Self-Adjusting Quorum Systems For Byzantine Fault Tolerance

by

Evelyn Tumlin Pierce, B.A., M.Sc.

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 2000

To Dai

Acknowledgments

First, I would like to thank my advisor, Lorenzo Alvisi, for helping me find my path and keep my enthusiasm up when that path seemed very long. I would also like to thank Dahlia Malkhi of the Hebrew University of Jerusalem and Michael Reiter of Lucent Technologies for their cooperation and guidance in this work, and for the pioneering research that inspired it.

Thanks also to Edsger W. Dijkstra, for teaching me a new standard of beauty and elegance in reasoning; all that is best and clearest in this thesis owes something to his moral influence. In addition, I would like to thank Allen and Leisa Emerson, Ben Kuipers, Jay Misra, and Rebecca Wright, as well as the members of my dissertation committee and a host of other mentors, colleagues, friends and family without whose support this work might never have reached fruition.

Last and foremost, I would like to thank my husband, love, colleague and best friend David Pierce, for being there for me throughout the long journey and beyond.

Evelyn Tumlin Pierce

The University of Texas at Austin
December 2000

Self-Adjusting Quorum Systems For Byzantine Fault Tolerance

Publication No. _____

Evelyn Tumlin Pierce, Ph.D.
The University of Texas at Austin, 2000

Supervisor: Lorenzo Alvisi

The purpose of this work has been to design protocols for a data service that tolerates Byzantine server faults by shifting between multiple tolerance modes at run time, monitoring itself for faulty server behavior and dynamically adjusting its own tolerance capabilities accordingly. Our goal is a system that runs in an efficient low-fault mode most of the time, but can adjust itself to cope with new faults as they occur.

Our approach is based on the *masking quorum systems* of Malkhi and Reiter. Like the protocols originally proposed for these systems, our techniques are relatively economical in that updates and reads need to be performed only at a subset ("quorum") of data servers, thus decreasing the workload on individual servers and simplifying the recovery process when failures occur. However, our protocols implement the following additional capabilities:

1. Strengthened read and write protocols for serializability of completed operations.

2. Protocols for dynamically adjusting the threshold (and thus quorum size) of threshold masking quorum systems in response to information about the number and/or probability of faults in the system.

3. Protocols for detecting Byzantine server behavior in threshold masking quorum systems.

As a subsidiary goal, we have sought to make our methods not only effective against random faulty server behavior, but also resistant to sabotage by a deliberate adversary. To this end, we have limited our use of standard assumptions, such as independence of failures and *a priori* limits on the number of faults, that restrict the applicability of many Byzantine fault tolerance techniques to questions of security.

Contents

Chapter 1

Introduction

As the world grows more and more interconnected, distributed data services are becoming simultaneously more necessary and more problematic. They are necessary in that a growing number of users are conducting business transactions and research online, thus increasing the demand for a variety of data that must be accessible from a variety of locations. They are problematic in that such data must be protected from accidental or malicious corruption while maintaining this high level of availability.

At minimum, a reliable data service ought to tolerate (i.e., store and retrieve data correctly in spite of) server crashes and network partitions. If the stored information is particularly critical, or if servers are particularly vulnerable to corruption or sabotage, it may also be necessary for the service to tolerate actual incorrect or misleading information from a subset of its servers.

This assortment of possible misbehaviors is modeled by the *Byzantine fault model*. Under this model, a faulty data server may respond to queries incorrectly (corruption), be unresponsive (crash or partition), or even sometimes respond correctly. In short, there are no real restrictions on its behavior. A data service that can tolerate server faults of this type is likely to be extremely robust. Unfortunately, Byzantine fault tolerance techniques typically require numerous up-to-date copies of each data item at various locations, i.e., replication. Such techniques are apt to be expensive, and are thus often dismissed as too impractical to be used as a precaution against a relatively rare, if potentially devastating, condition.

The purpose of this work has been to design asynchronous data service protocols that address this concern by allowing the service to shift between multiple tolerance modes at run time, monitoring itself for faulty server behavior and dynamically adjusting its own tolerance capabilities accordingly. Our goal is a system that runs in an efficient low-fault mode most of the time, but can adjust itself to

cope with new faults as they occur.

As a subsidiary goal, we have striven to make our methods not only effective against random faulty server behavior, but also resistant to sabotage by a deliberate adversary. To this end, it is necessary to limit the use of standard assumptions, such as independence of failures and *a priori* limits on the number of faults, that would limit the applicability of our techniques to questions of security. While it may be impossible to eliminate such assumptions altogether (see note at the end of this chapter), we rely less heavily on them than do many traditional solutions to the problem of Byzantine fault tolerance.

1.1 Contributions of This Thesis

Our approach is based on *masking quorum systems*, first proposed by Malkhi and Reiter [MR98]. Masking quorum systems are an attractive alternative to earlier approaches to Byzantine fault tolerance in that read and write operations need to be performed only at a subset ("quorum") of the servers, thus decreasing the workload on individual servers and simplifying the recovery process when failures occur.

Using masking quorum systems as an economical alternative for Byzantine fault tolerance requires a number of enhancements to the original approach, however. For one thing, the original read and write protocols of [MR98] do not guarantee the integrity of read operations that are concurrent with one or more write operations. Furthermore, such systems may still be expensive in that tolerating multiple faults requires large quorums.

The results presented in this document fall into three main categories:

1. Strengthening of the read and write protocols for masking quorum systems for serializability of completed operations

2. Protocols for dynamically adjusting the threshold (and thus quorum size) of threshold masking quorum systems in response to information about the number and/or probability of faults in the system

3. Protocols for detecting Byzantine server behavior in threshold masking quorum systems.

1.2 Related Work

The foremost established approach to Byzantine fault-tolerant data services is that of *state machine replication*, which is surveyed in detail in [Sch93]. In this approach, as in ours, each data item is stored by all members of a set of n data servers. In state machine replication, however, all copies must be kept current, and special algorithms are needed to ensure that all read and write operations on a given data item are performed in the same order. A client of such a service reads a variable value by sending a request to all n servers, and accepting a response if and only if it is returned by a sufficient number of them, e.g., $b+1$ for a fault tolerance threshold of b. In spite of the overhead required to ensure the consistent ordering of operations as well as of recovering failed servers to the current system state, the state machine approach has been implemented numerous times, e.g., in Rampart [Rei95], usually with a fault-tolerance limit of $\lfloor \frac{n-1}{3} \rfloor$.

Castro and Liskov have recently developed a less costly variant of state machine replication, using checkpointing and view changes with occasional garbage collection [CL99]. They have further strengthened their approach by means of various features including efficient and proactive state recovery for servers that may have failed [CL00]. However, keeping the server states consistent across the system remains a necessity.

In this work we take a sharply constrasting approach to those described above by eliminating the need for all servers to maintain identical states or to process identical sequences of operations. Instead, our protocols specify that each read and write operation is performed on a subset (quorum) of the server set; the set of possible quorums is defined to have specific intersection properties that enable clients to determine the current value of any given data item. A correct system state thus inherently includes some out-of-date servers, thus simplifying recovery – provided a quorum of servers is up to date, additional failed servers may be brought back online without state update.

We summarize work related to each of our specific results in the corresponding chapters.

1.3 Structure of Thesis

Chapter 2 contains the system model and definitions that will be used throughout this thesis, including the original masking quorum system definitions and protocols from [MR98].

In Chapter 3 we give an abstract definition of a class of variables called *TS-variables*, to which masking quorum system variables belong. We give rigor-

ous definitions of Lamport's semantic properties of *safeness*, *regularity*, and *atomicity* [Lam86] for these variables, where safeness describes the semantics of the original protocols of [MR98] and atomicity describes the semantics of full serializability. We show how the problem of an atomic read and write protocol for masking quorum systems can be solved given a solution to the weaker problem of a regular protocol for such systems, and present a non-live solution to the latter problem, i.e., one in which certain read operations may abort. Finally, we give a simple set of defining properties that are common to the protocols we have studied, and show that *any* regular protocol with these properties is inherently non-live in an asynchronous environment without concurrency control.

In Chapter 4, we present protocols to change the size of the quorums of a threshold masking quorum system so that the system's degree of fault tolerance (i.e., threshold) can be adjusted between a set minimum and maximum without blocking ordinary read and write operations. We show how to modify the read/write protocols for threshold masking quorum systems, including both the original protocols of [MR98] and the enhanced protocols presented in Chapter 3, to use the dynamically defined quorums. We show that each of the new dynamic protocols maintains the semantics of its static counterpart. This chapter is adapted from [AMPRW00].

As a complement to these protocols, in Chapter 5 we present statistical methods for monitoring the approximate number of faults in threshold masking quorum systems, as well as specifically identifying some of the faulty servers. A version of this chapter appears in published form as [AMPR00].

Chapter 6 consists of concluding remarks and directions for future work.

The appendix to this document contains additional material that is closely related to the results presented in Chapters 4 and 5 and appears in the papers from which those chapters are adapted. It is included for completeness, but is primarily the work of other co-authors of these papers.

A note on security

Some researchers (e.g. [DR86]) have suggested that Byzantine fault tolerance is applicable to problems of security. Such applicability would provide a powerful motivation for the use and further exploration of techniques such as those presented in this thesis.

Unfortunately, Byzantine fault-tolerant protocols that depend on variations of a voting algorithm, such as those in this thesis as well as those of the more established approach of state machine replication are vulnerable to the worst-case scenario of Byzantine server failure: simultaneous unanimous corruption of most or all of the set of replicated servers. This vulnerability leads to one of the most

cogent objections to applying the Byzantine fault model to security: the presence of a malign agent may make this worst-case scenario into an intelligently pursued goal rather than a vanishingly unlikely statistical phenomenon.

In our view, however, this objection is not fatal; although such techniques should probably not be regarded as a *preventative* against security breaches, they can be designed to make such breaches considerably more difficult than they would otherwise be, and may thus be considered a *deterrent*. One of the goals of this work has been to keep our techniques as free as possible from assumptions that would be questionable in the presence of an intelligent adversary, such as independence of failures. This is, in fact, an additional motivation for the elimination of static fault tolerance thresholds (Chapter 4).

Chapter 2

System Model and Definitions

In this chapter we describe our system model and give some of the historical definitions on which the results described in this thesis are based.

2.1 System Model

Our system consists of a set S of n server processes and an unspecified number of client processes. Server processes maintain values for a set of variables, serving and updating them in response to *read* and *write* requests received from client processes. Every server maintains a value for every variable in the system. Each client is connected asynchronously to each server via a two-way FIFO channel; no other channels are required or used by our protocols.

Each server $s \in S$ stores and updates the value of each variable V in conjunction with an associated timestamp; thus, at each server, the variable state at the time a read or write request is received can be expressed as a pair $\langle val, ts \rangle$; we call this pair an *image* of V.

We stipulate that timestamps have the following properties, which justify the use of the word "timestamp" in this asynchronous context. These properties are implemented by the read and write protocols, which we will describe later.

- Each client selects the timestamp values associated with updates to a particular variable in a monotonically increasing fashion.
- The timestamp field of the image of a particular variable is monotonically nondecreasing at each server.

Servers may be either *correct* or *faulty*. A correct server behaves according to its specification, i.e., it returns its current image of a given variable upon request.

In contrast, a faulty server may behave in a *Byzantine* (arbitrary) fashion, e.g., by serving incorrect or outdated data or by failing to respond to queries. Clients and channels are assumed to be reliable for the scope of this work.

2.2 Definitions: Byzantine Quorum Systems

Mathematically, a *quorum system* is a set of pairwise intersecting sets:

Definition 1 *A* quorum system *on a set X is a set $\mathcal{Q} \subseteq 2^X$ such that*

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

A member of $\mathcal{Q}$ is known as a quorum.

A data service based on the server set S can be designed as a quorum system as follows: let $\mathcal{Q} \subset 2^S$ be a quorum system, and let each read and write operation be issued to and performed on some quorum Q of $\mathcal{Q}$. Hereafter we use the term "quorum system" and the label $\mathcal{Q}$ to refer both to the mathematical construct as defined above and to a data service based on the construct. This slight overloading is straightforward, and should not result in confusion.

If the set of servers is subject only to crash failures (i.e., unresponsive servers), the intersection property of $\mathcal{Q}$ ensures that each read operation has access to the value of the "most recently completed" write operation in some serialization of write operations.

A note on operations and concurrency

A read or write *operation* is performed by a client on a quorum of servers according to a specification called a *protocol*. We will present several read and write protocols in this document, all of which share the same basic structure: an operation is initiated when a client issues a corresponding *request* to the servers in a quorum, and is completed when the steps of the protocol have all been executed.

As indicated above, each read and write operation in a quorum system is performed on multiple servers. As we are assuming an asynchronous environment without concurrency control mechanisms, it is possible for two or more such operations to be *concurrent*:

Definition 2 *Operations op_a and op_b are said to be* concurrent *if there exists a pair of servers $s_1, s_2 \in S$ such that s_1 processes op_a before op_b while server s_2 processes op_b before op_a.*

In proving the semantic properties of variables maintained by quorum systems, we must take such concurrency into account; we handle this problem in detail in Chapter 3.

A note on data semantics

To analyze the correctness of our protocols, we will follow the example of [MR98] by using Lamport's notions of *safe*, *regular*, and *atomic* data semantics [Lam86], adapted to allow for the possibility of concurrent write operations. We give specific technical definitions of these concepts in Chapter 3, but the basic intuitions are as follows:

- **safe**: A read/write protocol for a variable is considered *safe* if it ensures that any read operation that is not concurrent with any write operation on the same variable returns the value of the most recent write to that variable.[1]
- **regular**: A protocol is *regular* if it is safe *and* ensures that any read operation that is concurrent with one or more write operations on the same variable returns either the value of the most recent write or the value of one of the concurrent writes to that variable.
- **atomic**: A protocol is *atomic* if it ensures that all read and write operations on a given variable are serializable, i.e., that they behave as though they were performed sequentially in some order that is compatible with the partial order in which they are actually performed.

Note that under a regular protocol, one read may occur later than another, yet return the value of an earlier write; thus regularity does not imply atomicity.

Our use of these terms may appear to differ from that of [Lam86] in that we treat these properties as belonging to protocols rather than to data objects. This difference is illusory, however; a data object has a given semantic property if and only if the protocol used to access it has that property.

2.2.1 Masking quorum systems

If servers may be subject to arbitrary (Byzantine) failures, the minimal intersection guaranteed by a quorum system as defined in Definition 1 is not enough to guarantee data consistency: two quorums might intersect in a single server that returned a faulty response, so that a read operation does not receive the correct up-to-date

[1] The monotonicity of timestamps at each server ensures that writes are serializable by timestamp; thus we may refer to the write with the highest timestamp as the "most recent" write.

value. For such services, *masking quorum systems* [MR98] have been proposed. We summarize the definition of these systems here.

During any given read or write operation, a server set that is subject to Byzantine failures may be partitioned into two subsets: those that behave according to their specification throughout the operation, and those that do not. Members of the latter set, which we will usually call F, may fail to respond to requests, or may respond with out-of-date or arbitrary data. We refer to such a set F as a *failure configuration.*

A masking quorum system is defined in terms of a *failure pattern*, which is a set $\mathcal{B} \subset 2^S$ such that

$$\forall B_1, B_2 \in \mathcal{B}, B_1 \not\subset B_2$$

In operational terms, the failure pattern defines the set of failure configurations that the system is designed to tolerate. Specifically, the system is expected to behave correctly as long as its failure configuration is contained in some element of $\mathcal{B}$, i.e., if:

$$\exists B \in \mathcal{B} : F \subset B$$

The definition of a masking quorum system is as follows:

Definition 3 *A masking quorum system on a server set S and failure pattern $\mathcal{B}$ is a quorum system $\mathcal{Q} \subseteq 2^S$ such that:*

- $\forall Q_1, Q_2 \in \mathcal{Q}, \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \backslash B_1 \not\subseteq B_2$
- $\forall B \in \mathcal{B}$, $\exists Q \in \mathcal{Q} : B \cap Q = \emptyset$

If the set F of faulty servers during any given operation is contained in some element of $\mathcal{B}$, and all read and write operations are performed on elements of $\mathcal{Q}$, then the two bullets above ensure that:

- the intersection of any pair of quorums (e.g., a read quorum and the previous write quorum) contains a set of nonfaulty servers whose response, if unanimous, can be identified as correct.
- at least one quorum is always available even if faulty servers are nonresponsive.

In this work we will be concerned primarily with a family of masking quorum systems called *b-masking quorum systems*, defined below. Here and for the remainder of this dissertation, we will use the notation $\#(X)$ to denote the size of a set X.

Definition 4 *A* b-masking quorum system *on a server set S is any masking quorum system defined on the failure pattern*

$$\mathcal{B} = \{B \subset 2^S : \#(B) = b\}$$

This failure pattern encodes the assumption that all server faults are contained within some set of size b, i.e., that there are no more than b faults in the system during a given operation. (This is the standard "threshold" assumption common to many techniques for Byzantine fault tolerance.) In such systems, the requirement that

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

means simply that the minimum intersection size between quorums is $2b+1$ servers.

A b-masking quorum system of particular interest to this work is the *Uniform quorum system* [MR98],[2] whose definition is elegantly parameterized in terms of b:

Definition 5 *The* Uniform quorum system *for a server set S ($\#(S) = n$) with threshold b ($n \geq 4b+1$) is the set $\mathcal{Q}$, where*

$$\mathcal{Q} = \{Q \subset S : \#(Q) = \lceil \frac{n+2b+1}{2} \rceil\}$$

It is easily seen that this definition satisfies Definition 3. For simplicity in our discussions, we will assume hereafter that n is odd, so that we may eliminate the ceiling operator $\lceil\rceil$ from our formulae and calculations.

Read and write protocols

The basic read and write protocols for masking quorum systems, as originally proposed in [MR98], consist of the following steps:

Write: For a client to write value v to variable V, it performs the following steps:

1. For some $Q \in \mathcal{Q}$ and $B \in \mathcal{B}$, query all servers in the set $Q \setminus B$ to obtain the set A of images of V at those servers.[3]
2. Select a new timestamp t greater than the largest timestamp appearing in any element of A and greater than any timestamp previously chosen by the client. (To avoid timestamp collisions, every client c chooses its timestamps from a unique set $\mathcal{T}_c$, where $c \neq c' \Rightarrow \mathcal{T}_c \cap \mathcal{T}_{c'} = \emptyset$.)
3. Send each server in some quorum Q' a write request containing an identifier for V and the new image $\langle v, t \rangle$.

A server that receives such a write request performs the update if and only if the timestamp of new image is greater than the timestamp of the current image of V. The timestamp of V at any correct server is therefore monotonically nondecreasing.

[2] In [MR98], this type of system was called a *threshold quorum system*; the name was changed for clarity in [AMPR00].

[3] Note that such a set necessarily intersects all quorums in at least one correct server.

Read: For a client to read variable V, it performs the following steps:

1. Query all servers in some quorum Q to obtain the set A of images of V at those servers.
2. Determine the set $A' \subseteq A$ of images that are *vouched for* according to the definition below.
3. Select the member of A' with the highest timestamp, and return the corresponding value. If A' is empty, return $\perp$ (a null value).

Definition 6 *A set B^+ of servers is called a* voucher set *for failure pattern $\mathcal{B}$ if*

$$\forall B \in \mathcal{B} : B^+ \not\subseteq B$$

An image is vouched for *during a variable read if it is returned by all members of some voucher set. A value is vouched for if it appears in an image that is vouched for.*

Essentially, a voucher set is any set of servers that cannot all be faulty. If a voucher set agrees on an image, that image is not fabricated, though it may be out-of-date.

Example: For a b-masking quorum system, a voucher set is any set of $b + 1$ or more servers.

Remark: In our discussions of various protocols, we will sometimes refer to *the* voucher set for a successful read operation; by this we mean the voucher set containing all servers that return the image accepted by the read.

The read/write protocol above has safe semantics (Section 2.2). As it does not specify the behavior of read operations that are concurrent with writes, however, safeness is a relatively weak semantic property. For systems that have access to reliable third-party authentication protocols, stronger properties can be implemented by means of *dissemination quorum systems* [MR98].

2.2.2 Dissemination quorum systems

Like masking quorum systems, dissemination quorum systems are defined in terms of a failure pattern $\mathcal{B}$. They are very similar to masking quorum systems in structure as well, except that the intersection between each pair of quorums only needs to contain at least *one* correct server:

Definition 7 *A dissemination quorum system on a server set S and failure pattern $\mathcal{B}$ is a set $\mathcal{Q} \subseteq 2^S$ such that:*

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

Again, the following special case is worth separate mention:

Definition 8 *A* Uniform dissemination quorum system *on a server set S and failure pattern $\mathcal{B} = \{B \subset S : \#(B) = b\}$ is the set:*

$$\mathcal{Q} = \{Q \subset S : \#(Q) = \lceil \frac{n+b+1}{2} \rceil\}$$

The smaller intersection property of dissemination quorum systems is sufficient because of the existence of authentication protocols: if every variable image written to a server is verifiable by means of, say, a digital signature, then even a Byzantine server cannot convincingly forge data; it must serve either correct or out-of-date data if it responds at all. Thus a read operation can accept the highest-timestamped response it receives as being both correct and up-to-date.

The read and write protocols originally proposed for dissemination quorum systems are almost identical to those for masking quorum systems, except that a value is vouched for even if it is returned by only a single server. The resulting protocol still provides safe semantics, with the additional property that a read that is concurrent with one or more writes returns either the value of the most recently completed write or one of the values being written; in other words, it has regular semantics (Section 2.2).

In [MR98], the protocol for dissemination quorum systems was upgraded to one with atomic semantics (Section 2.2) by the simple expedient of adding a fourth step to the end of each read operation:

4. Write the accepted image $\langle v, t \rangle$ back to a quorum of servers.

This final step ensures that a read that follows another read does not return an earlier value, even if both are concurrent with the same set of write operations.

Unfortunately, the type of authentication needed for dissemination quorum systems requires every client to be able to authenticate every other client, e.g., by means of public keys. Such a scheme may be impractical for systems with large numbers of clients, as it would require every client to keep track of a long and perhaps frequently changing list of keys. In the next chapter we study the problem of atomic semantics for non-authenticated masking quorum systems.

Chapter 3

On Atomic Semantics for Masking Quorum Systems

3.1 Introduction

As indicated in the previous chapter, an important limitation of masking quorum systems as originally proposed is the lack of an atomic read and write protocol for such systems. Although Malkhi and Reiter have shown how this problem can be solved for dissemination quorum systems by adding a simple writeback command to the end of the read protocol [Ph98], the implementation of atomic reads and writes for masking quorum systems has remained an open problem.

As the primary result of this chapter, we generalize the atomicity result of [Ph98] by proving that the writeback mechanism in fact reduces the problem of atomic variable semantics for masking quorum system variables to the simpler problem of regular semantics. Our result provides a "recipe" for atomic semantics in masking quorum systems without such cryptographic authentication tools; any solution to the problem of regular semantics for such systems can be upgraded to an atomic solution by simply adding a writeback step to the end of the read protocol. In fact, the correctness of the atomic protocol of [Ph98] can be viewed as a corollary of our result, as the cryptographic framework of dissemination quorum systems (sans writeback) enforces regular semantics.

In proving this result, we introduce the application of the *calculational* proof style [DS90, GS95, Rao95] to quorum systems. Although such proofs are often more difficult to read than prose proofs, they have two distinct advantages. First, they are readily verifiable, as the steps are neatly laid out and justified. Secondly, they are accessible to the general computer scientist who may not have the area-specific knowledge necessary to follow intuitive leaps.

As a follow-up to the result described, we present a read/write protocol that provides what we call *pseudo-regular* semantics (regular semantics with a possibility of aborted operations) for masking quorum systems without resorting to authentication protocols. We then show that this protocol can be combined with the writeback mechanism to provide *pseudo-atomic* semantics (analogously defined) for such systems.

Finally, we prove the slightly startling fact that there is a very simple set of characteristics, shared by the non-dissemination protocols we have studied thus far, that makes regularity *incompatible* with liveness in an asynchronous environment without concurrency control. Some form of pseudo-atomic semantics are thus the most that can be implemented by such protocols in such an environment.

3.1.1 Related work

The concepts of safe, regular and atomic were presented in [Lam86] as semantic properties of hardware registers. In that paper, Lamport also gave several algorithms for constructing stronger registers from weaker ones, including one for constructing an atomic single-reader register from regular registers (which could in turn be constructed from safe registers). The problem of constructing a multi-reader atomic register was left open, but has since been solved by various researchers including [KKV87, SAG94].

These algorithms are designed in the specific context of hardware registers, and are unsuitable for large-scale distributed data services in two ways. First, they require an a priori limit on the number of users that may access a particular data element at any given time. Second, they are polynomial in this number, leading to undesirably expensive operations.

Fortunately, although these limitations are inescapable in the context of hardware registers (the solution presented in [SAG94] is asymptotically optimal), we are able to avoid them in our application by means of the convenient abstraction of unbounded timestamps. Using this mechanism, we are able to simply strengthen the read/write protocols for quorum system variables so as to provide pseudo-atomicity directly.

3.2 Definitions: TS-Variables and Data Semantics

3.2.1 Formalizing masking quorum system variables: TS-variables

In order to reason formally about Byzantine quorum system variables as a class, we need an abstraction that defines the important features of such variables indepen-

dently of operational details. To this end, in this section we introduce the concept of *TS-variables*. We begin by defining the more general concept of "timestamped variables" as well as a number of useful functions on such variables:

Definition 9 *A* timestamped variable *is a variable of any type whose value is read and updated in conjunction with an associated* timestamp*, where timestamps are drawn from some unbounded totally ordered set* $\mathcal{T}$*.*

Let RW be a set of read and write operations on some timestamped variable with a given read/write protocol and a domain set D; let $R \subset RW$ be the set of reads, and let $W \subset RW$ be the set of writes. Then the following function definitions hold ($\mathbf{R}$ and $\mathbf{B}$ represent the set of reals and the set of booleans, respectively):

$value : RW \rightarrow D$: For $op \in RW$, if op is a read, then $value(op)$ is the value returned by the read; if op is a write, then $value(op)$ is the value written.

$ts : RW \rightarrow \mathcal{T}$: For $op \in RW$, if op is a read, then $ts(op)$ is the timestamp of the value returned by the read; if op is a write, then $ts(op)$ is the timestamp assigned to the value written.

readsfrom: $R \times W \rightarrow \mathbf{B}$: for $r \in R$, $w \in W$, $readsfrom(r, w) \equiv true$ iff r reads the result of write w. For timestamped variables, we define this to be equivalent to:

$$readsfrom(r, w) \equiv value(r) = value(w) \wedge ts(r) = ts(w)$$

For the purposes of the next two functions, we postulate a global "clock" (e.g., the age of the universe in milliseconds) that provides an absolute timescale for system events. As the systems we discuss are asynchronous, individual processes do not have access to global clock values or to these functions, which are used only for reasoning purposes.

$start : RW \rightarrow \mathbf{R}$: The start time of the operation in global time.

$end : RW \rightarrow \mathbf{R}$: The end time of the operation in global time.

The purpose of these functions is to give us a convenient shorthand for reasoning about the *possibility* of concurrency between operations as defined in Chapter 2, without being specific about the actual (nondeterministic, in an asynchronous environment) order in which servers process requests. Essentially, if

$$end(op1) < start(op2) \vee end(op2) < start(op1)$$

then $op2$ is not concurrent with $op1$, whereas if

$$start(op2) \leq end(op1) \wedge start(op1) \leq end(op2)$$

such concurrency may exist and thus needs to be resolved in any proposed serialization of $op1$ and $op2$. For simplicity, we will therefore treat the latter expression as our definition of concurrency hereafter.

In keeping with their hypothetical meaning, we stipulate that the *start* and *end* functions meet the following restriction:

$$\forall op \in RW : start(op) < end(op)$$

TS-variables

Ignoring issues of type and address, we consider a variable to be defined by the specification of the operations that may be performed on it, including at least *read* and *write*.[1] We refer to such a specification as a *variable protocol*. Read and write activity on a variable is described in terms of a *run* of its protocol:

Definition 10 *A* run *of a variable V is a set of operations on V, all of which meet the specification of V's protocol. We call a run RW* complete *if, for all read operations $r \in RW$, there exists a write operation $w \in RW$ such that $readsfrom(r, w)$.*

In this chapter we will continue to use the label RW to represent a variable run; subscripts will be used to distinguish between runs when the context is not otherwise clear. The projection of a run RW onto its read operations will be denoted R; the corresponding projection onto write operations will be denoted W.

Although some researchers use the terms "run" and "execution" interchangeably, in this work we find it useful to follow the example of [Lam86], which gives them distinct technical meanings. Specifically, an execution associates a run with a precedence relation on the operations of that run, i.e.:

Definition 11 *An* execution *of a variable v is a pair $\langle RW, \longrightarrow \rangle$, where RW is a run of v and $\longrightarrow$ is a precedence relation (irreflexive partial order) on the operations in RW.*

We now define two specific types of execution that are of special importance to this work:

[1] We do not concern ourselves with read-only variables in the context of this work.

Definition 12 *An execution* $\langle RW, \longrightarrow \rangle$ *is said to be* real-time consistent *if*

$$\forall op1, op2 \in RW : end(op1) < start(op2) \Rightarrow op1 \longrightarrow op2$$

Definition 13 *An execution* $\langle RW, \longrightarrow \rangle$ *is said to be* write-ordered *if it satisfies the following:*

1. $\forall w_i, w_j \in W : w_i \neq w_j : w_i \longrightarrow w_j \vee w_j \longrightarrow w_i$
2. $\langle W, \longrightarrow \rangle$ *is real-time consistent.*

In other words, (1) in a write-ordered execution, the write operations are totally ordered by $\longrightarrow$, and (2) the order is consistent with the partial order of write operations in real time.

Definition 14 *For all runs RW of a timestamped variable v, the relation* $\xrightarrow{ts}$ *is defined by:*

1. $\forall op \in RW, \forall w \in W : op \xrightarrow{ts} w \equiv ts(op) < ts(w)$
2. $\forall w \in W, \forall r \in R : w \xrightarrow{ts} r \equiv ts(w) \leq ts(r)$
3. $\forall r_a, r_b \in R : r_a \xrightarrow{ts} r_b \equiv ts(r_a) < ts(r_b)$

It is easy to see that $\xrightarrow{ts}$ is irreflexive, antisymmetric and transitive. It is therefore an irreflexive partial order. (Note that operations with identical timestamps are not necessarily ordered by $\xrightarrow{ts}$.)

We now define *TS-variables* as follows:

Definition 15 *A* TS-variable *is a timestamped variable v such that, for all complete runs RW of v,* $\langle RW, \xrightarrow{ts} \rangle$ *is write-ordered.*

Note that Definitions 14 and 15 imply that writes are uniquely identified by timestamp; thus for any given read, there is at most one write with the same timestamp. We can therefore make the following observation, which provides a simplified form of the definition of *readsfrom*() for TS-variables:

Observation 1 *For any read operation r and write operation w of a complete TS-variable run,*

$$readsfrom\,(r, w) \equiv ts(r) = ts(w)$$

3.2.2 Formalizing data semantics for TS-variables

We now define what it means for a write-ordered execution to be *safe*, *regular* or *atomic*. The definitions of safe and regular are based on the following concept, adapted from [Lam86]:[2]

Definition 16 *For a write-ordered execution* $\langle RW, \longrightarrow \rangle$*, let* $w_0, w_1, \ldots$ *be the ordered list of write operations from* RW *as defined by* $\longrightarrow$*. Furthermore, for a given read operation* r*, let* i *be the index of the last write that precedes* r*, i.e.,* $i = max\{k : end(w_k) < start(r)\}$*. Then we say that* r sees $W' \subseteq W$*, where:*

$$W' = \{w_i\} \cup \{w_k : start(r) \leq end(w_k) \wedge start(w_k) \leq end(r)\}$$

We express this relationship in predicate form as $sees(r, W', \longrightarrow)$*.*

The definition of the predicate *sees* is based on the idea that once a write to a variable has completed, previous values of that variable should not be read. Thus the values that a read sees are those that might be legitimately returned by that read, i.e., the value of the most recently completed write w_i and the values of any concurrent writes. The fact that $\langle RW, \longrightarrow \rangle$ is write-ordered implies that all writes seen by r fall within a well-defined range:

Observation 2 *For a given read* r*, let* i *be defined as in Definition 16, and let* $j = max\{k : start(w_k) < end(r)\}$*. Then:*[3]

$$sees(r, W', \longrightarrow) \Rightarrow \langle \forall w_k \in W' : i \leq k \leq j \rangle$$

We now define a *safe execution* as follows, continuing to use w_i to denote the i^{th} write in the order defined by $\longrightarrow$:

Definition 17 *An execution* $\langle RW, \longrightarrow \rangle$ *is* safe *if:*

- *it is write-ordered, and*
- $sees(r, \{w_i\}, \longrightarrow) \Rightarrow readsfrom(r, w_i)$

[2] [Lam86] defined this concept for a single-writer register, whose write operations are thus necessarily serial. We relax this requirement, defining our version of "sees" in terms of *serializable*, rather than serial, writes. Thus our definition can be applied to variables with multiple writers.

[3] Note that the reverse is not true. It is possible for a write to fall within the given range without being seen if the "invisible" write occurs after read r, but concurrently with w_j.

In other words, an execution is safe if any read that sees only one write returns the value of that write. In operational terms, a read that is concurrent with no writes returns the result of the "most recent" write according to the serialization defined by the write-ordering.

A *regular execution* is defined as follows:

Definition 18 *An execution* $\langle RW, \longrightarrow \rangle$ *is* regular *if:*

- *it is write-ordered, and*
- $\forall r \in R : sees(r, W', \longrightarrow) \Rightarrow \langle \exists w : w \in W' : readsfrom(r, w) \rangle$

In other words, a write-ordered execution is regular if every read returns some value that it sees. Note that a regular execution is necessarily safe.

For TS-variables, this definition has a useful consequence:

Lemma 1 *Let* $\langle RW, \xrightarrow{ts} \rangle$ *be regular. Then*

$$\forall r \in R, \forall w \in W : end(w) < start(r) \Rightarrow ts(w) \leq ts(r)$$

The proof of this lemma consists of showing that any arbitrary write that precedes a given read has a timestamp less than or equal to that of the read. Let r be an arbitrary read, let w_i be the i^{th} write in the total order imposed by $\xrightarrow{ts}$ for some arbitrary i, let W' be the set of writes such that $sees(r, W', \xrightarrow{ts})$, and let w_j be the write such that $readsfrom(r, w_j)$. Then:

$$\begin{aligned}
& end(w_i) < start(r) \\
\Rightarrow\ & \{\text{definition of } max, \text{ write-ordering of TS-variables}\} \\
& i \leq max\{k : end(w_k) < start(r)\} \\
\equiv\ & \{\text{Definition 13, Observation 2}\} \\
& i \leq min\{k : w_k \in W'\} \\
\Rightarrow\ & \{j \in W', \text{ Observation 2}\} \\
& i \leq j \\
\Rightarrow\ & \{\text{write-ordering of TS-variables}\} \\
& ts(w_i) \leq ts(w_j) \\
\Rightarrow\ & \{\text{definition of } ts() \text{ for reads}\} \\
& ts(w_i) \leq ts(r)
\end{aligned}$$

Finally, we define an *atomic execution* as an execution that behaves as though the operations were totally ordered in a real-time consistent way, that is:

Definition 19 *An execution* $\langle RW, \longrightarrow \rangle$ *is* atomic *if:*

- $\longrightarrow$ *is a total order on* RW*,*
- $\forall r \in R$*,* $readsfrom(r, w_i) \Rightarrow i = max\{k : w_k \longrightarrow r\}$*, and*
- $\langle RW, \longrightarrow \rangle$ *is real-time consistent.*

Note that the second and third bullets of the definition above imply that any atomic execution is also regular, while the reverse is not necessarily true.

We now define what it means for a *variable* to be safe, regular or atomic.

Definition 20 *A variable protocol is* safe (regular, atomic) with respect to a precedence relation $\longrightarrow$ *if, for all complete runs* RW *of the protocol, the execution* $\langle RW, \longrightarrow \rangle$ *is safe (regular, atomic). A protocol is safe (regular, atomic) if it is safe (regular, atomic) with respect to some precedence relation. A variable is safe (regular, atomic) if its protocol is safe (regular, atomic).*

3.3 Reducing the Atomic Semantics Problem

In this section we show how to construct an atomic TS-variable v_{atom} given a regular TS-variable v_{reg}. We accomplish this by means of the following steps:

1. Add a new operation to the protocol for v_{reg}, specify the operations of v_{atom} in terms of this expanded regular protocol, and show that the resulting v_{atom} is a TS-variable.
2. Define a total order $\xrightarrow{ts'}$ on operations of v_{atom} that extends $\xrightarrow{ts}$, i.e.,
$$op_a \xrightarrow{ts} op_b \Rightarrow op_a \xrightarrow{ts'} op_b$$
3. Use Definition 20 to prove that v_{atom} is atomic with respect to $\xrightarrow{ts'}$.

3.3.1 Defining the atomic protocol

Let v_{reg} be a regular TS-variable. We expand the protocol of v_{reg} by defining a third operation in addition to read and write: *writeback*. The writeback operation is similar to the write operation of v_{reg} except that whereas write operations calculate their own timestamps, a writeback takes its timestamp as an argument; thus writebacks are not necessarily ordered by $\xrightarrow{ts}$. We stipulate, however, that all runs RW_{exp} of the expanded protocol continue to satisfy Lemma 1, as well as the following additional property:

Property 1 *For all read operations r, write operations w and writeback operations b in RW_{exp},*

- $end(b) < start(r) \Rightarrow ts(b) \leq ts(r)$
- $end(b) < start(w) \Rightarrow ts(b) < ts(w)$

(In masking quorum systems, as in dissemination quorum systems, both Lemma 1 and Property 1 are implemented by having a write/writeback perform a null operation at any server whose current timestamp for the variable is higher than that of the write/writeback; thus monotonicity of timestamps is enforced at each server.)

We now define our proposed atomic variable protocol v_{atom} as follows, where $read_{reg}$ and $write_{reg}$ are the read and write protocols of v_{reg}, and val, ts are the value and timestamp respectively of the $read_{reg}$ operation:

$$\begin{aligned} \textbf{Write}_{atom}&: write_{reg} \\ \textbf{Read}_{atom}&: read_{reg};\ writeback(val, ts) \end{aligned}$$

In other words, a write operation of v_{atom} consists of a single write operation of v_{reg}, while a read operation of v_{atom} consists of a read operation of v_{reg} followed by a writeback of the resulting value and timestamp. The timestamp of each $\textbf{Read}_{atom}$ or $\textbf{Write}_{atom}$ operation is identical to the timestamp of the underlying $read_{reg}$ or $write_{reg}$ operation. Because each write operation of v_{atom} consists exactly of one write operation of v_{reg}, it follows that v_{atom} is also a TS-variable. (For clarity, we will hereafter follow the convention that operations of v_{atom} are represented in boldface, while operations of v_{reg} are represented in italics.)

3.3.2 A total order over operations on v_{atom}

In preparation for proving v_{atom} atomic, we specify a precedence relation that totally orders all runs RW_{atom} of v_{atom}. The $\xrightarrow{ts}$ relation that we have already defined is not sufficient, as it does not order read operations that share the same timestamp. We therefore propose to define an extension $\xrightarrow{ts'}$ of $\xrightarrow{ts}$ using the following additional function of type $\mathcal{O}$, where $\mathcal{O}$ is some totally ordered set:

gtf:[4] $RW \rightarrow \mathcal{O}$: An arbitrary function with the following three properties:

- *Sequentiality*: $\forall op_a, op_b \in RW : end(op_a) < start(op_b) \Rightarrow gtf(op_a) < gtf(op_b)$.
- *Uniqueness*: $\forall op_a, op_b \in RW : gtf(op_a) = gtf(op_b) \equiv op_a = op_b$.
- *Read Promotion*: $\forall r \in R, w \in W : start(w) \leq end(r) \equiv gtf(w) < gtf(r)$

[4] "gtf" stands for "global time function".

(An example of such a function is a mapping from $op \in RW$ to the pair $(time(op), id)$, where id is a unique, real-valued operation identifier whose first bit is 1 for a read or 0 for a write, and $time(op) = end(op)$ for $op \in R$ and $time(op) = start(op)$ for $op \in W$.)

The purpose of function gtf is to act as a supplement to timestamps when we define a serialization of the operations. Sequentiality ensures that the order imposed by gtf is compatible with the partial order of the operations in real-time, Uniqueness ensures that the function can act as a "tie-breaker" for operations with the same timestamp, and Read Promotion ensures that any given read operation has a higher gtf than any write that might affect it.[5]

We now define $\xrightarrow{ts'}$ as follows:

Definition 21 *For any given run RW_{atom} of v_{atom},*
$\forall op_a, op_b \in RW_{atom}$:

$$op_a \xrightarrow{ts'} op_b \equiv ts(op_a) < ts(op_b) \vee (ts(op_a) = ts(op_b) \wedge gtf(op_a) < gtf(op_b))$$

In other words, $\xrightarrow{ts'}$ is the lexicographic ordering on the pair $(ts(op), gtf(op))$. It is therefore a total order by virtue of the Uniqueness property of gtf and the fact that ts and gtf have totally ordered codomains.

As a consequence of this definition, we have the following lemma and corollary, which allow us to use Definition 19 to prove atomicity:

Lemma 2 $\forall op_a, op_b \in RW_{atom} : op_a \xrightarrow{ts} op_b \Rightarrow op_a \xrightarrow{ts'} op_b$.

Proof: If op_a is a read or op_b is a write, the property follows immediately from Definitions 14 and 21. Otherwise (op_a is a write and op_b is a read) it follows from these two definitions and the Read Promotion property of gtf.

Corollary 1 *All executions $\langle RW_{atom}, \xrightarrow{ts'} \rangle$ of v_{atom} are write-ordered.*

Proof: Write operations of v_{atom} have unique timestamps by virtue of the fact that v_{reg} is a TS-variable. Thus, by Definition 21 and Lemma 2, we have:

$$\forall \mathbf{w_i}, \mathbf{w_j} \in W_{atom} : \mathbf{w_i} \xrightarrow{ts'} \mathbf{w_j} \equiv \mathbf{w_i} \xrightarrow{ts} \mathbf{w_j}$$

Therefore, since $\langle RW_{atom}, \xrightarrow{ts} \rangle$ is write-ordered (again by virtue of the fact that v_{atom} is a TS-variable), it follows that $\langle RW_{atom}, \xrightarrow{ts'} \rangle$ is also write-ordered.

[5] In fact, these properties are sufficient to allow us to define a total order strictly in terms of gtf. However, gtf alone does not specify the behavior of timestamps, and so does not allow us to reason directly about the behavior of reads via the *readsfrom* function. We will therefore use gtf as indicated above.

3.3.3 Proving v_{atom} atomic

Our remaining goal is to prove that $\langle RW_{atom}, \xrightarrow{ts'} \rangle$ is atomic for all runs RW_{atom} of v_{atom}, thus proving that v_{atom} is an atomic variable:

Theorem 1 *For all runs RW_{atom} of v_{atom}, the execution $\langle RW_{atom}, \xrightarrow{ts'} \rangle$ is atomic.*

As we have already shown that $\xrightarrow{ts'}$ write-orders RW_{atom}, our remaining obligations are to prove:

- $\forall r \in R_{atom}$, $readsfrom(r, w_i) \Rightarrow i = max\{k : w_k \xrightarrow{ts'} r\}$,[6] and
- $\langle RW, \xrightarrow{ts'} \rangle$ is real-time consistent.

Proof that $\forall r \in R_{atom}$, $readsfrom(r, w_i) \Rightarrow i = max\{k : w_k \xrightarrow{ts'} r\}$:

$$
\begin{aligned}
& readsfrom(r, w_i) \\
\equiv\ & \{\text{Observation 1}\} \\
& ts(w_i) = ts(r) \\
\equiv\ & \{\text{write-ordering}\} \\
& ts(w_i) = ts(r) \wedge \forall\{k : k < i : w_k \xrightarrow{ts'} w_i\} \\
\equiv\ & \{\text{definition of } \xrightarrow{ts'}\} \\
& ts(w_i) = ts(r) \wedge \forall\{k : k < i : ts(w_k) \leq ts(w_i)\} \\
\equiv\ & \{\text{Definition 13}\} \\
& ts(w_i) = ts(r) \wedge \forall\{k : k < i : ts(w_k) < ts(w_i)\} \\
\equiv\ & \{\text{definition of } max, \text{ definition of } \leq\} \\
& i = max\{k : ts(w_k) \leq ts(r)\} \\
\Rightarrow\ & \{\text{Lemma 2 , definition of } \xrightarrow{ts'}\} \\
& i = max\{k : w_k \xrightarrow{ts'} r\}
\end{aligned}
$$

□

Proof that $\langle RW, \xrightarrow{ts'} \rangle$ is real-time consistent: Our obligation is to prove that:

$$\forall op_a, op_b \in RW_{atom} : end(op_a) < start(op_b) \Rightarrow op_a \xrightarrow{ts'} op_b$$

We prove this separately for each of the four possible cases: two writes, a write followed by a read, a read followed by a write, and two reads. For simplicity, we will use the convention that **r** and **w** (with possible subscripts) refer to operations

[6] According to the convention we adopted earlier, R_{atom} is the set of read operations from RW_{atom}.

of RW_{atom}, while r, w, and b denote the corresponding read, write and writeback operations of the expanded regular protocol:

Case 1: two writes

$$
\begin{aligned}
& end(\mathbf{w_i}) < start(\mathbf{w_j}) \\
\equiv\ & \{\text{definition of } v_{atom}\} \\
& end(w_i) < start(w_j) \\
\Rightarrow\ & \{\text{write-ordering of TS-variables}\} \\
& w_i \xrightarrow{ts} w_j \\
\equiv\ & \{\text{definition of } v_{atom}\} \\
& \mathbf{w_i} \xrightarrow{ts} \mathbf{w_j} \\
\Rightarrow\ & \{\text{Lemma 2}\} \\
& \mathbf{w_i} \xrightarrow{ts'} \mathbf{w_j}
\end{aligned}
$$

Case 2: write, then read

$$
\begin{aligned}
& end(\mathbf{w}) < start(\mathbf{r}) \\
\equiv\ & \{\text{definition of } v_{atom}\} \\
& end(w) < start(r) \\
\Rightarrow\ & \{\text{Lemma 1}\} \\
& ts(w) \leq ts(r) \\
\equiv\ & \{\text{definition of } \xrightarrow{ts}\} \\
& w \xrightarrow{ts} r \\
\equiv\ & \{\text{definition of } v_{atom}\} \\
& \mathbf{w} \xrightarrow{ts} \mathbf{r} \\
\Rightarrow\ & \{\text{Lemma 2}\} \\
& \mathbf{w} \xrightarrow{ts'} \mathbf{r}
\end{aligned}
$$

Case 3: read, then write

$$
\begin{aligned}
& end(\mathbf{r}) < start(\mathbf{w}) \\
\equiv\ & \{\text{definition of } v_{atom}\} \\
& end(b) < start(w) \\
\Rightarrow\ & \{\text{Property 1 of } \textit{writeback}\} \\
& ts(b) < ts(w) \\
\Rightarrow\ & \{\text{definition of } v_{atom}\} \\
& ts(r) < ts(w) \\
\equiv\ & \{\text{definition of } \xrightarrow{ts}\} \\
& r \xrightarrow{ts} w
\end{aligned}
$$

$\equiv$ {definition of v_{atom}}
$\quad \mathbf{r} \xrightarrow{ts} \mathbf{w}$
$\Rightarrow$ {Lemma 2}
$\quad \mathbf{r} \xrightarrow{ts'} \mathbf{w}$

Case 4: two reads

$\quad end(\mathbf{r}_a) < start(\mathbf{r}_b)$
$\equiv$ {Sequentiality property of gtf}
$\quad end(\mathbf{r}_a) < start(\mathbf{r}_b) \wedge gtf(\mathbf{r}_a) < gtf(\mathbf{r}_b)$
$\equiv$ {definition of v_{atom}}
$\quad end(b_a) < start(r_b) \wedge gtf(\mathbf{r}_a) < gtf(\mathbf{r}_b)$
$\Rightarrow$ {Property 1 of *writeback*}
$\quad ts(b_a) \leq ts(r_b) \wedge gtf(\mathbf{r}_a) < gtf(\mathbf{r}_b)$
$\equiv$ {definition of v_{atom}}
$\quad ts(r_a) \leq ts(r_b) \wedge gtf(\mathbf{r}_a) < gtf(\mathbf{r}_b)$
$\equiv$ {definition of v_{atom}}
$\quad ts(\mathbf{r}_a) \leq ts(\mathbf{r}_b) \wedge gtf(\mathbf{r}_a) < gtf(\mathbf{r}_b)$
$\equiv$ {definition of $ts'()$}
$\quad \mathbf{r}_a \xrightarrow{ts'} \mathbf{r}_b$

This completes the proof of the theorem. □

Thus we have reduced the open problem of atomic semantics for TS-variables, including those implemented by masking quorum systems, to that of regular semantics. Although the problem of regular semantics for masking quorum systems also remains open, it is less difficult in that all atomic protocols are also regular (a direct implication of Definition 19). In the next section we present a non-cryptographic regular protocol for systems that can tolerate occasional aborted reads.

3.4 Implementing Pseudo-Regular Semantics

We improve the semantics of masking quorum systems as defined in [MR98] (see Section 2.2.1) to pseudo-regular by enhancing the read protocol so that read operations abort when no seen value is vouched for. (Note that our protocol need not – and does not – *always* abort when a read is concurrent with a write.)

Read: For a client to read the current value of a variable V, it queries each server in some quorum Q to obtain the set A of images of V, i.e., $A = \{v_u, t_u\}_{u \in Q}$. Of the

images that are vouched for, it selects the image $\langle v, t \rangle$ with the highest timestamp t. If there is no such image, or if $\langle v, t \rangle$ is *countermanded* as defined below, it sets v to $\perp$ (i.e., it aborts).

Definition 22 *A variable image $\langle v, t \rangle$ is* countermanded *for a given read if all members of some voucher set return images with timestamps greater than t. A variable value v is countermanded if all the vouched-for images in which it appears are countermanded.*

This protocol provides pseudo-regular semantics by virtue of the fact that any value that is vouched for but not seen is countermanded. The full statement and proof of the theorem follow:

Theorem 2 *For all runs RW consisting of non-aborted operations of the above protocol, $\langle RW, \xrightarrow{ts} \rangle$ is regular.*

Proof:[7] Let $W' \subset W$ be the set such that $sees(r, W', \xrightarrow{ts})$ is true for read r. Because r did not abort, there exists some w_x such that $readsfrom(r, w_x)$. As it is not possible for an image to be vouched for before the write that writes it begins, we have $start(w_x) \leq end(r)$.

Now, let $i = max\{k : end(w_k) < start(r)\}$. By the safeness of the protocol and the monotonicity of image timestamps, the values of all w_k such that $k < i$ are countermanded in all reads that follow w_i, therefore $x \geq i$, i.e.:

$$x = max\{k : end(w_k) < start(r)\} \vee end(w_x) \geq start(r)$$

By Boolean algebra, this and the result of our first paragraph imply:

$$x = max\{k : end(w_k) < start(r)\} \vee (start(w_x) \leq end(r) \wedge end(w_x) \geq start(r))$$

Therefore, by Definition 16, $w_x \in W'$. The monotonicity of timestamps ensures write-ordering, so Definition 18 is satisfied.□

Corollary 2 *There exists a pseudo-atomic protocol for masking quorum systems.*

Proof: The protocol is constructed from the pseudo-regular one as described in Section 3.3, except that read operations aborted in the regular protocol are also aborted in the atomic one. □

[7] Because this proof is based on inspection of the protocol rather than on formal definitions and lemmata, we do not employ the calculational proof style in this case.

3.5 On the Necessity of Aborted Operations

In the previous section, we have shown how to implement pseudo-atomic semantics in an asynchronous system in which little or no concurrency control has been established. Specifically:

- Any client may send a write request to a quorum of servers at any time, using its choice of timestamp; i.e., writes are always enabled.
- No additional ordering or scheduling is imposed on read and write requests.
- Read and write requests are processed by servers in the order received, where "processing" a write request with a sufficiently high timestamp changes the state of the variable image at the server.

We describe such a system in the ensuing arguments as a *nonrestricted system.*

All the quorum system protocols that we have studied here, including the pseudo-atomic protocol, share the following characteristics:

1. Each server maintains a single version of the variable image at any given time.
2. A read returns a non-$\perp$ value only if some appropriately defined voucher set of servers responds to its query with identical images.

For the remainder of this discussion, we refer to such a protocol as a *classic quorum protocol.* We then define:

Definition 23 *A* classic b-masking protocol *is a classic quorum protocol whose voucher set is defined as any set of* $b+1$ *or more servers.*

In this section, we show that there is no way to implement a classic b-masking protocol that is fully regular[8] in a nonrestricted system. We do this by showing that certain possible server responses to a read query in such a system are unresolvable; i.e., the corresponding operation must either return a possibly faulty value, abort, or retry the query (leaving open the possiblity of non-termination).

We then show that the same is true even if each server maintains a bounded history of the variable images it has received.

[8]We use the term "fully regular" in this section to emphasize the distinction from pseudo-regular semantics: in a (fully) regular protocol, every read returns some seen value; therefore it terminates successfully rather than aborting or failing to terminate.

3.5.1 Definitions

We begin with a number of useful definitions. Let $\mathcal{P}$ be a classic quorum protocol, and let r and w be operations under $\mathcal{P}$ such that r is a read operation and w is the most recently completed (as determined by timestamp) write operation as of the beginning of r. Let Q_r and Q_w be the quorums on which r and w respectively are performed.

Definition 24 *The* intersection set *for r is the set $Q_r \cap Q_w$.*

Let $F \subset Q_r$ be the set of servers that return faulty responses during read r.

Definition 25 *The* informed set *for r is the set $Q_r \cap Q_w \setminus F$.*

Note that if there are no writes concurrent with r, so that no servers in the intersection set have been overwritten since w, then the informed set for r is the voucher set for r. In any case, all servers in the informed set return the results of writes that r *sees* (Definition 16), and in the worst case these are the only servers that do so. We can therefore observe:

Observation 3 *Protocol $\mathcal{P}$ is fully regular iff all possible sets of responses to a read by informed sets contain identical responses from at least one voucher set.*

3.5.2 Nonliveness of classic b-masking protocols

Let $\mathcal{Q}$ be a quorum system with classic quorum protocol $\mathcal{P}$ defined for a fault threshold b, and let *minint* be the size of the smallest intersection set for $\mathcal{Q}$. Then the following lemma is a straightforward consequence of the definitions above:

Lemma 3 *The smallest possible informed set for a read operation on $\mathcal{Q}$ is $minint - b$.*

In such a system, the smallest informed set represents the worst-case scenario for a successful read. Let v be the minimum voucher set size for $\mathcal{P}$.[9] Suppose for a moment that any read operation under $\mathcal{P}$ is concurrent with at most k write operations. Then:

Theorem 3 *$\mathcal{P}$ is fully regular for quorum system $\mathcal{Q}$ iff*

$$\lceil (minint - b)/(k+1) \rceil \geq v$$

[9] For example, $v = b + 1$ for b-masking quorum systems, and $v = 1$ for b-dissemination quorum systems.

Proof: For an arbitrary read operation r let $\mathcal{I}_r = \{I_0, \ldots, I_k\}$ be the partitioning of the informed set such that I_0 contains the servers that return the result of the most recently completed write operation and each I_i contains the servers that return the result of the i^{th} write that is concurrent with r. In a nonrestricted system, any or all of the sets I_i may be nonempty, depending on the order in which concurrent operation requests are received at individual servers. We prove the "if" and "only if" portions of the theorem separately.

If: If $\lceil (minint - b)/(k+1) \rceil \geq v$, then for any read r, some $I_i \in \mathcal{I}_r$ contains a voucher set by the Extended Pigeonhole Principle, which states that at least one member of a partition contains at least the average number of elements for the partition.

Only if: Suppose $\lceil (minint - b)/(k+1) \rceil < v$. Let r be a read operation with the smallest possible informed set, and suppose that r is concurrent with exactly k writes. Furthermore, let $\mathcal{I}_r$ be an even partition, i.e., a partition in which every set contains either the ceiling or the floor of the average number of elements. $\mathcal{I}_r$ does not contain a voucher set, so r is unable to return a seen value. Since this scenario is unpreventable in a system such as that described above, the protocol is not live.□

Lemma 4 *$\mathcal{P}$ is fully regular in a nonrestricted system[10] iff its minimum voucher set size v is* 1.

Thus, if a quorum system read protocol requires agreement between multiple servers in order to determine a correct result then it is not fully regular in an unrestricted system. Since classic b-masking protocols have this requirement by definition, we have:

Corollary 3 *No classic b-masking protocol is fully regular in a nonrestricted system.*

It is worth noting that benign quorum systems and dissemination quorum systems, which do not require agreement between multiple servers, are already known to be fully regular for their appropriate failure models (benign, Byzantine-limited-by-authenticated-data respectively).

3.5.3 Non-liveness of classic protocols with bounded history

We define a *classic b-masking protocol with bounded history* as a b-masking protocol with the following characteristics:

[10] By definition, such a system allows arbitrary values of k

1. Each server maintains a bounded history of its images for a given variable, i.e., a list of the last m images received.

2. A read returns a non-$\perp$ value only if it receives identical images from at least $b+1$ servers, for a specified $b > 0$.

Even if each server responds to every query with its entire history of m images, it remains possible for a read query to be unresolvable in a nonrestricted system, i.e.:

Lemma 5 *No classic b-masking protocol with bounded history is fully regular in a nonrestricted system.*

Proof: In a nonrestricted system, any given read operation may be concurrent with an unbounded number of writes. Suppose some read operation r is concurrent with $m{\cdot}s$ write operations, where m is the size of the bounded history and s is the size of the informed set for r. For $1 \leq i \leq s$, suppose server S_i receives the first $m{\cdot}i$ write requests before receiving the request for r. Then the history of S_1 will contain the images of the first m writes, the history of S_2 will contain the images of the next m writes (which displace the first m because the history is bounded), and so forth. In response to its query, r therefore receives $m{\cdot}s$ *different* variable images, each from exactly one server. It is therefore unable to resolve the query.□

3.5.4 Generalizing to non-size-based systems

In systems where the failure pattern is not defined by size, minimal voucher sets may not have a single well-defined size either. For example, for a masking quorum system on server set S such that the full set $\mathcal{B}$ is required to define the set of possible failure configurations, the set $\mathcal{V}$ of minimal voucher sets is defined by:

$$\mathcal{V} = \{B \cup \{s\} : B \in \mathcal{B}, s \in S \setminus B\}$$

We define a generalization on classic b-masking protocols, which we will call *classic masking protocols* as follows:

1. Each server maintains a single version of the variable image at any given time.

2. A read can only return a non-$\perp$ value if all servers in some member of $\mathcal{V}$ respond to its query with identical images.

In this more general case, members of $\mathcal{V}$ may vary in size. A slightly looser version of Lemma 4 can be therefore formulated for the more general case.

Lemma 6 *A classic masking protocol for a quorum system $\mathcal{Q}$ with failure pattern $\mathcal{B}$ is fully regular in a nonrestricted system iff every possible informed set under $\mathcal{Q}$, $\mathcal{B}$ contains some singleton voucher set.*

In most practical systems, this loosening is not likely to be particularly helpful, as it is not clear how to take advantage of it without postulating a trusted subset of servers. However, we include it for the sake of completeness.

3.6 Conclusion

In this chapter we have presented a reduction that allows us to promote a regular Byzantine quorum system protocol to an atomic one. This reduction has the considerable advantage of being based on an entire class of protocols, i.e., those described by our definition of TS-variables, rather than on any specific algorithm. Thus our result can be used both as a recipe for improving the semantics of masking quorum systems and as a formal correctness proof of the atomic protocol for dissemination quorum systems [Ph98]. We have also shown how to construct a pseudo-atomic protocol for masking quorum systems using this reduction. Such a protocol provides the *safety* properties[11] of an atomic protocol, though it does not guarantee *liveness*. While such a protocol is probably not suitable for applications where variables are frequently overwritten, as it may render them often unreadable, it should be of practical value for systems with few writes and a tolerance for occasionally retrying reads. Finally, we have proven that there is a general approach behind these protocols that makes them inherently non-live in an uncontrolledly concurrent asynchronous environment.

[11] Not to be confused with the "safeness" used in Section 3.4

Chapter 4

Dynamic Quorum Adjustment

4.1 Introduction

In this chapter, we present a method of dynamically raising and lowering the fault tolerance limit of a Uniform masking quorum system in response to estimates of the number of server failures. (For some initial failure detection methods, see Chapter 5.) The goal of this work is to design protocols that allow a quorum system to respond *without blocking* to the presence or absence of detected faults. This flexibility comes at a cost: tolerating a given maximum number of faults requires more servers in our approach than in a static system. However, with a fixed number of servers, our protocols allow a system to operate in low-threshold mode with smaller quorums than a static approach would require for the same worst-case threshold. A natural way of using a dynamic quorum system is to increase the threshold when faults are detected, and decrease it again when the failures have been dealt with. The threshold could also be raised or lowered based on external evidence that the threat of an attack has increased or decreased, such as information in server logs or new information about the value of the data being stored.

The problem of dynamically adjusting a Uniform masking quorum system is not trivial. The primary difficulty can be illustrated by the following example:

Example: Consider a system consisting of $n = 9$ replicated servers with quorums consisting of all sets of 6 servers. This configuration ensures that every pair of quorums intersects in 3 servers or more, and can tolerate a threshold $b = 1$ of Byzantine server failures while still guaranteeing that the majority of every quorum intersection is correct. Now, suppose that some client, detecting a possible failure in the system, wishes to reconfigure the quorum system to raise the resilience threshold to $b = 2$. This can be accomplished by making every set of 7 servers a quorum,