Linearizable Read/Write Objects*

Marios Mavronicolas†
Department of Computer Science
University of Cyprus
Nicosia CY-1678, Cyprus

Dan Roth‡
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

September 1998

*This paper combines, unifies and extends results that appear in preliminary form in [46] and [47].

†Currently at AT&T Labs – Research, NJ, as a visitor to the Special Year on Networks, organized by the DIMACS Center for Discrete Mathematics and Theoretical Computer Science, Rutgers University, NJ. Part of the work of this author was performed while at Aiken Computation Laboratory, Harvard University, supported by ONR contract N00014-91-J-1981, at Department of Computer Science, University of Crete, and at Institute of Computer Science, Foundation for Research and Technology – Hellas. Partially supported by funds for the promotion of research at University of Cyprus. E-mail: mavronic@turing.cs.ucy.ac.cy

‡Part of the work of this author was performed while at Aiken Computation Laboratory, Harvard University, supported by NSF grant CCR-89-02500, and at Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science. E-mail: danr@cs.uiuc.edu

Abstract

We study the cost of using message passing to implement *linearizable read/write objects* for shared-memory multiprocessors under various assumptions on the available timing information. We take as cost measures the *worst-case response times* for performing read and write operations in distributed implementations of virtual shared memory consisting of such objects, and the sum of these response times. It is assumed that processes have clocks that run at the same rate as real time and are within δ of each other, for some known *precision* constant $\delta \geq 0$. All messages incur a delay in the range $[d-u, d]$ for some known constants u and d, $0 \leq u \leq d$.

For the *perfect clocks* model, where clocks are perfectly synchronized, i.e., $\delta = 0$, and every message incurs a delay of exactly d, we present a linearizable implementation which achieves worst-case response times for read and write operations of βd and $(1-\beta)d$, respectively; β is a trade-off parameter, $0 \leq \beta \leq 1$, which may be tuned to account for the relative frequencies of read and write operations. This implementation is optimal with respect to the sum of the worst-case response times for read and write operations.

We next turn to the *approximately synchronized clocks* model, where clocks are only approximately synchronized, i.e., $\delta > 0$, and message delays can vary, i.e., $u > 0$. Our first major result is the first known linearizable implementation for this model which achieves worst-case response times of less than $\beta d + 3u + \min\{\delta, u\} + \varepsilon$, and $(1-\beta)d + 3u$ for read and write operations, respectively, under a mild restriction on the trade-off parameter β, $0 \leq \beta < 1 - u/d$; ε is any arbitrary constant such that $0 \leq \varepsilon \leq \min\{2u, d-u\}$. This implementation employs a novel use of approximately synchronized clocks in order to utilize the lower bound on message delay time and achieve bounds on worst-case response times that depend on the message delay uncertainty u. For a wide range of values of u, these bounds improve upon previously known ones for implementations that support consistency conditions even weaker than linearizability.

Our next major result is a lower bound of $d + \min\{\delta, u\}/2$ on the sum of the worst-case response times for read and write operations, for the approximately synchronized clocks model. This bound applies to linearizable implementations possessing some natural symmetry properties; the bound is shown using the technique of "shifting" executions. Corresponding lower bounds, but with no symmetry assumptions, are shown on the individual worst-case response times for read and write operations.

Our bounds for the approximately synchronized clocks model extend naturally to the *imperfect clocks* model, where clocks may be arbitrarily far from each other, i.e., $\delta = \infty$.

1 Introduction

The shared-memory model has been proven a useful model of logically shared data in concurrent computation. Perhaps this is so because it allows processes to access local and remote information in a transparent and uniform way, which results in simplifying the programming of distributed applications. Thus, the shared-memory model is an attractive paradigm of an interprocessor communication model, as it provides the programmers the illusion of a global shared memory across distributed processes.

Shared-memory implementations must allow user programs to run "concurrently," i.e., to access shared data by interleaving steps or truly in parallel. Many such implementations have employed the technique of *caching,* i.e., maintaining multiple copies of the same logical piece of shared data; the performance of such implementations can be measured in terms of, e.g., the worst-case time to access a piece of data, availability of data to processes, or tolerance to process faults. Even in the simplest cases, however, problems arise since concurrent data accesses cannot be executed instantaneously, while their interleaving causes additional "correctness" problems.

Thus, a need arises for a *consistency mechanism* to support the illusion of atomic operations on single copies of memory objects. Such a mechanism may allow operations to be executed concurrently on multiple copies of objects but must still guarantee that the operations will appear as if executed atomically in some sequential order consistent with the order in which individual processes "observe" them to occur. If, in addition, this order is required to respect the order of non-overlapping operations at processes, the consistency mechanism is said to guarantee *linearizability* [32];* otherwise, it is said to guarantee *sequential consistency* [38]. Clearly, linearizability implies sequential consistency. It has been argued quite convincingly [32] that linearizability is the correctness condition that best guarantees "acceptable" concurrent behavior; indeed, linearizability enjoys a number of nice properties such as compositionality;† this has made it quite attractive for different applications, such as concurrent programming, multiprocessor operating systems, distributed file systems, etc., where concurrency is of primary interest.

Attiya and Welch [15] initiated a comparative study of the impact of the strength of correctness guarantees provided by sequential consistency and linearizability on the cost of supporting them. In more detail, they considered caching implementations of read/write objects in non-bused distributed systems; they took as cost measures the *worst-case response times* for performing read and write operations on such objects, and the sum of these times, in the best possible implementation supporting each of the consistency conditions. In this paper, we continue this study and present new lower and upper bounds on these costs for sequentially consistent and linearizable implementations. We attach some particular emphasis on the costs of supporting linearizability, since our motivation is to further illuminate the advantages

*Also called *atomicity* in [31, 39, 48] for the case of read/write objects.

†Roughly speaking, a consistency condition is said to be *compositional* if the system as a whole satisfies the condition whenever each individual object does.

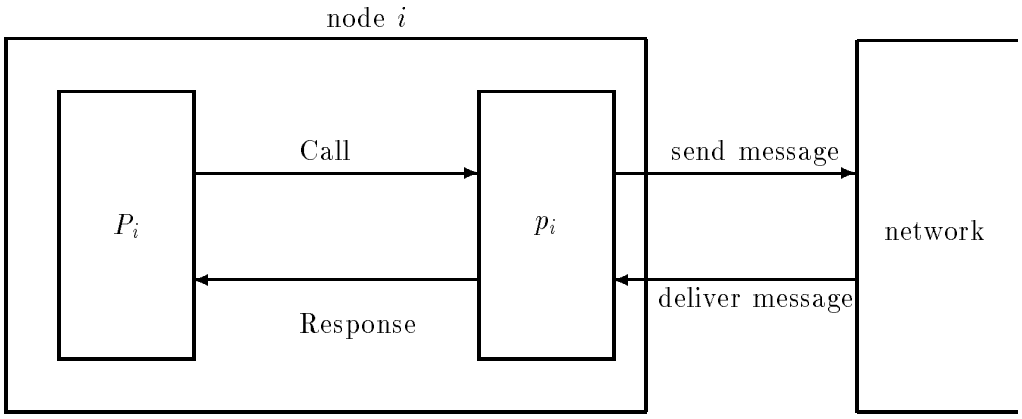


Figure 1: System Architecture

of linearizability over other, seemingly "cheaper," correctness conditions, such as sequential consistency. In particular, we are interested in understanding the dependence of the relation between linearizability and sequential consistency on timing assumptions made by different models of distributed computation.

We follow Attiya and Welch [15] and consider a model consisting of a collection of application programs running concurrently and communicating through virtual shared memory, which consists of a collection of read/write objects. These programs are running in a distributed system consisting of a collection of processes located at the nodes of a complete communication network.[‡] The shared memory abstraction is implemented by a *memory consistency system* (MCS), which uses local memory at each process node. Each MCS process executes a protocol, which defines the actions it takes on operation requests by the application programs. Specifically, each application program may submit requests to access shared data to a corresponding MCS process; the MCS process responds to such a request, based, possibly, on information from messages it receives from other MCS processes. In doing so, the MCS must, throughout the network, provide the proper read/write semantics with respect to the values returned to application programs. Figure 1 (directly adapted from [15, Section 2]) illustrates a node on which an application program and the corresponding MCS process are running. The model we consider captures characteristics of existing shared memory multiprocessor architectures, such as the *Reflective Memory System architecture* in the *Encore 91 Series* [23], which provides efficient coupling of multiple processor nodes for time-critical applications.

We make the following timing assumptions about the system. At each node, there is a real-time clock, readable by the MCS process at the node, which runs at the same rate as real time. It is assumed that the maximum difference between local times of any two processes in the system at the same real time is at most δ, for some *precision* constant $\delta \geq 0$; moreover, all message delays are in the range $[d-u, d]$, for some known constants u and d, $0 \leq u \leq d$.

[‡]The assumption of a complete communication network is made only for simplicity and can be removed.

It turns out that the timing information available in the system has a critical impact on the efficiency of implementing sequential consistency and linearizability.

We start with the *perfect clocks* model, where processes have perfectly synchronized clocks, i.e., $\delta = 0$, and message delays are constant, i.e., $u = 0$. We present a linearizable implementation, parameterized by some constant β, $0 \leq \beta \leq 1$; the worst-case response times for read and write operations are βd and $(1-\beta)d$, respectively, both dependent on the network's latency d; the parameter β precisely determines these dependencies and may be appropriately chosen in order to degrade the less frequently occurring operation. Roughly speaking, a read operation returns after time βd, while a write operation returns after time $(1-\beta)d$. This implementation naturally generalizes those in [15, Theorems 3.2 & 3.3], which are but the special cases with $\beta = 0$ and $\beta = 1$, respectively. Lipton and Sandberg [41] show a lower bound of d on the sum of the worst-case response times for read and write operations in any sequentially consistent implementation, and for any model assuming an upper bound of d on end-to-end message delay; thus, our implementation is optimal with respect to this measure.

We continue to present the first known linearizable implementation of read/write objects for the more realistic *approximately synchronized clocks* model, where clocks are only approximately synchronized, i.e., $\delta > 0$, and message delays can vary, i.e., $u > 0$. As for the case of the perfect clocks model, the worst-case response times achieved by our implementation are parameterized by a tunable constant β; this constant satisfies the mild restriction $0 \leq \beta < 1 - u/d$. More specifically, the worst-case response times for read and write operations are less than $\beta d + 3u + \min\{\delta, u\} + \varepsilon$ and $(1-\beta)d + 3u$, respectively; the constant $\varepsilon > 0$ is arbitrarily small and no more than $\min\{2u, d-u\}$. Roughly speaking, a read operation first waits for time βd; following this, it returns as soon as a value has resided for time at least u in the local memory of the corresponding MCS process. For a write operation, a "time-slicing" technique is used. Once it reaches an appropriate "time slice," the MCS process broadcasts the value to be written; following this, it waits for an additional time $(1-\beta)d$ before returning. Naturally, the specific details of the "time-slicing" technique directly or indirectly determine the worst-case response times for both write and read operations. However, a major ingredient of our implementation is that the value returned in a read operation need not be the one to which the local memory of the reading process was most recently updated; instead, the value to be returned is chosen among values of write operations on the same object performed by processes within a recent, small time interval. The specific choice is based on information shown to be shared by all MCS processes. This turns out to result not only in preserving the relative order of values returned by different reading processes, but also in maintaining consistent copies of local memory throughout the network; the latter result is shown to imply linearizability.

Our linearizable implementation for the approximately synchronized clocks model relies heavily on the provided finite clock precision in order to exploit the known lower bound of $d - u$ on message delay time and achieve better bounds on worst-case response times which, unlike previous ones, depend on the message delay uncertainty u. Although we assumed that this precision is a parameter of our model, in practice, it can be externally controlled by software protocols (see the many works on clock synchronization, e.g., [30, 36, 44], or [51] for a

survey). It is known that the externally achievable precision depends critically on the timing uncertainty inherent to the system. For the specific system model we consider, Lundelius and Lynch [44] have shown that $(1 - 1/n)u$ is the optimal achievable precision and provided a clock synchronization protocol achieving it. We present a significantly simpler protocol which achieves a precision of u that is only slightly inferior. This protocol only uses messages of constant size, in contrast to those in [44] that carry explicit timing information, and is of independent interest. Plugging in this precision of u, our bounds on the worst-case response times for read and write operations become $\beta d+4u+\varepsilon$ and $(1-\beta)d+3u$, respectively. In case the message delay uncertainty u is sufficiently small, these last bounds significantly improve those in [15] that correspond to an even weaker correctness condition, namely sequential consistency. (For a more detailed description of the results in [15], see Section 7.)

Moreover, we support optimality of our implementation for the approximately synchronized clocks model by presenting corresponding lower bounds under general and mild assumptions on the pattern of sharing properties of processes. Our main negative result is a lower bound of $d + \min\{\delta, u\}/2$ on the sum of the worst-case response times for any sequentially consistent implementation in which processes handle operations on each object identically and independently of operations on other objects. This implies a corresponding lower bound for linearizable implementations. We also show lower bounds of $\min\{\delta, u\}/2$ on the individual worst-case response times for read and write operations, in any linearizable implementation. For the case where $u \leq \delta$, th lower bound for the read operation improves on a result of Attiya and Welch [15] showing a lower bound of $u/4$. Our lower bounds are shown using the technique of *shifting* executions, introduced in [44] for showing a lower bound on the precision achievable by clock synchronization algorithms.

The dependence on d of the upper bounds achieved by our implementation for the approximately synchronized clocks model is minimal: the sum of the worst-case response times for read and write operations contains only a single additive term of d, which, by our lower bound, is inherent. Furthermore, although the analysis of our implementation is technically challenging, the implementation itself is fairly simple, it does not use complicated control mechanisms, and it is message-economical. It can be also considered as a natural generalization of the one for the perfect clocks model with $\beta = 0$, since, as u tends to 0, it almost "coincides" with it and achieves almost identical worst-case response times.

Our result for the approximately synchronized clocks model, in particular, the upper bound of $d + O(u)$ on the sum of the worst-case response times for read and write operations in a linearizable implementation, along with the lower bound of $d + O(\min\{\delta, u\})$ on this sum, may suggest that sequential consistency and linearizability are actually "closer" than thought before in the specific system models we consider. All of these, even the imperfect clocks model, assume that all processor clocks move at exactly the same speed and that there is a known bound on message delays. Given that the primary difference between sequential consistency and linearizability is with respect to timing, it is perhaps not too surprising that the two concepts would tend to converge in models with strong synchrony. These bounds imply that it is more cost-effective to support linearizability in systems with low message delay uncertainty.

The rest of the paper is organized as follows. Section 2 presents our formal definitions, and surveys some preliminary facts and related background. Bounds for the perfect clocks model are included in Section 3. Sections 4 and 5 contain our upper and lower bounds, respectively, for the approximately synchronized clocks model. Bounds for the related imperfect clocks model are stated in Section 6. We conclude, in Section 7, with a discussion of our results, a survey of related work, and some open problems.

2 Definitions, Preliminaries and Background

In this section, we present the formal system model and its various timing aspects; we also introduce the memory objects, the consistency conditions, and the costs of their message-passing implementations. Towards the end, we review the shifting technique. Our definitions are patterned after those in [15], which they somehow refine and extend.

For any real vector $\vec{s}$, denote $\|\vec{s}\|_{\infty}$ and $\|\vec{s}\|_{-\infty}$ the maximum and minimum, respectively, entries of $\vec{s}$.

2.1 System Model

We consider a collection of *application programs* running concurrently and communicating through virtual shared memory; the latter consists of a collection $\mathcal{X}$ of *read/write objects,* or *objects* for short. Each object $X \in \mathcal{X}$ attains values from a *domain,* a set $\mathcal{V}$ of *values* that includes a special "undefined" value $\perp$; a total order $<_{\mathcal{V}}$ is defined on $\mathcal{V}$. We assume a system consisting of a collection N of *nodes,* connected via a *communication network*; take $|N| = n$.

The shared memory abstraction is implemented by a *memory consistency system (MCS),* consisting of a collection of MCS processes, one at each node; these processes use local memory, execute some local protocol, and communicate through exchanging *messages,* drawn from some message alphabet $\mathtt{M}$, along the network. Each MCS process p_i, located at node i, is associated with an application program P_i; p_i and P_i interact by using *call* and *response* events. Formally, the following *external events* may occur at the MCS process p_i.

- *Call events:* They represent initiation of operations by the application program P_i; they are $\mathsf{Read}_i(X)$ and $\mathsf{Write}_i(X, v)$, for all objects $X \in \mathcal{X}$ and values $v \in \mathcal{V}$.
- *Response events:* They represent responses by p_i to operations initiated by the application program P_i; they are $\mathsf{Return}_i(X, v)$ and $\mathsf{Ack}_i(X)$, for all objects $X \in \mathcal{X}$ and values $v \in \mathcal{V}$.
- *Message-send events:* They represent sending of a message by p_i to any other MCS process; they are $\mathsf{Send}_i(\mathtt{m}, j)$ for all messages $\mathtt{m} \in \mathtt{M}$ and MCS processes p_j, $j \neq i$.
- *Message-deliver events:* They represent delivery of a message from any other MCS process to p_i; they are $\mathsf{Del}_i(\mathtt{m}, j)$, for all messages $\mathtt{m} \in \mathtt{M}$ and MCS processes p_j, $j \neq i$.

For each index i, $1 \leq i \leq n$, there is a physical, real-time clock at node i, readable by MCS process p_i but not under its control, that runs at the same rate as real time. Formally, the *local clock* of process p_i, denoted γ_i, is a monotonically increasing function from $\Re$ (*real time*) to $\Re$ (*clock time*) of the form $\gamma_i(t) = t + g_i$; g_i is a real number called the *local clock parameter* of p_i.[§] (The local clock parameters are fixed for each "run" of the system, but they are unknown to the processes.) The local clocks at various nodes may be initially "out-of-phase"; this happens whenever $g_i \neq g_j$ for any process indices i and j. Moreover, the local clocks cannot be modified by the processes.

Processes do not have access to real time; instead, each process obtains its only information about (real) time from its local clock. The local clock reliably measures how much real time has elapsed, although its actual value is not equal to real time. Moreover, process p_i may use its local clock for "timing" itself. Formally, this is done through the following *internal events:*

- *Timer-set events:* They represent setting of a timer by p_i to "go off" after a specified amount of local clock time elapses and return a message; they are $\mathsf{TimerSet}_i(T, \mathtt{m})$ for all real numbers $T > 0$ and messages $\mathtt{m} \in \mathtt{M}$.
- *Timer-expire events:* They represent a timer expiration returning a message at p_i; they are $\mathsf{TimerExpire}_i(\mathtt{m})$ for all messages $\mathtt{m} \in \mathtt{M}$.

The call, message-deliver, and timer-expire events are called *interrupt events*; the response, message-send, and timer-set events are called *react* events.

Each MCS process p_i is modeled as a *state machine* with a (possibly infinite) set of *states,* including an *initial state,* and a *transition function.* Each interrupt event at MCS process p_i causes an application of its transition function; thus, computations of the system are "interrupt-driven". More specifically, the transition function is a function from tuples of a state, a local clock time, and an interrupt event to tuples of a state and sets of react events; in more detail, the transition function takes as input the current state, the local clock time, and an interrupt event, and returns a new state, a set of response events to the corresponding application program, a set of messages to be sent to other MCS processes, and a set of timer-set events. Formally, a *computation step* of process p_i is a pair of tuples $(\langle q, \gamma, \mathsf{i} \rangle, \langle q', \mathcal{R}, \mathcal{S}, \mathcal{T} \rangle)$, where q and q' are states, γ is a real number, called the *local clock time,* i is an interrupt event, $\mathcal{R}$ is a set of response events, $\mathcal{S}$ is a set of message-send events, and $\mathcal{T}$ is a set of timer-set events, so that q', $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ result from the application of p_i's transition function on q, γ and i.

A *history for MCS process p_i with clock γ_i* is a mapping h_i from $\Re$ (real time) to finite sequences of computation steps by p_i such that:

1. for each real time t, there is only a finite number of (real) times $t' < t$ such that the corresponding sequence of computation steps $h_i(t')$ is non-empty; thus, the concatenation of all such sequences in real-time order is also a sequence, called the *history sequence*;

[§] Although it is possible to make the local clock of each process a part of its (local) state, which we will soon introduce, we chose to keep local clocks separate from states so that we would not need to put restrictions on how those parts of states may be modified.

2. the old state for the first computation step in the history sequence is p_i's initial state;

3. the old state for each subsequent computation step is the new state for the previous computation step in the history sequence;

4. for each real time t, the local clock time of every computation step in the sequence $h_i(t)$ is equal to $\gamma_i(t)$;

5. for each real time t, there is at most one computation step whose interrupt event is a timer-set event, and this step is ordered last in the sequence $h_i(t)$;

6. there is a one-to-one correspondence between timer-set and timer-expire events appearing in computation steps of the history sequence; moreover, each timer-expire event occurs at local clock time T later than the corresponding timer-set event, where T is the real number specified in the timer-set event;

7. at most one call event at p_i is "pending" at a time;[¶]

8. there is a one-to-one correspondence between call and response events appearing in computation steps of the history sequence. For each call event, the corresponding response event appears later in the history sequence; moreover, for each call event $\mathsf{Read}_i(X)$, the corresponding response event is an event $\mathsf{Return}_i(X, v)$ for some value $v \in \mathcal{V}$, while for each call event $\mathsf{Write}_i(X, v)$, the corresponding response event is an event $\mathsf{Ack}_i(X)$.

Each pair of matching call and response events forms an *operation*. The call event marks the start of the operation, while the response event marks its end. An operation *op* is *invoked* when the application program issues the appropriate call event for *op*; *op terminates* when the MCS process issues the appropriate response for *op*.

For a given MCS, an *execution* σ is a tuple of histories $\langle h_1, h_2, \ldots, h_n \rangle$, one for each MCS process p_l with a corresponding local clock γ_l, such that for any pair of MCS processes p_i and p_j, there is a one-to-one correspondence between the messages sent by p_i to p_j, and those delivered at p_j that were sent by p_i. Use this message correspondence to define the *delay* of any message in the execution σ to be the real time of delivery minus the real time of sending. Execution σ is *admissible* if every message in σ incurs a delay in the range $(d - u, d]$, for some fixed and known constants d and u, $0 \leq u < d$; d is the *message delay latency,* while u is the *message delay uncertainty.*

2.2 Timing Assumptions and Clock Synchronization

Fix a (known) constant δ, called *clock precision,* such that $0 \leq \delta \leq \infty$. Say that an execution σ is a *δ-execution* if for all pairs of MCS processes p_i and p_j and all real times t, $|\gamma_i(t) - \gamma_j(t)| \leq \delta$;

[¶] This outlaws pipelining or prefetching at the interface between an application program and the corresponding MCS process.

notice that, by definition of local clocks, this happens if and only if $|g_i - g_j| \leq \delta$. In particular, a 0-execution will be called an *in-phase* execution.

The *inverse local clock* of process p_i, denoted γ_i^{-1}, is the inverse function of p_i's local clock. By definition of local clock, the inverse clock is a monotonically increasing function from $\Re$ (local clock time) to $\Re$ (real time) of the form $\gamma_i^{-1}(c) = c - g_i$; hence, for any pair of MCS processes p_i and p_j, for all real times t and local clock times c, $\gamma_i^{-1}(c) - \gamma_j^{-1}(c) = g_j - g_i = (t - g_i) - (t - g_j) = \gamma_i(t) - \gamma_j(t)$. Hence, it follows:

Proposition 2.1 *Fix any δ-execution. Then, for any pair of MCS processes p_i and p_j, and for all local clock times c,*

$$|\gamma_i^{-1}(c) - \gamma_j^{-1}(c)| \quad \leq \quad \delta\,.$$

The next simple claim relates the difference between local clock times at which message-send events occur in a δ-execution, with the difference between real times at which corresponding message-deliver events occur in the same execution.

Lemma 2.2 *Consider message-send events $\mathsf{Send}_{i_1}(\mathtt{m}_1, j_1)$ and $\mathsf{Send}_{i_2}(\mathtt{m}_2, j_2)$ in a δ-execution σ, occurring at (real) times t_1 and t_2, respectively. Let $\mathsf{Del}_{j_1}(\mathtt{m}_1, i_1)$ and $\mathsf{Del}_{j_2}(\mathtt{m}_2, i_2)$ be the corresponding message-deliver events occurring at (real) times t_1' and t_2', respectively, in σ. Assume that $\gamma_{i_2}(t_2) - \gamma_{i_1}(t_1) > \delta'$. Then, $t_2' - t_1' > \delta' - \delta - u$.*

Proof: Clearly,

$$\begin{array}{rcll} \gamma_{i_2}(t_2) - \gamma_{i_1}(t_1) & = & t_2 + g_{i_2} - (t_1 + g_{i_1}) & \text{(by definition of local clocks)} \\ & = & g_{i_2} - g_{i_1} + t_2 - t_1 & \\ & \leq & \delta + t_2 - t_1 & \text{(since } \sigma \text{ is a } \delta\text{-execution)}\,; \end{array}$$

thus,

$$\begin{array}{rcll} t_2 - t_1 & \geq & \gamma_{i_2}(t_2) - \gamma_{i_1}(t_1) - \delta & \\ & > & \delta' - \delta & \text{(by assumption)}\,. \end{array}$$

Since σ is admissible, $t_2' \geq t_2 + d - u$ and $t_1' \leq t_1 + d$, so that

$$\begin{array}{rcl} t_2' - t_1' & \geq & t_2 + d - u - (t_1 + d) \\ & = & t_2 - t_1 - u \\ & > & \delta' - \delta - u\,, \end{array}$$

as needed. ■

Transition Relation:

Pre:	—	*Pre:*	$\mathsf{Del}_i(\mathtt{synch})$
Eff:	$\mathsf{TimerSet}_i(d, \mathtt{synch})$	*Eff:*	$Corr_i \leftarrow -\gamma_i$
	$\mathsf{Broadcast}_i(\mathtt{synch})$		
Pre:	$\mathsf{TimerExpire}_i(\mathtt{synch})$		
Eff:	$Corr_i \leftarrow -\gamma_i$		

Figure 2: The algorithm $\mathcal{A}^{synch}$: precondition-effect code for process p_i

Although we shall treat the clock precision δ as a fixed parameter, it is possible to have each process obtain a *logical clock* $\widehat{\gamma}_i$ that is "closer" to those of other processes by computing an *additive* "software" correction to its local clock time through a *clock synchronization algorithm (CSA)*; see, e.g., [30, 36, 44, 51] or [16, Section 6.3]. Say that a CSA *achieves clock precision* Δ if the maximum difference between the logical clock times of any two processes at any real time after all processes have terminated executing the algorithm is at most Δ. There are, however, known limitations on the best achievable clock precision, as a function of the number of processes n and the message delay uncertainty u.

Proposition 2.3 (Lundelius and Lynch [44]) *No CSA achieves clock precision less than* $(1 - 1/n)u$.

We proceed to present a simple clock synchronization algorithm $\mathcal{A}^{synch}$ that achieves clock precision u. We start with an informal description of $\mathcal{A}^{synch}$. Each process p_i broadcasts a special synchronization message `synch`, and sets a timer for time d thereafter. On either the first receipt of some `synch` message from some other process, or on expiration of its own timer, whichever happens first, p_i sets its (logical) clock time to 0. In more detail, if first receipt or expiration occurs at (real) time t, p_i adopts an additive correction of $-\gamma_i(t)$ to its local clock, which results in vanishing its logical clock time at time t. In all future discussion, we will use local clock time to refer to logical clock time. Figure 2 presents the code for process p_i in a precondition-effect style that is commonly used to describe I/O automata [45]. We show:

Proposition 2.4 $\mathcal{A}^{synch}$ *achieves clock precision* u.

Proof: Fix any admissible execution σ. For each process p_l, let t_l be the minimum among all (real) times t such that either $\mathsf{TimerExpire}_l(\mathtt{synch})$ or $\mathsf{Del}_l(\mathtt{synch})$ occurs at time t. Denote $t_{max} = \max_{l \in [n]} t_l$; thus, t_{max} is the time at which the last process completes the execution of $\mathcal{A}^{synch}$. Let p_i be the last process to complete the execution of $\mathcal{A}^{synch}$ so that $t_{max} = t_i$. We start by showing:

Lemma 2.5 *For any process* p_l, $\mathsf{Broadcast}_l(\mathtt{synch})$ *occurs no earlier than time* $t_{max} - d$.

Proof: Assume, by way of contradiction, that for some process p_l, $\mathsf{Broadcast}_l(\mathtt{synch})$ occurs at real time less than $t_{max} - d$ in σ. Since σ is admissible, $\mathsf{Del}_i(\mathtt{synch})$ occurs at real time less than $t_{max} - d + d = t_{max}$. Thus, $t_i < t_{max}$. A contradiction. ■

We continue to show:

Lemma 2.6 *For any process p_l, $\mathsf{Del}_l(\mathtt{synch})$ occurs no earlier than time $t_{max} - u$.*

Proof: Since σ is admissible, $\mathsf{Del}_l(\mathtt{synch})$ occurs at real time which is at least $d-u$ later than the real time at which a broadcast event occurs. By Lemma 2.5, it follows that $\mathsf{Del}_l(\mathtt{synch})$ occurs no earlier than time $t_{max} - d + d - u = t_{max} - u$, as needed. ■

We finally show:

Lemma 2.7 *For any process p_l, $\mathsf{TimerExpire}_l(\mathtt{synch})$ occurs no earlier than time t_{max}.*

Proof: By the algorithm, $\mathsf{TimerSet}_l(d, \mathtt{synch})$ and $\mathsf{Broadcast}_l(\mathtt{synch})$ occur at the same real time. Thus, by Lemma 2.5, $\mathsf{TimerSet}_l(d, \mathtt{synch})$ occurs no earlier than time $t_{max} - d$. It follows that $\mathsf{TimerExpire}_l(\mathtt{synch})$ occurs no earlier than time $t_{max} - d + d = t_{max}$, as needed. ■

Consider any process p_l. If p_l completes the execution of $\mathcal{A}^{synch}$ on $\mathsf{Del}_l(\mathtt{synch})$, then, by Lemma 2.6 and definition of t_{max}, $t_{max} - u \leq t_l \leq t_{max}$. If p_l completes the execution of $\mathcal{A}^{synch}$ on $\mathsf{TimerExpire}_l(\mathtt{synch})$, then, by Lemma 2.7 and definition of t_{max}, $t_{max} \leq t_l \leq t_{max}$, so that $t_l = t_{max}$. This implies:

Lemma 2.8 *For any process p_l, $t_{max} - u \leq t_l \leq t_{max}$.*

Consider any real time $t \geq t_{max}$, and any pair of processes p_j and p_k, $j \neq k$. Clearly,

$$\begin{aligned} \widehat{\gamma}_i(t) - \widehat{\gamma}_j(t) &= t - t_i - (t - t_j) \\ &\quad \text{(by the algorithm)} \\ &= t_j - t_i \,. \end{aligned}$$

By Lemma 2.8, $t_{max} - u \leq t_i \leq t_{max}$ and $t_{max} - u \leq t_j \leq t_{max}$, so that $|t_j - t_i| \leq u$. Thus, $|\widehat{\gamma}_i(t) - \widehat{\gamma}_j(t)| \leq u$. It follows that $\mathcal{A}^{synch}$ achieves clock precision u, as needed. ■

We remark that Lundelius and Lynch [44] have shown that clock precision of $(1 - 1/n)u$ is indeed achievable, which is slightly better than u, achieved in Proposition 2.4. Lundelius and Lynch [44, Section 4] present a clock synchronization algorithm carrying explicit timing information, i.e., local clock values, in all messages exchanged between processes; by that algorithm,

each process needs also to "count" the number of messages it receives from other processes. In contrast, neither timing information is carried in messages sent by our clock synchronization algorithm, which are of constant size, nor processes need to "count". In these respects, our clock synchronization algorithm is more efficient in both message size and space overhead than the one of Lundelius and Lynch. Thus, we choose to use our own clock synchronization algorithm in some of our later algorithms in order to keep those correspondingly efficient as well.

In the *perfect clocks* model, MCS processes have perfectly synchronized (*perfect*) clocks, i.e., $\delta = 0$. This is modeled by assuming that for each MCS process p_i, $\gamma_i(t) = t$. Attiya and Welch [15] note that the assumption of perfect clocks is equivalent to the assumption of constant (and known) message delays, which, in our formal model, can be modeled by assuming $u = 0$. If clocks are perfect and there is a constant and known upper bound d on message delay, then constant message delays can be simulated by time-stamping each message with the local clock time of the sender at sending time, and having each recipient delay any message that arrives with a delay smaller than d until the delay is exactly d. If the message delay is constant and known, then a simple clock synchronization algorithm can synchronize the clocks perfectly; each message is time-stamped with the local clock time of the sender at sending time, which allows the recipient to exactly synchronize its local clock to that of the sender.

In the more realistic *approximately synchronized clocks* model, MCS processes have local clocks with *finite* clock precision; that is, $0 \leq \delta < \infty$. Proposition 2.4 implies that we can assume a clock precision of $\min\{\delta, u\}$ for all δ-executions in the approximately synchronized clocks model.

In the *imperfect clocks* model, clocks may be arbitrarily far from each other, i.e., $\delta = \infty$. Proposition 2.4 implies that we can assume a clock precision of $\min\{\infty, u\} = u$ for all executions in the imperfect clocks model.

2.3 Memory Objects

Each object X has a *serial specification* [32] which describes its behavior in the absence of concurrency and failures. Formally, it defines.

- A set $OP(X)$ of *operations on* X, which are ordered pairs of *call* and *response* events. Each operation $op \in OP(X)$ has a value $val(op)$ associated with it.
- A set of *legal operation sequences for* X, which are the allowable sequences of operations on X.

The set $OP(X)$ contains a *read* operation $[\mathsf{Read}_i(X), \mathsf{Return}_i(X, v)]$ on X, and a *write* operation $[\mathsf{Write}_i(X, v), \mathsf{Ack}_i(X)]$ on X, for each index $i \in [n]$ and value $v \in \mathcal{V}$; v is the value associated with each of these operations. The set of legal operation sequences for X contains all sequences of operations on X for which, for any read operation *rop* in the sequence, either

$val(rop) = \perp$ and there is no preceding write operation in the sequence, or $val(rop) = val(wop)$, where *wop* is the latest preceding write operation. Thus, each legal operation sequence obeys the usual read/write semantics: every read operation on X returns the value of the latest preceding write operation on X, if there is one, or, otherwise, an "undefined" value.

Let τ be a sequence of operations. Denote by $\tau \mid i$ the restriction of τ to operations at the MCS process p_i; similarly, denote by $\tau \mid X$ the restriction of τ to operations on the object X. A sequence of operations τ for a collection of processes and objects is *legal* if, for every object $X \in \mathcal{X}$, $\tau \mid X$, is in the set of legal operation sequences for X.

We often speak informally of an operation on an object as in "the read operation on the object X". An operation in our formal model is intended to represent a single "execution" of an operation as used in the informal sense.

2.4 Correctness Conditions

Correctness conditions are specified at the interface between the application programs (written by the users), and the MCS processes (supplied by the system).

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order, breaking ties for each real time t as follows. First, order all response events whose matching call events occur before time t, using process identification numbers (*id's*) to break any remaining ties. Then, order all operations whose call and response events both occur at time t. Preserve the relative ordering of operations for each process, and break any remaining ties using process id's. Finally, order all call events whose matching response events occur after time t, using process id's to break any remaining ties. For an execution σ, the definitions of $\tau \mid i$ and $\tau \mid X$ can be extended in the natural way to yield $ops(\sigma) \mid i$ and $ops(\sigma) \mid X$, respectively.

An execution σ specifies a partial order $\stackrel{\sigma}{\longrightarrow}$ on the operations appearing in σ as follows. For any operations op_1 and op_2 appearing in σ, $op_1 \stackrel{\sigma}{\longrightarrow} op_2$ if the response for op_1 precedes the call for op_2 in $ops(\sigma)$; that is, $op_1 \stackrel{\sigma}{\longrightarrow} op_2$ if op_1 completely precedes op_2 in $ops(\sigma)$.

Given an execution σ, an operation sequence τ is a *serialization* of σ if it is a permutation of $ops(\sigma)$. A serialization τ of σ is a *linearization* of σ if it extends $\stackrel{\sigma}{\longrightarrow}$; that is, if $op_1 \stackrel{\sigma}{\longrightarrow} op_2$, then $op_1 \stackrel{\tau}{\longrightarrow} op_2$. Roughly speaking, the definitions for sequential consistency and linearizability involve, for each execution σ, the existence of a serialization τ of σ that possesses certain properties. The formal definitions for sequential consistency and linearizability follow.

Definition 2.1 (Sequential Consistency, Lamport [38]) *An execution σ is* sequentially consistent *if there exists a legal serialization τ of σ such that for each MCS process p_l, $ops(\sigma) \mid l = \tau \mid l$.*

Definition 2.2 (Linearizability, Herlihy and Wing [32]) *An execution σ is* linearizable *if there exists a legal linearization τ of σ such that for each MCS process p_l, $ops(\sigma) \mid l = \tau \mid l$.*

Intuitively, σ is sequentially consistent if the sequence of operations in σ can be permuted to yield an operation sequence τ that is legal and maintains the order of call and response events seen at each process; if, in addition, τ preserves the order of any two non-overlapping operations in σ, σ is said to be linearizable.[‖]

An MCS is a *sequentially consistent implementation* of $\mathcal{X}$ if every admissible execution of the MCS is sequentially consistent; similarly, an MCS is a *linearizable implementation* of $\mathcal{X}$ if every admissible execution of the MCS is linearizable.

A correctness condition is *compositional* (or local) [32] if the combination of memory objects each of which individually satisfies the condition yields an implementation that satisfies the condition as well. An important distinction holds between sequential consistency and linearizability with respect to compositionality.

Proposition 2.9 (Herlihy and Wing [32]) *Linearizability is local; sequential consistency is not.*

Proposition 2.9(ii) implies that to give a linearizable implementation of $\mathcal{X}$, it suffices to give a linearizable implementation of a *single* object $X \in \mathcal{X}$. In contrast, for sequential consistency, all objects must be implemented together. (This causes development costs to increase and makes it hard to apply separate optimizations to different objects; see [32] for an expanded discussion.)

2.5 Cost Measures

In general, the efficiency of an implementation $\mathcal{A}$ of $\mathcal{X}$ is measured by the *worst-case response time* for any operation on an object $X \in \mathcal{X}$. Given a particular MCS $\mathcal{A}$ and a read/write object X implemented by it, the time $|op_{\mathcal{A}}(X,\sigma)|(\delta)$ taken by an operation op on X in an admissible δ-execution σ of $\mathcal{A}$ is the maximum difference between the times at which the response and call events of op occur in σ, where the maximum is taken over all occurrences of op in σ. In particular, we denote by $|\mathbf{R}_{\mathcal{A}}(X,\sigma)|(\delta)$ and $|\mathbf{W}_{\mathcal{A}}(X,\sigma)|(\delta)$ the maximum time taken by a read and a write operation, respectively, on X in σ, where the maximum is taken over all occurrences of the corresponding operations in σ.

Define $|\mathbf{R}_{\mathcal{A}}(X)|(\delta)$ (resp., $|\mathbf{W}_{\mathcal{A}}(X)|(\delta)$) as the maximum of $|\mathbf{R}_{\mathcal{A}}(X,\sigma)|$ (resp., $|\mathbf{W}_{\mathcal{A}}(X,\sigma)|$) over all δ-executions σ of $\mathcal{A}$. Define $|\mathbf{R}_{\mathcal{A}}|(\delta)$ (resp., $|\mathbf{W}_{\mathcal{A}}|(\delta)$) as the maximum of $|\mathbf{R}_{\mathcal{A}}(X)|(\delta)$ (resp., $|\mathbf{W}_{\mathcal{A}}(X)|(\delta)$), over all read/write objects X implemented by the MCS $\mathcal{A}$. Let also $|\mathbf{R}|(\delta)$ and $|\mathbf{W}|(\delta)$ denote the minimum, over all implementations $\mathcal{A}$ of $\mathcal{X}$, of $|\mathbf{R}_{\mathcal{A}}|(\delta)$ and $|\mathbf{W}_{\mathcal{A}}|(\delta)$, respectively.

[‖] Linearizability may be viewed as a special case of *strict serializability* (see, e.g., [18, 49]), a basic correctness condition for concurrent computations on databases, where transactions are restricted to appear to be a single operation on a single object.

Finally, let $|\mathbf{R}|$ and $|\mathbf{W}|$ be the minimum of $|\mathbf{R}|(\delta)$ and $|\mathbf{W}|(\delta)$, respectively, over all achievable precisions δ. It follows from Theorem 2.3 that $|\mathbf{R}| \geq |\mathbf{R}|((1-1/n)u)$ and $|\mathbf{W}| \geq |\mathbf{R}|((1-1/n)u)$. The sum $|\mathbf{R}| + |\mathbf{W}|$ is also considered as a measure of efficiency.

2.6 Shifting Executions and Clocks

Our presentation closely follows a corresponding one in [15].

In our later proofs of lower bounds (Section 5), we use the technique of *shifting*, originally introduced by Lundelius and Lynch [44] to prove lower bounds on the clock precision achievable by clock synchronization algorithms. Shifting is used to change the timing and the ordering of events in an execution of the system, while preserving the "local views" of the processes.

Roughly speaking, given an execution, if for each process p_i, p_i's history is changed so that the real times at which the events at p_i occur are shifted by some amount, and if p_i's clock is shifted by the same amount, then the result is another execution in which every process still "sees" the same events happening at the same local clock time. The intuition is that the changes in the real times at which events at a process occur cannot be detected by the process because its clock has changed by a corresponding amount.

More precisely, the *view of process* p_i *in execution* $\sigma = \{h_1, h_2, \ldots, h_n\}$, denoted $view_l(\sigma)$, is the history sequence defined by the history h_i in σ. Note that the real times of occurrences of events at p_l are *not* represented in the view of p_l.

Say that executions σ_1 and σ_2 are *equivalent* if, for each MCS process p_l, $view_l(\sigma_1) = view_l(\sigma_2)$. Intuitively, equivalent executions are indistinguishable to the processes; only an "outside observer" with access to real time can tell them apart.

Given a history h_i of MCS process p_i with clock γ_i and a real number s, a new history $h_i' = shift(h_i, s)$ is defined by $h_i'(t) = h_i(t+s)$ for all real times t. That is, all sequences of computation steps are shifted earlier in h_i' by s if s is positive, and later by $-s$ if s is negative. Given a clock γ_i for MCS process p_i and a real number s, a new clock $\gamma_i' = shift(\gamma_i, s)$ is defined by $\gamma_i'(t) = \gamma_i(t) + s$ for all real times t. That is, the clock is shifted forward by s if s is positive, and backward by $-s$ if s is negative. The following claim observes that simultaneously shifting a process's history and clock by the same amount yields another process history.

Lemma 2.10 *Let h_i be a history of MCS process p_i with clock γ_i, and let s be a real number. Then, $shift(h_i, s)$ is a history of p_i with clock $shift(\gamma_i, s)$.*

Given an execution σ and a real vector $\vec{s} = \langle s_1, s_2, \ldots, s_n \rangle$, a new execution $\sigma' = shift(\sigma, \vec{s})$ is defined by replacing, for each MCS process p_i, the history h_i of p_i in σ by (the history) $shift(h_i, s_i)$, while retaining the same correspondence between sent and delivered messages. (Technically, the correspondence is redefined so that a pairing in σ that involves a message-send or message-deliver event for an MCS process p_i at time t, it involves, in σ', the event for p_i occurring at time $t - s_i$.)

Given a tuple of clocks $\Gamma = \{\gamma_i \mid 1 \leq i \leq n\}$, and a real vector $\vec{s} = \langle s_1, s_2, \ldots, s_n \rangle^{\mathrm{T}}$, a new tuple of clocks $\Gamma' = \mathit{shift}(\Gamma, \vec{s})$ is defined by replacing, for each MCS process p_i, local clock γ_i by local clock $\mathit{shift}(\gamma_i, s_i)$.

The following claim observes that shifting each process's history and clock by the same amount in an execution yields another execution that is equivalent to the original.

Lemma 2.11 (Lundelius and Lynch [44]) *Let σ be an execution with clocks Γ, and consider any real vector $\vec{s}$. Then, $\mathit{shift}(\sigma, \vec{s})$ is an execution with clocks $\mathit{shift}(\Gamma, \vec{s})$ that is equivalent to σ with clocks Γ.*

The following claim quantifies how message delays change when an execution is shifted.

Lemma 2.12 (Lundelius and Lynch [44]) *Let $\vec{s}$ be a real vector. For any pair of MCS processes p_i and p_j, if the delay of a message $\mathtt{m}$ from p_i to p_j in the execution σ with clocks Γ is equal to Δ, then the delay of $\mathtt{m}$ in the execution $\mathit{shift}(\sigma, \vec{s})$ is equal to $\Delta + s_l - s_m$.*

Lemma 2.12 implies that the result of shifting an admissible execution is not necessarily admissible. The next simple claim precisely determines the change in clock precision due to shifting an execution.

Lemma 2.13 *Assume σ is a Δ-execution with clocks Γ. Then, for any real vector $\vec{s}$, the execution $\mathit{shift}(\sigma, \vec{s})$ with clocks $\Gamma' = \mathit{shift}(\Gamma, \vec{s})$ is a $(\Delta + |\|\vec{s}\|_\infty - \|\vec{s}\|_{-\infty}|)$-execution.*

Proof: Clearly, for any MCS processes p_i and p_j and real time t,

$$\begin{aligned}
|\gamma_i'(t) - \gamma_j'(t)| &= |\gamma_i(t) + s_i - (\gamma_j(t) + s_j)| \\
&\leq |\gamma_i(t) - \gamma_j(t)| + |\gamma_i - \gamma_j| && \text{(by triangle inequality)} \\
&\leq \Delta + |s_i - s_j| && \text{(since } \sigma \text{ is a } \Delta\text{-execution)} \\
&\leq \Delta + |\|\vec{s}\|_\infty - \|\vec{s}\|_{-\infty}| \,,
\end{aligned}$$

which implies that the execution $\mathit{shift}(\sigma, \vec{s})$ with clocks $\Gamma' = \mathit{shift}(\Gamma, \vec{s})$ is a $(\Delta + |\|\vec{s}\|_\infty - \|\vec{s}\|_{-\infty}|)$-execution, as needed. ■

2.7 Notation

In this section, we introduce some notation that will be used in the sequel. Consider any execution σ, and let $op = [\mathsf{Call}(op), \mathsf{Response}(op)]$ be any operation in σ. We denote by $t_c^{(\sigma)}(op)$ and $t_r^{(\sigma)}(op)$ the (real) times at which $\mathsf{Call}(op)$ and $\mathsf{Response}(op)$, respectively, occur in σ. When σ is not clear from context, we use $val^{(\sigma)}(op)$ to denote the value associated with the "execution" of operation op in σ.

For any real numbers x_1 and x_2, $x_1 \geq 0$ and $x_2 > 0$, $\mathrm{fmod}(x_1, x_2)$ denotes the remainder of the division of x_1 by x_2, i.e., $\mathrm{fmod}(x_1, x_2) = x_1 - \lfloor x_1/x_2 \rfloor$. For a real interval $I = [i_1, i_2]$, $\lfloor I \rfloor = i_1$ and $\lceil I \rceil = i_2$; the length $i_2 - i_1$ of I is denoted by $|I|$.

For any index i and message $\mathtt{m} \in \mathcal{M}$, we use $\mathsf{Broadcast}_i(\mathtt{m})$ to denote the set of message-send events $\{\mathsf{Send}_i(\mathtt{m}, j) : j \in [n]\}$.

3 Perfect Clocks

In this section, we consider the perfect clocks model, where $\delta = 0$ and $u = 0$. We show:

Theorem 3.1 *For the prefect clocks model, there exists a linearizable implementation $\mathcal{A}^{per}$ of read/write objects such that $|\mathbf{R}_{\mathcal{A}^{per}}|(0) = \beta d$, and $|\mathbf{W}_{\mathcal{A}^{per}}|(0) = (1-\beta)d$, for any constant β, $0 \leq \beta \leq 1$.*

By Proposition 2.9(ii), it suffices to provide an implementation of a single object $X \in \mathcal{X}$. In Section 3.1, we describe the implementation $\mathcal{A}^{per}$, while a correctness proof and complexity analysis for $\mathcal{A}^{per}$ are presented in Sections 3.2 and 3.3, respectively.

3.1 The Algorithm

We start with an informal description of $\mathcal{A}^{per}$. Each process p_i keeps a local copy X_i of object X; denote $val(X_i)$ the value currently held by X_i, initially $\perp$. Upon a $\mathsf{Read}_i(X)$ event, p_i waits for time βd and issues $\mathsf{Return}_i(X, val(X_i))$. Upon a $\mathsf{Write}_i(X, v)$ event, p_i sends update messages $\mathtt{update}(X, v)$ to all other processes; after time $(1-\beta)d$ passes, p_i issues $\mathsf{Ack}_i(X)$ and waits for an additional time of βd to set X_i to v. Furthermore, upon receipt of an update message for X from another process, p_i immediately updates X_i to the value being written.**

We remark that $\mathcal{A}^{per}$ guarantees that all local memories of processes undergo "identical" changes with respect to each write operation; that is, all processes simultaneously update their local copies to the value being written.

The code for process p_i appears in Figure 3 in the same style as Figure 2.

3.2 Correctness Proof

Fix any admissible 0-execution σ of $\mathcal{A}^{per}$. We construct a legal linearization τ of σ such that, for each MCS process p_i, $ops(\sigma) \mid i = \tau \mid i$; read and write operations are "serialized" to

**If p_i receives several such update messages simultaneously, it updates X_i to the minimal (with respect to $\mathcal{V}$) of the corresponding values.

Local State:
X_i: The local copy of object X, initially $\perp$

Transition Relation:

$[\mathsf{Read}_i(X), \mathsf{Return}_i(X,v)]$:

$\mathsf{Read}_i(X)$	*Pre:*	$\mathsf{Read}_i(X)$
	Eff:	$\mathsf{TimerSet}_i(\beta d, \mathtt{read}(X))$
$\mathsf{Return}_i(X,v)$	*Pre:*	$\mathsf{TimerExpire}_i(\mathtt{read}(X))$
	Eff:	$\mathsf{Return}_i(X, val(X_i))$

$[\mathsf{Write}_i(X,v), \mathsf{Ack}_i(X)]$:

$\mathsf{Write}_i(X,v)$	*Pre:*	$\mathsf{Write}_i(X,v)$
	Eff:	$\mathsf{Broadcast}_i(\mathtt{update}(X,v))$;
		$\mathsf{TimerSet}_i(\beta d, \mathtt{write}(X))$;
		$\mathsf{TimerSet}_i(d, \mathtt{update}(X,v))$;
$\mathsf{Ack}_i(X)$	*Pre:*	$\mathsf{TimerExpire}_i(\mathtt{write}(X))$
	Eff:	$\mathsf{Ack}_i(X)$
$X_i \leftarrow v$	*Pre:*	$\mathsf{TimerExpire}_i(\mathtt{update}(X,v))$
	Eff:	$X_i \leftarrow v$

Update of X_i:

	Pre:	$\mathsf{Del}_i(\mathtt{update}(X,v), j)$
	Eff:	$X_i \leftarrow v$

Figure 3: The algorithm $\mathcal{A}^{per}$: precondition-effect code for process p_i

occur at their times of call and response in σ, respectively, breaking ties by ordering all write operations before read ones that are "serialized" together and then using $<_{\mathcal{V}}$.

Formally, we assign a time $\mathcal{T}(op)$ to each operation $op = [\mathsf{Call}(op), \mathsf{Response}(op)]$ in σ as follows. Define $\mathcal{T}(op)$ to be either $t^c_\sigma(op)$ if op is a read operation, or $t^r_\sigma(op)$ if op is a write operation. We construct τ as follows:

1. for any pair of operations op_1 and op_2 in σ such that $\mathcal{T}(op_1) < \mathcal{T}(op_2)$, $op_1 \xrightarrow{\tau} op_2$;
2. for any pair of operations op_1 and op_2 in σ such that $\mathcal{T}(op_1) = \mathcal{T}(op_2)$,
   (a) if op_1 is a write operation and op_2 is a read operation, then $op_1 \xrightarrow{\tau} op_2$;
   (b) if op_1 and op_2 are either both read operations or both write operations, then, if $val(op_1) <_{\mathcal{V}} val(op_2)$, then $op_1 \xrightarrow{\tau} op_2$, else ($val(op_1) = val(op_2)$) op_1 and op_2 are ordered arbitrarily in τ.

We start by showing:

Lemma 3.2 *τ is a linearization of σ.*

Proof: Let op_1 and op_2 be any operations in σ such that $op_1 \xrightarrow{\sigma} op_2$. By definition of $\xrightarrow{\sigma}$, $t^r_\sigma(op_1) \leq t^c_\sigma(op_2)$. By definition of $\mathcal{T}$, $\mathcal{T}(op_1) \leq t^r_\sigma(op_1)$, and $\mathcal{T}(op_2) \geq t^c_\sigma(op_2)$. It follows that $\mathcal{T}(op_1) \leq \mathcal{T}(op_2)$. By construction of τ, the only non-trivial case occurs when $\mathcal{T}(op_1) = \mathcal{T}(op_2)$. This happens if and only if $\mathcal{T}(op_1) = t^r_\sigma(op_1)$ and $\mathcal{T}(op_2) = t^c_\sigma(op_2)$. Then, by definition of $\mathcal{T}$, op_1 is a write operation, while op_2 is a read operation. Hence, by construction of τ, $op_1 \xrightarrow{\tau} op_2$, as needed. ■

We continue to prove:

Lemma 3.3 *For each MCS process p_i, $ops(\sigma) \mid i = \tau \mid i$.*

Proof: Fix any MCS process p_i. For any operations op_1 and op_2, say $op_1 \xrightarrow{\tau|i} op_2$ (resp., $op_1 \xrightarrow{\sigma|i} op_2$) if op_1 precedes op_2 in $\tau \mid i$ (resp., $\sigma \mid i$). To show that $\tau \mid i = \sigma \mid i$, it suffices to show that the order of any two operations in $\tau \mid i$ is the same to their order in $\sigma \mid i$.

Consider any pair of operations op_1 and op_2 such that $op_1 \xrightarrow{\sigma|i} op_2$. Clearly, $op_1 \xrightarrow{\sigma} op_2$. Lemma 3.2 implies that $op_1 \xrightarrow{\tau} op_2$. It follows that $op_1 \xrightarrow{\tau|i} op_2$, as needed. ■

We continue to show that τ is a legal operation sequence. We define a relation $\overset{\sigma}{\hookrightarrow}$ between the set of write operations in σ and the set of read operations in σ, as follows. For any pair of write and read operations wop and rop, respectively, in σ, $wop \overset{\sigma}{\hookrightarrow} rop$ if $val(wop) = val(rop)$

and the most recent update (in σ) of the local copy of X by the reading process, before it returns on *rop*, is to *val*(*wop*) as a result of either receipt of an update message $\mathtt{update}(X, val(wop))$ from the writing process, or a timer expiration event $\mathsf{TimerExpire}_i(\mathtt{update}(X, v))$. Roughly speaking, $\overset{\sigma}{\hookrightarrow}$ captures causality and relates each read operation in σ to the most "recent" write operation in σ writing the returned value. We start with a simple claim.

Lemma 3.4 *Consider any pair* $wop = [\mathsf{Write}_i(X, v), \mathsf{Ack}_i(X)]$ *and* $rop = [\mathsf{Read}_k(X), \mathsf{Return}_k(X, v)]$ *of write and read operations, respectively, in* σ*, for some value* $v \in \mathcal{V}$ *and indices* $i, k \in [n]$*, such that* $wop \overset{\sigma}{\hookrightarrow} rop$*. Then,* $wop \xrightarrow{\tau} rop$*.*

Proof: Since all message delays are exactly d and, by the algorithm, each local update is performed time d later than the invocation of the corresponding write operation, it follows that $t^r_\sigma(rop) \geq t^c_\sigma(wop) + d$. Since $\mathcal{T}(rop) = t^c_\sigma(rop) = t^r_\sigma(rop) - \beta d$, and $\mathcal{T}(wop) = t^r_\sigma(wop) = t^c_\sigma(wop) + (1-\beta)d$, it follows that $\mathcal{T}(rop) \geq \mathcal{T}(wop)$. We proceed by case analysis. If $\mathcal{T}(rop) > \mathcal{T}(wop)$, then, by definition of τ (case 1), $wop \xrightarrow{\tau} rop$; furthermore, if $\mathcal{T}(rop) = \mathcal{T}(wop)$, then, by definition of τ (case 2), $wop \xrightarrow{\tau} rop$. Thus, in every case, $wop \xrightarrow{\tau} rop$, as needed. ■

Note that Lemma 3.4 implies that whenever a read operation in τ would return a value "out of order", that is, a value other than that of the immediately preceding it write operation in τ, such a read operation were to be related through $\overset{\sigma}{\hookrightarrow}$ to a write operation that still precedes it in τ. Thus, Lemma 3.4 "restricts" in a sense the way in which τ may violate legality. We finally show:

Lemma 3.5 *τ is a legal operation sequence.*

Proof: An informal outline of our proof follows. We assume that some read operation returns a value other than that of the immediately preceding it write operation; we derive a contradiction by showing that the superseded written value is "known" to the reading process before the read operation returns. We now present the details of the formal proof.

Assume, by way of contradiction, that τ is not legal. It follows, by Lemma 3.4, that there exist operations $wop_1 = [\mathsf{Write}_i(X, v_1), \mathsf{Ack}_i(X)]$, $wop_2 = [\mathsf{Write}_j(X, v_2), \mathsf{Ack}_j(X)]$ and $rop = [\mathsf{Read}_k(X), \mathsf{Return}_k(X, v_1)]$, for some indices i, j and $k \in [n]$, and values $v_1, v_2 \in \mathcal{V}$, such that $wop_1 \xrightarrow{\tau} wop_2$, $wop_2 \xrightarrow{\tau} rop_1$, and there is no write operation wop in τ such that $wop_2 \xrightarrow{\tau} wop \xrightarrow{\tau} rop$; that is, wop_2 is the most "recent" write operation in τ that precedes rop.

By construction of τ, $\mathcal{T}(wop_1) \leq \mathcal{T}(wop_2) \leq \mathcal{T}(rop_1)$; thus, $t^r_\sigma(wop_1) \leq t^r_\sigma(wop_2) \leq t^c_\sigma(rop_1)$. In fact, we prove:

Claim 3.6 $\mathcal{T}(wop_1) < \mathcal{T}(wop_2)$

Proof: Assume, by way of contradiction, that $\mathcal{T}(wop_1) = \mathcal{T}(wop_2)$. By construction of τ, $v_2 <_{\mathcal{V}} v_1$. Moreover, by definition of $\mathcal{T}$, $t^r_\sigma(wop_1) = t^r_\sigma(wop_2)$, which implies that $t^c_\sigma(wop_1) = t^c_\sigma(wop_2)$. Since all message delays equal d, p_k receives update messages simultaneously from p_i and p_j; since it later returns v_1 it must have set X_k to v_1. Hence, by the algorithm, $v_1 <_{\mathcal{V}} v_2$. A contradiction. ■

Note that Claim 3.6 implies that $t^c_\sigma(wop_1) < t^c_\sigma(wop_2)$. Since all message delays equal d, and, by the algorithm, a writing process waits for time d to update its local copy to the value being written, it follows that each process sets its local copy of X to v_1 strictly before it sets it to v_2. Moreover,

$$\begin{aligned} t^r_\sigma(rop) &= t^c_\sigma(rop) + \beta d \\ &\geq t^r_\sigma(wop_2) + \beta d \\ &= t^c_\sigma(wop_2) + (1-\beta)d + \beta d \\ &= t^c_\sigma(wop_2) + d\,; \end{aligned}$$

thus, p_k updates X_k to v_2 no later than time $t^r_\sigma(rop_1)$. It follows that rop returns v_2. A contradiction. ■

By Lemmas 3.2, 3.3, and 3.5, it follows that τ is a legal linearization of σ such that, for each MCS process p_i, $\tau \mid i = \sigma \mid i$. Since σ was chosen arbitrarily, this implies that $\mathcal{A}^{per}$ is a linearizable implementation, as needed.

3.3 Complexity Analysis

Clearly, in any admissible 0-execution of $\mathcal{A}^{per}$, the response time for every read operation is βd, and the response time for every write operation is $(1-\beta)d$, implying that $|\mathbf{R}_{\mathcal{A}^{per}}|(0) = \beta d$ and $|\mathbf{W}_{\mathcal{A}^{per}}|(0) = (1-\beta)d$, as needed.

4 Approximately Synchronized Clocks: Upper Bound

In this section, we present our upper bound for the approximately synchronized clocks model.

Fix throughout any arbitrary constant ε subject to the constraint $0 < \varepsilon \leq \min\{2u, d-u\}$. We show:

Theorem 4.1 *For the approximately synchronized clocks model, there exists a linearizable implementation $\mathcal{A}^{as}$ of read/write objects such that $|\mathbf{R}_{\mathcal{A}^{as}}|(\delta) < \beta d + 3u + \min\{\delta, u\} + \varepsilon$, and $|\mathbf{W}_{\mathcal{A}^{as}}|(\delta) \leq (1-\beta)d + 3u$, for any constant β such that $0 \leq \beta < 1 - u/d$.*

By Proposition 2.9(ii), it suffices to provide a linearizable implementation of a single object $X \in \mathcal{X}$. In Section 4.1, we describe one such implementation $\mathcal{A}^{as}$, while some of its preliminary timing properties are shown in Section 4.2. A correctness proof and complexity analysis for $\mathcal{A}^{as}$ are presented in Sections 4.3 and 4.4, respectively.

4.1 The Algorithm

We start with an informal description of $\mathcal{A}^{as}$. Each process p_i keeps a local copy X_i of object X; denote $val(X_i)$ the value currently held by X_i, initially $\perp$. In addition, p_i keeps a register $LCT_i(X)$ holding the local clock time at the most recent update of X_i, or $\perp$ if this time is at least u earlier than the current local clock time; finally, p_i maintains a set $Pend_i(X)$ of "pending" update messages for object X. Each update message has the form $(\texttt{update}(X, v), c)$ for some value $v \in \mathcal{V}$ and a real number c, which represents the local time of some process.

We now describe the "timings" of $\mathcal{A}^{as}$.

- Upon a $\mathsf{Read}_i(X)$ event, p_i first sets a timer to expire at time βd thereafter, where $0 \leq \beta < 1 - u/d$; then, p_i waits to return until time u has passed without any update of X_i;
- a "time-slicing" technique is used for handling writes; roughly speaking, p_i "slices" each time interval of length $3u + \varepsilon$ into a "write-prohibited" interval of length $3u$, in which actions on a write request may not be initiated by a writing process, followed by an interval of length ε in which they may. Upon a $\mathsf{Write}_i(X, v)$ event, and when outside a "write-prohibited" time interval, p_i broadcasts an $(\texttt{update}(X, v), c)$ message, where c is the local time of p_i at the time of broadcasting. Then, p_i waits for an additional time of $(1 - \beta)d$ to set X_i to v and issue $\mathsf{Ack}_i(X)$.
- On receipt of $(\texttt{update}(X, v), c)$ from a different (writing) process, p_i immediately sets X_i to v.

We now describe the mechanism by which p_i "selects" the value to be returned in a read operation; candidate values are found in the set $Pend_i(X)$. More specifically, p_i considers only values to which it previously set X_i, whose local broadcasting time (accompanying the update message) is within $2u$ of that of the update message with the currently maximal local broadcasting time. (As we will show, the most recently received value is one of the values considered.) The set $Pend_i(X)$ is maintained by p_i as follows. Whenever p_i updates X_i to v, on receipt of $(\texttt{update}(X, v), t)$ as a result of a write operation by another process or by itself, it adds (v, t) to $Pend_i(X)$. [††] At the time of return, p_i returns the maximal (with respect to $<_{\mathcal{V}}$) of the value components of elements of $Pend_i(X)$.

The code for process p_i appears in Figure 4. p_i uses the messages $\texttt{waitread}(X)$ and $\texttt{read}(X)$, and $\texttt{write}(X)$ for implementing the timers needed for the read and write operations, respectively.

For the rest of this section, fix any admissible δ-execution σ of $\mathcal{A}^{as}$. For any write operation $wop = [\mathsf{Write}_i(X, v), \mathsf{Ack}_i(X, v)]$ in σ, denote by $t^{br}_{\sigma}(wop)$ and $t^{del}_{\sigma}(wop)$ the (real) times at which the writing process p_i broadcasts a message $\texttt{update}(X, v)$ (together with its local broadcasting time) and the message $\texttt{update}(X, v)$ is delivered at a process, respectively.

[††] To keep the size of $Pend(X)$ small, at each update p_i removes from $Pend(X)$ all elements (v', t') such that t' is not within $2u$ of the currently maximum time component of elements of $Pend_i(X)$.

Local State:

γ_i:	The local clock component
X_i:	The local copy of object X, initially $\perp$
$LCT_i(X)$:	The local clock time at the most recent change of X_i, or $\perp$, if this time is $\geq u$
$Pend_i(X)$:	A set of "pending" update messages (v', t') for object X
$tmax_i(X)$:	$\max\{t' : (v', t') \in Pend_i(X)\}$

Transition Relation:

$[\mathsf{Read}_i(X), \mathsf{Return}_i(X, v)]$:

$\mathsf{Read}_i(X)$	*Pre:*	$\mathsf{Read}_i(X)$
	Eff:	$\mathsf{TimerSet}_i(\beta d, \mathtt{waitread}(X))$
	Pre:	$\mathsf{TimerExpire}_i(\mathtt{waitread}(X))$ & $LCT_i(X) \neq \perp$ & $\gamma_i - LCT_i(X) < u$
	Eff:	$\mathsf{TimerSet}_i(u - \gamma_i + LCT_i, \mathtt{read}(X))$
$\mathsf{Return}_i(X, v)$	*Pre:*	$(\mathsf{TimerExpire}_i(\mathtt{waitread}(X))$ & $LCT_i(X) = \perp)$ or $\mathsf{TimerExpire}_i(\mathtt{read}(X))$
	Eff:	$X_i \leftarrow \max_{<_V}\{v : (v, t) \in Pend_i(X)\}$; $\mathsf{Return}_i(X, val(X_i))$

$[\mathsf{Write}_i(X, v), \mathsf{Ack}_i(X)]$:

$\mathsf{Write}_i(X, v)$	*Pre:*	$\mathsf{Write}_i(X, v)$ & $\mathrm{fmod}(\gamma_i, 3u + \varepsilon) \leq 3u$
	Eff:	$\mathsf{TimerSet}_i(3u - \mathrm{fmod}(\gamma_i, 3u + \varepsilon), \mathtt{write}(X, v))$
	Pre:	$(\mathsf{Write}_i(X, v)$ & $\mathrm{fmod}(\gamma_i, 3u + \varepsilon) > 3u)$ or $\mathsf{TimerExpire}_i(\mathtt{write}(X, v))$
	Eff:	$\mathsf{Broadcast}_i(\mathtt{update}(X, v))$; $\mathsf{TimerSet}_i((1 - \beta)d, \mathtt{update}(X, v))$
$\mathsf{Ack}_i(X)$	*Pre:*	$\mathsf{TimerExpire}_i(\mathtt{update}(X, v))$
	Eff:	$X_i \leftarrow v$; $Pend_i(X) \leftarrow Pend_i(X) \cup \{(v, \gamma_i - (1 - \beta)d)\}$; $Pend_i(X) \leftarrow \{(v', t') : tmax_i - t' \leq 2u\}$; $\mathsf{Ack}_i(X)$
Update of X_i:	*Pre:*	$\mathsf{Del}_i((\mathtt{update}(X, v), t))$
	Eff:	$Pend_i(X) \leftarrow Pend_i(X) \cup \{(v, t)\}$; $Pend_i(X) \leftarrow \{(v', t') : tmax_i - t' \leq 2u\}$; $LCT_i(X) \leftarrow \gamma_i$

Figure 4: The Algorithm $\mathcal{A}^{as}$: precondition-effect code for process p_i

4.2 Timing Properties

We start by showing that every process "hears" about a value currently being written no later than time βd after the corresponding write operation acknowledges.

Proposition 4.2 *For any write operation wop in σ, $t^r_\sigma(wop) \geq t^{del}_\sigma(wop) - \beta d$.*

Proof: Clearly,

$$\begin{aligned} t^r_\sigma(wop) &= t^{br}_\sigma(wop) + (1-\beta)d && \text{(by the algorithm for writes)} \\ &\geq t^{del}_\sigma(wop) - d + (1-\beta)d \\ &= t^{del}_\sigma(wop) - \beta d\,, \end{aligned}$$

as needed. ■

We define a relation $\overset{\sigma}{\hookrightarrow}$ between write and read operations in σ as follows. For any write and read operations *wop* and *rop*, respectively, in σ, $wop \overset{\sigma}{\hookrightarrow} rop$ if $val(wop) = val(rop)$ and the latest update (in σ) of the local copy of X by the reading process, before it returns on *rop*, is to $val(wop)$, as a result of either receipt of an update message $\mathtt{update}(X, val(wop))$ from the writing process, or a result of a timer expiration event $\mathsf{TimerExpire}_i(\mathtt{update}(X, v))$. Roughly speaking, $\overset{\sigma}{\hookrightarrow}$ captures causality and relates each read operation in σ to the most "recent" write operation in σ writing the returned value. We show that each write operation in σ returns no later than a related read operation in σ.

Proposition 4.3 *Assume $wop \overset{\sigma}{\hookrightarrow} rop$. Then, $t^r_\sigma(rop) \geq t^r_\sigma(wop)$.*

Proof: If *wop* and *rop* occur at the same process, the claim follows trivially from definition of history sequence. So assume that *wop* and *rop* occur at different processes. Clearly,

$$\begin{aligned} t^r_\sigma(rop) &\geq t^{del}_\sigma(wop) + u \\ &\quad \text{(by the algorithm for reads)} \\ &\geq t^{br}_\sigma + d - u + u \\ &\quad \text{(since } \sigma \text{ is admissible)} \\ &= t^r_\sigma(wop) - (1-\beta)d + d \\ &\quad \text{(by the algorithm for writes)} \\ &= t^r_\sigma(wop) + \beta d \\ &\geq t^r_\sigma(wop)\,, \end{aligned}$$

as needed. ■

We continue with timing properties of the slicing technique. We show that for each process p_i, there exists a sequence of "quiet" (update-free) time intervals $quiet_i(k)$, one for each integer $k \geq 1$, with the following properties:

- p_i receives no update messages in $quiet_i(k)$;
- $|quiet_i(k)| \geq 2u - \min\{\delta, u\}$;
- any two consecutive intervals, $quiet_i(k)$ and $quiet_i(k+1)$, are separated by a time interval of length at most $2u + \varepsilon$.

These properties are shown formally in the next two claims.

Proposition 4.4 *For each process p_i, there exists, for each integer $k \geq 1$, a time interval $quiet_i(k)$ in which p_i receives no update messages. Furthermore, $|quiet_i(k)| \geq u$.*

Proof: Consider any writing process p_j. For any integer $k \geq 1$, any (update) message sent from p_j to p_i while $\gamma_j < k(3u+\varepsilon)$ is delivered to p_i while $\gamma_j < k(3u+\varepsilon) + d$; on the other hand, any message sent from p_j to p_i while $\gamma_j > k(3u+\varepsilon) + 3u$ is delivered to p_i while $\gamma_j > k(3u+\varepsilon)+3u+d-u = k(3u+\varepsilon)+d+2u$. (Recall that, by the algorithm, p_j cannot send any update messages while $k(3u+\varepsilon) \leq \gamma_j \leq k(3u+\varepsilon)+3u$.) Thus, no message from p_j is delivered to p_i while $k(3u+\varepsilon)+d \leq \gamma_j \leq k(3u+\varepsilon)+d+2u$. It follows that for each $j \in [n]$, no update message from p_j is delivered to p_i in the time interval $[\gamma_j^{-1}(k(3u+\varepsilon))+d, \gamma_j^{-1}(k(3u+\varepsilon))+d+2u]$. Hence, no message from any process is delivered to p_i in the time interval $quiet_i(k)$, where

$$\begin{aligned} quiet_i(k) &= \bigcap_{j\in[n]} [\gamma_j^{-1}(k(3u+\varepsilon)) + d, \gamma_j^{-1}(k(3u+\varepsilon)) + d + 2u] \\ &= [\max_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) + d, \min_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) + d + 2u]\,. \end{aligned}$$

Hence,

$$\begin{aligned} |quiet_i(k)| &= \min_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) + d + 2u - \max_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) - d \\ &= 2u + \min_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) - \max_{j\in[n]} \gamma_j^{-1}(k(3u+\varepsilon)) \\ &= 2u - \max_{j,j'\in[n]} (\gamma_j^{-1}(k(3u+\varepsilon)) - \gamma_{j'}^{-1}(k(3u+\varepsilon))) \\ &\geq 2u - \min\{\delta, u\} \\ &\quad \text{(by Proposition 2.1, with } \min\{\delta, u\} \text{ for } \delta) \\ &\geq 2u - u \;=\; u\,, \end{aligned}$$

as needed. ■

We continue to show an upper bound on the "gap" between consecutive quiet intervals. For each integer $k \geq 1$, define $gap_i(k) = [\lceil quiet_i(k) \rceil, \lfloor quiet_i(k+1) \rfloor]$. Note that $gap_i(k) \neq \emptyset$. We show:

Proposition 4.5 *For each integer $k \geq 1$, $|gap_i(k)| \leq \min\{\delta, u\} + u + \varepsilon$.*

Proof: Clearly,

$$\begin{aligned}
|gap_i(k)| &= \lfloor quiet_i(k+1) \rfloor - \lceil quiet_i(k) \rceil \\
&= \max_{j \in [n]} \gamma_j^{-1}((k+1)(3u+\varepsilon)) + d - \min_{j \in [n]} \gamma_j^{-1}(k(3u+\varepsilon)) - d - 2u \\
&\leq \max_{j \in [n]} \gamma_j^{-1}(k(3u+\varepsilon)) + 3u + \varepsilon - \min_{j \in [n]} \gamma_j^{-1}(k(3u+\varepsilon)) - 2u \\
&= \max_{j \in [n]} \gamma_j^{-1}(k(3u+\varepsilon)) - \min_{j \in [n]} \gamma_j^{-1}(k(3u+\varepsilon)) + u + \varepsilon \\
&\leq \min\{\delta, u\} + u + \varepsilon \\
&\quad \text{(by Proposition 2.1, with } \min\{\delta, u\} \text{ for } \delta)
\end{aligned}$$

as needed. ■

We continue with a crucial property of the "slicing" intervals. Roughly speaking, we prove that local broadcasting times that are within $2u$ fall within the "same" time slice. Formally, we show:

Proposition 4.6 *Consider write operations wop_1 and wop_2 at processes p_i and p_j, respectively, such that*

$$(3u+\varepsilon)k_1 - \varepsilon \;<\; \gamma_i(t_\sigma^{br}(wop_1)) \;\leq\; (3u+\varepsilon)k_1\,,$$

and

$$(3u+\varepsilon)k_2 - \varepsilon \;<\; \gamma_j(t_\sigma^{br}(wop_2)) \;\leq\; (3u+\varepsilon)k_2\,,$$

for some positive integers k_1 and k_2. Then,

$$|\gamma_i(t_\sigma^{br}(wop_1)) - \gamma_j(t_\sigma^{br}(wop_2))| \;<\; 2u$$

if and only if $k_1 = k_2$.

Proof: By assumption,

$$\begin{aligned}
(3u+\varepsilon)k_1 - \varepsilon - (3u+\varepsilon)k_2 &< \gamma_i(t_\sigma^{br}(wop_1)) - \gamma_j(t_\sigma^{br}(wop_2)) \\
&< (3u+\varepsilon)k_1 - ((3u+\varepsilon)k_2 - \varepsilon)\,,
\end{aligned}$$

so that

Claim 4.7 $|\gamma_i(t_\sigma^{br}(wop_1)) - \gamma_j(t_\sigma^{br}(wop_2)) - (3u+\varepsilon)(k_1-k_2)| < \epsilon$

Assume first that $k_1 \neq k_2$; without loss of generality, take $k_1 \geq k_2 + 1$. Clearly,

$$\begin{aligned}
\gamma_i(t_\sigma^{br}(wop_1)) - \gamma_j(t_\sigma^{br} br(wop_2)) &> (3u+\varepsilon)(k_1-k_2) - \varepsilon \\
&\quad \text{(by Claim 4.7)} \\
&\geq (3u+\varepsilon) - \varepsilon \\
&\quad \text{(since } k_1 \geq k_2 + 1\text{)} \\
&= 3u > 0\,.
\end{aligned}$$

Hence,

$$\begin{aligned}
|\gamma_i(t_\sigma^{br}(wop_1)) - \gamma_j(t_\sigma^{br}(wop_2))| &= \gamma_i(t_{br}(wop_1)) - \gamma_j(t_{br}(wop_2)) \\
&> 3u > 2u
\end{aligned}$$

as needed. Assume now that $k_1 = k_2$. By Claim 4.7,

$$\begin{aligned}
|\gamma_i(t_{br}(wop_1)) - \gamma_j(t_{br}(wop_2))| &< \varepsilon \\
&\leq \min\{2u, d-u\} \\
&\quad \text{(by assumption on } \varepsilon\text{)} \\
&\leq 2u\,,
\end{aligned}$$

as needed. ■

4.3 Correctness Proof

We construct a legal linearization τ of σ such that, for each MCS process p_i, $ops(\sigma) \mid i = \tau \mid i$. We start with an informal outline of the construction of τ and the main ideas used in proving its properties.

The construction proceeds in two phases. In the first phase, each read or write operation in σ is "serialized" to occur at the time of its response in σ, breaking ties by ordering all write operations before read ones that are "serialized" together and then using $<_{\mathcal{V}}$. Call τ' the resulting operation sequence. Clearly, by construction, τ' preserves both the order of operations at each MCS process and the order of non-overlapping operations. However, τ' might not be legal.

In the second phase, we trace all legality violations in τ', and inductively fix each of them. The fix still guarantees that τ' is a linearization of σ which preserves the order of operations at each process. Roughly speaking, we scan τ' and fix each violation of legality by "locally" permuting operations. We show that the index of the first operation "witnessing" a legality violation strictly grows after each fix, as we proceed; thus, inductively, this results in a legal

linearization τ of σ which preserves the order of operations at each process. We now present the details of the formal proof.

Formally, we construct τ' as follows. For any operations op_1 and op_2 in σ if $t^r_\sigma(op_1) < t^r_\sigma(op_2)$, then $op_1 \xrightarrow{\tau'} op_2$; if $t^r_\sigma(op_1) = t^r_\sigma(op_2)$, then, if op_1 and op_2 are write and read operations, respectively, then $op_1 \xrightarrow{\tau'} op_2$, else ($op_1$ and op_2 are either both reads or both writes), if $val(op_1) <_{\mathcal{V}} val(op_2)$, then $op_1 \xrightarrow{\tau'} op_2$.

We now elaborate on the second phase of the construction. We scan τ' till a read operation rop is reached such that $wop_1 \xrightarrow{\tau'} wop_2 \xrightarrow{\tau'} rop$ for some write operations wop_1 and wop_2 in τ' such that $val(wop_1) = val(rop)$, $val(wop_2) \neq val(rop)$, and there is no write operation wop in τ' such that $wop_2 \xrightarrow{\tau'} wop \xrightarrow{\tau'} rop$; call it a *non-admissible* triple. Let $i_{viol}(\tau')$ be the index of rop_1 in τ'. We permute wop_2 to immediately precede wop_1 in τ'. Let τ_1 be the resulting sequence.

Our proof proceeds in two steps. First, we show that a non-admissible triple is the only cause of a legality violation; we next prove that $i_{viol}(\tau_1) > i_{viol}(\tau')$, by showing that the prefix of τ_1 ending with rop_1 is a legal sequence of operations; induction implies, then, the correctness of our construction.

Our first simple claim characterizes a legality violation; and it implies that legality may only be violated because of a non-admissible triple. In all of our discussion, wop_i and rop_i will denote write and read operations on object X such that v_i is the associated value with each of them. Since, by construction, write operations precede in τ' read operations that occur at the same time, Proposition 4.3 implies that wop_i precedes rop_i in τ'. It follows that that a non-admissible triple is, indeed, the only possible form of a legality violation. We show that the values of the involved write operations must have been broadcast "very close" in time.

Lemma 4.8 *Assume that* $wop_1 \xrightarrow{\tau'} wop \xrightarrow{\tau'} rop_1$. *Then,*

$$|\gamma_i(t^{br}_\sigma(wop_1)) - \gamma_j(t^{br}_\sigma(wop))| \leq 2u\,.$$

Proof: Assume, by way of contradiction, that

$$|\gamma_i(t^{br}_\sigma(wop_1)) - \gamma_j(t^{br}_\sigma(wop))| > 2u\,.$$

We proceed by case analysis on the sign of $\gamma_i(t^{br}_\sigma(wop_1)) - \gamma_j(t^{br}_\sigma(wop))$.

1. Assume first that $\gamma_i(t^{br}_\sigma(wop_1)) - \gamma_j(t^{br}_\sigma(wop)) > 0$; It follows that $\gamma_i(t^{br}_\sigma(wop_1)) - \gamma_j(t^{br}_\sigma(wop)) > 2u$. By Lemma 2.2, it folllows that $t^{br}_\sigma(wop_1) - t^{br}_\sigma(wop_2) > 2u - \min\{\delta, u\} > 2u - u = u > 0$. By the algorithm for writes, for each $i \in \{1,2\}$, $t^r_\sigma(wop_i) = t^{br}_\sigma(wop_i) + (1-\beta)d$. It follows that $t^r_\sigma(wop_1) > t^r_\sigma(wop_2)$. By construction of τ', $wop_2 \xrightarrow{\tau'} wop_1$. A contradiction.

2. Assume now that $\gamma_j(t^{br}_\sigma(wop_2)) - \gamma_i(t^{br}_\sigma(wop_1)) > 2u$. By the algorithm and the way τ' was constructed, $t^{br}(wop_2) < t^r(wop_2) \leq t^r(wop_1)$. Proposition 4.2 implies that $t^{del}(wop_2) \leq t^r(wop_1)$. Thus, at time $t^r(rop_1)$, both v_1 and v_2 reside in the memory of the reading process. It follows, however, that $tmax_i - \gamma_i(t^{br}(wop_1)) \geq \gamma_i(t_{br}(wop_2)) > 2u$. This contradicts the fact that rop_1 returns v_1.

■

We continue to show a simple property of τ'.

Proposition 4.9 *Consider read operations rop and rop′ such that* $wop_1 \xrightarrow{\tau'} rop \xrightarrow{\tau'} wop_2$, *and* $wop_1 \xrightarrow{\tau'} rop' \xrightarrow{\tau'} wop_2$. *Assume there is no write operation wop in* τ' *such that* $wop_1 \xrightarrow{\tau'} wop \xrightarrow{\tau'} wop_2$. *Then,* $val(rop) = val(rop')$.

Proof: By construction, $t^r(rop), t^r(rop') < t^r(wop')$. Hence, it follows by Proposition 4.3 that $val(rop), val(rop') \neq val(wop')$. By Claim 4.2, every process receives $val(wop)$ and all values of preceding write operations in τ' by time $t^r(wop) + \beta d$. Since there is no write operation in the interval of operations $(wop, wop')_{\tau'}$, no process modifies its $Pend(X)$ set except on receipt of $val(wop')$ in the time interval $(t^r(wop), t^r(wop'))$. Notice, however, that a process that modifies $Pend(X)$ on receipt of an update message for wop') may return for a read operation no earlier than $t^r(wop')$, and, by construction, such a read operation is not included in $(wop, wop')_{\tau'}$. This implies that every read operation in $(wop, wop')_{\tau'}$ returns the same value, as needed. ■

Proposition 4.9 implies that we may assume, without loss of generality, that at most one read operation may be completed between any two successive completions of write operations in τ'. The next claim argues that once a value is returned by a read operation, no (later) read operation in τ' may return a value of a preceding (in τ') write operation.

Proposition 4.10 *Consider a write operation wop such that* $wop_1 \xrightarrow{\tau'} wop \xrightarrow{\tau'} rop_1$. *Then, there exists no read operation rop such that* $wop \overset{\tau'}{\rightsquigarrow} rop$.

Proof: Assume, by way of contradiction, that there exists a read operation rop such that $wop \overset{\tau'}{\rightsquigarrow} rop$. We proceed by case analysis.

1. Assume first that $t^r_\sigma(rop_2) > t^r_\sigma(rop_1)$. It follows, by Claim 4.9, that there must be at least one write operation on X in the interval of operations $(rop_1, rop_2)_{\tau'}$; let wop_3 be one with the maximal broadcasting time among all such write operations. We consider the intervals $(wop_1, rop_1)_{\tau'}$ and $(wop_2, rop_2)_{\tau'}$; it follows from Proposition 4.6 and Lemma 4.8 that the local broadcasting times of $val(wop_1)$ and $val(wop_3)$ are in the same time slice, as are those of $val(wop_2)$ and $val(wop_3)$. Since every process receives both $val(wop_1)$ and $val(wop_2)$ by time $t_r(wop_2) + \beta d$, it follows, by the algorithm, that all values v_1, v_2 and v_3 were considered in both rop_1 and rop_2 as candidate values to be returned. Thus, both $v_1 <_{\mathcal{V}} v_2$ and $v_2 <_{\mathcal{V}} v_1$. A contradiction.

2. Assume now that $t^r(rop_2) < t^r(rop_1)$. We apply an identical reasoning to the intervals $(wop_1, rop_2)_{\tau'}$ and $(wop_1, rop_1)_{\tau'}$. Let $tmax_1$ and $tmax_2$ be the maximal time components of elements of $Pend(X)$ of the processes performing rop_1 and rop_2, respectively, at the time they return. Clearly, by time $t^r(rop_2)$, each processes modifies its $Pend(X)$ set as a result of a write operation on X completed by time $t^r(rop_2)$. Thus, any modification of $Pend(X)$ at time $> t_r(rop_2)$ corresponds to a write operation returning at time $> t_r(rop_2)$; hence, the broadcasting time of such an operation is greater than the broadcasting time of any write operation completed by time $t_r(wop_2)$, and the addition of its value to $Pend(X)$ of any process can only increase $tmax_2$. Hence, $tmax_1 \geq tmax_2$. Clearly, $tmax_2 - \gamma_j(t^r(wop_2)) \leq 2u$, and $tmax_1 - \gamma_i(t^r(wop_1)) \leq 2u$. This implies that $tmax_1 - t^r(wop_2) \leq 2u$. By the algorithm and the way wop_1 returns, $v_1 <_{\mathcal{V}} v_2$. Hence, by the way wop_2 returns, $tmax_2 - t^r(wop_1) > 2u$. A contradiction.

■

Clearly, Proposition 4.10 implies that, after the reordering, the prefix of τ' ending with rop_1 is a legal operation sequence. We next prove that this prefix is also a linearization of σ by showing that the reordered operations wop_1 and wop_2 (in the non-admissible triple wop_1, wop_2, rop_1) "overlap" in σ.

Proposition 4.11 *The reordered prefix of τ' ending with rop_1 is a linearization of σ.*

Proof: By Proposition 4.6, the local broadcasting times of wop_1 and wop_2 fall in the same slice. It follows by Proposition 2.4 that $|t^{br}_\sigma(wop_1) - t^{br}_\sigma(wop_2)| \leq u + \epsilon$. Hence,

$$t^c_\sigma(wop_2) \;\leq\; t^{br}_\sigma(wop_2) \;<\; t^{br}_\sigma(wop_1) + \varepsilon + u \;=\; t^r_\sigma(wop_1) - d + \varepsilon + u \;\leq\; t^r_\sigma(wop_1)\,,$$

by assumption on ε, as needed. ■

Since, in permuting τ', we reordered only wop_1 and wop_2, which, by Proposition 4.11, "overlap", wop_1 and wop_2 may not be performed by the same process. This implies that our reordering yields an operation sequence preserving the order of operations at each process. Hence, $i_{viol}(\tau_1) > i_{viol}(\tau')$. By induction, it follows that $\mathcal{A}^{as}$ is a linearizable implementation.

4.4 Complexity Analysis

The upper bound of $(1-\beta)d + 3u$ on $|\mathbf{W}_{\mathcal{A}^{as}}|(\delta)$ is obvious since, by the algorithm, p_i first waits for time at most $3u$ till it exits a "write-prohibited" interval, and then for an additional time $(1-\beta)d$ to issue an acknowledgment. We proceed to show that $|\mathbf{R}_{\mathcal{A}^{as}}|(\delta) < \beta d + 3u + \min\{\delta, u\} + \varepsilon$.

Consider a $\mathsf{Read}_i(X)$ event that occurs at time t in σ. We show that a matching response occurs by time $t' < t + \beta d + 3u + \min\{\delta, u\} + \varepsilon$. Observe that such a response may only be prevented if an update message is delivered to p_i. It seems as if a starvation may occur due to successive update events; however, the "slicing" technique assures that this is not the case. Clearly, it is possible that $\mathsf{Read}_i(X)$ occurs (at time t) within some interval $quiet_i(k)$, for some integer k, but p_i enters $gap_i(k)$, by receiving some update message before it may issue a response to $\mathsf{Read}_i(X)$. Such an update message must be received no earlier than time $t + u$, since, otherwise, $LCT_i(X)$ would have attained the value u. By Claim 4.5, p_i enters $quiet_i(k+1)$ by time $< t + u + 2u + \varepsilon = t + 3u + \varepsilon$. Since, by Claim 4.4, $|quiet_i(k+1)| \geq 2u - \min\{\delta, u\}$, $LCT_i(X)$ attains the value u within $quiet_i(k+1)$; hence, p_i issues $\mathsf{Return}_i(X, val(X_i))$ by time $< t + \beta d + 3u + \min\{\delta, u\} + \varepsilon$, as needed.

Since σ was chosen arbitrarily, this implies that $|\mathbf{R}_{\mathcal{A}^{as}}|(\delta) < \beta d + 3u + \min\{\delta, u\} + \varepsilon$, as needed.

In addition, it does not seem that the better clock precision achieved by the clock synchronization algorithm of Lundelius and Lynch [44, Section 4] can considerably improve our results.

5 Approximately Synchronized Clocks: Lower Bounds

In this section, we present lower bounds for the approximately synchronized clocks model.

This section is organized as follows. In Section 5.1, we present a lower bound on the sum of worst-case response times for read and write operations; this bound applies to a certain class of sequentially consistent implementations, and it implies a corresponding lower bound for linearizable implementations. In Sections 5.2 and 5.3, we present lower bounds on individual worst-case response times for read and write operations, respectively; these bounds apply to any linearizable implementation.

5.1 Read and Write Operations

Our lower bound on the sum of the worst-case response times for read and write operations applies to a certain class of implementations of objects, called *object-separable* and *object-symmetric*; roughly speaking, such implementations satisfy the following conditions.

1. Each process handles activity involving a certain object independently of all activity, concurrent or even previous, involving other objects; hence, the sequence of actions taken by the process on this object is completely separated from and not affected by the presence or abscence of events involving other objects.
2. Each process handles activity involving a certain object in precisely the same way it handles activity on any other object.

Our formal definitions follow.

An implementation $\mathcal{A}$ is *object-separable* if for each process p_i, every state s of p_i contains $|\mathcal{X}|$ components $s_1, s_2, \ldots, s_{|\mathcal{X}|}$, one for each object, so that if an interrupt event i_k involves object X_k and $(\langle q, \gamma, \mathsf{i}_k\rangle, \langle q', \mathcal{R}, \mathcal{S}, \mathcal{T}\rangle)$ is a computation step of process p_i, then *(i)* $q'_l = q_l$ for every $l \neq k$; *(ii)* q'_k, $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ result from the application of p_i's transition function on q_k, γ and i_k, and *(iii)* each of $\mathcal{R}$, $\mathcal{S}$ and $\mathcal{T}$ contains events that involve only object X_k. Thus, the transition function of a process in an object-separable implementation may be regarded as the "parallel composition" of $|\mathcal{X}|$ transition functions, one for each state component associated with a specific object. If, in addition, these $|\mathcal{X}|$ transition functions are "identical", the implementation is said to be object-symmetric.

Formally, an object-separable implementation $\mathcal{A}$ is *object-symmetric* if for each process p_i, for any identical up to object interrupt events i_k and i_l involving objects X_k and X_l, respectively, if $(\langle q, \gamma, \mathsf{i}_k\rangle, \langle q', \mathcal{R}, \mathcal{S}, \mathcal{T}\rangle)$ and $(\langle \widehat{q}, \gamma, \mathsf{i}_k\rangle, \langle \widehat{q}', \widehat{\mathcal{R}}, \widehat{\mathcal{S}}, \widehat{\mathcal{T}}\rangle)$ are computation steps of p_i such that $q_k = \widehat{q}_l$ are identical, then each of the pairs $\mathcal{R}$ and $\widehat{\mathcal{R}}$, $\mathcal{S}$ and $\widehat{\mathcal{S}}$, and $\mathcal{T}$ and $\widehat{\mathcal{T}}$ are identical up to object.

We start with two properties which will later be used in the proof of the lower bound on the sum of the worst-case response times for read and write operations; these are simple properties of sequentially consistent, object-separable and object-symmetric implementations, which may be of independent interest.

Throughout this section, assume that $\mathcal{A}$ is any sequentially consistent, object-separable and object-symmetric implementation of read/write objects.

5.1.1 First Property

Loosely speaking, we establish that in any execution of $\mathcal{A}$, objects "identically written" by processes "respond identically" to read operations. This property is inspired by and generalizes a result of Lipton and Sandberg [41, Theorem 1], formalized and strengthened by Attiya and Welch [15, Theorem 3.1] for the perfect clocks model where $u = d$, to the approximately synchronized clocks model.

Formally, consider objects X and Y; by the serial specifications of X and Y, there exists an admissible δ-execution σ_1 of $\mathcal{A}$ consisting of the following operations at processes p_i and p_j:

- p_i performs a write operation wop_i on Y with $val(wop_i) = v$ and $t^c_{\sigma_1}(wop_i) = 0$, immediately followed by a read operation rop_i on X with $t^c_{\sigma_1}(rop_i) = t^r_{\sigma_1}(wop_i)$;
- p_j performs a write operation wop_j on X with $val(wop_j) = v$ and $t^c_{\sigma_1}(wop_j) = 0$, immediately followed by a read operation rop_j on X with $t^c_{\sigma_1}(rop_j) = t^r_{\sigma_1}(wop_j)$.

We assume that message delays in σ_1 are as follows. Each message from p_l to p_j, $l \neq j$, incurs a delay of d; each message from p_j to p_l, $l \neq j$, incurs a delay of $d - \min\{\delta, u\}$; any

other message incurs a delay of $d - \min\{\delta, u\}/2$. Furthermore, we assume that for each $l \neq j$, $\gamma_l(t) = t$, while $\gamma_j(t) = t - \min\{\delta, u\}/2$. We show:

Proposition 5.1 $val_{\sigma_1}(rop_i) = val_{\sigma_1}(rop_j) = v$

Proof: We start with an informal outline of our proof. By "perturbing" σ_1, we obtain an execution σ_1', which appears "symmetric" with respect to objects X and Y, and has the following properties: *(i)* each process "sees" each event happening at the same (local) time in both σ_1 and σ_1'; *(ii)* each of the objects X and Y undergoes the same "changes" at the same (local) time in σ_1'. By *(i)*, it suffices to show that both read operations return v in σ_1', which follows from *(ii)* and object-symmetry. We now present the details of the formal proof.

We describe how to "perturb" σ_1 in order to obtain another admissible δ-execution σ_1' of $\mathcal{A}$. Consider the real vector $\vec{s} = \langle s_0, s_1, \ldots, s_{n-1} \rangle$, where $s_l = \min\{\delta, u\}/2$ if $l = j$, and 0 otherwise. Then, $\sigma_1' = shift(\sigma_1, \vec{s})$ with clocks $\Gamma_1' = shift(\Gamma_1, \vec{s})$. That is, each event at process p_j that occurs at real time t in σ_1 will occur at real time $t - \min\{\delta, u\}/2$ in σ_1', while times of events at all other processes remain unchanged; p_j's clock is shifted forward by $\min\{\delta, u\}/2$, while all other clocks remain unchanged. By Lemma 2.11, it follows:

Lemma 5.2 *σ_1' is an execution of $\mathcal{A}$ with clocks Γ_1' that is equivalent to σ_1 with clocks Γ.*

We proceed to show:

Lemma 5.3 *σ_1' is an admissible δ-execution of $\mathcal{A}$.*

Proof: We first show:

Claim 5.4 *σ_1' is a δ-execution of $\mathcal{A}$.*

Proof: Fix any processes p_l and p_m. We proceed by case analysis.

1. Assume that none of p_l and p_m is p_j. Then, for any real time t,

$$\begin{aligned} |\gamma_l'(t) - \gamma_m'(t)| &= |\gamma_l(t) - \gamma_m(t)| \\ &\quad \text{(by construction of } \Gamma') \\ &= |t - t| \\ &\quad \text{(by construction of } \mathcal{C}) \\ &= 0 < \delta, \end{aligned}$$

   as needed.

2. Assume now that some of p_l and p_m, say p_l, is p_j. Then, for any real time t,

$$\begin{aligned}
|\gamma'_l(t) - \gamma'_m(t)| &= |\gamma'_j(t) - \gamma'_m(t)| \\
&= |\gamma_j(t) + \frac{\min\{\delta, u\}}{2} - \gamma_m(t)| \\
&\quad \text{(by construction of } \Gamma') \\
&= |t - \frac{\min\{\delta, u\}}{2} + \frac{\min\{\delta, u\}}{2} - t| \\
&\quad \text{(by construction of } \Gamma) \\
&= 0 < \delta ,
\end{aligned}$$

as needed.

■

We continue to show that all delays are in the range $[d-u, d]$. Fix any MCS processes p_l and p_m, and let d_{lm} be the delay of any message $\mathtt{m}$ from p_l to p_m in σ_1. By Lemma 2.12, the delay $d_{lm}\prime$ of $\mathtt{m}$ in $\sigma_1\prime$ is $d_{lm} + s_l - s_m$. We proceed by case analysis.

1. Assume that both $l \neq j$ and $m \neq j$, so that $d_{lm} = d - \min\{\delta, u\}/2$, and $s_l = s_m = 0$. Then, $d'_{lm} = d - \min\{\delta, u\}/2 + 0 - 0 = d - \min\{\delta, u\}/2$.
2. Assume now that $l = j$, so that $d_{lm} = d - \min\{\delta, u\}$, $s_l = \min\{\delta, u\}$, and $s_m = 0$. Then, $d'_{lm} = d - \min\{\delta, u\} + + \min\{\delta, u\}/2 - 0 = d - \min\{\delta, u\}/2$.
3. Assume now that $m = j$, so that $d_{lm} = d$, $s_l = 0$, and $s_m = \min\{\delta, u\}/2$. Then, $d_{lm}\prime = d + 0 - \min\{\delta, u\}/2 = d - \min\{\delta, u\}/2$.

Notice that since $\min\{\delta, u\}/2 \leq u$, $d - \min\{\delta, u\}/2 \geq d - u/2 \geq d - u$; hence, $d'_{lm} \in [d-u, d]$. This implies that σ'_1 is an admissible execution. By Claim 5.4, it follows that σ'_1 is an admissible δ-execution of $\mathcal{A}$, as needed. ■

Lemma 5.2 implies that $val_{\sigma'_1}(rop_i) = val_{\sigma_1}(rop_i)$ and $val_{\sigma'_1}(rop_j) = val_{\sigma_1}(rop_j)$. Thus, it suffices to show that $val_{\sigma'_1}(rop_i) = val_{\sigma'_1}(rop_j) = v$.

Notice that in σ'_1, by construction, $\gamma'_i(0) = 0$, while $\gamma'_j(0) = 0 - \min\{\delta, u\}/2 + \min\{\delta, u\}/2 = 0$ Thus, local clocks of p_i and p_j are identical in σ'_1. Since all message delays are equal, object symmetry implies that $v'_i = v'_j$. Notice that $v'_i = v'_j = \perp$ contradicts sequential consistency. Therefore, $v'_i = v'_j = v$, as needed. ■

5.1.2 Second Property

Loosely speaking, we consider an execution of $\mathcal{A}$ with "conflicting" write operations on some object, and "late" read operations on this object, performed after processes "hear" about the write operations; we establish that the "late" read operations must return the same value.

Formally, consider an object X, holding the value $\perp$ at time 0. By the serial specification of X, there exists an admissible δ-execution σ_2 of $\mathcal{A}$ consisting of the following operations at processes p_i, p_j, p_k and p_l:

- p_i performs a write operation wop_i on X with $val(wop_i) = v_i$ and $t^c_{\sigma_2}(wop_i) = 0$;
- p_j performs a write operation wop_j on X with $val(wop_j) = v_j$ and $t^c_{\sigma_2}(wop_j) = 0$;
- p_k performs a read operation rop_k on X with $t^c_{\sigma_2}(rop_k) > d + |\mathbf{W}_{\mathcal{A}}|(\delta)$;
- p_l performs a read operation rop_l on X with $t^c_{\sigma_2}(rop_l) > d + |\mathbf{W}_{\mathcal{A}}|(\delta)$.

Furthermore, we assume that message delays in σ_2 are all equal, and that all local clocks are perfectly synchronized.

We show:

Proposition 5.5 $val_{\sigma_2}(rop_k) = val_{\sigma_2}(rop_l)$

Proof: Assume, by way of contradiction, that $val_{\sigma_2}(rop_k) \neq val_{\sigma_2}(rop_l)$. We construct an admissible δ-execution σ_2' of $\mathcal{A}$ that is not sequentially consistent.

We start with an informal outline of our proof. We obtain an admissible δ-execution σ_2' by "augmenting" σ_2 as follows. Each of p_k and p_l performs an additional later read operation on X, preceded by a pair of a write and a read operation on two other objects Y and Z. We use object symmetry to argue that the operations on Y and Z must be "interleaved" in any legal serialization of σ_2'. This will prevent all read operations on X by one of p_k and p_l to precede all such operations by the other in any legal serialization. Since σ_2' is an "augmentation" of σ_2, the "early" read operations on X in σ_2' must return different values, as in σ_2. We use object separability to argue that each "later" read operation on X returns the same value as the corresponding earlier read operation by the same process. Since read operations on X by p_k and p_l must be "interleaved", this contradicts sequential consistency. We now present the details of the formal proof.

Consider objects Y and Z. By the serial specifications of X, Y and Z, there exists an admissible δ-execution σ_2' of $\mathcal{A}$ consisting of the following operations at processes p_i, p_j, p_k and p_l:

- p_i performs a write operation wop_i on X with $val_{\sigma_2'}(wop_i) = x_i$ and $t^c_{\sigma_2'}(wop_i) = 0$;

- p_j performs a write operation wop_j on X with $val_{\sigma_2'}(wop_j) = x_j$ and $t^c_{\sigma_2'}(wop_j) = 0$;
- p_k performs a read operation rop_k on X with $t^c_{\sigma_2'}(rop_k) = t^c_{\sigma_2}(rop_k)$, followed by a write operation wop_k on Y with $val_{\sigma_2'}(wop_k) = y$ and $t^c_{\sigma_2'}(wop_k) = t^r_{\sigma_2'}(rop_k)$, followed by a read operation $rop_k^{(1)}$ on Z with $t^c_{\sigma_2'}(rop_k^{(1)}) = t^r_{\sigma_2'}(wop_k)$, and finally followed by a read operation $rop_k^{(2)}$ on X with $t^c_{\sigma_2'}(rop_k^{(2)}) = t^r_{\sigma_2'}(rop_k^{(2)})$;
- p_l performs a read operation rop_l on X with $t^c_{\sigma_2'}(rop_l) = t^c_{\sigma_2}(rop_l)$, followed by a write operation wop_l on Z with $val_{\sigma_2'}(wop_k) = z$ and $t^c_{\sigma_2'}(wop_l) = t^r_{\sigma_2'}(rop_l)$, followed by a read operation $rop_l^{(1)}$ on Y with $t^c_{\sigma_2'}(rop_l^{(1)}) = t^r_{\sigma_2'}(wop_l)$, and finally followed by a read operation $rop_l^{(2)}$ on X with $t^c_{\sigma_2'}(rop_l^{(2)}) = t^r_{\sigma_2'}(rop_l^{(2)})$.

Furthermore, we assume that all message delays in σ_2' are equal, and that all local clocks are perfectly synchronized.

By object separability, $val_{\sigma_2'}(rop_k^{(2)}) = val_{\sigma_2'}(rop_k)$ and $val_{\sigma_2'}(rop_l^{(2)}) = val_{\sigma_2'}(rop_l)$. Since all message delays are equal, object symmetry implies that either $val(rop_k^{(1)}) = val(wop_l)$ and $val(rop_l^{(1)}) = val(wop_k)$, or $val(rop_k^{(1)}) = val(rop_l^{(1)}) = \perp$. However, notice that $val(rop_k^{(1)}) = val(rop_l^{(1)}) = \perp$ violates sequential consistency. It follows that $val(rop_k^{(1)}) = val(wop_l)$ and $val(rop_l^{(1)}) = val(wop_k)$.

Since σ_2' is sequentially consistent, there exists a legal serialization τ of σ_2' such that for each MCS process p_i, $ops(\sigma_2') \mid i = \tau \mid i$. Clearly, either $rop_k^{(2)} \xrightarrow{\tau} rop_l$ or $rop_l^{(2)} \xrightarrow{\tau} rop_k$; without loss of generality, assume the former. Since $\sigma_2' \mid l = \tau \mid l$, $rop_l \xrightarrow{\tau} wop_l$. By the serial specification of Z, $wop_l \xrightarrow{\tau} rop_k^{(1)}$. Since $\sigma_2' \mid k = \tau \mid k$, $rop_k^{(1)} \xrightarrow{\tau} rop_k^{(2)}$. It follows that $rop_l \xrightarrow{\tau} rop_k^{(2)}$. A contradiction. ■

We now present our main lower bound result.

Theorem 5.6 *For the approximately synchronized clocks model, in any sequentially consistent, object-separable and object-symmetric implementation $\mathcal{A}$ of at least three objects accessed by at least four processes,*

$$(|\mathbf{R}_{\mathcal{A}}| + |\mathbf{W}_{\mathcal{A}}|)(\delta) \;\geq\; d + \frac{\min\{\delta, u\}}{2}.$$

Proof: Assume, by way of contradiction, that there exists a sequentially consistent, object-separable and object-symmetric implementation $\mathcal{A}$ of such objects for which $(|\mathbf{R}_{\mathcal{A}}|+|\mathbf{W}_{\mathcal{A}}|)(\delta) < d + \min\{\delta, u\}/2$. We construct an admissible δ-execution of $\mathcal{A}$ that is not sequentially consistent.

We start with an informal outline of our proof. We construct an admissible δ-execution σ of $\mathcal{A}$ in which each of two MCS processes p_k and p_l performs an "early" and a "late" read operation on an object X; we use object-symmetry to "force" p_k and p_l to either return different values in different order, which, clearly, violates sequential consistency, or to maintain "inconsistent" copies of the same object, also shown to violate sequential consistency. These different values are written by "conflicting" write operations on X by processes p_i and p_j. We appropriately choose message delays in σ so that, under the assumption $(|\mathbf{R}_{\mathcal{A}}|+|\mathbf{W}_{\mathcal{A}}|)(\delta) < d+\min\{\delta,u\}/2$, p_k "gathers" fast information about the write operation by p_i, but cannot "hear" about the write operation by p_j till late. (The roles of delays of messages from p_i and p_j are reversed for p_l.) Thus, by object-symmetry, read operations by p_k and p_l return different values in different order, establishing the contradiction. We now present the details of the formal proof.

Consider objects X and Y, each holding the value $\perp$ at time 0. By the serial specifications of X and Y, there exists an admissible δ-execution σ_1' of $\mathcal{A}$ consisting of the following operations at processes p_i, p_j, p_k and p_l:

- p_i performs a write operation wop_i on X with $val(wop_i) = v_i$ and $t^c_\sigma(wop_i) = \min\{\delta,u\}/2$, followed by a read operation rop_i on Y with $t^c_\sigma(rop_i) = t^r_\sigma(wop_i)$;
- p_j performs a write operation wop_j on X with $val(wop_j) = v_j$ and $t^c_\sigma(wop_j) = \min\{\delta,u\}/2$, followed by a read operation rop_j on Y with $t^c_\sigma(rop_j) = t^r_\sigma(wop_j)$;
- p_k performs a write operation wop_k on Y with $val(wop_k) = v_k$ and $t^c_\sigma(wop_k) = 0$, followed by two consecutive read operations $rop_k^{(1)}$ and $rop_k^{(2)}$ on X, with $t^c_\sigma(rop_k^{(1)}) = t^r_\sigma(wop_k)$ and $t^c_\sigma(rop_k^{(2)}) > |\mathbf{W}_{\mathcal{A}}|(\delta) + \min\{\delta,u\}2$;
- p_l performs a write operation wop_l on Y with $val(wop_l) = v_l$ and $t^c_\sigma(wop_l) = 0$, followed by two consecutive read operations $rop_l^{(1)}$ and $rop_l^{(2)}$ on X, with $t^c_\sigma(rop_l^{(1)}) = t^r_\sigma(wop_l)$ and $t^c_\sigma(rop_l^{(2)}) > |\mathbf{W}_{\mathcal{A}}|(\delta) + \min\{\delta,u\}/2$.

We assume that the message delays in σ are as follows. Each message from p_i to p_m, $m \neq i$, incurs a delay of either d if $m = l$, or $d-u$ if $m \neq l$; each message from p_j to p_m, $m \neq j$, incurs a delay of either d if $m = k$, or $d-u$ if $m \neq k$; any other message incurs a delay of $d-u/2$. Furthermore, we assume that in σ, $\gamma_m(t) = t$ if $m \notin \{i,j\}$, or $t - \min\{\delta,u\}/2$ if $m \in \{i,j\}$. We remark that any message sent by p_i or p_j while performing write operations on X is delivered before the late read operations on X by p_k and p_l are invoked.

Since $(|\mathbf{R}_{\mathcal{A}}| + |\mathbf{W}_{\mathcal{A}}|)(\delta) < d + \min\{\delta,u\}/2$, it follows that $t^r_\sigma(rop_k^{(1)}) < d + \min\{\delta,u\}/2$, hence, the assumed message delays imply that p_k may not receive a message from p_j till after time $t^r_\sigma(rop_k^{(1)})$. Thus, Proposition 5.1 applies on the prefices of $\sigma \mid i$ and $\sigma \mid j$ consisting of all events at p_i and p_j occurring no later than time $t^c_\sigma(rop_k^{(1)})$ in σ to yield that $val(rop_k^{(1)}) = v_i$. A symmetric argument establishes that $val(rop_l^{(1)}) = v_j$.

By the symmetry in delays of messages sent by processes p_i and p_j (writing to X) to processes p_k and p_l, there are two possibilities: either $val_{\sigma_2}(rop_k) = x_j$ and $val_{\sigma_2}(rop_l) = x_i$,

or $val_{\sigma_2}(rop_k) = x_i$ and $val_{\sigma_2}(rop_l) = x_j$. Clearly, the first possibility immediately contradicts sequential consistency. On the other hand, the second possibility contradicts, by object-separability, Proposition 5.5. Thus, in every case, a contradiction is reached. ■

We remind the reader that although, apparently, the assumption of at least three objects is not explicitly used in the proof of Theorem 5.6, this assumption is necessary since it is used in the proof of Proposition 5.5.

Since linearizability implies sequential consistency, it immediately follows:

Corollary 5.7 *For the approximately synchronized clocks model, in any linearizable, object separable and object symmetric implementation of at least three objects accessed by at least four processes,*

$$(|\mathbf{R}_{\mathcal{A}}| + |\mathbf{W}_{\mathcal{A}}|)(\delta) \quad \geq \quad d + \frac{\min\{\delta, u\}}{2}\,.$$

5.2 Read Operations

We prove a lower bound on the worst-case response time for a read operation; this applies to any linearizable implementation of read/write objects, under reasonable assumptions on the sharing pattern of processes. More specifically, we consider any linearizable implementation of read/write objects including one with at least two readers and a distinct writer; we show that the worst-case response time for a read operation on this object is no less than $\min\{\delta, u\}/2$. The proof constructs an execution for which if read operations are too short, then linearizability can be violated by appropriately shifting process' histories. We show:

Theorem 5.8 *Assume that $\mathcal{A}$ is a linearizable implementation of read/write objects including an object X with at least two readers and a distinct writer. Then, for the approximately synchronized clocks model,*

$$|\mathbf{R}_{\mathcal{A}}(X)|(\delta) \quad \geq \quad \min\{\delta, u\}/2\,.$$

Proof: Assume, by way of contradiction, that there exists a linearizable implementation $\mathcal{A}$ of X for which $|\mathbf{R}_{\mathcal{A}}(X)|(\delta) < \min\{\delta, u\}/2$. We construct an admissible δ-execution of $\mathcal{A}$ which is not linearizable.

Let p_i and p_j be two processes that read X, and let p_k be a process that writes X. An informal outline of our proof follows. We start with an execution in which p_i reads $\perp$ from X, then p_j and p_i alternate reading from X while p_k is writing x to X, and finally p_j reads x from X. Thus, there exists a read operation rop_0, say by p_i, that returns $\perp$ and is immediately followed by a read operation rop_1 by p_j that returns x. If p_i's process history is shifted later by $\min\{\delta, u\}/2$, while p_j's process history is shifted earlier by $\min\{\delta, u\}/2$, the result is an

execution in which rop_1 precedes rop_0; in the meanwhile, processes' clocks are appropriately shifted so that p_i and p_j still "see" the same events occurring at the same local time in the new execution. Since rop_1 returns x, while rop_0 returns $\perp$, this contradicts linearizability. We now present the details of the formal proof.

Let $b = \lceil |\mathbf{W}_{\mathcal{A}}(X)|(\delta)/\min\{\delta, u\} \rceil$. By the serial specification of X, there exists an admissible δ-execution σ of $\mathcal{A}$ consisting of the following operations at processes p_i, p_j, and p_k:

- for each l, $0 \leq l \leq b$, p_i performs a read operation $rop_i^{(2l)}$ on X with $t_\sigma^c(rop_i^{(2l)}) = l \min\{\delta, u\}$;
- for each l, $0 \leq l \leq b$, p_j performs a read operation $rop_j^{(2l+1)}$ on X with $t_\sigma^c(rop_j^{(2l+1)}) = l \min\{\delta, u\} + \min\{\delta, u\}/2$;
- p_k performs a write operation wop_k on X with $t_\sigma^c(wop_k) = \min\{\delta, u\}/2$ and $val(wop_k) = x$.

We assume that the message delays in σ are as follows. Each message from p_i to p_l, $l \neq i$, incurs a delay of either d if $l = j$ or $d - \min\{\delta, u\}/2$ if $l \neq j$; each message from p_j to p_l, $l \neq j$, incurs a delay of either $d - \min\{\delta, u\}$ if $l = i$ or $d - \min\{\delta, u\}/2$ if $l \neq i$; each message from p_l to p_i or p_j, $l \notin \{i, j\}$, incurs a delay of $d - \min\{\delta, u\}/2$. Moreover, we assume that all local clocks are perfectly synchronized in execution σ.

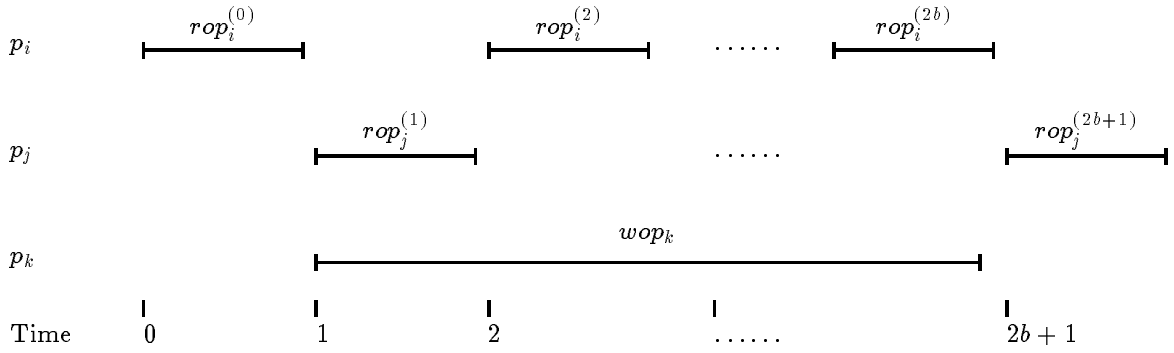
Figure 5(a) depicts the execution σ, where time runs from left to right, each horizontal line represents events at a single process and time points that are used in the proof are marked at the bottom.

Since $\mathcal{A}$ is linearizable, there exists a legal linearization τ of σ such that for each MCS process p_l, $ops(\sigma) \mid l = \tau \mid l$. The following sequence of simple claims describes the sequence τ.

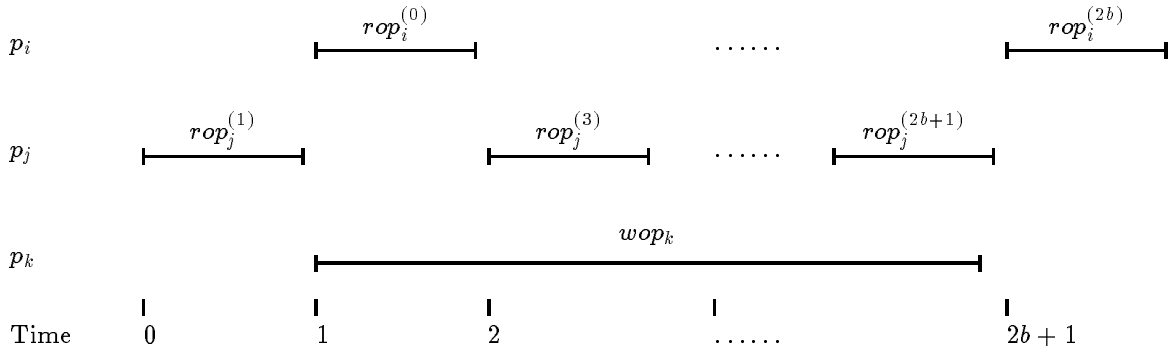
Claim 5.9 $rop_i^{(0)} \xrightarrow{\tau} wop_k$

Proof: Clearly,

$$\begin{aligned} t_\sigma^r(rop_i^{(0)}) &\leq t_\sigma^c(rop_i^{(0)}) + |\mathbf{R}_{\mathcal{A}}(X)|(\delta) \\ &\quad \text{(by definition of } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)\text{)} \\ &< 0 + \frac{\min\{\delta, u\}}{2} \\ &\quad \text{(by construction of } \sigma \text{ and assumption on } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)\text{)} \\ &= t_\sigma^c(wop_k) \\ &\quad \text{(by construction of } \sigma\text{)}. \end{aligned}$$



(a) The execution σ



(b) The execution σ'

Figure 5: The executions σ and σ'. Time is measured in units of $\min\{\delta, u\}/2$

Hence, by definition of $\xrightarrow{\sigma}$, $rop_i^{(0)} \xrightarrow{\sigma} wop_k$, which implies, by definition of linearization, that $rop_i^{(0)} \xrightarrow{\tau} wop_k$, as needed. ■

We continue by showing:

Claim 5.10 $wop_k \xrightarrow{\tau} rop_j^{(2b+1)}$

Proof: Clearly,

$$\begin{aligned} t_\sigma^r(wop_k) &\leq t_\sigma^c(wop_k) + |\mathbf{W}_{\mathcal{A}}(X)|(\delta) \\ &\quad \text{(by definition of } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\ &\leq \frac{\min\{\delta, u\}}{2} + b\min\{\delta, u\} \\ &\quad \text{(by construction of } \sigma \text{ and definition of } b) \\ &= t_\sigma^c(rop_j^{(2b+1)}) \\ &\quad \text{(by construction of } \sigma). \end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma}$, $wop_k \xrightarrow{\sigma} rop_j^{(2b+1)}$, which implies, by definition of linearization, that $wop_k \xrightarrow{\tau} rop_j^{(2b+1)}$, as needed. ■

For each r, $0 \leq r \leq 2b+1$, let $rop^{(r)} = rop_i^{(r)}$ if r is even, or $rop_j^{(r)}$ if r is odd. We show:

Claim 5.11 *For each* r, $0 \leq r \leq 2b$, $rop^{(r)} \xrightarrow{\tau} rop^{(r+1)}$

Proof: Clearly, for any r, $0 \leq r \leq 2b$,

$$\begin{aligned} t_\sigma^r(rop^{(r)}) &\leq t_\sigma^c(rop^{(r)}) + |\mathbf{R}_{\mathcal{A}}(X)|(\delta) \\ &\quad \text{(by definition of } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)) \\ &< t_\sigma^c(rop^{(r)}) + \frac{\min\{\delta, u\}}{2} \\ &\quad \text{(by assumption on } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)) \\ &= t_\sigma^c(rop^{(r+1)}) - \frac{\min\{\delta, u\}}{2} + \frac{\min\{\delta, u\}}{2} \\ &\quad \text{(by construction of } \sigma) \\ &= t_\sigma^c(rop^{(r+1)}). \end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma}$, $rop^{(r)} \xrightarrow{\sigma} rop^{(r+1)}$, which implies, by definition of linearization, that $rop^{(r)} \xrightarrow{\tau} rop^{(r+1)}$, as needed. ■

It follows by Claims 5.9, 5.10 and 5.11 that there exists an index r_0, $0 \leq r_0 \leq 2b$, such that $rop^{(r_0)} \xrightarrow{\tau} wop_k \xrightarrow{\tau} rop^{(r_0+1)}$. Since τ is a legal sequence of operations, it follows that $val_\sigma(rop^{(r_0)}) = \perp$ and $val_\sigma(rop^{(r_0+1)}) = x$. Assume, without loss of generality, that r_0 is even, so that $rop^{(r_0)}$ is a read operation by process p_i.

We now show how to "perturb" σ to obtain another admissible δ-execution σ' of $\mathcal{A}$ that is not linearizable. Define the real vector $\vec{s} = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ as follows. For each index $l \in [n]$, s_l is equal to $-\min\{\delta, u\}/2$ if $l = i$, $\min\{\delta, u\}/2$ if $l = j$, and 0 otherwise. Then, $\sigma' = shift(\sigma, \vec{s})$ with clocks $\Gamma' = shift(\Gamma, \vec{s})$. That is, each event at process p_i that occurs at real time t in σ will occur at real time $t + \min\{\delta, u\}/2$ in σ', each event at process p_j that occurs at real time t in σ will occur at real time $t - \min\{\delta, u\}/2$ in σ', and times of events at all other processes remain unchanged; p_i's local clock is shifted backward by $\min\{\delta, u\}/2$, p_i's local clock is shifted forward by $\min\{\delta, u\}/2$, and all other clocks remain unchanged. The execution σ' is depicted in Figure 5(b), using the same conventions as in Figure 5(a).

By Lemma 2.11, it follows that:

Lemma 5.12 *σ' is an execution of $\mathcal{A}$ with clocks Γ' that is equivalent to σ with clocks Γ.*

We proceed to show:

Lemma 5.13 *σ' is an admissible δ-execution of $\mathcal{A}$.*

Proof: Since all local clocks are perfectly synchronized in execution σ and

$$|\|\vec{s}\|_{\max} - \|\vec{s}\|_{\min}| \;=\; |\frac{\min\{\delta,u\}}{2} - (-\frac{\min\{\delta,u\}}{2})| \;=\; 2\,\frac{\min\{\delta,u\}}{2} \;=\; \min\{\delta,u\} \;\leq\; \delta\,,$$

Lemma 2.13 immediately implies that σ' is a δ-execution. We continue to show that all delays are in the range $[d-u, d]$. Fix any MCS processes p_l and p_m. Let d_{lm} be the delay of any message $\mathtt{m}$ from p_l to p_m in σ; By Lemma 2.12, the delay d'_{lm} of $\mathtt{m}$ in σ' is $d_{lm} + s_l - s_m$. Clearly, if $l \notin \{i, j\}$ and $m \notin \{i, j\}$, so that $s_l = s_m = 0$, then $d'_{lm} = d_{lm}$. We proceed to consider all remaining cases.

1. Assume that $l = i$ and $m = j$, so that $d_{lm} = d$, $s_l = -\min\{\delta, u\}/2$, and $s_m = \min\{\delta, u\}/2$. Then, $d'_{lm} = d - \min\{\delta, u\}$.
2. Assume that $l = j$ and $m = i$, so that $d_{lm} = d - \min\{\delta, u\}$, $s_l = \min\{\delta, u\}/2$, and $s_m = -\min\{\delta, u\}/2$. Then, $d'_{lm} = d$.
3. Assume that $l = i$ and $m \neq j$, so that $d_{lm} = d - \min\{\delta, u\}/2$, $s_l = -\min\{\delta, u\}/2$, and $s_m = 0$. Then, $d'_{lm} = d - \min\{\delta, u\}$.
4. Assume that $l = j$ and $m \neq i$, so that $d_{lm} = d - \min\{\delta, u\}/2$, $s_l = \min\{\delta, u\}/2$, and $s_m = 0$. Then, $d'_{lm} = d$.

5. Assume that $m = i$ and $l \neq j$, so that $d_{lm} = d - \min\{\delta, u\}/2$, $s_l = 0$, and $s_m = -\min\{\delta, u\}/2$. Then, $d'_{lm} = d$.

6. Assume that $m = j$ and $l \neq i$, so that $d_{lm} = d - \min\{\delta, u\}/2$, $s_l = 0$, and $s_m = \min\{\delta, u\}/2$. Then, $d'_{lm} = d - \min\{\delta, u\}$.

Since $d - u \leq d - \min\{\delta, u\} \leq d$, it follows that in all cases $d'_{lm} \in [d-u, d]$, as needed. This completes the proof that σ' is an admissible δ-execution of $\mathcal{A}$. ■

Since $\mathcal{A}$ is linearizable, there exists a legal linearization τ' of σ' such that for each MCS process p_l, $ops(\sigma') \mid l = \tau' \mid l$. We show:

Claim 5.14 $rop^{(r_0+1)} \xrightarrow{\tau'} rop^{(r_0)}$

Proof: Clearly,

$$\begin{aligned}
t^r_{\sigma'}(rop^{(r_0+1)}) &\leq t^c_{\sigma'}(rop^{(r_0+1)}) + |\mathbf{R}_{\mathcal{A}}(X)|(\delta) \\
&\quad \text{(by definition of } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)) \\
&< t^c_{\sigma'}(rop^{(r_0+1)}) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by assumption on } |\mathbf{R}_{\mathcal{A}}(X)|(\delta)) \\
&= t^c_{\sigma}(rop^{(r_0+1)}) - \frac{\min\{\delta, u\}}{2} + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma') \\
&= t^c_{\sigma}(rop^{(r_0+1)}) \\
&= t^c_{\sigma}(rop^{(r_0)}) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma) \\
&= t^c_{\sigma'}(rop^{(r_0)}) \\
&\quad \text{(by construction of } \sigma').
\end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma'}$, $rop^{(r_0+1)} \xrightarrow{\sigma'} rop^{(r_0)}$, which implies, by definition of linearization, that $rop^{(r_0+1)} \xrightarrow{\tau'} rop^{(r_0)}$, as needed. ■

However, Lemma 5.12 implies that $val_{\sigma'}(rop^{(r_0+1)}) = x$ and $val_{\sigma'}(rop^{(r_0)}) = \perp$; since τ' is a legal operation sequence, this implies that $rop^{(r_0)} \xrightarrow{\tau'} rop^{(r_0+1)}$. A contradiction. ■

We remark that the general structure of the proof of Theorem 5.8 follows the one of [15, Theorem 3.1] showing a lower bound of $u/4$ for the imperfect clocks model. However, due to the more delicate timing assumptions in the approximately synchronized clocks model, our proof has required more careful timing arguments. Our improvement over [15, Theorem 3.1] is achieved by carefully choosing message delays in shifting process histories.

5.3 Write Operations

We finally show that, under reasonable assumptions on the sharing pattern of processes, in any linearizable implementation of read/write objects including one with at least two writers and a distinct reader, the worst-case response time for a write operation is at least $\min\{\delta, u\}/2$. The proof constructs an execution for which if write operations are too short, then linearizability can be violated by appropriately shifting process histories.

Theorem 5.15 *Assume X is an object with at least two writers and a distinct reader. Then, for the approximately synchronized clocks model, in any linearizable implementation $\mathcal{A}$ of X, $|\mathbf{W}_{\mathcal{A}}(X)|(\delta) \geq \min\{\delta, u\}/2$.*

Proof: Let p_i and p_j be two processes that write X, and let p_k be a process that reads X. Assume, by way of contradiction, that there exists a linearizable implementation $\mathcal{A}$ of X for which $|\mathbf{W}_{\mathcal{A}}(X)|(\delta) < \min\{\delta, u\}/2$. We construct an admissible δ-execution of $\mathcal{A}$ that is not linearizable.

An informal outline of our proof follows. We start with an execution in which p_i writes x_i to X, then p_j writes x_j to X, and finally p_k reads x_j from X. If p_i's process history is shifted later by $\min\{\delta, u\}/2$, while p_j's process history is shifted earlier by $\min\{\delta, u\}/2$, the result is an execution in which the write operation by p_j precedes the write operation by p_i, while p_k still "sees" the same events occurring at the same local time; thus, p_k still reads x_j from X, which contradicts linearizability. We now present the details of the formal proof.

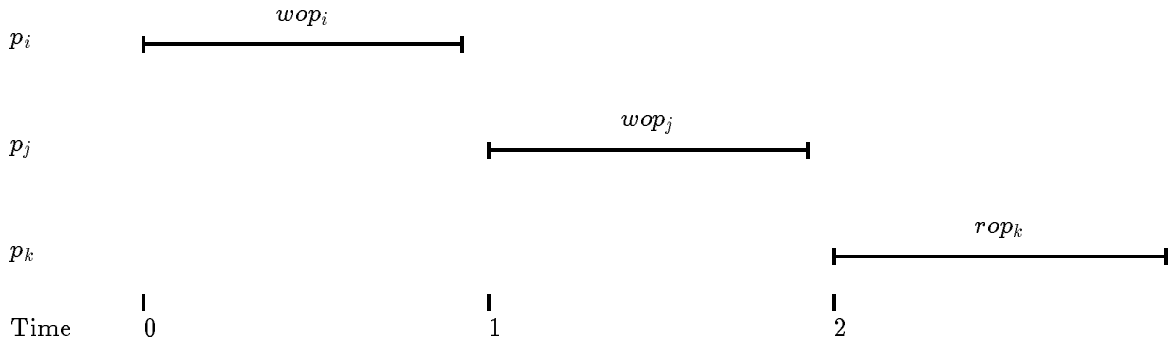
By the serial specification of X, there exists an admissible, synchronized execution σ of $\mathcal{A}$ consisting of the following operations at processes p_i, p_j and p_k:

- p_i performs a write operation wop_i on X with $t^c_\sigma(wop_i) = 0$ and $val(wop_i) = x_i$;
- p_j performs a write operation wop_j on X with $t^c_\sigma(wop_j) = \min\{\delta, u\}/2$ and $val(wop_j) = x_j$;
- p_k performs a read operation rop_k on X with $t^c_\sigma(rop_k) = \min\{\delta, u\}$.

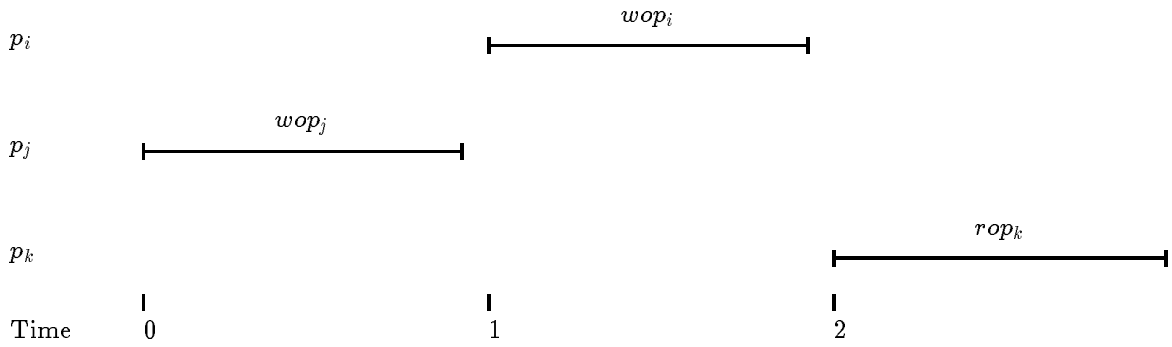
We assume that the message delays in σ are as follows. Each message from p_i to p_l, $l \neq i$, incurs a delay of either d if $l = j$ or $d - \min\{\delta, u\}/2$ if $l \neq j$; each message from p_j to p_l, $l \neq j$, incurs a delay of either $d - \min\{\delta, u\}$ if $l = i$ or $d - \min\{\delta, u\}/2$ if $l \neq i$; each message from p_l to p_i or p_j, $l \notin \{i, j\}$, incurs a delay of $d - \min\{\delta, u\}/2$. Moreover, we assume that all local clocks in Γ are perfectly synchronized.

Figure 5(a) depicts the execution σ, where time runs from left to right, each horizontal line represents events at a single process and time points that are used in the proof are marked at the bottom.

Since $\mathcal{A}$ is linearizable, there exists a legal linearization τ of σ such that for each MCS process p_l, $ops(\sigma) \mid l = \tau \mid l$. The following sequence of simple claims describes the sequence τ.



(a) The execution σ



(b) The execution σ'

Figure 6: The executions σ and σ'. Time is measured in units of $\min\{\delta, u\}/2$

Claim 5.16 $wop_i \xrightarrow{\tau} wop_j$

Proof: Clearly,

$$\begin{aligned} t^r_\sigma(wop_i) &\leq t^c_\sigma(wop_i) + |\mathbf{W}_{\mathcal{A}}(X)|(\delta) \\ &\quad \text{(by definition of } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\ &< 0 + \frac{\min\{\delta, u\}}{2} \\ &\quad \text{(by construction of } \sigma \text{ and assumption on } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\ &= t^c_\sigma(wop_j) \\ &\quad \text{(by construction of } \sigma). \end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma}$, $wop_i \xrightarrow{\sigma} wop_j$, which implies, by definition of linearization, that $wop_i \xrightarrow{\tau} wop_j$, as needed. ■

We continue to show:

Claim 5.17 $wop_j \xrightarrow{\tau} rop_k$

Proof: Clearly,

$$\begin{aligned} t^r_\sigma(wop_j) &\leq t^c_\sigma(wop_j) + |\mathbf{W}_{\mathcal{A}}(X)|(\delta) \\ &\quad \text{(by definition of } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\ &< 0 + \frac{\min\{\delta, u\}}{2} \\ &\quad \text{(by construction of } \sigma \text{ and assumption on } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\ &= t^c_\sigma(rop_k) \\ &\quad \text{(by construction of } \sigma). \end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma}$, $wop_j \xrightarrow{\sigma} rop_k$, which implies, by definition of linearization, that $wop_j \xrightarrow{\tau} rop_k$, as needed. ■

Since τ is a legal operation sequence, it follows by Claims 5.16 and 5.17 that $val_\sigma(rop_k) = val_\sigma(wop_j) = x_j$.

We now show how to "perturb" σ to obtain an admissible δ-execution σ' of $\mathcal{A}$ that is not linearizable. Define the real vector $\vec{s} = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ as follows. For each index $l \in [n]$, s_l is equal to $-\min\{\delta, u\}/2$ if $l = i$, $\min\{\delta, u\}/2$ if $l = j$ and 0 otherwise. Then, $\sigma' = shift(\sigma, \vec{s})$ with clocks $\Gamma' = shift(\Gamma, \vec{s})$. That is, each event at process p_i that occurs at real time t in σ will occur at real time $t + \min\{\delta, u\}/2$ in σ', each event at process p_j that occurs at real time t in σ will occur at real time $t - \min\{\delta, u\}/2$ in σ', and times of events at all other processes remain unchanged; p_i's local clock is shifted backward by $\min\{\delta, u\}/2$, p_j's local clock is shifted forward by $\min\{\delta, u\}/2$, and all other clocks remain unchanged. The execution σ' is depicted in Figure 6(b), using the same conventions as in Figure 6(a). By Lemma 2.11, it follows that:

Lemma 5.18 *σ' is an execution with clocks Γ' that is equivalent to σ with clocks Γ.*

We proceed to show:

Lemma 5.19 *σ' is an admissible δ-execution of $\mathcal{A}$.*

Proof: Since all local clocks are perfectly synchronized in execution σ and

$$|\|\vec{s}\|_{\max} - \|\vec{s}\|_{\min}| \;=\; |\frac{\min\{\delta,u\}}{2} - (-\frac{\min\{\delta,u\}}{2})\,| \;=\; 2\,\frac{\min\{\delta,u\}}{2} \;=\; \min\{\delta,u\} \;\leq\; \delta\,,$$

Lemma 2.13 immediately implies that σ' is a δ-execution. We continue to show that all delays are in the range $[d-u,d]$. Fix any MCS processes p_l and p_m. Let d_{lm} be the delay of any message $\mathtt{m}$ from p_l to p_m in σ. By Lemma 2.12, the delay d'_{lm} of $\mathtt{m}$ in σ' is $d_{lm} + s_l - s_m$. Clearly, if $l \notin \{i,j\}$ and $m \notin \{i,j\}$, so that $s_l = s_m = 0$, then $d'_{lm} = d_{lm}$, and d'_{lm} is in the range $[d-u,d]$ since d_{lm} is. We proceed to consider all remaining cases.

1. Assume that $l = i$ and $m = j$, so that $d_{lm} = d$, $s_l = -\min\{\delta,u\}/2$, and $s_m = \min\{\delta,u\}/2$. Then, $d'_{lm} = d - \min\{\delta,u\}$.
2. Assume that $l = j$ and $m = i$, so that $d_{lm} = d - \min\{\delta,u\}$, $s_l = \min\{\delta,u\}/2$, and $s_m = -\min\{\delta,u\}/2$. Then, $d'_{lm} = d$.
3. Assume that $l = i$ and $m \neq j$, so that $d_{lm} = d - \min\{\delta,u\}/2$, $s_l = -\min\{\delta,u\}/2$, and $s_m = 0$. Then, $d'_{lm} = d - \min\{\delta,u\}$.
4. Assume that $l = j$ and $m \neq i$, so that $d_{lm} = d - \min\{\delta,u\}/2$, $s_l = \min\{\delta,u\}/2$, and $s_m = 0$. Then, $d'_{lm} = d$.
5. Assume that $m = i$ and $l \neq j$, so that $d_{lm} = d - \min\{\delta,u\}/2$, $s_l = 0$, and $s_m = -\min\{\delta,u\}/2$. Then, $d'_{lm} = d$.
6. Assume that $m = j$ and $l \neq i$, so that $d_{lm} = d - \min\{\delta,u\}/2$, $s_l = 0$, and $s_m = \min\{\delta,u\}/2$. Then, $d'_{lm} = d - \min\{\delta,u\}$.

Since $d - u \leq d - \min\{\delta,u\} \leq d$, it follows that in all cases $d'_{lm} \in [d-u,d]$. This completes the proof that σ' is an admissible δ-execution of $\mathcal{A}$. ■

Since $\mathcal{A}$ is linearizable, Lemma 5.19 implies that there exists a legal linearization τ' of σ' such that for each MCS process p_l, $ops(\sigma') \mid l = \tau' \mid l$. The following sequence of simple claims describes the operation sequence τ'.

Claim 5.20 $wop_j \xrightarrow{\tau'} wop_i$

Proof: Clearly,

$$\begin{aligned}
t^r_{\sigma'}(wop_j) &\leq t^c_{\sigma'}(wop_j) + |\mathbf{W}_{\mathcal{A}}(X)|(\delta) \\
&\quad \text{(by definition of } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\
&< t^c_{\sigma'}(wop_j) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by assumption on } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\
&= t^c_{\sigma}(wop_j) - \frac{\min\{\delta, u\}}{2} + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma') \\
&= t^c_{\sigma}(wop_j) \\
&= t^c_{\sigma}(wop_i) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma) \\
&= t^c_{\sigma'}(wop_i) \\
&\quad \text{(by construction of } \sigma').
\end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma'}$, $wop_j \xrightarrow{\sigma'} wop_i$, which implies, by definition of linearization, that $wop_j \xrightarrow{\tau'} wop_i$, as needed. ■

We continue to show:

Claim 5.21 $wop_i \xrightarrow{\tau'} rop_k$

Proof: Clearly,

$$\begin{aligned}
t^r_{\sigma'}(wop_i) &\leq t^c_{\sigma'}(wop_i) + |\mathbf{W}_{\mathcal{A}}(X)|(\delta) \\
&\quad \text{(by definition of } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\
&< t^c_{\sigma'}(wop_i) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by assumption on } |\mathbf{W}_{\mathcal{A}}(X)|(\delta)) \\
&= t^c_{\sigma}(wop_i) - \frac{\min\{\delta, u\}}{2} + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma') \\
&= t^c_{\sigma}(wop_i) \\
&= t^c_{\sigma}(rop_k) + \frac{\min\{\delta, u\}}{2} \\
&\quad \text{(by construction of } \sigma) \\
&= t^c_{\sigma'}(rop_k) \\
&\quad \text{(by construction of } \sigma').
\end{aligned}$$

Hence, by definition of $\xrightarrow{\sigma'}$, $wop_i \xrightarrow{\sigma'} rop_k$, which implies, by definition of linearization, that $wop_i \xrightarrow{\tau'} rop_k$, as needed. ■

Since τ is a legal operation sequence, it follows by Claims 5.16 and 5.17 that $val_\sigma(rop_k) = val(wop_j) = x_j$. Since τ' is a legal operation sequence, it follows by Claims 5.20 and 5.21 that $val_{\sigma'}(rop_k) = val(rop_i) = x_i$. However, Lemma 5.18 implies that $val_{\sigma'}(rop_k) = val_\sigma(rop_k)$. A contradiction. ■

We remark that the general structure of the proof of Theorem 5.15 follows the one of [15, Theorem 3.2] showing a corresponding lower bound of $u/2$ for the imperfect clocks model. However, due to the more delicate timing assumptions in the approximately synchronized clocks model, our proof has required more careful timing arguments.

We can show that the algorithm in Theorem 3.1 for the perfect clocks model still works for the imperfect clocks model, and, hence, for the approximately synchronized clocks model too, if there are either a single reader and more than one writers or a single writer and more than one readers.(See [15, Section 3.1.1] for a corresponding observation.) This implies that the assumptions about the numbers of readers and writers made in Theorems 5.8 and 5.15, respectively, are necessary.

6 Imperfect Clocks

In this section, we state our upper and lower bounds for the imperfect clocks model.

6.1 Upper Bound

Fix any arbitrarily small constant ε subject to the constraint $0 < \varepsilon \leq \min\{2u, d-u\}$. Since the imperfect clocks model can be simulated by the approximately synchronized clocks model with $\delta = u$, Theorem 4.1 immediately implies:

Theorem 6.1 *For the imperfect clocks model, there exists a linearizable implementation $\mathcal{A}^{imp}$ of read/write objects that achieves $|\mathbf{R}_{\mathcal{A}^{imp}}|(\infty) < \beta d + 4u + \varepsilon$ and $|\mathbf{W}_{\mathcal{A}^{as}}|(\infty) \leq (1-\beta)d + 3u$, for any constant β such that $0 \leq \beta < 1 - u/d$.*

6.2 Lower Bounds

Theorem 5.6 immediately implies:

Theorem 6.2 *For the imperfect clocks model, in any sequentially consistent, object-separable and object-symmetric implementation $\mathcal{A}$ of at least three objects accessed by at least four processes,*

$$(|\mathbf{R}_{\mathcal{A}}| + |\mathbf{W}_{\mathcal{A}}|)(\delta) \geq d + \frac{u}{2}.$$

Since linearizability implies sequential consistency, Theorem 6.2 immediately implies:

Corollary 6.3 *For the imperfect clocks model, in any linearizable, object-separable and object-symmetric implementation $\mathcal{A}$ of at least three objects accessed by at least four processes,*

$$(|\mathbf{R}_{\mathcal{A}}| + |\mathbf{W}_{\mathcal{A}}|)(\delta) \geq d + \frac{u}{2}.$$

Theorem 5.8 immediately implies:

Theorem 6.4 *Assume X is an object with at least two readers and a distinct writer. Then, for the imperfect clocks model, in any linearizable implementation $\mathcal{A}$ of X, $|\mathbf{R}_{\mathcal{A}}(X)|(\infty) \geq u/2$.*

Finally, Theorem 5.15 immediately implies:

Theorem 6.5 *Assume X is an object with at least two writers and a distinct reader. Then, for the imperfect clocks model, in any linearizable implementation $\mathcal{A}$ of X, $|\mathbf{W}_{\mathcal{A}}(X)|(\infty) \geq u/2$.*

7 Discussion and Future Research

In this section, we provide a review of our results, a survey of related work, and directions for further research.

7.1 Review

We have shown a collection of lower and upper bounds for linearizable implementations of shared memory consisting of read/write objects, in models of perfect, imperfect, and approximately synchronized clocks. For the perfect clocks model, we presented a parameterized linearizable implementation, achieving worst-case response times of βd and $(1-\beta)d$ for read and write operations, respectively, where β is a trade-off parameter, $0 \leq \beta \leq 1$. For the approximately synchronized clocks model, our linearizable implementation achieves worst-case response times of less than $\beta d + 3u + \min\{delta, u\} + \varepsilon$, and of $(1-\beta)d + 3u$ for read and write operations, respectively, where $\epsilon > 0$ is an arbitrarily small constant. For the approximately synchronized clocks model, we also showed a lower bound of $d + \min\{\delta, u\}/2$ on the sum of these

worst-case response times, assuming certain symmetry properties for the implementations, and a lower bound of $\min\{\delta, u\}/2$ on the worst-case response times for read and write operations. Although there remains a gap between our upper and lower bounds for the approximately synchronized clocks model, we feel that our work substantially answers the question of how the time requirements for read and write operations depend on the timing uncertainties of this model, as measured by the parameters d and u and δ. In particular, we have shown that only a single "long communication" (i.e., a communication requiring time d) is required for both read and write operations, and this communication cannot be avoided.

This paper continues the complexity-theoretic study of the cost of implementing memory objects in a message-passing system, under various correctness conditions and timing assumptions, which was initiated in [10, 15, 41]. Although our model ignores several important practical issues, like, e.g., limitations on local memory size, clock drift, and "hot spots", we believe that our algorithms can be adapted to work in more realistic systems. We also believe that our results contribute to the understanding of the fine and intrinsic relation between sequential consistency and linearizability.

7.2 Related Work and Comparison

In this section, we review works that present time bounds for message-passing implementations of read/write objects under sequential consistency and linearizability. As those works are directly related to our work, we comment in detail on the relation between the results they provide and our results.

Attiya and Welch [15]

For the perfect clocks model, Attiya and Welch [15, Theorems 3.2 & 3.3] present a *fast read* linearizable implementation that guarantees time 0 for a read and time d for a write, and another *fast write* linearizable implementation that guarantees the reverse. These implementations are the special cases of our implementation where $\beta = 0$ and $\beta = 1$, respectively. Both the implementations in [15] and ours rely heavily on using timers and use messages that carry explicit timing information.

Attiya and Welch next consider the *imperfect clocks* model; they present [15, Theorems 4.5 and 4.6] a sequentially consistent implementation that guarantees time 0 for a read and time $2d$ for a write, and another sequentially consistent implementation that guarantees the reverse. Both of these implementations use as a subroutine a fast *atomic broadcast* algorithm they devise, but their modularity allows the use of any atomic broadcast algorithm like e.g., the one in [19]. They also show lower bounds of $u/4$ and $u/2$ for read and write operations, respectively.

Attiya and Friedman [12]

Attiya and Friedman [12] introduced a new hybrid condition for shared memory multiprocessors, called *hybrid consistency,* which combines the expressiveness of *strong* consistency conditions, like, e.g., sequential consistency and linearizability, and the efficiency of *weak* consistency conditions, like, e.g., pipelined RAM [41] and causal memory [7]. In hybrid consistency, memory access operations are classified as either *weak* or *strong*. Attiya and Friedman defined two versions of hybrid consistency, one based on sequential consistency and another based on linearizability; they presented a completely asynchronous message-passing implementation of hybrid consistency based on linearizability allowing for instantaneous weak operations while the response for strong operations is linear in the network delay d.

Chaudhuri, Gawlick and Lynch [20]

Building on our work, Chaudhuri, Gawlick and Lynch [20, Section 6] show how to simulate our algorithm for the perfect clocks model (Section 3) in the imperfect clocks model and obtain a linearizable algorithm for that model which is simpler than ours. Their algorithm achieves worst-case response times of $u + c$ and $d + u - c$ for read and write operations, respectively, where c is a trade-off constant between 0 and d. For purpose of comparison, set $c = \beta d$, where $0 \leq \beta \leq 1$, so that these bounds can be written as $\beta d + u$ and $(1 - \beta)d + u$, respectively. These bounds are more tight than ours in terms of the number of additive multiples of the message delay uncertainty u. However, since our algorithm for the perfect clocks model uses messages that carry explicit timing information, the simulation algorithm in [20] does so too; in this aspect, the simulation algorithm in [20] is inferior to a fairly obvious modification of our algorithm for the imperfect clocks model that uses messages of bounded size. Furthermore, we believe that the time-slicing technique used by our main algorithm not only provides more insight into the inherent difficulties of implementing linearizability in message-passing environments, but will also prove useful to efficiently solving other problems in distributed computing.

Kosa [37]

Kosa [37] considers the worst-case response time for operations on *abstract data types* and studies the combined effect of the amount of synchrony, the strength of the consistency guarantee and *algebraic* properties of the operations on this response time. For a wide variety of algebraic properties, Kosa extends the following results, already shown for read/write objects by results in this paper and in [15]:

- sequential consistency and linearizability are equally costly in the perfect clocks model;
- linearizability is more expensive in the imperfect clocks model than in the perfect clocks model ([37, Theorems 4.1 and 4.2] are shown using the same techniques as Theorems 5.15 and 5.8, respectively, in this paper);

- sequential consistency is cheaper than linearizability in the imperfect clocks model.

For sake of completeness and comparison, we summarize in Table 1 our main results and other related results known to us that provide similar bounds for message-passing implementations of sequential consistency and linearizability.

7.3 Future Research

Our work leaves open several interesting questions. Most obviously, it would be interesting to see if our bounds for the approximately synchronized clocks model, and, hence, for the imperfect clocks model, can be further improved. (Partial improvements have been presented in [20] for the case of upper bounds for the imperfect clocks model; see Section 7.2 for a description.)

Our results assume that clocks are available to processes; what if processes have no timing information at all and computations are completely asynchronous? What is the tightest coefficient of d bounding $|R| + |W|$ for sequentially consistent or linearizable implementations of read/write objects in this case? Also, it will be very interesting to obtain bounds on the worst-case response times of implementing other memory objects like, e.g., atomic snapshots [3], under sequential consistency and linearizability. How does strengthening of the shared memory primitives affect the worst-case response times? (Partial answers have been provided by Friedman [25].)

It would be interesting to examine the benefits of using timing information for implementing *hybrid consistency* [12] and see how the time requirements for performing weak and strong operations depend on the timing uncertainties of the model studied in this paper. Another interesting open question related to hybrid consistency is whether hybrid consistency based on sequential consistency allows for more efficient implementations than hybrid consistency based on linearizability. Lower and upper bounds shown in [12] imply that as far as *fast* implementations are considered, i.e., implementations for which the response times for weak read and weak write operations are both strictly less than $d/2$, there is no significant improvement in performance for hybrid consistency based on sequential consistency over hybrid consistency based on linearizability. However, results in this paper and in [15] suggest that a higher gain in performance might be possible if the implementation is not required to be fast.

A wide avenue for further research suggested by our work is the study of the costs of implementing sequentially consistent and linearizable objects in the presence of *partial synchrony*. The assumption of clocks that advance at the same rate, that of real time, is crucial for the results in this paper. It would be interesting to see what might be achieved if there were a known bound on the relative speeds of processors' clocks, or if no such bound existed. Some preliminary steps in this direction have been taken by Eleftheriou and Mavronicolas [22], in the context of the *drifting clocks* model and under different assumptions on message delays.

Timing Model	Correctness Condition	Cost Measure	Lower bounds	Upper bounds
Perfect Clocks ($\delta = u = 0$)	Sequential Consistency, Linearizability	$\|\mathbf{R}\|$	0	βd^* , $0 \le \beta \le 1$ ($\beta = 0$ and 1 also in [15])
		$\|\mathbf{W}\|$	0	$(1-\beta)d^*$, $0 \le \beta \le 1$ ($\beta = 0$ and 1 also in [15])
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	d [15, 41]	d (also in [15])
Approximately Synchronized Clocks ($0 < \delta < \infty$, $u > 0$)	Sequential Consistency	$\|\mathbf{R}\|$	0	$\downarrow$
		$\|\mathbf{W}\|$	0	$\downarrow$
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	$\uparrow$	$\downarrow$
	Linearizability	$\|\mathbf{R}\|$	$\min\{\delta, u\}/2^*$	$\beta d + 3u + \min\{\delta, u\} + \varepsilon^*$ $0 \le \beta < 1 - u/d$ and $0 < \varepsilon \le \min\{2u, d-u\}$ $\downarrow$
		$\|\mathbf{W}\|$	$\min\{\delta, u\}/2^*$	$(1-\beta)d + 3u^*$ $0 \le \beta < 1 - u/d$ $\downarrow$
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	$d + u/2^*$	$d + 6u + \min\{\delta, u\} + \varepsilon^*$, $0 < \varepsilon \le \min\{2u, d-u\}$ $\downarrow$
Imperfect Clocks ($\delta = \infty$, $u > 0$)	Sequential Consistency	$\|\mathbf{R}\|$	0	0 [15]
		$\|\mathbf{W}\|$	0	0 [15]
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	$\uparrow$	$2d$ [15]
	Linearizability	$\|\mathbf{R}\|$	$\uparrow$ $u/4$ [15]	$\beta d + 4u + \varepsilon,^*$ $0 \le \beta < 1 - u/d$ and $0 < \varepsilon \le \min\{2u, d-u\}$ $\beta d + u$, $0 \le \beta \le 1$ [20]
		$\|\mathbf{W}\|$	$\uparrow$ (also in [15])	$(1-\beta)d + 3u^*$ $0 \le \beta < 1 - u/d$ $(1-\beta)d + u$, $0 \le \beta \le 1$ [20]
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	$\uparrow$	$d + 7u + \varepsilon$, $0 < \varepsilon \le \min\{2u, d-u\}$ $d + 2u$ [20]
No Clocks	Linearizability	$\|\mathbf{R}\|$	—	$5d$ [12]
		$\|\mathbf{W}\|$	—	$5d$ [12]
		$\|\mathbf{R}\| + \|\mathbf{W}\|$	—	$10d$ [12]

Table 1: Summary of time bounds for message-passing implementations of read/write objects under sequential consistency and linearizability. Results marked by * are shown in this paper; references for other results are also given. An arrow $\uparrow$ (resp., $\downarrow$) indicates that the result follows from the corresponding result for the stronger (resp., weaker) timing model.

For additional work on memory consistency conditions and related issues of complexity, implementations, performance, verification and programming, the reader is referred to a substantial body of recent research [1, 2, 5, 6, 7, 8, 9, 11, 12, 13, 14, 25, 24, 26, 27, 28, 29, 33, 34, 35, 37, 40, 42, 43, 50, 53].

Acknowledgments:

Our work has been inspired and heavily influenced by the earlier pioneering work of Hagit Attiya and Jennifer Welch on a quantitative comparison of sequential consistency and linearizability [10, 15] under various timing assumptions. In particular, we would like to thank Hagit Attiya for making early versions of [10, 15] available to us and for helpful discussions. We owe special thanks to Harry Lewis for conjecturing the existence of implementations for the perfect clocks model falling between the extreme ones of Attiya and Welch [15]; these implementations subsequently led us to discover those for the approximately synchronized clocks model that trade the network latency cost between read and write operations. We are also thankful to Soma Chaudhuri, Maria Eleftheriou, Maurice Herlihy and Nancy Lynch for helpful discusions and comments, and to Roy Friedman, Martha Kosa, Yishay Mansour and the WDAG'92 Program Committee members and referees for their comments on earlier versions of our papers.

References

[1] S. Adve and M. Hill, "A Unified Formalization of Four Shared-Memory Models," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 4, No. 6, pp. 613–624, June 1993.

[2] S. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Computer,* Vol. 29, No. 12, pp. 66–76, December 1996.

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit, "Atomic Snapshots of Shared Memory," *Journal of the ACM,* Vol. 40, No. 4, pp. 873–890, September 1993.

[4] Y. Afek, G. Brown and M. Merritt, "Lazy Caching," *ACM Transactions on Programming Languages and Systems,* Vol. 15, No. 1, pp. 182–205, January 1993.

[5] D. Agrawal, M. Choy, H. V. Leong and A. K. Singh, "Mixed Consistency: A Model for Parallel Programming," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 101–110, August 1994.

[6] M. Ahamad, R. Bazzi, R. John, P. Kohli and G. Neiger, "The Power of Processor Consistency," *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 251–260, June/July 1993.

[7] M. Ahamad, J. Burns, P. Hutto and G. Neiger, "Causal Memory," *Distributed Computing,* Vol. 9, pp. 37–49, 1995.

[8] M. Ahamad, P. Hutto and R. John, "Implementing and Programming Causal Distributed Shared Memory," *Proceedings of the 11th International Conference on Distributed Computing Systems,* pp. 274–281, May 1991.

[9] R. Alur, K. McMillan and D. Peled, "Model-Checking of Correctness Conditions for Concurrent Objects," *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science,* pp. 219–228, July 1996.

[10] H. Attiya, "Implementing FIFO Queues and Stacks," *Proceedings of the 5th International Workshop on Distributed Algorithms (WDAG'91),* pp. 80–94, Lecture Notes in Computer Science, Vol. # 579, S. Toueg, P. G. Spirakis and L. Kirousis eds., Springer-Verlag, October 1991.

[11] H. Attiya, S. Chaudhuri, R. Friedman and J. L. Welch, "Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies," *SIAM Journal on Computing,* Vol. 27, No. 1, pp. 65–89, February 1998.

[12] H. Attiya and R. Friedman, "A Correctness Condition for High-Performance Multiprocessors," *Proceedings of the 24th Annual ACM Symposium on Theory of Computing,* pp. 679–690, May 1992.

[13] H. Attiya and R. Friedman, "Programming DEC-Alpha Based Multiprocessors the Easy Way," *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 157–166, June 1994.

[14] H. Attiya and R. Friedman, "Limitations of Fast Consistency Conditions for Distributed Shared Memory," *Information Processing Letters,* Vol. 57, No. 5, pp. 243–248, March 1996.

[15] H. Attiya and J. L. Welch, "Sequential Consistency versus Linearizability," *ACM Transactions on Computer Systems,* Vol. 12, No. 2, pp. 91–122, May 1994.

Preliminary version: *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 304–315, July 1991.

[16] H. Attiya and J. L. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw-Hill, 1998.

[17] H. Bal, M. Kaashoek, and A. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Transactions on Software Engineering,* Vol. 18, No. 3, pp. 190–205, March 1992.

[18] P. Bernstein, V. Hadzilacos and H. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading, Mass., 1987.

[19] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems,* Vol. 5, No. 1, pp. 47–76, February 1990.

[20] S. Chaudhuri, R. Gawlick and N. Lynch, "Designing Algorithms for Distributed Systems Using Partially Synchronized Clocks," *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing,* pp. 121–132, August 1993.

[21] B. Coan and G. Thomas, "Agreeing on a Leader in Real Time," *Proceedings of the 11th IEEE Real-Time Systems Symposium,* pp. 166–172, December 1990.

[22] M. Eleftheriou and M. Mavronicolas, "Linearizability in the Presence of Partial Synchrony and Under Different Delay Assumptions," preprint, Department of Computer Science, University of Cyprus, February 1998.

[23] *Encore 91 Series Technical Summary,* 1991.

[24] A. Fekete, F. Kaashoek and N. Lynch, "Providing Sequentially Consistent Shared Objects Using Group and Point-to-point Communication," *Journal of the ACM,* Vol. 45, No. 1, pp. 35–69, January 1998.

[25] R. Friedman, "Implementing High-Level Synchronization Operations in Hybrid Consistency," *Distributed Computing,* Vol. 9, No. 3, pp. 119–129, December 1995.

[26] K. Gharachorloo and P. Gibbons, "Detecting Violations of Sequential Consistency," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 316–326, June 1991.

[27] P. Gibbons and E. Korach, "On Testing Cache-Coherent Shared Memories," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 177–188, June 1994.

[28] P. Gibbons and M. Merritt, "Specifying Non-Blocking Shared Memories," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 292–303, July 1992.

[29] P. Gibbons, M. Merritt and K. Gharachorloo, "Proving Sequential Consistency of High-Performance Shared Memories," *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 292–303, June 1991.

[30] J. Halpern, N. Megiddo and A. Munshi, "Optimal Precision in the Presence of Uncertainty," *Journal of Complexity,* Vol. 1, pp. 170–196, 1985.

[31] M. Herlihy, "Wait-Free Implementations of Concurrent Objects," *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing,* pp. 276–290, August 1988.

[32] 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, July 1990.

[33] P. Hutto and M. Ahamad, "Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories," *Proceedings of the 10th International Conference on Distributed Computing Systems,* pp. 302–311, May 1990.

[34] M. Inoue, T. Masuzawa, and N. Tokura, "Efficient Linearizable Implementation of Shared FIFO Queues and General Objects on a Distributed System," *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences,* Vol. E81A, No. 5, pp. 768–775, May 1998.

[35] J. James and A. K. Singh, "Fault Tolerance Bounds for Memory Consistency," *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-97),* M. Mavronicolas and Ph. Tsigas, eds., pp. 200–214, Lecture Notes in Computer Science (Vol. # 1320), Springer-Verlag, Saarbrücken, Germany, September 1997.

[36] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers,* Vol. C-36, No. 8, pp. 933–939, August 1987.

[37] M. J. Kosa, "Making Operations of Concurrent Data Types Fast," *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing,* pp. 32–41, August 1994.

[38] 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, September 1979.

[39] L. Lamport, "On Interprocess Communication, Parts I and II," *Distributed Computing,* Vol. 1, No. 2, pp. 77–101, 1986.

[40] L. Lamport, "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor," DEC SRC Research Report # 96, dated February 14, 1993.

[41] R. Lipton and J. Sandberg, "A Scalable Shared Memory," Technical Report CS-TR-180-88, Princeton University, September 1988.

[42] B. Liskov, "Practical Uses of Synchronized Clocks in Distributed Systems," *Distributed Computing,* Vol. 6, pp. 211–219, August 1993.

[43] V. Luchangco, "Precedence-Based Memory Models," *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-97),* M. Mavronicolas and Ph. Tsigas, eds., pp. 215–229, Lecture Notes in Computer Science (Vol. # 1320), Springer-Verlag, Saarbrücken, Germany, September 1997.

[44] J. Lundelius and N. Lynch, "An Upper and Lower Bound for Clock Synchronization," *Information and Control,* Vol. 62, No. 2/3, pp. 190–204, August/September 1984.

[45] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly,* Vol. 2, No. 3, pp. 219–246, September 1989.

[46] M. Mavronicolas and D. Roth, "Sequential Consistency and Linearizability: Read/Write Objects," *Proceedings of the 29th Annual Allerton Conference on Communication, Control and Computing,* pp. 683–692, October 1991.

[47] M. Mavronicolas and D. Roth, "Efficient, Strongly Consistent Implementations of Shared Memory," *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92),* pp. 346–361, Lecture Notes in Computer Science, Vol. # 647, A. Segall and S. Zaks eds., Springer-Verlag, November 1992.

[48] J. Misra, "Axioms for Memory Access in Asynchronous Hardware Systems," *ACM Transactions on Programming Languages and Systems,* Vol. 8, No. 1, pp. 142–153, January 1986.

[49] C. Papadimitriou, "The Serializability of Concurrent Database Updates," *Journal of the ACM,* Vol. 26, No. 4, pp. 631–653, October 1979.

[50] F. Pong and M. Dubois, "The Verification of Cache Coherence Protocols," *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 11–20, June/July 1993.

[51] B. Simons, J. Welch and N. Lynch, *"An Overview of Clock Synchronization,"* IBM Technical Report RJ 6505, October 1988.

[52] R. Strong, D. Dolev and F. Christian, "New Latency Bounds for Atomic Broadcast," *Proceedings of the 11th IEEE Real-Time Systems Symposium,* pp. 156–165, December 1990.

[53] R. N. Zucker and J.-L. Baer, "A Performance Study of Memory Consistency Models," *Proceedings of the 19th ACM International Symposium on Computer Architecture,* pp. 2–12, May 1992.