Consistency Conditions for Multi-Object Distributed Operations

Neeraj Mittal [*]
neerajm@cs.utexas.edu
Dept. of Computer Sciences

Vijay K. Garg [†]
garg@ece.utexas.edu
Dept. of Electrical and Computer Engg.

The University of Texas at Austin, Austin, TX 78712

Abstract

The traditional Distributed Shared Memory (DSM) model provides atomicity at levels of read and write on single objects. Therefore, multi-object operations such as double compare and swap, and atomic m-register assignment cannot be efficiently expressed in this model. We extend the traditional DSM model to allow operations to span multiple objects. We show that memory consistency conditions such as sequential consistency and linearizability can be extended to this general model. We also provide algorithms to implement these consistency conditions in a distributed system.

1 Introduction

Applications such as distributed file systems, transaction systems, cache coherence for multiprocessors require concurrent accesses to shared data. The underlying system must provide certain guarantees about the values returned by data accesses, possibly to distinct copies of a single logical data object. A consistency condition specifies what guarantees are provided by the system. The consistency conditions should be strong enough to enable easy programming. Sequential consistency and linearizability are two well-known consistency conditions defined in the literature.

Sequential Consistency was proposed by Lamport [14] to formulate a correctness criterion for a multiprocessor shared-memory system. It requires that all data operations (same as actions in our model) appear to have executed atomically, in some sequential order that is consistent with the order seen by individual processes.

Linearizability was introduced by Herlihy and Wing [12] to exploit the semantics of abstract data types. It provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and response. Linearizability is stronger than sequential consistency and has two advantages over it. First, it is more convenient to use because it preserves real-time ordering of operations, and hence corresponds more naturally to the intuitive notion of atomic execution of operations. Consequently, it is easier to develop programs assuming a linearizable implementation of shared objects. Second, linearizability satisfies the local property, that is the system as a whole is linearizable whenever the implementation of each object is linearizable.

These and other consistency conditions [15, 3, 8, 6, 13, 9] are based on the model in which an operation is invoked on a single object. In fact, the traditional Distributed Shared Memory (DSM) provides atomicity only at levels of read and write on single objects. While this may be appropriate for models at the level of hardware, they do not provide an expressive [11] and convenient abstraction for concurrent programming. Herlihy [12] extended the model to arbitrary operations on single objects. That allows the representation of more powerful concurrent objects, for example test and set, fetch and add, FIFO queues and stacks. However, the model assumes that all operations are unary, that is, they are invoked on a single object. There are many applications in which operations are more naturally expressed as encompassing multiple objects. For example, operations like double compare and swap (DCAS)[1] [10] cannot be efficiently expressed in that model. DCAS reduces the allocation and copy cost thereby permitting a more efficient implementation of concurrent objects. As another application, if a transaction in a database is viewed as an atomic operation then it is clear that it operates, in general, on multiple data items.

In this paper, we develop a framework for consistency conditions for distributed objects with multi-object operations (or multi-methods). We introduce a formal model for execution of operations that span multiple objects. In this model, each process executes multiple operations and each operation consists of mul-

[*]supported in part by the MCD Fellowship

[†]supported in part by the NSF Grants ECS-9414780, CCR-9520540, Texas Higher Education Board grant ARP-320, a General Motors Fellowship, and an IBM grant

[1]DCAS atomically updates locations $addr_1$ and $addr_2$ to values new_1 and new_2 respectively if $addr_1$ holds value old_1 and $addr_2$ holds old_2 when the operation is invoked.

tiple actions (possibly on different objects). With the increasing popularity of distributed objects it is important to understand the conditions for their consistency in presence of replication and caches.

Besides practical implications, our model has nice theoretical consequences. It serves to unify results from two areas. By restricting the number of operations per process to one, the model reduces to that of database transactions. Similarly, if we restrict each operation to execute actions on a single object then the model reduces to that of distributed shared memory [1] on concurrent objects [12]. Thus with our model, one set of consistency conditions, their implementation, and complexity results are applicable to both the areas.

It has been shown that determining whether a given execution is sequentially consistent when the operations are restricted to read/write on a single object is an NP-complete problem [22]. We show that the problem of checking whether a given history is linearizable is also NP-complete in our model. This is true even when the reads-from relation (defined later) is known. Note that when the operations are restricted to a single read/write register and the reads-from relation is known, then linearizability can be checked in polynomial time [18].

We show that execution constraints proposed by Raynal *et al* [20] to ensure efficient implementation for sequential consistency can also be used for operations that span multiple objects. Specifically, under these execution constraints, it is necessary and sufficient to ensure legality of reads to guarantee sequential consistency (and linearizability).

Finally, we provide algorithms for ensuring proposed consistency conditions in a distributed system. Several papers [2, 4, 17, 20] have proposed sequentially consistent implementations for read/write objects. Attiya and Welch [4] provide sequentially consistent and linearizable implementations for read/write objects, FIFO queues and stacks. In addition, they also give an analysis of the response time of their implementations. But their implementation for linearizability assumes that clocks are perfectly synchronized and there is an upper bound on the delay of the message. Our algorithm for sequential consistency is an extension of the algorithm proposed by them. We show that their algorithm also works for multi-object operations. More importantly, we provide an algorithm for implementation of linearizability in an asynchronous distributed system which does not make any assumptions about clock synchronization or the message delay.

It should be noted that there may be a temptation to model multi-methods by defining an aggregate object that represents the state of all objects. However, this technique has serious drawbacks. For example, if there are n read-write registers and one multi-method *sum* that takes two registers as arguments, the technique will force all registers to be treated as one object. This results in loss of locality and concurrency.

This paper is organized as follows. Section 2 gives our model of a concurrent system with multi-object operations and presents the consistency conditions appropriate in this model. In Section 3 we show the NP-completeness of verification of linearizability. Section 4 imposes additional constraints on execution for efficient implementation of distributed objects. In Section 5 we present algorithms for implementation of sequential consistency and linearizability in distributed systems.

2 Definitions

2.1 System Model

A *concurrent system* consists of a finite set of sequential threads of control called *processes*, denoted by $P_1, P_2, \ldots, P_n$, that communicate through a set of shared data structures called *objects* (or *concurrent objects*) X . Each object can be accessed by *read* and *write* actions (same as traditional operation on a single object). A write into an object defines a new value for the object; a read allows to obtain the value of the object. A write action on an object x is denoted by $w(x)v$, where v is the value written to x by this action. A read action on x is denoted by $r(x)v$, where v is the value of object x returned by this action.

Processes are sequential and manipulate objects through operations. An operation is a sequence of actions possibly spanning several objects. Each process applies a sequence of operations to objects, alternately issuing an invocation and then receiving the associated response. Let $\alpha(arg, res)$ be an operation issued at P_i; arg and res denote α's input and output parameters respectively. Execution of an operation takes certain time; this is modeled by two events, namely an *invocation* event and a *response* event. For an operation α, invocation and response events, $inv(\alpha(arg))$ at P_i and $resp(\alpha(res))$ at P_i, will be abbreviated as $inv(\alpha)$ and $resp(\alpha)$ when parameters and process identity are not necessary. An event e *occurs-before* event f, denoted by $e < f$, iff event e precedes event f in real time. We will use greek symbols α, β, γ, δ, etc. to denote operations.

If two operations α and β are issued by the same process, say P_i, and α is issued before β, then we say α precedes β in P_i's *process order* and is written as $\alpha \leadsto_{P_i} \beta$. If process identity is not important then process order is denoted by $\leadsto_P$. In Figure 1, $\alpha \leadsto_{P_1} \beta$.

If a read action $r(x)v$ reads the value written by the write action $w(x)v$, then $r(x)v$ is said to *read-from*

$w(x)v$. An operation α *reads-from* a distinct operation β the value of object x, written as $\beta \leadsto_{rf} \alpha$, if there exists at least one read action of α that reads from some write action of β the value of object x. In Figure 1, $\alpha \leadsto_{rf} \delta$ and $\eta \leadsto_{rf} \delta$.

We assume that an imaginary operation that writes to all objects is performed to initialize the objects before the first operation by any process is executed. In all the examples considered in this paper, unless specified otherwise, we assume that initial value of all objects is 0.

2.2 Histories

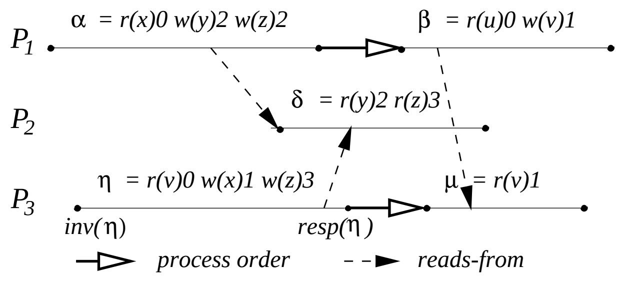

Figure 1: An execution history $\mathcal{H}'$

Informally, an execution of a concurrent system is modeled by a *history*, which is a finite sequence of operation invocation and response events. Formally, a history $\mathcal{H}$ is denoted by a tuple $(op(\mathcal{H}), \leadsto_{\mathcal{H}})$, where $op(\mathcal{H})$ is the set of operations and $\leadsto_{\mathcal{H}}$ is some irreflexive transitive relation defined on the set of operations which includes the partial order imposed by process orders and reads-from relation.

A history $\mathcal{S}$ is *sequential* iff (1) its first event is an invocation event, (2) each invocation event is immediately followed by a matching response event, and (3) $\leadsto_{\mathcal{S}}$ is a total order consistent with the order of operation invocation events.

A *process subhistory* or *local history* of P_i of a history $\mathcal{H}$, denoted by $\mathcal{H}|P_i$, is the subsequence of all events in $\mathcal{H}$ associated with the process P_i. A history is *well-formed* iff each process subhistory is sequential. All histories considered in this paper are assumed to be well-formed.

Two histories $\mathcal{H}$ and $\mathcal{G}$ are *equivalent* iff for every process P_i, $\mathcal{H}|P_i = \mathcal{G}|P_i$ and they have the same reads-from relation.

Intuitively, a read action is legal if it does not read from an overwritten write action. It should be noted that if there exists a write action $w(x)v$ before a read action $r(x)u$ in an operation (such that $w(x)v$ is the last write on x before $r(x)u$) then u must be equal to v. In the rest of the paper, we ignore such read actions. Let $op(a(x)v)$ denote the operation associated with the action $a(x)v$. A read action $r(x)v$ is *legal* iff there exists a write action $w(x)v$ such that $r(x)v$ reads from $w(x)v$ and there does not exist another write action $w'(x)u$ such that $op(w(x)v) \leadsto_{\mathcal{H}} op(w'(x)u) \leadsto_{\mathcal{H}} op(r(x)v)$. An operation is *legal* iff all its read actions are legal. A history $\mathcal{H}$ is *legal* iff all its operations are legal.

A history $\mathcal{H}$ is *admissible* iff it is equivalent to some legal sequential history that respects $\leadsto_{\mathcal{H}}$.

2.3 Consistency Conditions

A consistency policy makes the behavior of a concurrent system equivalent to that of a non-concurrent system. A consistency condition provides guarantees about the values returned by data accesses in the presence of interleaved and/or overlapping accesses. Sequential Consistency and Linearizability are two well known consistency conditions. We extend their definitions to include operations on multiple objects. Our definitions are based on the definition of admissibility with the partial order $\leadsto_H$ appropriately defined.

Let $proc(\alpha)$ and $objects(\alpha)$ denote the process and the set of objects respectively associated with an operation α. In Figure 1, $proc(\alpha) = P_1$ and $objects(\alpha) = \{x, y, z\}$. The operations α and β are related by *real-time order*, denoted by $\alpha \leadsto_t \beta$, iff the response of α is received before the invocation of β, that is, $resp(\alpha) < inv(\beta)$. The operations α and β are related by *object order*, denoted by $\alpha \leadsto_X \beta$, iff both the operations share an object and the response of α is received before the invocation of β, that is, $(objects(\alpha) \cap objects(\beta) \neq \phi) \wedge (resp(\alpha) < inv(\beta))$. In Figure 1, $\alpha \leadsto_t \mu$, $\eta \leadsto_t \beta$ and $\eta \leadsto_X \beta$.

Sequential Consistency requires that all data operations appear to have executed atomically, in some sequential order that is consistent with the order seen by individual processes. Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history, where $\leadsto_{\mathcal{H}}$ includes process orders, and reads-from relation. Then $\mathcal{H}$ is *sequentially consistent* iff it is admissible. If operations are restricted to a single read or write action, then our definition reduces to Lamport's definition of sequential consistency.

Linearizability requires that: (1) each operation should appear to take effect instantaneously somewhere between its invocation and response, and (2) the order of non-concurrent operations should be preserved. Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history, where $\leadsto_{\mathcal{H}}$ includes process orders, reads-from relation and real-time order. Then $\mathcal{H}$ is *linearizable* iff it is admissible.

Garg and Raynal [8] proposed another definition of consistency which is based on object-order rather than real-time order. Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history, where $\leadsto_{\mathcal{H}}$ includes process orders, reads-from relation, and object order. Then $\mathcal{H}$ is *normal* iff it is admissible. Normality is less restrictive than linearizability since it does not order two non-concurrent operations unless they act on a common object. The results of Section 3 and Section 4 also hold for normality. Since the protocol for linearizability also implements normality, we will focus on linearizability in the rest of the paper.

3 NP-completeness of Consistency Conditions

It has been shown that ascertaining whether a given execution is sequentially consistent when the operations are restricted to a single object is an NP-complete problem [22]. Since our model is a generalization of the traditional DSM model, determining whether a given execution is sequentially consistent in our model is NP-complete too. Misra proved that checking whether an execution satisfies atomic consistency is solvable in polynomial time when reads-from relation is known [18]. It turns out that this is not the case when the operations can encompass multiple objects. In this section we show that determining whether a given execution is linearizable is an NP-complete problem when the operations are allowed to span multiple objects and even when reads-from relation is known. We will use the results in databases to prove the NP-completeness of linearizability.

Much work on databases uses *serializability* [21, 5] as the basic correctness condition for concurrent computations. Several notions of equivalence such as *view equivalence*, *strict view equivalence*, and *conflict equivalence* are defined [21]. If we restrict each process to contain a single operation (one for each transaction) then the notion of correctness in the database world can be viewed as special case of the consistency conditions in our model. For instance, view equivalence can be considered as a special case of sequential consistency; strict view equivalence can be viewed as a special case of linearizability, and conflict equivalence can be considered as a special case of normality under OO-constraint (defined later). Since determining whether a schedule is strict view serializable is an NP-complete problem, hence checking whether a history is linearizable is also an NP-complete problem. It should be noted that checking for linearizability of history $\mathcal{H}$ is not same as checking for acyclicity of $\leadsto_{\mathcal{H}}$. In particular, $\leadsto_{\mathcal{H}}$ may be acyclic but $\mathcal{H}$ may not be linearizable.

Theorem 1 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history. Then it is NP-complete to determine whether the history $\mathcal{H}$ is sequentially consistent.*

Theorem 2 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history. Then it is NP-complete to determine whether the history $\mathcal{H}$ is linearizable.*

Proof: To prove that determining whether a history $\mathcal{H}$ is linearizable is NP-hard we reduce strict view serializability[2] to linearizability. Let $\mathcal{S} = (trans(\mathcal{S}), \leadsto_S)$ be a schedule of transactions in a database consisting of finite set of entities $E = \{x_1, x_2, \ldots\}$, where $trans(\mathcal{S})$ denote the set of transactions $T_1, T_2, \ldots, T_n$, and $\leadsto_S$ represents the order of actions in the schedule. We construct a distributed system consisting of sequential processes $P_0, P_1, P_2, \ldots P_n, P_\infty$, one for each transaction in the augmented schedule[3], and shared objects E. For each step in the schedule there is a corresponding action. An action a_i reads from action a_j if the corresponding step of action a_i reads-from the corresponding step of action a_j in the schedule $\mathcal{S}$. Each process P_i executes a single operation α_i whose actions correspond to the steps of the transaction T_i executed in the same order. The first and last steps of a transaction define the invocation and response events respectively of the corresponding operation. It is easy to see that two transactions are concurrent in the schedule $\mathcal{S}$ if and only if the corresponding operations are concurrent in $\mathcal{H}$. The history $\mathcal{H}$ of the system is the history $(op(\mathcal{H}), \leadsto_{\mathcal{H}})$ where $op(\mathcal{H})$ is the set of transactions and $\leadsto_{\mathcal{H}}$ consists of reads-from relation and real-time order. It can be easily proved that schedule $\mathcal{S}$ is strict view serializable if and only if the history $\mathcal{H}$ is linearizable. Moreover, it can be easily verified that the problem is indeed in NP since, given a sequential history, we can easily check that it is legal and equivalent to $\mathcal{H}$. ■

4 Consistency Conditions with Constraints

Due to Theorem 1 and Theorem 2 it is unlikely that there exists an efficient algorithm that realizes sequential consistency (linearizability), that is, allows all sequentially consistent (linearizable) histories and only these. Thus we need to impose constraints on each history to ensure efficient implementations of consistent DSMs. Raynal *et al* [20] identified two such constraints, namely WW-and OO-constraints, for sequential consistency. We extend their work in two ways: we show that (1) their results extend to the case when the operations can span multiple objects, and (2) similar results also hold for linearizability. Before proceeding further, we give some definitions we use in this section.

Two actions are said to be *conflicting* iff both act on the same object and at least one of them is a write

[2]A schedule $\mathcal{S}$ is *strict view serializable* if it is view equivalent to a serial schedule in which transactions that do not overlap in $\mathcal{S}$ are in the same order as in $\mathcal{S}$.

[3]a schedule augmented with an initial transaction writing values to each entity and a final transaction reading values from each entity.

action. Two operations are said to be *conflicting* iff one of them contains an action that conflicts with some action of the other. The operations α, β, γ are said to *interfere* in history $\mathcal{H}$ iff γ writes to some object that α reads from β. In Figure 1, α conflicts with η, and operations δ, η and α interfere.

A history $\mathcal{H}$ satisfies *WO-constraint* iff any pair of operations performing write actions on a common object are ordered under $\leadsto_{\mathcal{H}}$. A history $\mathcal{H}$ satisfies *WW-constraint* iff any pair of operations performing write actions are ordered under $\leadsto_{\mathcal{H}}$. A history $\mathcal{H}$ satisfies *OO-constraint* iff any pair of conflicting operations are ordered under $\leadsto_{\mathcal{H}}$.

We will see that these constraints permit an efficient implementation of consistency conditions. In this section, we prove that legality is the necessary and sufficient condition for an execution history $\mathcal{H}$ under WW- or OO-constraint to be admissible.

Theorem 3 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history. If $\mathcal{H}$ is admissible then it is legal.*

Proof: Assume, on the contrary, that $\mathcal{H}$, is not legal. Therefore there exist operations α, β and γ that interfere in $\mathcal{H}$ such that $\beta \leadsto_{\mathcal{H}} \gamma \leadsto_{\mathcal{H}} \alpha$ holds. Let $\mathcal{S} = (op(\mathcal{S}), \leadsto_{\mathcal{S}})$ be the legal sequential history equivalent to $\mathcal{H}$ that respects $\leadsto_{\mathcal{H}}$. Thus $\beta \leadsto_{\mathcal{S}} \gamma \leadsto_{\mathcal{S}} \alpha$ holds. Since $\mathcal{S}$ has the same reads-from relation as $\mathcal{H}$, therefore $\mathcal{S}$ is not legal - a contradiction. ■

We now show that legality is sufficient to guarantee that a history $\mathcal{H}$ under OO-constraint is admissible. So a protocol based on OO-constraint just needs to ensure the legality of all its operations and that will guarantee that all executions generated by it are admissible.

Theorem 4 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history under OO-constraint. If $\mathcal{H}$ is legal then it is admissible.*

Proof: Since $\leadsto_{\mathcal{H}}$ defines an irreflexive partial order on $op(\mathcal{H})$, extend $\leadsto_{\mathcal{H}}$ to any total order on $op(\mathcal{H})$, say $\leadsto_{\mathcal{S}}$. Now we have to show that $\leadsto_{\mathcal{S}}$ defines a legal relation on $op(\mathcal{H})$. Let α, β and γ be operations that interfere in $\mathcal{H}$. Since history is under OO-constraint, either $\gamma \leadsto_{\mathcal{H}} \beta$ or $\beta \leadsto_{\mathcal{H}} \gamma$ holds. In the first case, $\gamma \leadsto_{\mathcal{S}} \beta \leadsto_{\mathcal{S}} \alpha$ holds. In the second case, since the history $\mathcal{H}$ is legal and under OO-constraint, $\beta \leadsto_{\mathcal{S}} \alpha \leadsto_{\mathcal{S}} \gamma$ holds. Since the operations α, β and γ were chosen arbitrarily, $\mathcal{S} = (op(\mathcal{S}) = op(\mathcal{H}), \leadsto_{\mathcal{S}})$ is legal. Thus $\mathcal{H}$ is admissible. ■

We next show that legality is a sufficient condition to guarantee admissibility of a history under WW-constraint. A history under WW-constraint permits the operations, one of which only reads from an object and the other writes on the same object, to execute concurrently. Therefore we define a *logical read-write* precedence, denoted by $\leadsto_{rw}$, between two such operations which are not ordered under $\leadsto_{\mathcal{H}}$. Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history and let α, β and γ be operations that interfere in $\mathcal{H}$. Then $\beta \leadsto_{\mathcal{H}} \gamma \Rightarrow \alpha \leadsto_{rw} \gamma$. We define an extended relation as $\leadsto^{e}_{\mathcal{H}} = (\leadsto_{\mathcal{H}} \cup \leadsto_{rw})^{+}$.

Lemma 5 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history under WO-constraint. If $\leadsto^{e}_{\mathcal{H}}$ is acyclic then $\mathcal{H}$ is admissible.*

Proof: Let $\leadsto_{\mathcal{S}}$ be any total order that extends $\leadsto^{e}_{\mathcal{H}}$ (it can be done since $\leadsto^{e}_{\mathcal{H}}$ is acyclic). We now need to prove that $\leadsto_{\mathcal{S}}$ is legal. Let α, β and γ be operations that interfere in $\mathcal{H}$. Since $\mathcal{H}$ is under WO-constraint, either $\gamma \leadsto_{\mathcal{H}} \beta$ or $\beta \leadsto_{\mathcal{H}} \gamma$ holds. In the first case, $\gamma \leadsto_{\mathcal{S}} \beta \leadsto_{\mathcal{S}} \alpha$ holds. In the second case, we have $\alpha \leadsto_{rw} \gamma$, and therefore $\beta \leadsto_{\mathcal{S}} \alpha \leadsto_{\mathcal{S}} \gamma$ holds. Hence γ cannot be ordered between β and α in $\mathcal{S}$. Since operations α, β and γ were chosen arbitrarily, $\mathcal{S} = (op(\mathcal{S}) = op(\mathcal{H}), \leadsto_{\mathcal{S}})$ is legal. Thus $\leadsto_{\mathcal{H}}$ is admissible. ■

Lemma 6 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history under WW-constraint. If $\mathcal{H}$ is legal then $\leadsto^{e}_{\mathcal{H}}$ is acyclic.*

Proof: Since $\leadsto_{\mathcal{H}}$ is an irreflexive transitive relation, any cycle in $\leadsto^{e}_{\mathcal{H}}$ must involve at least one pair of operations ordered by $\leadsto_{rw}$. We will prove that $\leadsto^{e}_{\mathcal{H}}$ is acyclic by induction on number of pair of operations n ordered by $\leadsto_{rw}$ relation in a cycle.
Base Case $(n = 1)$ **:** Any cycle is of the form $\alpha \leadsto_{rw} \gamma \leadsto_{\mathcal{H}} \alpha$ ($\leadsto_{\mathcal{H}}$ is transitive). By definition of $\leadsto_{rw}$, there exists an operation β such that α, β and γ interfere in $\mathcal{H}$, and $\beta \leadsto_{\mathcal{H}} \gamma$ holds. Therefore $\beta \leadsto_{\mathcal{H}} \gamma \leadsto_{\mathcal{H}} \alpha$ holds, and hence $\mathcal{H}$ is not legal - a contradiction.
Induction Step : Consider a cycle with $n > 1$ pair of operations ordered by $\leadsto_{rw}$ relation. The cycle is of the form $\alpha \leadsto_{rw} \gamma \leadsto \cdots \leadsto_{rw} \delta \leadsto \cdots \leadsto \alpha$. Since $\mathcal{H}$ is under WW-constraint, either $\gamma \leadsto_{\mathcal{H}} \delta$ or $\delta \leadsto_{\mathcal{H}} \gamma$ holds. In either case we have a cycle involving less than n pair of operations ordered by $\leadsto_{rw}$ relation. Hence by induction $\leadsto^{e}_{\mathcal{H}}$ is acyclic. ■

Theorem 7 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history under WW-constraint. If $\mathcal{H}$ is legal then it is admissible.*

Theorem 8 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history under OO- or WW-constraint. Then $\mathcal{H}$ is admissible if and only if it is legal.*

A protocol based on OO- or WW-constraint only needs to ensure that all reads are legal and that will guarantee that all executions generated by it are admissible.

5 Implementation of Consistency Conditions

This section presents the implementations of sequential consistency and linearizability in an asynchronous message passing system and proves their correctness. Our protocols assume that the processes and the communication channels are reliable.

Each process maintains a private copy M of the abstract shared memory $\mathcal{M}$. We divide the set of operations into two types: operations that perform only read actions, denoted by $READ$, and operations that perform at least one write action, denoted by $WRITE$. The result of an operation α on application to memory M is represented by $\alpha(M)$. Our both protocols are based on WW-constraint and rely on atomic broadcasting for synchronization of write operations. If the underlying hardware provides an atomic broadcast facility, these protocols can be implemented efficiently.

In addition to a copy of the shared memory, each process also maintains a vector ts of natural numbers, one entry for each object, called timestamp. Two timestamps can be compared by comparing their components. A timestamp $ts_k \preceq ts_l$ if each of ts_k's components is less than or equal to ts_l's corresponding component; $ts_k \prec ts_l$ if $ts_k \preceq ts_l$ and ts_k is not equal to ts_l.

We associate a timestamp with each operation α which is denoted by $ts(\alpha)$. Before describing the protocol, we show that if timestamps of the operations satisfy certain properties, then the execution is admissible.

Lemma 9 *Let $\mathcal{H} = (op(\mathcal{H}), \leadsto_{\mathcal{H}})$ be an execution history such that $\leadsto_{\mathcal{H}}$ is an irreflexive transitive closure of $\leadsto$, where $\leadsto$ is some irreflexive relation defined on $op(\mathcal{H})$ (which includes process orders and reads-from relation). Let $ts(\alpha)$ denote the timestamp of an operation α. If the timestamp satisfies the following properties :*

1. *If $\alpha \leadsto \beta$ then*
   - *(a) $ts(\alpha) \preceq ts(\beta)$,*
   - *(b) β writes on $x \Rightarrow ts(\alpha)[x] < ts(\beta)[x]$, and*
   - *(c) $\alpha, \beta \in READ \Rightarrow resp(\alpha) < inv(\beta)$*
2. *If β reads from α the value of object x then*
   - *(a) β writes on $x \Rightarrow ts(\alpha)[x] = ts(\beta)[x] - 1$,*
   - *(b) β does not write on $x \Rightarrow ts(\alpha)[x] = ts(\beta)[x]$*
3. *$\forall \alpha, \beta \in WRITE$, either $\alpha \leadsto \beta$ or $\beta \leadsto \alpha$.*

then $\mathcal{H}$ is admissible.

Proof: Intuitively, the properties $1(a)$, $1(b)$ and $1(c)$ imply that the relation $\leadsto$ is acyclic and hence $\mathcal{H}$ is indeed a valid execution history. The properties $2(a)$ and $2(b)$ imply that a read action does not read from an overwritten write action and the property 3 implies that the history satisfies WW-constraint.

We claim that $\leadsto$ is acyclic. Assume, on the contrary, that $\leadsto$ contains a cycle, namely $\alpha \leadsto \beta \leadsto \cdots \leadsto \gamma \leadsto \alpha$. If there exists at least one operation δ in the cycle such that $\delta \in WRITE$, then by properties $1(a)$ and $1(b)$ we get $ts(\alpha) \preceq ts(\beta) \cdots \prec ts(\delta) \preceq \cdots \preceq ts(\alpha)$. Hence $ts(\alpha) \prec ts(\alpha)$, a contradiction. Therefore, none of the operations in the cycle is in $WRITE$. Hence all operations of the cycle are in $READ$, and from property $1(c)$ we get $resp(\alpha) < inv(\beta) < resp(\beta) < \cdots < inv(\alpha) < resp(\alpha)$, again a contradiction. Therefore we can conclude that $\leadsto$ is acyclic and $\leadsto_{\mathcal{H}}$ is an irreflexive transitive relation. Using property 3 we can infer that $\mathcal{H}$ satisfies WW-constraint.

Now we need to show that $\mathcal{H}$ is legal. Assume again, on the contrary, that the history is not legal. Hence there exist operations α, β and γ and an object x such that α, β and γ interfere in $\mathcal{H}$ on object x and $\beta \leadsto_{\mathcal{H}} \gamma \leadsto_{\mathcal{H}} \alpha$ holds. Note that since $\leadsto_{\mathcal{H}}$ is an irreflexive transitive closure of $\leadsto$, properties $1(a)$ and $1(b)$ also hold for $\leadsto_{\mathcal{H}}$. Therefore we have $ts(\beta)[x] < ts(\gamma)[x]$. There are two cases to consider: α only reads from x or it also writes to x. In the first case, by property $2(b)$, we get $ts(\beta)[x] = ts(\alpha)[x]$ and $ts(\gamma)[x] \leq ts(\alpha)[x]$ and hence $ts(\beta)[x] < ts(\beta)[x]$, a contradiction. In the second case, by property $2(a)$, we get $ts(\beta)[x] = ts(\alpha)[x] - 1$ and $ts(\gamma)[x] < ts(\alpha)[x]$. Hence $ts(\beta)[x] < ts(\beta)[x]$, again a contradiction. Therefore $\mathcal{H}$ is legal which together with Theorem 8 implies that execution history $\mathcal{H}$ is admissible. ∎

Now we present the protocols for implementing sequential consistency and linearizability. For each protocol, we associate a timestamp with each operation and then show that the timestamps satisfy the properties required in the Lemma 9.

5.1 Implementation of Sequential Consistency

The protocol in Figure 2 consists of an initialization routine and three basic actions each of which is executed locally and atomically. The statements enclosed in curly brackets are not part of the protocol but are used to prove the correctness of the algorithm.

```
Process $P_i$

/* Initialization */
foreach $x \in \mathcal{M}$ do
    $M[x] \leftarrow \perp$
    { $ts[x] \leftarrow 0$ }

/* $\alpha \in READ$ */
return($\alpha(M)$)

/* $\alpha \in WRITE$ */
atomically broadcast the operation to all processes

/* On receiving atomic broadcast of operation $\alpha$
    from process $P_j$ */
apply the operation to the memory $M$
{ foreach $x$ such that $\alpha$ writes on $x$ do $ts[x] \leftarrow ts[x] + 1$}
if $proc(\alpha) = P_i$ then return($\alpha(M)$)
```

Figure 2: Protocol for Sequential Consistency

Theorem 10 *All the executions generated by the protocol in Figure 2 are sequentially consistent.*

Proof: Consider any execution $\mathcal{H}$ generated by the protocol. Let the reads-from relation $\leadsto_{rf}$ be defined as: a read action of an operation α reads from the last write action on that object in $proc(\alpha)$'s memory. Let $\leadsto_{ww}$ denotes the order in which operations in $WRITE$ are broadcasted. We define $\leadsto$ (as in Lemma 9) to be the union of process orders ($\leadsto_P$), reads-from relation ($\leadsto_{rf}$) and atomic broadcast order ($\leadsto_{ww}$). Note that $\alpha \leadsto_P \beta$ or $\alpha \leadsto_{rf} \beta$ or $\alpha \leadsto_{ww} \beta$ imply that α is applied to $proc(\beta)$'s memory before β. Let $ts_i(\alpha)$ denote the timestamp of process P_i just after application of α to P_i's memory.

We define the timestamp of an operation α issued by process P_i as $ts(\alpha) = ts_i(\alpha)$. Note that only operations in $WRITE$ modify the timestamp ts. Since all operations in $WRITE$ are applied in same order on every process, therefore for every operation $\alpha \in WRITE$ the following holds: for every process P_j, $ts(\alpha) = ts_j(\alpha)$.

If $\alpha \leadsto \beta$ holds, then α is applied to $proc(\beta)$'s memory before β. Since vector timestamp never decreases, the property $1(a)$ holds. If β writes on object x, then $proc(\beta)$ will increment $ts_{proc(\beta)}[x]$ by 1 after applying β to its memory. Therefore the property $1(b)$ holds.

Since operations belonging to $READ$ can only be ordered by process-order, property $1(c)$ trivially follows.

Property $2(a)$ and $2(b)$ follow from the fact that since β reads from α the value of object x, α is the last operation to write on x in $proc(\beta)$'s memory. Therefore $ts_{proc(\beta)}[x]$ just before the application of β is equal to $ts(\alpha)[x]$. Now if β writes on x then it increments the entry for x, and therefore $ts(\alpha)[x] = ts(\beta)[x] - 1$, otherwise $ts(\alpha)[x] = ts(\beta)[x]$.

Property 3 follows from the fact that all operations in $WRITE$ are atomically broadcasted and hence ordered under $\leadsto_{ww}$.

Hence, by Lemma 9, the history $\mathcal{H}$ is sequentially consistent, and therefore the protocol only generates sequentially consistent executions. ■

5.2 Implementation of Linearizability

The protocol in Figure 3 consists of an initialization routine and five basic actions each of which is executed locally and atomically. Each process maintains another vector timestamp ts' which records the latest writes by any process when it issues an operation in $READ$.

```
Process $P_i$

/* Initialization */
foreach $x \in \mathcal{M}$ do
    $M[x], ts[x] \leftarrow \perp, 0$
    $M'[x], ts'[x] \leftarrow \perp, 0$

/* $\alpha \in READ$ */
send query to all the processes for objects $\in \mathcal{M}$

/* $\alpha \in WRITE$ */
atomically broadcast the operation to all processes

/* On receiving atomic broadcast of operation $\alpha$
    from process $P_j$ */
apply the operation to the memory $M$
foreach $x$ such that $\alpha$ writes on $x$ do $ts[x] \leftarrow ts[x] + 1$
if $proc(\alpha) = P_i$ then return($\alpha(M)$)

/* On receiving query for set of objects $X_j$
    from process $P_j$ */
send $(ts, M)$ to process $P_j$

/* On receiving response, denoted by $(ts'_j, M'_j)$,
    for the query from process $P_j$ */
foreach $x \in \mathcal{M}$ do
    if $ts'_j[x] > ts'[x]$ then $M'[x], ts'[x] \leftarrow M'_j[x], ts'_j[x]$
if all the responses for the last query have been received
then return($\alpha(M')$)
```

Figure 3: Protocol for Linearizability

Theorem 11 *All the executions generated by the protocol in Figure 3 are linearizable.*

Proof: The proof is similar to the proof of Theorem 10 but is more involved. Due to the lack of space we will not present the proof here. The interested reader can refer to the technical report [19].

6 Conclusion

We extend the traditional model of concurrent objects to allow operations that span multiple objects. We give the consistency conditions in this model, analyze their verification complexity and give efficient algorithms for ensuring them in distributed systems.

References

[1] Sarita V. Adve and K. Gharachorloo. "Shared Memory Consistency Models: A Tutorial". *IEEE Computer*, pages 66–76, December 1996.

[2] Y. Afek, G. Brown, and M. Merritt. "Lazy Caching". *ACM Transactions on Programming Language and Systems*, 15(1):182–205, January 1993.

[3] Mustaque Ahamad, Phillip W. Hutto, and Ranjit John. "Causal memory: Definitions, Implementation and Programming". Technical Report 93/55, College of Computing, Georgia Institute of Technology, September 1993.

[4] Hagit Attiya and Jennifer L. Welch. "Sequential Consistency versus Linearizability". *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman. *"Concurrency Control and Recovery in Database Systems"*. Addison-Wesley, Reading, MA, 1987.

[6] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. "Dag-Consistent Distributed Shared Memory". In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, April 15-19, 1996.

[7] C. J. Fidge. "Logical Time in Distributed Computing Systems". *IEEE Computer*, 24(8):28–33, 1991.

[8] Vijay K. Garg and Michel Raynal. "Normality: A Consistency Conditions for Concurrent Objects". Technical Report TR-PDS-1996-010, The University of Texas at Austin, May 1996. To appear in Parallel Processing Letters.

[9] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors". In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] Michael Greenwald and David Cheriton. "The Synergy Between Non-blocking Synchronization and Operating System Structure". In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, USENIX, Seattle, October 1996.

[11] Maurice Herlihy. "Wait-Free Synchronization". *ACM Transactions on Programming Language and Systems*, 11(1):124–149, January 1991.

[12] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects". *ACM Transactions on Programming Language and Systems*, 12(3):463–492, July 1990.

[13] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. "Treadmarks: Distributed Shared Memory on Standard Workstations and operating systems". In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[14] Leslie Lamport. "How to make a multiprocessor computer that correctly executes multiprocess programs". *IEEE Transactions on Computers*, C28(9):690–691, September 1979.

[15] Richard J. Lipton and Jonathan S. Sandberg. "PRAM: A scalable shared memory". Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[16] Friedemann Mattern. "Virtual time and global states of distributed systems". *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, October 1988.

[17] Marios Mavronicolas and Dan Roth. "Sequential Consistency and Linearizability: Read/Write objects". In *Proceedings of Twenty-Ninth Annual Allerton Conference on Communication, Control and Computing*, pages 683–692, October 1991.

[18] Jayadev Misra. "Axioms for memory access in asynchronous hardware systems". *ACM Transactions on Programming Language and Systems*, 8(1):142–153, January 1986.

[19] Neeraj Mittal and Vijay K. Garg. "Consistency Conditions for Multi-Object Distributed Operations". Technical Report TR-PDS-1998-005, The University of Texas at Austin, 1998.

[20] M. Mizuno, M. Raynal, and J.Z. Zhou. "Sequential Consistency in Distributed Systems: Theory and Implementation". Technical Report 871, INRIA, Rennes, France, October 1994.

[21] C. H. Papadimitriou. *"The Theory of Concurrency Control"*. Computer Science Press, May 1986.

[22] Richard N. Taylor. "Complexity of Analyzing the Synchronization Structure of Concurrent Programs". *Acta Informatica*, 19:57–84, 1983.