View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M Repository

Multi-Writer Consistency Conditions for Shared Memory Registers*

Texas A&M University Department of Computer Science and Engineering Technical Report 2010-1-1

Cheng Shao Jennifer L. Welch Evelyn Pierce Hyunyoung Lee

Abstract

Regularity is a shared memory consistency condition that has received considerable attention. Lamport's original definition of regularity assumed a single-writer model, however, and is not well defined when the shared register may have multiple writers. In this paper, we consider four possible definitions of multi-writer regularity. The definitions are motivated by variations on a quorum-based algorithm schema for implementing them. 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 variables — 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 application 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.

Perhaps the most desirable consistency condition for shared memory variables is *atomicity* (Lamport [17]), also known as *linearizability* (Herlihy and Wing [13]), in which read and write operations behave as though they were executed sequentially, i.e., with 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 some systems, the related but weaker condition of *regularity* (Lamport [17]) may be

*This work was done while the authors were at Texas A&M University. This work was supported in part by NSF grant 0098305; NSF grant 0500265; Texas Higher Education Coordinating Board grants ARP-00512-0091-2001, ARP-00512-0007-2006, and NHARP 000512-0130-2007; and Texas Engineering Experiment Station funds. A preliminary version of this paper appeared in the 2003 International Symposium on Distributed Computing [29].

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* (Attiya et al. [4], Malkhi and Reiter [21], Lee and Welch [18], Lynch and Shvartsman [20]).

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 register in any execution is well-defined; in fact, it was for this model that Lamport gave his definition of regularity [17]. In a multi-writer model, however, multiple processes may perform overlapping write operations to the same register 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 Malkhi and Reiter [21]:

- 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 precise definition of multi-writer regularity, but to our knowledge none has yet been proposed.

1.2 Contributions

In this paper, we extend the notion of regularity to a multi-writer model. Specifically, we propose four possible 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 first condition is implemented with a basic algorithm, while the other three conditions are obtained by adding three different mechanisms to the basic algorithm. Our algorithms are designed for asynchronous message-passing systems in which no messages are lost. For simplicity of presentation, we assume that no processes are faulty; in the conclusion we discuss how to accommodate crash failures of processes.

The definitions form a lattice with respect to their strength, and the implementations have varying costs with respect to number of messages, size of messages, time delay, and local memory requirements. Taken together, the definitions point out the ambiguity of the informal notion of multi-writer regularity and the algorithms suggest that different costs may be associated with different choices for disambiguating.

A consistency condition is said to be local if the consistency condition is satisfied on a per-variable basis. Locality is a desirable property of consistency conditions: as mentioned in Herlihy and Wing [13], locality enhances modularity and concurrency. We show that all our proposed definitions satisfy locality. We also study the relationships between our definitions of multi-writer regularity and several existing consistency conditions.

Finally, we provide a practical context for our results by studying the behavior of two well-known algorithms for mutual exclusion when the variables satisfy our proposed consistency conditions. The algorithms we examine are Peterson's algorithm for two processes [23] and Dijkstra's algorithm (as presented by Raynal [25]). We find that

[1] In the *single-writer* model, only one process can write to each shared variable (and the writes are sequential); other processes can only read from it.

Peterson's algorithm remains correct under all the new conditions. Dijkstra's algorithm satisfies only some of the constraints of the mutual exclusion problem under the new conditions.

1.3 Related Work

There is copious literature on consistency conditions for shared memory, both implementations and applications (e.g., Lamport [16], Lipton and Sandberg [19], Goodman [12], Herlihy and Wing [13], Ahamad et al. [1, 2], Raynal and Schiper [26], and Garg and Raynal [11]). Our work builds on the notion of regularity as introduced by Lamport [17].

Friedman et al. [10] and Steinke and Nutt [30] identify building blocks and use various combinations of such building blocks to explore potential consistency conditions. The difference between our work and theirs is that in theirs the building blocks are identified at the definition level while in our work, the building blocks are identified at the implementation level. We use a similar framework and system model to those introduced by Raynal and Schiper [27]; the major difference is that the partial order used by Raynal and Schiper is a combination of per-process order and the "reads-from" relation, while we use the real-time order in which operations occur.

The locality property of consistency conditions was first proposed and studied by Herlihy and Wing [13]. Vitenberg and Friedman [31] studied the locality of a set of conditions and developed some general criteria for when a condition is local and when it is not.

Our algorithms are based on quorums. The use of quorum systems in distributed computing for replicating data has a long history. The most relevant papers to our work are those by Malkhi and Reiter [21], Bazzi [7], and Malkhi et al. [22]. Our generic algorithm schema is a generalization of these algorithms, although without provisions for tolerance to Byzantine failures of servers.

We follow the example of Attiya and Friedman [5], Ahamad et al. [1], and Higham and Kawash [14] in using the mutual exclusion problem as an application for our consistency conditions. Attiya and Friedman [5] revised Peterson's 2-process algorithm [23] to solve the mutual exclusion problem under their hybrid consistency model. Ahamad et al. [1] examined the correctness of Peterson's algorithm and Lamport's bakery algorithm [15] 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. Higham and Kawash [14] investigated other mutual exclusion algorithms, including Dekker's and Dijkstra's, none of which guarantees mutual exclusion under PCG.

1.4 Roadmap of Paper

The rest of the paper is organized as follows. Section 2 consists of the definitions we will use to discuss shared memory consistency conditions, as well as a description of our system model, and an overview of quorum-based shared memory algorithms. In Section 3 we describe our generic quorum-based algorithm and the different building block mechanisms for implementing a shared read/write variable. Section 4 presents our proposed definitions of multi-writer regularity and their implementations. In Section 5 we discuss the locality property of our proposed definitions and compares their relative strengths. In Section 6 we study the correctness of two mutual exclusion algorithms — Peterson's algorithm for two processes and Dijkstra's algorithm — when their shared variables satisfy our proposed definitions. Section 7 concludes this paper and discusses future work.

2 Preliminaries

2.1 Shared Read/Write Registers and Consistency Conditions

A shared *read/write register* supports concurrent execution of read and write operations performed by some set, P_A, of n application processes, $p_0, p_1, \ldots, p_{n-1}$. Each operation has an *invocation* and a *response*. For a read

operation, the invocation is denoted ***read***$_i$ (where i is the index of the application process performing the operation), and the response has the form ***return***$_i(v)$, where v, the value read from the register, is drawn from a predetermined set of possible values that the register can take on. For a write operation, the invocation has the form ***write***$_i(v)$, where v is the value to be written to the register, and the response is denoted ***ack***$_i$, indicating that the write operation has completed.

If the operations on a register occur sequentially, without any overlap, so that each invocation is immediately followed by a *matching* response (i.e., each ***read***$_i$ is followed by a ***return***$_i$ and each ***write***$_i$ is followed by an ***ack***$_i$), then we would expect each read to return the value of the latest preceding write. This behavior is captured in the next definition; note that it is only relevant to sequences in which operations do not overlap.

Definition 1 *A total order of operations on a shared register is* legal *if each read returns the value of the latest preceding write; if there is no preceding write, then the read returns the initial value of the register.*

When operations do overlap, because of concurrent execution by multiple processes, invocations and responses are interleaved. We assume, however, that each process has at most one operation pending at a time. To capture such "well-formedness" constraints, we define the notion of a schedule next. If σ is a sequence of operation invocations and responses, we denote by $\sigma|i$ the subsequence of σ containing all the invocations and responses performed by process p_i.

Definition 2 *A sequence σ of invocations and responses is a* schedule *if, for each i, $0 \leq i < n$, the following hold:*

- *$\sigma|i$ consists of alternating invocations and matching responses, beginning with an invocation; and*
- *if the number of steps taken by p_i is finite, then the last step by p_i is a response, i.e., every invocation has a matching response.*

Note that this definition of a schedule allows arbitrary asynchrony of process steps, i.e., no constraints are placed on the relative speed with which operations complete or on the time between operation invocations. However, for convenience of analysis, we follow the example of Lamport [17] and Chandra and Toueg [9] 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. This is equivalent to the global-time model introduced by Ben-David [8] and Anger [3].

A *consistency condition* is specified by a particular set of schedules. Thus the relative strength of two consistency conditions can be compared by considering the sets of schedules defining the two conditions; in particular, consistency condition C_1 is *stronger* than consistency condition C_2 if $C_1 \subset C_2$.

By the definition of a schedule, each invocation has a matching response, namely the response by the same process that follows it most closely; an invocation and its matching response form an operation. Given a schedule σ, we denote by $ops(\sigma)$ the set of all operations whose invocations and responses appear in σ.[2] We use $writes(\sigma)$ and $reads(\sigma)$ to denote the set of all writes and the set of all reads appearing in $ops(\sigma)$.

We define a partial order on $ops(\sigma)$, denoted $<_\sigma$, as $op_1 <_\sigma op_2$ if and only if the response of op_1 occurs in σ before the invocation of op_2. We frequently are concerned with a partial order on a subset S of $ops(\sigma)$ where the ordering relation is inherited from σ; we denote such a partial order by $(S, <_\sigma)$. A total order on a subset of $ops(\sigma)$ that respects the partial order $<_\sigma$ is said to be a *linearization* of the partial order; i.e., if $op_1 <_\sigma op_2$, then op_1 precedes op_2 in the total order. We call such a total order *σ-consistent*.

We now use our framework to state Lamport's original definition of (single-writer) regularity, which we call SWReg. A schedule is *single-writer* if only one process invokes write operations.

[2] Assume for convenience that each operation in σ has a unique id, for instance, the j-th operation invoked by process p_i; this mechanism allows us to distinguish between two reads (or two writes) of the same value by the same process that occur at different points in the schedule.

Definition 3 *A single-writer schedule σ satisfies* SWReg *if, for every read r in $ops(\sigma)$, there exists a legal linearization of $(writes(\sigma) \cup \{r\}, <_\sigma)$. A shared register satisfies SWReg if all schedules on it satisfy SWReg.*

The next definition identifies writes that could possibly influence a read, namely those that start before the read ends.

Definition 4 *A write w in $ops(\sigma)$ is* relevant *to a read r in $ops(\sigma)$ if $r \not<_\sigma w$;* rel-writes(σ, r) *is the set of all writes in $ops(\sigma)$ that are relevant to r.*

We next formalize the notion of a read reading from a write[3]. To model a read returning the initial value of the register, we posit the existence in $ops(\sigma)$ of a special write, w_{init}, which writes the initial value of the register and satisfies $w_{init} <_\sigma op$ for every other op in $ops(\sigma)$.

Definition 5 *Given a schedule σ, consider a function ρ from $reads(\sigma)$ to $writes(\sigma)$. ρ is a* reads-from *function if for each read r, the value returned by r is the same as the value written by $\rho(r)$, $\rho(r)$ is relevant to r, and there is no write w in $writes(\sigma)$ such that $\rho(r) <_\sigma w <_\sigma r$.*

If $w = \rho(r)$, we say that *r reads from w with respect to ρ*; when ρ is understood from context, we simply say that *r reads from $\rho(r)$*. A read r can only read from a write w if either w overlaps r, or w precedes r and no other write is strictly between w and r. The reads-from function is not necessarily unique for a schedule, and in fact might not exist.

2.2 System Model

We assume a system consisting of a collection P of processes that communicate with each other through message-passing. Each *process* is modeled as a (possibly infinite) state machine, with an initial state and a transition function. The state machine represents the code for the register simulation that is running at the process. A *configuration* of the system is a vector of local states, one per process. An *initial* configuration contains an initial state for each process.

There are two kinds of *events* that can occur in the system, input and output events. Each event occurs at a single process. The input events are the receipt of a message and the invocation of a shared register operation. The output events are the sending of a message and the response of a shared register operation. Each input event triggers its corresponding process to take a step: the transition function is applied to the current state of the process and the particular event, and produces a new state of the process and a set of output events. The output events consist of a set of messages sent by the process and at most one shared-register operation response to occur at the process.

An *event list* is a sequence of events, all taking place at the same process, that begins with an input, followed by any number of message sends, and ends with at most one operation response.

An *execution* is a sequence[4] $d_0\ell_1 d_1\ell_2 d_2 \ldots$ of alternating configurations d_k and event lists ℓ_k, starting with an initial configuration d_0, that satisfies the following conditions.

- Consider any $d_{k-1}\ell_k d_k$ in the sequence, where ℓ_k takes place at process q_i. Then applying q_i's transition function to q_i's state in d_{k-1} and the first event in ℓ_k produces the remaining events in ℓ_k and q_i's state in d_k. All other components of d_k are the same as in d_{k-1}. That is, the process states and events occurring in the sequence are consistent with the processes' transition functions.

[3]This definition is similar to that of *writes-into order* from Ahamad et al. [2]

[4]Event lists occurring simultaneously at different processes appear in the execution in arbitrary order. Since each event list is concerned only with local computation at a single process, the common assumption that local processing time is negligible compared to message delays allows us to model concurrent events as a sequence without loss of generality.

- Every message sent is received exactly once and subsequent to its send; only messages sent are received. That is, the communication is reliable.
- If event list ℓ_k occurring at process q_i begins with an operation invocation, then the most recent preceding invocation or response at q_i (if any) is a response. That is, the component that is generating the invocations waits for one operation to finish before invoking the next one.

We can now state our main correctness condition for simulating a register with a particular consistency condition.

Definition 6 *The system* implements a read/write register with consistency condition C *if, for every execution of the system, the projection onto the set of invocations and responses of the register is a schedule that is in (i.e., satisfies)* C.

2.3 Quorum Systems

Although having processes communicate through shared variables is generally viewed as desirable from a software development perspective, most distributed systems do not directly provide such functionality. However, the illusion of shared variables can be provided through a shared variable simulation layer that runs in a message-passing communication environment. This software layer simulates shared registers on top of the message-passing layer.

The algorithms in this paper for simulating a shared register use the notion of a quorum system, which is a technique for handling replicated data. Some processes in the system play the role of "servers", which maintain replicas, while others play the role of "clients", which handle invocations of operations on the replicated data. There is one client process corresponding to each application process in P_A. Let $P_{\mathcal{S}}$ be the set of server processes and $P_{\mathcal{C}}$ be the set of client processes. (It is possible for a single physical node to host both a client and a server process.)

A *quorum system* $\mathcal{Q}$ (over $P_{\mathcal{S}}$) is a collection of subsets of $P_{\mathcal{S}}$, each of which is called a *quorum*, satisfying the property that for every two distinct quorums Q and Q', $Q \cap Q' \neq \emptyset$.

3 Algorithm Schema

3.1 Generic Algorithm

A generic algorithm that uses quorums to implement a shared read/write register with initial value v_0 is given in Figure 1. Upon receiving a read or write invocation on the shared register, a client process chooses a quorum using some quorum selection strategy and then queries each member of this quorum about its current "view" of the shared register, which consists of the value of the register and the timestamp associated with the value. After gathering all the responses, the process decides which timestamp among the responses is the latest, using function $MaxTS()$. The operations then continue as follows:

- **write:** The process increments the timestamp returned by $MaxTS()$, using the function $IncTS()$, sends an UPDATE message with the new value and the incremented timestamp to every member of some quorum, and waits to receive a DONE message from each quorum member.
- **read:** The process calls the function $GetValue()$, which uses the timestamp returned by $MaxTS()$ to decide which value will be returned. The reading process then calls a function $WriteBack()$, which optionally updates a quorum of servers regarding the value that the process plans to return.

The server responds to queries by sending the value and timestamp information that it has stored, and responds to an UPDATE message by setting its stored information to that in the message if the timestamp in the message is larger than the stored timestamp.

Code for client process $c_i \in P_{\mathcal{C}}$:

$write_i(v)$:
1 for some quorum $Q \in \mathcal{Q}$, send $\langle$QUERY$\rangle$ to each $s_j \in Q$
2 wait to receive $\langle$VIEW$, u, t\rangle$ from each $s_j \in Q$
3 $V :=$ set of (u, t) pairs received in Line 2
4 $ts :=$ *MaxTS*(V)
5 $ts :=$ *IncTS*(ts)
6 $val := v$
7 for some quorum $Q' \in \mathcal{Q}$, send $\langle$UPDATE$, val, ts\rangle$ to each $s_j \in Q'$
8 wait to receive $\langle$DONE$\rangle$ from each $s_j \in Q'$
9 $ack_i()$

$read_i()$:
1 for some quorum $Q \in \mathcal{Q}$, send $\langle$QUERY$\rangle$ to each $s_j \in Q$
2 wait to receive $\langle$VIEW$, u, t\rangle$ from each $s_j \in Q$
3 $V :=$ set of (u, t) pairs received in Line 2
4 $t :=$ *MaxTS*(V)
5 $v :=$ *GetValue*(V, t)
6 *WriteBack*()
7 *return*$_i(v)$

Code for server process $s_j \in P_{\mathcal{S}}$:

local variables: /* values persist over time */
val /* local copy of shared register, initially v_0 */
ts /* local copy of timestamp, initially smallest timestamp value */

When s_j receives $\langle$QUERY$\rangle$ from c_i:
1 send $\langle$VIEW$, val, ts\rangle$ to c_i

When s_j receives $\langle$UPDATE$, v, t\rangle$ from c_i:
1 if ($ts < t$) then
2 $val := v$
3 $ts := t$
4 endif
5 send $\langle$DONE$\rangle$ to c_i

Figure 1. A generic quorum-based algorithm to implement a shared read/write register

By plugging in different implementations for the functions $MaxTS()$, $IncTS()$, $GetValue()$, and $WriteBack()$, we obtain registers satisfying different consistency conditions. This algorithm is a generalization of several existing quorum-based protocols. For example, the appropriate instantiations of the functions yield the algorithms by Malkhi and Reiter [21], Bazzi [7], and Malkhi et al. [22].

The following lemma states a key property of the generic algorithm which follows directly from the code.

Lemma 1 *Assume $IncTS()$ always returns a timestamp larger than its argument. Then the sequence of timestamp values taken on by the replica on any server is nondecreasing during any execution of the generic algorithm.*

We denote by $ts(op)$ the timestamp of operation op. For a write operation, this is the timestamp appearing in Line 7 of the $write_i$ procedure. For a read operation, this is the timestamp associated with the value returned in Line 7 of the $read_i$ procedure.

3.2 Building Blocks in the Generic Algorithm

We identify three building blocks that we use to create specific instantiations from the generic algorithm. Different combinations of the building blocks give us different algorithms, which in turn yield shared registers with different

consistency conditions. The three building blocks are:

Code modifications for client process $c_i \in P_{\mathcal{C}}$ without ID building block:

timestamp type is integer

MaxTS(V):
1 $T := \{t : (v, t) \in V$ for some $v\}$
2 return $\max(T)$ /* $\max$ operates on integers */

IncTS(ts):
1 return $ts + 1$

Code modifications for client process $c_i \in P_{\mathcal{C}}$ with ID building block:

timestamp type is ordered pair of integers

MaxTS(V):
1 $T := \{t : (v, t) \in V$ for some $v\}$
2 return $\max(T)$ /* $\max$ operates lexicographically on ordered pairs of integers */

IncTS$((t, id))$:
1 return $(t + 1, i)$

Figure 2. Code with and without ID building block

Code modifications for client process $c_i \in P_{\mathcal{C}}$ without WB building block:

WriteBack():
1 return /* do nothing */

Code modifications for client process $c_i \in P_{\mathcal{C}}$ with WB building block:

WriteBack():
1 for some quorum $Q' \in \mathcal{Q}$, send $\langle$UPDATE$, val, ts\rangle$ to each $s_j \in Q'$
2 wait to receive $\langle$DONE$\rangle$ from each $s_j \in Q'$
3 return

Figure 3. Code with and without WB building block

- **Building block ID: Including unique id in timestamp.** (See Figure 2 for pseudocode.) If this building block is not used, then timestamps are natural numbers, $MaxTS()$ returns the largest integer in its argument, and $IncTS()$ increments its argument by one. Note that timestamps are not necessarily unique.

  If this building block is used, then timestamps are ordered pairs of natural numbers, the first component being a counter and the second component being a process id. $MaxTS()$ returns the largest timestamp in lexicographic order among its arguments, and $IncTS()$ increments the first component of its argument by one and replaces the second component with the id of the executing process. The cost of using this building block is an additional $O(\log n)$ bits to store a timestamp.

- **Building block WB: Write-back phase in the read procedure.** (See Figure 3 for pseudocode.) When this building block is not used, the read procedure does nothing in Line 6.

Code modifications for client process $c_i \in P_C$ without LC building block:

GetValue(V, t):
1 $S :=$ set of all elements of V with timestamp t
2 choose any element $(v, t) \in S$
3 return v

Code modifications for client process $c_i \in P_C$ with LC building block:

local variables: /* local cache: values persist over time */
val /* local copy of shared register, initially v_0 */
ts /* local copy of timestamp, initially smallest timestamp value */

GetValue(V, t):
1 if $t \leq ts$ then /* either compares ints or compares pairs of ints lexicographically */
2 return val
3 else
4 $S :=$ set of all elements of V with timestamp t
5 choose any element $(v, t) \in S$
6 $val := v$ /* record return value in local cache */
7 $ts := t$ /* record associated timestamp in local cache */
8 return v
9 endif

Figure 4. Code with and without LC building block

When this building block is used, after choosing the return value, the reader sends the value to be returned and its associated timestamp in an UPDATE message to all the servers in some quorum. After a DONE message is received from these servers, the read returns. The cost of using this building block is an extra $O(c)$ messages, where c is the size of the biggest quorum in the system. Furthermore, the time for a read is increased by a round-trip message delay.

- **Building block LC: Local cache at clients**. (See Figure 4 for pseudocode.) If this building block is not used, then $GetValue(V, t)$ returns any element contained in V whose associated timestamp is t (there might be more than one such element if the ID building block is not used).

  If this building block is used, then each client keeps a copy of the most recently written or read value and its associated timestamp. This information persists even when no operation is currently underway at the client. Specifically, the local variables val and ts at each client are declared as persistent. The write procedure updates them with the value to be written and the timestamp calculated for that value; the parameter to $IncTS()$ is the cached timestamp ts. For the read procedure, in $GetValue(V, t)$, if t, the largest timestamp obtained from querying a quorum, does not exceed the cached timestamp ts, then the cached value is returned. Otherwise, any element in V with timestamp t is chosen to be returned and the cached value and timestamp are updated to be this value and t respectively. The cost of using this building block is that clients now have to keep this information for each shared register and thus it introduces space and robustness issues.

3.3 Lattice of Algorithms and Conditions

A summary of our results is shown in Figure 5. The lattice shows all the algorithms instantiated from the generic algorithm by applying different combinations of the building blocks. Above each algorithm (rectangle) in the lattice is the name of the consistency condition that the corresponding algorithm implements. The arrows go from a weaker condition to a stronger condition. In the next section, we will walk up the lattice to present the algorithms and their

associated consistency conditions. For now we only focus on the case of a single register. In Section 5, we discuss extensions to multiple registers.

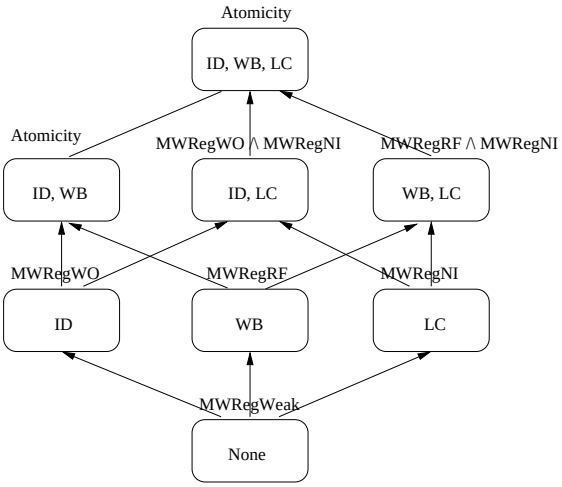

Figure 5. Lattice of algorithms and consistency conditions

4 Multi-Writer Consistency Conditions: Specifications and Implementations

4.1 MWRegWeak: No Building Blocks

We first consider the generic algorithm when none of the three building blocks is used, denoted Alg_None. We specify a condition that we call MWRegWeak and show that Alg_None implements this condition.

Definition 7 (MWRegWeak) *A schedule* σ satisfies MWRegWeak *if, for every read operation* r *in* $ops(\sigma)$*, there exists a legal linearization of* $(writes(\sigma) \cup \{r\}, <_\sigma)$*. A shared register satisfies MWRegWeak if all schedules on it satisfy MWRegWeak.*

A schedule satisfies MWRegWeak if each read r returns the value of some write w that either overlaps or precedes r, as long as no other write falls completely between w and r. Different reads are allowed 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 the schedule.

Figure 6 shows a schedule that satisfies MWRegWeak. (In our figures, $W(x, v)$ denotes a write operation that writes value v to register x, and $R(x, v)$ denotes a read operation on register x that returns value v. Time increases from left to right. We use similar schedules to illustrate other proposed definitions; the schedules differ only in the return values of some of the read operations.) A possible linearization for each read is given below.

$$
\begin{array}{ll}
R_1\text{:} & W(x,2), W(x,1), R_1(x,1), W(x,4), W(x,3) \\
R_2\text{:} & W(x,1), W(x,2), R_2(x,2), W(x,3), W(x,4) \\
R_3\text{:} & W(x,1), W(x,2), R_3(x,2), W(x,4), W(x,3) \\
R_4\text{:} & W(x,1), W(x,2), W(x,4), R_4(x,4), W(x,3) \\
R_5\text{:} & W(x,1), W(x,2), R_5(x,2), W(x,3), W(x,4) \\
R_6\text{:} & W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4)
\end{array}
$$

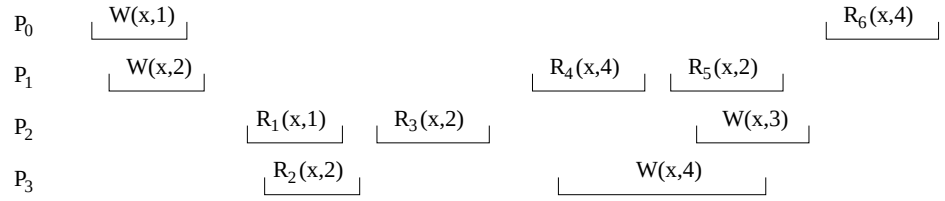

Figure 6. Schedule that satisfies MWRegWeak

The following lemma states that the *timestamp order* (numerical order by timestamp) of certain operations extends the partial order $<_\sigma$.

Lemma 2 *Consider any execution of Alg_None and let σ be its schedule.*

(a) For every read operation r and every write operation w in $ops(\sigma)$, if $w <_\sigma r$, then $ts(w) \leq ts(r)$.

(b) For every pair of write operations w_1 and w_2 in $ops(\sigma)$, if $w_1 <_\sigma w_2$, then $ts(w_1) < ts(w_2)$.

Proof. (a) Suppose write w ends before read r begins in σ. Let s be a server process that is in the intersection of the quorum that w uses for its update (Lines 7-8) and the quorum that r uses for its query (Lines 1-2). According to Lemma 1, the sequence of timestamp values taken on at s is non-decreasing. Since w finishes before r starts, s returns to r a timestamp that is at least $ts(w)$. Since r chooses the value associated with the largest timestamp returned from its query, r's timestamp is no less than w's.

(b) Using a similar argument to that in (a), we can show that some server process s returns to w_2 a timestamp that is at least as large as $ts(w_1)$. Since $ts(w_2)$ is larger than the largest timestamp obtained in the query, w_2's timestamp is larger than w_1's. ■

We next define a function ρ from reads to writes and then show that it is a reads-from function. Given a schedule σ of Alg_None, for each read r in $reads(\sigma)$, choose a write in $writes(\sigma)$ to be $\rho(r)$ as follows. Let (v, t) be the element of V chosen in the execution of GetValue(V, t) as containing the value to be returned. The reason (v, t) is in V is that the client executing r previously received a VIEW message from some server s_j containing (v, t). If $t = 0$ (the initial timestamp), then let $\rho = w_{init}$. Otherwise, since every executed write calculates a positive timestamp and since servers are not faulty and communication is reliable, s_j sent this VIEW message because it had earlier received an UPDATE message containing (v, t) from some client on behalf of a write w. Let $\rho(r) = w$ in this case; if there are multiple choices for $\rho(r)$, choose one arbitrarily. The function ρ is not necessarily unique, but it does not need to be. In the analysis of Alg_None, references to "reads from" are with respect to this specific reads-from function. Note that $ts(r) = ts(\rho(r))$.

Lemma 3 *For every schedule σ of Alg_None, the function ρ just defined is a reads-from function.*

Proof. We show that ρ satisfies the three properties of a reads-from function. (1) By construction, the value returned by r is the value written by $\rho(r) = w$. (2) If $\rho(r) = w_{init}$, then $\rho(r)$ is relevant to r because $w_{init} <_\sigma r$. If $\rho(r) \neq w_{init}$, then $\rho(r)$ is relevant to r since messages are not received before they are sent. (3) Suppose in contradiction there is some other write $w' \in writes(\sigma)$ such that $w <_\sigma w' <_\sigma r$. By Lemma 2(b), $ts(w) < ts(w')$, and by Lemma 2(a), $ts(w') \leq ts(r)$, contradicting the fact that by construction, $ts(w) = ts(r)$. ■

Theorem 4 *Algorithm Alg_None implements MWRegWeak.*

Proof. Consider any execution of Alg_None, let σ be its schedule and ρ be its reads-from function. For each read operation r in $ops(\sigma)$, we construct a total order L_r on $writes(\sigma) \cup \{r\}$ as follows. Let $\rho(r) = w^*$ and $ts(r) = t^*$. Order the writes in *rel-writes*(σ, r) in any total order consistent with their timestamps subject to the condition that w^* is the latest among all writes with timestamp t^*. Order the writes not in *rel-writes*(σ, r) in any total order consistent with their timestamps. Order every write in *rel-writes*(σ, r) before every write not in *rel-writes*(σ, r). Finally, order r immediately after $\rho(r)$. L_r is legal by construction.

We now show that L_r is σ-consistent. Consider any write w such that $w <_\sigma r$. If $ts(w) \leq t^*$, then w is ordered before r in L_r by construction. By Lemma 2(a), it is not possible for $ts(w)$ to be greater than t^*.

Consider any write w such that $r <_\sigma w$. Since w is not in *rel-writes*(σ, r), w appears after r in L_r.

Consider two writes w_1 and w_2 with $w_1 <_\sigma w_2$. If both are in *rel-writes*(σ, r) or both are not in *rel-writes*(σ, r), then they are ordered consistently in L_r by Lemma 2(b) and construction. If w_1 is in *rel-writes*(σ, r) and w_2 is not in *rel-writes*(σ, r), then they are ordered consistently in L_r by construction. Since $w_1 <_\sigma w_2$, it is not possible for w_2 to be in *rel-writes*(σ, r) and w_1 not to be. ■

4.2 MWRegWO: Building Block ID

MWRegWeak is a weak consistency condition in that the read operations do not have a common view on the order of preceding write operations even for the read operations performed by the same process. For instance, in Figure 6, p_2's first read, R_1, indicates that the write of 1 should follow the write of 2, but p_2's next read, R_3, indicates the opposite. By using the ID building block to break ties between timestamps, we can obtain a stronger condition, called MWRegWO. (WO is for "write order".) This condition requires the linearization of two reads to agree on the ordering of all writes that are relevant for both the reads. For writes that are relevant to only one or none of them, the total orders are allowed to differ.

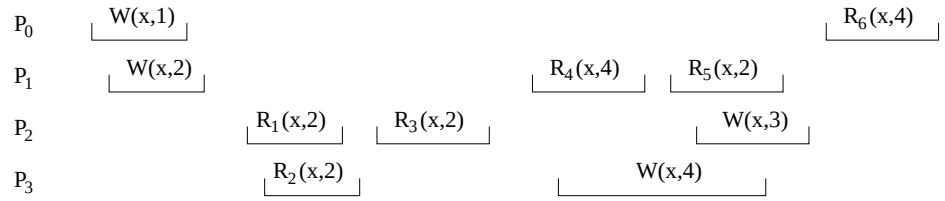

Figure 7. Schedule that satisfies MWRegWO

Definition 8 (MWRegWO) *A schedule* σ satisfies MWRegWO *if for each read* r *in* $ops(\sigma)$*, there is a legal linearization* L_r *of* $(writes(\sigma) \cup \{r\}, <_\sigma)$*, satisfying the following condition. For all reads* r_1 *and* r_2 *in* $ops(\sigma)$*, for all writes* w_1 *and* w_2 *in rel-writes*$(\sigma, r_1) \cap$ *rel-writes*(σ, r_2)*, it holds that* $w_1 <_{L_{r_1}} w_2$ *if and only if* $w_1 <_{L_{r_2}} w_2$*. A shared register satisfies MWRegWO if all schedules on it satisfy MWRegWO.*

The schedule in Figure 6 does not satisfy MWRegWO, since R_1's linearization must have $W(x, 1)$ after $W(x, 2)$, but R_2's linearization must have $W(x, 2)$ after $W(x, 1)$. However, the schedule in Figure 7 does satisfy MWRegWO; a linearization for each read is given below. The order of $W(x, 3)$ and $W(x, 4)$ in R_4's total order differs from that in R_6's total order (and it must do so); this difference is allowable since $W(x, 3)$ is not a relevant write for R_4.

$R_1 : W(x,1), W(x,2), R_1(x,2), W(x,3), W(x,4)$
$R_2 : W(x,1), W(x,2), R_2(x,2), W(x,3), W(x,4)$
$R_3 : W(x,1), W(x,2), R_3(x,2), W(x,3), W(x,4)$
$R_4 : W(x,1), W(x,2), W(x,4), R_4(x,4), W(x,3)$
$R_5 : W(x,1), W(x,2), R_5(x,2), W(x,3), W(x,4)$
$R_6 : W(x,1), W(x,2), W(x,3), W(x,4), R_6(x,4)$

We now show that by using the ID building block, as in the algorithm of Malkhi and Reiter [21], the resulting register satisfies MWRegWO. We call this algorithm Alg_ID. Since we use the timestamp of Alg_None as the first element of the timestamp in Alg_ID, Lemma 2 still holds. We define the function ρ for Alg_ID as we did in Section 4.1 for Alg_None. Lemma 3 still holds and thus ρ is a reads-from function for Alg_ID. The use of the process id in the second element of the timestamp as a tiebreaker further ensures that:

Lemma 5 *The write operations performed using Algorithm Alg_ID are totally ordered by timestamp.*

Theorem 6 *Algorithm Alg_ID implements MWRegWO.*

Proof. Consider any execution of Alg_ID, let σ be its schedule and ρ the reads-from function defined above for Alg_ID. Let r be any read in $ops(\sigma)$. We construct L_r, a linearization of $writes(\sigma) \cup \{r\}$, as follows. Order every write in *rel-writes*(σ, r) before any write not in *rel-writes*(σ, r). Order all the writes in *rel-writes*(σ, r) by timestamp order. Order all the writes not in *rel-writes*(σ, r) by timestamp order. Order r immediately after $\rho(r)$ (the write from which it reads), ensuring that L_r is legal.

We now show that L_r is σ-consistent. For any two writes in *rel-writes*(σ, r) and for any two writes not in *rel-writes*(σ, r), σ-consistency follows from Lemma 2(b). For write w_1 in *rel-writes*(σ, r) and write w_2 not in *rel-writes*(σ, r), σ-consistency follows from the fact that w_1 starts before r ends and w_2 starts after r ends, and thus $w_2 \not<_\sigma w_1$. For read r and any write w such that $r <_\sigma w$, σ-consistency follows from the fact that w is not in *rel-writes*(σ, r) but $\rho(r)$ is in *rel-writes*(σ, r) since ρ is a reads-from function. For read r and any write w such that $w <_\sigma r$, σ-consistency follows if we can show that $ts(w) \leq ts(w')$, where $w' = \rho(r)$. By definition of ρ, $ts(r) = ts(w')$, and by Lemma 2 (a), $ts(w) \leq ts(r)$.

Now we show that all linearizations agree on the order of relevant writes. Consider two reads, r_1 and r_2, in $ops(\sigma)$ and two writes, w_1 and w_2, that are both in *rel-writes*$(\sigma, r_1) \cap$ *rel-writes*(σ, r_2). By construction of L_{r_1} and L_{r_2} and by Lemma 5, w_1 and w_2 are ordered in both linearizations in timestamp order. ■

4.3 MWRegRF: Building Block WB

In this subsection, we consider the use of the write-back (WB) building block, resulting in algorithm Alg_WB. The use of the writeback prevents old-new inversions in the values returned by reads. To capture the condition ensured by this mechanism, we need the concept of a partial order that respects reads-from relationships between reads and writes.

For a given schedule σ and a reads-from function ρ on σ, define the relation $<_{\sigma,\rho}$ on $ops(\sigma)$ to be the transitive closure of the union of the $<_\sigma$ and ρ orders[5]. We show that this relation is a partial order. (All references to "reads from" in the definition of the next consistency condition are with respect to the reads-from function ρ just fixed.)

Lemma 7 $<_{\sigma,\rho}$ *is a partial order.*

Proof. Suppose in contradiction $<_{\sigma,\rho}$ is not a partial order. Let C be a shortest cycle in $<_{\sigma,\rho}$. Then C is of the form $op_0, op_1, \ldots, op_{m-1}$ for some even $m \geq 2$, where, for all i, $1 \leq i \leq m/2$,

[5]If $<_{\sigma,\rho}$ is applied to a subset of $ops(\sigma)$, first identify the pairs from $ops(\sigma)$ that are in the relation, and then project onto the subset.

- op_{2i-1} is a read that reads from write op_{2i-2}, and
- $op_{2i-1} <_\sigma op_{(2i) \bmod m}$.

I.e., the cycle consists of alternating reads and writes, with each read reading from the preceding write, and each write strictly following the preceding read. Note that read operations have odd indexes and write operations have even indexes.

For each i, $1 \leq i \leq m/2$, op_{2i-2} begins before op_{2i-1} ends (by definition of a reads-from function), and op_{2i-1} ends before $op_{(2i) \bmod m}$ begins (by definition of $<_\sigma$). Thus op_0 begins before op_0 begins, which is a contradiction. ■

The partial order $<_{\sigma,\rho}$ is more restrictive than $<_\sigma$ since $<_{\sigma,\rho}$ extends $<_\sigma$ by taking into consideration reads-from relationships between reads and writes.

Our next condition, which we call MWRegRF where the RF stands for "reads from", strengthens MWRegWeak by requiring that the linearization for each read be (σ, ρ)-consistent, not just σ-consistent.

Definition 9 (MWRegRF) *A schedule* σ satisfies MWRegRF *if there is a reads-from function* ρ *on* σ *such that for every read operation* r *in* $ops(\sigma)$*, there exists a legal linearization of* $(writes(\sigma) \cup \{r\}, <_{\sigma,\rho})$*. A shared register satisfies MWRegRF if all schedules on it satisfy MWRegRF.*

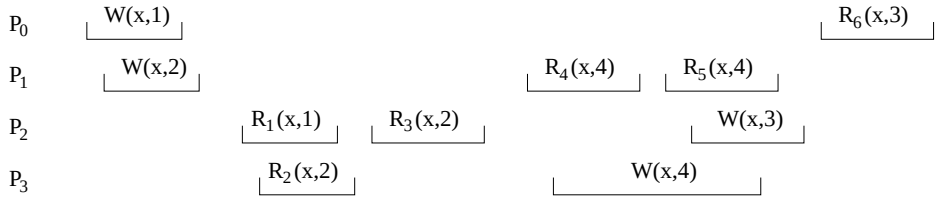

Figure 8. Schedule that satisfies MWRegRF.

The schedules shown in Figures 6 and 7 do not satisfy MWRegRF: In both schedules, $W(x,4) <_{\sigma,\rho} W(x,3)$, since R_4 reads from $W(x,4)$ and $R_4 <_\sigma W(x,3)$; since $W(x,3) <_\sigma R_6$, in any legal linearization consistent with both the order of non-overlapping operations and any reads-from function, R_6 must return 3 instead of 4. However, the schedule shown in Figure 8 satisfies MWRegRF; possible linearizations for the reads are given below.

$$
\begin{array}{ll}
R_1\text{:} & W(x,2), W(x,1), R_1(x,1), W(x,4), W(x,3) \\
R_2\text{:} & W(x,1), W(x,2), R_2(x,2), W(x,4), W(x,3) \\
R_3\text{:} & W(x,1), W(x,2), R_3(x,2), W(x,4), W(x,3) \\
R_4\text{:} & W(x,1), W(x,2), W(x,4), R_4(x,4), W(x,3) \\
R_5\text{:} & W(x,1), W(x,2), W(x,4), R_5(x,4), W(x,3) \\
R_6\text{:} & W(x,1), W(x,2), W(x,4), W(x,3), R_6(x,3)
\end{array}
$$

The schedule in Figure 8 does not satisfy MWRegWO for the same reason that the schedule in Figure 6 does not; thus MWRegRF and MWRegWO are incomparable in terms of strength.

To show that Alg_WB implements MWRegRF, we first look at how the write-back building block affects the relationship between the operations and their timestamps. Since we use the same timestamp in Alg_WB as in Alg_None, Lemma 2 still holds. We define the function ρ for Alg_WB as we did in Section 4.1 for Alg_None. Lemma 3 still holds and thus ρ is a reads-from function for Alg_WB.

Lemma 8 *Consider any execution of Alg_WB and let σ be its schedule.*

(a) For every read operation r and every write operation w in $ops(\sigma)$, if $r <_\sigma w$, then $ts(r) < ts(w)$.

(b) For every pair of read operations r_1 and r_2 in $ops(\sigma)$, if $r_1 <_\sigma r_2$, then $ts(r_1) \leq ts(r_2)$.

Proof. The key is the non-empty intersection of quorums used in queries by read and write operations and quorums used in updates by write operations. Essentially the same argument as in the proof of Lemma 2(b) is used to prove (a), and essentially the same argument as in the proof of Lemma 2(a) is used to prove (b). ■

Theorem 9 *Algorithm Alg_WB implements MWRegRF.*

Proof. Consider any schedule σ resulting from an execution of Alg_WB, let ρ be the reads-from function for σ just defined, and let r be any read in $ops(\sigma)$. We construct a total order L_r on $writes(\sigma) \cup \{r\}$ as follows. Divide $writes(\sigma)$ into two groups, $G_1 = \{w | ts(w) \leq ts(r)\}$ and $G_2 = \{w | ts(w) > ts(r)\}$. In the total order, all operations in G_1 precede r, and r precedes all operations in G_2. The operations in G_1 are ordered by their timestamps, breaking ties arbitrarily with the exception that $\rho(r)$ (the write operation from which r reads) is ordered at the end. The operations in G_2 are ordered by their timestamps, breaking ties arbitrarily. By construction, L_r is legal; Lemmas 2 and 8 and the construction ensure that it is (σ, ρ)-consistent. Thus Alg_WB implements a MWRegRF shared register. ■

4.4 MWRegNI: Building Block LC

The use of the LC building block, by which each reader keeps a local cache with the latest timestamp of any returned value, prevents a particular reader from returning an older value after it has already returned a newer value. Since the LC and WB building blocks both prevent some kinds of new-old inversions, it is worth explicitly comparing them. With LC, reads by the same client are always handled consistently, even with respect to concurrent writes that have the same timestamp; however, reads by different clients can experience new-old inversions even with respect to writes with different timestamps. In contrast, when WB is used, writes with different timestamps are handled consistently by all clients, but writes with the same timestamp can be viewed inconsistently even by a single client.

The next consistency condition, which we call MWRegNI where NI stands for "no inversion", captures the property ensured by the use of the LC building block.

Definition 10 (MWRegNI) *A schedule σ* satisfies MWRegNI *if there is a reads-from function ρ on σ such that the following is true for every process p_i. Let W_i be $\{\rho(r) : r \in reads(\sigma|i)\}$. Then there exists a legal linearization of $(W_i \cup reads(\sigma|i), <_\sigma)$. A shared register satisfies MWRegNI if all schedules on it satisfy MWRegNI.*

The idea is that, for each process p_i, there must be a fixed ordering of a certain set of writes, W_i, and all the reads by p_i. W_i consists of all writes that are read from by at least one read by p_i. The ordering must be legal and σ-consistent.

The schedules in Figures 6, 7, and 8 do not satisfy MWRegNI: In Figures 6 and 8, reads R_1 and R_3 by process p_2 are problematic, while in Figure 7, reads R_4 and R_5 by process p_1 are problematic. However, the schedule shown in Figure 9 satisfies MWRegNI, as the following per-process linearizations witness:

$$
\begin{array}{ll}
p_0\text{:} & W(x,4), R_6(x,4) \\
p_1\text{:} & W(x,4), R_4(x,4), R_5(x,4) \\
p_2\text{:} & W(x,1), R_1(x,1), R_3(x,1) \\
p_3\text{:} & W(x,2), R_2(x,2)
\end{array}
$$

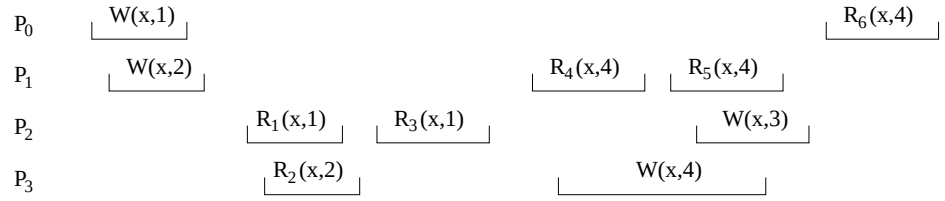

Figure 9. Schedule that satisfies MWRegNI.

The schedule in Figure 9 does not satisfy MWRegWO or MWRegRF for the same reasons that the schedule in Figure 6 does not.

The next lemma shows how the local cache affects the relationships between two operations and their timestamp order. Since we use the same timestamp in Alg_LC as in Alg_None, Lemma 2 still holds. We define the function ρ for Alg_LC as for Alg_None; even though the value returned by a read might have been obtained by $GetValue()$ from the client's local cache, it is still true that it was contained in an UPDATE message previously received from a server. Lemma 3 still holds and thus ρ is a reads-from function for Alg_LC.

Lemma 10 *Consider any execution of Alg_LC and let σ be its schedule and ρ its reads-from function.*

(a) For every read operation r and every write operation w by the same process in $ops(\sigma)$, if $r <_\sigma w$, then $ts(r) < ts(w)$.

(b) For every pair of read operations r_1 and r_2 by the same process in $ops(\sigma)$, if $r_1 <_\sigma r_2$, then $ts(r_1) \leq ts(r_2)$.

(c) For every pair of read operations r_1 and r_2 by the same process in $ops(\sigma)$, if $ts(r_1) = ts(r_2)$, then $\rho(r_1) = \rho(r_2)$.

Proof. Parts (a) and (b) follow from the code.

(c) Suppose $ts(r_1) = ts(r_2)$; call this value t. When r_1 finishes, the timestamp in p_i's cache is t. When r_2 executes, the largest timestamp among all the views received in Line 2 is at most t, otherwise, t would not be the timestamp of r_2. Also, the timestamp in p_i's local cache during the execution of r_2 is at most t, otherwise t would not be the timestamp of r_2. Since the timestamp in p_i's local cache never decreases, it must equal t. Thus r_2 uses the data in the local cache to determine its return value, and $\rho(r_2) = \rho(r_1)$. ■

Theorem 11 *Algorithm Alg_LC implements MWRegNI.*

Proof. Consider any execution of Alg_LC and let σ be its schedule and ρ its reads-from function as defined just above. Let p_i be any process and let W_i be $\{\rho(r) : r \in reads(\sigma|i)\}$. By Lemma 10 (c), there is at most one write in W_i with a given timestamp. None of the reads in $reads(\sigma|i)$ overlap each other. Construct a total order L_i on $W_i \cup \{reads(\sigma|i)\}$ as follows. Order all the writes in W_i according to their timestamp. For each write w in W_i, order immediately after w, in a σ-consistent total order, every read r in $reads(\sigma|i)$ such that $\rho(r) = w$.

By construction, L_i is legal. We now show L_i is σ-consistent. By Lemma 2 (b), the relative order of two non-overlapping writes is correct. By Lemma 10 (b), the relative order of two non-overlapping reads is correct: the timestamp of the earlier read, r_1, is at most the timestamp of the later read, r_2, so either $\rho(r_1) = \rho(r_2)$ or $\rho(r_1)$ is ordered in L_i before $\rho(r_2)$. Suppose $w <_\sigma r$. By Lemma 2 (a), $ts(w) \leq ts(r)$. Thus $\rho(r)$ equals w or a write that appears later than w in L_i, and so r is placed after w in L_i. Suppose $r <_\sigma w$. By Lemma 10 (a), $ts(r) < ts(w)$. Thus $\rho(r)$ is a write that precedes w in L_i, and so r is placed before w in L_i. ■

4.5 Combining the Building Blocks

If the building blocks are combined, we obtain stronger conditions than if they are used separately.

When ID and LC are combined, the resulting algorithm Alg_ID_LC implements MWRegWO ∩ MWRegNI. The schedule in Figure 10 satisfies MWRegWO ∩ MWRegNI, but not MWRegRF for the same reason that the schedule in Figure 6 does not.

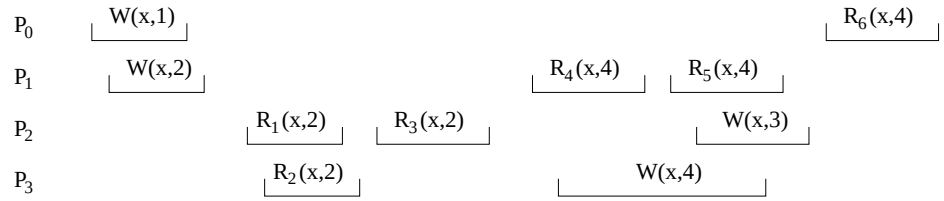

Figure 10. Schedule that satisfies MWRegWO ∩ MWRegNI.

When WB and LC are combined, the resulting algorithm Alg_WB_LC implements MWRegRF ∩ MWRegNI. The schedule in Figure 11 satisfies MWRegRF ∩ MWRegNI, but not MWRegWO for the same reason that the schedule in Figure 6 does not.

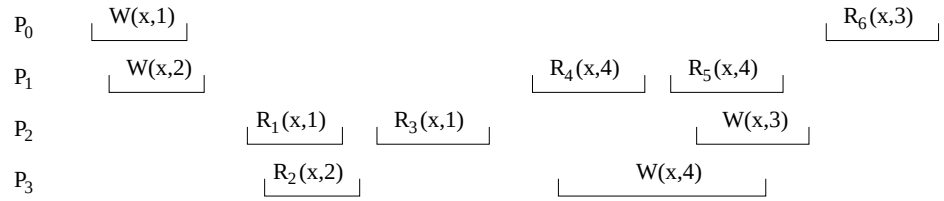

Figure 11. Schedule that satisfies MWRegRF ∩ MWRegNI

When ID and WB are combined, the resulting algorithm Alg_ID_WB satisfies Atomicity (see Appendix for definition) as we prove next. The schedule in Figure 12 satisfies Atomicity.

Since we use the timestamp of Alg_None as the first element of the timestamp in Alg_ID_WB, Lemma 2 still holds. We define the function ρ for Alg_ID_WB as for Alg_None. Lemma 3 still holds and thus ρ is a reads-from function for Alg_ID_WB. In addition, Lemmas 5 and 8 continue to hold.

Theorem 12 *Alg_ID_WB implements Atomicity.*

Proof. Consider any execution of Alg_ID_WB and let σ be its schedule and ρ the reads-from function defined just above. We will construct a legal linearization L of $(ops(\sigma), <_\sigma)$. Define L as follows. Order all the writes by timestamp. For all reads with timestamp T, order them after the write with timestamp T (which is unique by Lemma 5) and before the next write in timestamp order; order these reads among themselves according to their invocation order. L is legal by construction. We now show that L is σ-consistent.

Case 1: Consider two writes w_1 and w_2 where $w_1 <_\sigma w_2$. From Lemma 2(b), $ts(w_1) < ts(w_2)$ and thus w_1 is ordered before w_2 in L.

Case 2: Consider a read r and a write w where $r <_\sigma w$. By Lemma 8(a), $ts(r) < ts(w)$. Let $w' = \rho(r)$. Since $ts(w') = ts(r)$, w' is ordered before w in L. Since r is ordered after w' but before the next write in timestamp order, r is ordered before w in L.

Case 3: Consider a write w and a read r where $w <_\sigma r$. By Lemma 2(a), $ts(w) \leq ts(r)$. Let $w' = \rho(r)$. Since $ts(w') = ts(r)$, it follows that $ts(w) \leq ts(w')$. If $ts(w) = ts(w')$, then $w = w'$ by Lemma 5, and the construction of L ensures that w is ordered before r. If $ts(w) < ts(w')$, then in L, w is ordered before w', which is ordered before r.

Case 4: Consider two reads r_1 and r_2 where $r_1 <_\sigma r_2$. By Lemma 8(b), $ts(r_1) \leq ts(r_2)$. If $ts(r_1) = ts(r_2)$, then $\rho(r_1) = \rho(r_2)$ and the construction of L ensures that r_1 is ordered before r_2. If $ts(r_1) < ts(r_2)$, then $ts(\rho(r_1)) < ts(\rho(r_2))$; the construction of L ensures that r_1 is ordered before r_2. ■

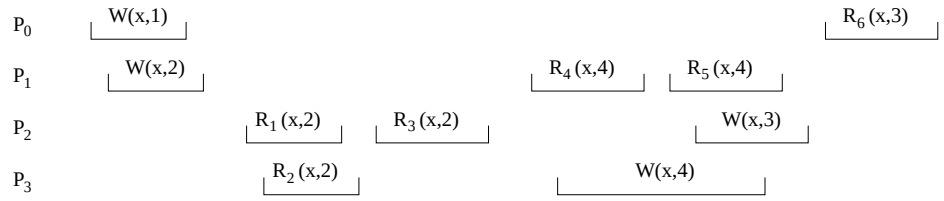

Figure 12. Schedule that satisfies Atomicity

5 Properties of the Definitions

This section explores some of the properties of our new consistency conditions: how they compare to the original definition when there is just one writer, how they can be extended to the multi-register situation, and how they relate to some previously known consistency conditions.

5.1 Relation to the Original Single-Writer Definition

The following lemma states the relationship between our proposed definitions and SWReg, the single-writer definition of Lamport.

Lemma 13 *Suppose there is only a single writer. Then MWRegWeak and MWRegWO are equivalent to SWReg while MWRegRF and MWRegNI are stronger than SWReg.*

Proof. If there is only a single writer, then the definition of MWRegWeak is the same as that of SWReg.

The definition of MWRegWO implies SWReg for a single writer. On the other hand, for any schedule σ that satisfies SWReg, for every read there is a total ordering of all the writes and itself that is legal and σ-consistent. Since there is only one writer, σ-consistency implies that all the writes appear in the same order in each read's total order. Thus for any two reads, the writes that are relevant to both the reads appear in the same order in the linearizations for the reads. Therefore SWReg $\subseteq$ MWRegWO. Thus SWReg $=$ MWRegWO.

The schedule in Figure 13 has one reader and one writer, satisfies SWReg, but satisfies neither MWRegRF nor MWRegNI. By inspection, the definitions of MWRegRF and MWRegNI both imply the definition of SWReg when only one writer is considered, ■

It follows that when there is only one writer, MWRegWO $\cap$ MWRegNI and MWRegRF $\cap$ MWRegNI are also stronger than SWReg.

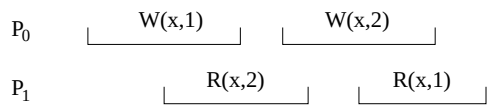

Figure 13. Single-writer schedule that satisfies SWReg but neither MWRegRF nor MWRegNI.

5.2 Locality

A consistency condition C is *local* whenever a schedule σ satisfies C if and only if the projection of σ on each shared variable satisfies C. Locality is a desirable property of consistency conditions: as mentioned in Herlihy and Wing [13], locality enhances modularity and concurrency. In particular, if a consistency condition is local, then each shared variable in a system can be implemented individually, and the composition of multiple such variable implementations produces a system that satisfies the condition.

The specifications given in Section 4 of the four new consistency conditions apply to the multi-register case after making some small alterations to the constituent definitions. In particular, a total order of operations on a set of registers is *legal* if, for each register, the restriction of the total order to that register is legal. No changes are needed to the definitions of schedule or relevant writes. The definition of a reads-from function ρ becomes the following, with the modifications in bold:

Definition 11 *Given a schedule σ, consider a function ρ from $reads(\sigma)$ to $writes(\sigma)$. ρ is a* reads-from *function if for each read r, r* **and** *$\rho(r)$* **operate on the same register** *x, the value returned by r is the same as the value written by $\rho(r)$, $\rho(r)$ is relevant to r, and there is no write w in* **writes**$(\sigma|x)$ *such that $\rho(r) <_\sigma w <_\sigma r$.*

If $w = \rho(r)$, we say that r *reads from* w *with respect to* ρ; when ρ is understood from context, we simply say that r *reads from* $\rho(r)$.

Theorem 14 *MWRegWeak is local.*

Proof. For any schedule σ, if σ satisfies MWRegWeak, then clearly $\sigma|x$ satisfies MWRegWeak for every register x.

For the other direction, consider any schedule σ such that $\sigma|x$ satisfies MWRegWeak for every register x. We show that σ satisfies MWRegWeak. Consider any read operation r in $ops(\sigma)$ on some shared register x. We construct a total order L_r on $writes(\sigma) \cup \{r\}$ as follows. Let L_r^x be a legal linearization of $(writes(\sigma|x) \cup \{r\}, <_{\sigma|x})$ which exists since $\sigma|x$ satisfies MWRegWeak. Define the relation $<_r$ on $writes(\sigma) \cup \{r\}$ to be the transitive closure of the union of $<_\sigma$ and L_r^x; $<_r$ is a partial order since L_r^x is consistent with $<_\sigma$. Let L_r be any linearization of $<_r$. By construction, L_r is σ-consistent. Since L_r^x is legal and L_r contains no additional operations on x, L_r is also legal. ■

The next three proofs use the technique of Herlihy and Wing [13] for proving the locality of linearizability. The only difference between the proofs is the construction of a partial order, which is then extended into a total order.

Theorem 15 *MWRegWO is local.*

Proof. Let σ be any schedule that satisfies MWRegWO. Without loss of generality, assume that for every write w to every register x, there is at least one read of x that ends after w begins. It follows from the definition that, for each register x, $\sigma|x$ satisfies MWRegWO.

For the other direction, consider any schedule σ such that, for each register x, $\sigma|x$ satisfies MWRegWO. We must show that σ satisfies MWRegWO. For each read r in $ops(\sigma)$, we first define a partial order $<_r$ on $writes(\sigma) \cup \{r\}$, and then let L_r be a linearization of $<_r$. We then show that these linearizations satisfy the definition of MWRegWO for σ.

Fix a read r of register x in $ops(\sigma)$. Let L_r^x be a linearization of $(writes(\sigma|x) \cup \{r\}, <_{\sigma|x})$ guaranteed by the assumption that $\sigma|x$ satisfies MWRegWO. For each register $y \neq x$, let r_y be the first read of y that starts after r ends. If there is no such read, then let r_y be the last read of y. The important point is that every write to y that starts before r ends also starts before r_y ends. Let $L_{r_y}^y$ be a linearization of $(writes(\sigma|y) \cup \{r_y\}, <_{\sigma|y})$ guaranteed by the assumption that $\sigma|y$ satisfies MWRegWO. Define the relation $<_r$ on $writes(\sigma) \cup \{r\}$ to be the transitive closure of the union of $<_\sigma$, L_r^x, and all the $L_{r_y}^y$ linearizations for each $y \neq x$.

Claim: $<_r$ is a partial order.
Proof of claim: Suppose in contradiction it is not, and let $C = op_0, op_1, \ldots, op_{m-1}, op_0$ be a shortest cycle in $<_r$. Since C is shortest, the edges alternate between $<_\sigma$ and $L_{r'}^{x'}$ for some r' (for possibly different x' registers) and thus m is even. Without loss of generality, let $op_0 <_\sigma op_1$ and $op_1 <_{L_{r'}^{x'}} op_{2 \bmod m}$. Note that $L_{r'}^{x'}$ is $(\sigma|x')$-consistent, by assumption that $\sigma|x'$ satisfies MWRegWO.

Suppose $m = 2$. Then $op_{2 \bmod m} = op_0$ and we have that op_0 ends before op_1 begins, and op_1 begins before op_0 ends, contradiction. Thus m must be at least 4.

Then in σ, op_0 ends before op_1 begins, op_1 begins before op_2 ends, and op_2 ends before op_3 begins. But then op_0 ends before op_3 begins, i.e., $op_0 <_\sigma op_3$, and we can get a shorter cycle by deleting op_1 and op_2 from C, a contradiction. *End of proof of claim.*

Let L_r be a linearization of $<_r$ in which incomparable operations are ordered in a deterministic way (say, in order of the writers' identifiers). Since r is the only read in the set of operations over which L_r is defined, the legality of L_r follows from the legality of L_r^x. L_r is σ-consistent because $<_r$ includes $<_\sigma$.

We now show that the linearizations agree on the relevant writes. Consider any two reads r_1 and r_2 in $ops(\sigma)$. Let w_1 and w_2 be any two writes in *rel-writes*$(\sigma, r_1) \cap$ *rel-writes*(σ, r_2).

Suppose w_1 and w_2 both write to the same register, say y. By construction, L_{r_1} includes $L_{r_1'}^y$, where r_1' is some read of y such that every write to y that starts before r_1 ends also starts before r_1' ends. Similarly, L_{r_2} includes $L_{r_2'}^y$, where r_2' is some read of y such that every write to y that starts before r_2 ends also starts before r_2' ends. By the definition of MWRegWO, $L_{r_1'}^y$ and $L_{r_2'}^y$ agree on the order of all writes to y that are relevant to both r_1' and r_2'. Since w_1 and w_2 are relevant to both r_1' and r_2', L_{r_1} and L_{r_2} agree on the order of w_1 and w_2.

Suppose w_1 and w_2 write to different registers. If w_1 and w_2 do not overlap, then L_{r_1} and L_{r_2} agree on the order of w_1 and w_2 because they are both σ-consistent. If w_1 and w_2 overlap, then agreement follows because of the deterministic method used in the linearizations to resolve the ordering of incomparable operations. ■

Theorem 16 *MWRegRF is local.*

Proof. Let σ be any schedule that satisfies MWRegRF. To show that $\sigma|x$ satisfies MWRegRF for each register x, set the reads-from function for $\sigma|x$ to be the projection onto operations on x of the reads-from function for σ. The result follows from the definition.

For the other direction, consider a schedule σ such that, for each shared register x, $\sigma|x$ satisfies MWRegRF. In other words, for each x, there is a reads-from function ρ^x on $\sigma|x$ such that the following holds: For each read r in $\sigma|x$, there exists a legal linearization L_r^x of $(writes(\sigma|x) \cup \{r\}, <_{\sigma|x,\rho^x})$. We must show that the original schedule σ satisfies MWRegRF. That is, we must show that there exists a reads-from function ρ on σ such that the following holds: For each read r in σ, there exists a legal linearization L_r of $(writes(\sigma) \cup \{r\}, <_{\sigma,\rho})$.

First, define ρ, a reads-from function for σ, as $\rho(r) = \rho^x(r)$, where x is the register that r accesses.

Now consider any read r in $ops(\sigma)$ on some register x. For each register $y \neq x$, choose any read r_y on y and consider the linearization $L^y_{r_y}$. Define the relation $<_r$ on $writes(\sigma) \cup \{r\}$ to be the transitive closure of the union of $<_{\sigma,\rho}$, L^x_r, and all the $L^y_{r_y}$ linearizations.

Claim: $<_r$ is a partial order.
Proof of claim: Suppose in contradiction $<_r$ contains a cycle and let $C = op_0, op_1, \ldots, op_{m-1}, op_0$ be a shortest cycle. Since C is a shortest cycle, it consists of alternating $<_{\sigma,\rho}$ edges and $L^{x'}_{r'}$ edges, and thus m is even. In fact, since ρ only relates two operations on the same register, and since all operations on the same register, say x', are ordered by $L^{x'}_{r'}$, the $<_{\sigma,\rho}$ edges are actually $<_\sigma$ edges and they relate operations on different registers. The same argument as in the proof of Theorem 15 shows that C can be shortened, which is a contradiction. *End of proof of claim.*

Let L_r be any linearization of the partial order $<_r$. Since L_r contains L^x_r, which is legal, and no additional operations on x are considered, L_r is also legal. Finally, L_r is (σ, ρ)-consistent since $<_r$ contains $<_{\sigma,\rho}$. ■

Theorem 17 *MWRegNI is local.*

Proof. Let σ be any schedule and ρ be any reads-from function for σ that satisfy MWRegNI. To show that $\sigma|x$ satisfies MWRegNI for each register x, use the reads-from function for $\sigma|x$ that is the projection of ρ onto operations on x. The result follows from the definition.

For the other direction, consider any schedule σ such that, for each register x, $\sigma|x$ satisfies MWRegNI. That is, for each register x, there is a reads-from function ρ^x from $reads(\sigma|x)$ to $writes(\sigma|x)$ such that for each process p_i, there exists a legal linearization L^x_i of $(W^x_i \cup reads(\sigma|x|i), <_{\sigma|x})$, where $W^x_i = \{\rho^x(r) : r \in reads(\sigma|x|i\}$ (the set of all writes to x that are read from by at least one read by p_i).

We must show that σ satisfies MWRegNI. We will define a reads-from function ρ from $reads(\sigma)$ to $writes(\sigma)$ such that for each process p_i, there exists a legal linearization L_i of $(W_i \cup reads(\sigma|i), <_\sigma)$, where $W_i = \{\rho(r) : r \in reads(\sigma|i)\}$ (the set of all writes that are read from by at least one read by p_i).

Let ρ be defined for each r in $reads(\sigma)$ as $\rho(r) = \rho^x(r)$, where x is the register that r accesses. Fix a process p_i. Define the relation $<_i$ on $W_i \cup reads(\sigma|i)$ to be the transitive closure of the union of $<_\sigma$ and all the L^x_i, where x ranges over all the registers. We will show that $<_i$ is a partial order, then take any linearization of it as L_i and show that L_i is legal and σ-consistent.

Claim: $<_i$ is a partial order.
Proof of claim: Suppose in contradiction it is not, and let $C = op_0, op_2, \ldots, op_{m-1}, op_0$ be a shortest cycle in $<_i$. Since C is shortest, the edges alternate between $<_\sigma$ and L^x_i (for possibly different registers x), and thus m is even. The same argument as in the proof of Theorem 15 shows that C can be shortened, which is a contradiction. *End of proof of claim.*

Let L_i be any linearization of $<_i$. It is legal because each constituent L^x_i is legal. It is σ-consistent because $<_i$ includes $<_\sigma$. ■

Vitenberg and Friedman [31] prove that any consistency condition that is the intersection of two local conditions is also local. Thus, since MWRegWO, MWRegRF, and MWRegNI are each local, the intersection of any two of them is also local.

5.3 Comparison

In this section, we first compare our proposed consistency conditions to each other; then we relate our definitions to other existing consistency conditions.

5.3.1 Comparison Between Proposed Definitions

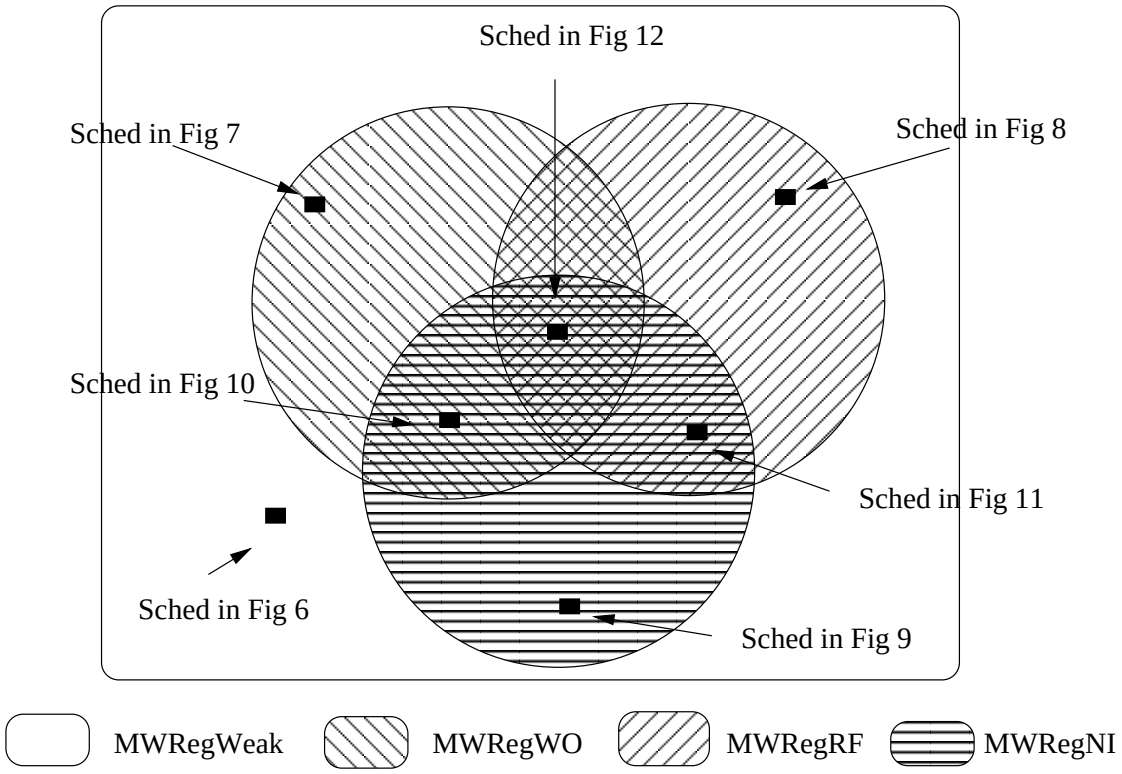

Figure 14. Venn diagram of the proposed definitions

The Venn diagram in Figure 14 summarizes the relationships between our proposed definitions. The diagram describes sets of schedules. The outer rectangle indicates all schedules that satisfy MWRegWeak. The left circle represents all schedules that satisfy MWRegWO, the right circle those that satisfy MWRegRF, and the bottom circle those that satisfy MWRegNI. The small solid squares give the locations of the seven example schedules discussed in Section 4.

Lemma 18 *The relationships between the conditions are as indicated in Figure 14. That is,*

(a) MWRegWO, MWRegRF, and MWRegNI are proper subsets of MWRegWeak.

(b) Each pair of MWRegWO, MWRegRF, and MWRegNI have a non-empty intersection but are incomparable.

(c) Furthermore, Atomicity is contained in the intersection of MWRegWO, MWRegRF,and MWRegNI.

Proof. (a) The schedule in Figure 6 satisfies MWRegWeak, but does not satisfy MWRegWO, MWRegRF, or MWRegNI. Therefore, to complete the proof, it suffices to show that each of the three conditions of interest is a subset of MWRegWeak.

MWRegWO $\subseteq$ MWRegWeak since the only difference between the definition of MWRegWO and MWRegWeak is that the former puts an additional constraint on the required per-read linearizations. The same argument shows that MWRegRF $\subseteq$ MWRegWeak.

To show that MWRegNI $\subseteq$ MWRegWeak, consider any schedule σ that satisfies MWRegNI. Consider any read r in $ops(\sigma)$. From the definition of MWRegNI, there exists a reads-from function ρ on σ. Let $w = \rho(r)$.

Case 1: $w <_\sigma r$: Define the following total order L_r on $writes(\sigma) \cup \{r\}$. First, put all the writes that precede or overlap w in any σ-consistent total order. Put w next in the total order. Then put r next in the total order. Finally, follow r with any σ-consistent total order on all the remaining writes.

By construction L_r is legal. To show that L_r is σ-consistent, we check that r is ordered properly with respect to each non-overlapping write w' other than w. Suppose $w' <_\sigma r$. Then w' either precedes or overlaps w by the definition of a reads-from function, and thus w' appears before r in L_r. Suppose $r <_\sigma w'$. Since $w <_\sigma r$, w' follows w and appears after r in L_r.

Case 2: w and r overlap in σ. Define the following total order L_r on $writes(\sigma) \cup \{r\}$. First, put all the writes that precede r in any σ-consistent total order. Put w next in the total order. Then put r next in the total order. Finally, follow r with any σ-consistent total order on all the remaining writes. L_r is legal and σ-consistent by construction.

(b) The schedule in Figure 10 shows that MWRegWO and MWRegNI have a nonempty intersection, while the schedules in Figures 7 and 9 show that they are incomparable. The schedule in Figure 11 shows that MWRegRF and MWRegNI have a nonempty intersection, while the schedules in Figures 8 and 9 show that they are incomparable. The schedule in Figure 12 shows that MWRegWO and MWRegRF have a nonempty intersection, while the schedules in Figures 7 and 8 show that they are incomparable.

(c) The definition of Atomicity appears in the Appendix. Let σ be any schedule satisfying Atomicity and let L be the linearization of $(ops(\sigma), <_\sigma)$. To show that σ also satisfies MWRegWO, for each read r, let L_r be the restriction of L to $writes(\sigma) \cup \{r\}$. To show that σ also satisfies MWRegRF, let ρ be the reads-from function such that each read reads from the write that most recently precedes it in L, and, for each read r, let L_r be the same as for the MWRegWO argument. To show that σ also satisfies MWRegNI, use the same ρ as for the MWRegRF argument, and for each process p_i, let the projection of L onto $W_i \cup reads(\sigma|i)$, where W_i is the set of all writes that are read from by process p_i, be the desired linearization. ■

Recall that Theorem 4 shows that every execution of Algorithm Alg_None satisfies MWRegWeak. But it does not show that every schedule satisfying MWRegWeak can be generated by Algorithm Alg_None. Perhaps Algorithm Alg_None actually guarantees a stronger condition than MWRegWeak, i.e., perhaps the set of schedules that can be generated by Algorithm Alg_None is a proper subset of those satisfying MWRegWeak. In fact, Alg_ID cannot generate all MWRegWO schedules: Consider a schedule in which p_0 writes 1 and simultaneously p_1 writes 2, and then later p_2 reads 1. Although this schedule satisfies MWRegWO (and in fact, satisfies Atomicity), it cannot be generated by the algorithm, which would give priority to p_1's write, since p_1's ID is larger than p_0's.

We partially address the issue of characterizing exactly the schedules that each algorithm can generate by showing that Algorithm Alg_None can generate the schedule in Figure 6, which, as indicated in the Venn diagram, does not satisfy any of our other consistency conditions. Similarly, we can show that Algorithm Alg_ID, which implements MWRegWO, can generate the schedule in Figure 7, etc. For detailed constructions of the schedules by the algorithms, see Shao's thesis [28]. Thus, our results show that Alg_ID, Alg_WB, and Alg_LC generate incomparable sets

of schedules.

5.3.2 Comparison with Existing Consistency Conditions

The relationships between our new consistency conditions and the known consistency conditions Sequential Consistency, Goodman's Processor Consistency (PCG), Causal Consistency, PRAM and Coherence are shown as a partial order in Figure 15. Definitions of these additional conditions are given in Appendix A. The relationships on the right side come from Ahamad et al. [1, 2] and Vitenberg and Friedman [31]. This figure indicates that all of our new conditions are incomparable with the existing conditions listed above; these relationships are proved in the next theorem.

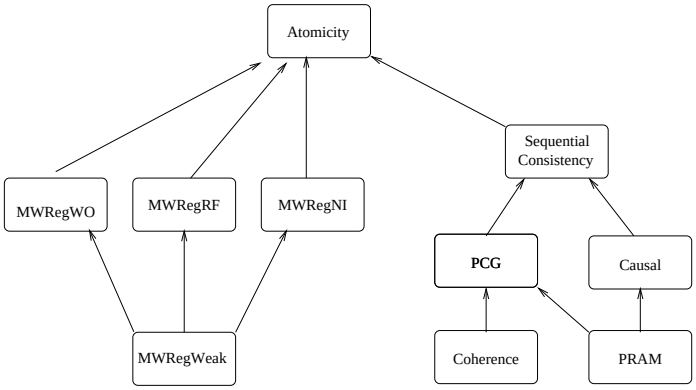

Figure 15. Partial order among some consistency conditions

Theorem 19 *Each of MWRegWO, MWRegRF, and MWRegNI is incomparable with each of Sequential Consistency, PCG, Causal Consistency, Coherence, and PRAM.*

Proof. We show that for each pair of conditions (C_1, C_2), where C_1 is one of our four new conditions and C_2 is one of the five previously proposed conditions, there is a schedule that satisfies C_1 but not C_2 and vice versa. For several of the pairs, we can use the same schedule.

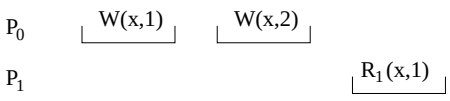

Figure 16. Schedule that is Sequentially Consistent but not MWRegWeak.

The schedule in Figure 16 satisfies Sequential Consistency, and thus also PCG, Causal Consistency, Coherence, and PRAM: the operations can be ordered $W(x,1), R_1(x,1), W(x,2)$. However, this schedule does not satisfy MWRegWeak, and thus does not satisfy MWRegWO, MWRegRF, or MWRegNI: the only legal linearization for R_1 is $W(x,1), R_1(x,1), W(x,2)$, but this violates σ-consistency.

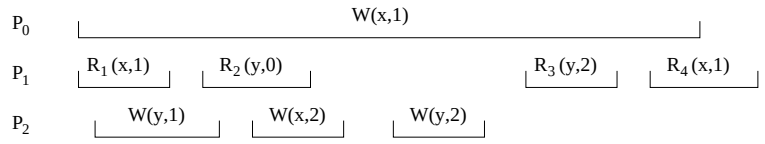

Figure 17. Schedule that satisfies MWRegWO, MWRegRF, and MWRegNI but not PRAM

The schedule in Figure 17 satisfies MWRegWO, MWRegRF, and MWRegNI, and thus also MWRegWeak, as we now show. The linearizations given next—one for each read—show that the schedule satisfies MWRegWO. The same linearizations also show that it satisfies MWRegRF.

$$\begin{array}{ll} R_1\text{:} & W(x,1), R_1(x,1), W(y,1), W(x,2), W(y,2) \\ R_2\text{:} & R_2(y,0), W(y,1), W(x,2), W(y,2), W(x,1) \\ R_3\text{:} & W(x,1), W(y,1), W(x,2), W(y,2), R_3(y,2) \\ R_4\text{:} & W(y,1), W(x,2), W(y,2), W(x,1), R_4(x,1) \end{array}$$

To show that the schedule satisfies MWRegNI, we just need a linearization for p_1, since the other processes never read. We can use $W(x,1), R_1(x,1), R_2(y,0), W(y,2), R_3(y,2), R_4(x,1)$.

However, the schedule in Figure 17 does not satisfy PRAM, and thus does not satisfy Causal Consistency or Sequential Consistency: in order to satisfy legality for the first three reads of p_1 as well as program order, the linearization for p_1 must be $W(x,1), R_1(x,1), R_2(y,0), W(y,1), W(x,2), W(y,2), R_3(y,2), R_4(x,1)$. But the return value of R_4 violates legality, as it should be 2 instead of 1.

The schedule in Figure 7 satisfies MWRegWO (and thus also MWRegWeak) as discussed earlier. However, it does not satisfy Coherence, and thus it also does not satisfy PCG or Sequential Consistency: To be legal, $W(x,2)$ must precede $R_2(x,2)$, and $W(x,4)$ must precede $R_4(x,4)$ in the linearization for x. To respect the per-process orders, though, $R_2(x,2)$ must precede $W(x,4)$, and $R_4(x,4)$ must precede $R_5(x,2)$. But then it is not legal for R_5 to return 2.

Finally, the schedule in Figure 18 satisfies MWRegRF and MWRegNI (and thus also MWRegWeak). It satisfies MWRegRF with the following linearizations for the reads:

$$\begin{array}{ll} R(x,1)\text{:} & W(x,0), W(x,1), R(x,1) \\ R(x,0)\text{:} & W(x,1), W(x,0), R(x,0) \end{array}$$

It satisfies MWRegNI with the following per-process linearizations:

$$\begin{array}{ll} p_1\text{:} & W(x,1), R(x,1) \\ p_2\text{:} & W(x,0), R(x,0) \end{array}$$

However, this schedule does not satisfy Coherence, and thus does not satisfy PCG or Sequential Consistency, since p_0 must view $W(x,1)$ after $W(x,0)$ whereas p_1 must view $W(x,0)$ after $W(x,1)$. ■

6 Mutual Exclusion Using Multi-Writer Regular Shared Registers

In this section, we use the mutual exclusion problem as a practical context to evaluate the strength of our new specifications for multi-writer regular shared registers. Specifically, we study the correctness of two well-known

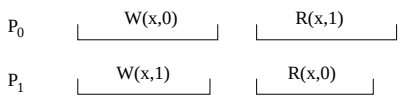

Figure 18. Schedule that satisfies MWRegRF and MWRegNI but not Coherence

1. Peterson's Algorithm for 2 Processes
Code for process p_i, $i \in \{0, 1\}$:

shared registers:
  $Flag[0..1]$: 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 Processes
Code for process p_i, $0 \leq i \leq n-1$:

shared registers:
  $Flag[0..n-1]$: *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

Figure 19. Algorithms for mutual exclusion

algorithms for mutual exclusion when the variables satisfy the consistency conditions we have proposed. The algorithms we examine are Peterson's algorithm for two processes [23] and Dijkstra's algorithm for n processes (as presented by Raynal [25]). The algorithms are shown in Figure 19.[6]

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.

A *run of an algorithm* (not to be confused with an execution on a shared register) is defined as an 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 algorithm run onto (the actions performed by) each individual process is consistent with the order of operations imposed by the local algorithm for that process, and
- the projection of the algorithm run onto the shared-memory operations on each register is a schedule on that register.

[6] Although Lamport's Bakery algorithm [15] and Peterson and Fischer's algorithm [24] are often studied in this context, they are not of interest to us here since these algorithms use only single-writer shared variables.

Table 1. Correctness of mutual exclusion algorithms using multi-writer regular registers.

	Peterson's Algorithm	Dijkstra's Algorithm
MWRegWeak	ME, EP, NL	ME
MWRegWO	ME, EP, NL	ME, EP
MWRegRF	ME, EP, NL	ME
MWRegNI	ME, EP, NL	ME
Atomicity	ME, EP, NL	ME, EP

(In this context, we consider a shared-register "request" to be the invocation of a request by a process, and a shared-register "response" to be the receipt of a response by a process. They are thus process actions, but can nevertheless be meaningfully projected onto the register also.) We say that an algorithm *runs under consistency condition* C if its projection onto the invocations and responses on the set of shared registers satisfies C.

We say that an algorithm A *solves mutual exclusion under consistency condition C* if, for each run of A 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):**[7] 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.[8]

We now examine the two mutual exclusion algorithms shown in Figure 19. 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 Atomic.

We first consider Peterson's algorithm for two processes [23]. This algorithm uses two single-writer shared variables, $Flag[0]$ and $Flag[1]$, and one multi-writer shared variable, $Turn$.

Theorem 20 *Peterson's Algorithm solves mutual exclusion (ME, EP, and NL) under MWRegWeak, and thus under all the proposed definitions.*

Proof. First we show that ME is satisfied. Suppose in contradiction there is a run in which ME is violated and let t be the time of the first violation. Without loss of generality, let $Turn = 0$ at time t. Since $Turn$ is only changed when a process leaves the critical section, and is only set to 0 by process p_1, $Turn$ remains 0 throughout the entry section of p_1 that most recently precedes t, as well as throughout the critical section of p_1 that includes time t. In this entry section of p_1, the last execution of Line 5 reads 0 from $Flag[0]$ since $Turn$ is 0. The latest preceding execution of Line 4 writes 1 to $Flag[1]$.

Case 1: p_0 reads 0 from $Turn$ in its last execution of Line 6 preceding time t. Thus the last execution of the wait construct in Line 6 reads 0 from $Flag[1]$. See Figure 20.

[7] 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.

[8] Although NL implies EP, we include both requirements, partly for historical reasons (e.g.,Higham and Kawash [14]) 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 MWRegWO and MWRegNI.

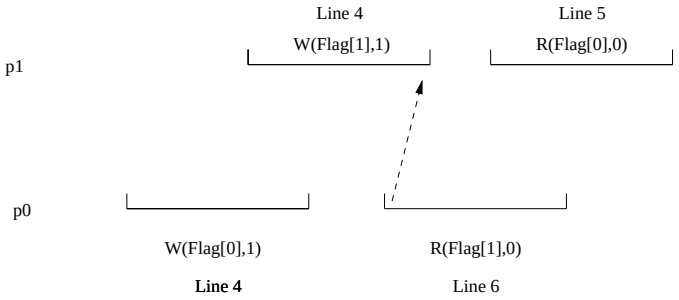

Figure 20. Figure for Case 1 of proof of Theorem 20.

By the definition of MWRegWeak, this read by p_0 of $Flag[1]$ must start before the end of the write of 1 by p_1 to $Flag[1]$ (in p_1's last execution of Line 4), because Flag[1] remains 1 after this execution of Line 4 until p_1 exits its critical section. But before p_0 executes Line 6, it executes Line 4 and writes 1 to $Flag[0]$. Thus it violates MWRegWeak for p_1's last execution of Line 5 to read 0 from $Flag[0]$, contradiction.

Case 2: p_0 reads 1 from $Turn$ in its last execution of Line 6 preceding time t. By MWRegWeak, this read must begin before the end of the previous write of 0 to $Turn$ by p_1 when p_1 was last in its exit section. See Figure 21.

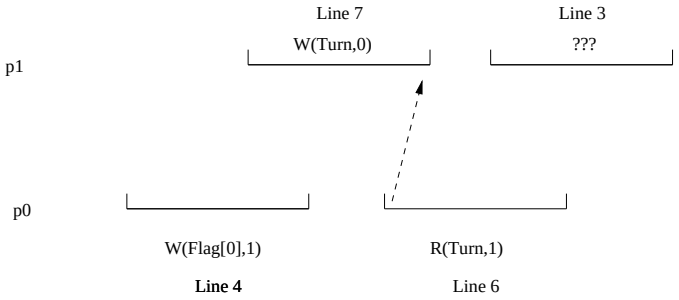

Figure 21. Figure for Case 2 of proof of Theorem 20.

Before p_0's last execution of Line 6, p_0 did its last execution of Line 4, and wrote 1 to $Flag[0]$. Thus, after the end of p_1's write of 0 to $Turn$ until (at least) time t, $Flag[0]$ remains 1 and $Turn$ remains 0. But then p_1 is stuck in a loop at Line 3, checking for $Flag[0] = 0$ or $Turn = 1$, and it cannot be that p_1 is in its critical section at time t, contradiction.

Now we show EP. If only one process, say p_i, is contending for the critical section, then there are no concurrent accesses to the shared variables and p_i will pass all the tests (Lines 3, 5, and 6) and enter the critical section. So in order to violate EP, both processes must be contending. Let t be the time after which both are stuck in their entry sections forever. Without loss of generality, let $Turn = 0$ at time t. $Turn$ is never changed subsequently, since only a process in its exit section writes to $Turn$ and no process is ever in its exit section after time t. But then p_0 passes all the tests (Lines 3, 5 and 6) and enters the critical section, contradiction.

Finally, we show NL. Without loss of generality, suppose for contradiction that p_0 is locked out starting at time t. Since EP holds, p_1 must enter the critical section infinitely often. During p_1's first exit after time t, p_1 sets $Turn$ to 0. Subsequently, $Turn$ is always 0, since only p_0 can change it to 1 and only when p_0 is in its exit section. But then

p_0 passes all the tests (Lines 3, 5, and 6) and enters the critical section, contradiction. ■

Dijkstra's algorithm for n processes uses n single-writer shared variables and one multi-writer shared variable (we follow the presentation of Raynal [25]). As shown next, under MWRegWO it behaves the same way as under atomicity: ME and EP are guaranteed, but not NL. Under MWRegRF and MWRegNI, however, only ME is guaranteed. It seems that a common σ-consistent total order on writes is necessary for Dijkstra's algorithm to satisfy EP.

Theorem 21 *The following are true of Dijkstra's algorithm:*
(a) ME is satisfied under MWRegWeak.
(b) EP is satisfied under MWRegWO.
(c) EP is satisfied under neither MWRegRF nor MWRegNI.

Proof. (a) Suppose in contradiction there is a run of Dijkstra's algorithm under MWRegWeak in which ME is violated. Let t be the earliest time in this run at which two processes, say p_i and p_j, are in the critical section simultaneously.

Let r_i be the last execution by p_i of Line 7 before time t; this is when p_i reads p_j's $Flag$ and sees that it is not *in-cs*. Let w_i be the last execution by p_i of Line 6 before r_i; this is when p_i sets its $Flag$ to *in-cs*. Similarly, let r_j be the last execution by p_j of Line 7 before time t (when p_j reads p_i's $Flag$ and sees that it is not *in-cs*) and let w_j be the last execution by p_j of Line 6 before r_j (when p_j sets its $Flag$ to *in-cs*).

By the definition of MWRegWeak, r_i begins before w_j ends, and by the algorithm, r_j begins after w_j ends. Thus r_i begins before r_j begins. But by a symmetric argument, r_j begins before r_i begins, hence a contradiction.

(b) Suppose in contradiction there is a run of Dijkstra's algorithm under MWRegWO in which EP is violated. Let t be the earliest time in this run after which no process changes its section and every process in its entry section has finished executing Line 2 at least once after entering its entry section. Thus there is a fixed nonempty set S of processes that are stuck in their entry section and the rest of the processes are in their remainder section.

First, we show that $Turn$ is written only finitely often. Suppose in contradiction there are infinitely many writes to $Turn$. After time t, all the writes to $Turn$ are by processes that are in S; as a result of such a write, $Turn$ holds the id of a process in S. Also, every read of $Flag[i]$ that starts after time t returns *idle* if and only if $i \notin S$. Thus eventually each read of $Turn$ returns the id of a process in S and thus the condition in Line 4 (whether $Flag[Turn] = idle$) is false, and $Turn$ is not written. This contradicts the assumption that $Turn$ is written infinitely often.

Let W_T be the (finite) set of all writes to $Turn$. Let r be any read of $Turn$ that starts after time t and after all writes in W_T have finished. By the definition of MWRegWO, there must be a total order on all the writes in the run, plus r, that is legal and consistent with the order of non-overlapping operations in the run. Let $w_1, w_2, \ldots, w_m, r$ be the projection of this total order onto $W_T \cup \{r\}$. Note that r is at the end and r returns the value written by w_m.

Let p_k be the process that performs w_m, implying that w_m writes k to $Turn$. We next show that p_k is in S. Suppose not. The process that executes r, obtaining k as the value of $Turn$, next reads $Flag[k]$ as being *idle*, and then writes $Turn$, contradicting the choice of r as occurring after the last write to $Turn$.

By the definition of MWRegWO, *every* read of $Turn$ occurring after the last write in W_T and after time t must order the writes in W_T the same way. If the read is by a process other than p_k, the process remains stuck in the while loop of Lines 3–5 forever, with its $Flag$ equal to *requesting*. If the read is by p_k, though, p_k falls through the while loop, finds the condition in Line 7 true, and enters the critical section. This is a contradiction to the assumption that EP is violated.

(c) We describe a run of the algorithm under MWRegRF that violates EP. Suppose that only p_1 and p_2 are active and they execute in lockstep. They both execute Line 2 and write *requesting* to their $Flag$ variables. Then they both execute Line 3 and read 0 from $Turn$. Then they both execute Line 4 and read *idle* from $Flag[0]$, after which they write their respective ids to $Turn$. Then they both execute Line 3 again and read $Turn$; p_1 obtains 2 while p_2

obtains 1. Since MWRegRF allows the total order for these reads to differ, p_1 views p_2's write to $Turn$ as occurring later than its own write to $Turn$, and vice versa for p_2. Then they both execute Line 4; p_1 reads $Flag[2]$ and obtains *requesting*, while p_2 reads $Flag[1]$ and obtains *requesting*. Then they repeat executing Lines 3 and 4 forever, and neither ever enters the critical section.

It can be shown that the same run also satisfies MWRegNI and thus under MWRegNI, EP can be violated. ■

7 Conclusion

The ever-growing popularity of data sharing on the web through applications such as Wikis and peer-to-peer file sharing indicates the usefulness of providing data that can be updated by multiple users but that does not require stringent consistency guarantees. Thus it is essential that weaker conditions, such as Lamport's regularity, be extended into the multi-writer model. While this extension is simple in the case of atomicity, it is more difficult and potentially ambiguous for the weaker condition of regularity.

This paper grew out of an attempt to extend Lamport's definition of regularity from the single-writer model [17] to the more general multi-writer model. We have shown that the extension is not trivial. While there exist various ways to extend the single-writer definition, the resulting definitions have different strengths. We started from a generic algorithm, which is a generalization of several existing protocols that use quorum systems to implement a read/write register. We then identified three building blocks from the algorithms. By applying different combinations of the building blocks, we obtained different algorithms. For each algorithm, we proposed a new consistency condition and proved that the corresponding algorithm implemented the definition. Out of the four new consistency conditions that we studied, two of them (MWRegWeak and MWRegWO) are especially good candidates for being a multi-writer version of regularity, since in the presence of only one writer, the condition reduces to the original definition.

The definitions form a lattice with respect to their strength, and the implementations have varying costs with respect to number of messages, size of messages, time delay, and local memory requirements. Taken together, the set of definitions point out the ambiguity of the informal notion of regularity and the algorithms suggest that different costs may be associated with different choices for disambiguating. Locality is a desirable property of consistency conditions, which enhances modularity and concurrency. We showed that all our proposed definitions satisfy locality. We have also analyzed the relationships between these consistency conditions and a number of other well-known consistency conditions.

Finally, we provided a practical context for our results by studying the correctness of two well-known algorithms for mutual exclusion when the variables satisfy our proposed consistency conditions. We found that Peterson's algorithm is fully correct under all the conditions, while Dijkstra's algorithm satisfies only some of the constraints of the mutual exclusion problem under any of the conditions.

Although the presentation in this paper did not take into account any kind of failures, crash failures of clients can be handled as in other papers (e.g., [13]). The definition of a schedule would be modified to allow some invocations (namely, the last one by a crashed client) to lack a matching response. Then the set of operations that appear in the definitions of the conditions, instead of being all the operations, would be all the completed operations and some subset of the uncompleted writes. In the algorithms, crash failures of some of the servers can be accommodated via the quorum system by having clients continue to send queries or updates to servers until hearing back from at least one quorum of servers. As long as crash failures have not disrupted all the quorums, the clients will be able to make progress. However, handling more malignant failures of the servers, especially Byzantine failures, as is done in the work of, e.g., Malkhi and Reiter [21] and Bazzi [7], is left for future work.

Our algorithms exhibit differences in cost, but we do not know if the differences are inherent. It would be interesting to show a complexity separation between our proposed conditions, i.e., to prove some lower bound on the cost of any algorithm for some consistency condition. Similarly, it would be worthwhile to identify certain problems that cannot be solved at all under some consistency conditions The still weaker condition of *safety* (Lamport [17])

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 Malkhi et al. [22], which operate more efficiently than strict quorum systems at the expense of occasionally providing outdated information. Finally, exploring the semantics and consistency model of other data structures, which are built on top of quorum systems, is also of interest.

Acknowledgments: We thank the anonymous referees for their thoughtful comments that greatly improved the paper. We also thank Hyun-Chul Chung, Scott Pike, Gautam Roy, Srikanth Sastry, and Saira Viqar for helpful conversations.

References

[1] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. *ACM Symposium on Parallel Algorithms and Architectures*, Velen, Germany, pp. 251–260, 1993.

[2] M. Ahamad, P. W. Hutto, G. Neiger, J. E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. TR GIT-CC-93/55, Georgia Institute of Technology, July 1994.

[3] F. Anger. On Lamport's Interprocess Communication Model. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 3, pp. 404–417, 1989.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, Vol. 42, No. 1, pp. 124–142, 1995.

[5] H. Attiya and R. Friedman. A Correctness Condition for High Performance Multiprocessors. *SIAM Journal on Computing*, Vol. 27, No. 6, pp. 1637–1670, 1998.

[6] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics, 2nd Ed.* Hoboken, New Jersey, USA, John Wiley and Sons, 2004.

[7] R. A. Bazzi. Synchronous Byzantine Quorum Systems. *Distributed Computing*, Vol. 13, No. 1, pp. 45–52, 2000.

[8] S. Ben-David. The Global Time Assumption and Semantics for Concurrent Systems. *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pp. 223-231, 1988.

[9] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, Vol. 43, No. 2, pp. 225–267, 1996.

[10] R. Friedman, R. Vitenberg, and G. Chockler. On the Composability of Consistency Conditions. *Information Processing Letters*, Vol. 86, pp. 169–176, 2003.

[11] V. K.Garg and M. Raynal. Normality: A Consistency Condition for Concurrent Objects. *Parallel Processing Letters*, Vol. 9, No. 1, pp. 123–134, 1999.

[12] J. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[13] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, pp. 463–492, 1990.

[14] L. Higham and J. Kawash. Tight Bounds for Critical Sections in Processor Consistent Platforms. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 17, No. 10, pp. 1072–1083, 2006.

[15] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, Vol. 17, No. 8, pp. 453–455, 1974.

[16] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690–691, 1979.

[17] L. Lamport. On Interprocessor Communication. Part I and II. *Distributed Computing*, Vol. 1, No. 2, pp. 77–101, 1986.

[18] H. Lee and J. Welch. Randomized Registers and Iterative Algorithms. *Distributed Computing*, Vol. 17, No. 3, pp. 209–221, 2005.

[19] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report 180-88, Department of Computer Science, Princeton University, 1988.

[20] N. Lynch and A. Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. *Proc. 16th International Symposium on Distributed Computing*, pp. 173–190, 2002.

[21] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, Vol. 11, No. 4, pp. 203–213, 1998.

[22] D. Malkhi, M. Reiter, A. Wool, and R. Wright. Probabilistic Quorum Systems. *Information and Computation*, Vol. 170, No. 2, pp. 184–206, 2001.

[23] G. L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, Vol. 12, No. 3, pp. 115–116, 1981.

[24] G. L. Peterson and M. J. Fischer. Economical Solutions for the Critical Section Problem in a Distributed System. *Proc. 9th ACM Symposium on Theory of Computing*, pp. 91–97, 1977.

[25] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, Massachusetts, USA, 1986.

[26] M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. *Proc. 15th Conference on Foundations of Software Technologies and Theoretical Computer Science*, Bangalore, India, pp. 180–194, 1995.

[27] M. Raynal and A. Schiper. A Suite of Formal Definitions for Consistency Criteria in Distributed Shared Memories. *Proc. 9th International Conference on Parallel and Distributed Computing Systems*, pp. 125–131, 1996.

[28] C. Shao. Multi-Writer Consistency Conditions for Shared Memory Objects. M.S. Thesis, Department of Computer Science, Texas A&M University, 2007.

[29] C. Shao, Evelyn Pierce and J. L. Welch. Multi-Writer Consistency Conditions for the Shared Memory Objects. *Proc. 17th International Symposium on Distributed Computing*, pp. 92-105, 2003.

[30] R. C. Steinke and G. J. Nutt. A Unified Theory of Shared Memory Consistency. *Journal of the ACM*, Vol. 51, No. 5, pp. 800–849, 2004.

[31] R. Vitenberg and R. Friedman. On the Locality of Consistency Conditions. *Proc. 17th International Conference on Distributed Computing*, pp. 92–105, 2003.

Appendix: Existing Consistency Models

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

Atomicity (Lamport [17]), also called Linearizability (Herlihy and Wing [13]) is a strong condition requiring that there exist a total ordering of all the operations in a schedule that respects both the semantics of the objects and the partial order of executions of the operations. A schedule σ satisfies Atomicity if:

Definition 12 (Atomicity) *There exists a legal linearization of* $(ops(\sigma), <_\sigma)$.

Sequential consistency (Lamport [16]) requires that there exist a total order of all the operations in a schedule that respects the semantics of the objects and is consistent with the order of operations executed by each process; operations by different processes can be reordered. A schedule σ satisfies Sequential Consistency if:

Definition 13 (Sequential Consistency) *There exists a legal linearization of* $(ops(\sigma), \cup_{i \in P_A} <_{\sigma|i})$.

PRAM was introduced by Lipton and Sandberg [19]. This consistency condition requires that the write operations of a process be observed by other processes in the order in which they are performed. A schedule σ satisfies PRAM if:

Definition 14 (PRAM) *For each process* $i \in P_A$*, there exists a legal linearization of* $(ops(\sigma|i) \cup writes(\sigma), \cup_{j \in P_A} <_{\sigma|j})$.

Coherence (Goodman [12]) 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. A schedule σ satisfies Coherence if:

Definition 15 (Coherence) *For each register* x*, there exists a legal linearization of* $(ops(\sigma|x), \cup_{i \in P_A} <_{\sigma|i})$.

Goodman's Processor Consistency (PCG) is rigorously defined by Ahamad et al. [1]. It is a combination of Coherence and PRAM. A schedule σ satisfies PCG if:

Definition 16 (PCG) *For each process* $i \in P_A$*, there exists a legal linearization* L_i *of* $(ops(\sigma|i) \cup writes(\sigma), \cup_{j \in P_A} <_{\sigma|j})$*. Furthermore, for all processes* p_i *and* p_j *in* P_A*, and for all registers* x*,* L_i *and* L_j *agree on the ordering of all writes to* x*.*

Causal Consistency was introduced by Ahamad et al. [2]. It requires that each process have a legal view of its own operations and all writes that is consistent with the per-process order and some reads-from relation. A schedule σ satisfies Causal Consistency if:

Definition 17 (Causal Consistency) *There exists a reads-from relation* ρ *such that for each process* $i \in P_A$*, there exists a legal linearization of* $(ops(\sigma|i) \cup writes(\sigma), \cup_{j \in P_A} <_{\sigma|j,\rho})$.