Multi-Writer Consistency Conditions for Shared Memory Objects *

Cheng Shao   Evelyn Pierce   Jennifer L. Welch

Technical Report 2003-7-1
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112, U.S.A
{cshao, pierce, welch}@cs.tamu.edu

August 20, 2003

Abstract

Regularity is a shared memory consistency condition that has received considerable attention, notably in connection with quorum-based shared memory. Lamport's original definition of regularity assumed a single-writer model, however, and is not well defined when each shared variable may have multiple writers. In this paper, we address this need by formally extending the notion of regularity to a multi-writer model. We give three possible definitions of regularity in the presence of multiple writers. We then present a quorum-based algorithm to implement each of the three definitions and prove them correct. We study the relationships between these definitions and a number of other well-known consistency conditions, and give a partial order describing the relative strengths of these consistency conditions. Finally, we provide a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion under each of our proposed consistency conditions.

1 Introduction

1.1 Overview

Distributed computer systems are ubiquitous today, ranging from multiprocessors to local area networks to wide-area networks such as the Internet. Shared memory, the exchange of information between processes by the reading and writing of shared objects, is an important mechanism for interprocess communications in distributed systems. A consistency condition in a shared memory system is a set of constraints on values returned by data accesses when those accesses may be interleaved or overlapping. A shared memory system with a strong consistency condition may be easy to design protocols for, but may require a high-cost implementation. Conversely, a shared memory system with a weak consistency condition may be easy to implement, but difficult for the user to program or reason about. Finding a consistency condition that can be implemented efficiently and that is nonetheless strong enough to solve practical problems is one of the aims of shared memory research.

The preferred consistency condition for shared memory objects is atomicity (or linearizability) ([12]), in which read and write operations behave as though they were executed sequentially, i.e., with

---

*This work was supported in part by NSF grant 0098305, Texas Higher Education Coordinating Board grant ARP-00512-0091-2001, and Texas Engineering Experiment Station funds.
no interleaving or overlap, in a sequence that is consistent with the relative order of non-overlapping operations. In many cases, however, this semantics is difficult to implement, particularly in distributed systems where variables are replicated and where the number of processes with access to the variable is not known in advance. For such systems, the related but weaker condition of regularity ([12]) may be easier to implement while retaining some usefulness. For this reason, it has received considerable attention in its own right, notably in connection with quorum-based shared memory ([2], [16], [15] and [14]).

Informally speaking, regularity requires that every read operation return either the value written by the latest preceding write (in real time) or that of some write that overlaps the read. This description is sufficiently clear for the single-writer model\(^1\), in which the order of the writes performed on a given variable in any execution is well-defined; in fact, it was for this model that Lamport gave his formal definition of regularity. In a multi-writer model, however, multiple processes may perform overlapping write operations to the same variable so that the “latest preceding write” for a given read may have no obvious definition.

A common way to circumvent this problem is to rely on a plausible generalization of the informal definition above, e.g. the following, which appears in [16]:

- A read operation that is concurrent with no write operations returns a value written by the last preceding write operation in some serialization of all preceding write operations, and

- A read operation that is concurrent with one or more write operations returns either the value written by the last preceding write operation in some serialization of all preceding write operations, or any of the values being written in the concurrent write operations.

Such a definition, however, leaves a good deal of room for interpretation. What is meant by “some serialization” in this context? Is there a single serialization of the writes for which the above is true for all read operations, or does it suffice for there to be some (possibly different) such serialization for each operation? Or should all read operations of the same process perceive writes as occurring in the same order? Such ambiguities can be avoided with a formal definition of multi-writer regularity, but to our knowledge none has yet been proposed.

1.2 Contributions of This Paper

In this paper, we formally extend the notion of regularity to a multi-writer model. Specifically, we give three possible formal definitions of regularity in the presence of multiple writers. We then present a quorum-based algorithm to implement each of these definitions and prove the algorithms correct. The definitions are strictly increasing in strength, while the implementations are of comparable complexity. Although there is no obvious practical advantage to the weaker definitions over the stronger, the first two formalizations serve to point out the ambiguity of the informal notion of regularity, while the third provides stronger guarantees at essentially no additional cost.

We also study the relationships between our definitions of multi-writer regularity and several existing consistency conditions: linearizability ([8]), sequential consistency ([11]), coherence ([7]), PRAM ([13]) and PCG ([1]). As part of this analysis, we give a partial order describing the relative strengths of these consistency conditions.

Finally, we provide a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion when the variables are implemented under our three proposed consistency conditions. The algorithms we examine are Peterson’s algorithm for 2 processes ([19]) and Dijkstra’s algorithm ([20]). We find that Peterson’s algorithm is fully correct under all three conditions.

\(^1\)In the single-writer model, only one process can write to a shared object; other processes can only read from it.
Dijkstra’s algorithm satisfies only some of the constraints of the mutual exclusion problem under any of the conditions.

1.3 Related Work

There is copious literature on consistency conditions for shared memory, both implementations and applications (e.g., [11], [13], [7], [8] and [1]). Our work builds on the notion of regularity as introduced in [12].

We follow the example of [3] and [1] in using the mutual exclusion problem as an application for our consistency conditions. In [3], Attiya and Friedman revised Peterson’s 2-process algorithm ([19]) to solve the mutual exclusion problem under their hybrid consistency model. In [1], Ahamad et al. examined the correctness of Peterson’s algorithm and Lamport’s bakery algorithm under the PCG consistency model, showing that Peterson’s algorithm solves the mutual exclusion problem under PCG, while Lamport’s algorithm fails to do so. In a later study, Higham et al. ([9]) investigated other mutual exclusion algorithms, including Dekker’s and Dijkstra’s, none of which guarantees mutual exclusion under PCG.

2 Preliminaries

Every shared object is assumed to have a *sequential specification* that indicates its desired behavior. A sequential specification is a prefix-closed set of sequences of operations, representing the set of behaviors of the object in the absence of concurrency. We define members of this set as *legal sequences* of operations:

**Definition 1** A sequence of operations on a shared object is a legal sequence if it belongs to the sequential specification of the shared object.

In this paper, we consider only read/write objects. For such objects, a sequence of operations is legal if each read returns the value of the most recent preceding write in the sequence. If there is no such write, it returns the initial value of the object.

A *consistency condition* on a shared memory object specifies a relationship between the sequential specification of the object and the set of executions on the object, where an execution is a sequence of possibly interleaved operation invocations and responses.

We assume a system of $n$ processes labeled $p_0, \ldots, p_{n-1}$. For a given execution $\sigma$, we use the symbol $\sigma|_i$ to denote the subsequence of $\sigma$ containing all the invocations and responses performed by process $p_i$.

**Definition 2** An execution $\sigma$ is admissible if, for each $i$, $0 \leq i \leq n-1$:

- if the number of steps taken by $p_i$ is finite, then the last step by $p_i$ is a response; and

- $\sigma|_i$ consists of alternating invocations and matching responses, beginning with an invocation. Thus, at any given point in the execution, each process has at most one operation pending.

Note that this definition allows arbitrary asynchrony of process steps — no constraints are placed on the relative speed with which operations complete or on the time between operation invocations. However, for convenience in analyzing executions, we follow the example of [12] and [6] in employing the useful abstraction of an imaginary global clock. All our references to “real time” in the sequel are with respect to this imaginary clock, which is not available to the processes themselves.
For the remainder of this paper, we will be concerned only with admissible executions.

Given an execution $\sigma$, we use the symbol $\text{ops}(\sigma)$ to denote the set of all operations whose invocations and responses appear in $\sigma$. (Thus $\text{ops}(\sigma|i)$ denotes the set of all operations that are performed in $\sigma$ by process $p_i$.) Finally, we let $\text{writes}(\sigma)$ denote the set of all write operations in execution $\sigma$.

A permutation on a subset of $\text{ops}(\sigma)$ is $\sigma$-consistent if it preserves the partial order of the operations in $\sigma$.\(^2\) The formal definition is given below.

**Definition 3** Given an execution $\sigma$, a permutation $\pi$ of a subset of $\text{ops}(\sigma)$ is $\sigma$-consistent if, for any operations $o_1$ and $o_2$ in $\pi$, $o_1$ precedes $o_2$ in $\pi$ whenever $o_1$ ends in $\sigma$ before $o_2$ begins.

Note that this definition implies that the per-process order of operations is preserved in a $\sigma$-consistent permutation.

3 Multi-Writer Regular Variables: Three Specifications

In this section we present three possible definitions of a multi-writer regular variable, in increasing order of strength; we name these $\text{MWR1}$, $\text{MWR2}$ and $\text{MWR3}$ respectively. The first two are distinct ways of straightforwardly generalizing Lamport’s regularity for single-writer variables, while the third generalizes a slight strengthening of Lamport’s definition.\(^3\)

**Definition 4 (MWR1)**

An execution $\sigma$ is mw1-regular (or satisfies MWR1) if, for each read operation $r \in \text{ops}(\sigma)$, there exists a permutation $\pi_r$ of $\text{writes}(\sigma) \cup \{r\}$ such that:

- $\pi_r$ is a legal sequence.

- $\pi_r$ is $\sigma$-consistent.

A shared memory object is mw1-regular if all executions on that object are mw1-regular.

Informally, an execution $\sigma$ satisfies MWR1 if each read $r \in \text{ops}(\sigma)$ returns the value of some write $w$ that either overlaps or precedes $r$ in $\sigma$, as long as no other write falls completely between $w$ and $r$. This definition allows different reads to behave as though the set of writes occurred in different orders, as long as all such orderings are consistent with the partial order of the writes in $\sigma$.

Figure 1 shows an execution that satisfies MWR1. (In our figures, $W(x,v)$ denotes a write operation that writes value $v$ to variable $x$, and $R(x,v)$ denotes a read operation on variable $x$ that returns value $v$.) For $p_2$’s first read, $W(x,2)$ is considered to be the latest preceding write, and the read perceives the permutation $W(x,1), W(x,2), R(x,2)$. For the second read, $W(x,1)$ is considered to be the latest preceding write; thus the permutation is $W(x,2), W(x,1), R(x,1)$.

As this example shows, MWR1 is actually a very weak consistency condition, as it does not require write operations to behave as though they occurred in any particular order, even from the point of view of a single process. It might, therefore, be desirable to construct a stronger definition of regularity for the multi-writer case.

One straightforward approach might be to simply require that all read operations perceive the same ordering of the write operations, i.e., to add to MWR1 the requirement that $\pi_r - \{r\}$ is equal for all $r$.

\(^2\)In most situations of interest, $\sigma$ represents the order of operation invocations and responses in real time.

\(^3\)We justify the continued use of the term “regularity” for this last definition because in the multi-writer model it remains weaker than linearizability, the next stronger consistency condition defined in Lamport’s hierarchy.
However, we contend that such a consistency condition is actually too strong for distributed multi-writer systems, where message delays may cause a given write to be “lost” when it is, in essence, overwritten by an overlapping write without being read. Requiring all reads to place this “invisible” write in the same position in their perceived order of writes is unnecessary, and may be difficult.

Consider, for example, the execution in Figure 2. This execution does not satisfy the proposed specification above, but is possible under several variations of the classic quorum-based shared memory protocols described in Section 5.2.

In order to accommodate behavior of this kind, we propose a more sophisticated definition for our second and stronger version of multi-writer regularity, by requiring any pair of reads to agree only on the ordering of writes that are “relevant” to both of them. Toward this end, we use the following additional notation: \( \text{writes}_{\rightarrow r}(\sigma) = \{ w | w \in \text{writes}(\sigma) \text{ and } w \text{ begins before } r \text{ ends in } \sigma \} \).

**Definition 5 (MWR2)** An execution \( \sigma \) is mw2-regular (or satisfies MWR2) if there exists a permutation \( \pi \) of all the operations in \( \text{ops}(\sigma) \) such that, for any read operation \( r \), the projection \( \pi_r \) of \( \pi \) onto \( \text{writes}_{\rightarrow r}(\sigma) \cup \{ r \} \) satisfies:

- \( \pi_r \) is a legal sequence.
- \( \pi_r \) is \( \sigma \)-consistent.\(^4\)

*A shared memory object is mw2-regular if all executions on that object are mw2-regular.*

This definition is similar to that of mw1-regularity, except that for any two reads \( r_1 \) and \( r_2 \), the set of writes that do not strictly follow either \( r_1 \) or \( r_2 \) must be perceived by both reads as occurring in the same order. As before, each read returns the value of an overlapping write or of the last preceding write in the order.

The execution in Figure 2 satisfies MWR2 as shown by the following argument. Let \( \pi = W(x, 2), W(x, 1), R_1(x, 1), R_2(x, 1) \). Then the projections for the two reads are \( W(x, 1), R_1(x, 1) \) and \( W(x, 2), W(x, 1), R_2(x, 1) \) respectively. Because the “lost” write strictly follows \( R_1 \), it does not appear in the set of ordered writes for that read operation; it can thus be regarded without inconsistency as occurring before \( W(x, 1) \) from the point of view of \( R_2 \). It is easy to verify that these two sequences satisfy the two conditions of MWR2. Figure 3 shows another execution that satisfies MWR2. (To see this, consider

\(^4\)Note that if there are only a finite number of reads in a given execution, the writes after the last read are not constrained by MWR2 to appear in any particular order. We consider this to be acceptable, as such writes are never observed.
the permutation \( \pi = W(x, 1), R_2(x, 1), R_3(x, 1), W(x, 2), R_2(x, 2), R_3(x, 2) \). It is easy to see that the projections for all four read operations satisfy the two conditions in Definition 5.

The execution shown in Figure 1 does not satisfy MWR2, as there is no single way to order the writes that is consistent with the values returned by the reads, given the partial order of the operations in real time.

A straightforward extension of MWR2 leads to a still stronger consistency condition.

**Definition 6 (MWR3)** An execution \( \sigma \) is mw3-regular (or satisfies MWR3) if there exists a permutation \( \pi \) of all the operations in \( \text{ops}(\sigma) \) such that, for any read operation \( r \), the projection \( \pi_r \) of \( \pi \) onto \( \text{writes}_{\leftarrow r}(\sigma) \cup \{ r \} \) satisfies:

- \( \pi_r \) is a legal sequence.
- \( \pi_r \) is \( \sigma \)-consistent.
- If \( r_1 \) and \( r_2 \) are two read operations of a given \( p_i \), and \( p_i \) performs \( r_1 \) before \( r_2 \), then the writes that appear before \( r_1 \) in \( \pi_{r_1} \) appear before \( r_2 \) in \( \pi_{r_2} \).

A shared memory object is mw3-regular if all executions on that object are mw3-regular.

MWR3 is similar to MWR2, but places the following additional constraint on the read operations: any two read operations performed by the same process must appear in \( \pi \) in the order in which they occur at that process.

This is equivalent to the requirement that once a process reads from a given write, it never reads from an “earlier” write in the order of writes perceived by that process, i.e., individual processes read from writes in nondecreasing order. In [15], variables with this property are called *monotone variables*.

Although the execution in Figure 3 satisfies MWR2, it does not satisfy MWR3. To see this, note that the definition of MWR3 requires that \( W(x, 1) \) and \( W(x, 2) \) appear in the same order in the permutation for each of the four read operations. Suppose \( W(x, 1) \) appears before \( W(x, 2) \). Then \( \pi_{r_1} \), the permutation for the first read of \( p_2 \), and \( \pi_{r_2} \), the permutation for the second read of \( p_2 \), do not satisfy the last condition of Definition 6. If we reverse the order of the two write operations, however, the same problem occurs with respect to \( p_3 \)’s two read operations. (Note that if the second read of process \( p_2 \) were to return 2 instead of 1, then the execution would satisfy MWR3.)

The following lemma emphasizes the relationship between our definitions and the single-writer definition of Lamport.

**Lemma 1** For a single-writer shared variable, MWR1 and MWR2 are identical and are equivalent to the specification of regularity in [12]. MWR3 is strictly stronger than Lamport’s regularity in the single-writer case, but remains weaker than linearizability. Furthermore, if there is only a single reader, then MWR3 is equivalent to linearizability.

The proof of this lemma is straightforward and is left to the reader.
4 A Partial Order of Consistency Conditions

In this section we examine the relationships between our three proposed definitions of multi-writer
regularity and the following existing consistency models: linearizability ([8]), sequential consistency
([11]), coherence ([7]), PRAM ([13]) and PCG ([1]). Although our three definitions form a strict
hierarchy in terms of strength, they are not comparable to any of the more established conditions
except linearizability and, in the case of MWR3, coherence. For those who are not familiar with the
conditions listed above, we provide formal definitions in the appendix.

We begin by ranking our consistency conditions in terms of strength.

Lemma 2 MWR3 is strictly stronger than MWR2, which is strictly stronger than MWR1.

Proof. The definition of MWR3 is identical to the definition of MWR2 except for the imposition
of an additional constraint. The fact that this constraint strengthens the definition can be seen from
the fact that the execution in Figure 3 satisfies MWR2 but not MWR3. Therefore MWR3 is strictly
stronger than MWR2.

MWR2 can be seen to imply MWR1 because the projection $\pi_r$ for each read operation $r$ under
MWR2 is also a permutation that satisfies the conditions in the definition of MWR1. As we have
already noted, however, the execution in Figure 1 does not satisfy MWR2. Thus MWR2 is strictly
stronger than MWR1.

As we have noted previously, linearizability is strictly stronger than MWR3, and is therefore also
stronger than MWR2 and MWR1. We now establish comparisons with the other consistency conditions
mentioned above.

4.1 Coherence

Both MWR3 and coherence define constraints only on sequences of operations on a single variable.
However, MWR3 requires that each permutation be $\sigma$-consistent, while coherence does not. This
difference suggests that MWR3 is strictly stronger than coherence. This is in fact the case, as we now
show.

Lemma 3 MWR3 is strictly stronger than coherence.

Proof. For an execution $\sigma$, let $\pi$ be a permutation of $\text{ops}(\sigma)$ that validates MWR3. The write
operations in $\pi$ preserve the partial order of writes in $\sigma$. Otherwise, there would be at least one
projection $\pi_r$ that did not satisfy the second condition of MWR3.

The third condition of MWR3 implies that all the read operations from the same process appear
in $\pi$ in the same order as in $\sigma$.

Since $\sigma$-consistency implies per-process order, MWR3 implies coherence. However, the execution
in Figure 4 satisfies coherence but does not satisfy MWR3.

\[
\begin{array}{c}
\text{P0} & \text{W(x,0)} & \text{W(x,1)} & \text{R(x,0)} \\
\text{P1} & & & \\
\end{array}
\]

Figure 4: Execution that is coherent but not MWR3

To see that MWR2 and MWR1 are not comparable with coherence, observe that the execution in
Figure 3 satisfies MWR2 (and therefore MWR1) but is not coherent, while the execution in Figure 4
is coherent without satisfying MWR2 or MWR1.
4.2 Other Consistency Conditions

Sequential consistency, PCG and PRAM, unlike MWR3, place constraints on the relative order of operations on multiple variables. In order to compare these notions, therefore, we extend the definition of MWR3 to multiple-variable systems in Definition 7. We use the symbol \( \sigma[x] \) to denote the subsequence of \( \sigma \) containing all the invocations and responses performed by all the processes on the shared object \( x \).

**Definition 7** A system with multiple shared variables satisfies MWR3 if, for any execution \( \sigma \), the projection \( \sigma[x] \) satisfies MWR3 for each variable \( x \) in the system.

MWR1 and MWR2 are extended analogously. (The resulting extended definitions are similar to that of coherence.)

Even with these multi-variable extensions, however, MWR1, MWR2 and MWR3 do not place constraints on the relative order among operations on different variables, as sequential consistency, PCG and PRAM do. On the other hand, they do require \( \sigma \)-consistent permutations of the operations, which the other three conditions do not. Thus our three definitions of regularity cannot be compared with these consistency conditions in terms of strength, as we now show.

The execution in Figure 4 is sequentially consistent, and thus satisfies PCG and PRAM. It does not, however, satisfy MWR1, MWR2 or MWR3.

Now consider the execution in Figure 5. For variable \( x \), the permutation \( W(x, 2), W(x, 1), R(x, 1), R(x, 1) \) is legal. For variable \( y \), the permutation \( R(y, 0), W(y, 1), W(y, 2), R(y, 2) \) is legal. Thus the execution satisfies MWR3 (and thus MWR2 and MWR1). However, in order to construct a permutation on \( p_1 \)’s operations and all the other processes’ write operations, it is necessary to place \( W(y, 1) \) after \( R(y, 0) \) and place \( W(y, 2) \) before \( R(y, 2) \). However, the resulting sequence is not legal because \( R(x, 1) \) should return 2. Therefore the execution is not PRAM and thus it is not PCG or sequentially consistent.

![Figure 5: Execution that is MWR3 but not PRAM](image)

The relationships between all the consistency conditions discussed above are shown as a partial order in Figure 6.

![Figure 6: Relationships among Existing Consistency Conditions](image)
Code for $p_i, \ 0 \leq i \leq n-1$:

$\text{write}(x, v)$
1. for some quorum $Q$, send($\text{read}, x$) to each $q \in Q$;
2. wait to receive response $(v_q, ts_q)$ from each $q \in Q$;
3. $t := \text{MaxTS}\{ts_q|q \in Q\}$;
4. $t_w = \text{IncTS}(t)$;
5. for some quorum $Q'$, send($\text{write}, x, v, t_w$) to each $q' \in Q'$;
6. wait to receive $\text{ack}$ from each $q' \in Q'$;
7. $\text{ack}(x)$;

$\text{read}(x)$:
1. for some quorum $Q$, send($\text{read}, x$) to each $q \in Q$;
2. wait to receive response $(v_q, ts_q)$ from each $q \in Q$;
3. $t := \text{MaxTS}\{ts_q|q \in Q\}$;
4. $v := \text{GetValue}(t)$;
5. $\text{return}(x, v)$;

Code for server $q$:

local:
$v_q$ /* local copy of shared variable, initially 0 */
$t_q$ /* local copy of timestamp, initially 0 */

When $q$ receives ($\text{read}, x$) from $p_i$:
1. $\text{send}(v_q, t_q)$ to $p_i$;

When $q$ receives ($\text{write}, x, v, t$) from $p_i$:
1. if ($t_q < t$) then
2. $v_q = v; t_q = t$;
3. endif
4. $\text{send}(\text{ack})$ to $p_i$;

Figure 7: A generic quorum-based algorithm to implement a read/write shared object

5 Implementations

In this section, we show how to implement a quorum-based shared memory object satisfying each of the definitions of multi-writer regularity proposed in Section 3. A quorum system $Q$ over a set $S$ of servers is a set of subsets of the server set, i.e., $Q \subseteq 2^S$. Each element of a quorum system is called a quorum. In this paper, we assume that the intersection of each pair of quorums is nonempty, i.e., that the quorum system is strict. When we use a quorum system to implement a shared object, each server maintains a local copy of the shared object along with an associated timestamp.

A generic algorithm that uses quorums to implement a shared object with read/write semantics is given in Figure 7. The main idea of the algorithm is as follows. For any operation, a process begins by querying each member of some quorum for its current view of the value and timestamp of the shared object. It then uses the function $\text{MaxTS}()$ to obtain the largest timestamp from the resulting set of responses. The operations then continue as follows:

- **read**: The process uses a function $\text{GetValue}()$ to find a value associated with the resulting timestamp, and returns that value as the result of the read.

- **write**: The process increments the resulting timestamp using a function $\text{IncTS}()$, and writes the new value and the incremented timestamp back to every member of some quorum (which can, but need not, be the same quorum that was queried).

This algorithm is a generalization of several existing quorum-based protocols; the appropriate instantiations of $\text{MaxTS}(), \text{IncTS}()$ and $\text{GetValue}()$ yield, e.g., the algorithms in [16], [5] and [17].

The three algorithms that we present in this section differ in their implementations of the functions used by the generic algorithm. This allows them to use different types of timestamps and different policies by which to select returned values. We will explicitly point out those differences as we introduce each of the algorithms.

5.1 Algorithm Alg,MW1 for Implementing MWR1

The timestamp used by Alg,MW1 is chosen from the set of natural numbers $\mathcal{N}$. In this case, the $\text{MaxTS}()$ function simply returns the largest timestamp in numerical order, while $\text{IncTS}(ts)$ increments its argument by 1. Since timestamps are not necessarily unique under this algorithm, several
different values may share the same largest timestamp value. In this case, Get\(Value()\) simply chooses one arbitrarily and returns the corresponding value.

We define the timestamp of a write operation as the timestamp the write operation uses to write to the quorum. Similarly, the timestamp of a read operation is the timestamp value associated with the variable value returned by the operation. For both cases, we use the symbol \(ts(op)\) to denote the timestamp of operation \(op\). We say that a read operation \(r\) reads from a write operation \(w\) if the value-timestamp pair returned by \(r\) is equal to the value-timestamp pair of \(w\). If there are several such writes, then we choose some arbitrary write \(w\) among them and say that \(r\) reads from \(w\).

The following lemma states that the timestamp order (numerical order by timestamp) of certain operations extends the partial order of those operations in real time.

**Lemma 4** For any execution \(\sigma\) of Alg\(_{MW1}\), there exist the following relationships between the operations and their timestamps:

- For any read operation \(r\) and any write operation \(w\): if \(w\) precedes \(r\) in \(\sigma\), then \(ts(w) \leq ts(r)\).
- For any two write operations \(w_1\) and \(w_2\): if \(w_1\) precedes \(w_2\) in \(\sigma\), then \(ts(w_1) < ts(w_2)\).

**Proof.** Suppose a read operation \(r\) begins after a write operation \(w\) ends. By the quorum intersection property, the quorums used by \(w\) and \(r\) intersect. Therefore, \(r\) reads the timestamp used by \(w\) if that timestamp has not been overwritten by a concurrent or subsequent write. If \(w\) has been overwritten, then it follows from the algorithm that \(w\)'s timestamp is smaller than that of the "winning" write. Therefore \(r\)'s timestamp is no less than \(w\)'s.

By the quorum intersection property, for any two write operations \(w_1\) and \(w_2\), the quorums used by \(w_1\) and \(w_2\) intersect. Thus the algorithm for a write operation reads the current timestamp and increments it by 1. As a result, the timestamp of the later write is strictly greater than that of the earlier one.

**Theorem 5** Algorithm Alg\(_{MW1}\) implements MWR1.

**Proof.** For a read operation \(r\), we construct \(\pi_r\), as follows. We partition the set of writes into two subsets:

- The set of writes that begin before \(r\) ends and whose timestamps are at most that of \(r\), i.e., \(\{w | w \in writes_{\leq r}(\sigma) \text{ and } ts(w) \leq ts(r)\}\)
- The set of all remaining writes

Each of these two sets is arranged in increasing order of timestamp; writes with identical timestamps are ordered arbitrarily. We insert \(r\) into the first sequence immediately after the write that it reads from and then append the second sequence to the first sequence.

The reader can easily verify that the resulting sequence satisfies the two conditions of MWR1.

**5.2 Algorithm Alg\(_{MW2}\) for Implementing MWR2**

We implement a shared variable satisfying MWR2 by adding the process id (as in [16]) to the timestamps used by the generic algorithm. In other words, we define the timestamp of an operation as a pair \(\langle ts, id \rangle\), where \(ts\) is a natural number, and \(id\) is a unique process id. Since no individual process chooses the same \(ts\) value for two different writes, each write operation is guaranteed a unique timestamp value. This ensures that no matter how many write operations overlap, all read operations that begin after
all these write operations finish are able to agree on which is the “last” write. Note that this is a commonly used approach in the implementation of shared variables using quorum systems.

For timestamps of this format, we define $MaxTS()$ as the function that returns the largest timestamp in lexicographic order on the pair $(ts, id)$. Because this timestamp is unique, $GetValue()$ simply returns the unique value associated with it. Finally, $IncTS()$ increments the $ts$ field by 1 and places the calling process identifier in the $id$ field.

The proof of correctness of Algorithm Alg_MW2 is based on the following supporting lemma:

**Lemma 6** The write operations performed using Algorithm Alg_MW2 are totally ordered by timestamp, and this total order is consistent with the partial order of the write operations in real time.

**Proof.** Because the timestamp includes the process id to break ties, the timestamp order is a total order. Since we still use the timestamp of Alg_MW1 as the first field in the lexicographic order, Lemma 4 implies that the real-time order of the write operations is preserved.

**Theorem 7** Algorithm Alg_MW2 implements MWR2.

**Proof.** We construct the permutation $\pi$ as follows. We begin by ordering the write operations lexicographically as described above. We then insert each read operation $r$ after the write operation that $r$ reads from and before the next write operation in the total order. Read operations with identical timestamps are ordered arbitrarily. Now we prove that for any $r$, the projection $\pi_r$ satisfies the conditions in Definition 5.

The sequence $\pi_r$ is legal by construction, as $r$ appears immediately after the write that it reads from.

Now, consider any two operations $op_1$ and $op_2$ in $\pi_r$ such that $op_1$ finishes before $op_2$ starts in $\sigma$. There are two possible cases:

- $op_1$ and $op_2$ are both write operations. Then according to Lemma 6 and our construction method, their order in $\pi_r$ is $\sigma$-consistent.

- $op_1$ is a write operation and $op_2 = r$. If $r$ reads from $op_1$, then $op_1$ appears immediately before $r$ according to our construction. Otherwise, $r$ reads from a write $w$ whose timestamp is larger than that of $op_1$. Therefore, they appear in $\pi_r$ as $op_1, w, r$, so the order of $op_1$ and $r$ is again $\sigma$-consistent.

There are no other cases, as writes that begin after $r$ completes are not included in $writes_{\pi_r}(\sigma)$, and thus do not appear in $\pi_r$.

5.3 Algorithm Alg_MW3 for Implementing MWR3

Algorithm MW3 is similar to Algorithm Alg_MW2 except that in Algorithm MW3, each process keeps a local copy of the value-timestamp pair of the shared variable $x$. The function $MaxTS()$ and $IncTS()$ are defined in the same way as for Alg_MW2. We change the function $GetValue()$ as follows. If the timestamp returned by $MaxTS()$ is not greater than the timestamp in the local copy, then $GetValue()$ returns the value stored in the local copy. Otherwise, $GetValue()$ updates the local copy with the value associated with the timestamp and returns that value.

**Theorem 8** Algorithm Alg_MW3 implements MWR3.
1. **Peterson’s Algorithm for 2 Processors**

Code for process $P_i$, $i \in \{0, 1\}$:

shared variables:
- $Flag[0..1]$: integer /* initially 0 */
- $Turn$: integer /* initially 0 */

/* entry section */
1 repeat
2 $Flag[i] := 0$;
3 wait until ($Flag[1-i] = 0$ or $Turn = i$);
4 $Flag[i] := 1$;
5 until ($Turn = i$ or $Flag[1-i] = 0$)
6 if ($Turn = i$) then wait until ($Flag[1-i] = 0$);

Critical Section

/* exit section */
7 $Turn := 1 - i$;
8 $Flag[i] := 0$;

Remainder Section

2. **Dijkstra’s Algorithm for n Processors**

Code for process $P_i$, $0 \leq i \leq n - 1$:

shared variables:
- $Flag[0..n-1]$: idle, requesting, in-cs /* Initially, idle */
- $Turn$: integer /* Initially 0 */

/* entry section */
1 repeat
2 $Flag[i] :=$ requesting;
3 while ($Turn \neq i$) do
4 if ($Flag[Turn] = idle$) then $Turn := i$;
5 end while
6 $Flag[i] :=$ in-cs;
7 until ($\forall j \neq i, Flag[j] \neq$ in-cs)

Critical Section

/* exit section */
8 $Flag[i] :=$ idle;

Remainder Section

---

**Proof.** We construct $\pi$ as in the proof of Theorem 7, except that read operations with identical timestamps are ordered consistently with their partial order in $\sigma$. Now we prove that $\pi$ satisfies the conditions in Definition 6.

The first two conditions can be proved using the same arguments as in the proof of Theorem 7.

As for the third condition, consider two read operations $r_1$ and $r_2$ of the same process, where $r_1$ completes before $r_2$ begins. Because $\pi_{r_1}$ and $\pi_{r_2}$ are projected from the same sequence $\pi$, it is sufficient to prove that (1) $r_1$ appears before $r_2$ in $\pi$, and (2) all writes that appear in $\pi_{r_1}$ also appear in $\pi_{r_2}$.

The first claim follows from the fact that, by Alg.MW3, the timestamp of $r_2$ is at least that of $r_1$, so our construction method places them in $\pi$ in the order indicated. The second claim follows from the fact that $writes_{r_1} \subseteq writes_{r_2}$, which is clear by definition of $writes_{r}$ (see Section 3). Thus all writes that appear in $\pi_{r_1}$ also appear in $\pi_{r_2}$.

---

6 Mutual Exclusion Using Regular Shared Variables

In this section, we use the mutual exclusion problem as a practical context to evaluate the strength of our three specifications on multi-writer regular shared variables. Specifically, we study the correctness of two well-known algorithms for mutual exclusion when the variables are implemented according to the three consistency conditions we have proposed. The algorithms we examine are Peterson’s algorithm for 2 processes ([19]) and Dijkstra’s algorithm for $n$ processes ([20]). The algorithms are shown in Figure 8.

Algorithms for solving mutual exclusion are assumed to have four sections: *entry*, *critical*, *exit* and *remainder*. The *critical* section is code that must be protected from concurrent execution. The *entry* section is the code executed in preparation for entering the critical section. The *exit* section is executed to release the critical section. The rest of the code is in the *remainder* section.

An execution of a program (not to be confused with an execution on an object) is defined as an

---

5 Although Lamport’s Bakery algorithm ([10]) and Peterson-Fischer’s algorithm ([18]) are often studied in this context, they are not of interest to us here since these algorithms use only single-writer shared variables.
interleaving of local operations and shared-memory operation invocations and responses performed by the participating processes, such that the following are satisfied:

- the projection of the program execution onto (the actions performed by) each individual process is consistent with the order of operations imposed by the algorithm, and

- the projection of the program execution onto the shared-memory operations on each variable is an admissible execution on that variable.

(In this context, we consider a shared-memory object “response” to be the receipt of a response by a process. It is thus a process action, but can nevertheless be meaningfully projected onto the object also.) We say that a program execution runs under consistency condition \( C \) if its projection onto each shared variable satisfies \( C \).

We say that an algorithm \( A \) solves mutual exclusion under consistency condition \( C \) if, for each execution of \( A \) that runs under \( C \), the following constraints hold:

- **mutual exclusion** (ME): there is at most one process in the critical section at any point in the execution.

- **eventual progress** (EP):\(^6\) if there is some process waiting to enter the critical section, then eventually some process enters the critical section.

- **no lockout** (NL): if some process is waiting to enter the critical section, then eventually that process enters the critical section.\(^7\)

We now examine the two mutual exclusion algorithms shown in Figure 8. Table 1 shows which of the conditions of mutual exclusion described above are met by each algorithm when implemented with variables satisfying each of our consistency conditions. As a comparison, we also list the conditions that are guaranteed by these algorithms when the shared variables are linearizable.

<table>
<thead>
<tr>
<th></th>
<th>MWR1</th>
<th>MWR2</th>
<th>MWR3</th>
<th>Linearizability</th>
</tr>
</thead>
</table>

*Table 1: Correctness of mutual exclusion algorithms using MW-regular variables.*

We first consider Peterson’s algorithm for two processors ([19]).\(^8\) This algorithm uses two single-writer shared variables and one multi-writer shared variable. The proof of the next theorem is very similar to the proof of Theorem 4.10 in [4]. Although [4] assumes that all the variables are atomic, the argument holds unchanged for variables that satisfy MWR1, and therefore MWR2 and MWR3 also.

**Theorem 9** Peterson’s Algorithm solves mutual exclusion under all the three definitions of regularity.

---

\(^6\) We use this term, rather than the more traditional ND (“no deadlock”) in order to avoid ambiguity: the term “deadlock” sometimes includes “livelock” (in which processes continue taking steps but keep one another trapped in a loop due to timing issues) and sometimes does not. The definition of “eventual progress” explicitly precludes either situation.

\(^7\) Although NL implies EP, we include both requirements, partly for historical reasons (e.g., [9]) but primarily because it gives us a finer gauge of the effectiveness of various consistency conditions, viz. Dijkstra’s algorithm, which solves EP but not NL under MWR2 and MWR3.

\(^8\) We use the presentation of the algorithm from [4].
Dijkstra's algorithm for $n$ processors uses $n$ single-writer shared variables and one multi-writer shared variable ([20]). Under both MWR2 and MWR3 it behaves the same way as under linearizability: ME and EP are guaranteed, but not NL. Under MWR1, only ME is guaranteed. The proof of the corresponding theorem is shown below.

**Theorem 10** Dijkstra's Algorithm satisfies ME under MWR1 and satisfies ME and EP under both MWR2 and MWR3, but does not satisfy NL under any of the conditions.

In the following proof, we use two relations between operations: $\rightarrow$ and $\rightarrow\rightarrow$ (defined in [12]). Informally, $op_1 \rightarrow op_2$ means that $op_1$ ends before $op_2$ starts, and $op_1 \rightarrow\rightarrow op_2$ means that $op_1$ starts before $op_2$ ends. The rule below holds for $\rightarrow$ and $\rightarrow\rightarrow$ ([12]):

$$\text{If } op_1 \rightarrow op_2 \rightarrow\rightarrow op_3 \rightarrow op_4, \text{ then } op_1 \rightarrow op_4$$

**Proof. (Theorem 10)** We first show that the algorithm satisfies ME under MWR1. Assume that two processes $p_0$ and $p_1$ enter the critical section simultaneously. It follows that both perform write operation $W(Flag[i], in-CS)$ (Line 6 of Dijkstra's algorithm in Figure 8), and that therefore neither of the read operations $R(Flag[j])$ (Line 7) return in-CS. Consider $R_0(Flag[1])$ and $W_1(Flag[1], in-CS)$. As $R_0(Flag[1])$ does not return in-CS by the argument above, it follows that $R_0(Flag[1])$ begins before $W_1(Flag[1], in-CS)$ ends, i.e., $R_0(Flag[1]) \rightarrow W_1(Flag[1], in-CS)$. Therefore, as each process performs only one operation at a time, we have $W_0(Flag[0], in-CS) \rightarrow R_0(Flag[1]) \rightarrow W_1(Flag[1], in-CS) \rightarrow R_1(Flag[0])$. It follows from the rule above that $W_0(Flag[0], in-CS) \rightarrow R_1(Flag[0])$. As there are no other writes to $Flag[0]$, it follows from the definition of MWR1 that $R_1(Flag[0])$ returns in-CS; therefore $p_1$ does not enter the critical section, and we have a contradiction.

However, Eventual Progress (EP) may be violated under MWR1. Consider the following execution:

$$w_0(Flag[0], requesting), r_0(Turn, 1), r_0(Flag[1], requesting), r_0(Turn, 1), ...$$

$$w_1(Flag[1], requesting), r_1(Turn, 0), r_1(Flag[0], idle), w_1(Turn, 1), r_1(Turn, 2), r_1(Flag[2], requesting), r_1(Turn, 2), ...$$

$$w_2(Flag[2], requesting), r_2(Turn, 0), r_2(Flag[0], idle), w_2(Turn, 2), r_2(Turn, 3), r_2(Flag[3], requesting), r_2(Turn, 3), ...$$

$$...$$

Initially, $Turn$ is set to 0 and $Flag[i]$ is set to idle for all $i$. In this execution, $p_0$ is slow at the beginning, so all the processes except $p_0$ enter the repeat loop and update their $Flag$ entries to requesting. Next, all the processes except $p_0$ enter the while loop and read $Turn = 0$ and $Flag[0] = idle$; thus they all write their ids to $Turn$. Suppose that these writes are performed concurrently.

Once $p_0$ updates its $Flag$ to requesting, each process continues by repeatedly reading $Turn$ in line 3 until it receives its own id as the result of some read. However, under MWR1 consistency, the order of concurrent writes may be observed differently by subsequent reads; thus any of these reads may return any of the concurrently written values of $Turn$, so there is no guarantee that any process will ever read its own id. If none does so, none will pass the while loop, and EP is violated.

Next we show that progress is guaranteed when the algorithm executes under MWR2. Note that each process writes to $Turn$ at most once before some process enters the critical section. By the definition of MWR2, there exists a sequence of all the write operations on $Turn$ such that all read operations that begin after the last write to $Turn$ return the id of that write. Thus, the process whose write is last in that sequence will pass the write loop and enter the critical section. Therefore progress occurs. As MWR3 is stronger than MWR2, MWR3 also guarantees progress.

It is easy to show that MWR3 does not guarantee NL. Consider an execution where all but one of the processes are waiting for the remaining process to leave the critical section and set its $Flag$ entry to idle. When this occurs, each of the remaining processes writes its own id to $Turn$. Suppose that
there is some process $p_k$ such that every time this scenario occurs, $p_k$ performs $w_k(Turn, id_k)$ earlier than anyone else. Then $p_k$’s write is always overwritten, so that $p_k$ never passes the while statement, and thus never enters the critical section, violating NL.

As MWR1 and MWR2 are weaker than MWR3, they also do not guarantee NL. This completes the proof of the theorem.

7 Conclusion

If Lamport’s consistency conditions continue to be of interest in the area of distributed shared memory, as seems likely, it is essential that these conditions be formally extended into the multi-writer model. While this extension is simple in the case of linearizability, it is more difficult and potentially ambiguous for the weaker condition of regularity.

In this paper we have given three possible formal extensions of Lamport’s definition of regularity from the single-writer model ([12]) to the more general multi-writer model. We have analyzed the relationships between these extended consistency conditions and a number of other well-known consistency conditions. We have given quorum-based algorithms to implement each of the extended consistency conditions, and proved their correctness. Finally, we have analyzed the correctness of two well-known algorithms for mutual exclusion under each of our proposed consistency conditions.

The still weaker condition of safeness [12] can also be extended to the multi-writer model by means of similar techniques to those we have used here; this is one possible avenue of future work. It might also be worthwhile to explore ways of formalizing the multi-writer version of consistency conditions met by the probabilistic quorum systems of [17], which operate more efficiently than strict quorum systems at the expense of occasionally providing outdated information.

References


Appendix: Existing Consistency Models

We give the formal definitions of several existing consistency conditions using the model we defined in Section 2.

Linearizability, also called atomicity, is the strongest consistency condition ([8]). It requires that there exist a total ordering of all the operations in an execution that respects both the semantics of the objects and the partial order of execution of the operations. The formal definition is given below.

Definition 8 (Linearizability) There exists a permutation π of all the operations in σ such that π is a legal sequence and, if the response of operation o1 occurs in σ before the invocation of operation o2, then o1 appears before o2 in π.

Sequential consistency([11]) requires that there exist a total order of all the operations in an execution that respects the semantics of the objects and is consistent with the order of operations executed by each process.
Definition 9 (Sequential Consistency) There exists a permutation π of all the operations in σ such that π is a legal sequence and, if the response for operation o₁ at process pᵢ occurs in σ before the invocation for operation o₂ at process pᵢ, then o₁ appears before o₂ in π.

PRAM was introduced in [13]. This consistency condition requires that the write operations of a process be observed by other processes in the order in which they are performed. Formally speaking, a memory consistency condition is PRAM if it satisfies the following:

Definition 10 (PRAM) For each process pᵢ, there is a permutation πᵢ of all the operations in (σ|ᵢ ∪ σ|w) such that πᵢ is a legal sequence and, for all j, if the response for operation o₁ at process pⱼ occurs in σ before the invocation for operation o₂ at process pⱼ, then o₁ appears before o₂ in πᵢ.

Coherence ([7]) requires sequential consistency on a per-object basis, which means that the operations on different objects executed by the same process may be observed in an order other than that in which they are invoked.

Definition 11 (Coherence) For each variable x, there exists a permutation πₓ of all the operations in σ|x such that πₓ is a legal sequence and, if the response for operation o₁ at process pᵢ occurs in σ before the invocation for operation o₂ at process pᵢ, then o₁ appears before o₂ in πₓ.

Goodman’s Processor Consistency (PCG) is rigorously defined in [1]. It is a combination of coherence and PRAM.

Definition 12 (PCG) For each process pᵢ, there exists a permutation πᵢ of all the operations in (σ|ᵢ ∪ σ|w) such that:

- πᵢ is a legal sequence.

- if the response for operation o₁ at process pⱼ (j may equal to i) occurs in σ before the invocation for operation o₂ at process pⱼ, then o₁ appears before o₂ in πᵢ.

- for each variable x, if there are two write operations w₁, w₂ on x and w₁ occurs before w₂ in πᵢ, then w₁ appears before w₂ in πⱼ (the legal permutation for process pⱼ).