A Recipe for Atomic Semantics for Byzantine Quorum Systems

Evelyn Pierce* Lorenzo Alvisi†

April 18, 2000

Abstract

We present a formal proof that the problem of *atomic semantics* [Lam86] for Byzantine quorum systems with trustworthy clients can be reduced to that of regular semantics. Specifically, any *regular* protocol for such systems can be combined with a writeback mechanism to produce an atomic protocol. We then give a pseudo-regular protocol for general *masking quorum systems* [MR98a] and show that it can be used in this way to build a pseudo-atomic protocol.

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

*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: 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 Byzantinely 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 Finally, the fact that the service is designed to tolerate out-of-date servers (e.g., those which were not part of the last write quorum) greatly simplifies the task of recovering from failures.

An important limitation of Byzantine quorum systems to date has been the lack of general atomic read and write protocols. For example, the read and write operations defined for *masking quorum systems* [MR98a] provide only *safe semantics* [Lam86]. These semantics do not guarantee correct system behavior in the event of concurrent reads and writes, and are therefore of limited use in a distributed system. There do exist algorithms for building atomic data elements from safe ones [KKV87, SAG94], but these are designed for hardware registers with a limited number of bits. They 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. Secondly, they are polynomial in this number, leading to an undesirable overhead in expensive operations.

A better alternative in the context of quorum systems is to strengthen the read/write protocols themselves so as to provide atomicity directly. In [MR98b], Malkhi and Reiter use a writeback mechanism to solve this problem for a specialized type of quorum system called a *dissemination quorum system*, which incorporates authentication protocols such as those provided by a public key cryptosystem. However, the implementation of atomic reads and writes for other types of Byzantine quorum systems, particularly general masking quorum systems, is an open problem.

As the primary result of this paper, we generalize the atomicity result of [MR98b] by proving that the writeback mechanism in fact reduces the problem of atomic variable semantics in masking quorum systems to that of regular semantics. In doing so, we provide a general "recipe" for atomic semantics in these systems that does not depend on the presence of authentication protocols.

As a follow-up to our main result, we present a read/write protocol that provides pseudo-regular semantics (regular semantics with a possibility of aborted operations) for masking quorum systems in general without resorting to authentication protocols. (A preliminary version of this result, for a specialized type of quorum system variable, appears in [AMPRW00].) We then show that this protocol can be combined with the writeback mechanism to provide pseudo-atomic semantics for such systems.

Because proofs of variable semantics in a distributed environment are apt to be delicate and error-prone, we move away from the "intuitive" proof style that has been traditional in this area. Instead, we make extensive use of *calculational proofs* [GS95, Rao95]. 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 accessable to the general computer scientist who may not have the area-specific knowledge necessary to follow intuitive leaps. As far as we know, this is the first paper to apply calculational proofs to quorum systems.

The structure of this paper is as follows. In Section 2, we define a generalization on Byzantine quorum system variables, which we call *BQ-variables*. This section also includes a number of useful definitions and a short review of the formalisms introduced in [Lam86] for reasoning about register semantics. In Section 3 we expand on these formalisms to give a formal proof that an atomic read/write protocol for a BQ-variable can be implemented using a regular read/write protocol. In Section 4 we show how this result can be applied to masking quorum systems by giving a protocol that provides pseudo-regular semantics for such systems. We conclude in Section 5.

2 Preliminaries

In this section we present a number of concepts and definitions that will be used throughout the paper.

2.1 BQ-Variables

Our goal of reducing the problem of atomic semantics for quorum system variables to that of regular semantics requires us to reason generally about a multitude of possible read/write protocols. To this end, we define a class of timestamped variables called *BQ-variables*. This class includes, but is not restricted to, masking quorum system and dissemination quorum system variables.

We begin by introducing a number of functions that will be helpful in both defining and proving properties of these variables. 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; we will call such a set a *variable run*. 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 ($\mathcal{R}$ and $\mathcal{B}$ represent the set of reals and the set of booleans, respectively):

$value : RW \to D$: 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.

$ts : RW \to \mathcal{R}$: 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 $value(op)$ is the timestamp assigned to the value written.

$readsfrom : R \times W \to \mathcal{B}$: For $r \in R, w \in W, readsfrom(r, w) = true$ iff the value returned by r was written by w.

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

$end : RW \to \mathcal{R}$: A function representing the end time of the operation in global time. The only restriction on this function and the previous one is that, for all $op \in RW$, $start(op) < end(op)$.

Using these functions, we now define *BQ-variables* as follows:

Definition 1 *A variable is known as a* BQ-variable *if the following properties hold for all possible runs RW on that variable (with R and W defined as above):*

- *(initialization):* $\exists w \in W\ \forall op \in RW : ts(w) \leq ts(op) \wedge start(w) \leq start(op)$
- *(read/write pairing):* $\forall r \in R, w \in W : readsfrom(r, w) \equiv ts(r) = ts(w)$
- *(read validity)* $\forall r \in R, \exists w \in W : readsfrom(r, w)$
- *(timestamp monotonicity of writes):* $\forall w_a, w_b \in RW : end(w_a) < start(w_b) \Rightarrow ts(w_a) < ts(w_b)$

2.2 Formalisms for Register Semantics

As part of our purpose is to reason formally about the properties defined in [Lam86] in the context of masking quorum systems, we begin by adopting some of the notation used in that paper. Specifically, we consider triples of the form $\langle rw, \longrightarrow, \dashrightarrow \rangle$, where rw is a set of individual operations, and $\longrightarrow$ and $\dashrightarrow$ are a pair of precedence relations on rw, roughly translatable as "precedes" and "can affect". Such a triple is called a *system execution* [Lam86].[1]

For reasoning about variable or register semantics, certain properties of a system execution $\langle S, \longrightarrow, \dashrightarrow \rangle$ must be defined. In [Lam86], these semantic properties are specified for a subset of the set S of operations in the execution; this subset consists of all read/write operations on a particular register and the precedence relations between those operations. As we are concerned *only* with the semantics of individual variables, we simplify this approach by restricting S to just those operations. In other words, we describe our variables using triples of the form $\langle rw, \longrightarrow, \dashrightarrow \rangle$, where rw is the set of all read and write operations on a particular variable over the course of its lifetime, and $\longrightarrow$ and $\dashrightarrow$ are precedence relations on $rw \times rw$. In doing so, we are essentially regarding variables as abstract registers with unlimited storage (for unrestricted timestamping). We will refer to system executions of this type as *variable executions.*

We now present the first three semantics properties. Properties 2 and 3 are paraphrased from Axioms B1 and B2 in [Lam86]. Property 1, however, is a generalization of Axiom B0 from the same source, as the original does not accommodate the possibility of concurrent writes.

Property 1 *There exists a relation $\xrightarrow{\prime}$ such that the set of write operations in rw is totally ordered by $\xrightarrow{\prime}$ and*

$$\forall w_i, w_j \in W, w_i \longrightarrow w_j \Rightarrow w_i \xrightarrow{\prime} w_j$$

Furthermore, the first write in the order imposed by $\xrightarrow{\prime}$, denoted w_0, has the property that $\forall r \in R, w_0 \longrightarrow r$.

Because of the initialization and timestamp monotonicity properties of BQ-variables, it will usually be convenient to use $w_i \xrightarrow{\prime} w_j \equiv ts(w_i) < ts(w_j)$ in cases where $\longrightarrow$ indicates real-time precedence.

Property 2 *$\forall r \in R$ and $w \in W$, $r \dashrightarrow w \vee w \dashrightarrow r$.*

Property 3 *Every read r in rw returns some value in D (the domain of the variable).*

(Property 3 is independent of the definitions of $\longrightarrow$ and $\dashrightarrow$, and we will simply stipulate its truth for the variable constructions we are reasoning about. This can be informally justified by defining D as the set of all values that may be returned by a read.)

The remaining three semantic properties, also from [Lam86], are based on the following definition:

Definition 2 *For a given variable execution $\langle rw, \longrightarrow, \dashrightarrow \rangle$, let $w_0, w_1, \ldots$ be the sequence of write operations from rw. Then for a given read operation r, we define the predicate* sees *as follows:*

$$sees(r, i, j) \equiv i = max\{k : r \not\dashrightarrow w_k\} \wedge j = max\{k : w_k \dashrightarrow r\}$$

The remaining semantic properties are then paraphrased as:

[1]The full definition of a system execution requires $\longrightarrow$ and $\dashrightarrow$ to satisfy certain simple axioms [Lam86]. It is straightforward to prove that all the precedence relations in this paper do meet this requirement, but we omit the proofs for reasons of brevity.

Property 4 (safe): $\forall r \in rw,\ sees(r,i,i) \Rightarrow readsfrom(r,w_i)$

Property 5 (regular): $\forall r \in rw,\ sees(r,i,j) \Rightarrow$
$\langle \exists k : i \leq j : readsfrom(r,w_k) \rangle$

Property 6 (atomic): $\forall r \in RW,\ sees(r,i,j) \Rightarrow i = j$

2.2.1 Using the Properties

In order to clarify our objectives, we will introduce the following additional terms:

Definition 3 *A pair* $\langle \longrightarrow, \dashrightarrow \rangle$ *of precedence relations is called a* serial witness *for a variable* v *if for all runs* rw *of* v*, the triple* $\langle rw, \longrightarrow, \dashrightarrow \rangle$ *is a variable execution that satisfies Properties 1 - 6.*

Definition 4 *A variable* v *is said to be* abstractly atomic *if there exists a serial witness for it.*

Informally, an abstractly atomic variable is one on which read and write operations behave as though executed in some specific total order (i.e., the order specified by the "serial witness"). This is not sufficient for true atomicity, however, as there is no stated requirement that this order be consistent with the partial order in which the operations are executed in real time. For example, a read operation on an abstractly atomic variable might behave as though it were executed before a write operation when in fact it was executed afterwards in real time. This is, of course, unacceptable for a truly atomic variable.

This problem is addressed in [Lam86] by introducing a formalized notion of *implementation*, as follows:

Definition 5 *Variable execution* $\langle RW, \longrightarrow, \dashrightarrow \rangle$ *is called an* implemention *of variable execution* $\langle RW', \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow} \rangle$ *if:*

- *Every operation in* RW' *consists of a finite set of operations from* S
- *Every operation in* RW *appears in a finite, nonzero number of operations of* RW'
- $\forall A, B \in S, A \xrightarrow{\prime} B \Leftarrow \langle \forall a \in A, \forall b \in B : a \longrightarrow b \rangle$

Let $\xrightarrow{t}$ and $\overset{t}{\dashrightarrow}$ represent the real-time precedence operators, i.e., $op_a \xrightarrow{t} op_b \equiv end(op_a) < start(op_b)$ and $op_a \overset{t}{\dashrightarrow} op_b \equiv op_a \not\xrightarrow{t} op_b$. We will refer to implementations of the form $\langle RW, \xrightarrow{t}, \overset{t}{\dashrightarrow} \rangle$ as *real-time implementations.*

Using this definition, we can define an atomic variable as a variable whose serialized operations have a real-time implementation.

Definition 6 *A variable* v *is* atomic *if it has a serial witness* $\langle \longrightarrow, \dashrightarrow \rangle$ *such that for all runs* rw_a *of* v*,* $\langle rw_a, \longrightarrow, \dashrightarrow \rangle$ *has a real-time implementation.*

We are now ready to prove our reduction result.

3 Reducing the Atomic Semantics Problem

In this section we show how the problem of atomic semantics for BQ-variables can be reduced to that of regular semantics. Our strategy consists of two major steps:

1. Define a pair of precedence relations $\xrightarrow{q}$, $\dashrightarrow^{q}$ such that $\langle \xrightarrow{q}, \dashrightarrow^{q} \rangle$ is a serial witness for any BQ-variable, thus proving that all BQ-variables are abstractly atomic.

2. Given a variable whose temporal executions $\langle rw, \xrightarrow{t}, \dashrightarrow^{t} \rangle$ satisfy Properties 1- 5 show how to construct an implementation of any abstractly atomic variable execution $\langle RW, \xrightarrow{q}, \dashrightarrow^{q} \rangle$. (see Section 2.2)

3.1 Defining $\xrightarrow{q}, \dashrightarrow^{q}$

As our goal is to provide a serial witness for BQ-variables, we need some function that maps operations to a totally ordered set. The timestamp function would be an obvious choice, except for the fact that multiple operations may have the same timestamp (e.g. a write and a subsequent read). We therefore propose the following additional function of type $\mathcal{T}$ for some totally ordered set $\mathcal{T}$:

gtf: $RW \to \mathcal{T}$: An arbitrary function with the following three properties:

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

(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, and $time = end(op)$ for $op \in R$ and $time = start(op)$ for $op \in W$.)

The primary purpose of the *gtf()* function 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.[2]

We now define $\xrightarrow{q}$ and $\dashrightarrow^{q}$ as follows: for any given variable run RW, $\forall op_a, op_b \in RW$:

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

and

$$\forall op_a, op_b \in RW : op_a \dashrightarrow^{q} op_b \equiv op_b \not\xrightarrow{q} op_a$$

In other words, $\xrightarrow{q}$ is the lexicographic ordering on the pair $(ts(op), gtf(op))$, and $\dashrightarrow^{q}$ is the inverse of $\xrightarrow{q}$.

We now prove that the relations $\xrightarrow{q}$ and $\dashrightarrow^{q}$ are identical in the case where $op_a \neq op_b$, a fact that we will use repeatedly in the remainder of the paper.

[2] In fact, these properties are sufficient to allow us to define a total order strictly in terms of *gtf()*. As timestamps are much easier to reason about directly, however, we will use *gtf()* as indicated above.

Lemma 1 *For all $op_a, op_b \in RW$ such that $op_a \neq op_b$,*

$$op_a \xrightarrow{q} op_b \equiv op_a \overset{q}{\dashrightarrow} op_b$$

Proof: The lemma follows from the fact that $\xrightarrow{q}$ is antisymmetric. (For the interested reader, a detailed calculational proof is provided in the Appendix.) □

3.2 Abstract Atomicity of BQ-variables

Having defined relations $\xrightarrow{q}$ and $\overset{q}{\dashrightarrow}$, we are now ready to prove that they are a serial witness for any BQ-variable, and thus that all BQ-variables are abstractly atomic:

Theorem 1 *BQ-variables are abstractly atomic by virtue of serial witness $\langle \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$.*

Proof: We prove each property for variable execution $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$, where RW is a run of a BQ-variable. As we have stipulated the truth of Property 3 for systems of interest, we omit it. We apply the calculational style to three of the property proofs.

Proof that $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$ satisfies Property 1 :

By the *Uniqueness* property of the *gtf()* function, *gtf()* maps every write operation in W to a unique value; therefore $w_a \xrightarrow{q} w_b \vee w_b \xrightarrow{q} w_a$ holds for all $w_a, w_b \in W$. Define the total order on the writes by $w_a < w_b \equiv w_a \xrightarrow{q} w_b$, and let w_0 be the first write in this order. Then for any read operation r, we have $w_0 \xrightarrow{q} r$ by the *initialization* property of BQ-variables and the *Sequentiality* property of the *gtf()* function.□

Proof that $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$ satisfies Property 2 :

$$\begin{aligned}
& r \overset{q}{\dashrightarrow} w \vee w \overset{q}{\dashrightarrow} r \\
\equiv\ & \{\text{Lemma } 1,\ r \neq w\} \\
& r \xrightarrow{q} w \vee w \overset{q}{\dashrightarrow} r \\
\equiv\ & \{\text{Definition of } \overset{q}{\dashrightarrow}\} \\
& r \xrightarrow{q} w \vee r \overset{q}{\not\rightarrow} w \\
\equiv\ & \{A \vee \neg A \equiv true\} \\
& true\ \square
\end{aligned}$$

Proof that $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$ satisfies Property 4 :

$$\begin{aligned}
& sees(r, i, i) \\
\Rightarrow\ & \{\text{Definition of } sees\} \\
& i = max\{k : w_k \overset{q}{\dashrightarrow} r\} \\
\equiv\ & \{\text{Lemma } 1\} \\
& i = max\{k : w_k \xrightarrow{q} r\} \\
\equiv\ & \{\text{Definition of } \xrightarrow{q}\} \\
& i = max\{k : ts(w_k) < ts(r) \vee (ts(w_k) = ts(r) \wedge gtf(w_k) < gtf(r)\} \\
\Rightarrow\ & \{\text{Read validity property of BQ-variables, Lemma 2}\} \\
& ts(w_i) = ts(r) \wedge gtf(w_i) < gtf(r) \\
equiv\ & \{\text{Read promotion property of } gtf()\} \\
& ts(w_i) = ts(r) \wedge start(w) \leq end(r)
\end{aligned}$$

$$
\begin{aligned}
&\equiv \{\text{Timestamp validity property of BQ-variables}\}\\
&\quad ts(w_i) = ts(r)\\
&\equiv \{\text{Read/write pairing property of BQ-variables}\}\\
&\quad readsfrom(r, w_i)\Box
\end{aligned}
$$

Proof that $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow}\rangle$ *satisfies Property* 5 : This property is an immediate consequence of Properties 4 and 6.

Proof that $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow}\rangle$ *satisfies Property* 6 :

$$
\begin{aligned}
&\quad sees(r, i, j)\\
&\equiv \{\text{Definition of } sees\}\\
&\quad i = max\{k : r \not\overset{q}{\dashrightarrow} w_k\} \wedge j = max\{k : w_k \overset{q}{\dashrightarrow} r\}\\
&\equiv \{\text{Lemma } 1\}\\
&\quad i = max\{k : r \not\xrightarrow{q} w_k\} \wedge j = max\{k : w_k \overset{q}{\dashrightarrow} r\}\\
&\equiv \{\text{Definition of } \overset{q}{\dashrightarrow}\}\\
&\quad i = max\{k : w_k \overset{q}{\dashrightarrow} r\} \wedge j = max\{k : w_k \overset{q}{\dashrightarrow} r\}\\
&\equiv \{\text{substitution of equals}\}\\
&\quad i = j\Box
\end{aligned}
$$

This completes the proof that all BQ-variables are abstractly atomic.

In the next section we show how to construct a real-time implementation of such a variable using a regular protocol; this will complete our reduction goal.

3.3 Implementing $\langle RW, \xrightarrow{q}, \overset{q}{\dashrightarrow}\rangle$

We will now show how to implement a truly atomic BQ-variable given a regular variable protocol $\mathcal{RW}_r$; i.e, a protocol such that, for all runs RW_r of $\mathcal{RW}_r$, $\langle RW_r, \xrightarrow{t}, \overset{t}{\dashrightarrow}\rangle$ satisfies Properties 1 - 5.

We begin by defining a third operation in addition to read and write: *writeback*. The writeback operation is identical to the write operation of $\mathcal{RW}_r$ except that it takes its timestamp as an argument; thus the set W_r' of writeback operations in a given run RWW' of the expanded protocol need not have timestamp monotonicity as defined in Section 2. W_r' does, however, have the following property, which is easily shown to hold for W (the set of write operations in RWW') as well:

Property 7 $\forall w' \in W_r'\ \forall r \in R_r : w' \xrightarrow{t} r \Rightarrow ts(w') \leq ts(r)$

We now define our proposed atomic protocol as follows, where $read_r$ and $write_r$ are the read and write protocols of $\mathcal{RW}_r$, and val, ts are the value and timestamp respectively of the $read_r$ operation:

Write: $write_r$
Read: $read_r$, $writeback(val, ts)$

It is not difficult to see that this protocol satisfies the definition of a BQ-variable if $\mathcal{RW}_r$ does.

Let RW_a be a run of this protocol, and let rww' be the corresponding run of the expanded regular protocol. We now prove the following:

Theorem 2 $\langle rww', \xrightarrow{t}, \overset{t}{\dashrightarrow}\rangle$ *implements* $\langle RW_a, \xrightarrow{q}, \overset{q}{\dashrightarrow}\rangle$

Proof: $\langle rww', \xrightarrow{t}, \overset{t}{\dashrightarrow} \rangle$ meets the first two requirements of an implementation of $\langle RW_a, \xrightarrow{q}, \overset{q}{\dashrightarrow} \rangle$ by definition of rww'. Our remaining obligation is to prove that:

$$\forall op_a, op_b \in RW_a, op_a \xrightarrow{t} op_b \Rightarrow op_a \xrightarrow{q} op_b$$

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.

Case 1: two writes

$$
\begin{aligned}
& W_i \xrightarrow{t} W_j \\
\Rightarrow\ & \{\text{timestamp monotonicity}\} \\
& ts(W_i) < ts(W_j) \\
\Rightarrow\ & \{\text{definition of } \xrightarrow{q}\} \\
& W_i \xrightarrow{q} W_i
\end{aligned}
$$

Case 2: write, then read

$$
\begin{aligned}
& W_i \xrightarrow{t} R \\
\Rightarrow\ & \{\text{definition of } max\} \\
& i \leq max\{k : W_k \xrightarrow{t} R\} \\
\Rightarrow\ & \{\ 1, \text{ set } maxk = max\{k : W_k \xrightarrow{t} R\} \\
& ts(W_i) \leq ts(W_{maxk}) \\
\Rightarrow\ & \{\text{Property } 5, \text{ Definition } 2, \text{ read/write pairing}\} \\
& ts(W_i) \leq ts(R) \\
\Rightarrow\ & \{\text{definition of } \xrightarrow{q}, \text{ read promotion property of } \mathit{gtf()}\} \\
& W_i \xrightarrow{q} R
\end{aligned}
$$

Case 3: read, then write

$$
\begin{aligned}
& R \xrightarrow{t} W_i \\
\equiv\ & \{\text{definition of } \overset{t}{\dashrightarrow}\} \\
& W_i \not\xrightarrow{t} R \\
\Rightarrow\ & \{\text{definition of } \overset{t}{\dashrightarrow}, \text{ definition of } max\} \\
& i > max\{k : W_k \overset{t}{\dashrightarrow} R\} \\
\Rightarrow\ & \{\text{Property 1, set } maxk = max\{k : W_k \overset{t}{\dashrightarrow} R\} \\
& ts(W_i) > ts(W_{maxk}) \\
\Rightarrow\ & \{\text{Property 5, Definition 2, read/write pairing}\} \\
& ts(W_i) > ts(R) \\
\Rightarrow\ & \{\text{definition of } \xrightarrow{q}\} \\
& W_i \xrightarrow{q} R
\end{aligned}
$$

Case 4: two reads

$$
\begin{aligned}
& R_a \xrightarrow{t} R_b \\
\equiv\ & \text{Sequentiality property of } \mathit{gtf()} \\
& R_a \xrightarrow{t} R_b \wedge \mathit{gtf}(R_a) < \mathit{gtf}(R_b) \\
\Rightarrow\ & \{\text{definition of } \mathcal{RW}_a,\ 7\}
\end{aligned}
$$

$$
\begin{aligned}
& ts(R_a) \leq ts(R_b) \wedge gtf(R_a) < gtf(R_b) \\
\Rightarrow & \{\text{definition of } \xrightarrow{q}\} \\
& R_a \xrightarrow{q} R_b
\end{aligned}
$$

This completes the proof of the theorem. $\Box$

Thus we have proved that our proposed protocol has a real-time implementation. As it is also abstractly atomic by virtue of being a BQ-variable protocol, it is a truly atomic protocol. Thus the problem of atomic semantics for BQ-variables (and therefore Byzantine quorum systems) is no more difficult than that of regular semantics.

The problem of regular semantics has not been solved for all types of Byzantine quorum system, however. In particular, it has remained an open problem for masking quorum systems in the absence of cryptographic authentication protocols. In the next section we present a non-cryptographic regular protocol for systems that can tolerate occasional aborted reads.

4 Masking Quorum Systems With Pseudo-regular Semantics

4.1 Definitions

Masking quorum systems are defined in [MR98a] for systems consisting of a set U of servers ($\mid U \mid = n$), some of which may be faulty, and an unspecified finite number of clients. Faulty servers may exhibit *Byzantine* behavior, i.e., they may behave in arbitrarily incorrect ways. Clients are assumed to be correct. Clients and servers are connected by means of asynchronous FIFO channels, which are also assumed to be correct.

Masking quorum systems are defined in terms of a *failure pattern* $\mathcal{B} \subseteq 2^U$ such that $\forall B_1, B_2 \in \mathcal{B}, B_1 \not\subset B_2$:

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

- $\forall Q_a, Q_b \in \mathcal{Q} : B_x, B_y \in \mathcal{B}, (Q_a \cap Q_b) \backslash B_x \not\subseteq B_y$
- $\forall B \in \mathcal{B} \ \exists Q \in \mathcal{Q} : B \cap Q = \emptyset$

Members of $\mathcal{Q}$ are called *quorums.* Note that the above definition implies that every pair of quorums in $\mathcal{Q}$ intersect; this is the defining characteristic of quorum systems in general.

In operational terms, if all the faulty servers in the system are contained in some element of $\mathcal{B}$, and all read and write operations are performed on elements of $\mathcal{Q}$, then every read quorum intersects the quorum of the most recently completed write in an identifiably correct set of servers, and at least one quorum is always available even if faulty servers are nonresponsive.

For convenience in the ensuing discussion, we introduce the following definition:

Definition 8 *A set B^+ of servers is called a* voucher set *for failure pattern $\mathcal{B}$ if*

$$\forall B_x, B_y \in \mathcal{B} : B^+ \not\subseteq B_x \cup B_y$$

A value/timestamp pair 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 a pair that is vouched for.*

The intuition is that a voucher set is a set of servers that cannot all be faulty under failure pattern $\mathcal{B}$. This definition is useful in formulating our protocols and proofs below.

4.1.1 Safe protocol for masking quorum systems

Using the above definition, we rephrase the original read/write protocol for masking quorum systems (which has been proven to provide safe semantics [MR98a]) as follows:

Read: For a client to read a variable x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{\langle v_u, t_u \rangle\}_{u \in Q}$. Of the pairs that are returned by all members of some voucher set, the client chooses the pair $\langle v, t \rangle$ with the highest timestamp, and chooses v as the result of the read operation. If there is no such pair, the client returns $\perp$ (a null value).

Write: For a client to write value v to a variable, it queries some quorum Q to determine the current timestamp value at each server, and selects a unique new timestamp t greater than the largest current timestamp and greater than any timestamp it has previously chosen, and sends each server in some quorum Q' the update request $\langle v, t \rangle$. (A server that receives such an update request performs the update only if the timestamp of the request is greater than the current timestamp of the variable. The timestamp of a variable is therefore monotonically nondecreasing at all correct servers.)

It is not difficult to see that a variable accessed by this protocol, if properly initialized before the first read, meets the requirements of a BQ-variable. The reason the protocol provides only safe semantics is that *during* a write, any given quorum may be partially overwritten to such a degree that none of the values from writes w_i to w_j (see previous section) is vouched for. The result of such a read is either $\perp$ or $value(w_k)$ for some $k < i$.

4.2 Enhancing the Read Protocol

We improve the semantics of masking quorum systems to pseudo-regular by enhancing the read protocol so that read operations abort when the situation above is encountered. (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 x, it queries each server in some quorum Q to obtain a set of value/timestamp pairs $A = \{v_u, t_u\}_{u \in Q}$. Of the pairs that are vouched for, it selects the pair $\langle b, t \rangle$ with the highest timestamp t. If there is no such pair, or if $\langle b, t \rangle$ is *countermanded* as defined below, it sets b to $\perp$ (i.e., it aborts).

Definition 9 *A value/timestamp pair $\langle v, t \rangle$ is* countermanded *for a given read if all members of some voucher set return pairs with a timestamp greater than t. A variable value v is countermanded if it appears only in countermanded pairs.*

The fact that this protocol provides pseudo-regular semantics is expressed by the following theorem:

Theorem 3 *For all runs RW consisting of non-aborted operations of the above protocol, $\langle RW, \xrightarrow{t}, \dashrightarrow^{t} \rangle$ satisfies Properties 1 - 5.*

Proof: For read operations that do not overlap write operations, our protocol behaves identically to the original read protocol, which has already been proven safe. Our protocol is therefore also safe (i.e., satisfies Properties 1 - 4). We prove Property 5 below.

Proof of Property 5 (by contradiction): Suppose $sees(r, i, j)$ is true for read r. Because r did not abort, there exists some w_x such that $readsfrom(r, w_x)$. If it is not true that $i \le x \le j$, then by the definition of $sees$ and $\dashrightarrow^{t}$ we have $x < i$. This is not possible, however: the safety of the original protocol requires that the values of all earlier writes be countermanded upon completion of w_i, and the monotonicity of timestamping at servers ensures that they remain countermanded for subsequent writes. Hence $i \le x \le j$, so Property 5 is satisfied. □

Corollary 1 *There exists a pseudo-atomic protocol for general masking quorum systems.*

Proof: The protocol is constructed from the pseudo-regular one as described in Section 3. □

5 Conclusion

In this paper 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 BQ-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 [MR98b]. We have also shown how to construct a pseudo-atomic protocol for masking quorum systems using this reduction.

There are several possible directions for future work. The problem of building a regular read/write protocol *without* aborted reads for general masking quorum systems remains open, as does the problem of stronger-than-safe semantics for Byzantine quorum systems with untrustworthy clients (who might, e.g., deliberately send different values to different servers under the same timestamp).

Acknowledgement: We would like to express our sincere thanks to Dahlia Malkhi and Michael Reiter for numerous thought-provoking discussions.

References

[AMPRW00] L. Alvisi, D. Malkhi, E. Pierce, M. Reiter, and R. Wright. Dynamic Byzantine Quorum Systems. *To appear in Proceedings of the 1st International Conference on Dependable Systems and Networks*, June 2000.

[Baz97] R. A. Bazzi. Synchronous Byzantine quorum systems. In *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 259–266, August 1997.

[GS95] D. Gries and F. B. Schneider *A Logical Approach to Discrete Math*, Springer-Verlag, 1995.

[KKV87] L. M. Kirosis, E. Kranakis, and P. M. B. Vitanyi. Atomic Multireader Register. *Proceedings of the 2nd International Workshop on Distributed Algorithms*, pages 278-296, July 1987.

[Lam86] L. Lamport. On Interprocess Communications. *Distributed Computing* 1:77-101, 1986.

[Mal99] D. Malkhi. Quorum systems. Chapter in *The Encyclopedia of Distributed Computing*, Joseph Urban and Partha Dasgupta, editors, Kluwer Academic Publishers. To be published.

[MR98a] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR98b] D. Malkhi and M. Reiter. Secure and Scalable Replication in Phalanx. *Proceedings of the 17th IEEE Symposium on Reliable Distributed Sysystems*, pages 51-58, October 1998.

[MRW97] D. Malkhi, M. Reiter, and A. Wool. Optimal Byzantine Quorum Systems. DIMACS Technical Report 97-10, April 1997.

[MRWr97] D. Malkhi, M. Reiter, and R. Wright. Probabilistic Quorum Systems *Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, August 1997.

[Rao95] J. R. Rao. *Extensions of the UNITY Methodology*, Springer-Verlag, 1995.

[SAG94] A. K. Singh, J. H. Anderson, and M. G. Gouda. The Elusive Atomic Register. *Journal of the ACM*, 41(2):311–339, 1994.

A Appendix: Calculational Proof of Lemma 1

For $op_a \neq op_b$, we prove that $op_a \xrightarrow{q} op_b \equiv op_a \overset{q}{\dashrightarrow} op_b$:

$op_a \overset{q}{\dashrightarrow} op_b$

$\equiv$ {definition of $\overset{q}{\dashrightarrow}$}

$op_b \not\xrightarrow{q} op_a$

$\equiv$ {definition of $\xrightarrow{q}$}

$\neg(ts(op_b) < ts(op_a) \vee (ts(op_b) = ts(op_a) \wedge gtf(op_b) < gtf(op_a)))$

$\equiv$ {distributivity of $\vee$ over $\wedge$}

$\neg((ts(op_b) < ts(op_a) \vee ts(op_b) = ts(op_a)) \wedge (ts(op_b) < ts(op_a) \vee gtf(op_b) < gtf(op_a)))$

$\equiv$ {definition of $\leq$ }

$\neg((ts(op_b) \leq ts(op_a)) \wedge (ts(op_b) < ts(op_a) \vee gtf(op_b) < gtf(op_a)))$

$\equiv$ {boolean algebra}

$ts(op_a) < ts(op_b) \vee (ts(op_a) leq ts(op_b) \wedge gtf(op_a) leq gtf(op_b))$

$\equiv$ {boolean algebra}

$ts(op_a) < ts(op_b) \vee (ts(op_a) = ts(op_b) \wedge gtf(op_a) \leq gtf(op_b))$

$\equiv$ {Uniqueness property of *gtf()*, $op_a \neq op_b$}

$ts(op_a) < ts(op_b) \vee (ts(op_a) = ts(op_b) \wedge gtf(op_a) < gtf(op_b))$

$\equiv$ {Definition of $\xrightarrow{q}$}

$op_a \xrightarrow{q} op_b \Box$