# A Framework for Semantic Reasoning about Byzantine Quorum Systems

Evelyn Pierce* Lorenzo Alvisi†

March 1, 2001

**Abstract**

We present a set of definitions and theorems that allow us to reason about the semantics of quorum system variables, including Byzantine quorum system variables, *as a class*. Using these tools, we present a formal proof that the problem of atomic semantics for such variables can be reduced to the simpler problem of regular semantics for such systems. Specifically, any regular masking quorum system protocol can be combined with a writeback mechanism to produce an atomic protocol. We then describe a subclass of TS-variables for which the latter problem is not solvable by traditional approaches in an asynchronous environment. Finally, for such variables we define the notion of pseudoregular and pseudoatomic semantics, and show briefly that the same reduction holds for these concepts.

**keywords: atomic variable semantics, byzantine fault tolerance, quorum systems, large-scale data services, calculational proofs**

Contact Author: Evelyn Pierce (tumlin@cs.utexas.edu, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712 FAX: (512)471-8885)

*Department of Computer Sciences, University of Texas, Austin, Texas; tumlin@cs.utexas.edu.
†Department of Computer Sciences, University of Texas, Austin, Texas; lorenzo@cs.utexas.edu.

# 1 Introduction

*Byzantine quorum systems* [MR98a] are a promising approach to the problem of efficiently implementing Byzantine fault-tolerant data services. There are several variations on this approach [Baz97, MRWr97, MRW97, MR98a], but the basic concept is the same for all of them: data are maintained simultaneously at multiple sites, and each read or write operation is processed at a subset (called a *quorum*) of those sites. Quorums are defined in such a way that the intersection of any two quorums contains enough servers to allow a query to determine and return accurate and up-to-date information even in the presence of a limited set of arbitrarily faulty servers. Furthermore, because only a subset of the servers is concerned with any given operation, such a system can also remain available in spite of limited server crashes or network partitions. Finally, the fact that the service is designed to tolerate out-of-date servers (e.g., those which were not part of the most recent write quorum) greatly simplifies the task of recovering from failures; as long as a quorum of servers is up to date, others may be brought back online without any need to recover their most recent state.

Analyzing the semantics of shared variables implemented by these quorum systems can be quite challenging. Heretofore, such analysis has been limited to individual protocols; there has been no framework for reasoning about the semantics of quorum variables as a family. For example, while there exist compelling arguments to the effect that fully serializable operations have been achieved for some types of quorum systems (notably the dissemination quorum systems of [MR98b]) and remain an open problem for others (e.g., masking quorum systems, [MR98a]), these arguments do not tell us why these discrepancies exist, or the degree to which individual solutions can be generalized.

One of the primary contributions of this paper is to address this need. We present a set of definitions and theorems that allow us to reason about the *class* of shared variables implemented by quorum systems, including the various Byzantine quorum systems; we call such variables *TS-variables* because of the important role of timestamps in their protocols.[1] Further, we give an adapted version of Lamport's formal definitions of the concepts of *safe*, *regular*, and *atomic* semantics [Lam86]. These concepts have traditionally been used to describe the semantics of Byzantine quorum systems, but their use has necessarily had to be somewhat informal, as Lamport's formal definitions and theorems were based on the assumption that variable writes were never concurrent with one another. Our adaptation is not dependent on this assumption, and so can be applied directly to the variables of interest in a fully calculational proof style. As far as we know, this is the first paper to apply calculational proofs to quorum system variables.

We use these formalisms to prove that the atomicity result of [MR98b] generalizes to an important theorem about TS-variables: the writeback mechanism used in that particular protocol in fact reduces the problem of atomic variable semantics *for any TS-variable* to the simpler problem of *regular semantics*. The correctness of the atomic protocol of [MR98b] can in fact be viewed as a corollary of this result, as the cryptographic framework of dissemination quorum systems (sans writeback) enforces regular semantics.

As a follow-up, we show why the problem of atomic semantics (fully serializable operations) has been straightforwardly solved for some types of quorum system while remaining unsolved for others. Specifically, we show that for a significant subclass of TS-variables, traditional approaches to protocol design will *always* have some danger of failed read queries (aborted, retried or incorrect) in an asynchronous environment. (In fact, the masking quorum systems of [MR98a], for which atomic semantics have proved stubbornly elusive, fall into this category.) Finally, we propose and briefly discuss the somewhat weaker notions of *pseudoregular* and *pseudoatomic* semantics for such systems.

The structure of this paper is as follows. In Section 2, we define TS-variables and a number of related concepts and theorems, including our adapted version of Lamport's semantic categories. In Section 3 we use

[1] In fact, our definition of TS-variables is not specific to quorum system variables; it simply captures those properties that are common to such variables and are relevant to our analysis. Our theorems therefore also hold for any other variable types that may share these properties.

these formalisms to give a fully calculational proof that any regular read/write protocol that satisfies the definition of a TS-variable protocol can be used to implement a corresponding atomic read/write protocol. In Section 4 we show that for an important class of possible protocols, traditional approaches to protocol design always result in a danger of unresolvable queries in an asynchronous system; we then define the weaker notions of pseudoregular and pseudoatomic semantics, which can be implemented in spite of such queries. We conclude in Section 5. (An example of a pseudoregular protocol for masking quorum systems is included in the appendix.)

# 2 Preliminaries

## 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 independently 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 1** *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.*

Let $RW$ be a set of read and write operations on some timestamped variable with a given read/write protocol; 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 (**R** and **B** represent the set of reals and the set of booleans, respectively):

*value*: 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*: 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*: 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 real-valued 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*: The start time of the operation in global time.

*end*: 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 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.[2]

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

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

### 2.1.1 TS-variables

A variable consists of a type, a memory address, and a specification of the operations that may be performed on it, including at least *read* and *write*.[3] 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 2** *A* run *of a variable $v$ is a set of operations performed 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 $readsfrom(r, w)$.*

It is useful to have a separate term for the run consisting of *all* operations performed on a variable during its lifetime:

**Definition 3** *The* history *of a variable is the run consisting of all operations performed on that variable during its lifetime.*

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 4** *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:

**Definition 5** *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 6** *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.*

[2] A more literal definition would be that two operations are concurrent if and only if there exist two servers that process them in different order. However, it will be readily seen that if $end(op1) < start(op2)$, then every server processes $op1$ before $op2$; hence they are not concurrent.

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

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 7** *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 8** *A* TS-variable *is a timestamped variable v such that, for all histories RW of v,* $\langle RW, \xrightarrow{ts} \rangle$ *is write-ordered.*

Note that Definitions 7 and 8 imply that TS-variable 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,*

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

## 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 idea that once a write to a variable has completed, previous values of that variable should not be read. This concept is expressed in [Lam86] in terms of the set of writes that a given read "sees":[4]

**Definition 9** *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* $sees(r, W', \longrightarrow)$*.*

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 above definition is difficult to use directly. Fortunately, the fact that $\langle RW, \longrightarrow \rangle$ is write-ordered implies that all writes seen by $r$ fall within a well-defined range – no such write is earlier than the last write (in terms of the write order) to precede $r$ or later than the last write that is concurrent with $r$:

[4] [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.

**Observation 2** *For a given read $r$, let $i$ be defined as in Definition 9, and let $j = max\{k : start(w_k) < end(r)\}$. Then:*[5]

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

We are now ready to give our definitions of *safe*, *regular*, and *atomic* executions.

### 2.2.1 Safe executions

In informal terms, an execution is safe if any read that sees only one write returns the value of that write. Operationally, this means that 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. Formally, continuing to use $w_i$ to denote the $i^{th}$ write in the order defined by $\longrightarrow$ we say:

**Definition 10** *An execution $\langle RW, \longrightarrow \rangle$ is* safe *if:*

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

### 2.2.2 Regular executions

A write-ordered execution is *regular* if every read returns some value that it sees, i.e., the result of the most recently completed write or a concurrent one. Formally:

**Definition 11** *An execution $\langle RW, \longrightarrow \rangle$ is* regular *if:*

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

Note that a regular execution is necessarily safe.

For TS-variables, this definition has a useful consequence: the timestamp of any given read is at least the timestamp of the most recently completed write. Formally:

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

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

A calculational proof of this lemma is given in Figure 1; it consists of showing that any arbitrary write that precedes a given read has a timestamp less than or equal to that of the read.

[5] 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 $w_j$.