. We also assume that all correct state machines have a **checkpoint**() function that operates on the state machine's current state and produces sequence of bits B. State machines also have a **restore**(B) function that operates on a sequence of bits. If a correct machine executes **checkpoint**() to produce some value B, then any correct machine that executes **restore**(B) will then be in state C. We discuss how to abstract nondeterminism and minor differences across different state machine replicas [130] in Section 6.3.1.

6.3 Separating Agreement from Execution

Figure 6.2(a) illustrates a traditional Byzantine fault tolerant state machine architecture that combines agreement and execution [31, 32, 130]. In such systems, clients send authenticated requests to the $3f + 1$ servers in the system, where f is the maximum number of faults the system can tolerate. The servers then use a three-phase protocol to generate cryptographically verifiable proofs that assign unique sequence numbers to requests. Each server then executes requests in sequence-number order and sends replies to the clients. A set of $f + 1$ matching replies represents a *reply certificate* that a client can regard as proof that the request has been executed and that the reply is correct.

Figure 6.2(b) illustrates our new architecture that separates agreement and execution. The observation that enables this separation is that the agreement phase of the traditional architecture produces a cryptographically-verifiable proof of the ordering of a request. This *agreement certificate* can be verified by any server, so it is possible for execution nodes to be separate from agreement nodes.

Figure 6.2(c) illustrates the first of the three enhancements enabled by the separation of execution and agreement: it allows us to separately optimize the agreement and execution clusters (the agreement cluster is the set of all agreement nodes).

In particular, it takes a minimum of $3f + 1$ servers to reach agreement in an eventually synchronous system that can suffer f Byzantine faults [49]. But, once incoming requests have been ordered, a simple majority of correct servers suffices to mask Byzantine faults among execution servers—$2g+1$ replicas can tolerate g faults. Note that the agreement and execution servers can be separate physical machines, or they can share the same physical machines.

Reducing the number of state machine execution replicas needed to tolerate a given number of faults can reduce both software and hardware costs of Byzantine fault-tolerance: as we discuss in Section 6.1, fewer execution replicas means reduced hardware, software, and maintenance costs.

In the remainder of this section we explain in detail how we separate agreement from execution and reduce the number of execution nodes.

6.3.1 Inter-Cluster Protocol

We first provide a cluster-centric description of the protocol among the client, agreement cluster, and execution cluster. Here, we treat the agreement cluster and execution cluster as opaque units that can reliably take certain actions and save certain state. In Sections 6.3.2 and 6.3.3 we describe how individual nodes within these clusters act to ensure this behavior.

Client behavior

To issue a request, a client sends a *request certificate* to the agreement cluster. In our protocol, request certificates have the form $\langle\text{“REQUEST”}, o, t, c\rangle_{c,\mathcal{A},1}$ where o is the operation requested, t is the timestamp, and c is the client that issued the request; the message is certified by the client c to agreement cluster $\mathcal{A}$ and one client's certification is all that is needed.[1] A correct client issues monotonically increasing timestamps; if a faulty client's clock runs backwards, its own requests may be ignored, but no other damage is done.

After issuing a request, a client waits for a reply certificate certified by at least $g + 1$ execution servers. In our protocol a reply certificate has the form: $\langle\text{“REPLY”}, v, n, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ where v was the view number in the agreement cluster when it assigned a sequence number to the request, n is the request's sequence

[1]Note that our message formats and protocol closely follow Castro and Liskov's [31, 32].

number, t is the request's timestamp, c is the client's identity, r is the result of the requested operation, and $\mathcal{E}$ is the set of execution nodes of which at least $g+1$ must certify the message to c.

If after a timeout the client has not received the complete reply certificate, it retransmits the request to all agreement nodes. Castro and Liskov suggest two useful optimizations to reduce communication [33]. First, a client initially sends a request to the agreement server that was the primary during the view v of the most recent reply received; retransmissions go to all agreement servers. Second, a client's request can designate a specific agreement node to send the reply certificate or can contain the token *ALL*, indicating that all agreement servers should send. The client's first transmission designates a particular server, while retransmissions designate *ALL*.

Agreement cluster behavior

The agreement cluster's job is to order requests, send them to the execution cluster, and relay replies from the execution cluster to the client. The agreement cluster acts on two messages—the intra-cluster protocols discussed later will explain how to ensure that these actions are taken reliably.

First, when the agreement cluster receives a valid client request certificate $\langle \text{“REQUEST”}, o, t, c\rangle_{c,\mathcal{A},1}$ the cluster proceeds with three steps, the first of which is optional.

1. Optionally, check $cache_c$ for a cached reply certificate with the same client c and a timestamp that is at least as large as the request's timestamp t. If such a reply is cached, send it to the client and stop processing the request. $cache_c$ is an optional data structure that stores the reply certificate for the

most recent request by client c. $cache_c$ is a performance optimization only, required for neither safety nor liveness, and any $cache_c$ entry may be discarded at any time.

2. Generate an agreement certificate that binds the request to a sequence number n. In our protocol, the agreement certificate is of the form $\langle \text{“COMMIT”}, v, n, d, \mathcal{A} \rangle_{\mathcal{A},\mathcal{E},2f+1}$ where v and n are the view and sequence number assigned by the agreement three phase commit protocol, d is the digest of the client's request ($d = D(m)$), and the certificate is authenticated by at least $2f + 1$ of the agreement servers $\mathcal{A}$ to the execution servers $\mathcal{E}$.

   Note that if the request's timestamp is no larger than the timestamp of a previous client request, then the agreement cluster still assigns the request a new sequence number. The execution cluster will detect the old timestamp t, assume such requests are retransmissions of earlier requests, and treat them as described below.

3. Send the request certificate and the agreement certificate to the execution cluster.

Second, when the agreement cluster receives a reply certificate $\langle \text{“REPLY”}, v, n, t, c, \mathcal{E}, r \rangle_{\mathcal{E},c,g+1}$ it relays the certificate to the client. Optionally, it may store the certificate in $cache_c$.

In addition to these two message-triggered actions, the agreement cluster performs retransmission of requests and agreement certificates if a timeout expires before it receives the corresponding reply certificate. Unlike the traditional architecture in Figure 6.2(a), communication between the agreement cluster and execution cluster is unreliable. And, although correct clients should repeat requests when they do not hear replies, it would be unwise to depend on (untrusted) clients to trigger the retransmissions needed to fill potential gaps in the sequence number space at execution nodes. For each agreement certificate, the timeout is set to an arbitrary initial value and then doubled after each retransmission (increasing the timeout is necessary to ensure that the timeout is eventually longer than the actual message delivery time). To bound the state needed to support retransmission, the agreement cluster has at most P requests outstanding in the execution pipeline and does not generate agreement certificate n until it has received a reply with a sequence number of at least $n - P$. Choosing P larger than the number of clients makes it possible to add a fairness mechanism that ensures that no client is starved.

Execution cluster behavior

The execution cluster implements application-specific state machines to process requests in the order determined by the agreement cluster. The intention is that all the correct replicas in the execution cluster have the same state Q, and that after a request r_1 is executed all the correct replicas have the same state Q_1 as a correct node executing r_1 would. This state includes not only the result of executing the client requests but also additional information necessary for retransmissions of replies. Links are unreliable, so a reply to some client c may be lost and the client's retransmission protocol can cause a request r to be seen again even though, unbeknownst to the client, r has already been executed. To support exactly-once semantics, execution nodes must not execute r again. Instead, they send the contents of $Reply_c$, the last reply certificate sent to client c. Execution nodes store one such certificate for each client.

When the execution cluster receives a valid request $\langle\text{“REQUEST”}, o, t, c\rangle_{c,\mathcal{A},1}$

and a valid agreement certificate $\langle\text{“COMMIT”}, v, n, d, \mathcal{A}\rangle_{\mathcal{A},\mathcal{E},2f+1}$ for that request, the cluster waits until all requests with sequence numbers smaller than n have been received and executed. Then, if the request's sequence number exceeds by one the highest sequence number executed so far, the cluster takes one of the following three actions.

1. If the request's timestamp exceeds the timestamp of the reply in $Reply_c$, then the cluster executes the new request, updates $Reply_c$, and sends the reply certificate to the agreement cluster.

2. If the request's timestamp equals $Reply_c$'s timestamp, then the request is a client-initiated retransmission of an old request, so the cluster generates, caches, and sends a new reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n.

3. If the request's timestamp is smaller than $Reply_c$'s timestamp, then the cluster must acknowledge the new sequence number so that the agreement cluster can continue to make progress, but it should not execute the lower-timestamped client request; therefore, the cluster acts as in the second case and generates, caches, and sends a new reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n.

The above three cases are relevant when the execution cluster processes a new sequence number. If, on the other hand, a request's sequence number is not larger than the highest sequence number executed so far, the execution cluster assumes the request is a retransmission from the agreement cluster, and it retransmits the client's last reply certificate from $Reply_c$; this reply is guaranteed to have a sequence number at least as large as the request's sequence number.

Note that in systems not using the Privacy Firewall architecture described in Section 6.10, a possible optimization is for the client to send the request certificate directly to both the agreement cluster and the execution cluster and for the execution cluster to send reply certificates directly to both the agreement cluster and the client.

Non-determinism

State machines replicated by the system must be deterministic to ensure their replies to a given request match and that their internal states do not diverge. However, many important services include some nondeterminism when executing requests. For example, in the network file system NFS, different replicas may choose different file handles to identify a file [130] or attach different last-access timestamps when a file is read [33]. To address this issue, we extend the standard technique [33, 130] of resolving nondeterminism which has the agreement phase select and sanity-check any nondeterministic values needed by a request. To separate more cleanly (generic) agreement from (application-specific) execution, our agreement cluster is responsible for generating nondeterministic inputs that the execution cluster deterministically maps to any application-specific values it needs.

For the applications we have examined, the agreement cluster simply includes a timestamp and a set of pseudo-random bits in each agreement certificate; similar to the BASE protocol, the primary proposes these values and the other agreement nodes (i) sanity-check them and (ii) refuse to agree to unreasonable timestamps. Then, the abstraction layer [130] at the execution nodes executes a deterministic function that maps these inputs to the values needed by the application. We believe that this approach will work for most applications, but future work is needed to determine if more general mechanisms are needed.

6.3.2 Internal Agreement Cluster Protocol

Above, we describe the high-level behavior of the agreement cluster as it responds to request certificates, reply certificates, and timeouts. Here we describe the internal details of how nodes in our system behave to meet these requirements.

For simplicity, our implementation uses Rodrigues et al.'s BASE (BFT with Abstract Specification Encapsulation) library [130], which implements a replicated Byzantine state machine by receiving, sequencing, and executing requests. We treat BASE as a Byzantine agreement module that handles the details of three-phase commit, sequence number assignment, view changes, checkpoints, and garbage collecting logs [33, 130]. In particular, clients send their requests to the BASE library on the agreement nodes to bind requests to sequence numbers. But when used as our agreement library, the BASE library does not execute requests directly against the *application* state machine, which is in our execution cluster. Instead, we install a message queue (described in more detail below) as the BASE library's *local* state machine, and the BASE library "executes" a request by calling **msgQueue.insert**(*request certificate, agreement certificate*). From the point of view of the existing BASE library, when this call completes, the request has been executed. In reality, this call enqueues the request for asynchronous processing by the execution cluster, and the replicated message queues ensure the request is eventually executed by the execution cluster. Our system makes four simple changes to the existing BASE library to accommodate this asynchronous execution.

1. Whereas the original library sends the result of the local execution of a request

to the client, the modified library does not send replies to clients; instead, it relies on the message queue to do so.

2. To ensure that clients can eventually receive the reply to each of their requests, the original BASE library maintains a cache of the last reply sent to each client and sends the cached value when a client retransmits a request. But when the modified library receives repeated requests from a client, it does not send the locally stored reply since the locally stored reply is the result of the enqueue operation, not the result of the executing the body of the client's request. Instead, it calls **msgQueue.retryHint(***request certificate***)**, telling the message queue to send the cached reply or to retry the request.

3. BASE periodically generates consistent checkpoints from its replicas' internal state so that buffered messages can be garbage collected while ensuring that nodes that have fallen behind or that have recovered from a failure can resume operation from a checkpoint [31, 32, 130]. In order to achieve a consistent checkpoint at some sequence number n across message queue instances despite their asynchronous internal operation, the modified BASE library calls **msgQueue.sync()** after inserting message n. This call returns after bringing the local message queue state to a consistent state as required by the BASE library's checkpointing and garbage collection algorithms.

4. The sequence number for each request is determined according to the order in which it is inserted into the message queue.

Message queue design

Each node in the agreement cluster has an instance of a message queue as its local state machine. Each message queue instance stores $maxN$, the highest sequence number in any agreement certificate received, and *pendingSends*, a list of request certificates, agreement certificates, and timeout information for requests that have been sent but for which no reply has been received. Optionally, each instance may also store $cache_c$, the reply certificate for the highest-numbered request by client c.

When the library calls **msgQueue.insert(***request certificate, agreement certificate***)**, the message queue instance stores the certificates in *pendingSends*, updates $maxN$, and multicasts both certificates to all nodes in the execution cluster. It then sets a per-request timer to the current time plus an initial time-out value. As an optimization, when a message is first inserted, only the current primary needs to send it to the execution cluster; in that case, all nodes retransmit if the timeout expires before they receive the reply. In order to bound the state needed by execution nodes for buffering recent replies and out of order requests, a pipeline depth P bounds the number of outstanding requests; an **msgQueue.insert(**...**)** call for sequence number n blocks (which prevents the node from participating in the generation of sequence numbers higher than n) until the node has received a reply with a sequence number at least $n - P$.

When an instance of the message queue receives a valid reply certified by $g+1$ execution cluster nodes, it deletes from *pendingSends* the request and agreement certificates for that request and for all requests with lower sequence numbers; it also cancels the retransmission timer for those requests. The message queue instance then forwards the reply to the client. Optionally, the instance updates $cache_c$ to store the reply certificate for client c.

When the modified BASE library calls **msgQueue.retryHint(***request certificate***)** for a request r from client c with timestamp t, the message queue instance first checks to see if $cache_c$ contains a reply certificate with a timestamp of at least

t. If so, it sends the reply certificate to the client. Otherwise, if a request and agreement certificate with matching c and t are available in *pendingSends*, then the queue resends the certificates to the execution cluster. Finally, if neither the *cache* nor the *pendingSends* contains a relevant reply or request for this client-initiated retransmission request, the message queue uses BASE to generate a new agreement certificate with a new sequence number for this old request and then calls **msgQueue.insert(**. . . **)** to transmit the certificates to the execution cluster.

When the retransmission timer expires for a message in *pendingSends*, the instance resends the request and agreement certificates and updates the retransmission timer to the current time plus twice the previous timeout interval.

Finally, when the modified BASE library calls **msgQueue.sync()**, the message queue stops accepting **insert(**. . .**)** requests or generating new agreement certificates and waits to receive a reply certificate for a request with sequence number $maxN$ or higher. Once it processes such a reply (*pendingSends* is therefore empty), the **sync()** call returns and the message queue begins accepting **insert(**. . .**)** calls again. Note that *cache* may differ across servers and is not included in checkpoints.

6.3.3 Internal Execution Cluster Protocol

Above, we described the high-level execution cluster's behavior in response to request and agreement certificates. Here, we describe execution node behaviors that ensure these requirements are met.

Each node in the execution cluster maintains the application state, a pending request list of at most P received but not yet executed requests (where P is the maximum pipeline depth of outstanding requests by the execution cluster), the largest sequence number $maxN$ that has been executed, and a table *reply* where

reply_c stores the node's piece of its most recent reply certificate for client c. Each node also stores the most recent *stable checkpoint*, which is a checkpoint across the application state and the table *reply* that is certified by at least $g + 1$ execution nodes. Nodes also store zero or more newer checkpoints that have not yet become stable.

When a node receives a valid request certificate certified by a client c and a valid agreement certificate certified by at least $2f + 1$ agreement nodes, it stores the certificates in the pending request list. Then, once all requests with lower sequence numbers have been received and processed, the node processes the request. If the request has a new sequence number (i.e., $n = maxN + 1$), the node takes one of two actions: (1) if $t > t'$ (where t is the request's timestamp and t' is the timestamp of the reply in reply_c), then the node handles the new request by updating $maxN$, executing the new request, generating the node's share of the full reply certificate, storing the partial reply certificate in reply_c, and sending the partial reply certificate to all nodes in the agreement cluster; or (2) if $t \leq t'$, then the node handles the client-initiated retransmission request by updating $maxN$ and generating, caching, and sending a new partial reply certificate containing the cached timestamp t', the cached reply body r', the request's view v, and the request's sequence number n. On the other hand, if the request has an old sequence number (i.e., $n \leq maxN$), then the node simply resends the partial reply certificate in reply_c, which is guaranteed to have a sequence number at least as large as the request's sequence number.

Liveness and retransmission

To eliminate gaps in sequence numbers caused by the unreliable communication between the agreement and execution clusters, the system uses a two-level retransmission strategy. For a request with sequence number n, retransmissions by the agreement cluster ensure that eventually at least one correct execution node receives and executes request n, and an internal execution cluster retransmission protocol ensures once that happens, all correct execution nodes eventually learn of request n or of some stable checkpoint newer than n. In particular, if an execution node receives request n but not request $n-1$, it multicasts to other nodes in the execution cluster a retransmission request for the request with missing sequence number. When a node receives such a message, it replies with the specified request and agreement certificates unless it has a stable checkpoint with a higher sequence number, in which case it sends the proof of stability for that checkpoint (see below.)

Checkpoints and garbage collection

Execution nodes periodically construct checkpoints in order to garbage collect pending request logs. Note that the inter-cluster protocol is designed so that garbage collection in the execution cluster requires no additional coordination with the agreement cluster and vice versa. Execution nodes generate checkpoints at prespecified sequence numbers (e.g., after executing request n where $n \bmod CP\_FREQ = 0$). Nodes checkpoint both the application state and their $reply_c$ list of replies to clients, but they do not include their pending request list in checkpointed state. As in previous systems [31, 32, 130], to reduce the cost of producing checkpoints, nodes can make use of copy on write and incremental cryptography [24].

After generating a checkpoint, execution servers assemble a proof of stability for it. When server i produces checkpoint C for sequence number n, it computes a digest of the checkpoint $d = D(C)$ and attest to its view of the checkpoint to the rest of the cluster by multicasting $\langle \text{“CHECKPOINT”}, n, d \rangle_{i,\mathcal{E},1}$ to all execution nodes.

Once a node receives $g+1$ such messages, it assembles them into a full checkpoint certificate:$\langle$“CHECKPOINT”$, n, d\rangle_{\mathcal{E},\mathcal{E},g+1}$

Once a node has a valid and complete checkpoint certificate for sequence number n, it can garbage collect state by discarding older checkpoints, discarding older checkpoint certificates, and discarding older agreement and request certificates from the pending request log.

6.3.4 Correctness

Agreement Cluster

Lemma 35. *The agreement cluster only generates agreement certificates for valid client requests.*

Proof. Correct nodes in the agreement cluster check the validity of the client request before assigning a sequence number. An invalid message can receive at most f signatures (from faulty agreement nodes), not enough for an agreement certificate. □

Lemma 36. *The agreement cluster never assigns the same sequence number to two different client requests.*

Proof. The protocol running on BASE assigns a new sequence number to every client request. Byzantine nodes in the agreement cluster could assign a sequence number that is out of sequence, but they are not numerous enough to form a valid agreement certificate. □

Lemma 37. *Every request from a correct client is eventually assigned a sequence number.*

Proof. Clients retransmit their requests until they receive a reply certificate. Reply certificates are only generated for requests with a sequence number, so clients retransmit until their request is assigned a sequence number. Since links are fair and clients are retransmitting to all agreement nodes, at least one of these nodes is correct and will eventually accept the request. At this point, the BASE protocol guarantees that the request will be assigned a sequence number. □

Execution Cluster

Lemma 38. *Correct nodes in the execution cluster process requests in sequence order.*

Proof. There is no gap in the sequence numbers generated by the agreement cluster, and the execution cluster protocol requires all lower-numbered requests to have been processed before processing on a new request can begin. □

Lemma 39. *Once the agreement cluster assigns a sequence number to a request, the execution cluster will eventually receive and execute the request.*

Proof. The message queue used by the agreement cluster ensures that the message is resent until a reply vouched by $f + 1$ execution nodes is received. This ensures that a single correct execution node has executed the request. The replicated state machine protocol guarantees that if one correct node executes a request, eventually all the correct nodes will. □

System

Informally, the system should behave (from the point of view of the client) in the same way as a single correct execution node, so in particular it should go through the same states Q as a possible execution from the single correct node. The lemma below expresses these requirements formally.

Lemma 40. *A client receives a reply* $\langle\text{“REPLY”}, v, n, t, c, \mathcal{E}, r\rangle_{\mathcal{E},c,g+1}$ *only if all of the following four conditions hold.*

- *earlier the client issued a request* $\langle\text{“REQUEST”}, o_n, t, c\rangle_{c,\mathcal{A},1}$*,*
- *the reply value* r *reflects the output of state machine in state* Q_{n-1} *executing request operation* o_n*,*
- *there exists some sequence of operations* $\mathcal{O}$ *such that state* Q_{n-1} *is the state reached by starting at initial state* Q_0 *and sequentially executing each request* o_i $(0 \leq i < n)$ *in* $\mathcal{O}$ *as the* i*th operation on the state machine, and*
- *a valid reply certificate for any subsequent request reflects a state in which the execution of* o_i *is the* i*'th action by the state machine* $(0 \leq i \leq n)$*.*

Proof. The reply is signed by at least one correct execution node. The first condition holds because correct execution nodes only answer valid requests, and the client name and timestamp of the reply will match those of the request. The second and third conditions hold because correct execution nodes execute requests in sequence number order (Lemma 38). The fourth condition holds because valid reply certificates can only be generated if at least one of the replying nodes is correct, and the state of that correct node will reflect the execution of the client's request. □

Lemma 41. *If a client* c *sends a request with timestamp* t*, where* t *exceeds any timestamp in any previous request by* c*, and* c *repeatedly sends that request and no other request until it receives a reply, then eventually* c *will receive a valid reply for that request.*

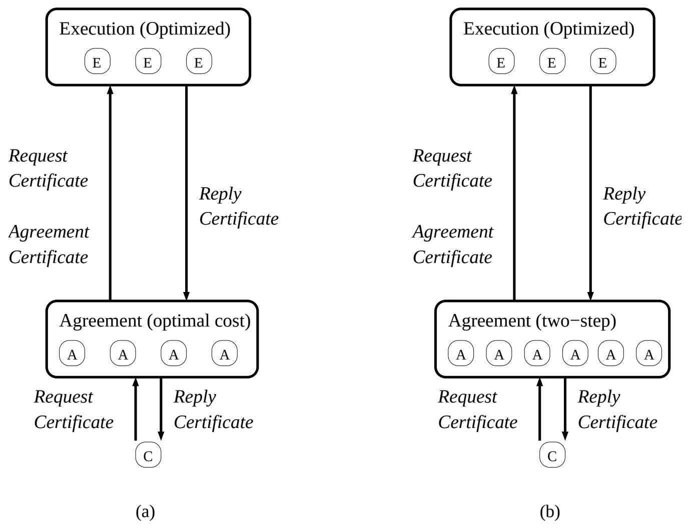

Figure 6.3: Trade-off when optimizing the agreement cluster: (a) minimal number of nodes for minimal cost or (b) two-step consensus at the cost of additional nodes.

Proof. Since links are fair, continually resending the request ensures that a correct agreement node will eventually receive it. Since the request is valid and has a new timestamp, the agreement cluster will assign a new sequence number to the request (Lemma 35). The request will then be eventually executed (Lemma 39). Every correct agreement node resends until it receives a reply, and correct execution nodes resend their reply if they receive the request again. So, eventually all correct agreement nodes receive the reply, and they will forward it to the client. □

6.4 Fast Byzantine Consensus

We now focus our attention on the agreement cluster (Figure 6.3). Since the agreement replicas are simple and cheap, we look into adding agreement replicas to reach consensus in two communication steps in the common case instead of three steps in previous consensus protocols (we say that the protocol is *two-step*).

At the heart of the agreement cluster is the Consensus protocol [62]. In the Consensus protocol, all nodes have an initial value. When the protocol ends, all correct nodes must learn the same value v and v must be the initial value from some node (instead of *learn*, some use the term *decide*). The Consensus protocol can be generalized so that not all nodes propose a value (only *proposers*) and not all nodes learn a value (only *learners*). Nodes can assume several roles. The Paxos-style definition of consensus [84], shown below, uses these roles.

CS1 Only a value that has been proposed may be chosen.

CS2 Only a single value may be chosen.

CS3 Only a chosen value may be learned by a correct learner.

CL1 Some proposed value is eventually chosen.

CL2 Once a value is chosen, correct learners eventually learn it.

In the next few sections we show the following two results for an asynchronous Byzantine consensus protocol that must be able to tolerate f Byzantine failures.

1. The minimal number of agreement replicas for two-step consensus despite t actual failures is $3f + 2t + 1$ when all nodes can propose and learn.
2. The new Parameterized Fast asynchronous Byzantine Paxos protocol (Parameterized FaB Paxos) matches this lower bound in the common case. Parameterized FaB Paxos also avoids digital signatures in the common case.

6.5 Lower Bound on Two-Step Consensus

We show that $3f + 2t + 1$ is the minimal number of processes for parameterized two-step consensus that can tolerate f Byzantine failures and completes in two communications steps in the common case despite t failures, when all the nodes can propose and learn. Our proof does not label nodes as proposers, acceptors, or learners, since we focus on the case where all nodes can play all roles.

The proof proceeds by constructing two executions that are indistinguishable although they should learn different values. We now define these notions precisely.

Consider a system of n processes that communicate through a fully connected network. Process execution consists of a sequences of events, which can be of three types: *local*, *send*, and *deliver*. We call the sequence of events exhibited by a process its *local history*.

Execution of the protocol proceeds in asynchronous rounds. In a round, each correct process (i) sends a message to every other process, (ii) waits until it receives a (possibly empty) message sent in that round from $n-f$ distinct processes (ignoring any extra messages), and (iii) performs a (possibly empty) sequence of local events. We say that the process takes a *step* in each round. During an execution, the system goes through a series of *configurations*, where a configuration C is an n-vector that stores the state of every process. We also talk about the *state* of a set of processes, by which we mean a vector that stores the state of the processes in the set.

This proof depends on the notion of indistinguishability. The notions of *view* and *similarity* help us capture this precisely.

Definition 10. *Given an execution ρ and a process p_i, the* view *of p_i in ρ, denoted by $\rho|p_i$, is the local history of p_i together with the state of p_i in the initial configuration of ρ.*

Definition 11. *Let ρ_1 and ρ_2 be two executions, and let p_i be a process that is correct in ρ_1 and ρ_2. Execution ρ_1 is* similar *to execution ρ_2 with respect to p_i, denoted as $\rho_1 \overset{p_i}{\sim} \rho_2$, if $\rho_1|p_i = \rho_2|p_i$.*

If an execution ρ results in all correct processes learning a value v, we say that v is the *consensus value* of ρ, which we denote $c(\rho)$. For the remainder of this section we only consider executions of consensus that result in all correct processes learning a value.

Lemma 42. *Let ρ_1 and ρ_2 be two executions, and let p_i be a process which is correct in ρ_1 and ρ_2. If $\rho_1 \overset{p_i}{\sim} \rho_2$, then $c(\rho_1) = c(\rho_2)$.*

Proof. Since we are only considering executions of consensus that result in all correct processes learning a value, $c(\rho_1)$ and $c(\rho_2)$ are well-defined. The correct process p_i cannot distinguish between ρ_1 and ρ_2, so it will learn the same value in both executions. Consensus requires that all correct learners learn the consensus value, so $c(\rho_1) = c(\rho_2)$. □

Definition 12. *Let $\mathcal{F}$ be a subset of the processes in the system. An execution ρ is $\mathcal{F}$-silent if in ρ no process outside $\mathcal{F}$ delivers a message from a process in $\mathcal{F}$.*

Definition 13. *Let a* two-step execution *be an execution in which all correct processes learn by the end of the second round. A consensus protocol is* (t,2)-step *if it can tolerate f Byzantine failures and if for every initial configuration I and every set $\mathcal{F}$ of at most t processes ($t \leq f$), there exists a two-step execution of the protocol from I that is $\mathcal{F}$-silent. If the protocol is (f,2)-step then we simply say that it is* two-step.

Definition 14. *Given a (t,2)-step consensus protocol, an initial configuration I is* (t,2)-step bivalent *if there exist two disjoint sets of processes $\mathcal{F}_0$ and $\mathcal{F}_1$, ($|\mathcal{F}_0| \leq t$*

and $|\mathcal{F}_1| \leq t$) an $\mathcal{F}_0$-silent two-step execution ρ_0 and an $\mathcal{F}_1$-silent two-step execution ρ_1 such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$.

Lemma 43. *For every (t,2)-step consensus protocol with $n > 2f$ there exists a (t,2)-step bivalent initial configuration.*

Proof. Consider a (t,2)-step consensus protocol P. For each i, $0 \leq i \leq n$, let I^i be the initial configuration in which the first i processes propose 1, and the remaining processes propose 0. By the definition of (t,2)-step, for every I^i and for all $\mathcal{F}$ such that $|\mathcal{F}| \leq t$ there exists at least one $\mathcal{F}$-silent two-step execution ρ^i of P. By property CS1 of consensus, $c(\rho^0) = 0$ and $c(\rho^n) = 1$. Consider now $\mathcal{F}_0 = \{p_j : 1 \leq j \leq t\}$. There must exist two neighbor configurations I^i and I^{i+1} and two $\mathcal{F}_0$-silent two-step executions ρ^i and ρ^{i+1} such that $c(\rho^i) \neq c(\rho^{i+i})$ and ρ^{i+1} is the lowest-numbered execution with consensus value 1. Note that $i \geq t$, since both ρ^i and ρ^{i+1} are $\mathcal{F}_0$-silent and the consensus value they reach cannot depend on the value proposed by the silent processes in $\mathcal{F}_0$. We claim that one of I^i and I^{i+1} is (t,2)-step bivalent. To prove our claim, we set $x = min(i + t, n)$ and define $\mathcal{F}_1$ as the set $\{p_j : x + 1 - t \leq j \leq x\}$. Note that, by construction, $\mathcal{F}_0$ and $\mathcal{F}_1$ are disjoint and $(i + 1) \in \mathcal{F}_1$. Since P is two-step, there must in turn exist two two-step executions π^i and π^{i+1} that are $\mathcal{F}_1$-silent, where π^i starts from configuration I^i and π^{i+1} starts from I^{i+1}. The only difference between configurations I^i and I^{i+1} is the value proposed by p_{i+1}, which is silent in π^i and π^{i+1}, since it belongs to $\mathcal{F}_1$. Hence, all processes outside of $\mathcal{F}_1$ (at least one of which is correct) have the same view in π^i and π^{i+1}, and $c(\pi^i) = c(\pi^{i+1})$. Since $c(\rho^i) \neq c(\rho^{i+1})$ and $c(\pi^i) = c(\pi^{i+1})$, either I^i or I^{i+1} has two two-step executions that lead to different consensus values. This is the definition of a (t,2)-step bivalent configuration. $\square$

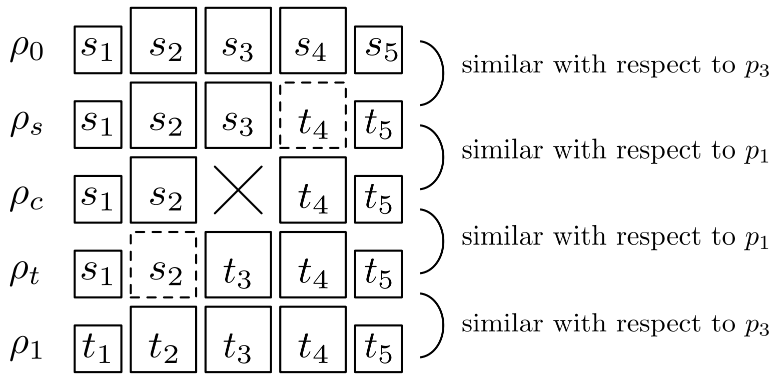
ρ_0
ρ_s
ρ_c
ρ_t
ρ_1
s_1 s_2 s_3 s_4 s_5
s_1 s_2 s_3 t_4 t_5
s_1 s_2 t_4 t_5
s_1 s_2 t_3 t_4 t_5
t_1 t_2 t_3 t_4 t_5
similar with respect to p_3
similar with respect to p_1
similar with respect to p_1
similar with respect to p_3

Figure 6.4: Contradiction sketch: The figure represents a system with too few ($3f + 2t$) processes. Each row represents an execution, and the boxes represent sets of processes. Dotted boxes contain Byzantine nodes. The first execution (ρ_0) learns 0, and the last learns 1. Each execution is similar to the next, leading to the contradiction.

Figure 6.4 shows a sketch of the idea at the core of the proof: with only $3f+2t$ processes we can construct two executions (ρ_0 and ρ_1) that are indistinguishable, even though they learn different values.

Theorem 10. *Any (t,2)-step Byzantine fault-tolerant consensus protocol requires at least* $3f + 2t + 1$ *processes.*

Proof. By contradiction. Suppose there exists a (t,2)-step fault-tolerant consensus protocol P that (i) tolerates up to f Byzantine faults, (ii) is two-step despite t failures, and (iii) requires only $3f + 2t$ processes. We partition the processes in five sets, $p_1 \ldots p_5$.

By Lemma 43 there exist a (t,2)-step bivalent configuration I_b and two two-step executions ρ_0 and ρ_1, respectively $\mathcal{F}_0$-silent and $\mathcal{F}_1$-silent, such that $c(\rho_0) = 0$ and $c(\rho_1) = 1$. We name the sets of processes so that $\mathcal{F}_0 = p_5$ and $\mathcal{F}_1 = p_1$ (so p_1 and p_5 have size t). The remaining sets have size f.

We focus on the state of $p_1, \ldots, p_5$ at the end of the first round, where the state of p_i is a set of local states, one for each process in p_i. In particular, let s_i and t_i denote the state of p_i at the end of the first round of ρ_0 and ρ_1, respectively. p_i has state s_i (respectively, t_i) at the end of any execution that produces for its nodes the same view as ρ_0 (respectively, ρ_1). It is possible for some processes to be in an s state at the end of the first round while at the same time others are in a t state. Consider now three new (not necessarily two-step) executions of P, ρ_s, ρ_t, and ρ_c, that at the end of their first round have p_1 and p_2 in their s states and p_4 and p_5 in their t states. The state of p_3 is different in the three executions: in ρ_s, p_3 is in state s_3; in ρ_t, p_3 is in state t_3; and in ρ_c, p_3 crashes at the end of the first round. Otherwise, the three executions are very much alike: all three executions are p_3-silent from the second round on—in ρ_c because p_3 has crashed, in ρ_s and ρ_t because all processes in p_3 are slow. Further, all processes other than those in p_3 send and deliver the same messages in the same order in all three executions, and all three executions enter a period of synchrony from the second round on, so that in each execution consensus must terminate and some value must be learned. We consider three scenarios, one for each execution.

ρ_s **scenario:** In this scenario, the f nodes in p_4 are Byzantine: they follow the protocol correctly in their messages to all processes but those in p_3. The messages that nodes in p_4 send to p_3 in round two are consistent with p_4 being in state s_4, rather than t_4. Further, in the second round of ρ_s the messages from p_5 to p_3 are the last to reach p_3 (and are therefore not delivered by p_3), and all other messages are delivered by p_3 in the same order as in ρ_0. The view of p_3 at the end of the second round of ρ_s is the same as in the second round of ρ_0; hence nodes in p_3 learn 0 at the end of the second round of ρ_s (it must learn then because ρ_0 is two-step).

Since nodes in p_3 are correct and for each node $p \in p_3$ $\rho_s \overset{p}{\sim} \rho_0$, then $c(\rho_s) = c(\rho_0)$ and all correct processes in ρ_s eventually learn 0.

ρ_t **scenario:** In this scenario, the f nodes in p_2 are Byzantine: they follow the protocol correctly in their messages to all processes but those in p_3. In particular, the messages that nodes in p_2 send to p_3 in round two are consistent with p_2 being in state t_2, rather than s_2. Further, in the second round of ρ_t the messages from p_1 to p_3 are the last to reach p_3 (and are therefore not delivered by p_3), and all other messages are delivered by p_3 in the same order as in ρ_1. The view of p_3 at the end of the second round of ρ_t is the same as in the second round of ρ_1; hence nodes in p_3 learn 1 at the end of the second round of ρ_t. Since nodes in p_3 are correct and for each node $p \in p_3$, $\rho_t \overset{p}{\sim} \rho_1$, then $c(\rho_t) = c(\rho_1)$ and all correct processes in ρ_t eventually learn 1.

ρ_c **scenario:** In this scenario, the f nodes in p_3 have crashed, and all other processes are correct. Since ρ_c is synchronous from round two on, every correct process must eventually learn some value.

Consider now a process p in p_1 that is correct in ρ_s, ρ_t, and ρ_c. By construction, $\rho_c \overset{p}{\sim} \rho_t$, and therefore $c(\rho_c) = c(\rho_t) = c(\rho_1) = 1$. However, again by construction, $\rho_c \overset{p}{\sim} \rho_s$, and therefore $c(\rho_c) = c(\rho_s) = c(\rho_0) = 0$. Hence, p in ρ_c must learn both 0 and 1: this contradicts CS2 and CS3 of consensus, which together imply that a correct learner may learn only a single value. $\square$

6.6 Fast Byzantine Consensus Protocol

We now present *FaB Paxos* (for Fast asynchronous Byzantine Paxos), a two-step Byzantine fault-tolerant consensus protocol that requires $5f+1$ processes—since FaB

variable	initial	comment
Globals		
p, a, l		Number of proposers, acceptors, learners
f		Number of Byzantine failures tolerated
Proposer variables		
Satisfied	$\emptyset$	Set of proposers that claim to be satisfied
Learned	$\emptyset$	Set of learners that claim to have learned
Acceptor variables		
$accepted$	$(\perp, \perp)$	Value accepted and the corresponding proposal number
i		The acceptor number
Learner variables		
$accepted[j]$	$(\perp, \perp)$	Value and matching proposal number acceptor j says it accepted
$learn[j]$	$(\perp, \perp)$	Value and matching proposal number learner j says it learned
$learned$	$(\perp, \perp)$	Value learned and the corresponding proposal number

Figure 6.5: Variables for the FaB Paxos pseudocode

Paxos is two-step regardless of the number of actual failures (up to f), it matches the lower bound proven in the previous section. More precisely, In FaB Paxos we assign three roles to the nodes: acceptors, proposers and learners. FaB Paxos requires $a \geq 5f + 1$ acceptors, $p \geq 3f + 1$ proposers, and $l \geq 3f + 1$ learners; as in Paxos, each process in FaB Paxos can play one or more of these three roles. We describe FaB Paxos in stages: we start by describing a simple version of the protocol that relies on relatively strong assumptions, and we proceed by progressively weakening the assumptions and refining the protocol accordingly.

6.6.1 The Common Case

We first describe how FaB Paxos works in the common case, when there is a unique correct leader, all correct acceptors agree on its identity, and the system is in a period of synchrony (i.e. a period during which messages are reliably delivered, messages are delivered and processed within some time bound, and all clocks run at the same rate).

FaB Paxos is very simple in the common case, as can be expected by a protocol that terminates in two steps. Figure 6.5 shows the variables used, and Figure 6.6 shows the protocol's pseudocode. The *pnumber* variable (proposal number) indicates which process is the leader; in the common case, its value will not change.

```
1.  leader.onStart():
2.      // proposing (PC is null unless recovering)
3.      send ("PROPOSE",$value, pnumber, PC$) to all acceptors
4.      until $|Satisfied| >= \lceil (p + f + 1)/2 \rceil$
5.
6.  proposer.onLearned(): from learner $l$
7.      $Learned := Learned \cup \{l\}$
8.      if $|Learned| >= \lceil (l + f + 1)/2 \rceil$ then
9.          send ("SATISFIED") to all proposers
10.
11. proposer.onStart():
12.     wait for timeout
13.     if $|Learned| < \lceil (l + f + 1)/2 \rceil$ then
14.         leader-election.suspect(
15.             leader-election.getRegency() )
16.
17. proposer.onSatisfied(): from proposer $x$
18.     $Satisfied := Satisfied \cup \{x\}$
19.
20. acceptor.onPropose($value,pnumber,progcert$): from leader
21.     if not already accepted then
22.         $accepted := (value, pnumber)$ // accepting
23.         send ("ACCEPTED",$accepted$) to all learners
24.
25. learner.onAccepted($value,pnumber$): from acceptor $ac$
26.     $accepted[ac] := (value, pnumber)$
27.     if there are $\lceil (a + 3f + 1)/2 \rceil$ acceptors $x$
28.     such that $accepted[x] == (value, pnumber)$ then
29.         $learned := (value, pnumber)$ // learning
30.         send ("LEARNED") to all proposers
31.
32. learner.onStart():
33.     wait for timeout
34.     while (not learned) send ("PULL") to all learners
35.
36. learner.onPull(): from learner $ln$
37.     If this process learned some pair $(value, pnumber)$ then
38.         send ("LEARNED",$value, pnumber$) to $ln$
39.
40. learner.onLearned($value,pnumber$): from learner $ln$
41.     $learn[ln] := (value, pnumber)$
42.     if there are $f + 1$ learners $x$
43.     such that learn$[x] == (value, pnumber)$ then
44.         $learned := (value, pnumber)$ // learning
```

Figure 6.6: FaB pseudocode (excluding recovery)

The code starts executing in the **onStart()** methods. In the first step, the leader proposes its value to all acceptors (line 3). In the second step, the acceptors accept this value (line 22) and forward it to the learners (line 23). Learners learn a value v when they observe that $\lceil (a + 3f + 1)/2 \rceil$ acceptors have accepted the value (line

27). In the common case, a value will be learned before the timeout at line 12 is triggered. We will use that code later; the leader election interface is given in Figure 6.7. FaB Paxos avoids digital signatures in the common case because they are computationally expensive. Adding signatures would reduce neither the number of communication steps nor the number of servers since FaB Paxos is already optimal in these two measures.

Correctness We defer the full correctness proof for FaB Paxos until after the recovery protocol in Section 6.6.4—in the following we give an intuition of the correctness argument.

Let correct acceptors only accept the first value they receive from the leader and let a value v be *chosen* if $\lceil (a + f + 1)/2 \rceil$ correct acceptors have accepted it. These two requirements are sufficient to ensure CS1 and CS2: only a proposed value may be chosen and there can be at most one chosen value since at most one value can be accepted by a majority of correct acceptors. The last safety clause (CS3) requires correct learners to learn only a chosen value. Since learners wait for $\lceil (a+3f+1)/2 \rceil$ identical messages and at most f of those come from faulty acceptors, it follows that the value was necessarily chosen.

6.6.2 Fair Links and Retransmissions

In our discussion of the common case, we have assumed synchrony. While this is a reasonable assumption in the common case, our protocol must also be able to handle periods of asynchrony. We next weaken our network model to consider fair asynchronous authenticated links (see Section 6.2). Now, consensus may take more than two communication steps to terminate, e.g. when all messages sent by the leader in the first round are dropped.

Our end-to-end retransmission policy is based on the following pattern: the caller sends its request repeatedly, and the callee sends a single response every time it receives a request. When the caller is satisfied by the reply, it stops retransmitting. We alter the pattern slightly in order to accommodate the leader election protocol: other processes must be able to determine whether the leader is making progress, and therefore the leader must make sure that they, too, receive the reply. To that end, learners report not only to the leader but also to the other proposers (Figure 6.6, line 30). When proposers receive enough acknowledgments, we say they are *satisfied*. Satisfied proposers notify the leader (line 9). The leader only stops resending when it receives $\lceil (p + f + 1)/2 \rceil$ such notifications (line 4). If proposers do not hear from $\lceil (l + f + 1)/2 \rceil$ learners after some time-out period, they start suspecting the leader (line 13). If $\lceil (p+f+1)/2 \rceil$ proposers suspect the leader, then a new leader is elected.[2] The retransmission policy therefore ensures that in periods of synchrony the leader will retransmit until it is guaranteed that no leader election will be triggered. Note that the proposers do not wait until they hear from all learners before becoming satisfied (since some learners may have crashed). It is possible therefore that the leader stops retransmitting before all learners have learned the value. To ensure that eventually all correct learners do learn the value, lines 32–44 of the protocol require all correct learners still in the dark to pull the value from their peers.

6.6.3 Recovery Protocol

Recovery is initiated when the leader election protocol elects a new leader. Although we can reuse existing leader election protocols as-is, it is useful to discuss the properties of leader election. The output of leader election is a regency number r. This

[2]We do not show the election protocol, because existing leader election protocols can be used here without modification, e.g. the leader election protocol in [33].

```
1.  int leader-election.getRegency()
2.      // return the number of the current regent (leader is regent % p)
3.      // if no correct node suspects it then the regency continues.
4.
5.  int leader-election.getLeader()
6.      return getRegency() % p
7.
8.  void leader-election.suspect(int regency)
9.      // indicates suspicion of the leader for regency.
10.     // if a quorum of correct nodes suspect the same regency r,
11.     // then a new regency will start
12.
13. void leader-election.consider(proof)
14.     // consider outside evidence that a new leader was elected
```

Figure 6.7: Interface for leader election protocol

number never decreases, and proposer $r \mod p$ is the leader. Each node in the system has an instance of a leader-election object, and different instances may initially indicate different regents. The leader-election object takes as input suspicion about node failures (Figure 6.7, line 8). If no more than f nodes are Byzantine and at least $2f+1$ nodes participate in leader election, then leader election guarantees that if no correct node suspects the current regent, then eventually (i) all leader-election objects will return the same regency number and (ii) that number will not change. Leader election also guarantees that if a quorum of correct nodes ($\lceil (p+f+1)/2 \rceil$ nodes out of p) suspects regent r, then the regency number at all correct nodes will eventually be different from r. Finally, leader election generates a $proof_r$ when it elects some regent r. If $proof_r$ from a correct node is given to a leader-election object o, then o will elect regency $r', r \leq r'$.

The interface to leader election is shown in Figure 6.7. **getRegency()** returns the current regency number, and **getLeader()** converts it to a proposer number. Nodes indicate their suspicion by calling **suspect(**r**)**. When leader-election elects a new leader, it notifies the node through the **onElected(**$regency$, $\boldsymbol{proof}_r$**)** callback (not shown). If necessary, $proof_r$ can then be given to other leader-election objects through the **consider**($proof_r$) method.

When proposers suspect the current leader is faulty, they trigger an election for a new leader that then invokes the recovery protocol. Two scenarios require special care.

First, some value v may have already been chosen: the new leader must then propose the same v to maintain CS2. Second, a previous malicious leader may have performed a *poisonous write* [110], i.e. a write that prevents learners from reading any value—for example, a malicious leader could propose a different value to each acceptor. If the new leader is correct, consensus in a synchronous execution should nonetheless terminate.

In our discussion so far, we have required acceptors to only accept the first value they receive (Figure 6.6, line 21–22). If we maintained this requirement, the new leader would be unable to recover from a poisonous write. We therefore allow acceptors to accept multiple values. Naturally, we must take precautions to ensure that CS2 still holds.

Progress certificates and the recovery protocol

If some value v was chosen, then in order to maintain CS2 a new correct leader must not propose any value other than v. In order to determine whether some value may have been chosen, the new leader queries acceptors for their states. It can gather at most $a - f$ replies. We call this set of replies a *progress certificate*. The progress certificate serves two purposes. First, it allows a new correct leader to determine whether some value v may have been chosen, in which case the leader proposes v. We say that a correct leader will only propose a value that the progress certificate *vouches for*—we discuss in the next subsection how a progress certificate vouches for a value. Second, the progress certificate allows acceptors to determine the legitimacy of the value proposed by the leader, so that a faulty leader can not corrupt the state after some value was chosen. In order to serve the second purpose, we require the replies in the progress certificate to be signed.

A progress certificate PC must have the property that if some value v was chosen, then PC only vouches for v (since v is the only proposal that maintains CS2). PC must also have the property that it always vouches for at least one value, to ensure progress despite poisonous writes.

In the recovery protocol, the newly elected correct leader α first gathers a progress certificate by querying acceptors and receiving $a - f$ signed responses. Then, α decides which value to propose: If the progress certificate vouches for some value v, then α proposes v. Otherwise, α is free to propose any value. To propose its value, α follows the normal leader protocol, piggybacking the progress certificate alongside its proposal to justify its choice of value. The acceptors check that the new proposed value is vouched for by the progress certificate, thus ensuring that the new value does not endanger safety.

As in Paxos, acceptors who hear of the new leader (when the new leader gathers the progress certificate) promise to ignore messages with a lower proposal number (i.e. messages from former leaders). In order to prevent faulty proposers from displacing a correct leader, the leader election protocol provides a proof-of-leadership token to the new leader (typically, a collection of signed "election" messages).

Constructing progress certificates

A straightforward implementation of progress certificates would consist of the currently accepted value, signed, from $a - f$ acceptors. If these values are all different, then no value was chosen: in this case the progress certificate can vouch for any value since it is safe for the new leader to propose any value.

Unfortunately, this implementation falls short: a faulty new leader could use such a progress certificate *twice* to cause two different values to be chosen. Further, this can happen even if individual proposers only accept a given progress certificate once. Consider the following situation. We split the acceptors into four groups; the first group has size $2f+1$, the second has size f and contains malicious acceptors, and the third and fourth have size f. Suppose the values they have initially accepted are "A","B","B", and "C", respectively. A malicious new leader λ can gather a progress certificate establishing that no value has been chosen. With this certificate, λ can first sway f acceptors from the third group to accept "A" (by definition, "A" is now chosen), and then, using the same progress certificate, persuade the acceptors in the first and fourth group to change their value to "B"—"B" is now chosen. Clearly, this execution violates CS2.

We make three changes to prevent progress certificates from being used twice. First, we allow a proposer to propose a new value only once while it serves as a leader. Specifically, we tie progress certificates to a *proposal number*, whose value equals the number of times a new leader has been elected.

Second, we associate a proposal number to proposed values. Acceptors now accept a value for a given proposal number rather than just a value. Where before acceptors forwarded just the accepted value (to help learners learn, or in response to a leader's query), now they forward both the accepted value *and* its proposal number—hence, progress certificates now contain (value, proposal number) pairs.

Learners learn a value v if they see that $\lceil (a+3f+1)/2 \rceil$ acceptors accepted value v for the *same* proposal number. We say that value v is *chosen for pn* if

```
101. leader.onElected(newnumber,proof) :
102.     // this function is called when leader-election picks a new regency
103.     // proof is a piece of data that will sway leader-election at the
104.     // other nodes.
105.     $pnumber := newnumber$ // no smaller than the previous pnumber
106.     if (not leader for $pnumber$) : return
107.     repeatedly send ("QUERY",$pnumber$,$proof$) to all acceptors
108.     until get ("REP", $\langle value_j, pnumber \rangle_j$) from $a - f$ acceptors $j$
109.     $PC :=$ the union of these replies
110.     if $PC$ vouches for $(v', pnumber)$ : $value := v'$
111.     onStart()

113. acceptors.onQuery(pn,proof) : from leader
114.     leader-election.consider(proof)
115.     if (leader-election.getRegency() $\neq pn$) :
116.         return // ignore bad requests
117.     send ("REP", $\langle value,pn \rangle_i$) to leader-election.getLeader()

118. acceptor.onPropose(value,pnumber,progcert) : from proposer
119.     if $pnumber \neq$ leader-election.getRegency() :
120.         return // only listen to current leader
121.     if accepted $(v, pn)$ and $pn == pnumber$ :
122.         return // only once per prop. number
123.     if accepted $(v, pn)$ $\wedge$ $v \neq value$ $\wedge$
124.     $progcert$ does not vouch for $(value, pnumber)$ :
125.         return // only change with progress certificate
126.     $accepted := (value, pnumber)$ // accepting
127.     send ("ACCEPTED",$accepted$) to all learners
```

Figure 6.8: FaB Paxos recovery pseudocode

$\lceil (a + f + 1)/2 \rceil$ correct acceptors have accepted that value for proposal number pn. We say that value v is chosen if there is some proposal number pn so that v is chosen for pn.

Third, we change the conditions under which acceptors accept a value (Figure 6.8). In addition to ignoring proposals with a proposal number other than the current regency (line 119), acceptors only accept one proposal for every proposal number (line 121) and they only change their accepted value if the progress certificate vouches for the new value and proposal number (lines 123–125).

We are now ready to define progress certificates concretely. A progress certificate contains signed replies (v_i, pn) from $a - f$ acceptors (Figure 6.8, line 108). An acceptor's reply contains that acceptor's currently accepted value and the proposal number of the leader who requested the progress certificate.

Definition 15. *A progress certificate* $\{\langle v_0, pn\rangle_{s_0}, \ldots, \langle v_{a-f}, pn\rangle_{s_{a-f}}\}$ vouches *for value v at proposal number pn if there is no value $v_i \neq v$ that appears $\lceil (a-f+1)/2 \rceil$ times in the progress certificate.*

A consequence of this definition is that if some specific pair (v, pn) appears at least $\lceil (a-f+1)/2 \rceil$ times in the progress certificate, then the progress certificate vouches only for value v at proposal pn. If there is no such pair, then the progress certificate vouches for any value as long as its proposal number matches the one in the progress certificate. As we prove in the next section, progress certificates guarantee that if v is chosen for pn, then all progress certificates with a proposal number following pn will vouch for v and no other value.

Let us revisit the troublesome scenario of before in light of these changes. Suppose, without loss of generality, that the malicious leader λ gathers a progress certificate for proposal number 1 (λ is the second proposer to become leader). Because of the poisonous write, the progress certificate allows the leader to propose any new value. To have "A" chosen, λ sends a new proposal ("A", 1) together with the progress certificate first to the acceptors in the first group and then to the acceptors in the third group. Note that the first step is critical to have "A" chosen, as it ensures that the $3f + 1$ correct acceptors in the first and third group accept the same value for the same proposal number.

Fortunately, this first step is also what prevents λ from using the progress certificate to sway the acceptors in the first group to accept "B". Because they have last accepted the pair ("A", 1), when λ presents to the acceptors in the first group the progress certificate for proposal number 1 for the second time, they will refuse it (Figure 6.8, line 121).

6.6.4 Correctness

We now prove that, for executions that are eventually synchronous [49], FaB Paxos solves consensus. Recall that a value v is chosen for proposal pn iff $\lceil (a+f+1)/2 \rceil$ correct acceptors accept v for proposal pn. As mentioned at the beginning of this section, we assume that $a > 5f$, $p > 3f$, and $l > 3f$.

Theorem 11 (CS1)**.** *Only a value that has been proposed may be chosen.*

Proof. To be chosen, a value must be accepted by a set of correct acceptors (by definition), and correct acceptors only accept values that are proposed (Figure 6.8, line 118). $\square$

We prove CS2 (only a single value may be chosen) by way of the following two lemmas.

Lemma 44. *For every proposal number pn, at most one value is chosen.*

Proof. Correct acceptors only accept one value per proposal number (line 121). In order for a value to be chosen for pn, the value must be accepted by at least a majority of the correct acceptors (by definition). Hence, at most one value is chosen per proposal number. $\square$

Lemma 45. *If value v is chosen for proposal pn, then every progress certificate for proposal number $pn' > pn$ will vouch for v and no other value.*

Proof. Assume that value v is chosen for proposal pn; then, by definition, at least $c = \lceil (a+f+1)/2 \rceil$ correct acceptors have accepted v for proposal pn. Let PC be a progress certificate for proposal number $pn' > pn$. All correct acceptors that accepted v for pn must have done so before accepting PC, since no correct acceptor would accept v for proposal pn if it had accepted PC with $pn' > pn$ (line 119 and the fact that the regency number never decreases). Consider the $a - f$ pairs contained in PC. Since these pairs are signed (line 117), they cannot have been manufactured by the leader; hence, at least $a - f + c - a = \lceil (a - f + 1)/2 \rceil$ of them must be signed by acceptors that accepted v for pn. By definition, then, PC vouches for v and no other value. □

Theorem 12 (CS2)**.** *Only a single value may be chosen.*

Proof. Consider the smallest proposal number pn for which a value is chosen. By lemma 44, a unique value v is chosen for pn. By lemma 45, no later progress certificate can be constructed that vouches for a value other than v, so none of the correct acceptors that accepted v will change to accept a different value in later proposals. Since these correct acceptors form a majority, no other value can be chosen. □

Theorem 13 (CS3)**.** *Only a chosen value may be learned by a correct learner.*

Proof. Suppose that a correct learner learns value v for proposal pn. There are two ways for a learner to learn a value in FaB Paxos.

- $\lceil (a + 3f + 1)/2 \rceil$ acceptors reported having accepted v for proposal pn (line 27). At least $\lceil (a + f + 1)/2 \rceil$ of these acceptors are correct, so by definition v was chosen for pn.
- $f + 1$ other learners reported that v was chosen for pn (line 42). One of these learners is correct—so, by induction on the number of learners, it follows that v was indeed chosen for pn.

□

We say that a value is *stable* if it is learned by $\lceil (l - f + 1)/2 \rceil$ correct learners.

Lemma 46. *Some value is eventually stable.*

Proof. The system is eventually synchronous and in these synchronous periods, leaders that do not create a stable value are eventually suspected by all correct proposers (Figure 6.6, line 13). In this situation, the leader election protocol elects a new leader. Byzantine learners or proposers cannot prevent the election: even if the f faulty learners pretend to have learned a value, the remaining correct proposers form a quorum and thus can trigger an election (see Section 6.6.3).

Since the number of proposers p is larger than f, eventually either some value is stable or a correct leader α is elected. In a period of synchrony, Byzantine proposers alone cannot trigger an election to replace a correct leader (see Section 6.6.3). We show that if α is correct then some value will be stable.

The correct leader will gather a progress certificate (Figure 6.8, line 108) and propose a value to all the acceptors. By construction, all progress certificates vouch for at least one value—and correct acceptors will accept a value vouched by a progress certificate. Since α is correct, it will propose the same value to all acceptors and all $a - f$ correct acceptors will accept the proposed value. Given that $a > 3f$, $\lceil (a + f + 1)/2 \rceil \leq a - f$, so by definition that value will be chosen.

The end-to-end retransmission protocol (Figure 6.6, line 4) ensures that α will continue to resend its proposed value at least until it hears from $\lceil (l + f + 1)/2 \rceil$ learners that they have learned a value—that is, until the value is stable (line 8). □

Theorem 14 (CL1)**.** *Some proposed value is eventually chosen.*

Proof. By Lemma 46 eventually some value is stable, i.e. $\lceil (l + f + 1)/2 \rceil > f$ correct learners have learned it. By CS3 a correct learner only learns a value after it is chosen. Therefore, the stable value is chosen. □

Our proof for CL1 only relies on the fact that the correct leader does not stop retransmission until a value is chosen. In practice, it is desirable for the leader to stop retransmission once a value is chosen. Since $l > 3f$, there are at least $\lceil (l + f + 1)/2 \rceil$ correct learners, so eventually all correct proposers will be satisfied (line 8) and the leader will stop retransmitting (line 4).

Theorem 15 (CL2)**.** *Once a value is chosen, correct learners eventually learn it.*

Proof. By Lemma 46, some value v is eventually stable, i.e. $\lceil (l - f + 1)/2 \rceil \geq f + 1$ correct learners eventually learn the value.

Even if the leader is not retransmitting anymore, the remaining correct learners can determine the chosen value when they query their peers with the "pull" requests (lines 34 and 36–38) and receive $f + 1$ matching responses (line 42). So eventually, all correct learners learn the chosen value. □

6.7 Parameterized FaB Paxos

Previous Byzantine consensus protocols require $3f + 1$ processes and may complete in three communication steps when there is no failure; FaB Paxos requires $5f + 1$ processes and may complete in two communication steps despite up to f failures—the protocol uses the additional replication for speed. In this section, we explore scenarios between these two extremes: when fewer than $5f+1$ processes are available or when it is not necessary to ensure two-step operation even when *all* f processes fail.

We generalize FaB Paxos by decoupling replication for fault-tolerance from replication for speed. The resulting protocol, Parameterized FaB Paxos (Figure 6.9) spans the whole design space between minimal number of processes (but no guar-

```
leader.onStart() :
    // proposing (PC is null unless recovering)
    repeatedly send ("PROPOSE",$value, number, PC$) to all acceptors
    until $|Satisfied| >= \lceil (p+f+1)/2 \rceil$

leader.onElected(newnumber,proof) :
    $pnumber$ := $newnumber$ // no smaller than previous pnumber
    if (not leader for $pnumber$) : return
    repeatedly send ("QUERY",$pnumber, proof$) to all acceptors
    until receive ("REP", $\langle value_j, pnumber, commit\_proof_j, j \rangle_j$) ) from
        $a - f$ acceptors $j$
    $PC$ := the union of these replies
    if ∃ v' s.t. vouches-for($PC, v', pnumber$) : $value := v'$
    onStart()

proposer.onLearned() : from learner $l$
    $Learned := Learned \cup \{l\}$
    if $|Learned| >= \lceil (l+f+1)/2 \rceil$ :
        send ("SATISFIED") to all proposers

proposer.onStart():
    wait for timeout
    if $|Learned| < \lceil (l+f+1)/2 \rceil$ :
        suspect the leader

proposer.onSatisfied(): from proposer $x$
    $Satisfied := Satisfied \cup \{x\}$

acceptor.onPropose(value,pnumber,progcert) : from leader
    if $pnumber \neq$ leader-election.getRegency() :
        return // only listen to current leader
    if accepted $(v, pn)$ and (($pnumber <= pn$) or (($v \neq value$)
    and not vouches-for(progcert,value,pnumber))) :
        return // only change with progress certificate
    $accepted := (value, number)$ // accepting
    send ("ACCEPTED",$accepted$) to all learners
    // i is the number of this acceptor
    send $\langle$"ACCEPTED",$value, pnumber, i \rangle_i$ to all acceptors

acceptor.onAccepted(value,pnumber,j) : signed by acceptor $j$
    if $pnumber > tentative\_commit\_proof[j].pnumber$ :
        $tentative\_commit\_proof[j] := \langle$"ACCEPTED",$value, pnumber, j \rangle_j$
    if valid(tentative_commit_proof,value,leader-election.getRegency()) :
        $commit\_proof := tentative\_commit\_proof$
        send ("COMMITPROOF",$commit\_proof$) to all learners
```

Figure 6.9: Parameterized FaB Paxos with recovery (part 1)

```
acceptors.onQuery(pn,proof) : from proposer
    leader-election.consider(proof)
    if (leader-election.getRegency() $\neq pn$) :
        return // ignore bad requests
    $leader$ := leader-election.getLeader()
    send (“REP”, $\langle accepted.value, pn, commit\_proof, i\rangle_i$) to leader

learner.onAccepted(value,pnumber) : from acceptor ac
    $accepted[ac] := (value, pnumber)$
    if there are $\lceil (a + 3f + 1)/2 \rceil$ acceptors $x$
    such that $accepted[x] == (value, pnumber)$ :
        learn(value,pnumber) // learning

learner.onCommitProof(commit_proof) : from acceptor $ac$
    $cp[ac] := commit\_proof$
    $(value, pnumber) := accepted[ac]$
    if there are $\lceil (a + f + 1)/2 \rceil$ acceptors $x$
    such that valid($cp[x], value, pnumber$) :
        learn(value,pnumber) // learning

learner.learn(value,pnumber) :
    $learned := (value, pnumber)$ // learning
    send (“LEARNED”) to all proposers

learner.onStart() :
    wait for timeout
    while (not learned) send (“PULL”) to all learners

learner.onPull() : from learner $ln$
    if this process learned some pair $(value, pnumber)$ :
        send (“LEARNED”,$value, pnumber$) to $ln$

learner.onLearned(value,pnumber) : from learner $ln$
    $learn[ln] := (value, pnumber)$
    if there are $f + 1$ learners $x$
    such that $learn[x] == (value, pnumber)$ :
        $learned := (value, pnumber)$ // learning

valid(commit_proof,value,pnumber) :
    $c := commit\_proof$
    if there are $\lceil (a + f + 1)/2 \rceil$ distinct values of $x$ such that
        $(c[x].value == value) \wedge (c[x].pnumber == pnumber)$
    : return true
    else return false

vouches-for(PC,value,pnumber) :
    if there exist $\lceil (a - f + 1)/2 \rceil$ $x$ such that
        all $PC[x].value == d$
        $\wedge\ d \neq value$
    : return false
    if there exists $x$, $d \neq value$ such that
        valid($PC[x].commit\_proof, d, pnumber$)
    : return false
    return true
```

Figure 6.9: Parameterized FaB Paxos with recovery (part 2)

antee of two-step executions) and two-step protocols (that require more processes). This trade-off is expressed through the new parameter t ($0 \leq t \leq f$). Parameterized FaB Paxos requires $3f + 2t + 1$ processes, is safe despite up to f Byzantine failures (it is live as well in periods of synchrony), and all its executions are two-step in the common case despite up to t Byzantine failures: the protocol is *(t,2)-step*. FaB Paxos is just a special case of Parameterized FaB Paxos, with $t = f$.

Several choices of t and f may be available for a given number of machines. For example, if seven machines are available, an administrator can choose between tolerating two Byzantine failures and slowing down after the first failure ($f = 2, t = 0$) or tolerating only one Byzantine failure but maintaining two-step operation despite the failure ($f = 1, t = 1$).

The key observation behind this protocol is that FaB Paxos maintains safety even if $n < 5f + 1$ (provided that $n > 3f$). It is only liveness that is affected by having fewer than $5f+1$ acceptors: even a single crash may prevent the learners from learning (the predicate at line 27 of Figure 6.6 would never hold). In order to restore the liveness property even with $3f < n < 5f+1$, we merge a traditional BFT three-phase-commit [33] with FaB Paxos. While merging the two, we take special care to ensure that the two features never disagree as to which value should be learned. The Parameterized FaB Paxos code does not include any mention of the parameter t: if there are more than t failures, then the two-step feature of Parameterized FaB Paxos may never be triggered because there are not enough correct nodes to send the required number of messages.

First, we modify acceptors so that, after receiving a proposal, they sign it (including the proposal number) and forward it to each other so each of them can collect a commit proof. A *commit proof* for value v at proposal number *pn* consists of $\lceil (a+f+1)/2 \rceil$ statements from different acceptors that accepted value v for proposal number pn (function **valid(. . .)**, line 284). The purpose of commit proofs is to give evidence for which value was chosen. If there is a commit proof for value v at proposal pn, then no other value can possibly have been chosen for proposal pn. We include commit proofs in the progress certificates (line 252) so that newly elected leaders have all the necessary information when deciding which value to propose. The commit proofs are also forwarded to learners (line 245) to guarantee liveness when more than t acceptors fail.

Second, we modify learners so that they learn a value if enough acceptors have a commit proof for the same value and proposal number (line 263).

Finally, we redefine "chosen" and "progress certificate" to take commit proofs into account.

We now say that value v is *chosen for* proposal number pn if $\lceil (a+f+1)/2 \rceil$ correct acceptors have accepted v in proposal pn or if $\lceil (a+f+1)/2 \rceil$ acceptors have (or had) a commit proof for v and proposal number pn. Learners learn v when they know v has been chosen. The protocol ensures that only a single value may be chosen.

Progress certificates still consist of $a-f$ entries, but each entry now contains an additional element: either a commit proof or a signed statement saying that the corresponding acceptor has no commit proof. A progress certificate *vouches* for value v' at proposal number pn if all entries have proposal number pn, there is no value $d \neq v'$ contained $\lceil (a-f+1)/2 \rceil$ times in the progress certificate, and the progress certificate does not contain a commit proof for any value $d \neq v'$ (function **vouches-for**(. . .), line 291). The purpose of progress certificates is, as before, to allow learners to convince acceptors to change their accepted value.

These three modifications maintain the properties that at most one value can be chosen and that, if some value was chosen, then future progress certificates will vouch only for it. This ensures that the changes do not affect safety. Liveness is maintained despite f failures because there are at least $\lceil (a + f + 1)/2 \rceil$ correct acceptors, so, if the leader is correct, then eventually all of them will have a commit proof, thus allowing the proposed value to be learned. The next section develops these points in more detail.

6.7.1 Correctness

The proof that Parameterized FaB Paxos implements consensus follows the same structure as that for FaB.

Theorem 16 (CS1)**.** *Only a value that has been proposed may be chosen.*

Proof. To be chosen, a value must be accepted by a set of correct acceptors (by definition), and correct acceptors only accept values that are proposed (line 229). $\square$

The proof for CS2 follows a similar argument as the one in Section 6.6.4. We first consider values chosen for the same proposal number, then we show that once a value v is chosen, later proposals also propose v. Parameterized FaB Paxos uses a different notion of chosen, so we must show that a value, once chosen, remains so if no correct node accepts new values.

Lemma 47. *If value v is chosen for proposal number pn, then it was accepted by $\lceil (a + f + 1)/2 \rceil$ acceptors in proposal pn.*

Proof. The value can be chosen for two reasons according to the definition: either $\lceil (a + f + 1)/2 \rceil$ correct acceptors accepted it (in which case the lemma follows directly), or because $\lceil (a + f + 1)/2 \rceil$ acceptors have a commit proof for v at pn. At least one of them is correct, and a commit proof includes answers from $\lceil(a+f+1)/2\rceil$ acceptors who accepted v at pn (lines 243 and 286–289). □

Corollary 1. *For every proposal number pn, at most one value is chosen.*

Proof. If two values were chosen, then the two sets of acceptors who accepted them intersect in at least one correct acceptor. Since correct acceptors only accept one value per proposal number (line 232), the two values must be identical. □

Corollary 2. *If v is chosen for proposal pn and no correct acceptor accepts a different value for proposals with a higher number than pn, then v is the only value that can be chosen for any proposal number higher than pn.*

Proof. Again, the two sets needed to choose distinct v and v' would intersect in at least a correct acceptor. Since by assumption these correct acceptors did not accept a different value after pn, $v = v'$. □

Lemma 48. *If v is chosen for pn then every progress certificate PC for a higher proposal number pn' either vouches for no value, or vouches for value v.*

Proof. Suppose that the value v is chosen for pn. The higher-numbered progress certificate PC will be generated in lines 209–212 by correct proposers. We show that all progress certificates for proposal numbers $pn' > pn$ that vouch for a value vouch for v (we will show later that in fact all progress certificates from correct proposers vouch for at least one value).

The value v can be chosen for pn for one of two reasons. In each case, the progress certificate can only vouch for v.

First, v could be chosen for pn because there is a set A of $\lceil(a + f + 1)/2\rceil$ correct acceptors that have accepted v for proposal pn. The progress certificate for

pn', PC, consists of answers from $a - f$ acceptors (line 210). These answers are signed so each answer in a valid progress certificate come from a different acceptor. Since acceptors only answer higher-numbered requests (line 249; regency numbers never decrease), all nodes in A that answered have done so after having accepted v in proposal pn. At most f acceptors may be faulty, so PC includes at least $\lceil (a - f + 1)/2 \rceil$ answers from A. By definition, it follows that PC cannot vouch for any value other than v (lines 292–295).

Second, v could be chosen for pn because there is a set B of $\lceil (a + f + 1)/2 \rceil$ acceptors that have a commit proof for v for proposal pn. Again, the progress certificate PC for pn' includes at least $\lceil (a - f + 1)/2 \rceil$ answers from B. Up to f of these acceptors may be Byzantine and lie (pretending to never have seen v), so PC may contain as few as $\lceil (a - 3f + 1)/2 \rceil$ commit proofs for v. Since $a > 3f$, PC contains at least one commit proof for v, which by definition is sufficient to prevent PC from vouching for any value other than v (lines 296–297). □

Lemma 49. *If v is chosen for pn then v is the only value that can be chosen for any proposal number higher than pn.*

Proof. In order for a different value v' to be chosen, a correct acceptor would have to accept a different value in a later proposal (Corollary 2). Correct acceptors only accept a new value v' if it is accompanied with a progress certificate that vouches for v' (lines 232–234). The previous lemma shows that no such progress certificate can be gathered. □

Theorem 17 (CS2)**.** *Only a single value may be chosen.*

Proof. Putting it all together, we can show that CS2 holds (by contradiction). Suppose that two distinct values, v and v', are chosen. By Corollary 1, they must have been chosen in distinct proposals pn and pn'. Without loss of generality, suppose $pn < pn'$. By Lemma 49, $v' = v$. $\square$

Theorem 18 (CS3)**.** *Only a chosen value may be learned by a correct learner.*

Proof. Suppose that a correct learner learns value v after observing that v is chosen for pn. There are three ways for a learner to make that observation in Parameterized FaB Paxos.

- $\lceil (a + 3f + 1)/2 \rceil$ acceptors reported having accepted v for proposal pn (line 256). At least $\lceil (a + f + 1)/2 \rceil$ of these acceptors are correct, so by definition v was chosen for pn.
- $\lceil (a + f + 1)/2 \rceil$ acceptors reported a commit proof for v for proposal pn (lines 263–265). By definition, v was chosen for pn.
- $f + 1$ other learners reported that v was chosen for pn (lines 280–282). One of these learners is correct—so, by induction on the number of learners, it follows that v was indeed chosen for pn.

$\square$

Lemma 50. *All valid progress certificates vouch for at least one value.*

Proof. The definition allows for three ways for a progress certificate PC to vouch for no value at all. We show that none can happen in our protocol.

First, PC could vouch for no value if there were two distinct values v and v', each contained $\lceil (a - f + 1)/2 \rceil$ times in the PC. This is impossible because PC only contains $a - f$ entries in total (line 211).

Second, PC could vouch for no value if it contained two commit proofs for distinct values v and v'. Both commit proofs contain $\lceil (a+f+1)/2 \rceil$ identical entries (for v and v' respectively) from the same proposal (lines 286–287). These two sets intersect in a correct proposer, but correct proposers only accept one value per proposal number (line 232). Thus, it is not possible for PC to contain two commit proofs for distinct values.

Third, there could be some value v contained $\lceil (a - f + 1)/2 \rceil$ times in PC, and a commit proof for some different value v'. The commit proof includes values from $\lceil (a + f + 1)/2 \rceil$ acceptors, and at least $\lceil (a - f + 1)/2 \rceil$ of these are honest so they would report the same value (v') in PC. But $\lceil (a - f + 1)/2 \rceil$ is a majority and there can be only one majority in PC, so that scenario cannot happen. □

Recall that a value is stable if it is learned by $\lceil (l - f + 1)/2 \rceil$ correct learners. We use Lemma 46, which shows that some value is eventually stable, to prove CL1 and CL2.

Theorem 19 (CL1)**.** *Some proposed value is eventually chosen.*

Theorem 20 (CL2)**.** *Once a value is chosen, correct learners eventually learn it.*

Proof. The proofs for CL1 and CL2 are unchanged. They still hold because although the parameterized protocol makes it easier for a value to be chosen, it still has the property that the leader will resend its value until it knows that the value is stable (lines 203–204, 216–219). A value that is stable is chosen (ensuring CL1) and it has been learned by at least $\lceil (l - f + 1)/2 \rceil$ correct learners (ensuring CL2 because of the pull subprotocol on lines 267–282). □

6.8 Three-Step State Machine Replication

Fast consensus translates directly into fast state machine replication: in general, state machine replication requires one fewer round with FaB Paxos than with a traditional three-round Byzantine consensus protocol.

A straightforward implementation of Byzantine state machine replication on top of FaB Paxos requires only four rounds of communication—one for the clients to send requests to the proposers; two (rather than the traditional three) for the learners to learn the order in which requests are to be executed; and a final one, after the learners have executed the request, to send the response to the appropriate clients. FaB can accommodate existing leader election protocols (e.g. [33]).

The number of rounds of communication can be reduced down to three using *tentative execution* [33, 76], an optimization used by Castro and Liskov for their PBFT protocol that applies equally well to FaB Paxos. As shown in Figure 6.10, learners tentatively execute clients' requests as supplied by the leader before consensus is reached. The acceptors send to both clients and learners the information required to determine the consensus value, so clients and learners can at the same time determine whether their trust in the leader was well put. In case of conflict, tentative executions are rolled back and the requests are eventually re-executed in the correct order.

FaB Paxos loses its edge over PBFT, however, in the special case of read-only requests that are not concurrent with any read-write request. In this case, a second optimization proposed by Castro and Liskov allows both PBFT and FaB Paxos to service these requests using just two rounds.

The next section shows further optimizations that reduce the number of learners and allow nodes to recover.

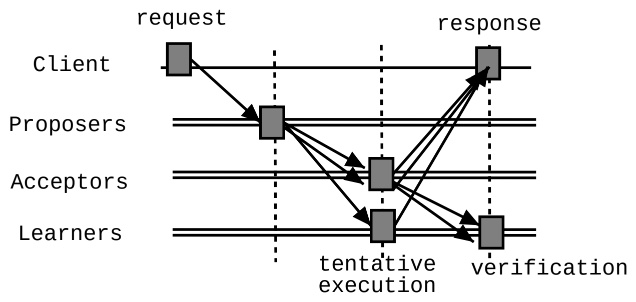

Figure 6.10: FaB Paxos state machine with tentative execution.

6.9 Optimizations

6.9.1 $2f + 1$ Learners

Parameterized FaB Paxos (and consequently FaB Paxos, its instantiation for $t = f$) requires $3f + 1$ learners. We now show how to reduce the number of learners to $2f + 1$ without delaying consensus. This optimization requires some communication and the use of signatures in the common case, but still reaches consensus in two communication steps in the common case.

In order to ensure that all correct learners eventually learn, Parameterized FaB Paxos uses two techniques. First, the retransmission part of the protocol ensures that $\lceil (l+f+1)/2 \rceil$ learners eventually learn the consensus value (line 218) and allows the remaining correct learners to pull the learned value from their up-to-date peers (lines 267–282).

To adapt the protocol to an environment with only $2f + 1$ learners, we first modify retransmission so that proposers enter the *satisfied* state with $f + 1$ acknowledgments from learners—retransmission may now stop when only a single correct learner knows the correct response.

Second, we have to modify the "pull" mechanism because now a single correct learner must be able to convince other learners that its reply is correct. We therefore strengthen the condition under which we call a value stable (line 204) by adding information in the acknowledgments sent by the learners. In addition to the client's request and reply obtained by executing that request, acknowledgments must now also contain $f + 1$ signatures from distinct learners that verify the same reply.

After learning a value, learners now sign their acknowledgment and send that signature to all learners, expecting to eventually receive $f+1$ signatures that verify their acknowledgment. Since there are $f + 1$ correct learners, each is guaranteed to be able to eventually gather an acknowledgment with $f + 1$ signatures that will satisfy the leader's stability test. Thus, after the leader determines that its proposal is stable, at least one of the learners that sent a valid acknowledgment is correct and will support the pull subprotocol: learners query each other, and eventually all correct learners receive the valid acknowledgment and learn the consensus value. This exchange of signatures takes an extra communication step, but this step is not in the critical path: it occurs *after* learners have learned the value.

The additional messages are also not in the critical path when this consensus protocol is used to implement a replicated state machine: the learners can execute the client's operation immediately when learning the operation, and can send the result to the client without waiting for the $f+1$ signatures. Clients can already distinguish between correct and incorrect replies since only correct replies are vouched for by $f + 1$ learners.

6.9.2 Rejoin

By allowing repaired servers (for example, a crashed node that was rebooted) to rejoin, the system can continue to operate as long as at all times no more than f servers are either faulty or rejoining. The rejoin protocol must restore the replica's state, and as such it is different depending on the role that the replica plays.

The only state in proposers is the identity of the current leader. Therefore, a joining proposer queries a quorum of acceptors for their current proof-of-leadership and adopts the largest valid response.

Acceptors must never accept two different values for the same proposal number. In order to ensure that this invariant holds, a rejoining acceptor queries the other acceptors for the last instance of consensus d, and it then ignores all instances until $d + k$ (k is the number of instances of consensus that may run in parallel). Once the system moves on to instance $d + k$, the acceptor has completed its rejoin.

The state of the learners consists of the ordered list of operations. A rejoining learner therefore queries other learners for that list. It accepts answers that are vouched by $f + 1$ learners (either because $f + 1$ learners gave the same answer, or in the case of $2f + 1$ Parameterized FaB Paxos a single learner can show $f + 1$ signatures with its answer). Checkpoints could be used for faster state transfer as has been done before [33, 84].

6.10 Privacy Firewall

Traditional BFT systems face a fundamental tradeoff between increasing availability and integrity on the one hand and strengthening confidentiality on the other. Increasing diversity across replicas (e.g., increasing the number of replicas or increasing the heterogeneity across implementations of different replicas [10, 77, 96, 156]) improves integrity and availability because it reduces the chance that too many replicas simultaneously fail. Unfortunately, it also increases the chance that at least one replica contains an exploitable bug. If an attacker manages to compromise one

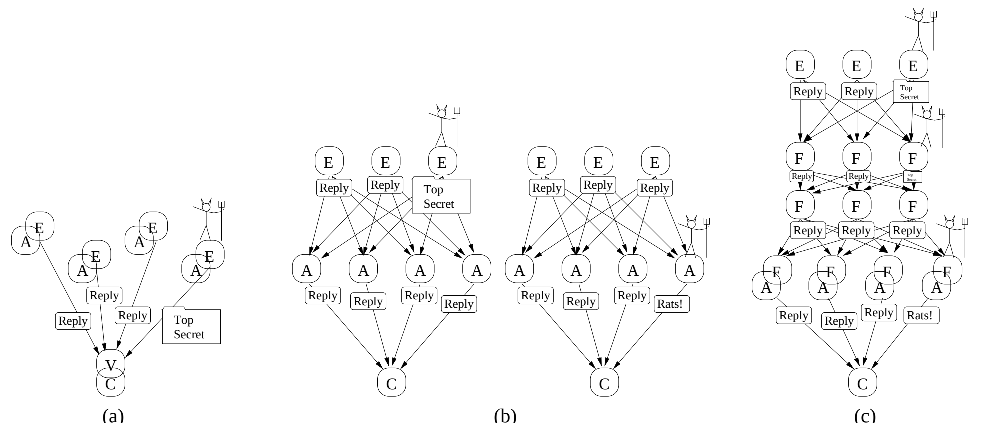

Figure 6.11: Illustration of confidentiality filtering properties of (a) traditional BFT architectures, (b) architectures that separate agreement and execution, and (c) architectures that separate agreement and execution and that add additional Privacy Firewall nodes.

replica in such a system, the compromised replica may send confidential data back to the attacker.

Compounding this problem, as Figure 6.11(a) illustrates, traditional replicated state machine architectures delegate the responsibility of combining the state machines' outputs to a voter at the client. Fate sharing between the client and the voter ensures that the voter does not introduce a new single point of failure; to quote Schneider [139], *"the voter—a part of the client—is faulty exactly when the client is, so the fact that an incorrect output is read by the client due to a faulty voter is irrelevant"* because a faulty voter is then synonymous with a faulty client. But a Byzantine client can ignore the voter and talk directly with a compromised replica.

Solving this problem seems difficult. If we move the voter away from the client, we lose fate sharing, and the voter becomes a single point of failure. It is not clear how to replicate the voter to eliminate this single point of failure without miring ourselves in endless recursion ("Who votes on the voters?").

As illustrated by Figure 6.11(b), the separation of agreement from execution provides the opportunity to reduce a system's vulnerability to compromising confidentiality by having the agreement nodes filter incorrect replies before sending reply certificates to clients. It now takes a failure of both an agreement node and an execution node to compromise privacy if we restrict communications so that (1) clients can communicate with agreement nodes but not execution nodes and (2) request and reply bodies are encrypted so that clients and execution nodes can read them but agreement nodes cannot. In particular, if all agreement nodes are correct, then the agreement nodes can filter replies so that only correct replies reach the client. Conversely, if all execution nodes are correct, then faulty agreement nodes can send information to clients, but not information regarding the confidential state of the state machine.

Although this simple design improves confidentiality, it is not entirely satisfying. First, it can not handle multiple faults: a single fault in both the agreement and execution clusters can allow confidential information to leak. Second, it allows an adversary to leak information via a covert channel, for instance by manipulating membership sets in agreement certificates.

In the rest of this section, we describe a general confidentiality filter architecture—the *Privacy Firewall*. If the agreement and execution clusters have a sufficient number of working machines, then a Privacy Firewall of $h + 1$ rows of $h + 1$ filters per row can tolerate up to h faults while still providing availability, integrity, and confidentiality. We first define the protocol, then explain the rationale behind specific design choices. Finally, we state the end-to-end confidentiality guarantees provided by the system, highlight the limitations of these guarantees, and discuss ways to strengthen these guarantees.

6.10.1 Protocol Definition

Figure 6.11(c) shows the organization of the privacy firewall. We insert *filter nodes* F between execution servers E and agreement nodes A to pass only information sent by correct execution servers. Filter nodes are arranged into an array of $h+1$ rows of $h+1$ columns; if the number of agreement nodes is at least $h+1$, then the bottom row of filters can be merged with the agreement nodes by placing a filter on each server in the agreement cluster. Information flow is controlled by restricting communication to only the links shown in Figure 6.11(c). Each filter node has a physical network connection to all filter nodes in the rows above and below but no other connections. Request and reply bodies are encrypted so that the client and execution nodes can read them but agreement nodes and firewall nodes cannot.

Each filter node maintains $maxN$, the maximum sequence number in any valid agreement certificate or reply certificate seen, and $state_n$, information relating to sequence number n. $State_n$ contains *null* if request n has not been seen, contains *seen* if request n has been seen but reply n has not, and contains a reply certificate if reply n has been seen. Nodes limit the size of $state_n$ by discarding any entries whose sequence number is below $maxN - P$ where P is the pipeline depth that bounds the number of requests that the agreement cluster can have outstanding (see Section 6.3.1).

When a filter node receives from below a valid request certificate and agreement certificate with sequence number n, it ignores the request if $n < maxN - P$. Otherwise, it first updates $maxN$ and discards entries in $state$ with sequence numbers smaller than $maxN - P$. Finally it sends one of two messages. If $state_n$ contains a reply certificate, the node multicasts the stored reply to the row of filter nodes or agreement nodes below. But, if $state_n$ does not yet contain the reply, the node sets

$state_n = seen$ and multicasts the request and agreement certificates to the next row above. As an optimization, nodes in all but the top row of filter nodes can unicast these certificates to the one node above them rather than multicasting.

Although request and agreement certificates flowing up can use any form of certificate including MAC-based authenticators, filter nodes must use threshold cryptography [43] for reply certificates they send down. When a filter node in the top row receives $g + 1$ partial reply certificates signed by different execution nodes, it assembles a complete reply certificate authenticated by a single threshold signature representing the execution nodes' split group key. Then, after a top-row filter node assembles such a complete reply certificate with sequence number n or after any other filter node receives and cryptographically validates a complete reply certificate with sequence number n, the node checks $state_n$. If $n < maxN - P$ then the reply is too old to be of interest, and the node drops it; if $state_n = seen$, the node multicasts the reply certificate to the row of filter nodes or agreement nodes below and then stores the reply in $state_n$; or if $state_n$ already contains the reply or is empty, the node stores reply certificate n in $state_n$ but does not multicast it down at this time.

The protocol described above applies to any deterministic state machine. We describe in Section 6.3.1 how agreement nodes pick a timestamp and random bits to obliviously transform non-deterministic state machines into deterministic ones without having to look at the content of requests. Note that this approach may allow agreement nodes to infer something about the internal state of the execution cluster, and balancing confidentiality and non-determinism in its full generality appears hard. To prevent the agreement cluster from even knowing what random value is used by the execution nodes, execution nodes could cryptographically hash the input value with a secret known only to the execution cluster; we have not yet implemented this feature. Still, a compromised agreement node can determine the time that a request enters the system, but as Section 6.10.3 notes, that information is already available to agreement nodes.

6.10.2 Design Rationale

The Privacy Firewall architecture provides confidentiality through the systematic application of three key ideas: (1) redundant filters to ensure filtering in the face of failures, (2) elimination of non-determinism to prevent explicit or covert communication through a correct filter, and (3) restriction of communication to enforce filtering of confidential data sent from the execution nodes.

Redundant filters

The array of $h + 1$ rows of $h + 1$ columns ensures the following two properties as long as there are no more than h failures: (i) there exists at least one *correct path* between the agreement nodes and execution nodes consisting only of correct filters and (ii) there exists one row (the *correct cut*) consisting entirely of correct filter nodes.

Property (i) ensures availability by guaranteeing that requests can always reach execution nodes and replies can always reach clients. Observe that availability is also necessary for preserving confidentiality, because a strategically placed rejected request could be used to communicate confidential information by introducing a termination channel [136].

Property (ii) ensures a faulty node can either access confidential data or communicate freely with clients but not both. Faulty filter nodes above the correct cut might have access to confidential data, but the filter nodes in the correct cut ensure that only replies that would be returned by a correct server are forwarded. And, although faulty nodes below the correct cut might be able to communicate any information they have, they do not have access to confidential information.

Eliminating non-determinism

Not only does the protocol ensure that a correct filter node transmits only correct replies (vouched for by at least $g + 1$ execution nodes), it also eliminates nondeterminism that an adversary could exploit as a covert channel by influencing nondeterministic choices.

The contents of each reply certificate is a deterministic function of the request and sequence of preceeding requests. The use of threshold cryptography makes the encoding of each reply certificate deterministic and prevents an adversary from leaking information by manipulating certificate membership sets. The separation of agreement from execution is also crucial for confidentiality: agreement nodes outside the Privacy Firewall assign sequence numbers so that the non-determinism in sequence number assignment cannot be manipulated as a covert channel for transmitting confidential information.

In addition to these restrictions to eliminate non-determinism in message bodies, the system also restricts (but as Section 6.10.3 describes, does not completely eliminate) non-determinism in the network-level message retransmission. The per-request *state* table allows filter nodes to remember which requests they have seen and send at most one (multicast) reply per request message. This table reduces the ability of a compromised node to affect the number of copies of a reply certificate that a downstream firewall node sends.

Restricting communication

The system restricts communication by (1) physically connecting firewall nodes only to the nodes directly above and below them and (2) encrypting the bodies of requests and replies. The first restriction enforces the requirement that all communication between execution nodes and the outside world flow through at least one correct firewall. The second restriction prevents nodes below the correct cut of firewall nodes from accumulating and revealing confidential state by observing the bodies of requests and replies.

6.10.3 Filter Properties and Limitations

There are $h + 1$ rows of firewall nodes, so there exists a row, the *correct cut*, that consists entirely of correct firewall nodes. All information sent by the execution servers pass through the correct cut,[3] and the correct cut provides *output set confidentiality* in that any sequence of outputs of our correct cut is also a legal sequence of outputs of a correct unreplicated implementation of the service accessed via an asynchronous unreliable network that can discard, delay, replicate, and reorder messages. More formally, suppose that C is a correct unreplicated implementation of a service, S_0 is the abstract [130] initial state, I is a sequence of input requests, and O the resulting sequence of output replies transmitted on an asynchronous unreliable network to a receiver. The network can delay, replicate, reorder, and discard these replies; thus the receiver observes an output sequence O' that belongs to a set of output sequences $\mathcal{O}$, where each O' in $\mathcal{O}$ includes only messages from O.

The correct cut of our replicated system is output set confidential with respect to C, so given the same initial abstract state S_0 and input I its output O''

[3]We are of course only considering the information that the adversary can capture, namely information sent over network links.

also belongs to $\mathcal{O}$. The output set confidentiality property is guaranteed because the correct firewall nodes only let messages through if they have a valid signature from the execution cluster. This signature can only be created by combining threshold signatures from more than f execution nodes—so a correct execution node must have sent this answer, and correct execution nodes execute requests in the same order as C and return the same responses (Lemma 40). Also, every reply from these correct nodes will reach the correct cut because there are $h+1$ columns in the Privacy Firewall, so one of them consists entirely of correct nodes that will forward the messages from the execution cluster to the correct cut.

Because our system replicates arbitrary state machines, the above definition describes confidentiality with respect to the behavior of a single correct server's state machine. The definition does not specify anything about the internal behaviors of or the policies enforced by the replicated state machine, so it is more flexible and less strict than the notion of *non-interference* [136], which is sometimes equated with information flow security in the literature and which informally states that the observable output of the system has no correlation to the confidential information stored in the system. Our output set confidentiality guarantee is similar in spirit to the notion of possibilistic non-interference [114], which characterizes the behavior of a nondeterministic program by the *set* of possible results and requires that the set of possible observable outputs of a system be independent of the confidential state stored in the system.

A limitation is that although agreement nodes do not have access to the body of requests, they do need access to the identity of the client (in order to buffer information about each client's last request), the arrival times of the requests and replies, and the encrypted bodies of requests and replies. Faulty agreement or filter nodes in our system could leak information regarding traffic patterns. For example, a malicious agreement node could leak the frequency that a particular client sends requests to the system or the average size of a client's requests. Techniques such as forwarding through intermediaries and padding messages can reduce a system's vulnerability to traffic analysis [34], though forwarding can add significant latencies and significant message padding may be needed for confidentiality [150].

Also note that although output set confidentiality ensures that the set of output messages is a deterministic function of the sequence of inputs, the nondeterminism in the timing, ordering, and retransmission of messages might be manipulated to create covert channels that communicate information from above the correct cut to below it (known as *timing channels* [44]). For example, a compromised node directly above the correct cut of firewalls might attempt to influence the timing or sequencing of replies forwarded by the nodes in the correct cut by forwarding replies more quickly or less quickly than its peers, sending replies out of order, or varying the number of times it retransmits a particular reply. Given that any resulting output sequence and timing is a "legal" output that could appear in an asynchronous system with a correct server and an unreliable network, it appears fundamentally difficult for firewall nodes to completely eliminate such channels.

It may, however, be possible to systematically restrict such channels by engineering the system to make it more difficult for an adversary to affect the timing, ordering, and replication of symbols output by the correct cut. The use of the *state* table to ensure that each reply is multicast at most once per request received is an example of such a restriction. This rule makes it more difficult for a faulty node to encode information in the number of copies of a reply sent through the correct cut and approximates a system where the number of times a reply is sent is a deterministic function of the number of times a request is sent. But, with an asynchronous unreliable network, this approximation is not perfect—a faulty firewall node can still slightly affect the probability that a reply is sent and therefore can slightly affect the expected number of replies sent per request (e.g., not sending a reply slightly increases the probability that all copies sent to a node in the correct cut are dropped; sending a reply multiple times might slightly reduce that probability). Also note that for simplicity the protocol described above does not use any additional heuristic to send replies in sequence number order, though similar processing rules could be added to make it more difficult (though not impossible in an asynchronous system) for a compromised node to cause the correct cut to have gaps or reorderings in the sequence numbers of replies it forwards.

Restricting the nondeterminism introduced by the network seems particularly attractive when a firewall network is deployed in a controlled environment such as a machine room. For example, if the network can be made to deliver messages reliably and in order between correct nodes, then the correct cut's output sequence can always follow sequence number order. In the limit, if timing and message-delivery nondeterminism can be completely eliminated, then covert channels that exploit network nondeterminism can be eliminated as well. We conjecture that a variation of the protocol can be made perfectly confidential if agreement nodes and clients continue to operate under the asynchronous model but execution and firewall nodes operate under a synchronous model with reliable message delivery and a time bound on state machine processing, firewall processing, and message delivery between correct nodes. This protocol variation extends the *state* table to track when requests arrive and uses this information and system time bounds to restrict when replies are transmitted. As long as the time bounds are met by correct nodes and links between correct nodes, the system should be fully confidential with respect to a single correct server in that the only information output by the correct cut of firewall nodes is the information that would be output by a single correct server. If, on the other hand, the timing bounds are violated, then the protocol continues to be safe and live and provides output set confidentiality.

6.10.4 Optimality

We show that the number of nodes in the Privacy Firewall is minimal, even if one were to consider a different topology. In the following proof, we model the Privacy Firewall as a graph with two additional nodes, A and E. The graph with A and E forms a single connected component.

Lemma 51. *If the Privacy Firewall is safe despite h Byzantine failures, then the shortest path from A to E through the Privacy Firewall has length at least $h+1$.*

Proof. If the shortest path through the Privacy Firewall has length h or less, then all the nodes in the path may be Byzantine and they may forward confidential information from an execution node, violating the safety requirement. □

Lemma 52. *If the Privacy Firewall is live despite h Byzantine failures, then there is no set C of nodes of size h or less such that removing C from the graph would disconnect A from E.*

Proof. If the size of C is h or less, then if these nodes are Byzantine then they can block all messages between A and E, violating liveness of the Privacy Firewall. □

Theorem 21. *The smallest Privacy Firewall that is both safe and live has $(h+1)^2$ nodes.*

Proof. Label each node with its minimal distance to A. Let m be the length of the shortest path from A to E. The set C_l of nodes that have the same label l ($0 \leq l < m$) would disconnect A from E, because for every choice of l ($0 \leq l < m$), all paths from A to E contain a node with label l (that path must contain a node with label 1 and a node with label $m-1$, and the successor of node i on that path must have label $i+1$ or lower).

Each node has only one label, so the different C_l do not overlap. Each C_l must have size at least $h+1$ (Lemma 52). The value of m is at least $h+1$ (Lemma 51). So there are at least $h+1$ non-overlapping sets of size at least $h+1$: the graph must include at least $(h+1)^2$ nodes. $\square$

6.11 Evaluation

In this section, we experimentally evaluate the latency, overhead, and throughput of our prototype system under microbenchmarks. We also examine the system's performance acting as a network file system (NFS) server.

6.11.1 Prototype Implementation

We have constructed a prototype system that separates agreement and replication and that optionally provides a Privacy Firewall. As described above, our prototype implementation builds on Rodrigues et al.'s BASE library [130].

Our evaluation cluster comprises seven 933Mhz Pentium-III and two 500MHz Pentium-III machines, each with 128MB of memory. The machines run Redhat Linux 7.2 and are connected by a 100 Mbit ethernet hub.

Note that three aspects of our configuration would not be appropriate for production use. First, both the underlying BASE library and our system store important persistent data structures in memory and rely on replication across machines to ensure this persistence [17, 31, 35, 95]. Unfortunately, the machines in our evaluation cluster do not have uninterruptible power supplies, so power failures are a potentially significant source of correlated failures across our system that could cause our current configuration to lose data. Second, our Privacy Firewall architecture assumes a network configuration that physically restricts communication paths between agreement machines, privacy filter machines, and execution machines. Our current configuration uses a single 100 Mbit ethernet hub and does not enforce these restrictions. We would not expect either of these differences to affect the results we report in this section. Third, to reduce correlated failures, the nodes should be running different operating systems and different implementations of the agreement, privacy, and execution cluster software. We only implemented these libraries once, and we use only one version of the application code.

6.11.2 Latency

Past studies have found that Byzantine fault tolerant state machine replication adds modest latency to network applications [31, 32, 130]. Here, we examine the same latency microbenchmark used in these studies. Under this microbenchmark, the application reads a request of a specified size and produces a reply of a specified size with no additional processing. We examine request/reply sizes of 40 bytes/40 bytes, 40 bytes/4 KB, and 4 KB/40 bytes.

Figure 6.12 shows the average latency (all run within 5%) for ten runs of 200 requests each. The bars show performance for different system configurations with the *algorithm/machine configuration/authentication algorithm* indicated in the legend. BASE/Same/MAC is the BASE library with 4 machines hosting both the

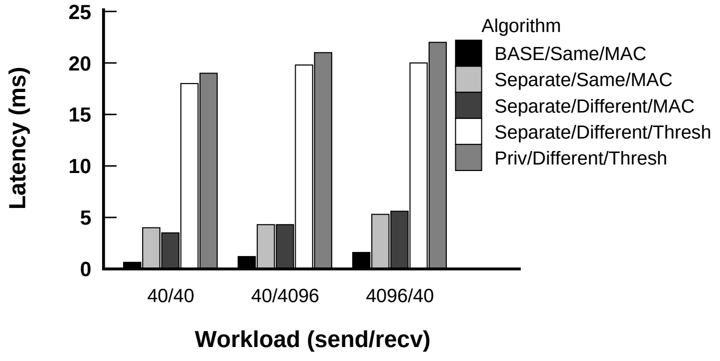

Figure 6.12: Latency for null-server benchmark for three request/reply sizes.

agreement and execution servers and using MAC authenticators; Separate/Same/-MAC shows our system that separates agreement and replication with agreement running on 4 machines and with execution running 3 of the same set of machines and using MAC authenticators; Separate/Different/MAC moves the execution servers to 3 machines physically separate from the 4 agreement servers; Separate/Different/-Thresh uses the same configuration but uses threshold signatures rather than MAC authenticators for reply certificates; finally, Priv/Different/Thresh adds an array of Privacy Firewall servers between the agreement and execution cluster with a bottom row of 4 Privacy Firewall servers sharing the agreement machines and an additional row of 2 firewall servers separate from the agreement and execution machines.

The BASE library imposes little latency on requests, with request latencies of 0.64ms, 1.2ms, and 1.6ms for the three workloads. Our current implementations of the library that separates agreement from replication has higher latencies when running on the same machines—4.0ms, 4.3ms, and 5.3ms. The increase is largely caused by two inefficiencies in our current implementation: (1) rather than using the agreement certificate produced by the BASE library, each of our message queue nodes generates a piece of a new agreement certificate from scratch, (2) in our current prototype, we do a full all-to-all multicast of the agreement certificate and request certificate from the agreement nodes to the execution nodes, of the reply certificate from the execution nodes to the agreement nodes, and (3) our system does not use hardware multicast. We have not implemented the optimizations of first having one node send and having the other nodes send only if a timeout occurs, and we have not implemented the optimization of clients sending requests directly to the execution nodes. However, we added the optimization that the execution nodes send their replies directly to the clients. Separating the agreement machines from the execution machines adds little additional latency. But, switching from MAC authenticator certificates to threshold signature certificates increases latencies to 18ms, 19ms, and 20ms for the three workloads. Adding two rows of Privacy Firewall filters (one of which is co-located with the agreement nodes) adds a few additional milliseconds.

As expected, the most significant source of latency in the architecture is public key threshold cryptography. Producing a threshold signature takes 15ms and verifying a signature takes 0.7ms on our machines. Two things should be noted to put these costs in perspective. First, the latency for these operations is comparable to I/O costs for many services of interest; for example, these latency costs are similar to the latency of a small number of disk seeks and are similar to or smaller than wide area network round trip latencies. Second, signature costs are expected to fall quickly as processor speeds increase; the increasing importance of distributed systems security may also lead to widespread deployment of hardware acceleration of encryption primitives. The FARSITE project has also noted that technology trends are making it feasible to include public-key operations as a building block for practical distributed services [4].

6.11.3 Throughput and Cost

Although latency is an important metric, modern services must also support high throughput [157]. Two aspects of the Privacy Firewall architecture pose challenges to providing high throughput at low cost. First, the Privacy Firewall architecture requires a larger number of physical machines in order to physically restrict communication. Second, the Privacy Firewall architecture relies on relatively high-overhead public key threshold signatures for reply certificates. Two factors mitigate these costs.

First, although the new architecture can increase the total number of machines, it also can reduce the number of application-specific machines required. Application-specific machines may be more expensive than generic machines both in terms of hardware (e.g., they may require more storage, I/O, or processing resources) and in terms of software (e.g., they may require new versions of application-specific software.) Thus, for many systems we would expect the application costs (e.g., the execution servers) to dominate. Like router and switch box costs today, agreement node and privacy filter boxes may add a relatively modest amount to overall system cost. Also, although filter nodes must run on $(h+1)^2$ nodes (and this is provably the minimal number to ensure confidentiality), even when the Privacy Firewall architecture is used, the number of machines is relatively modest when the goal is to tolerate a small number of faults. For example, to tolerate up to one failure among the execution nodes and one among either the agreement or privacy filter servers, the system would have four generic agreement and privacy filter machines, two generic privacy filter machines, and three application-specific execution machines. Finally, in configurations without the Privacy Firewall, the total number of machines is not necessarily increased since the agreement and execution servers can occupy the same physical machines. For example, to tolerate one fault, four machines can act as agreement servers while three of them also act as execution replicas.

Second, a better metric for evaluating hardware costs of the system than the number of machines is the overhead imposed on each request relative to an unreplicated system. On the one hand, by cleanly separating agreement from execution and thereby reducing the number of execution replicas a system needs, the new architecture often reduces this overhead compared to previous systems. On the other hand, the addition of Privacy Firewall filters and their attendant public key encryption add significant costs. Fortunately, these costs can be amortized across batches of requests. In particular, when load is high the BASE library on which we build bundles together requests and executes agreement once per bundle rather than once per request. Similarly, by sending bundles of requests and replies through the Privacy Firewall nodes, we allow the system to execute public key operations on bundles of replies rather than individual replies.

To put these two factors in perspective, we consider a simple model that accounts for the application execution costs and cryptographic processing overheads across the system (but not other overheads like network send/receive.) The *relativeCost* of executing a request is the cost of executing the request on a replicated system divided by the cost of executing the request on an unreplicated system. For our system and the BASE library, the *relativeCost* is:

$$relativeCost = \frac{numExec \cdot proc_{app} + overhead_{req} + \frac{overhead_{batch}}{numPerBatch}}{proc_{app}}$$

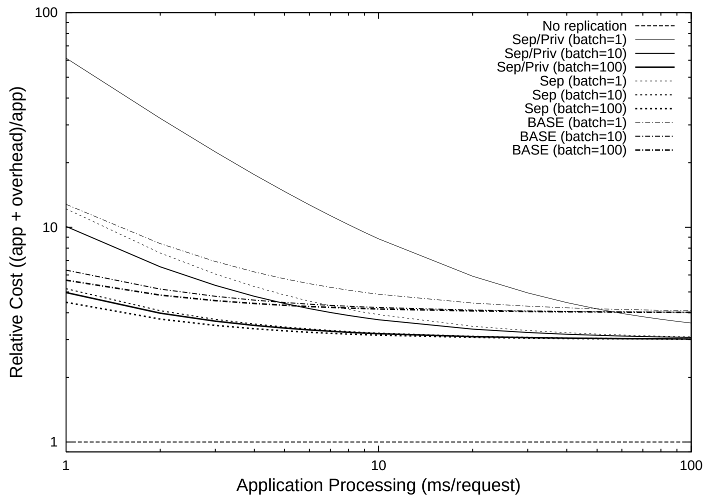

Figure 6.13: Estimated relative processing costs including application processing and cryptographic overhead for an unreplicated system, privacy firewall system, separate agreement and replication system, and BASE system for batch sizes of 1, 10, and 100 requests/batch.

The cryptographic processing overhead has three flavors: MAC-based authenticators, public threshold-key signing, and public threshold-key verifying. To tolerate 1 fault, the BASE library requires 4 execution replicas, and it does 8 MAC operations per request[4] and 36 MAC operations per batch. Our architecture that separates agreement from replication requires 3 execution replicas and does 7 MAC operations per request and 39 MAC operations per batch.[5] Our Privacy Firewall architecture requires 3 execution replicas and does 7 MAC operations per request and 39/3/6 MAC operations/public key signatures/public key verifications per batch.

Given these costs, the lines in Figure 6.13 show the relative costs for BASE (dot-dash lines), separate agreement and replication (dotted lines), and Privacy

[4]Note that when authenticating the same message to or from a number of nodes the work of computing the digest on the body of a message can be re-used for all communication partners [31, 32]. For the small numbers of nodes involved in our system, we therefore charge 1 MAC operation per message processed by a node regardless of the number of sources it came from or destinations it goes to.

[5]Our unoptimized prototype does 44 MAC operations per batch both with and without the Privacy Firewall.

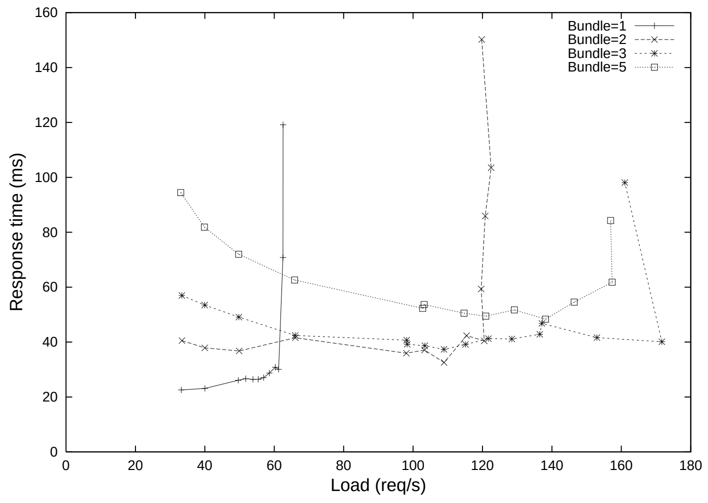

Figure 6.14: Microbenchmark response time as offered load and request bundling varies.

Firewall (solid lines) for batch sizes of 1, 10, and 100 requests/batch. The (unreplicated) application execution time varies from 1ms per request to 100ms per request on the x axis. We assume that MAC operations cost 0.2ms (based on 50MB/s secure hashing of 1KB packets), public key threshold signatures cost 15ms (as measured on our machines for small messages), and public key verification costs 0.7ms (measured for small messages.)

Without the Privacy Firewall overhead, our separate architecture has a lower cost than BASE for all request sizes examined. As application processing increase, application processing dominates, and the new architectures gain a 33% advantage over the BASE architecture. With small requests and without batching, the Privacy Firewall does greatly increase cost. But with batch sizes of 10 (or 100), processing a request under the Privacy Firewall architecture costs less than under BASE replication for applications whose requests take more than 5ms (or 0.2ms).

The simple model discussed above considers only encryption operations and application execution and summarizes total overhead. We now experimentally evaluate the peak throughput and load of our system. In order to isolate the overhead of our prototype, we evaluate the performance of the system when executing a simple Null server that receives 1 KB requests and returns 1 KB replies with no additional application processing. We program a set of clients to issue requests at a desired frequency and vary that frequency to vary the load on the system.

Figure 6.14 shows how the latency for a given load varies with bundle size. When bundling is turned off, throughput is limited to 62 requests/second, at which point the execution servers are spending nearly all of their time signing replies. Doubling the bundle size to 2 approximately doubles the throughput. Bundle sizes of 3 or larger give peak throughputs of 160-170 requests/second; beyond this point, the system is I/O limited and the servers have idle capacity. For example, with a bundle size of 10 and a load of 160 requests/second, the CPU utilization of the most heavily loaded execution machine is 30%. Note that our current prototype uses a static bundle size, so increasing bundle sizes increases latency at low loads. The existing BASE library limits this problem by using small bundles when load is low and increasing bundle sizes as load increases. Our current prototype uses fixed-sized bundles to avoid the need to adaptively agree on bundle size; we plan to augment the interface between the BASE library and our message queues to pass the bundle size used by the BASE agreement cluster to the message queue.

6.11.4 Network File System

For comparison with previous studies [31, 32, 130], we examine a replicated NFS server under the modified Andrew500 benchmark, which sequentially runs 500 copies of the Andrew benchmark [31, 70]. The Andrew benchmark has 5 phases: (1)

Phase	No Replication	BASE	Firewall
1	7	7	19
2	225	598	1202
3	239	1229	862
4	536	1552	1746
5	3235	4942	5872
TOTAL	4244	8328	9701

Figure 6.15: Andrew-500 benchmark times in seconds.

Phase	BASE	faulty server	faulty ag. node
1	12	19	33
2	1426	1384	1553
3	1196	1010	1102
4	1755	1898	2180
5	5374	6050	7071
TOTAL	9763	10361	11939

Figure 6.16: Andrew-500 benchmark times in seconds with failures.

recursive subdirectory creation, (2) copy source tree, (3) examine file attributes without reading file contents, (4) reading the files, and (5) compiling and linking the files.

We use the NFS abstraction layer by Rodrigues et al. to resolve nondeterminism by having the primary agreement node supply timestamps for modifications and file handles for newly opened files. We run each benchmark 10 times and report the average for each configuration. In these experiments, we assume hardware support in performing efficient threshold signature operations [144].

Figure 6.15 summarizes these results. Performance for the benchmark is largely determined by file system latency, and our firewall system's performance is about 16% slower than BASE. Also note that BASE is more than a factor of two slower than the no replication case; this difference is higher than the difference reported in [130] where a 31% slowdown was observed. We have worked with the authors of [130] and determined that much of the difference can be attributed to different versions of BASE and Linux used in the two experiments.

Figure 6.16 shows the behavior of our system in the presence of faults. We obtained it by stopping a server or an agreement node at the beginning of the benchmark. The table shows that the faults only have a minor impact on the completion time of the benchmark.

6.12 Related work

6.12.1 Separating Agreement from Execution

We use Rodrigues et al.'s BASE replication library [130] as the foundation of our agreement protocol, but depart significantly from their design in one key respect: our architecture explicitly separates the responsibility of achieving agreement on the order of requests from the processing the requests once they are ordered. Significantly, this separation allows us to reduce by one third the number of application-specific replicas needed to tolerate f Byzantine failures and to address confidentiality together with integrity and availability.

In [85], Lamport deconstructs his Paxos consensus protocol [84] by explicitly identifying the roles of three classes of agents in the protocol: *proposers*, *acceptors*, and *learners*. He goes on to present an implementation of the state machine approach in Paxos in which "each server plays all the roles (proposer, acceptor, and learner)". We employ a similar deconstruction of the state machine protocol: in Paxos parlance, our clients, agreement servers, and execution servers are performing the roles played, respectively, by proposers, acceptors, and learners. However, our acceptors and learners are physically, and not just logically, distinct. We show how to apply this principle to BFT systems to reduce replication cost and to provide confidentiality.

Baldoni, Marchetti, and Tucci-Piergiovanni advocate a three-tier approach to replicated services, in which the replication logic is embedded within a software middle-tier that sits between clients and end-tier application replicas [19]. Their goals are to localize to the middle tier the need of assuming a timed-asynchronous model [42], leaving the application replicas to operate asynchronously, and to enable the possibility of modifying on the fly the replication logic of end-tier replicas (for example, from active to passive replication) without affecting the client.

Our system also shares some similarities with the systems [25, 122] using stateless witness to improve fault-tolerance. However, our system differs in two respects. First, our system is designed to tolerate Byzantine faults instead of fail-stop failures. Second, our general technique replicates arbitrary state machines instead of specific applications such as voting and file systems.

To guarantee progress, the agreement protocol needs $2f+1$ nodes to participate. For load balancing, these nodes are normally chosen at random from the pool of $3f+1$ agreement nodes. Li and Tamir [91] use this observation to improve on our work so that the same $2f+1$ agreement nodes are chosen (a *preferred quorum*), and the remaining f can be idle and reduce their power consumption. Naturally, the idle nodes need to be involved in case of failure, but these are relatively rare. Lamport goes further, proposing a replicated state machine protocol that has the same properties but where f of the agreement nodes (called witnesses) can have even lower processing and storage requirements than the other agreement nodes [89].

6.12.2 Two-step Consensus

The two earlier protocols that are closest to FaB Paxos are the FastPaxos protocol by Boichat and colleagues [26], and Kursawe's Optimistic asynchronous Byzantine agreement [81]. Both protocols share our basic goal: to optimize the performance of the consensus protocol when runs are, informally speaking, well-behaved.

The most significant difference between FastPaxos and FaB Paxos lies in the failure model they support: in FastPaxos processes can only fail by crashing, while in FaB Paxos they can fail arbitrarily. However, FastPaxos only requires $2f+1$ acceptors, compared to the $3f+2t+1$ necessary for FaB Paxos. A subtler difference between the two protocols pertains to the conditions under which FastPaxos achieves consensus in two communication steps: FastPaxos can deliver consensus in two communication steps during *stable periods*, i.e. periods where no process crashes or recovers, a majority of processes are up, and correct processes agree on the identity of the leader. The conditions under which we achieve gracious executions are weaker than these, in that during gracious executions processes *can* fail, provided that the leader does not fail. As a final difference, FastPaxos does not rely, as we do, on eventual synchrony but on an eventual leader oracle; however, since we only use eventual synchrony for leader election, this difference is superficial.

Kursawe's elegant optimistic protocol assumes the same Byzantine failure model that we adopt and operates with only $3f+1$ acceptors, instead of $3f+2t+1$. However, the notion of well-behaved execution is much stronger for Kursawe's protocol than for FaB Paxos. In particular, his optimistic protocol achieves consensus in two communication steps only as long as channels are timely and *no* process is faulty: a single faulty process causes the fast optimistic agreement protocol to be permanently replaced by a traditional pessimistic, and slower, implementation of agreement. To be fast, FaB Paxos only requires gracious executions, which are compatible with process failures as long as there is a unique correct leader and all correct acceptors agree on its identity.

There are also protocols that use failure detectors to complete in two communication steps in some cases. Both the SC protocol [138] and the later FC protocol [73] achieve this goal when the failure detectors make no mistake and the coordinator process does not crash (their coordinator is similar to our leader). FaB Paxos differs from these protocols because it can tolerate unreliable links and Byzantine failures. Other protocols offer guarantees only for certain initial configurations. The oracle-based protocol by Friedman et al. [55], for example, can complete in a single communication step if all correct nodes start with the same proposal (or, in a variant that uses $6f+1$ processes, if at least $n-f$ of them start with the same value and are not suspected). FaB Paxos differs from these protocols in that it guarantees learning in two steps regardless of the initial configuration.

In a paper on lower bounds for asynchronous consensus [86], Lamport conjectures in "approximate theorem" 3a the existence of a bound $N > 2Q + F + 2M$ on the minimum number N of acceptors required by 2-step Byzantine consensus, where: (i) F is the maximum number of acceptor failures despite which consensus liveness is ensured; (ii) M is the maximum number of acceptor failures despite which consensus safety is ensured; and (iii) Q is the maximum number of acceptor failures despite which consensus must be 2-step. Lamport's conjecture is more general than ours—we do not distinguish between M and F—and more restrictive—unlike us, Lamport does not consider Byzantine learners but instead assumes that they can only crash. This can be limiting when using consensus for the replicated state machine approach: the learner nodes execute the requests, so their code is comparatively more complicated and more likely to contain bugs that result in unexpected behavior. Lamport's conjecture does not technically hold in the corner case where no learner can fail.[6] Dutta, Guerraoui and Vukolić have recently derived a compre-

[6]The counterexample can be found in Appendix B.1.

hensive proof of Lamport's original conjecture under the implicit assumption that at least one learner may fail [48]. In a later paper [87], Lamport gives a formal proof of a similar theorem for crash failures only. He shows that a protocol that reaches two-step consensus despite t crash failures and tolerates f crash failures requires at least $f + 2t + 1$ acceptors. In Section 6.7 we show that in the Byzantine case, the minimal number of processes is $3f + 2t + 1$.

After the initial publication of our results, Lamport has shown that in the case of crash failures, it was possible to reach consensus within two communication steps in the common case, even taking into account the initial message from the client (e.g. when consensus is used in a replicated state machine) [88]. The protocol we show in this chapter requires a third communication step in this setup, but it can tolerate Byzantine failures.

6.12.3 Confidentiality

Most previous efforts to achieve confidentiality despite server failures restrict the data that servers can access. A number of systems limit servers to basic "store" and "retrieve" operations on encrypted data [4, 80, 100, 105, 113, 134] or on data fragmented among servers [58, 74]. The COCA [159] online certification authority uses replication for availability, and threshold cryptography [43] and proactive secret sharing [69] to digitally sign certificates in a way that tolerates adversaries that compromise some of the servers. In general, preventing servers from accessing confidential state works well when servers can process the fragments independently or when servers do not perform any significant processing on the data. Our architecture provides a more general solution that can implement arbitrary deterministic state machines.

Secure multi-party computation (SMPC) [30] allows n players to compute an agreed function of their inputs in a secure way even when some players cheat. Although in theory it provides a foundation for achieving Byzantine fault-tolerant confidentiality, SMPC in practice can only be used to compute simple functions such as small-scale voting and bidding because SMPC relies heavily on computationally expensive oblivious transfers [51].

Firewalls that restrict incoming requests are a common pragmatic defense against malicious attackers. Typical firewalls prevent access to particular machines or ports; more generally, firewalls could identify improperly formatted or otherwise illegal requests to an otherwise legal machine and port. In principle, firewalls could protect a server by preventing all bad requests from reaching it (a request is *bad* if it causes a server to behave unexpectedly, e.g. by exploiting a bug in the implementation). An interesting research question is whether identifying all bad requests is significantly easier than building bug-free servers in the first place. The Privacy Firewall is inspired by the idea of mediating communication between the world and a service, but it uses redundant execution to filter mismatching (and presumptively wrong) outgoing replies rather than relying on *a priori* identification of bad incoming requests.

6.13 Conclusion

The main contribution of this chapter is to present the first study to apply systematically the principle of separation of agreement and execution to BFT state machine replication to (1) reduce the replication cost, (2) reduce the number of communication steps for agreement in the common case, and (3) enhance confidentiality properties for general Byzantine replicated services.

Separating agreement from execution allows us to build a system that uses the proven minimal number of nodes for agreement and execution, respectively. Although in retrospect this separation is straightforward, all previous general BFT state machine replication systems have tightly coupled agreement and execution, and have paid unneeded replication costs.

In traditional state machine architectures, the cost of additional replication is prohibitive. However, separating the cheaper agreement replicas from the more expensive execution replicas allows us to explore scenarios with additional agreement replicas. We find that adding $2t$ replicas allows agreement to complete in two communication steps (instead of three previously) in the common case despite t failures. We call that property two-step despite t failures and prove that it is impossible to be two-step despite t failures with any fewer nodes. We present Parameterized FaB Paxos, a new Byzantine-tolerant consensus protocol that is two-step despite t failures. Parameterized FaB Paxos is optimal in the number of nodes.

Separating agreement from execution allows us to build the Privacy Firewall and insert it in the middle, where it can filter out confidential information. The Privacy Firewall provides new confidentiality guarantees: it guarantees that, even if an adversary compromises some of the execution nodes, the messages sent to the client will be identical to messages that an uncompromised service would have sent. We prove a lower bound on the number of nodes that are needed for these guarantees and show that the Privacy Firewall meets the lower bound.

Chapter 7

Cooperative Services and the BAR Model

7.1 Introduction

In the previous chapters, we assumed a bound on the set B of nodes that deviate from the given protocol. We now explore an environment where this assumption may not be appropriate: cooperative services. In a *cooperative service*, nodes from multiple administrative domains collaborate in some way that is beneficial to each node, without a central authority controlling the nodes' actions. Some nodes can still deviate from the protocol because of hardware or software failures or because of malicious manipulation. But nodes can also deviate from the protocol for a new reason: freed from the central authority's oversight, the humans using the software (the *users*) can interfere with its configuration or even replace it with different software in order to maximize their benefit or minimize their costs. Selfish behavior has been investigated by economists for some time [63, 97, 123]. It has also been observed in distributed computer systems, in the context of network congestion [71] and "free riding" [3, 72] on file-sharing systems [60, 93]. For example, 66% of users on the Gnutella network share no file at all. Selfish behavior can lead to the well-known *tragedy of the commons* [63, 97], in which the actions best for individuals are detrimental to the system as a whole.

Existing models fall short when applied to cooperative services. The Byzantine fault-tolerance (BFT) model [90], which we used in the previous chapters, is not appropriate for cooperative services because it limits the number of deviating nodes (i.e. nodes that deviate from the protocol). All BFT protocols impose some bound on the set of Byzantine nodes, and no such protocol can give useful guarantees for the case where all nodes are Byzantine. In fact, there are problems for which the Byzantine model can only handle a smaller fraction of faulty nodes. For example, in the case of Byzantine consensus in the eventually synchronous model [49], it is well-known that no protocol can tolerate even a third of Byzantine nodes [49]. In cooperative services, especially if there is more at stake than free copies of music or video files, it is conceivable that *every* node will deviate from the protocol (because of the actions of the selfish users that are controlling them).

A number of researchers have studied systems in which all nodes are modelled as profit-maximizers [63, 97, 112, 123]. This approach, although it handles selfish behavior, is not appropriate for cooperative services either, because it is brittle in the face of Byzantine failures. Since Byzantine deviations may go against a node's best interest, they are not covered by this model. For example, the AS7007 incident [115]—where a misconfigured router announced that it was the best path to most of the Internet and disrupted global connectivity for over two hours—demonstrates the damage a single faulty node can cause in a system that is not

Byzantine-tolerant. A cooperative service must be able to tolerate arbitrary behavior from some nodes.

In this chapter we introduce the Byzantine Altruistic Rational (BAR) model. This model combines the advantages of Byzantine fault-tolerance and rational-tolerance: it can tolerate both rational misbehavior and Byzantine nodes. Unlike the Byzantine model, the BAR model allows one to build protocols in which all the nodes may deviate (rationally), and unlike a model that considers only rational nodes, BAR yields protocols that can tolerate some malicious nodes.

Given the potential for nodes to develop subtle tactics, it is not sufficient to verify experimentally that a protocol tolerates a collection of attacks identified by the protocol's creator. Instead, just as for Byzantine-tolerant protocols [90], it is necessary to design protocols that *provably* meet their goals, no matter what strategies nodes may concoct within the scope of the adversary model. After introducing the BAR model, we show an example of a BAR-Tolerant protocol and prove it correct.

7.2 The BAR Model

Consider a set N of n nodes; each node knows N and can identify the nodes it communicates with (either through authenticated links or through shared secrets). Each node i is given a *suggested protocol* σ_i. There is no central authority, so the user controlling node i might replace σ_i with some other protocol σ'_i. Some of the nodes may be broken and deviate from σ_i in ways that are not necessarily beneficial for the user.

The Byzantine Altruistic Rational (BAR) model addresses these considerations by classifying nodes into the following three categories.

- *Byzantine* nodes may deviate arbitrarily from the suggested protocol for any reason. They may be misconfigured, compromised, malfunctioning, misprogrammed, or they may just be optimizing for an unknown utility function that differs from the utility function used by rational nodes—for instance, by ascribing value to harm inflicted on the system or its users.

- *Altruistic* nodes follow their suggested protocol exactly. Intuitively, altruistic nodes correspond to *correct nodes* in the fault-tolerance literature. Altruistic nodes may reflect the existence of Good Samaritans and "seed nodes" in real systems. However, in the BAR model, nodes that crash cannot be classified as altruistic.[1]

- *Rational* nodes reflect self-interest and seek to maximize their benefit according to one of a known set U of utility functions. Rational nodes will deviate from the suggested protocol if and only if doing so increases their estimated utility from participating in the system. The utility function must account for a node's costs (e.g., computation cycles, storage, incoming and/or outgoing network bandwidth, power consumption, or threat of financial sanctions [92]) and benefits (e.g., access to remote storage [20, 41, 92, 104], network capacity [99], or computational cycles [143]) for participating in a system.

Nodes that deviate from the protocol (other than by crashing) have been considered Byzantine in the past. Tolerating Byzantine failures is costly and sometimes even infeasible, as we have seen. The BAR model allows us to model some of these deviating nodes using a stronger model in which it is possible to design protocols for situations were the Byzantine model would not apply (for example, when all nodes deviate because doing so is in their benefit). The BAR model is

[1]If needed, the BAR model could be expanded to allow for crashing altruistic nodes.

accurate [140] in the sense that rational behavior has been observed to take place, and we show that the BAR model is tractable in the sense that useful protocols can be designed for it. There may be other behaviors that are currently modeled as Byzantine that would benefit from a stronger, accurate, and tractable new model: addressing these behaviors is outside of the scope of this dissertation.

Under BAR, the goal is to provide guarantees similar to those from Byzantine fault-tolerance to "all rational and altruistic nodes" as opposed to "all correct nodes." We distinguish two classes of protocols that meet this goal.

- *Incentive-Compatible Byzantine Fault-Tolerant* (IC-BFT) protocols: A protocol is IC-BFT if it tolerates Byzantine nodes and if it is in the best interest of all rational nodes to follow the protocol exactly. An IC-BFT protocol therefore must define the optimal strategy for a rational node.

- *Byzantine Altruistic Rational Tolerant* (BAR-Tolerant) protocols: A protocol is BAR-Tolerant if it tolerates Byzantine and rational nodes, even if the rational nodes deviate from the protocol. Note that all IC-BFT protocols are also BAR-Tolerant.

7.3 Game Theory Background

Our work draws from the field of game theory [57]. Game theory aims to explain the actions of rational or self-interested nodes by modeling their interaction as a *game*. In these games, each node i chooses a *strategy* σ_i that represents that node's actions. The vector $\vec{\sigma} = (\sigma_0, \ldots, \sigma_{n-1})$ that assigns a strategy to each node is called a *strategy profile*. The game defines a function that takes the n nodes' strategies as input and outputs an *outcome*. The *utility function* u_i indicates the payoff that node

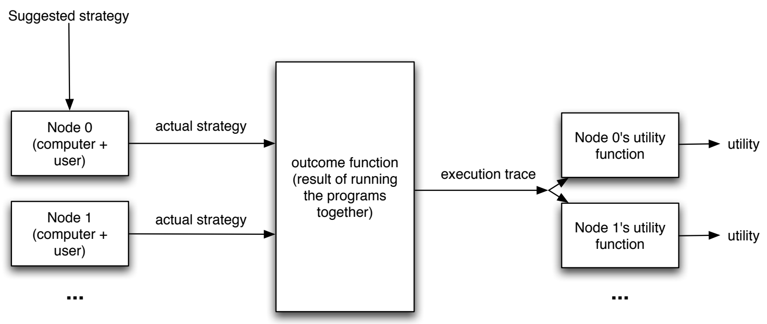

Figure 7.1: From suggested program to utility

i receives for that particular outcome. The utility for node i can be written as the function $u_i(outcome(\sigma_0, \ldots, \sigma_{n-1}))$, which we abbreviate $u_i(\vec{\sigma})$. We use $\vec{\sigma} \ominus \sigma'_j$ to represent the strategy profile where each node i follows strategy σ_i, except for node j that follows strategy σ'_j. Similarly, in $\vec{\sigma} \ominus \sigma'_X$ each node $i \notin X$ follows strategy σ_i and each node $j \in X$ follows strategy σ'_j. In game theory, all players choose the strategy that maximizes their payoff over the course of the game.

A strategy profile $\vec{\sigma}$ is a *Nash Equilibrium* [120] if no node r can improve its utility u_r by modifying its strategy unless another node also changes strategy. When individual utility functions are public knowledge, nodes can verify that a given $\vec{\sigma}$ is a Nash Equilibrium. Nash Equilibrium and its variants [12, 50, 64] play an important role in our fault models and protocol design as a starting point for our concept of Byzantine Nash Equilibrium.

7.4 Linking Game Theory and the BAR Model

The game theoretic concept of strategy corresponds, in cooperative services, to the protocol that nodes are running. The result of the outcome function is the execution trace as these protocols interact. Nodes then derive their utility from this trace, as illustrated in Figure 7.1. The utility function could, for example, take into account the number of computed digital signatures or the number of transmitted: information about both is available in the trace.

In game theory, cooperative solutions are sometimes not achievable in the context of one-shot games but can be achieved in *infinite horizon games* [16]—repeated games where the number of times the game is going to be played is unknown. Intuitively, repeated games can be structured so that nodes always follow the protocol in order to avoid punishment in the next repetition of the game; since the game has an infinite horizon, there is always a "next repetition" where punishment could take place. In order to be similarly always able to leverage a threat of future punishment, we assume an infinite-horizon game where each node participates only if the node gains a net benefit from its participation. Although assuming an infinite horizon game may appear somewhat unrealistic, in practice it may suffer, in order for the game to be solvable, that (i) the game is long, (ii) there is a small probability of the game ending after each instance, so the horizon is unknown, or (iii) for a node's real-world owner to risk punishment even after the game for bad behavior during a finite game [57].

7.4.1 Byzantine Nash Equilibrium

In our setup, each node i is given a protocol σ_i for consideration. We call σ_i node i's *suggested protocol*; it is the initial strategy of that node. Given a desired property P, our goal is to find a protocol σ that satisfies P when σ is given as the suggested protocol to each node in a cooperative service (or, more generally, to find a protocol profile $\vec{\sigma}$ with the same property; a protocol profile may assign a different protocol to each node).

The approach we follow is to build IC-BFT protocols: if σ is such that rational nodes never see a benefit in deviating (and therefore choose not to deviate), and if P holds despite deviations from the Byzantine nodes, then the protocol σ will maintain the property P in a cooperative service.

We must first specify the circumstances under which a rational node would deviate from the suggested protocol. In this chapter, we consider rational nodes that do not collude. In fact, we consider rational nodes that only deviate from the protocol if doing so increases their estimated utility, under the assumption that the other non-Byzantine nodes in the system follow the specified protocol. We also assume that rational nodes only consider protocols that terminate[2] (we denote this set of protocols with Σ). Even though we assume that the rational nodes do not collude, we can still tolerate a number of colluding nodes; they are simply classified as Byzantine.

Given that rational nodes only deviate if there is a unilateral benefit in doing so, one could think that "$\vec{\sigma} = (\sigma, \ldots, \sigma)$ is a Nash Equilibrium" is a sufficient condition for the protocol to be IC-BFT. Although that is the correct intuition, the concept of Nash Equilibrium is not sufficient for cooperative services. We need to introduce two concepts: the estimated utility and partial history. We introduce them by way of a toy example.

Consider a two-player infinitely repeated game. The game we use here is for illustration purposes only. Nodes take turn playing the role of sender. The sender can pick between 3 messages to send: "G", "Y", "R". The protocol specifies that the sender should always send "G". Non-sender nodes have two actions: Take or Ignore. The protocol specifies that non-sender node r should play "T" if the sender

[2]Recall that the protocol is repeated, so an infinite horizon game and terminating protocols are not mutually exclusive.

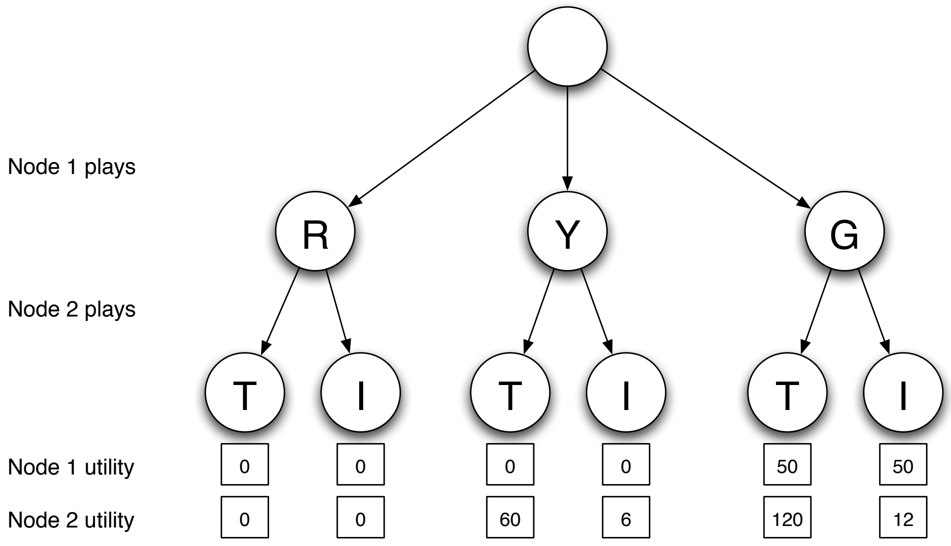

Figure 7.2: The RYG game

sent "G", otherwise r must play "I". A sender node always gets 50 points of utility if it sends "G", and 0 otherwise. Figure 7.2 shows the utility for non-sender nodes and the game tree for one instance of the protocol.

Suppose that a rational node r is playing the RYG game with a Byzantine node s. Suppose that the suggested strategy profile $\vec{\sigma} = (\sigma, \sigma)$ specifies that the sender should always play "G", and the recipient should always play "T" in response to "G" and "I" otherwise. It is not known in advance how the Byzantine nodes s will deviate (if at all) from $\vec{\sigma}$, so, in the presence of Byzantine nodes, more than one outcome is possible given a suggested strategy profile $\vec{\sigma}$. Thus, node r cannot compute the utility $u_r(outcome(\sigma, \sigma))$ directly. To distill this uncertainty down to a single number, we define the *estimated utility* function $\hat{u}_r$. In this toy example, the estimated utility is computed from considering the worst that the Byzantine node can do. For example, the estimated utility of instances where the Byzantine node s is the sender is 0, because the Byzantine node could play "R". The key point of the estimated utility is that it distils the uncertainty of the behavior of the Byzantine nodes down to a single number. Later in this chapter we build a Terminating Reliable Broadcast protocol (TRB); the estimated utility we use for TRB is introduced in Section 7.4.2.

The protocol σ is a Nash Equilibrium because no node r can increase its estimated utility by unilaterally deviating from σ. However, there are situations where it is rational for node r to deviate from the protocol. Consider the case where the Byzantine node s is sender, and plays "Y". This situation is represented by the vertex labeled "Y" in Figure 7.2. Each vertex in the figure represents a partial execution; we call it a *partial history*. The partial history represented by "Y" indicates a situation where node 2 (r) has received message "Y" from node 1 (s). At the partial history "Y", node r knows that it will receive 60 points of utility if it plays "T" and 6 points of utility if it plays "I". It is rational for node r to play "T". This behavior is different from what σ would have required r to do, which is to play "I" in answer to anything other than "G".

The Byzantine Nash Equilibrium condition requires that there is no partial history where rational nodes would be able to increase their estimated utility by deviating from the protocol. As the example illustrates, this is a stronger requirement than Nash Equilibrium.[3]

The set $\mathcal{K}_r$ is the set of all possible partial histories for r that (i) are consistent with the well-known assumptions made in the protocol (e.g. that at most f nodes are Byzantine), and (ii) are consistent with the protocol σ_r in the sense that for every $K_r \in \mathcal{K}_r$, when r receives a message it reacts by following σ_r. We are now

[3]The reader familiar with game theory may recognize that requiring the equilibrium to hold for all partial histories is similar to the concept of subgame perfection. Subgame perfection only applies when there is no uncertainty about the game, so it does not apply in the context of cooperative services.

ready to formally define our equilibrium condition.

Definition 16. *The strategy profile $\vec{\sigma} = (\sigma_0, \ldots, \sigma_{n-1})$ $(\sigma_i \in \Sigma\ \forall 0 \leq i < n)$ is a* Byzantine Nash Equilibrium *if and only if:*

$$\forall j, \forall \hat{u}_j \in \hat{U}, \forall K_j \in \mathcal{K}_j, \forall \sigma' \in \Sigma : \hat{u}_j(\vec{\sigma} \ominus \sigma'_j, K_j) \leq \hat{u}_j(\vec{\sigma}, K_j)$$

We say that the strategy σ is a Byzantine Nash Equilibrium if $\vec{\sigma} = (\sigma, \ldots, \sigma)$ is a Byzantine Nash Equilibrium.

Informally, a strategy σ is a Byzantine Nash Equilibrium if no node j can increase its estimated utility by deviating from the suggested strategy σ for any of the estimated utility functions that rational nodes may have (represented by $\hat{U}$) and regardless of what it may have seen other nodes do (represented by K_j).

Definition 17. *Given a fail-prone system $\mathcal{B}$ that describes the sets of nodes that may be Byzantine and a rational system $\mathcal{R}$ that similarly describes the sets of nodes that may be rational, and given the set U of utility functions that rational nodes use, we say that a protocol profile $\vec{\sigma}$ that satisfies some property P is* IC-BFT *if the following two conditions hold.*

1. *$\forall B \in \mathcal{B}$: $\vec{\sigma}$ satisfies P despite nodes in B being Byzantine, and*

2. *$\vec{\sigma}$ is a Byzantine Nash Equilibrium.*

We say that the protocol σ is IC-BFT if $\vec{\sigma} = (\sigma, \ldots, \sigma)$ is IC-BFT.

7.4.2 Estimating the Utility

We now present our choice of $\hat{u}$ for the protocols in this chapter. Our starting point is the utility function u: if we have sufficient information to determine the actions of each node—including the Byzantine nodes—then we can compute the outcome and consequently we can compute u directly. The formula below shows how j computes its estimated utility.

$$u_j(\vec{\sigma} \ominus \sigma'_j \ominus \phi_B, K_j, k^b) = u_j(outcome(\vec{\sigma} \ominus \sigma'_j \ominus \phi_B, K_j, k^b))$$

Where:

- $\vec{\sigma}$ is the suggested strategy profile.
- σ'_j is the protocol that node j follows.
- B is the set of nodes that are actually Byzantine (this information is not necessarily available to j).
- ϕ_B is the set of protocols that each Byzantine node will follow. Again, node j does not have this information.
- K_j is the partial history of node j
- k^b is the number of the last instance of the protocol that we are evaluating the utility over (the protocol does not stop then, just our accounting). k^b must naturally be at least as large as the last instance in K_j.

This function takes node j's partial history K_j into account because the protocol σ'_j may require node j to behave differently depending on what it has observed in the past. For example, after observing that some node x is Byzantine, σ'_j may indicate that j should not send further messages to x. The argument k^b allows us to compute the utility over several successive instances of the game. This compounding is necessary because the utility may be different for certain instances.

For example, a node may only receive a benefit on some instances: this is the case in the TRB example that we discuss in Section 7.5, where nodes only receive a benefit when they have the role of sender. The formula below shows how the estimated utility for node j is computed from knowledge that is available to j.

$$\hat{u}_j(\vec{\sigma} \ominus \sigma'_j, K_j) = \underbrace{\min_{B \in \mathcal{B}(K_j)}}_{(1)} \underbrace{\min_{\phi_B}}_{(2)} \underbrace{\lim_{\bar{s} \to \infty} \frac{1}{\bar{s}}}_{(3)} u_j(\vec{\sigma} \ominus \sigma'_j \ominus \phi_B, K_j, \bar{s})$$

The rightmost term is the function u we have seen earlier. Term (3) allows us to compute the average utility over an infinite game. Term (2) represents node j's risk aversion with respect to which protocol ϕ_B the Byzantine nodes will actually follow. Node j computes everything to the right of term (2) for every possible protocol ϕ_B and then choose the minimal value.[4] Term (1) represents node j's risk aversion with respect to which nodes are Byzantine: node j maximizes the worst-case utility over all sets B consistent with the previous observations K_j.

7.5 An Example

In this section we show that Lamport's classic Terminating Reliable Broadcast protocol [90] (LTRB) (Figure 7.3) fails if the nodes are rational. We show how to transform it into a BAR-Tolerant protocol that can handle both Byzantine and rational nodes.

In TRB, a distinguished node—the *sender*—initiates a broadcast. We call the broadcasted value the *proposal*. TRB guarantees four properties:

[4]Even though there is an infinite number of protocols that Byzantine nodes could follow, node j can compute the worst that the Byzantine node can do to it because (for a given σ'_j) there is a finite number of possible interactions between the Byzantine nodes and j (since σ'_j terminates).

- *Validity*: if the sender is correct and broadcasts a message m, then every correct node eventually delivers m;
- *Agreement*: all correct nodes deliver the same message;
- *Integrity*: all correct nodes deliver only one message; and
- *Termination*: every correct node eventually delivers some message.

Our goal is to derive a protocol that guarantees the same properties when "correct" is changed to "correct or rational".

7.5.1 LTRB is Byzantine-Tolerant

LTRB is a synchronous protocol that proceeds in rounds. In every round, nodes first send messages, then receive messages and process them. Each message that is sent in a round is received in the same round. LTRB implements TRB despite up to f Byzantine nodes. LTRB assumes *message authentication*, i.e. a mechanism by which correct nodes can apply unforgeable signatures to the messages they send (we denote with $m{:}p$ the result of node p applying its signature to message m; the abstraction of signatures can in practice be achieved with high probability using e.g. RSA [128]). We consider a situation where the protocol is run not just once but continuously, and each node in turn takes the role of sender.

In the protocol, correct nodes only consider messages that are valid, ignoring the rest. A message received in round i is *valid* if and only if it has the form $m{:}p_0{:}p_1 :, \ldots, {:}p_i$ where m is the message's value, p_0 is the sender, and all the p_j $(0 \leq j \leq i)$ are distinct. We assume that a correct node that receives a valid message can extract from it the value it carries. Formally, we say that node r

```
**Initialization** for process $p$ :
1.  **if** $p ==$ sender and wishes to broadcast $m$ :
2.      $extracted := relay := \{m\}$
3.  **else** : $extracted := relay := \emptyset$

**Round** $i, 1 \leq i \leq f+1$ :
4.  **for each** $s \in relay$ : send $s{:}p$ to all but sender
5.  **receive** round $i$ messages from all processes
6.  $relay := \emptyset$
7.  **for each** valid msg $m{:}p_0{:}\ldots{:}p_{i-1}$ received :
8.      **if** $m \notin extracted$ :
9.          $extracted := extracted \cup \{m\}$
10.         $relay := relay \cup \{s\}$

**End** of round $f+1$ :
11. **if** $\exists m$ s.t. $extracted == \{m\}$ : **deliver** $m$
12. **else** : **deliver** **SF**
```

Figure 7.3: Round-based version of Lamport's consensus protocol for arbitrary failures with message authentication.

extracts message $m{:}p_0{:}\ldots{:}p_{i-1}$ to mean that r adds m to its *extracted* set in round i.

The protocol runs for $f+1$ rounds. In the first round, the sender signs the value m it wants to broadcast and sends it to all the other nodes. A correct node t that receives a valid message s from the sender in round 1 extracts the value and adds it to its *extracted* set. Node t then applies its signature to s and relays it to all other nodes in round 2. In subsequent rounds, t extracts the value from each valid message it receives. Whenever it extracts a value for the first time, t applies its signature to the corresponding message and relays the message to all other nodes in the following round.

At the end of round $f+1$, node t uses the following delivery rule: if it has extracted exactly one value, then t delivers that value; otherwise, t concludes that the sender is faulty and delivers the default value **SF**. It is not hard to show that

LTRB satisfies Validity, Agreement, Integrity, and Termination [90].

7.5.2 LTRB is not BAR-Tolerant

If nodes act rationally to maximize their own benefit, then LTRB's safety properties are violated. We must emphasize that this is not the environment for which Lamport's TRB protocol was initially designed, so it should not come as a surprise that LTRB is not BAR-Tolerant. Seeing exactly *how* the protocol fails is enlightening, however, both because it provides insight into how the BAR model differs from the Byzantine model and because it will guide us toward a modified protocol that achieves BAR-Tolerance.

When moving to the BAR model, we must specify a utility function for the rational nodes to indicate what may motivate them to deviate from the protocol.

We show that LTRB is not BAR-Tolerant with respect to two reasonable utility functions. Both utility functions consider sending and signing messages as costs. They differ in their benefits: with the first utility function, a rational node x benefits in all instances of TRB where x is the sender and the four TRB properties are satisfied. With the second utility function, x benefits in each TRB instance in which the four TRB properties are satisfied, independent of who is the sender. We call the second utility function *safety-aligned*.

To show that LTRB is not BAR-Tolerant with respect to both utility functions, we first show that a rational node may increase its estimated utility by deviating from the LTRB protocol. Second, we show that if all rational nodes deviate, then the safety properties of LTRB are violated.

We start by formally defining the first utility function. We say that the utility u_x for some node x at the end of executing an instance of LTRB (where x

follows protocol σ_x) is $\alpha b - \kappa$, where:

- α is a positive constant (representing the benefit).
- b ("broadcast") is 0 when x does not have the role of sender. Otherwise, b is 1 if protocol σ_x followed by node r is such that Validity, Agreement, Integrity, and Termination hold when x is the sender, despite up to f Byzantine failures, if the other non-Byzantine nodes follow LTRB.
- κ represents the cost (in our example the cost is non-negative).

κ is equal to $\beta s + \gamma t$, where s is the number of times that x signed a message and t is the total number of bytes in messages sent by x. β and γ are positive constants that represent, respectively, the relative costs of signing messages and transmitting bytes. For simplicity, we assume that all signatures have the same size η_s, all numbers that are sent have the same size η_n, and the sender's proposal value is always padded to the same size η_p. Note that there exist choices for the constants α, β, γ that cause estimated utility $\hat{u}_x$ to be at most 0 regardless of what node x does. This corresponds to situations where the cost of running the protocol exceeds the benefit and no rational node chooses to participate. The set of utility functions U contains an instance of the function $\alpha b - (\beta s + \gamma t)$ for each choice of α, β and γ for which $\hat{u}_r$ is positive. All rational nodes use a utility function from U.

Having specified the utility function, the next step is to show that a rational node can increase its estimated utility by deviating from the protocol. In other words, the protocol is not a Byzantine Nash Equilibrium (so it is not IC-BFT, either).

The simplest proof that LTRB is not IC-BFT is to consider the deviation σ' where a rational node x simply does nothing unless it is the sender. Rational node x gets the same benefit in this case as it would when running LTRB (since b is only influenced by the instances where x is the sender), but its costs are significantly reduced (since it skips some instances of the protocol). So, LTRB is not a Byzantine Nash Equilibrium since rational nodes have an incentive to deviate. If every node follows deviation σ', then we have an instance of the tragedy of the commons: either Agreement is violated when the sender delivers its proposal and the deviating nodes deliver **SF**, or Integrity is violated when the deviating nodes fail to deliver any value at the end of the protocol.

Safety-aligned utility functions do not guarantee BAR-Tolerance

Having shown that LTRB is not BAR-Tolerant with respect to the first utility function, we turn to the second utility function we mentioned earlier, the one that is safety-aligned. One could think that if every rational node only considers deviations that maintain safety (as is the case with safety-aligned utility functions), then safety would always be maintained. We show that this is not the case.

Define the safety-aligned utility function u' for TRB as follows: node x's estimated utility still has the format $u'_x = \alpha b - (\beta s + \gamma t)$, and s and t are defined as before; we change b so that it is 1 if protocol σ_x followed by node x is such that Validity, Agreement, Integrity, and Termination hold at every instance, despite up to f Byzantine failures, if the other non-Byzantine nodes follow LTRB (regardless of whether x is the sender). Otherwise, b is 0.

The "Lazy TRB" (LLTRB) protocol of Figure 7.4 allows rational nodes to increase their utility (when using the safety-aligned utility function u'). They get the same benefit, but their costs are reduced. We say that a rational node that is following LLTRB is *lazy*. The LLTRB protocol differs from LTRB in that a lazy

```
**Initialization** for process $p$ :
1.  let $R :=$ set of all nodes except $p$ or the sender.
2.  if $|R| > f+1$ : $R :=$ a subset of $R$ of size $f+1$
3.  if $p ==$ sender and wishes to broadcast $m$ :
4.      $extracted := relay := \{m\}$
5.  else : $extracted := relay := \emptyset$

**Round** $i, 1 \leq i \leq f+1$ :
6.  if $i == f+1$ : $R :=$ all processes except $sender$ and $p$
7.  for each $s \in relay$ : send $s{:}p$ to all processes in $R$
8.  receive round $i$ messages from all processes
9.  $relay := \emptyset$
10. for each valid msg $m{:}p_0{:}\ldots{:}p_{i-1}$ received :
11.     if $m \notin extracted$ :
12.         $extracted := extracted \cup \{m\}$
13.         $relay := relay \cup \{s\}$

**End** of round $f+1$ :
14. if $\exists m$ s.t. $extracted == \{m\}$ : deliver $m$
15. else : deliver **SF**
```

Figure 7.4: A lazy version of Lamport's protocol. Rational nodes only follow this protocol if $f > 0 \wedge n > f+2$; otherwise they follow LTRB.

node q does not send messages to all other nodes, but only to $f+1$ other nodes. The intuition is that the message will reach at least one non-Byzantine node that follows LTRB, and that node will relay the message in q's place.

LLTRB is identical to LTRB unless $f > 0$ and $n > f+2$. The following lemma shows that lazy node q can correctly conclude that, as long as every other non-Byzantine node follows the Lamport protocol, Agreement cannot be violated by its decision to take a free ride in round 1. The intuition is that the lazy node q relies on other nodes to forward the round 1 messages in its place.

Lemma 53. *If one node follows LLTRB, the remaining non-Byzantine nodes follow LTRB and a non-Byzantine node extracts m, then every non-Byzantine node eventually extracts m.*

Proof. If $f = 0$ or $n \leq f+2$ then LLTRB and LTRB are identical. In this case the conclusion holds directly since Agreement holds for LTRB. We therefore focus on the case where $f > 0$ and $n > f + 2$.

Let i be the earliest round in which some altruistic or lazy node q extracts m. There are two cases that we consider: q can be the sender, or not.

If q is the sender, then $i = 0$ (Figure 7.4, line 2). Node q is either lazy or not (by assumption, q is not Byzantine). If it is not lazy, then it sends a valid message to all nodes in round 1 (Figure 7.3, line 4) and then all the non-Byzantine recipients extracts m (Figure 7.3, line 9, or Figure 7.4, line 12). If q is the sender and q is lazy, then it sends a valid message to at least $f + 1$ nodes in round 1 (Figure 7.4, line 7). There is at least one non-lazy and non-Byzantine recipient. It extracts m by the end of round 1 (Figure 7.3, line 9) and then relays that value to all at the start of round 2 (Figure 7.3, lines 10 and 4) (there is a round 2 since $f > 0$). All non-Byzantine nodes will then extract m. The lazy node l also extracts m: if $l = q$ then it extracts the value in round 0. Otherwise, it extracts it in round 1 after receiving it from the sender. So, the conclusion holds if q is the sender.

Otherwise, non-sender node q extracted m in round $i > 0$. We show that $i \leq f$. Node q extracted m because it received a valid message $m{:}p_0{:}\ldots{:}p_{i-1}$. By the definition of valid message, all $p_0, \ldots, p_{i-1}$ are distinct.

We show that nodes $p_0, \ldots, p_{i-1}$ are all Byzantine. Suppose, for contradiction, that there is some node p_j that is not Byzantine ($0 \leq j < i$). Since the signature of a non-Byzantine process cannot be forged, it follows that p_j signed and relayed the message $m{:}p_0{:}\ldots{:}p_j$ in round j. Since p_j is not Byzantine and it forwarded the value, p_j must have extracted m in round $j < i$ (Figure 7.3, lines 9–10, or Figure 7.4, lines 12–13), contradicting the assumption that i is the earliest round in which a non-Byzantine process extracts m. Hence, if a message is forwarded before being extracted by a non-Byzantine node q, then the forwarding node is Byzantine. There are at most f Byzantine nodes, so i must be at most f.

Having extracted m in round $i \leq f$, non-sender node q will send a valid message $m{:}p_0{:}\ldots{:}p_{i-1}{:}q$ in round $i + 1 \leq f + 1$. Node q sends the message to all if $i == f$. Otherwise nodes q sends the message to at least one non-Byzantine node k, and k will have one more round in which to relay m to all nodes, unless it has done so already. □

Lemma 54. *If one node follows LLTRB and the remaining non-Byzantine nodes follow LTRB, then Agreement holds.*

Proof. From Lemma 53 it follows that all non-Byzantine processes extract the same set of values; hence, they all deliver the same message, proving Agreement. □

Lemma 55. *If one node follows LLTRB and the remaining non-Byzantine nodes follow LTRB, then Validity and Integrity hold.*

Proof. As before, we only need to consider the case $f > 0$ and $n > f + 2$, since otherwise LLTRB and LTRB are identical.

Integrity holds trivially, since non-Byzantine nodes only deliver once.

If the sender is non-Byzantine and it broadcasts a message m with value v, then at least one non-lazy non-Byzantine node receives m and extracts it in the first round (since $n > f+2$). Non-Byzantine nodes only extract valid messages, and valid messages include a signature from the sender. Since the sender only broadcasts a single message, no non-Byzantine node will extract any value other than the one contained in m. From Lemma 53, it follows that all non-Byzantine nodes extract the same set of values, so they will deliver v. □

The term b in u_r therefore has the same value whether r executes LTRB or LLTRB. Every signature and every message that occur in LLTRB also occur in LTRB, so running the LLTRB protocol incurs no more costs than running the LTRB protocol. If $f > 0$ and $n > f + 2$, and if $\vec{\sigma}$ is the suggested strategy of every node following LTRB, then $\hat{u_r'}(\vec{\sigma} \ominus LLTRB_r, K_r) < \hat{u_r'}(\vec{\sigma}, K_r)$ for all K_r (trivially, since the protocol ignores K_r): the LTRB protocol is not a Byzantine Nash Equilibrium with respect to u' because the lazy node r prefers to deviate rather than follow $\vec{\sigma}$.

Lemma 56. *If $f > 0$ and $n > f + 2$ and all rational nodes follow LLTRB, then Agreement can be violated.*

Proof. Consider the case where $n = f + 4$, all nodes are rational except node $f + 2$ that is Byzantine. The sender (node number $f + 3$) is rational, it signs and sends m to the first $f + 1$ nodes (Figure 7.4, line 7) and delivers m since no node sends it any message (Figure 7.4, line 1). The recipients of the sender's message are rational and they all relay the message to the first $f + 2$ nodes (excluding themselves) in round 2 (there is a second round since $f > 0$). Node number $f + 2$, being Byzantine, does nothing, and also the other recipients take no action, since they have already extracted m (Figure 7.4, line 11). The sender and the first $f + 1$ nodes deliver m at the end, and the rational node number $f + 4$ delivers **SF** since it received no message. Agreement is therefore violated. □

The LTRB protocol exhibit the tragedy of the commons regardless of whether the utility function is safety-aligned or not: safety is violated when all nodes behave rationally and deviate from the suggested protocol to increase their estimated utility.

7.5.3 TRB+ is BAR-Tolerant

We modify the LTRB protocol so that there is no incentive for rational nodes to deviate unilaterally from the protocol. Our modified protocol requires $n > 2f$. We use the utility function u, not safety-aligned utility function u' (the results hold for u' as well since in our modified protocol safety holds in all instances: any deviation that would increase u' would increase u as well). Our new protocol, TRB+, is based on three principles:

1. The protocol must specify a well-known pattern of messages: every time that a node a expects a message from a node b, then node a can compute deterministically, from the information available to it, a set E such that the message to be received is an element of E (as long as a and b follow the protocol, naturally). The set E is then used to detect nodes that deviate from the pattern of messages (this is a form of failure detector).

2. Nodes that are observed to deviate from this pattern of messages are subject to *punishment*, defined as a course of action that decreases the utility of the target.

3. The protocol must ensure that there is no benefit in unilateral deviations from the protocol that stay within the required pattern of messages.

These three principles together ensure that there is no unilateral benefit in deviating from the protocol. While every protocol can be said to have a pattern of messages (with E being the universe of all possible messages), when following the three principles the pattern of messages must be chosen carefully: in order to facilitate the third principle, the pattern of messages must be as restrictive as possible (i.e. the set E should be as small as possible).

constant	meaning
n	number of nodes
f	maximal number of Byzantine nodes
α	benefit
β	cost of sending a signature
γ	cost of sending a byte
η_s	size of a signature
η_p	size of a proposal value
η_n	size of a number
variable	meaning
G_p	set of nodes that p has not observed deviating
$G[x,y]$	*true* unless x sends a penance when y is sender
k	number of the instance of TRB+ that is executing
ξ	amount of filler in the **penance**(...) message
$\bot_1, \bot_2$	constants of size η_p contained in the **filler**(...) message

Table 7.1: Variables and constants for TRB+

Table 7.1 and Figures 7.5 and 7.6 show variables and pseudocode for the TRB+ protocol. We use $m{:}x{:}y$ to denote the message m, signed by nodes x and y. The function **head**($m{:}x{:}y$) returns m, and the function **tail**($m{:}x{:}y$) returns the two signatures. If $m = (a, b)$, then $m[0] = a$ and $m[1] = b$. Nodes cannot sign on behalf of other nodes, but we sometimes use the notation $m' == m{:}x$ when some node (not necessarily x) is checking that the message m' it received contains the value m and a valid signature from x. The TRB+ protocol is similar to LTRB in that nodes sign and forward values they extract, but we add **filler**(...) and **penance**(...) messages to follow the three principles outlined above. The next few paragraphs go through the three principles and explain how we modified LTRB to obtain TRB+.

Pattern of messages. We must modify the LTRB protocol because it does not have a pattern of messages that is restrictive enough. In LTRB, a node must forward every value that it extracts, but nodes do not know how many values other nodes have extracted, so their set E must allow other nodes to forward a value, or not.

```
**Initialization** for process $p$ on instance $k$ of the protocol :
1.    $G_p := \{0 \ldots (n-1)\} - p$
2.    $G[x,y] := true$ for each $x, y \in \{0 \ldots n-1\}$

**Round 1** for process $p$ :
10.   if $p == sender$ :
11.       send $((k,j){:}p,\ (k,msg){:}p)$ to every process $j$ in $G_p$
12.       deliver $msg$
13.   else : // $p \neq sender$
14.       if received $(k,p){:}sender$ : $ticket := (k,p){:}sender$
15.       else :
16.           $ticket :=$ **penance**$(p)$
17.           $G[p, sender] := false$
18.           $G_p := G_p - sender$
19.       if received $(k,m){:}sender \wedge |m| == \eta_p \wedge sender \in G_p$ :
20.           $extracted := \{m\}$
21.           $relay := \{\ (k,m){:}sender\ \}$
22.       else :
23.           $extracted := relay := \emptyset$
24.           $G_p := G_p - sender$

**Round $i$** $(2 \leq i \leq f+1)$ for process $p$ :
30.   // 1. Send penance when necessary
31.   if $i == 2$ : send $ticket$ to all procs. in $G_p - sender$
32.   // 2. Send two messages to all participants who have not deviated
33.   for each $s \in relay$ :
34.       send $s{:}p$ to all processes in $G_p - sender$
35.   if $|relay| < 1$ : send **filler**$(1,i,p)$ to all procs. in $G_p - sender$
36.   if $|relay| < 2$ : send **filler**$(2,i,p)$ to all procs. in $G_p - sender$
37.   $relay := \emptyset$
38.   // 3. Receive penances when necessary
39.   if $i == 2$ :
40.       for each process $j \in G_p - sender$ :
41            $G[j, sender] := G[j, sender] \wedge$ (received $(k,j){:}sender$ from $j$)
42.           if received neither $(k,j){:}sender$ nor **penance**$(j)$ from $j$ :
43.               $G_p := G_p - j$ // process $j$ deviated from the protocol
44.   // 4. Receive messages, check for deviation
45.   for each process $j \in G_p - sender$ :
46.       if also received from $j$ $s_1$ and $s_2$ s.t. **valid**$(s_1,i,j) \wedge$ **valid**$(s_2,i,j) \wedge$ **head**$(s_1) \neq$ **head**$(s_2)$,
          and nothing else :
47.           for each integer $l$ s.t. $1 \leq l \leq 2$ :
48.               if **interesting**$(s_l,i,j)$ :
49.                   $extracted := extracted \cup \{$**head**$(s_l)\}$
50.                   if $|relay| < 2$ : $relay := relay \cup \{s_l\}$
51.       else : $G_p := G_p - j$ // process $j$ deviated from the protocol

**Postprocessing** : // At the end of the last round, decide
60.   if $\exists m$ s.t. $extracted == \{m\}$ : deliver $m$
61.   else : // sender faulty
62.       $G_p := G_p - sender$
63.       deliver SF
```

Figure 7.5: TRB+, an IC-BFT protocol for Terminating Reliable Broadcast. The protocol is run continuously. The sender for a given instance k of the protocol is chosen round-robin.

```
**valid**$(m, i, j)$ :
100. if **head**$(m)[0] \neq k$ : return *false*
101. if   **head**$(m)[1] \neq \perp_1 \wedge$ **head**$(m)[1] \neq \perp_2$
102.       $\wedge$ **head**(**tail**$(m)$) is a signature from *sender*
103.       $\wedge$ last signature on $m$ is from $j$
104.       $\wedge$ **tail**$(m)$ is a chain of $i$ signatures :
105.       return *true*
106. if $m ==$ **filler**$(1, i, j) \vee m ==$ **filler**$(2, i, j)$ : return *true*
107. return *false*

**interesting**$(m, i, j)$ :
110. return (
111.       **head**$(m)[0]{==}k$
112.       $\wedge$ **head**$(m)[1] \neq \perp_1 \wedge$ **head**$(m)[1] \neq \perp_2$
113.       $\wedge$ **head**(**tail**$(m)$) is a signature from *sender*
114.       $\wedge$ last signature on $m$ is from $j$
115.       $\wedge$ **tail**$(m)$ is a chain of $i$ *distinct* signatures
116.       $\wedge$ $m$ does not contain our own signature
117.       $\wedge$ $\forall x \in relay$ : **head**$(x) \neq$ **head**$(m)$ )

**padding**$(l)$ :
120. return a sequence of zeroes of length $l$.

**filler**$(pos, i, p)$ :
130. return $((k, \perp_{pos}),$ **padding**$(\eta_s(i-1)))$:$p$

**penance**$(j)$ :
140. $g := |\{x : G[j, x]\}|$
141. $sender\_msg := 3\eta_n + 2\eta_s + \eta_p$
142. $nonsender\_msg := f\eta_p + f\eta_n + f(f+3)\eta_s/2 + 2\eta_n + \eta_s$
143. $\delta := n(\eta_n + \eta_s)$
144. $\xi := (sender\_msg + (n-1) * nonsender\_msg + \delta)/g - \eta_n - \eta_s$
145. return $(k,$**padding**$(\xi))$:$j$
```

Figure 7.6: Helper functions for TRB+

TRB+, instead, specifies a simple pattern: in every round, every node must send a fixed number of messages whose contents are specified by the protocol.

The pattern of messages is the following: in the first round, only the sender sends a message (Figure 7.5, line 11). The sender sends no message in subsequent rounds. Every other node sends three messages in the second round (lines 31–36), and two messages in each subsequent round (lines 33–36). To make the protocol incentive compatible, nodes that fail to send the expected messages are punished (more on this below).

Dolev and Strong observed [45] that once a node has extracted two distinct valid values, its decision is set to **SF**, independent of how many more distinct values it extracts. Consequently, forwarding two values is sufficient for a TRB protocol. This observation motivates our choice of forwarding exactly two values in every round:[5] two values are sufficient for TRB, and sending exactly two (instead of at most two in Dolev-Strong's protocol) allows us to create a more restrictive pattern of messages. When nodes have extracted fewer than two values, they forward a value in a special **filler**(...) message instead, so that they can remain in the pattern of messages and avoid punishment.

Punishment for deviating from the pattern. Punishment in TRB+ comes from the **penance(...)** messages. Each node p keeps a set G_p of the nodes that have been following the pattern of messages so far. If node p observes that node x deviates, then x is removed from G_p (lines 18, 24, 43, 51, and 62) and, as a result, node p will not send the *ticket* message (sometimes simply called the *ticket*) to x (line 11). The pattern of messages requires all nodes to forward a message (line 31): either the *ticket* message (line 14) or, if they have not received it, the expensive **penance(...)** message instead (line 16). If node x deviates in its interaction with node p, then node p will not send a *ticket* to x and thus force x to send a more expensive message. There is no cost to p for punishing x, so node p has no incentive to withhold punishment. The constant ξ determines the size of the **penance(...)** message. In Lemma 65 we compute ξ to make sure that the **penance(...)** message is expensive enough to counterbalance any benefit obtained by deviating from the pattern of messages.

[5]Note that on the second round non-Byzantine nodes send three messages, yet forward only two values.

No beneficial deviation inside the pattern. Part of making sure there is no profitable deviation within the required pattern of messages is to require that the two messages nodes send at every round (lines 33–36) cost the same, regardless of how many values were extracted. Nodes that have extracted fewer than two values are required to instead send a **filler**(...) message (line 130) which has the same cost as forwarding an extracted value (in terms of number of signatures and number of bytes) even though it does not contain the sender's proposal but instead contains a constant ($\perp_1$ or $\perp_2$) of the same size. Both the forwarded value and the **filler(**...**)** message have a single signature.

A subtle technical detail that comes as a consequence of the **filler**(...) messages is that we cannot apply the same optimization as Dolev and Strong's protocol [45], where nodes relay messages only to those nodes whose signature does not already appear on the message. The purpose of this optimization is to avoid sending messages that would immediately be discarded: if the recipient r already signed a value, r must have already extracted it so r has nothing to learn from the message. This optimization, unfortunately, does not mix well with our requirement for a strict pattern of messages. There are two ways to combine this optimization with a pattern of messages: we could either allow recipients to expect sometimes only a single message, or we could require the sender to send two messages always, but send a **filler**(...) message rather than a message that the recipient will immediately discard. Either choice violates the three principles. The first choice would allow a node to benefit by deviating within the pattern, violating the third principle: a rational node would relay a single value even when it is supposed to relay two, and the recipient would not punish the rational node since the exchange satisfies the pattern of messages. The second choice also violates the third principle, although in a more subtle way. Consider a rational node r that has two values to relay. Suppose both values are signed by node b, so that node r should send two **filler**(...) messages to b, and the signed values to all other nodes. The protocol requires all of these messages to be signed, so according to the protocol, node r should sign both values and the two **filler**(...) messages, for a total of four signatures (all messages are distinct). In this scenario, node r can benefit by deviating: r can sign the **filler**(...) messages only (only two signatures), and send them to all nodes. Since the pattern of messages allows for the receipt of two **filler**(...) messages, there is again a benefit in deviating within the pattern, violating the third principle. Since Dolev and Strong's optimization cannot be combined with our three principles, we do not apply it: in TRB+ (as in LTRB), nodes sometimes relay messages even though these messages will be discarded immediately by their recipients. The function **interesting(...)** in Figure 7.6 checks whether a message should be discarded or whether its value should be extracted. The function **valid(...)** (Figure 7.6), instead, checks whether the received message fits within the pattern of messages. Since we could not apply the optimization, some messages that fit within the pattern of messages will be discarded immediately.

The TRB+ protocol is designed to be IC-BFT so that it can tolerate rational nodes. The proof that rational nodes will choose to follow this protocol involves two steps. First, we establish that the estimated utility from following the protocol is positive by showing that, if all rational nodes follow the protocol, then TRB+ implements Terminating Reliable Broadcast, i.e. Validity, Agreement, Integrity, and Termination are satisfied. Then, we establish that unilaterally deviating from the protocol does not increase a rational node's utility, i.e. the protocol is a Byzantine Nash Equilibrium.

Proving that the estimated utility is positive The first step to proving the correctness of TRB+ when non-Byzantine nodes follow the protocol is to show that non-Byzantine nodes do not shun each other.

Lemma 57. *If nodes a and b both follow the TRB+ protocol, then $a \in G_b$ and $b \in G_a$.*

Proof. Since the protocol is symmetric, we only need to show that $a \in G_b$. Node a is added to G_b during initialization. We check every line that changes G_b and show that a is never removed. The first such instance is lines 18 and 24: since a follows the protocol, it will send the required messages on line 11. Line 43 is next; that line is never executed because a sends the required messages on line 31. The next line that modifies G_b is line 51. Since a follows the protocol, it sends the two required messages on lines 33–36. Both message satisfy **valid**$(\ldots)$ by construction. Finally, G_b could be changed at the end of the protocol, on line 62. This does not happen because the protocol guarantees that **SF**is never delivered if the sender follows the protocol. $\square$

Having shown that no node that follows the protocol shuns other nodes that follow the protocol, the rest of the correctness proof is inspired by the proof for Dolev and Strong's protocol [45].

Lemma 58. *If all but the f Byzantine nodes follow the protocol, and if a node receives a valid message m that contains a signature from non-Byzantine node r, then r has extracted m.*

Proof. To show that non-Byzantine nodes only forward values that they extracted, we look at the two only places where values are forwarded. First, a value can be forwarded because it was put in the *relay* set at line 21. In this case, the relayed value was extracted on line 20. Second, a value can be forwarded though line 50. In that case, again, the value was necessarily extracted (line 49). □

Lemma 59. *If all but the f Byzantine nodes follow the protocol, and if a non-Byzantine node r adds message m to its* relay *set, then all non-Byzantine nodes eventually extract m.*

Proof. There are two cases: node r can add to its *relay* set on round $i < f + 1$, or it can add to it on round $i = f + 1$.

In the first case, the protocol will execute at least one more round and since r follows the protocol, it will send $m{:}r$ to all other non-Byzantine nodes (line 34 and Lemma 57). The function **interesting**($m{:}r$) returns true for each node x that has not yet extracted m since **interesting**(m) held at r: since x has not extracted m, message m does not contain a signature from x (Lemma 58).

We show that the second case can only occur if some other non-Byzantine node added m to its *relay* set on round $j < f + 1$. A valid message received on round i contains i signatures (line 104). A valid message m received on round $f+1$, therefore, contains at least one signature from a non-Byzantine node. It follows that a non-Byzantine node extracted m in an earlier round (Lemma 58). □

Lemma 60. *If all but the f Byzantine nodes follow the protocol, and if a non-Byzantine node r extract m, then all non-Byzantine nodes extract m, or extract two distinct messages m' and m'' (or both).*

Proof. Node r extracts m on line 49 or 20. If node r has extracted fewer than two values at this point, then it will add m to its *relay* set (line 50 or 21) and therefore the conclusion holds (by Lemma 59).

If node r has already extracted two other values when it extracts m, then these other values m' and m'' were relayed (since then $|relay| < 2$) and other non-Byzantine nodes extracted them. $\square$

Lemma 61. *If all but the f Byzantine nodes follow the protocol, then TRB+ satisfies Validity.*

Proof. If the sender is correct and broadcasts m then it sends it on round 1. Every non-Byzantine node then extracts it at line 20.

The sender does not sign any other proposal value, so only messages with proposal m are considered **interesting**$(\ldots)$. Therefore non-Byzantine nodes do not extract any value other than m. All non-Byzantine nodes therefore only extract at most one value.

We can then conclude from Lemma 60 that all non-Byzantine node extract the same value, m, so they all deliver m. $\square$

Lemma 62. *If all but the f Byzantine nodes follow the protocol, then TRB+ satisfies Agreement.*

Proof. It follows directly from Lemma 60 that if any non-Byzantine node extracts more than one value, then all will. Further, if two non-Byzantine nodes extract different values, then all non-Byzantine nodes extract at least two values. The only possible outcomes therefore are: (i) no non-Byzantine node extracts a value, and all deliver **SF**; (ii) all non-Byzantine nodes extract more than one value, and all deliver **SF**; (iii) all non-Byzantine nodes extract the same value, m, in which case all non-Byzantine nodes deliver m. $\square$

Lemma 63. *If all but the f Byzantine nodes follow the protocol, then TRB+ satisfies Integrity.*

Proof. Integrity specifies that non-Byzantine nodes deliver a single message. This can be established directly by observing that **deliver** is only called once, at the end of the last round (lines 60–63). □

Theorem 22. *TRB+ satisfies Validity, Agreement, Integrity, and Termination in the presence of f Byzantine nodes if all other nodes follow the protocol.*

Proof. The previous lemmas establish Validity (Lemma 61), Agreement (Lemma 62), and Integrity (Lemma 63). Termination follows directly since there are no loops and no blocking calls in the protocol. □

Earlier we said that we only consider rational nodes with a utility function that gives them a net benefit from participating in the protocol. For completeness, we compute which value of α (as a function of β and γ) is necessary to ensure a benefit. Recall that we are considering the first of the two utility functions introduced in Section 7.5.2, where nodes only get a benefit when they have the role of sender. The utility u_r for node r is $\alpha b - (\beta s + \gamma t)$. When r does not have the role of sender, b is 0. Otherwise, b is 1 if he protocol σ_r followed by node r is such that Validity, Agreement, Integrity, and Termination hold when r is the sender, despite up to f Byzantine failures, if the other non-Byzantine nodes follow LTRB. s is the number of times that r signed a message and t is the total number of bytes in messages sent by r.

Lemma 64. *The estimated utility $\hat{u}_r$ of a rational node r following the TRB+ protocol is positive as long as $\alpha > TC(n - f - 1)$.*

Proof. We compute the costs associated with n instances of TRB+. There are three possible cases: (a) r is the sender, (b) r receives a *ticket* message, or (c) r does not receive a *ticket* message. In the instance where r is the sender, it must sign the proposal message and the ticket for each node in G_r (line 14), so $1+g$ signatures, where $g = |G_r|$. To each of the g nodes that it doesn't shun, r sends 3 numbers, 2 signatures, and a proposal. The cost to r on instances where it is sender is therefore $\beta(1+g) + \gamma g(3\eta_n + 2\eta_s + \eta_p)$.

In the g instances where r is not the sender and r receives a *ticket* message, r must generate 2 signatures (one for each message it sends, either the value in *relay* or the result of calling **filler**$(\ldots)$; lines 33–36). These two messages contain, in round i, a proposal, a number, and i signatures. Note that the **filler**$(\ldots)$ message has the same size as the *relay* message (line 130). These messages are sent to $g-1$ nodes (the nodes in G_r minus *sender*). There are f rounds, so the cost for signing and sending these messages is $z = \sum_{i=2}^{f+1}(2\beta + 2\gamma(\eta_p + \eta_n + i\eta_s)(g-1))$. In addition to this cost, r forwards the *ticket* message for an additional cost of $\gamma(2\eta_n + \eta_s)$

In the remaining $n-g-1$ instances where r is not the sender and r does not receive a *ticket* message, node r sends the same messages to the other nodes as it did if it had the *ticket*, for a cost of z, except that now the messages are sent to g nodes instead of $(g-1)$ since *sender*$\notin G_r$. In addition, r sends the **penance**$(\ldots)$ message to g nodes for a cost of $\beta s + \gamma g(\eta_n + \xi + \eta_s)$ (line 145). We are now in a position to determine ξ (as a function of g) by computing the total cost and choosing ξ so that the cost increases as g decreases.

The total cost over n instances, $TC(g)$, is, after simplifying z:

$$
\begin{aligned}
1 * \big(\beta(1+g) + \gamma g(3\eta_n + 2\eta_s + \eta_p)\big) && \text{(a)}\\
+g * \Big(2\beta f + 2\gamma(g-1)(f\eta_p + f\eta_n + f(f+3)\eta_s/2) && \text{(b)}\\
+ \gamma(g-1)(2\eta_n + \eta_s)\Big) &&\\
+(n-1-g) * \Big(2\beta f + 2\gamma g(f\eta_p + f\eta_n + f(f+3)\eta_s/2) && \text{(c)}\\
+ \beta s + \gamma g(\eta_n + \xi + \eta_s)\Big) &&
\end{aligned}
$$

The value of ξ is $(3\eta_n + 2\eta_s + \eta_p)/(n-1-g) + \delta/((n-1-g)g) - \eta_n - \eta_s$, with $\delta := (2\eta_n + \eta_s)n^2/4$ (lines 140–144).

To ensure a net benefit in participating in the protocol, α must be larger than $TC(g)$ for all choices of g. As TC increases as g decreases (cf. Lemma 65), it suffices to check that α is larger than TC for the case $|G_r| = (n-f-1)$.

□

Proving that there is no benefit in unilateral deviations To ensure that rational nodes will choose to follow the protocol, we combine predictability (each non-Byzantine node must send two messages) with accountability (nodes that fail to do so face consequences).

We implement local accountability through a *grim trigger* scheme [57]. In a grim trigger scheme, if a node x observes that some other node y does not cooperate, then x reacts by never cooperating with y again. We implement this scheme in TRB+ as follows. If a node p notices that some node q fails to send two distinct valid messages to p in any round, then p removes q from p's set G_p. Hence, p no

Leaving the pattern of messages	*Lemma*
Not participating at all	64
Sending fewer messages	66
Sending additional messages	67
Sending unexpected messages	68
Deviating within the pattern of messages	*Lemma*
Sending an in-pattern message that is shorter than specified	69
Sending an in-pattern message with fewer signatures than specified	69
Resending an in-pattern message to avoid computing a signature	69
Deviating from operations that do not impact the messages sent	70

Table 7.2: Space of deviations

longer sends messages to q. Just like tit-for-tat [57], grim trigger is only effective for iterative games with infinite horizons [15, 16]; in our case the protocol is executed infinitely often, and the sender is selected in round-robin fashion. Once the turn comes for p to be the sender, it will not send any message to q, including the *ticket* message. Not having a *ticket* message forces q to either send a **penance(...)** message in round 2, or not send messages that other nodes expect, resulting in more nodes shunning q. The **penance(...)** message is designed to be more expensive than sending all expected messages to p for n iterations of our protocol, so that it is in node q's best interest to send these messages rather than having to incur the cost of the **penance(...)** message.

To show that no deviation from the protocol increases the estimated utility, we examine all the possible deviations. The deviations are listed in Table 7.2. There are two categories of deviations: detectable and non-detectable. The first category contains all deviations that will be detected by at least one other node because the perpetrator deviated from the pattern of messages. There are only three ways to detectably deviate: not sending a message that should be sent, sending an additional message that should not have been sent, or sending a message with contents that differ from what the recipient expects. We examine these three detectable deviations in the next few lemmas. The high-level idea is that in each of these cases, the node that detects the deviation will shun the deviating node and this shunning will reduce the deviating node's estimated utility because, as explained in Lemma 65, once a node is shunned by another it must generate expensive **penance**(...) messages or risk being shunned by even more nodes. Being shunned by all would of course bring node r's benefit to 0 since at that point it is impossible for node r to broadcast its proposal.

Lemma 65. *If rational node r unilaterally deviates by shunning a node that is not known to be Byzantine, then node r's estimated utility does not increase.*

Proof. We prove that omitting messages reduces the utility by computing the amount of filler ξ that should be in a **penance**(...) message as well as the number s of digital signature operations necessary to generate a **penance**(...) message. We choose ξ and s such that the cost of the **penance**(...) messages that result from omitting a messages is larger that the savings from not sending the omitted messages.

The protocol specifies that a node r not send messages to nodes outside of its set G_r. Nodes are removed from G_r when they are observed to deviate from the protocol. If node r does not send any message to some non-Byzantine node x, then node x will shun r in return and, in particular, will not send it any *ticket* message when it is sender (line 11, $r \notin G_x$). As a result, node r will have to send an expensive **penance**(...) message to every node it does not shun (not sending the **penance**(...) message to a node causes it to shun r). Since node x shuns r for any omitted message, node r can omit all the messages that it would normally send to x. These omitted messages result in some savings for r.

To ensure that there is no incentive for node r to remove correctly-behaving nodes from G_r, we show that the estimated utility $\hat{u}$ decreases as correctly-behaving nodes are removed from G_r—even though a smaller G_r may mean that r sends fewer **penance(...)** messages.

The total cost over n instances, $TC(g)$ where $g = |G_r|$, was computed in Lemma 64. It is:

$$\begin{aligned}
1 * \big(\beta(1+g) + \gamma g(3\eta_n + 2\eta_s + \eta_p)\big) & \qquad \text{(a)}\\
+g * \Big(2\beta f + 2\gamma(g-1)(f\eta_p + f\eta_n + f(f+3)\eta_s/2) & \qquad \text{(b)}\\
+ \gamma(g-1)(2\eta_n + \eta_s)\Big) & \\
+(n-1-g) * \Big(2\beta f + 2\gamma g(f\eta_p + f\eta_n + f(f+3)\eta_s/2) & \qquad \text{(c)}\\
+ \beta s + \gamma g(\eta_n + \xi + \eta_s)\Big) &
\end{aligned}$$

We see that the number of digital signatures increases as g decreases if $s > 0$ (term (c)), so we can set $s = 1$. Remains to compute ξ so that the total number of bytes sent increases as g decreases. We consider only the γ factors in $TC(g)$. Let $\mathit{sender\_msg} = 3\eta_n + 2\eta_s + \eta_p$, $\mathit{nonsender\_msg} = f\eta_p + f\eta_n + f(f+3)\eta_s/2 + 2\eta_n + \eta_s$, and $\mathit{cost\_penance} = \eta_n + \xi + \eta_s$. We see that (as long as $\xi \geq \eta_n$):

$$TC(g) \leq g * \mathit{sender\_msg} + (n-1)g * \mathit{nonsender\_msg} + (n-1-1)g * \mathit{cost\_penance}$$

This cost would decrease as g decreases if *cost_penance* were not a function of g. To maintain the total cost to at least its initial level as g decreases, the following must hold:

$$(n-1-g)g * cost\_penance \geq (n-1-g) * sender\_msg + (n-1)(n-1-g) * nonsender\_msg$$

Solving for ξ: $g(\eta_n + \xi + \eta_s) = sender\_msg + (n-1) * nonsender\_msg + \delta$, so $\xi = \frac{1}{g}(sender\_msg + (n-1) * nonsender\_msg) + \frac{1}{g}\delta - \eta_n - \eta_s$. To ensure that $\xi > \eta_n$, we set $\delta = n(\eta_n + \eta_s)$. Our choice of ξ ensures that shunning non-Byzantine nodes reduces the estimated utility. □

Lemma 66. *If rational node r unilaterally deviates by sending only a subset of the required messages, then node r's estimated utility does not increase.*

Proof. If node r deviates by omitting even a single message, then the recipient x will shun r; node r will then have to send **penance(...)** messages when x is the sender. The effect is exactly the same as that described in Lemma 65, except with potentially additional costs if r still sends some messages to x. If follows directly that $\hat{u}_r$ decreases. □

Lemma 67. *If rational node r unilaterally deviates by sending additional messages, then node r's estimated utility does not increase.*

Proof. Sending an additional message to a non-Byzantine node p incurs an additional bandwidth cost. The effect of the additional message is at best nothing (round 1), or at worst that the recipient p will remove the sender from G_p (later rounds, see line 51), resulting in a reduced $\hat{u}_r$.

Sending an unexpected message to a Byzantine node cannot improve that Byzantine node's worst-case behavior (if anything, it may help drive the system to an even less pleasant outcome). Therefore, no rational node sends an additional message to another node—Byzantine or not. □

The final detectable deviation is to send a message with contents that detectably deviate from what is expected, so that it will not be accepted by the recipient—meaning that it will not match what the recipient expects.

Lemma 68. *If rational node r unilaterally deviates by sending a message that will not be accepted, then node r's estimated utility does not increase.*

Proof. Sending such a message to a non-Byzantine node p always causes the sender to be removed from G_p (lines 18 or 24 for the first round, lines 43 and 51 for subsequent rounds). This removal directly results in a reduced estimated utility. $\square$

We have seen that no detectable deviation can increase node r's estimated utility. We now show that the same holds for non-detectable deviations. There are two kind of non-detectable deviations: either sending a different message that is still within the pattern of messages, or deviating from steps of the protocol that are not related to the sending of messages.

Lemma 69. *If rational node r unilaterally deviates by sending a message other than what the protocol requires, yet that message is be accepted, then node r's estimated utility does not increase.*

Proof. There is only one way for r to send a deviant message that is accepted: send another message in the pattern of messages instead of the one that the protocol requires. This can reduce cost if (i) the other message is shorter or requires fewer digital signatures, or (ii) r can reuse a previous digital signature.

Except for **penance**(...) messages, at every step, all messages that can be received in the pattern of messages have the same size and the same number of signatures, so a non-detectable deviation cannot reduce either cost. Line 19 shows that all proposals must be padded to the same size, line 46 shows that two proposals or **filler**(...) messages must be forwarded every round. Proposals and **filler**(...) messages have the same cost: $\beta + \gamma(\eta_n + \eta_p + i\eta_s)$ at round i.

The pattern of messages allows for two different-sized message at the beginning of the first round: node r must send either the *ticket* message it received from the sender, or the **penance**(...) message. The *ticket* message is cheaper to send. However, the *ticket* message includes a digital signature from the sender. Since node r cannot forge signatures, the only way to stay in the pattern of messages if r did not receive a *ticket* message is to send the **penance**(...) message.

We have seen that node r cannot reduce the number of bytes or digital signatures it has to send. Next we show that it must generate afresh each of the digital signatures that it must send. Each message is unique because messages grow in length with each round, and each message is tagged with the instance number k. No two messages are identical; in particular, the check at line 46 ensures that the two forwarded messages are distinct. Signatures cannot be reused, so a rational node r cannot increase its estimated utility by deviating from the protocol but sending messages that are accepted. □

The last possible deviation would be to change the behavior in a way that does not affect the messages that are sent.

Lemma 70. *If rational node r unilaterally deviates in a way that does not change the messages that are sent, then node r's estimated utility does not increase.*

Proof. None of the actions outside of generating or sending messages result in any cost for the nodes, so changing them will not decrease the estimated utility. □

We are now in a position to prove that the TRB+ protocol is BAR-Tolerant.

Theorem 23. *The TRB+ protocol is IC-BFT.*

Proof. We show that the TRB+ protocol satisfies the definition of IC-BFT (Definition 17).

1. TRB+ implements Terminating Reliable Broadcast despite up to f Byzantine nodes (Theorem 22).

2. TRB+ is a Byzantine Nash Equilibrium. The lemmas above show that (i) rational nodes do not benefit from leaving the system rather than participating in the protocol (Theorem 22 and Lemma 64), (ii) rational nodes do not benefit from unilaterally deviating from the protocol in a way that is detectable (Lemmas 65 to 68), and (iii) rational nodes do not benefit from unilaterally deviating in a way that is not detectable (Lemmas 69 and 70). This covers all possible unilateral deviations.

□

7.6 Related Work

Game theory not only describes games where the players have perfect knowledge, but also games with imperfect knowledge. In a *Bayesian game* [57], for example, each node is of a given *type* and that type determines its utility function. Nodes do not know the type of the other nodes: instead, they only know a probability distribution τ_i over the type profiles. For example, this distribution could encode the fact that two thirds of the nodes are of type "A" and the remainder are of type "B", but the nodes do not know beforehand which nodes are "A" and which are "B". Different nodes could have different probability distributions, reflecting unequal knowledge about the situation.

In a Bayesian game, a strategy and probability distribution profile $(\vec{\sigma}, \vec{\tau})$ is a *Bayesian Nash Equilibrium* [57] if no node r can improve its estimated utility by modifying its strategy unless either another node also changes strategy, or τ_r changes. In a Bayesian game, the estimated utility $\hat{u}_r$ is computed by taking the probability distribution profile τ_r into account. If the node is risk-neutral, the estimated utility is simply the expected utility. A risk-averse node may choose instead to consider the minimum over the utilities it may receive depending on the types of the other nodes. In our TRB+ protocol, nodes can be seen as having two types: rational or Byzantine. Altruistic nodes do not need a separate type because TRB+ is IC-BFT, so both rational and altruistic nodes follow the protocol. We can then say that a protocol is a Byzantine Nash Equilibrium if it is a Bayesian Nash Equilibrium for every possible τ_r that is consistent with our assumptions about Byzantine failures.

The field of *mechanism design* [112] studies how to maximize the *social welfare*, a function of the outcome that indicates how beneficial the outcome is to society as a whole (instead of the individual players). In mechanism design, the designer modifies the *outcome* function so that, when the rational nodes act to maximize their individual utility, the resulting social welfare is also maximized. This approach is useful in contexts where the protocol designer has control over what should happen in response to the nodes' actions, for example in auctions. There are similarities between our approach to building BAR-Tolerant protocols and mechanism design, but we differ on a fundamental point: in our approach to designing BAR-Tolerant protocols, we cannot change the *outcome* function: the outcome observed by the nodes is a direct result of their interactions, which in turn depend only on each nodes' s strategy. Since we cannot manipulate *outcome*, instead we manipulate the initial strategy σ that we propose to the nodes, with the goal of influencing their final strategy choice.

Several researchers have begun to examine constructing incentive compatible systems [5, 121, 146, 151]. There has been some success in applying mechanism design to distributed systems in routing [6, 52, 53] and caching [37]. Unfortunately, these efforts generally ignore Byzantine behavior and adopt a model where every node is rational.

Another limitation of some previous work is that it aims for incentive compatibility in a loose sense of the term, by which a protocol simply includes some level of incentive to encourage proper behavior. For example, BitTorrent [39] includes heuristics aimed at providing incentives, but researchers have identified ways in which a rational node can circumvent these ad-hoc techniques [40, 147]. In contrast, in this dissertation we have made sure to apply the term *incentive compatible* only to systems where each node's strategy is optimal, so rational nodes have no incentive to deviate from the protocol.

After our original publication of the BAR model [7], others have proposed models that would be appropriate for cooperative services. Abraham et al. [1] propose several equilibria from game theory that may apply to a distributed systems setting. They explore collusion with their k-resilient equilibrium concept, and Byzantine deviations with their concepts of t-immune and (k, t)-robust equilibrium. They differ from our model in that they consider finite games (rather than repeated games), and their motivation for the non-rational nodes in their model is that some nodes (which they call "altruistic") will deviate from the protocol in ways that these nodes believe are beneficial to the system as a whole. Our model, instead, separates the notion of altruistic nodes (in our case these nodes follow a specially tailored protocol that is beneficial to the system) from non-rational nodes (the Byzantine nodes, whose utility function is unknown and whose deviation may aim to help or harm the system).

Moscibroda et al. [116] analyze the interaction of rational and Byzantine nodes in the context of a specific one-shot game, the virus inoculation game. They measure the "price of anarchy" and the "price of malice", defined respectively as the impact of having rational nodes (instead of all altruistic) and having Byzantine and rational nodes (instead of all rational). Like us, they find that the presence of Byzantine nodes can simplify the design of BAR-Tolerant protocol when the nodes are risk-averse.

Chapter 8

Conclusion

This dissertation answers several questions about how to build practical Byzantine fault-tolerant systems. Byzantine fault-tolerance is attractive because it makes no assumption about the behavior of faulty nodes. The three main areas we explored are: (i) reducing the cost of BFT in terms of the number of nodes and number of communication steps, (ii) using BFT to maintain confidentiality in addition to availability and integrity, and (iii) using BFT in large distributed systems with no central administrator (cooperative services). The dissertation only introduces cooperative services and the BAR model; we have also [7] built a BAR-Tolerant cooperative backup service, where nodes store backup data on each other and the protocol ensures that there is no benefit from selfish behavior such as deleting data stored on behalf of other nodes.

In this dissertation we make the following contributions.

- A new lower bound for the number of nodes needed to implement a safe register that can tolerate f Byzantine failures: $3f+1$ (Section 3.3).

- A new protocol, Listeners, that matches this lower bound without requiring digital signatures and provides an atomic register (Section 3.4).
- A new protocol, Byzantine Listeners, that provides the same guarantees as Listeners despite Byzantine clients (Section 3.5).
- A new semantics, non-confirmable safe (respectively, regular or atomic), that can be achieved with fewer nodes than its confirmable counterpart with the only drawback of not informing clients of when their writes complete (Section 4.2).
- A new lower bound ($2f + 1$) for the number of nodes needed to implement a non-confirmable safe register that can tolerate f Byzantine failures (Section 4.3).
- A new protocol, Non-confirmable Listeners, that matches this lower bound and provides non-confirmable regular semantics (Section 4.4).
- A new dynamic quorum primitive, DQ-RPC, that allows both the set of nodes N and the number of tolerated failures f to change over time (Section 5.5).
- A new protocol, U-Dissemination with DQ-RPC, that provides atomic semantics with a dynamic N and f and only needs $3f + 1$ nodes (Section 5.4).
- A new way to think about state machine replication, by separating agreement from execution (Chapter 6).
- A replicated state machine that requires only $2f+1$ execution nodes to tolerate f Byzantine nodes, instead of $3f + 1$ previously (Section 6.3).

- A new lower bound on two-step consensus: any asynchronous consensus protocol that tolerates f Byzantine failures and completes in two communication steps in the common case despite t Byzantine failures must use at least $3f + 2t + 1$ nodes (Section 6.5).

- A new two-step consensus protocol, FaB Paxos, that matches the lower bound (Section 6.6).

- A new protocol, the Privacy Firewall, that introduces new confidentiality guarantees to replicated state machines (Section 6.10).

- A new model, BAR, that accurately describes cooperative services (Section 7.2).

- A new protocol, TRB+, that implements terminating reliable broadcast despite Byzantine, altruistic, and rational nodes—even in the absence of any altruistic node (Section 7.5).

We have shown in this dissertation how to build asynchronous Byzantine fault-tolerant replicated state machines and registers that provably use the minimal number of nodes, and how to reach consensus in the minimal number of communication steps. We have then explored two faulty behaviors that, although theoretically covered by the Byzantine model, are not handled by traditional BFT protocols.

The first such behavior comes from a hacker intent on stealing secrets. Although both register and replicated state machine protocols maintain integrity and availability in the face of this behavior, neither protects the confidentiality of the data when even a single node is faulty. At the root of the tension between confidentiality and fault-tolerance lies replication: each node has a copy of the data, so increasing replication can weaken confidentiality by creating more copies of the information that should be confidential. Both register and state machine protocols were designed to maintain integrity and agreement despite Byzantine failures, but not to maintain confidentiality. We resolved this tension by designing a new protocol, the Privacy Firewall, that guarantees safety and output set confidentiality despite f Byzantine nodes. We also determine the minimal number of nodes needed to provide output set confidentiality and show that the Privacy Firewall meets this lower bound.

The second behavior that is problematic for BFT protocols is that of nodes controlled by a selfish user who wants his computer to use as few resources as possible helping others. Our initial motivation for exploring this behavior was to design a cooperative backup service: there, nodes may want to free up disk space by deleting data that was entrusted to them. Although BFT protocols can tolerate up to f such nodes, in a cooperative service there is no central administrator to control the nodes, so it is possible that all nodes act selfishly. We move beyond the Byzantine model and introduce the BAR model to describe these environments. To show that it is possible to design interesting protocols in the BAR model, we derive a new terminating reliable broadcast protocol and prove that it is a Byzantine Nash equilibrium. The new protocol, in addition to the customary number of Byzantine nodes allowed in solutions based on traditional Byzantine fault-tolerance, tolerates also an arbitrary number of selfish nodes. Our cooperative backup service [7] (not discussed in the thesis) requires $3f + 2$ nodes to tolerate f Byzantine nodes and it can tolerate an arbitrary number of selfish nodes. Even though each assumption is a vulnerability, in the case of cooperative services, lack of assumptions is a liability.

Appendix A

Dynamic Quorums

A.1 Dissemination Protocol

A.1.1 Quorum Intersection Implies Transquorums

We have shown in Section 5.4.2 that U-dissemination provides atomic semantics for any TRANS-Q operation that has the transquorums property. The proof also follows for the hybrid dissemination protocol [152] since it follows the same schema. In this section, we show that the traditional implementation of Q-RPC (using quorum intersection) satisfies the transquorums property.

Both the u-dissemination and the crash protocol are special cases of the hybrid dissemination protocol. All three use quorums of size $\lceil \frac{n+b+1}{2} \rceil$ and requiree at least $2f+3b+1$ servers to tolerate f crash failures and b Byzantine failures from the servers (a total of $f+b$ failures). Any number of clients may crash. In the case of the U-dissemination protocol, f is zero. In the case of the crash protocol, b is zero.

The client protocol is shown in Figure 5.1. Servers store the highest-timestamped value they have received that has a valid signature (except for the crash protocol in which signatures are not necessary).

There must be at least $2f+3b+1$ servers. Servers do not communicate with each other; clients use the Q-RPC operation to communicate with servers. The Q-RPC operation sends a given message to a responsive quorum of servers.

Any two quorums intersect in $2q-n=b+1$ servers. At least one of these servers, s, is not Byzantine faulty (and has not crashed).

We use the same ordering o as Section 5.4.2, namely $\mathcal{W}$ calls are ordered according to their arguments, and $\mathcal{R}$ and $\mathcal{T}$ calls are ordered according to their return value. No $\mathcal{R}$ quorum operation ever returns $\perp$, so we do not need to consider that case. We prove the timeliness and soundness conditions separately.

Lemma 71 (timeliness)**.** *For the quorum size and ordering described above:* $\forall w \in \mathcal{W} \forall r \in \mathcal{R} \cup \mathcal{T} : w \rightarrow r \implies o(w) \leq o(r)$

Proof. The quorum to which the value was written with w intersects with the quorum from which r reads in one non-Byzantine server that has not crashed. That server will report the timestamp that was written in w; since the server is not Byzantine faulty that data has a valid signature. The ϕ function will therefore return a value that is at least as large as $o(w)$. The result of that function is equal to $o(r)$. □

Lemma 72 (soundness)**.** *For the quorum size and ordering described above:* $\forall r \in \mathcal{R} : \exists w \in \mathcal{W}$ *s.t.* $r \not\rightarrow w \wedge o(w) = o(r)$

Proof. Values selected through $\phi(\text{Q-RPC}_{\mathcal{R}})$ have a valid signature (by definition of ϕ). We know that valid values returned by $\mathcal{R}$ must come from a $\mathcal{W}$ operation since only $\mathcal{W}$ quorum operations introduce new values. Since these signatures cannot be faked, it follows that the $\mathcal{W}$ quorum operation w did not happen after after r. □

This proves that the dissemination protocols in Figure 5.1 are atomic when using the traditional Q-RPC.

A.2 Fault-Tolerant Dissemination View Change

Let $s_i := encrypt(\langle i, N, f, m, t, g, pub\rangle_{adm}, priv, k_i^t)$. The administrator sends $\langle(N, f, m, t, g), s_0 \ldots s_{n-1}\rangle_{admin}$ to a responsive quorum of new servers and then a responsive quorum of old servers.

New servers forward that message to the old servers, causing them to end the old view. The old servers acknowledge right away but they also start a new thread with which they send that message back to a responsive quorum of new servers. The new servers proceed as before (Figure 5.7), namely they wait for an acknowledgement from a quorum of old servers before joining the ready state in which they acknowledge to the administrator and tag their responses with the new view.

As a result, if a single correct old server ends view t then eventually a quorum of new servers will have received the message for the new view $t+1$. That is enough to guarantee that view $t+1$ has matured, so reads in the new view will go through. If on the other hand no old correct server ends view t then reads in view t will go through. Since in the event of an administrator crash the old servers are not turned off, in both cases the system will continue to process reads and writes and provide atomic semantics. If the view change does not include a generation change then the server transitions directly to the ready state.

The careful reader will have noticed that if a single faulty server in the old view has the view certificate for the new view but no correct server in the new view does (which may happen if faulty servers collude and the administrator crashes after sending its first message), the faulty old server can cause our implementation of DQ-RPC to block because the clients will try to get answers from the new servers even though the new servers do not process requests yet. However, the implementation of DQ-RPC that we describe in the optimizations Section (5.5.4) does not have this problem and will allow reads and writes to continue unhampered because the old view has not ended and DQ-RPC can process its replies.

A.3 Generic Data

A.3.1 Masking Protocols with Transquorums

In this section we show that the U-masking protocol provides partial-atomic semantics despite up to b Byzantine faulty servers. This protocol assumes that the network links are asynchronous authenticated and fair. Clients are assumed to be correct and the administrator machine may crash.

The U-masking protocol is shown in Figure A.1. The only change from its original form [126] is that we have substituted TRANS-Q for Q-RPC operations.

Partial-atomic semantics: All reads R either return $\perp$, or return a value that satisfies atomic semantics.

Theorem 24. *The U-masking protocol provides partial-atomic semantics if the Q-RPC operation it uses has the transquorums properties for the function o defined below.*

Proof. We define o and O in the exact same way as we did for the dissemination protocol in Section 5.4.2. Then we show the following three properties:

1. $X \rightarrow W \implies O(X) < O(W)$ and $W \rightarrow X \implies O(W) < O(X)$

2. $O(W_1) = O(W_2) \Longrightarrow W_1 = W_2$

3. Read R returns either $\perp$ or the value written by some W such that

   (a) $R \not\rightarrow W$, and

   (b) $\nexists W' : O(W) < O(W') < O(R)$

The first two points show that O defines a total order on the writes and that the ordering is consistent with "happens before". The third point shows that reads return the value of the most recent preceding write.

We prove that the protocols satisfy partial-atomic semantics by building an ordering function O for the read and write operations that satisfies the requirements for partial-atomic semantics.

Both read and write end with a $\mathcal{W}$ quorum operation w. The first quorum operation in writes never returns $\perp$. By the first property of transquorums, that operation therefore has a timestamp that is at least as large as that of w. The write operation then increases that timestamp further, ensuring that $X \rightarrow W \Longrightarrow O(X) < O(W)$. Our construction of the mapping O ensures that if a read happens after a write, then that read gets ordered after the write. These two facts imply property (1).

The o value includes the *writer_id*, which is different for each writer - and if the same writer performs two writes then (1) implies that they'll have different values. Therefore property (2) holds: $O(W_1) = O(W_2) \Longrightarrow W_1 = W_2$. These two properties together show that writes are totally ordered in a way that is compatible with the happens before relation.

Next we show that non-aborted reads return the value of the preceding write (property (3)). Soundness tells us that this value does not come from an operation that happened after R (3a). We know that the value returned by reads must come from a write operation since only writes can introduce new values that are reported by $b+1$ servers: so the value returned by a read R comes from some write W. Note that $O(R)$ and $O(W)$ have the same ts, $writer\_id$ and D; they only differ in the last element (so $O(W) < O(R)$). Thus, any write $W' > W$ will necessarily also be ordered after R since $O(W') > O(R)$ (3b). $\square$

If the Q-RPC operations have the *non-triviality property* that $\mathcal{R}$ quorum operations that are concurrent with no other quorum operation never return $\perp$, then U-masking has the property that reads that are not concurrent with any operation never return $\perp$ either. This follows directly from the fact that if no operation is concurrent with a read R then no quorum operation is concurrent with any of R's quorum operations. Our implementation of DQ-RPC has the non-triviality property.

A.3.2 DQ-RPC for Masking Quorums

In this section we show how to build the DQ-RPC and view change protocol for masking quorums, when data is not signed. Only one line needs to change: line 5 of ViewTracker's **consistentQuorum** (Figure 5.9), shown below.

$$\textbf{if } |recentMessages| < 2 * m\_maxMeta.f + 1 : \text{return } (\emptyset, \perp)$$

Thus read operations now wait until they get $2f+1$ servers vouching for the current generation instead of $f+1$. It follows that $f+1$ correct servers have entered the new generation, so they will be able to countermand any old value proposed by servers that have not finished the view change.

The view change protocol must be modified however, because as described in Section 5.5.3 it relies on the fact that the servers' read of the current value never

```
read() :
1.  $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")          // reply is of the form $(ts, writer\_id, data)$
2.  $r := \phi(Q)$
    // $\phi$ : the only non-countermanded value vouched by $b+1$ servers, or $\perp$
3.  if $r == \perp$ : return $\perp$
4.  $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$r$)
5.  return $r.data$

write($D$) :
1.  $Q$ := TRANS-Q$_{\mathcal{T}}$("GET_TS")
2.  $ts := max\{Q.ts\} + 1$
3.  $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE",$(ts, writer\_id, D)$)
```

Figure A.1: U-masking protocol for correct clients

fail. This is not true in the case of masking quorums, where reads may fail if some write is concurrent with it.

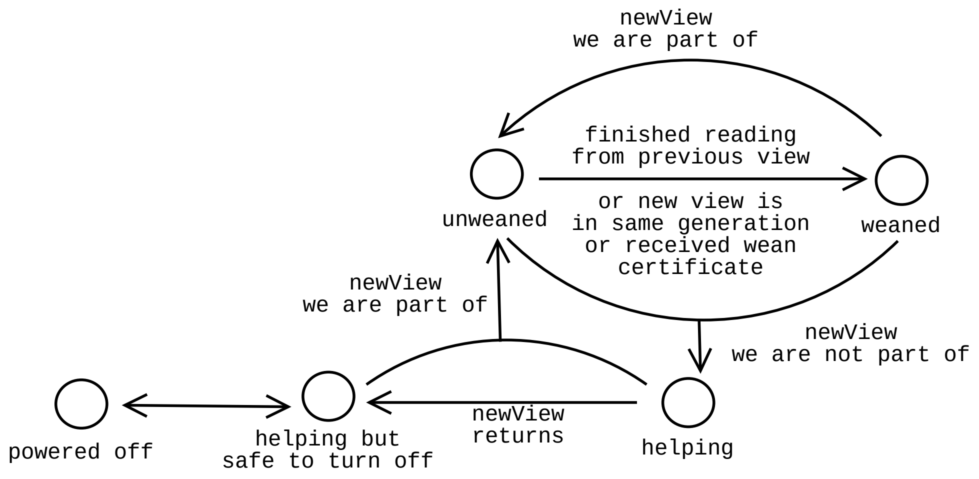

Figure A.2: Server transitions for the masking protocol

Figure A.3 gives the view change protocol for the administrator. If clients are correct then the function is guaranteed to eventually terminate. If no write is concurrent with the view change then the administrator only goes through the loop

```
newView() :
1. Give their view certificate to a quorum in the new view
2. Give info about the new view to a quorum in the old view
3. repeat :
4.     $a$ := read on old view
5.     $b$ := read on new view
6. until $(a \neq \perp \vee b \neq \perp)$
7. Generate wean certificate (“old view is gone now”)
8. Write $max(a, b)$ to a quorum in the new view
9. Write the wean certificate to a quorum in the new view
```

Figure A.3: View change protocol for masking quorums

once. Once the newView operation returns, it is safe to turn off the machines in the old view that are not part of the new view – we say that the new view is *weaned*.

In order to provide atomic semantics, we must ensure that reads reflect the values written previously, and thus we must propagate data from the old view to the new one. The view change protocol allows clients to query the new servers right away, before the administrator copies any data. How can this work?

The key is that (as shown in Figure A.5) the new servers will get their data from the old ones to service client requests. Once a new server has read some value from the old servers it never needs to contact the old servers again since writes are directed to the new ones (we say that the server is *weaned*). Once enough new servers have data stored locally, it is possible to shut down the old servers – we must just be careful that nothing bad happens to new servers that were in the middle of reading from the old ones.

So the new servers, when they are asked for data that they don’t have, first check whether the old servers are still available by checking whether a peer server has a wean certificate (using the READ_LOCAL call). If the server receives a wean certificate, it knows that there is no point in trying to contact the old servers: the server then returns whatever local data it has, possibly $\perp$. If there is no wean certificate then the server forwards the request to the old servers. If the old servers have been shut down in the mean time then this request may take forever; that's OK because the old servers are only turned off if the administrator completed successfully, and in that case the **waitForWean**(...) function will eventually stop any read thread that is stuck in this manner.

The **waitForWean**(...) function periodically queries the peers to see if they have a wean certificate. This ensures that if the new view is weaned then eventually all servers will know about it (or move on to an even more recent view).

When new unweaned servers receive a write request, they make sure that the old view has ended, then store the data and acknowledge. But servers do not consider themselves weaned as a result of a write. So when someone tries to read that data, the servers will still try to contact the servers in the old view to make sure the local data is recent enough.

The servers go through different states, as described in Figure A.2. A server that is not part of the current view is in the *helping* state. In that state it responds to queries but tags them with the most recent view certificate, thus directing clients to more recent servers. When a server receives a new view certificate (and the server is part of the new view), it moves on to the *unweaned* state. It accepts requests from clients right away and starts **waitForWean**(...) in a parallel thread to detect when the system becomes weaned. Read requests are forwarded to the old servers; if a non-$\perp$ reply can be determined then that reply is stored locally before being forwarded to the client and the server moves on to the *weaned* state. Servers will also move to *weaned* when they receive a wean certificate from their peers.

Server i's variables

m_D	the current data
m_ts	the data's timestamp (initially -1)
m_meta	current view meta-information: $(N, f, m, t, g, pubKey)$
m_oldMeta	meta-information for the previous view: $(N, f, m, t, g, pubKey)$
m_cert	admin certificate for (*m_meta*)
m_priv	private key matching certificate
m_weanCert	certificate that the view in *m_meta* is weaned
m_serverWeaned	true if the server is weaned (initially false)
m_oldEnded	true if the server knows that the old view ended (initially false)

Figure A.4: Server variables for masking quorums

A.3.3 DQ-RPC Satisfies Transquorums for Masking Quorums

We now show that DQ-RPC also satisfies transquorums when we use the masking quorum's ϕ operation. Recall that that ϕ returns the value that is vouched for by $f+1$ servers and that is not countermanded, or $\perp$ if there is no such value.

Lemma 73. *The masking DQ-RPC operations are timely.*

Proof. Recall that timeliness means $\forall w \in \mathcal{W}, \forall r \in \mathcal{R} \cup \mathcal{T}, o(r) \neq \perp \ : \ w \rightarrow r \Longrightarrow o(w) \leq o(r)$. The proof is similar to that for the dissemination case. If w and r picked views in the same generation then the two quorums intersect in at least $f+1$ correct servers. Since w happened before r and servers never decrease the timestamp they store, it follows that $o(r) \neq \perp \Rightarrow o(w) \leq o(r)$.

If w picked a view t that is in the previous generation from r's view (say v), then we consider the last view u in t's generation. As we have seen in the previous paragraph, non-aborted reads from a quorum $q(u)$ in u will result in a timestamp that is at least as large as $o(w)$. Since r picked a view that is in a more recent generation than u, it follows that r received $2f(v)+1$ replies in v's generation (so at least one correct). Correct servers in the new view only respond to a read request

```
write(ts,D) :
1.  if (m_ts<ts) : (m_ts,m_D) := (ts,D)
2.  if not m_oldEnded : askOldView()
    // m_oldEnded holds at this point
3. return "OK"

read() :
1.  if (m_serverWeaned ∨ m_weanCert ≠ ⊥) : return (m_ts,m_D)
2.  if askPeers() : return (m_ts,m_D)
3.  if askOldView() : return (m_ts,m_D)
    // not m_serverWeaned and m_weanCert == ⊥, and the read from the old servers failed
4.  return (−1, ⊥)

readLocal() :
1.  return m_weanCert

private askOldView() :
1.  Q':=Q-RPC("READ+HELP",m_cert) to a quorum of servers in m_oldMeta.N
2.  m_oldEnded := true
3.  if $\phi(Q') \neq \perp$ :
4.      m_serverWeaned := true
5.      if (m_ts, m_D) < $\phi$(Q') " (m_ts,m_D) := $\phi$(Q')
6.  if $|\{m \in Q' : m\_ts < m.ts\}| < m\_oldMeta.f + 1$ :
7.      m_serverWeaned := true
8.  return m_serverWeaned ∨ m_weanCert ≠ ⊥

private askPeers() :
1.  Q':=Q-RPC("READ_LOCAL") to a quorum of servers in m_meta.N
2.  if any response includes a valid wean certificate cert for this view :
3.      m_oldEnded := true
4.      m_weanCert := cert
5.  return m_serverWeaned ∨ m_weanCert ≠ ⊥

private waitForWean() // started on its own thread when the server hears of the new view
1.  while (not m_serverWeaned) ∧ (m_weanCert == ⊥) :
2.      askPeers()
3.      wait for some time
4.  kill any read() or write() thread that is waiting for the old servers
```

Figure A.5: Server protocol for masking quorums

until they know that either they or their view has weaned. It follows that the replies in r contained at least $f(v)+1$ replies C that are at least as recent as the highest-timestamped value whose write completed in view u, which in turn is at least $o(w)$. So if r were to pick any value such that $o(r) < o(w)$ then that value would be countermanded by C. Therefore $o(r) \neq \perp \Rightarrow o(w) \leq o(r)$.

It is not possible for w to pick a view in a later generation than what r picks if $w \rightarrow r$ since $\mathcal{R}$ masking DQ-RPCs wait until any previous generation has ended. This concludes our proof that $\forall w \in \mathcal{W}, \forall r \in \mathcal{R}, o(r) \neq \perp \ : \ w \rightarrow r \implies o(w) \leq o(r)$. □

Lemma 74. *The masking DQ-RPC operations are sound.*

Proof. Soundness requires that $\forall r \in \mathcal{R}, o(r) \neq \perp : \exists w \in \mathcal{W}$ s.t. $r \not\rightarrow w \wedge o(w) = o(r)$.

Correct servers only respond to read queries with data that was previously written – either directly to them or to the previous quorum. The ϕ function ensures this property by only accepting values that are vouched for by $f+1$ servers. □

Theorem 25. *DQ-RPC satisfies transquorums even if the old servers are taken offline after the newView call returns.*

The masking DQ-RPC also tolerates crashes from the administrator: all operations still eventually complete as long as the servers from the old view are *not* taken offline.

Lemma 75. *If some view t never ends then all quorum operations to that view eventually complete.*

Proof. An individual server s responds to read quorum operations once either it receives a reply from the old servers or it knows that its view is weaned (in that latter case the server only responds after the client resends its query). If the administrator failed then the old servers are not taken offline and thus they will eventually respond. If the administrator did not fail then eventually s will know that its view is weaned.

The wait quorum operation waits on the same conditions, thus individual servers will eventually respond to write quorum operations. The help and read_local operations do not block on anything, thus they will complete trivially. □

It follows from the above lemma that as long as some view stays around long enough, all DQ-RPC operations will complete.

Theorem 26. *DQ-RPC satisfies transquorums even if the administrator crashes during a view change, as long as both the old and the new servers are kept online.*

A.3.4 Masking Protocols for Byzantine Faulty Clients

We now turn our attention to a variant of the U-masking protocol that can handle Byzantine failures from the client. We use Phalanx [101] with an improved ϕ function that provides partial-atomic semantics (the original Phalanx only provides safe semantics). The client code is shown in Figure A.6. Phalanx does not require the clients to have public-private key pairs, but the servers do. In step 4 of the **write**(...) operation, the clients collect signatures from the servers (the *echoes*). Servers only accept writes if they are accompanied by a quorum of valid signatures. For write-backs, servers require $f + 1$ of a different type of signature. Notice that the signature step is neither timely nor sound: the signatures' only purpose is to make the write call succeed.

It is natural to ask why we consider Byzantine faulty clients. After all, nothing prevents faulty clients from continuously writing incorrect values. However, in many practical situations such faulty clients would eventually be identified and

```
**read** :
1.  $Q$ := TRANS-Q$_{\mathcal{R}}$("READ")
2.  $r$ := $\phi(Q)$
    // largest non-countermanded triple vouched for by at least $f+1$ servers
3.  **if** $r$==$\perp$ : **return** $\perp$
4.  let $V$ be $f+1$ valid signatures for $r$ taken from $Q$
5.  $Q$ := TRANS-Q$_{\mathcal{W}}$("WRITE-BACK",$r$, $V$)
6. **return** $r.data$

**write**($D$) :
1.  $Q$ := TRANS-Q$_{\mathcal{T}}$("READ_TS")
2.  $ts$ := $max\{Q.ts\}+1$
3.  let $m$ := $(ts, writer\_id, D)$
4.  $Q$' := TRANS-Q("SIGN",$m$)
5.  let $V$ be a quorum of valid signatures for $m$ taken from $Q$'
6.  $Q$" := TRANS-Q$_{\mathcal{W}}$("WRITE",$m, V$)
```

Figure A.6: Masking quorum or hybrid-m for Byzantine faulty clients

removed from the system. More to the point, our goal here is to show that the DQ-RPC operation can make many protocols dynamic. We have included the protocol of Figure A.6 for completeness.

The protocol guarantees partial-atomic semantics, meaning that all reads that return a value satisfy atomic semantics and reads that are not concurrent with any other operation always return a value.

Appendix B

Two-Step Consensus

B.1 Approximate Theorem Counterexample

In his "approximate theorem" 3a [86], Lamport states: "If there are at least two proposers whose proposals can be learned with a 2-message delay despite the failure of Q acceptors, or there is one such possibly malicious proposer that is not an acceptor, then $N > 2Q + F + 2M$." Here, N is the number of acceptors; M is the number of failures despite which safety must be ensured; and F is the number of failures despite which liveness must be ensured. The paper indicates that the term "approximate theorem" was chosen because there are special cases where the bounds do not hold. The paper does not include a specific special case for approximate theorem 3a, but that approximate theorem does not hold in systems where learners never fail.

In these systems, only $3f + 1$ acceptors are necessary to tolerate f Byzantine failures and be able to learn in two message delays despite up to f Byzantine failures (3a instead predicts that $5f+1$ acceptors would be needed). Learners learn v if $2f+1$ acceptors say they have accepted it. Since any two quorums of size $2f + 1$ intersect in a correct acceptor, no two learners will learn different values. If the leader is faulty then it is possible that no value is learned. In that case, a leader election, and the new leader asks the learners for the value that they have learned. Since learners are all correct, the new leader can wait for all learners to reply with a signed response. The leader can therefore choose a value to propose that will maintain the safety properties. Since the learners' answers are signed, the new leader can forward them to the acceptors to convince them to accept a new value.

Bibliography

[1] I. Abraham, D. Dolev, R. Gonen, and J. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multi-party computation. In *Proc. 25th PODC*, pages 53–62, July 2006.

[2] I. Abraham and D. Malkhi. Probabilistic quorums for dynamic systems. In *Proc. 17th DISC*, pages 113–124, Oct. 2003.

[3] E. Adar and B. A. Huberman. Free riding on Gnutella. *First Monday*, 5(10):2–13, Oct. 2000.

[4] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th OSDI*, pages 1–14, Dec. 2002.

[5] M. Afergan. *Applying the Repeated Game Framework to Multiparty Networked Applications*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Aug. 2005.

[6] M. Afergan and R. Sami. Repeated-game modeling of multicast overlays. In *IEEE INFOCOM 2006*, Apr. 2006.

[7] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, pages 45–58, Oct. 2005.

[8] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):996–1007, 2001.

[9] L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright. Dynamic Byzantine quorum systems. In *DSN*, pages 283–292, 2000.

[10] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.*, 37(4):418–425, 1988.

[11] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[12] R. J. Aumann. Subjectivity and correlation in randomized strategies. *Journal of Mathematical Economics*, 1(1):67–96, 1974.

[13] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC 77*, pages 149–155, Nov. 1977.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[15] R. Axelrod. *The Evolution of Cooperation.* Basic Books, New York, 1984.

[16] R. Axelrod. The evolution of strategies in the iterated prisoner's dilemma. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 32–41. Morgan Kaufman, 1987.

[17] M. Baker. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California at Berkeley, 1994.

[18] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. 13th SOSP*, pages 198–212, 1991.

[19] R. Baldoni, C. Marchetti, and S. Tucci Piergiovanni. Asynchronous active replication in three-tier distributed systems. In *Proc. 2002 Pacific Rim International Symposium on Dependable Computing*, pages 19–26, Dec. 2002.

[20] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.

[21] R. Bazzi and Y. Ding. Non-skipping timestamps for byzantine data storage systems. In *Proc. 18th DISC*, pages 405–419, Oct. 2004.

[22] R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proc. 16th PODC*, pages 259–266, Aug. 1997.

[23] R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing Journal*, 14(1):41–48, Jan. 2001.

[24] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *Advances in Cryptology—EUROCRYPT 97*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192. Springer-Verlag, May 1997.

[25] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, SRC, Palo Alto, CA, USA, 10 1993.

[26] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Reconstructing Paxos. *SIGACT News*, 34(2):42–57, 2003.

[27] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[28] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM Trans. on Computer Systems*, pages 18–36, Feb. 1990.

[29] 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, Sept. 1998.

[30] R. Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, Weizmann Institute of Science, 1995.

[31] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, Feb. 1999.

[32] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proc. 4th OSDI*, pages 273–288, Oct. 2000.

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

[34] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Comm. ACM*, 24(2):84–90, 1981.

[35] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell. The Rio file cache: Surviving operating system crashes. In *Proc. 7th ASPLOS*, pages 74–83, 1996.

[36] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[37] B.-G. Chun, K. Chaudhuri, H. Wee, M. Barreno, C. H. Papadimitriou, and J. Kubiatowicz. Selfish caching in distributed systems: a game-theoretic analysis. In *Proc. 23rd PODC*, pages 21–30. ACM Press, 2004.

[38] A. Clement, J. Napper, L. Alvisi, and M. Dahlin. BAR games. Technical Report 06-25, University of Texas at Austin Computer Sciences, May 2006.

[39] B. Cohen. The BitTorrent home page. http://bittorrent.com.

[40] B. Cohen. Incentives build robustness in BitTorrent. In *First Workshop on the Economics of Peer-to-Peer Systems*, June 2003.

[41] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proc. 19th SOSP*, pages 120–132, 2003.

[42] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[43] Y. Desmedt. Threshold cryptography. *European Trans. on Telecommun*, 5(4):449–457, July/August 1994.

[44] DoD. Trusted computer system evaluation criteria. 5200.28-STD, Dec. 1985.

[45] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *Siam Journal Computing*, 12(4):656–666, Nov. 1983.

[46] S. Dolev, S. Gilbert, N. Lynch, A. Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in mobile ad hoc networks. In *Proc. 17th DISC*, pages 306–320, Oct. 2003.

[47] J. R. Douceur. The Sybil attack. In *Proc. 1st IPTPS*, pages 251–260. Springer-Verlag, 2002.

[48] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report EPFL/IC/200499, EPFL, Feb. 2005.

[49] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[50] K. Eliaz. Fault tolerant implementation. *Review of Economic Studies*, 69:589–610, Aug 2002.

[51] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, 1985.

[52] J. Feigenbaum, C. Papadimitriou, R. Sami, and S. Shenker. A BGP-based mechanism for lowest-cost routing. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 173–182. ACM Press, 2002.

[53] J. Feigenbaum, R. Sami, and S. Shenker. Mechanism design for policy routing. In *Proc. 23rd PODC*, pages 11–20. ACM Press, 2004.

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

[55] R. Friedman, A. Mostefaoui, and M. Raynal. Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems. *IEEE Transactions on Dependable and Secure Computing*, 2(1):46–56, Jan. 2005.

[56] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. A decentralized algorithm for erasure-coded virtual disks. In *Proc. DSN-2004*, pages 125–134, June 2004.

[57] D. Fudenberg and J. Tirole. *Game theory.* MIT Press, Aug. 1991.

[58] J. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2):363–389, July 2000.

[59] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM Press, 1979.

[60] Gnutella. http://www.gnutella.com/.

[61] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proc. DSN-2004*, pages 135–144, June 2004.

[62] J. Gray. Notes on data base operating systems. In *Advanced Course: Operating Systems*, pages 393–481, 1978.

[63] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.

[64] J. Harsanyi. A general theory of rational behavior in game situations. *Econometrica*, 34(3):613–634, Jul. 1966.

[65] R. Haurwitz. Computer records on 197,000 people breached at UT. *Austin American Statesman*, Apr. 2006.

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

[67] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[68] M. Herlihy and J. D. Tygar. How to make replicated data secure. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 379–391, 1988.

[69] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or how to cope with perpetual leakage. In *Proc. of the 15th Annual Internat. Cryptology Conf.*, pages 457–469, 1995.

[70] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[71] B. Huberman and R. Lukose. Social dilemmas and internet congestion. *Science*, 277:535–537, July 1997.

[72] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: the bell tolls? *IEEE Distributed Systems Online*, 6(6), June 2005.

[73] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distributed Computing*, 12(4):209–223, Sept. 1999.

[74] A. Iyengar, R. Cahn, C. Jutla, and J. Garay. Design and Implementation of a Secure Distributed Data Repository. In *Proc. of the 14th IFIP Internat. Information Security Conf.*, pages 123–135, 1998.

[75] A. D. Joseph, F. A. deLespinasse, J. A. Tauber, D. K. G. ifford, 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.

[76] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):1018–1032, 2003.

[77] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *Software Engineering*, 12(1):96–109, Jan. 1986.

[78] L. Kong, A. Subbiah, M. Ahamad, and D. Blough. A reconfigurable Byzantine quorum approach for the agile store. In *Proc. 22nd SRDS*, pages 219–228, Oct. 2003.

[79] T. Kontzer. United Airlines computer snafu being investigated. *InformationWeek*, Jan. 2006.

[80] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th ASPLOS*, pages 190–201, Nov. 2000.

[81] K. Kursawe. Optimistic Byzantine agreement. In *Proc. 21st SRDS*, pages 262–267, Oct. 2002.

[82] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

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

[84] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[85] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, Dec. 2001.

[86] L. Lamport. Lower bounds for asynchronous consensus. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 22–23, June 2002.

[87] L. Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research, July 2004.

[88] L. Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[89] L. Lamport and M. Masa. Cheap paxos. In *Proc. DSN-2004*, pages 307–314, June 2004.

[90] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[91] M. Li and Y. Tamir. Practical Byzantine fault tolerance using fewer than 3f+1 active replicas. In *Proc. 17th ISCA*, pages 241–247, Sept. 2004.

[92] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX ATC*, pages 29–41, june 2003.

[93] LimeWire. http://www.limewire.com/.

[94] J.-L. Lions, D. Luebeck, J.-L. Fauquernbergue, G. Kahn, W. Kubbat, S. Levedag, L. Mazzini, D. Merle, and C. O'Halloran. Ariane 5 flight 501 failure, report by the inquiry board, July 1996.

[95] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. In *Proc. 13th SOSP*, pages 226–238, 1991.

[96] B. Littlewood, P. Popov, and L. Strigini. Modelling software design diversity: a review. *ACM Computing Surveys 33(2):177-208*, pages 177–208, June 2001.

[97] W. Lloyd. *Two Lectures on the Checks to Population.* Oxford University Press, 1833.

[98] N. Lynch and A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. 16th DISC*, pages 173–190, Oct. 2002.

[99] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Sustaining cooperation in multi-hop wireless networks. In *NSDI*, May 2005.

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

[101] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th SRDS*, pages 51–58, Oct. 1998.

[102] D. Malkhi, M. Reiter, and N. Lynch. A correctness condition for memory shared by byzantine processes. Unpublished manuscript, Sept. 1998.

[103] D. Malkhi, M. Reiter, and A. Wool. The load and availability of byzantine quorum systems. In *Proc.16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 249–257, August 1997.

[104] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. 19th SOSP*, pages 44–59. ACM Press, 2003.

[105] M. Marsh and F. B. Schneider. Codex: A robust and secure secret distribution system. *IEEE Trans. Dependable Sec. Comput.*, 1(1):34–47, 2004.

[106] J.-P. Martin and L. Alvisi. Minimal byzantine storage. Technical Report TR-02-38, Dept. of Computer Sciences, UT Austin, Aug. 2002.

[107] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 04), DCC Symposium*, Florence, Italy, June 2004.

[108] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 05), DCC Symposium*, pages 402–411, Yokohama, Japan, June 2005.

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

[110] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *16th International Conference on Distributed Computing, DISC 2002*, pages 311–325, Oct. 2002.

[111] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 02), DCC Symposium*, pages 374–383, June 2002.

[112] A. Mas-Colell, M. D. Whinston, and J. R. Green. *Microeconomic Theory*. Oxford University Press, 1995.

[113] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th SOSP*, pages 124–139, Dec. 1999.

[114] J. McLean. A general theory of composition for a class of possibilistic properties. *IEEE Transactions on Software Engineering 22(1)*, pages 53–67, Jan. 1996.

[115] S. Misel. Wow, AS7007! NANOG mail archives http://www.merit.edu/mail.archives/nanog/1997-04/msg00340.html.

[116] T. Moscibroda, S. Schmid, and R. Wattenhofer. When selfish meets evil: Byzantine players in a virus inoculation game. In *Proc. 25th PODC*, pages 35–44, July 2006.

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

[118] M. Naor and A. Wool. Access control and signatures via quorum secret sharing. In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 157–168, 1996.

[119] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, 1998.

[120] J. Nash. Non-cooperative games. *The Annals of Mathematics*, 54:286–295, Sept 1951.

[121] C. Papadimitriou. Algorithms, games, and the internet. In *Proc. 33rd STOC*, pages 749–753. ACM Press, 2001.

[122] J.-F. Paris and D. Long. Voting with regenerable volatile witnesses. In *Proceedings. Seventh International Conference on Data Engineering*, pages 112–119, Apr. 1991.

[123] D. C. Parkes. *Iterative Combinatorial Auctions: Achieving Economic and Computational Efficiency.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, May 2001.

[124] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM, 27(2):228-234*, Apr. 1980.

[125] 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.

[126] E. Pierce and L. Alvisi. A framework for semantic reasoning about byzantine quorum systems. In *Brief Announcements, Proc. 20th Symp. on Principles of Distributed Computing (PODC)*, pages 317–319, Aug. 2001.

[127] M. Reiter. The Rampart toolkit for building high-integrity services. In *Dagstuhl Seminar on Dist. Sys.*, pages 99–110, 1994.

[128] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

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

[130] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *Proc. 18th SOSP*, pages 15–28. ACM Press, Oct. 2001.

[131] R. Rodrigues and B. Liskov. A correctness proof for a byzantine-fault-tolerant read/write atomic memory with dynamic replica membership. Technical Report MIT CSAIL Technical Report TR/920, Massachusetts Institute of Technology, 2003.

[132] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT CSAIL Technical Report TR/932, Massachusetts Institute of Technology, 2003.

[133] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *SIGOPS European Workshop*, Sept. 2002.

[134] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th SOSP*, pages 188–201. ACM Press, 2001.

[135] A. S. S. Gilbert, N. Lynch. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proc. DSN-2003*, pages 259–268, June 2003.

[136] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan. 2003.

[137] M. Sachs and A. Varma. Fibre channel and related standards. *IEEE Communications*, 34(8):40–50, Aug. 1996.

[138] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, Apr. 1997.

[139] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.

[140] F. B. Schneider. What good are models and what models are good? In *Distributed Systems, 2nd ed.*, chapter 2, pages 17–25. Addison Wesley, July 1993.

[141] 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):1318–1335, Oct. 1991.

[142] Secure hash standard. Federal Information Processing Standards Publication (FIPS) 180-2, Aug. 2002.

[143] Seti@home. http://setiathome.ssl.berkeley.edu/.

[144] M. Shand and J. Vuillemin. Fast implementations of rsa cryptography. In *Proc. 11th Symposium on Computer Arithmetic*, pages 252–259, June 1993.

[145] C. Shao, E. Pierce, and J. L. Welch. Multi-writer consistency conditions for shared memory objects. In *Distributed Computing — DISC 2003*, Lecture Notes in Computer Science, pages 106–120. Springer-Verlag, Nov. 2003.

[146] J. Shneidman and D. C. Parkes. Specification faithfulness in networks with rational nodes. In *Proc. 23rd PODC*, pages 88–97. ACM Press, 2004.

[147] J. Shneidman, D. C. Parkes, and L. Massoulie. Faithfulness in internet algorithms. In *Proc. PINS*, pages 220–227, Portland, USA, 2004.

[148] A. Stephenson et al. Mars climate orbiter mishap investigation board phase I report, Nov. 1999.

[149] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: protecting data in compromised systems. In *Proc. 4th OSDI*, pages 165–180, Oct. 2000.

[150] Q. Sun, D. Simon, Y.-M. Wang, W. Russell, V. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings. 2002 IEEE Symposium on Security and Privacy*, pages 1–30, 2002.

[151] E. Tardos. Network games. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 341–342. ACM Press, 2004.

[152] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *Proc. 7th SRDS*, pages 93–100, 1988.

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

[154] G. Tsudik. Message authentication with one-way hash functions. In *INFOCOM*, pages 2055–2059, May 1992.

[155] A. Venkataramani, R. Kokku, and M. Dahlin. TCP Nice: a mechanism for background transfers. *SIGOPS Oper. Syst. Rev.*, 36(SI):329–343, 2002.

[156] U. Voges and L. Gmeiner. Software diversity in reacter protection systems: An experiment. In *IFAC Workshop SAFECOMP79*, May 1979.

[157] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. 18th SOSP*, pages 230–243, Oct. 2001.

[158] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, pages 253–267. ACM Press, Oct. 2003.

[159] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. In *ACM Transactions on Computer Systems 20,4*, pages 329–368, Dec. 2000.

Vita

Jean-Philippe Martin was born in Geneva, Switzerland in 1976. He attended the Swiss Federal Institute of Technology (EPFL) and was awarded a B.S. in computer science. After working for a year with InMotion Technologies (Switzerland) he joined the University of Texas at Austin under the supervision of Dr Lorenzo Alvisi. He received a M.S. in 2004.

Permanent Address: 15, ch. du Feuillet
CH-1255 Veyrier
Switzerland

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.