Timing-Based, Distributed Computation: Algorithms and Impossibility Results

A thesis presented

by

Marios Mavronicolas

to

The Division of Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

July MCMXCII

to my parents

Abstract

Real distributed systems are subject to timing uncertainties: processes may lack a common notion of real time, or may even have only inexact information about the amount of real time needed for performing primitive computation steps. In this thesis, we embark on a study of the complexity theory of such systems and present combinatorial results that determine the inherent costs of some accomplishable tasks.

We first consider *continuous-time* models, where processes obtain timing information from continuous-time clocks that run at the same rate as real time, but might not be initially synchronized. Due to an uncertainty in message delay time, absolute process synchronization is known to be impossible for such systems. We develop novel synchronization schemes for such systems and use them for building a distributed, *full caching* implementation of shared memory that supports *linearizability*. This implementation improves in efficiency over previous ones that support consistency conditions even weaker than linearizability and supports a quantitative degradation of the less frequently occurring operation. We present lower bound results which show that our implementation achieves efficiency close to optimal.

We next turn to *discrete-time* models, where the time between any two consecutive steps of a process is in the interval $[c, 1]$, for some constant c such that $0 \leq c \leq 1$. We show time separation results between *asynchronous* and *semi-synchronous* such models, defined by taking $c = 0$ and $c > 0$, respectively. Specifically, we use the *session problem* to show that the semi-synchronous model, for which the timing uncertainty, $\frac{1}{c}$, is bounded, is strictly more powerful than the asynchronous one under either message-passing or shared-memory interprocess communication. We also present tight lower and upper bounds on the degree of *precision* that can be achieved in the semi-synchronous model.

Our combinatorial results shed some light on the capabilities and limitations of distributed systems subject to timing uncertainties. In particular, the main argument of this thesis is that the goal of designing distributed algorithms so that their logical correctness is timing-independent, whereas their performance might depend on timing assumptions, will not always be achievable: for some tasks, the only practical solutions might be strongly timing-dependent.

Acknowledgements

I am profoundly indebted to my thesis adviser, Professor Harry R. Lewis. In the context of this thesis, he has introduced me to timing-based computation and constantly provided support and constructive criticism. In a more general context, I am grateful to him for pointing out to me the importance of the usefulness of research in Theoretical Computer Science to practice, and, most important, for setting up a prime example of academic attitude.

I am mostly grateful to Hagit Attiya for teaching me so much about distributed computing in such a short time, for directing me to the right questions and problems, for her friendship and her constant interest for and concern with my work and progress. It is to her that I owe most of my understanding of the whole field.

Thanks to all three members of my thesis committee, Professors Harry R. Lewis and Petros Maragos, and Dr. Maurice Herlihy from DEC CRL. Special thanks to Petros for his friendship and to Maurice for inventing linearizability and extremely insightful comments.

Warm thanks are due to people who made direct contributions to this thesis. Parts of the results have been obtained jointly with Hagit Attiya and Dan Roth. I have also benefited from discussions (both physical and electronic) with Nancy Lynch, Yishay Mansour, Isaac Saias and Jennifer Welch. A series of motivating papers by Hagit Attiya and Nancy Lynch on timing-based distributed computation provided valuable and influential inspiration.

I owe deep appreciation for the tuition, advice and guidance that I received from Professors John E. Diamessis and John G. Fikioris during my early steps at National Technical University of Athens. They both have constantly been concerned with my academic evolution and progress. I also appreciate the tuition of Professors Donald G. M. Anderson and Michael O. Rabin at Harvard, and Alok Aggarwal and Gian-Carlo Rota at MIT. Part of my research was supported by ONR contract N00014-91-J-1981 through Professor Michael O. Rabin.

Thanks to all my friends who made my years in graduate school a wonderful and enriching experience. Maria Karayiorgou and Joseph Gogos deserve all my love for their friendship. I have been very fortunate to have the constant support and love of my entire family. Special thanks to my parents whom this thesis is dedicated to.

Contents

Chapter 1

Introduction

1.1 Motivation–Overview

An important application area of computer science is real-time process control, where a computer system interacts with a real-world system in order to guarantee certain desirable real-world behavior. In most interesting cases, the requirements for real-world behavior impose certain real-time conditions, and so the behavior of the computer system is required to obey certain timing constraints – its components, for example, should operate within given speeds, or some outputs should not be released before certain time elapses.

It is evident that good theoretical work in the emerging field of real-time systems is necessary. In the past few years, several researchers have proposed new theoretical frameworks for specifying requirements and describing implementations of such systems, and, also, proving that the implementations satisfy the requirements. These frameworks are built upon, among others, state machines ([39, 33]), weakest precondition methods ([55]), temporal logic ([19, 34, 27, 3]), Petri nets ([22]) and process algebra ([17, 59]). Much work is still needed in evaluating and comparing the various models for their usefulness in reasoning about important problems in the area of real-time systems and, perhaps, in developing new models if these prove to be inadequate.

Work is also needed in developing the complexity theory of real-time systems; very little

work has so far been done in this area. An example of the kind of work needed is provided by the theory of asynchronous concurrent systems in which processes work at completely independent and unpredictable rates and have no way of estimating time. That theory contains many combinatorial results that show what can and cannot be accomplished by asynchronous systems; for tasks that can be accomplished, other combinatorial results determine the inherent costs (see, e.g., [6, 25, 48, 58, 50]). In addition to their individual importance, these results also provide a testbed for evaluating decisions on modeling asynchronous concurrent systems, and a stimulus for the development of algorithm verification techniques. Similar results should be possible for real-time systems. (Some examples of complexity results that have already been obtained for real-time systems are the many results on clock synchronization, surveyed in [56].)

In this thesis, we embark on a study of the complexity theory of real-time distributed systems. We aim at combinatorial lower and upper bound results that determine the inherent costs of tasks achievable by such systems. Our mission is to exemplify to the greatest possible extent the precise effect of the timing assumptions used to model real-time distributed systems on the time complexity of distributed algorithms that can be supported and based, perhaps, on timing assumptions.

We divide our study into two major parts according to the kind of these timing assumptions. The first major part involves timing assumptions of a *continuous* nature: processes obtain timing information from continuous, real-time clocks that run at the same rate as real time, but might not be initially synchronized; all message delays are in the range $[d-u, d]$, for some known constants u and d, $0 \leq u < d$. We study the effect of such timing assumptions on the case cost of supporting memory correctness conditions of varying strength in implementations of a global shared memory on distributed memory multiprocessor machines. We present novel synchronization schemes and build timing-based algorithms on them which achieve essentially optimal efficiency in supporting strong correctness conditions. Our specific results are motivated and described in detail in Section 1.2, and laid out in Chapter 3, where the role of timing dependence in achieving efficiency is pointed out.

The second major part of our study involves timing assumptions of a *discrete* nature: processes have access to inaccurate clocks that operate at approximately, but not exactly, the same rate. Such systems capture the essence of the important middle ground between the synchronous and asynchronous models of distributed computation, where processes operate neither at lock-step nor at a completely independent rate. We model these partially synchronous systems by assuming that there exist a lower an an upper bound on process step time that may enable processes to estimate time. More specifically, we make the following timing assumptions: In the *asynchronous* model, processes have step time in $[0, 1]$; in the *semi-synchronous* model, processes have step time in $[c, 1]$, for some parameter c such that $0 < c \leq 1$*. We assume that in these models interprocess communication can be achieved through either point-to-point message passing or accessing and modifying variables of shared memory, but not through both. We show time separation results between semi-synchronous (in particular, synchronous) and asynchronous models under both message-passing and shared-memory communication. We present lower bound results which show the inherent limitations on using timing information in the semi-synchronous model. We also study the problem of obtaining close estimates of real time in the semi-synchronous model. In Section 1.3, our results are motivated and described in detail; they are fully presented in Chapters 4 and 5.

Throughout this thesis, special emphasis is placed on formal techniques for defining problems (correctness conditions), describing algorithms and constructing correctness proofs, and deriving impossibility results.

1.2 The Power of Synchronization

The shared-memory model has been proven a useful model of logically shared data in concurrent computation. Perhaps, this is due to its success in allowing 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

*The *synchronous* model is a special case of the semi-synchronous model where $c = 1$.

paradigm of an interprocess communication model, as it provides to 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; their performance can be measured in terms of, e.g., 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, and 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* (cf. [28]); otherwise, it is said to guarantee *sequential consistency* (cf. [32]). Clearly, linearizability implies sequential consistency. It has been argued quite convincingly ([28]) that linearizability is the correctness condition that most appropriately guarantees "acceptable" concurrent behavior; indeed, linearizability enjoys a number of nice properties, like, e.g., locality[†], and 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 ([13]) 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. Specifically, they considered caching implementations of read/write objects in non-bused distributed systems, and took as cost the *worst-case response time* of performing

[†]Roughly speaking, a correctness condition is said to be *local* if it is possible to compose independent implementations of it and obtain an implementation of a larger set of objects.

an operation on such objects in the best possible implementation supporting each of the consistency conditions.

In this thesis, we continue this study and show how the timing assumptions of the continuous-time model can be used to obtain upper bounds on the same cost for new, efficient linearizable implementations. As a side effect, these bounds further enlighten the advantages of linearizability over other, seemingly "cheaper", correctness conditions, such as sequential consistency.

We follow [13] 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 send calls to access shared data to the corresponding MCS process; the MCS process responds to such a request, based, perhaps, on information from messages it receives from other MCS processes. In doing so, the MCS must provide the proper read/write semantics, with respect to the values returned to application programs, throughout the network.

We first consider the *perfect clocks* model, where processes have perfectly synchronized clocks and the message delays are constant, i.e., $u = 0$[§]. We present a family of linearizable implementations, parameterized by some tunable constant μ, $0 \leq \mu \leq 1$, for which the worst-case response times for read and write operations are μd and $(1 - \mu)d$, respectively, both dependent on the network's latency; the parameter μ precisely quantifies these dependencies and may be appropriately chosen in order, e.g., to degrade the less frequently occurring

[‡]The assumption of a complete communication network, made only for simplicity, is not necessary for our results and can be removed.

[§]As Attiya and Welch remark (cf. [13]), the assumptions that processes have perfectly synchronized clocks and that message delays are constant (and known to the processes) are equivalent.

operation. A read operation simply returns after time μd, while a write operation returns after time $(1-\mu)d$. Our implementations naturally generalize those in Theorems 3.2 and 3.3 of [13], which are the special cases where $\mu = 0$ and $\mu = 1$, respectively. By a result of Lipton and Sandberg ([35]) showing a lower bound of d on the sum of the worst-case response times for read and write operations in any linearizable implementation, each of our implementations is optimal.

We next turn to the more realistic *imperfect clocks* model, where clocks are not initially synchronized and message delays can vary, i.e., $u > 0$.

Our first major result is the first known linearizable implementation of read/write objects for this model, which achieves worst-case response times of less than $4u + \epsilon$ ($\epsilon > 0$ is an arbitrarily small constant) and of $d + 3u$ for read and write operations, respectively. A read operation may return only after a value has resided for time at least u in the local memory of a process. For a write operation, a "time-slicing" technique is used. Each process individually slices time using its own local clock and, once in the appropriate "time slice", it broadcasts the value to be written and waits for an additional time of d before acknowledging. 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 to, but, instead, it is chosen among values of write operations on the same object performed by processes within a small, recent time interval; this choice is based on information shown to be "known" to all processes. As we show, this results not only in preserving the order in which values are returned by different reading processes, but also in maintaining consistent copies of local memory throughout the network. As it turns out, this implies linearizability.

Our second major result is a modification of our previous implementation which achieves a quantitative degradation of the less frequently occurring operation, in a way similar to the implementation for the perfect clocks model. A read operation is delayed for time μd; then, it may return only after a value has resided for time at least u in the local memory of a process. (The value to be returned is chosen as in our previous implementation.) It is assumed that $0 \leq \mu < \frac{d-u}{d}$. For a write operation, a process, once in the appropriate

time slice, it broadacsts the value to be written and waits for an additional $(1-\mu)d$ time before acknowledging. This implementation achieves worst-case response times of less than $\mu d + 4u + \epsilon$ ($\epsilon > 0$ is an arbitrarily small constant) and of $(1-\mu)d + 3u$ for read and write operations, respectively.

Both previous implementations presuppose a run of an initialization phase, during which processes follow a simple synchronization algorithm to achieve an accuracy of u[¶] on which the implementations heavily rely in order to exploit the known lower bound of $d-u$ on message delay time and achieve bounds on worst-case response times which, unlike previous ones, depend on the message delay uncertainty u.

In the case where this uncertainty is sufficiently small, these implementations significantly outperform the ones by Attiya and Welch ([13]) which supported an even weaker consistency condition, namely sequential consistency; these latter implementations, not particularly tailored for a model in which a lower bound on message delay time is assumed, exploit only the (known) upper bound of d on message delay time and achieve worst-case response times for read and write operations whose sum amounts to $4d$ or $6d$. Most important, for all values of the parameters u and d, our implementations are more efficient than the ones proposed by Attiya and Friedman ([8]), when specialized to support only *strong* operations and guarantee linearizability; these latter implementations do not assume any non-zero lower bound on message delay time, and achieve worst-case response times for strong read and write operations whose sum amounts to $10d$.

The dependence of our upper bounds on d 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 [35], is inherent. Furthermore, although the analysis of our implementations is technically challenging, the implementations themselves are fairly simple to implement, they do not use complicated control mechanisms, and they are message-economical.

To argue optimality of our implementations, we present lower bound results on costs of

[¶]That is, the maximum difference between local times of any two processes in the system at the same real time is at most u.

consistent implementations of shared memory, under general and mild assumptions on the pattern of sharing properties of processes. Our main impossibility result is a lower bound of $d+\frac{u}{2}$ on the sum of the worst-case response times for any sequentially consistent implementation in which processes handle operations on each object in a "symmetric" way (with respect to the objects) and quite independently of operations on different objects. This implies a corresponding lower bound for linearizable implementations. We also show a lower bound of $\frac{u}{2}$ on the worst-case response time for a read operation in any linearizable implementation. The proof of the first lower bound combines symmetry arguments with the technique of "shifting" executions, originally introduced in [36]. The proof of the second lower bound makes use of this technique as well, to improve upon a corresponding lower bound of $\frac{u}{4}$ presented in [13].

Our results for the imperfect clocks model, in particular, the upper bound of $d + \Theta(u)$ on the sum of the worst-case response times for read and write operations in a linearizable implementation, along with the lower bound of $d + \Theta(u)$ on this sum, may suggest that sequential consistency and linearizability are actually "closer" than thought before.

Our work for the continuous-time model continues the complexity-theoretic study of the costs of implementing memory objects on distributed memory multiprocessor machines under various correctness conditions. This study was initiated in [35, 13] and further pursued in [5].

1.3 Semi-Synchrony versus Asynchrony

Central to the programming of distributed systems are *synchronization* problems, where a process is required to guarantee that *all* processes have performed a particular set of steps. Naturally, the timing information available to processes has critical impact on the time complexity of synchronization. Arjomandi, Fischer and Lynch ([4]) introduced the *session problem* in order to study the impact of timing information on the time complexity of synchronization. Roughly speaking, a *session* is a sequence of events that contains at least one step by each process. An algorithm for the *s-session problem* guarantees that each execution of the algorithm includes at least s disjoint sessions.

The session problem is an abstraction of the synchronization needed for the execution of

some tasks that arise in a distributed system, where separate components are each responsible for performing a small part of a computation. Consider, for example, a system which solves a set of equations by successive relaxation, where every process holds part of the data (cf. [18]). Interleaving of steps by different processes is necessary in order to ensure that a correct value was computed, since it implies sufficient interaction among the intermediate values computed by the processes. Any algorithm which ensures that sufficient interleaving has occurred also solves the s-session problem. The session problem is also an abstraction of some problems in real-time computing which involve synchronization of several computer system components, in order that they cooperate in performing a task involving real-world components. For example, multiple robots might cooperate to build a car on an assembly line, with each robot responsible for assembling a small piece of the machinery. Interleaving of assembly actions by different robots is necessary to ensure that pieces are assembled in the right order; a robot should not put the next item on the assembly line before all robots have completed a particular set of assembly actions making it possible for the item to "fit in". Clearly, any algorithm which ensures that sufficient interleaving has occurred also solves the s-session problem. Thus, the difficulty of solving the s-session problem reflects those of implementing the successive relaxation method and building the car on the assembly line.

Arjomandi, Fischer and Lynch ([4]) assumed that processes communicate via *shared variables* and studied the time complexity of the session problem in synchronous and asynchronous models. Informally, in a *synchronous* system, processes operate in lock-step, taking steps simultaneously, while in an *asynchronous* system, processes work at completely independent rates and have no way of estimating time. The results of Arjomandi, Fischer and Lynch ([4]) show that there is a significant gap between the time complexities of solving the session problem in the synchronous and the asynchronous models.

Following Arjomandi, Fischer and Lynch ([4]), we address the cost of synchronization in semi-synchronous and asynchronous systems by presenting upper and lower bounds on the time complexity of solving the s-session problem. Informally, the *time complexity* of an algorithm is the maximal time, over all executions, until every process stops executing the

algorithm. Our study considers two different, major interprocess communication models: *networks* and *shared memory*.

In a network, a collection of n processes are arranged at the nodes of an undirected graph G and communicate by sending messages along links of this graph. Messages sent over any communication link incur a delay in the range $[0, d]$, where $d \geq 0$ is a known constant.

Interpocess communication can also be achieved through shared memory. Following [4], we consider a particular shared memory primitive, *b-atomic registers*, where the integer $b > 1$ is an upper bound on the number of processes that may instantaneously and indivisibly access (read and, possibly, modify) each of the registers. Thus, b reflects the communication bound in the model and captures communication limitations of existing distributed systems such as those of a message-passing system which allows access of buffers with finite fan-in.

1.3.1 Networks

We first consider the *initialized* case of the session problem, where all processes are initially synchronized and take a step at time 0. We start with upper bounds. The first algorithm relies on explicit communication to ensure that the needed steps have occurred and does not use any timing information. In the asynchronous model, this algorithm has time complexity $diam(G)(d+1)(s-1)$, where the *diameter*, $diam(G)$, of an undirected graph G is the *maximum* distance between any two nodes. In the semi-synchronous model, this algorithm can be improved to take advantage of the initial synchronization and achieve a time complexity of $1 + diam(G)(d+1)(s-2)$. The second algorithm does not use any communication and relies only on timing information; it works only in the semi-synchronous model. The time complexity of this algorithm is $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$. These algorithms can be combined to yield a semi-synchronous algorithm for the s-session problem whose time complexity is $1 + \min\{\lfloor \frac{1}{c} \rfloor + 1, diam(G)(d+1)\}(s-2)$.

We then present lower bounds. For the asynchronous model, we prove an almost matching lower bound of $diam(G)d(s-1)$ on the time complexity of any algorithm for the s-session problem. For the semi-synchronous model, we prove two lower bounds. We first show a

simple lower bound of $\lfloor \frac{1}{c}(s-2) \rfloor$ for the case where no communication is used. We then present a lower bound of $1 + \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-2)$ for the time complexity of any semi-synchronous algorithm for the s-session problem; the proof relies on the assumption that $d \geq \frac{d}{\min\{\lfloor 1/2c \rfloor, diam(G)d\}} + 2$.

These bounds extend in a straightforward way to the case where delays on the communication links of $G = (V, E)$ are not uniform. That is, for every communication link $(i, j) \in E$, delivery time for a message sent over (i, j) is in the interval $[0, d(i, j)]$ for some fixed $d(i, j)$, $0 \leq d(i, j) < \infty$. (The study of this non-uniform case is partially inspired by recent work on cost-sensitive analysis of communication protocols appearing in [15].)

We also consider the *uninitialized* case, where processes are not initially synchronized. Specifically, at time 0 all processes but one, the *initiator*, start in a *quiescent* state. When in a quiescent state, a process can neither send out any messages nor pass to a non-quiescent state. However, upon a message delivery event, it may enter a non-quiescent, non-idle state. The results we obtain for this case are similar in flavor to those for the initialized case, although they are less tight.

We start with a straightforward adaptation of the asynchronous algorithm for the initialized case. The time complexity of this algorithm is $diam(G)(d+1)s$ in both the asynchronous and the semi-synchronous models. Our second algorithm combines some initial communication and timing information to achieve a time complexity of $3diam(G)(d+1)+(\lfloor \frac{1}{c} \rfloor + 1)(s-1)$; it works only in the semi-synchronous model. These algorithms can be combined to yield a semi-synchronous algorithm which solves the s-session problem on G in the uninitialized case and achieves a time complexity of $3diam(G)(d+1) + \min\{\lfloor \frac{1}{c} \rfloor + 1, diam(G)(d+1)\}(s-1)$. We also prove lower bounds for this case. For the asynchronous model, we prove an almost matching lower bound of $diam(G)ds$ for the time complexity of any algorithm for the s-session problem. For the semi-synchronous model we prove a lower bound of $diam(G)\lfloor d \rfloor + \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-1)$; the proof relies on the assumption that $d \geq \frac{d}{\min\{2, \lfloor 1/2c \rfloor, diam(G)d\}} + 2$.

For appropriate values of the various parameters (choose, for example, c, d and $diam(G)$

so that $\lfloor \frac{1}{2c} \rfloor < diam(G)d$, our results imply a time separation between semi-synchronous (in particular, synchronous) and asynchronous networks. The lower bound for the semi-synchronous model shows the inherent limitations on using timing information. In addition, it can also be used to derive a lower bound of $1 + diam(G)d(s-2)$ for a model in which processes' step time is in the range $(0, 1]$ (rather than in $[0, 1]$, as in the asynchronous model). This is equivalent to requiring that no two steps by the same process may occur at the same time[‖]. Fix some $c' > 0$ such that $\lfloor \frac{1}{2c'} \rfloor \geq diam(G)d$, and use the proof of the lower bound for the model where processes' step time is in the range $[c', 1]$; since $[c', 1] \subset (0, 1]$, the claim follows. This implies the first time separation between this model and the synchronous model. (The proof in [4] relies heavily on the ability to schedule many steps by the same process at the same time.) Note that our first (initialized) algorithm for the semi-synchronous model only requires that two steps by the same process do not occur at the same time. This implies that the almost matching upper bound of $1+diam(G)(d+1)(s-2)$ holds also for the asynchronous model where processes' step time is in the range $(0, 1]$.

The lower bounds shown for networks use the same general approach as in [4]. However, since we assume processes communicate by sending messages, while [4] assumes processes communicate via shared memory, the precise details differ substantially. The lower bound proof in [4] uses fan-in arguments, while our lower bounds are based on information propagation arguments using long delays of messages, combined with appropriate selection of processes and careful timing arguments. Our asynchronous lower bound is based and improves on a result of Lynch ([38]) showing a lower bound of $rad(G)d(s-1)$[**].

Awerbuch ([14]) introduced the concept of a *synchronizer* as a way to translate algorithms designed for synchronous networks to asynchronous networks. Although the results of [14] may suggest that *any* synchronous network algorithm can be translated into an asynchronous algorithm with constant time overhead, our results imply that this is *not* the case: for some values of the parameters, any translation of a semi-synchronous (in particular, synchronous)

[‖]We remark that this is the most common way of measuring time in an asynchronous system (e.g., [51]).

[**]The *radius*, $rad(G)$ of G is the *minimum*, over all nodes in V of the maximum distance from that node to any other node in V. For any undirected graph G, $rad(G) \leq diam(G) \leq 2rad(G)$.

algorithm for the s-session problem to an asynchronous algorithm *must* incur a non-constant time overhead.

Our results for networks are laid out in full in Section 4.1.

1.3.2 Shared Memory

We start by describing some simple upper bounds that can be deduced from either previous or our current work. An algorithm sketched in [4] relies on explicit communication through shared memory to ensure that the needed steps have occurred and does not use any timing information. This algorithm achieves time complexity of $O(s \log_b n)$ in both the asynchronous and the semi-synchronous models. On the other hand, since our semi-synchronous algorithm for networks does not use any communication, but relies on timing information, it also works for the shared memory model to achieve time complexity of $O(s\frac{1}{c})$. These two algorithms can be combined to yield a semi-synchronous algorithm for the s-session problem whose time complexity is $O(s \min\{\frac{1}{c}, \log_b n\})$. On the other hand, a lower bound of $\Omega(s \log_b n)$ shown in [4] holds for our asynchronous shared memory model as well and implies, for appropriate values of the various parameters, a time separation between semi-synchronous and asynchronous systems that use communication through atomic shared memory. (Choose, for example, c, b and n so that $\lfloor \frac{1}{2c} \rfloor < \lfloor \log_b(n-1) - 1 \rfloor$.)

At this point, it is natural to ask whether communication and timing information can be combined to yield an upper bound that is significantly better than $O(s \min\{\frac{1}{c}, \log_b n\})$. We show a lower bound of $1 + \min\{\lfloor \frac{1}{2c} \rfloor, \lfloor \log_b(n-1) - 1 \rfloor\}(s-2)$ for the time complexity of any semi-synchronous algorithm for the s-session problem[††]. This result shows the inherent limitations on using timing information and implies that such a combination is impossible.

As for networks, our lower bound result can also be used to derive a lower bound of $1 + \lfloor \log_b(n-1) - 1 \rfloor (s-2)$ for a shared memory model in which processes' step time is in the range $(0, 1]$ (rather than in $[0, 1]$, as in the asynchronous model). This is equivalent to

[††] An essentially identical lower bound has been obtained independently, but subsequently to ours, by Rhee and Welch (cf. [53]).

requiring that no two steps by the same process may occur at the same time. Fix some $c' > 0$ such that $\lfloor \frac{1}{2c'} \rfloor \geq \lfloor \log_b(n-1) - 1 \rfloor$, and use the proof of the lower bound for the model where the rate of processes steps is in $[c', 1]$; since $[c', 1] \subset (0, 1]$, the claim follows. This implies a time separation between this model and the synchronous shared-memory model. (We stress, again, that the proof in [4] relies heavily on the ability to schedule many steps by the same process at the same time.)

Our lower bound proof uses the same general approach as in [4] and our lower bound proof for semi-synchronous networks. Specifically, our proof combines fan-in and causality arguments as in [4], along with information propagation and careful timing arguments as in our lower bound proof for semi-synchronous networks.

Our results for shared memory are presented in detail in Section 4.2.

1.4 Semi-Synchrony versus Real Time

Central to the programming of distributed systems are *synchronization problems*, where processes are required to obtain some common notion of time so as to perform a particular action simultaneously. How closely can they be guaranteed to perform such an action?

Such synchronization problems were first investigated by Lamport in [31], where a simple algorithm was presented allowing a system of asynchronous processes to maintain a discrete clock that remains consistent with the ordering of receipt of communication messages by the processes. Several researchers have considered models of a distributed system in which processes have real-time clocks that run at the same rate as real time, but are arbitrarily offset from each other initially. In addition, there are known upper and lower bounds on message delay. (These models are essentially identical to the continuous-time model studied in Chapter 3.) The goal has been to prove limits on how closely clocks can be synchronized. In a completely connected network of n processes, Lundelius and Lynch ([36]) show a tight bound of $\eta(1 - \frac{1}{n})$ on how closely the clocks of n processes can be synchronized, where η is the difference between the bounds on the message delay. Their work was subsequently extended by Halpern, Megiddo and Munshi ([26]) to arbitrary networks. There has also been much

work done on the problem of devising fault-tolerant algorithms to synchronize real-time clocks that drift slightly in the presence of variable message delay.

We depart from conventional approaches to clock synchronization problems, focusing on models where processes obtain timing information from real-time (drifting or non-drifting) clocks, and address the problem of achieving coordinated action in semi-synchronous networks by studying the *tick synchronization problem*, which is the problem of achieving as close as possible time estimates by different processes in a semi-synchronous network.

The tick synchronization problem is an abstraction of the synchronization needed for the execution of some tasks that arise in a distributed system, where separate components need to agree on as a close as possible common value of real time. Consider, for example, version management and concurrency control problems for database systems; solutions to such problems heavily rely on the ability to assign timestamps and version numbers to files or other entities. Also, some algorithms that use timeouts, such as communication protocols, are strongly dependent on availability of a "common" notion of real time to communicating clients.

Roughly speaking, each process runs a tick synchronization algorithm, and enters, upon its completion, a *synchronized* state modelling the ability of the process to make use of the estimate of real time it has obtained from running the algorithm. Informally, the *precision* achieved by a tick synchronization algorithm is the maximum, over all processes p_i in the system, of the difference between the real-time estimate that p_i is making at precisely the time at which it is entering a synchronized state and the real-time estimate that any other process in a synchronized state is making at the same time.[‡‡]

More specifically, we consider a collection of n processes arranged at the nodes of a complete undirected graph G and communicating by sending messages along links of this graph. Messages sent over any communication link incur a delay in the range $[0, d]$, where $d \geq 0$ is a (known) constant. The time between any two computation steps of a process is in the

[‡‡]It is perhaps counter-intuitive that a precision of 0 is the best precision, under our definition, that can be achieved.

interval $[c, 1]$ for some parameter c such that $0 < c \leq 1$.

We show a lower bound of $\lfloor \frac{d-2}{2c} \rfloor$ on the precision achievable by any tick synchronization algorithm. Our proof follows a general technique, that of explicitly "shifting" and "shrinking" executions through retiming of events, reminiscent of a technique originally introduced in [36] and subsequently found applications in many different contexts, e.g., [10, 13]. (More instances of application of this technique appear also in Chapters 3 and 4 of this thesis.) Since, however, we are assuming that processes acquire information about time by receiving ticks from their inaccurate, discrete clocks, while [36] assumes processes have access to continuous clocks running at a perfect rate, that of real time, the precise details differ substantially. Thus, our lower bound proof focuses more on timing uncertainty rather than message delivery time uncertainty. Clearly, our lower bound is interesting when $d > 2$.

We also present a simple algorithm that achieves a precision of $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$. Our algorithm relies on explicit communication among the processes, so that each of them can estimate the difference between the local time estimate of every other process and its own, and add the average of these estimated differences to its local time estimate. This algorithm is a direct adaptation for the semi-synchronous model of one presented in [36]. Its analysis, however, is more involved, due to the fact that the timing assumptions in the semi-synchronous model are more "crisp" than the ones in [36], where clocks were assumed to run at a perfect rate, as in the continuous-time model.

Clearly, after all processes enter a synchronized state, clock drifts may bring the system out of synchronization again; so, it makes sense to consider the behavior of the system prior to the time at which the last process enters a synchronized state. Multiple runs of a tick synchronization algorithm, appropriately scheduled, possibly in a way similar to [37], may reduce such future "desynchronizations".

Our results for the tick synchronization problem are exhibited in Chapter 5.

Chapter 2

System Models

In this Chapter, we present descriptions and formal definitions for the *continuous-time* and *discrete-time* system models that we consider*.

Definitions for the continuous-time model appear in Section 2.1; they are tailored towards a message passing system supporting an implementation of shared memory on distributed memory machines, closely following [13, 5, 45, 46, 8]. Section 2.2 includes our definitions for the discrete-time model in which interprocess communication is achieved through either point-to-point message passing or shared memory, but not through both; these definitions are standard and similar in style to those in, e.g., [9, 12, 44, 11, 51, 4, 41].

2.1 Continuous-Time Model

We consider a collection of *application programs* running concurrently and communicating through virtual shared memory, consisting of a collection $\mathcal{X}$ of *read/write objects*, or simply *objects*, for short. Each object $X \in \mathcal{X}$ may attain values from a *domain*, a set $\mathcal{V}$ of *values*, which includes a special undefined value $\perp$; a total order $<_{\mathcal{V}}$ is defined on $\mathcal{V}$. We assume a system consisting of a collection P of nodes, $|P| = n$ connected via a communication network. The shared memory abstraction is implemented by a *memory consistency system* (MCS),

*These definitions could be expressed in terms of the general *timed automaton* model described in [9, 39, 47].

consisting of a collection of MCS processes, one at each node, that use local memory, execute some local protocol and communicate through sending *messages,* drawn from some message alphabet $\mathcal{M}$, 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. A correctness condition is specified at the interface between the application programs (written by the user) and the MCS processes (supplied by the system).

The following *external events* may occur at the MCS process p_i:

- *Call events:* They represent initiation of operations by the application program. They are $Read_i(X)$ and $Write_i(X, v)$, for all objects $X \in \mathcal{X}$ and values $v \in \mathcal{V}$.
- *Response events:* They represent response by p_i to operations initiated by the application program. They are $Return_i(X, v)$ and $Ack_i(X)$, for all objects $X \in \mathcal{X}$ and values $v \in \mathcal{V}$.
- *Message-delivery events:* They represent delivery of a message from any other MCS process to p_i. They are $del_i(j, m)$ for all messages m and MCS processes p_j.
- *Message-send events:* They represent sending of a message by p_i to any other MCS process. They are $send_i(j, m)$ for all messages m and MCS processes p_j.

For each i, 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. This is modeled by assuming that each state of p_i has a special time-varying componenet Cl_i, the *local clock* component, which is a function of the form: $Cl_i(t) = t + c_i$, where t, the *real time*, is a non-negative real number, and c_i, the local clock parameter of p_i is a *fixed* real number. The real-time clocks at various nodes might be initially non-synchronized; that is, it may be that $c_i \neq c_j$ for any i and j. 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 expire at a fixed, later time. They are $SetTimer_i(t)$ for all real numbers t.

- *Timer-expire events:* They represent expiration of a timer at p_i.

The call, message-delivery 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 an automaton with a (possibly infinite) set of states, including an initial state, and a transition function. Each interrupt event causes an application of the transition function; that is, computations of the system are interrupt driven. The transition function is a function from pairs of states and interrupt events to pairs of states and react events. That is, the transition function takes as input the current state and an interrupt event, and produces a new state, a set of response events for the corresponding application program, a set of messages to be sent to other MCS processes, and a set of timer-set events. A *computation step* of p_i is a tuple (q, e, q', R, S, T), where q and q' are states, e is an interrupt event, R is a set of response events, S is a set of message-send events, T is a set of timer-set events, so that s', R, S and T result from the application of p_i's transition function on q and e.

A *history* for an MCS process p_i is a mapping h_i from $\Re$, *real-time domain,* to finite sequences of computation steps by p_i such that:

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

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

3. The old state of each subsequent computation step is the new state of the previous computation step in the history sequence.

For a given MCS, an *execution* σ is a set of histories, one for each MCS process, such that the following conditions hold:

1. There is a one-to-one correspondence between the messages sent by p_i to p_j and the messages delivered at p_j that were sent by p_i, for any MCS processes p_i and p_j. (This

condition is imposed by *operational* characteristics of the network: the network reliably delivers all messages sent.)

2. Use the previous correspondence to define the *message-delivery time*, or *delay*, of any message for a set of histories to be the real time of delivery minus the real time of sent. Then, for every i and j, every message in σ from p_i to p_j incurs a delay in the range $[d-u, d]$, for some real numbers u and d, where $0 < d < \infty$ and $0 \leq u < d$; d is the *upper bound on message delay* and u is the message delay uncertainty. (This condition is imposed by *architectural* charactersistics of the network.)

3. There is a one-to-one correspondence between timer-set and timer-expire events at each MCS process p_i. Furthermore, each timer-expire event occurs at real time t later than the corresponding timer-set event, where t is the real number specified by the timer-set event. (This condition is imposed by *operational* characteristics of the physical clocks: clocks support reliable "timeouts".)

4. For every i, at most one call at p_i is pending at a time. (This condition is imposed by restrictions on the application programs: no pipelining is allowed at the interface between an application program and the corresponding MCS process.)

Each pair of a call event and a subsequent matching response event 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 for *op*; *op terminates* when the MCS process issues the appropriate response for *op*. Formally, each object X has a *serial specification* (cf. [28]) which describes its behavior in the absence of concurrency and failures. Formally, it defines:

- A set of *operations* on X, $Op(X)$, 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 *admissible operation sequences* for X, which consists of certain sequences of operations on X.

For each object X, $Op(X)$ contains a *read* operation on X, $[Read_i(X), Return_i(X,v)]$ and a *write* operation on X, $[Write_i(X,v), Ack_i(X)]$, for each index i and value v; v is the value associated with either of them. The set of admissible operation sequences for X contains all sequences of operations on X for which, for any read operation, rop, in the sequence, the latest preceding write operation, wop, satisfies: $val(wop) = val(rop)$. That is, all such sequences obey the usual read/write semantics: every read operation on X returns the value of the latest preceding write operation on X.

An operation sequence τ for a collection of processes and objects is *legal* if, for every object $X \in \mathcal{X}$, the restriction of τ to operations on X is in the set of admissible operation sequences for X, defined by its serial specification.

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

Given an execution σ, let $ops(\sigma)$ be the sequence of call and response events appearing in σ in real-time order, breaking ties by ordering all response events before any call event, and then using process identification numbers (id's). An execution σ specifies a partial order, $\stackrel{\sigma}{\longrightarrow}$, on the operations which appear in σ: $op_1 \stackrel{\sigma}{\longrightarrow} op_2$ if the response for operation op_1 precedes the call for operation 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 $op_1 \stackrel{\tau}{\longrightarrow} op_2$.

Let τ be an operation sequence. Denote by $\tau|i$ the restriction of τ to operations at the MCS process p_i; similarly, denote by $\tau|X$ the restriction of τ to operations on the object X.

Roughly speaking, our definitions for sequential consistency and linearizability will involve, for each execution σ, the existence of a serialization τ of σ which possesses certain properties. Formally, we define:

Definition 2.1.1 *An execution σ is sequentially consistent (cf. [32]) if there exists a legal*

serialization τ *of* σ*, such that for each MCS process* p_j*,* $\sigma|j = \tau|j$*.*

Definition 2.1.2 *An execution* σ *is linearizable (cf. [28]) if there exists a legal linearization* τ *of* σ*, such that for each MCS process* p_j*,* $\sigma|j = \tau|j$*.*

Intuitively, σ is sequentially consistent if the sequence of operations in σ can be permuted to yield a valid operation sequence τ that 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 execution of it is sequentially consistent; similarly, an MCS is a *linearizable implementation* of $\mathcal{X}$ if every execution of it is linearizable.

In general, the efficiency of an implementation 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 and a read/write object X implemented by it, the time taken by an operation *op* on X in an execution σ 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 $|R(X)|$ the maximum time taken by a read operation on X and by $|W(X)|$ the maximum time taken by a write operation on X, where the maximum is taken over all executions. We denote by $|R|$ the maximum of $|R(X)|$, and by $|W|$ the maximum of $|W(X)|$, over all read/write objects X implemented by the MCS.

Fix σ to be any execution and let $op = [call(op), response(op)]$ be any operation in σ. We denote by $t_c(op)$ and $t_r(op)$ the (real) times at which $call(op)$ and $response(op)$ occur in σ.

2.2 Discrete-Time Models

This Section is organized as follows. In Subsection 2.2.1, we present formal definitions for the discrete-time system model, and describe the time measure we will consider. In Subsec-

[†]Linearizability may be viewed as a special case of *strict serializability* (cf. [20, 49]), a basic correctness condition for concurrent computations on databases, where transactions are restricted to aggregate a single operation on a single object.

tions 2.2.2 and 2.2.3, we introduce and define the *s-session problem* and the *tick synchronization problem*, respectively.

2.2.1 The System Model

A *system* consists of a set P of n processes $p_1, \ldots, p_n$. Each process p_i is modeled as a (possibly infinite) state machine with *state set* Q_i. The state set Q_i contains a distinguished *initial state* $q_{0,i}$. The state set Q_i also includes a subset I_i of *idle* states; we assume $q_{0,i} \notin I_i$. A *configuration* is a vector $C = \langle q_1, \ldots, q_n \rangle$, where q_i is the local *state* of p_i; denote $state_i(C) = q_i$. The *initial configuration* is the vector $\langle q_{0,1}, \ldots, q_{0,n} \rangle$. Interprocess communication can be achieved through either a *network*, supporting point-to-point message passing, or *shared memory*, supporting access of atomic shared variables, but not through both. (Each kind of interprocess communication will shortly be described separately.) In either case, we consider an interleaving model of concurrency, in the style of Lynch and Tuttle (cf. [42]), where computations of the system are modeled as sequences of *atomic events*, or simply *events*.

Communication through Network

Processes are located at the nodes of a graph $G = (V, E)$, where $V = [n]$. For simplicity, we identify processes with the nodes they are located at and we refer to nodes and processes interchangeably. Processes communicate by sending *messages*, taken from some alphabet $\mathcal{M}$, to each other. A *send action* $send_i(j, m)$ represents the sending of message m to a neighboring process p_j. Let $\mathcal{S}_i$ denote the set of all send actions $send_i(j, m)$ for all $m \in \mathcal{M}$ and all $j \in [n]$, such that $(i, j) \in E$; that is, $\mathcal{S}_i$ includes all the send actions possible for p_i. Each event is either a *computation event*, representing a computation step of a single process, or a *delivery event*, representing the delivery of a message to a process. Each computation event is specified by $comp(i, S)$ for some $i \in [n]$. In the computation step associated with event $comp(i, S)$, the process p_i, based on its local state, changes its local state and performs some set S of send actions, where S is a finite subset of $\mathcal{S}_i$. Each delivery event has the form $del_i(j, m)$ for some $m \in \mathcal{M}$ and $j \in [n]$ such that $(i, j) \in E$. In a delivery step associated

with the event $del(i, m)$, the message m is added by the neighboring process p_j to $buffer_i$, p_i's message buffer[‡].

Each process p_i follows a deterministic local algorithm $\mathcal{A}_i$ that determines p_i's local computation, i.e., the messages to be sent and the state transition to be performed. More specifically, for each $q \in Q_i$, $\mathcal{A}_i(q) = (q', S)$ where q' is a state and S is a set of send actions. We assume that once a process enters an idle state, it will remain in an idle state, i.e., if q is an idle state, then q' is an idle state. An *algorithm* (or a *protocol*) is a sequence $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of local algorithms.

Communication through Shared Memory

Processes communicate by *b-atomic registers* (also called *shared variables*). Fix some integer $b > 0$ called the *fan-in*. Each shared variable may attain values from a *domain*, a set $\mathcal{V}$ of *values*, which includes a special "undefined" value, $\perp$. Each process p_i has a single *read-modify-write* atomic operation available to it that may read a *shared variable* $\mathcal{R}$, return its value v, and modify $\mathcal{R}$. Associated with each shared variable $\mathcal{R}$, is a set $Access(\mathcal{R})$ that includes the processes which may perform atomic operations on $\mathcal{R}$; we assume that, for each $\mathcal{R}$, $|Access(\mathcal{R})| \leq b$.

A configuration is extended to consist of the states of the processes and the values of the shared variables. Formally, an *extended configuration* $\tilde{C}$ is a vector $\langle q_1, \ldots, q_n, v_1, v_2, \ldots \rangle$, where q_i is the local state of p_i and v_k is the value of the shared variable $\mathcal{R}_k$; denote $state_i(\tilde{C}) = q_i$ and $value_k(\tilde{C}) = v_k$. The *initial configuration* is the configuration in which every local state is an initial state and all shared variables are set to $\perp$.

Each event is a *computation event* representing a computation step of a single process; it is specified by $comp(i, \mathcal{R})$ for some $i \in [n]$. In this computation step, the process, p_i, based on its local state performs an operation on a shared variable $\mathcal{R}$, performs some local computation, and changes to its next state.

[‡]The system model can be extended to allow arbitrary state change upon message delivery without changing the results; for clarity of presentation, we chose not to do so.

Each process p_i follows a deterministic local algorithm $\mathcal{A}_i$ that determines p_i's local computation, i.e., the register to be accessed and the state transition to be performed. More specifically, $\mathcal{A}_i$ determines:

- A shared variable $\mathcal{R}$ as a function of p_i's local state.
- Whether p_i is to modify $\mathcal{R}$ and, if so, the value v' to be written, and p_i's next state as a function of p_i's local state and the value v read from $\mathcal{R}$.

We assume that once a process enters an idle state, it will remain in an idle state. An *algorithm* (or a *protocol*) is a sequence $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$ of local algorithms.

Timing Assumptions

An *execution* is an infinite sequence of alternating configurations and events

$$\gamma = C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots ,$$

satisfying the following conditions:

1. C_0 is the initial configuration;
2. If $\pi_j = del(i, m)$, then $state_i(C_j)$ is obtained by adding m to $buffer_i$;
3. If $\pi_j = comp(i, S)$, then $state_i(C_j)$ and S are obtained by applying $\mathcal{A}_i$ to $state_i(C_{j-1})$;
4. If $\pi_j = comp(i, \mathcal{R}_k)$, then $\mathcal{R}_k$ is obtained by applying $\mathcal{A}_i$ to $state_i(\tilde{C}_{j-1})$, and $state_i(\tilde{C}_j)$ and $value_k(\tilde{C}_j)$ are obtained by applying $\mathcal{A}_i$ to $state_i(\tilde{C}_{j-1}$ and $value_k(\tilde{C}_{j-1})$;
5. If π_j involves process p_i and shared variable $\mathcal{R}_k$, then $state_l(\tilde{C}_{j-1}) = state_l(\tilde{C}_j)$ for every $l \neq i$ and $value_l(\tilde{C}_{j-1}) = value_l(\tilde{C}_j)$ for every $l \neq i$;
6. For each $m \in \mathcal{M}$ and each process p_i, let $S(i, m)$ be the set of j such that π_j is a send event $send_i(j, m)$ and let $D(i, m)$ be the set of j such that π_j is a delivery event $del_i(j, m)$. Then there exists a one-to-one onto mapping $\sigma_{i,m}$ from $S(i, m)$ to $D(i, m)$ such that $\sigma_{i,m}(j) > j$ for all $j \in S(i, m)$;

That is, in an execution the changes in processes' states and shared variables' values are according to the transition function, only a process which takes a step or to which a message is delivered changes its state, only a shared variable on which an operation is performed changes its value, and each sending of a message is matched to a later message delivery and each message delivery to an earlier send. We adopt the convention that finite prefixes of an execution end with a configuration, and denote the last configuration in a finite execution prefix γ by $last(\gamma)$. We say that $\pi_j = comp(i, S)$ or $\pi_j = comp(i, \mathcal{R})$ is a *non-idle step* of the execution if $state_i(C_{j-1}) \notin I_i$, i.e., it is taken from a non-idle state.

A *timed event* is a pair (t, π), where t, the "time", is a nonnegative real number, and π is an event. A *timed sequence* is an infinite sequence of alternating configurations and timed events

$$\alpha = C_0, (t_1, \pi_1), C_1, \ldots, (t_j, \pi_j), C_j, \ldots ,$$

where the times are nondecreasing and unbounded.

Timed executions in this model are defined as follows. Fix real numbers c and d, where $0 \leq c \leq 1$ and $0 \leq d < \infty$. Letting α be a timed sequence as above, we say that α is a *timed execution* of $\mathcal{A}$ provided that the following all hold:

1. $C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots$ is an execution of $\mathcal{A}$;
2. (Synchronous start) There are computation events for all processes with time 0;
3. (Upper bound on step time) If the jth timed event is $(t_j, comp(i_j, S))$ (resp., $(t_j, comp(i_j, R))$), then there exists a $k > j$ with $t_k \leq t_j + 1$ such that the kth timed event is $(t_k, comp(i_j, S'))$ (resp., $(t_k, comp(i_j, R'))$);
4. (Lower bound on step time) If the jth timed event is $(t_j, comp(i_j, S))$ (resp., $(t_j, comp(i_j, \mathcal{R}))$), then there does not exist a $k > j$ with $t_k < t_j + c$ such that the kth timed event is $(t_k, comp(i_j, S'))$ (resp., $(t_k, comp(i_j, \mathcal{R}'))$);
5. (Upper bound on message delivery time) If message m is sent to p_i at the jth timed event, then there exists $k > j$ such that the kth timed event is the matching delivery

$(t_k, del_i(j,m))$ (i.e., $\sigma_{i,m}(j) = k$) and $t_k \leq t_j + d$;

We say that α is an *execution fragment* of $\mathcal{A}$ if there is an execution α' of $\mathcal{A}$ of the form $\alpha' = \beta\alpha\beta'$. This definition is extended to apply to timed executions in the obvious way. For a finite execution fragment $\alpha = C_0, (t_1, \pi_1), C_1, \ldots, (t_k, \pi_k), C_k$, we define $t_{start}(\alpha) = t_1$ and $t_{end}(\alpha) = t_k$.

The *asynchronous* model is defined by taking $c = 0$, while the *semi-synchronous* model is defined by taking $0 < c \leq 1$; the *synchronous model* is a special case of the latter. Note that the asynchronous model, as defined above, allows two computation steps of the same process to occur at the same time (Condition 4 is vacuous when $c = 0$). (We remark that our proofs, as well as the proof in [4], use this property.) If we want to define the more common asynchronous model, where a process can have at most one computation step at each time, we have to replace Condition 4 above with:

> (Lower bound on step time) If the jth timed event is $(t_j, comp(i_j, S))$ (resp., $(t_j, comp(i_j, \mathcal{R}))$), then there does not exist a $k > j$ with $t_k = t_j$ such that the kth timed event is $(t_k, comp(i_j, S'))$ (resp., $(t_k, comp(i_j, \mathcal{R}'))$);

In both models, we say that a process p_i *enters an idle state by time* t' (in a timed execution α) if there exists a timed event (t_{j-1}, π_{j-1}) in α such that $t_{j-1} \leq t'$, either $\pi_{j-1} = comp(i, S)$ or $\pi_{j-1} = comp(i, R)$, and $state_i(C_j) \in I_i$.

We say that a process p_i *receives the message* m *by time* t' (in a timed execution α) if, by time t', p_i has a computation event that is preceded in α by a delivery event $del(i, m)$. For the rest, let D denote $d + 1$. Note that if m is sent to p_j at time t, then p_j receives m by time $t + D$. To simplify the expression of our time bounds in terms of the parameters d and c, we sometimes approximate the bounds in the case that $1 \ll d$. For example, in this case we have $D \approx d$.

Notation

Consider an undirected graph $G = (V, E)$. For any $i, j \in V$, let $dist(i, j)$ be the *distance* of i and j in G, i.e., the number of edges in the shortest path in G from i to j. The *diameter*

of G, $diam(G)$, is the maximum distance between any two nodes in V, i.e., $diam(G) = \max_{i,j \in V} dist(i,j)$.

A node $i \in V$ is a *peripheral* node of G if $\max_{j \in V} dist(i,j) = diam(G)$; informally, a peripheral node "realizes" the diameter of G. A node $j \in V$ is *antipodal* to a node $i \in V$ if $dist(i,j) = \max_{k \in V} dist(i,k)$; informally, j is a "farthest neighbor" of i in G. (Note that if j is antipodal to a peripheral node, then j is peripheral.)

2.2.2 The Session Problem

An execution fragment $C_1, \pi_1, C_2 \ldots, \pi_m, C_m$ is a *session* if for each i, $i \in [n]$, there exists at least one event $\pi_j = comp(i)$, for some $j \in [m]$, which is a non-idle step of the underlying execution. Intuitively, a session is an execution fragment in which each process takes at least one non-idle step. An execution α *contains* s *sessions* if it can be partitioned into at least s disjoint execution fragments such that each of them is a session. These definitions are extended to apply to timed executions in the obvious way.

An algorithm *solves the* s*-session problem within time* t if each of its timed executions α satisfies the following: α contains s sessions and all processes enter an idle state no later than time t in α.

2.2.3 The Tick Synchronization Problem

We first provide some intuition in support of our definition of the precision achievable by a tick synchronization algorithm.

At each receipt of a tick from its physical discrete clock, each process, p_i, increases the value of a special *real-time register*, L_i, by one. L_i represents p_i's "local time". Thus, L_i can be modified by p_i during an execution according to the rate at which p_i receives ticks from its physical discrete clock. For a particular execution, we define for each process p_i a function of (real) time t, $L_i(t)$, which gives p_i's local time at (real) time t. Notice that $L_i(t)$ is a piece-wise continuous function. We assume that for each i, $1 \leq i \leq n$, $L_i(0) = 0$. We also assume that a process p_i may modify L_i during the execution of a tick synchronization

algorithm in some way other than just incrementing it by one at the rate at which it receives its ticks. We assume that a process can start executing a synchronization algorithm either spontaneously or upon receipt of a message from a process that has already done so. Let t_i be the time at which p_i completes executing its synchronization algorithm. We say that p_i is in a *synchronized* state at any time $t \geq t_i$. Denote by $L_i(t_i+)$ the local, real-time estimate that p_i is making at t_i as a result of a run of a tick synchronization algorithm; that is, at time t_i, p_i potentially updates $L_i(t_i)$ to $L_i(t_i+)$, based on knowledge gathered during the execution of the algorithm and enters a synchronized state.

Clearly, it makes sense to compare $L_i(t)$ and $L_j(t)$ for $t \geq t_i, t_j$, when both p_i and p_j are guaranteed to be in a synchronized state. Furthermore, for a particular process p_i, it is most appropriate to compare $L_i(t)$ and $L_j(t)$, where p_j is any process that has also entered a synchronization state by t_i, at time *exactly* t_i, since further "asymmetry" in the rates at which p_i and p_j receive ticks can occur after t_i to separate L_i and L_j even more.

We formalize the above intuitive ideas as follows: we say that a tick synchronization algorithm, $\mathcal{A}$, *synchronizes the system within precision* Δ if for every execution of $\mathcal{A}$ and for every process p_i, $|L_p(t_i+) - L_p(t_i)| \leq \Delta$, for any process p_j such that $t_j \leq t_i$.

We will consider "symmetric" tick synchronization algorithms for which each process executes the same local protocol and treats uniformly all other processes.

Chapter 3

Timing-Based, Linearizable Read/Write Objects

In this Chapter, we show that the timing assumptions made for the continuous-time model can be exploited in order to achieve efficient, linearizable implementations of shared memory on distributed memory multiprocessor machines.

In all of our implementations, we will use, for an arbitrary execution σ of the implementation and a real-valued, non-negative function $\mathcal{T}$ defined on operations in σ, a serialization, τ, of σ, defined as follows:

Definition 3.0.1 (Serialization of σ with respect to $\mathcal{T}$) *For any operations op_1 and op_2 in σ:*

(1) *If $\mathcal{T}(op_1) < \mathcal{T}(op_2)$, then $op_1 \xrightarrow{\tau} op_2$; that is, operations are ordered in τ with respect to $\mathcal{T}$.*

(2) *If $\mathcal{T}(op_1) = \mathcal{T}(op_2)$, then:*

   (a) *if op_1 and op_2 are write and read operations, respectively, then $op_1 \xrightarrow{\tau} op_2$; that is, write operations precede in τ read ones with the same $\mathcal{T}$.*

   (b) *if op_1 and op_2 are both either read or write operations, then: if $val(op_1) <_{\mathcal{V}} val(op_2)$,*

then $op_2 \xrightarrow{\tau} op_1$, *while if* $val(op_1) = val(op_2)$, *then* op_1 *and* op_2 *are ordered arbitrarily in* τ*; that is, operations of the same type and with the same* $\mathcal{T}$ *are ordered in* τ *with respect to* $<_{\mathcal{V}}$.

We will call τ the *serialization of* σ *with respect to* $\mathcal{T}$.

This Chapter is organized as follows. Section 3.1 includes our results for the *perfect clocks* model, where processes have perfectly synchronized (*perfect*) clocks and all messages incur a constant and known delay d. We next turn to the more realistic *imperfect clocks* model, where clocks are not initially synchronized and message delays can vary. In Section 3.2, linearizable implementations of read/write objects are presented, while in Section 3.3, we present lower bounds to support optimality of our implementations.

3.1 Perfect Clocks

We present a family of linearizable implementations of read/write objects, $\mathcal{A}^{per}$, parameterized by some constant μ (known to the processes), $0 \leq \mu \leq 1$, which achieve worst-case response times of μd and $(1-\mu)d$ for read and write operations, respectively.

We start with an informal description of $\mathcal{A}^{per}$. Each process p_i keeps a local copy, X_i, of each object X. Upon a $Read_i(X)$ event, p_i waits for time μd and then issues $Return_i(X, v)$, where v is the value currently held by X_i. Upon a $Write_i(X, v)$ event, p_i sends update messages, $update(X, v)$, to all other processes; after time $(1-\mu)d$ elapses, it issues $Ack_i(X)$, but it waits for an additional time μd to set X_i to v, simultaneously with every other process receiving $update(X, v)$. (Note that this additional "busy-waiting" time μd does not contribute to $|W|$.) Upon receipt of an update message involving X, p_i immediately sets X_i to the value written. (If p_i receives several update messages at the same time, it updates X_i to the minimal with respect to $<_{\mathcal{V}}$ of the delivered values.)

The code for process p_i appears in more detail in Figure 3-1. On a call event, $Read_i(X)$ or $Write_i(X, v)$, or on receipt of an update message $update(X, v)$, p_i executes procedure $\mathsf{Read}_i(X)$, $\mathsf{Write}_i(X, v)$, or $\mathsf{Update}_i(X, v)$, respectively. By convention, a $SetTimer_i(t)$ event

postpones continuing the current execution of a procedure for time t later (till the corresponding timer expires). In procedure $\mathsf{Update}_i(X, v)$, the role of the variable $t_i^{(X)}$ is to store the time of the latest update of X_i, so that if p_i sequentially receives several update messages in message-delivery events occurring at the same real time, p_i detects this and updates X_i to the minimal of the simultaneously delivered values. Also, "interleaving" of procedures is possible: for example, p_i may execute $\mathsf{Update}_i(X, v)$ on receipt of $update(X, v)$, while waiting for expiration of the timer in executing $\mathsf{Read}_i(X)$; or, p_i may even initiate an execution of $\mathsf{Write}_i(X, v)$ while still waiting for expiration of the timer in a previous execution of it. Thus, the computation of p_i is the "parallel (self)-composition" of these procedures.

Formally, we show:

Algorithm $\mathcal{A}^{per}$: code for process p_i

Procedure $\mathsf{Read}_i(X)$; (/* executed upon a $Read_i(X)$ event */)
  $SetTimer_i(\mu d)$;
  $Return_i(X, X_i)$;
end procedure $\mathsf{Read}_i(X)$;

Procedure $\mathsf{Write}_i(X, v)$; (/* executed upon a $Write_i(X, v)$ event */)
  $broadcast_i(update(X, v))$;
  $SetTimer_i((1-\mu)d)$;
  $Ack_i(X)$;
  $SetTimer_i(\mu d)$;
  $X_i := v$;
end procedure $\mathsf{Write}_i(X, v)$;

Procedure $\mathsf{Update}_i(X, v)$; (/* executed upon receipt of $update(X, v)$ */)
  if $t_i^{(X)} = Cl_i$ **then** $X_i := \min_{<_\mathcal{V}}\{X_i, v\}$ **else** $X_i := v$;
end procedure $\mathsf{Update}_i(X, v)$;

Figure 3-1: The Algorithm $\mathcal{A}^{per}$: code for process p_i

---

Theorem 3.1 *For each μ, $0 \leq \mu \leq 1$, $\mathcal{A}^{per}$ is a linearizable implementation of read/write objects, which achieves: $|R| = \mu d$ and $|W| = (1-\mu)d$.*

Proof: Fix any execution σ of $\mathcal{A}^{per}$.

Clearly, in σ, the response time for every read operation is μd and the response time for every write operation is $(1-\mu)d$, implying $|R| = \mu d$ and $|W| = (1-\mu)d$.

We now show that $\mathcal{A}^{per}$ is a linearizable implementation of read/write objects. We construct a legal linearization τ of σ such that, for each MCS process p_j, $\tau|j = \sigma|j$; each read or write operation in σ is "serialized" to occur at the time of its call or response in σ, respectively, breaking ties by ordering all write operations before read ones that occur simultaneously in σ and then using $<_{\mathcal{V}}$. Formally, we assign a time $\mathcal{T}(op)$ to each operation $op = [call(op), response(op)]$ in σ: $\mathcal{T}(op) = t_c(op)$ if op is a read operation; else, $\mathcal{T}(op) = t_r(op)$. Let τ be the serialization of σ according to $\mathcal{T}$.

We first prove:

Lemma 3.1 *τ is a linearization of σ.*

Proof: Let op_1 and op_2 be operations in σ such that $op_1 \xrightarrow{\sigma} op_2$. By definition of $\xrightarrow{\sigma}$, $t_r(op_1) \leq t_c(op_2)$. By definition of $\mathcal{T}$, $\mathcal{T}(op_1) \leq t_r(op_1)$ and $\mathcal{T}(op_2) \geq t_c(op_2)$; it follows that $\mathcal{T}(op_1) \leq \mathcal{T}(op_2)$. If $\mathcal{T}(op_1) < \mathcal{T}(op_2)$, then $op_1 \xrightarrow{\tau} op_2$, by (1) in Definition 3.0.1. So assume $\mathcal{T}(op_1) = \mathcal{T}(op_2)$. By definition of $ops(\sigma)$, it follows that op_1 and op_2 are write and read operations, respectively. Hence, $op_1 \xrightarrow{\tau} op_2$, by (2)(a) in Definition 3.0.1. ■

We continue by showing:

Lemma 3.2 *For each MCS process p_j, $\tau|j = \sigma|j$.*

Proof: Consider operations op_1 and op_2 such that $op_1 \xrightarrow{\sigma|j} op_2$. It suffices to show that $op_1 \xrightarrow{\tau|j} op_2$. Clearly, $op_1 \xrightarrow{\sigma} op_2$. By Lemma 3.1, $op_1 \xrightarrow{\tau} op_2$, implying $op_1 \xrightarrow{\tau|j} op_2$, as needed. ■

We continue with a simple claim which will be used in showing that τ is a legal operation sequence. We define a relation $\overset{\sigma}{\hookrightarrow}$ between write and read operations in γ as follows: for any write and read operations wop and rop, respectively, on object X in σ, $wop \overset{\sigma}{\hookrightarrow} rop$ if $val(wop) = val(rop)$ and the latest update (in σ) on the local copy of X by the reading

process (before it returns on rop), as a result of receiving $update(X, val(wop))$ from the writing process, is to $val(wop)$. Roughly speaking, $\stackrel{\sigma}{\hookrightarrow}$ captures causality and relates each read operation in γ to the write operation writing the returned value. We have:

Claim 3.1 *Let $wop_1 = [Write_i(X, v_1), Ack_i(X)]$ and $rop_1 = [Read_k(X), Return_k(X, v_1)]$ be write and read operations, respectively, for some value $v_1 \in \mathcal{V}$, indices $i, k \in [n]$ and a read/write object X, such that $wop_1 \stackrel{\sigma}{\hookrightarrow} rop_1$. Then, $wop_1 \stackrel{\tau}{\longrightarrow} rop_1$.*

Proof: Since all message delays are exactly d, $t_r(rop_1) \geq t_c(wop_1) + d$. Since $\mathcal{T}(rop_1) = t_c(rop_1) = t_r(rop_1) - \mu d$, and $\mathcal{T}(wop_1) = t_r(wop_1) = t_c(wop_1) + (1 - \mu)d$, it follows that $\mathcal{T}(rop_1) \geq \mathcal{T}(wop_1)$. If $\mathcal{T}(rop_1) > \mathcal{T}(wop_1)$, then $wop_1 \stackrel{\tau}{\longrightarrow} rop_1$, by (1) in Definition 3.0.1; if $\mathcal{T}(rop_1) = \mathcal{T}(wop_1)$, then $wop_1 \stackrel{\tau}{\longrightarrow} rop_1$ by (2)(a) in Definition 3.0.1. ■

Notice that Claim 3.1 implies that if a read operation in τ returns a value "out of order", i.e., other than that of the immediately preceding (in τ) write operation on the same object, then this read operation is related (through $\stackrel{\sigma}{\rightsquigarrow}$) to a write operation that precedes it in τ. Thus, Claim 3.1 restricts the way in which τ may violate legality.

We finally show that τ is a legal operation sequence. An informal outline of our proof follows. We assume that some read operation returns a value other than that of the immediately preceding write operation on the same object and derive a contradiction by showing that the superseded written value is "known" to the reading process before the read operation returns. We have:

Lemma 3.3 *τ is a legal operation sequence.*

Proof: Assume, by way of contradiction, that τ is not legal. From Claim 3.1 and the assumption that τ is not legal, there must exist operations $wop_1 = [Write_i(X, v_1), Ack_i(X)]$, $wop_2 = [Write_j(X, v_2), Ack_j(X)]$ and $rop_1 = [Read_k(X), Return_k(X, v_1)]$, for some indices i, j and $k \in [n]$, a read/write object X and values $v_1, v_2 \in \mathcal{V}$, such that wop_1, wop_2, rop_1 is a subsequence of τ and wop_2 is the latest write operation on X in τ such that $wop_2 \stackrel{\tau}{\longrightarrow} rop_1$.

By construction of τ, $\mathcal{T}(wop_1) \leq \mathcal{T}(wop_2) \leq \mathcal{T}(rop_1)$, i.e., $t_r(wop_1) \leq t_r(wop_2) \leq t_c(rop_1)$. In fact, we prove:

Claim 3.2 $\mathcal{T}(wop_1) < \mathcal{T}(wop_2)$

Proof: Assume, by way of contradiction, that $\mathcal{T}(wop_1) = \mathcal{T}(wop_2)$. By construction of τ, $v_2 <_{\mathcal{V}} v_1$. By definition of $\mathcal{T}$, $t_r(wop_1) = t_r(wop_2)$, implying $t_c(wop_1) = t_c(wop_2)$. Since message delays are fixed, p_k receives update messages simultaneously from p_i and p_j; it must set X_k to v_1 (later returned). Hence, by the algorithm, $v_1 <_{\mathcal{V}} v_2$. A contradiction. ■

Note that Claim 3.2 implies that $t_c(wop_1) < t_c(wop_2)$. Since all message delays are equal to d and, by the algorithm, a writing process waits for time d to update its local copy to the written value, it follows that each process sets its local copy of X to v_1 strictly before it sets it to v_2. Also, note that

$$t_r(rop_1) = t_c(rop_1) + \mu d \geq t_r(wop_2) + \mu d = t_c(wop_2) + (1-\mu)d + \mu d = t_c(wop_2) + d \ ,$$

i.e., p_k updates X_k to v_2 no later than time $t_r(rop_1)$. Hence, it follows that rop_1 returns v_2. A contradiction. ■

Thus, τ is a legal linearization of σ such that, for each MCS process p_j, $\tau|j = \sigma|j$. Since σ was chosen arbitrarily, this implies that $\mathcal{A}^{per}$ is a linearizable implementation of read/write objects. ■

We remark that it is possible to use a different parameter, μ_X, for each different object X, according, perhaps, to the pattern of read and write operations on X. Under this modification, $\mathcal{A}^{per}$ is no more symmetric with respect to the objects. But, the main price paid is that the corresponding worst-case response times are $|R| = \max_{X \in \mathcal{X}} \mu_X d$, and $|W| = (1 - \min_{X \in \mathcal{X}} \mu_X)d$, so that $|R| + |W| = (1 + \max_{X \in \mathcal{X}} \mu_X - \min_{X \in \mathcal{X}} \mu_X)d$, which is possibly larger than d, established in [35] as a general lower bound on $|R| + |W|$ for any linearizable implementation of read/write objects, and met for the perfect clocks model by taking $\min_{X \in \mathcal{X}} \mu_X = \max_{X \in \mathcal{X}} \mu_X = \mu$, as in Theorem 3.1.

3.2 Imperfect Clocks: Upper Bounds

This Section is organized as follows. Subsection 3.2.2 presents $\mathcal{A}_1^{imp}$, a linearizable implementation of read/write objects. In Subsection 3.2.3, we show how to modify $\mathcal{A}_1^{imp}$ to obtain a family of linearizable implementations, $\mathcal{A}_2^{imp}$, supporting quantitative regulation of the response times of read and write operations. In an initialization phase, prior to each of these implementations, a simple synchronization procedure, $\mathcal{A}^{syn}$, runs in order to enable the processes acquire a certain amount of accuracy; this procedure is presented in Subsection 3.2.1.

3.2.1 A Synchronization Strategy

In executing $\mathcal{A}^{syn}$ in an initialization phase of the computation, 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 any other process, or on expiration of the timer, whichever happens first, p_i sets its local time to 0 at real time, say, t_0. That is, p_i constructs a *logical clock* out of its physical one by implicitly "changing" the local clock parameter c_i to $c_i - Cl_i(t_0)$, and at all future accesses to local time, p_i takes the local time $Cl_i(t)$ to be the difference of the physical clock time, read from its physical clock, and $Cl_i(t_0)$. (In all future discussion, we will use local clock time to refer to logical clock time "constructed" in this way.)

We say that an algorithm *synchronizes the system within accuracy* δ if the maximum difference between the local times of any two processes at any real time after all processes have completed the execution of the algorithm is at most δ. We show:

Theorem 3.2 *$\mathcal{A}^{syn}$ synchronizes the system within accuracy u.*

Proof: Since clocks do not drift, it suffices to consider the latest real time t at which a process p_i completes the execution of $\mathcal{A}^{syn}$ and show that the local clock time of any other process is $\leq u$ at time t.

Notice that every *synch* message was sent at time $\geq t - d$, since, otherwise, p_i would receive it (and halt) at time $< t$. Hence, every process p_j that completes the execution of

$\mathcal{A}^{syn}$ on receipt of *synch* does so within the time interval $[t-d+d-u, t] = [t-u, t]$; thus, the local clock time of p_j at time t is $\leq u$, as needed. So, consider a process p_j completing the execution of $\mathcal{A}^{syn}$ on expiration of its own timer; since p_j broadcasts *synch* at time $\geq t-d$, waits for time d and completes execution of $\mathcal{A}^{syn}$ at time $\leq t$ (before p_i does), it can only be that p_j completes execution of $\mathcal{A}^{syn}$ and set its local time to 0 at time t, as p_i does. ■

We remark that, by the results of Lundelius and Lynch in [36], an accuracy of $(1-\frac{1}{n})u$ is achievable in the imperfect clocks model. This accuracy is slightly better than u, but we chose to present and use the slightly weaker one since it can be achieved by a much simpler algorithm than the one in [36], which might also be of independent interest. Furthermore, the better accuracy achieved in [36] does not seem to considerably improve our subsequent results, if it improves them at all.

We conclude this Subsection with a simple observation encompassing the "common knowledge" acquired by the processes as a result of running $\mathcal{A}^{syn}$. Fix any execution σ and assume that all processes executed $\mathcal{A}^{syn}$ in a prior phase. We show:

Lemma 3.4 *Consider message-send events $send_{i_1}(j_1, m_1)$ and $send_{i_2}(j_2, m_2)$ occurring at (real) times t_1 and t_2, respectively, in σ such that $Cl_{i_2}(t_2) - Cl_{i_1}(t_1) > 2u$. Let the corresponding message-delivery events occur at (real) times t_1' and t_2', respectively, in σ. Then, $t_1' < t_2'$.*

Proof: Clearly, it must be that $t_2 > t_1$, since, otherwise,

$$Cl_{i_2}(t_2) - Cl_{i_1}(t_1) = Cl_{i_2}(t_1) - (t_1 - t_2) - Cl_{i_1}(t_1) \leq Cl_{i_2}(t_1) - Cl_{i_1}(t_1) \leq u \ ,$$

by Theorem 3.2. We have:

$$t_2' - t_1' \geq t_2 + d - u - t_1 - d = Cl_{i_2}(t_2) - Cl_{i_2}(t_1) - u \geq Cl_{i_2}(t_2) - Cl_{i_1}(t_1) - u - u > 2u - 2u = 0$$

as needed. ■

3.2.2 First Implementation

We present a linearizable implementation of read/write objects, $\mathcal{A}_1^{imp}$, which achieves worst-case response times of $< 4u + \epsilon$ and $d + 3u$ for read and write operations, respectively, where $\epsilon > 0$ is an arbitrarily small constant.

We first describe the "timings" of $\mathcal{A}_1^{imp}$, and then discuss how a process selects a value to return in a read operation.

- Upon a $Read_i(X)$ event, p_i returns at the earliest possible time, provided that time $\geq u$ elapsed since the previous update of X_i. This is detected by using $timer_i(X)$, a timer, set (to 0) on each update of X_i and reset (to $\perp$) each time p_i returns for a read operation on X.

- A "time-slicing" technique is used for handling write operations; roughly speaking, each process p_i slices each time period of $3u + \epsilon$ into an interval of length $3u$ in which actions on a call event of a write operation may not be initiated, followed by an interval of length ϵ in which they may. Upon a $Write_i(X, v)$ event and when in the appropriate time interval, p_i broadcasts an $update(X, v)$ message together with its local time at the (real) time of broadcasting; it then waits for an additional time d to set X_i to v and issue $Ack_i(X)$.

We now describe how process p_i selects a value to return in a read operation on object X: candidate values to be returned are those to which p_i previously set X_i to and whose local broadcasting time is within $2u$ of that with the currently maximum local broadcasting time. To do so, p_i maintains a set $Pend_i(X)$ of "pending" update messages that it recently received. Whenever p_i updates X_i to v, either on receipt of $update(X, v)$ or as a result of a write operation by itself on X, it adds (v, t) to $Pend_i(X)$, where t is the local time at which $update(X, v)$ was broadcasted by the writing process. To keep the size of $Pend_i(X)$ small, at each update of X_i, p_i removes from $Pend_i(X)$ all pairs (v', t') such that t' is not within $2u$ of the currently maximum time component of elements of $Pend_i(X)$. p_i returns the maximum (with respect to $<_{\mathcal{V}}$) of the value components of elements of $Pend_i(X)$. The

code for process p_i appears in more detail in Figure 3-2, following the same conventions as in Figure 3-1, where, for any real numbers r_1 and r_2, $\text{fmod}(r_1, r_2)$ denotes the (real) remainder of the integral division of r_1 by r_2. Formally, we show:

Algorithm $\mathcal{A}_1^{imp}$: code for process p_i

Procedure $\mathsf{Read}_i(X)$; (/* executed upon a $Read_i(X)$ event */)
  if $timer_i(X) \neq \perp$ **then waitfor** $timer_i(X) \geq u$;
    $timer_i(X) := \perp$;
  endif;
  $X_i := \max_{<_V}\{v : (v,t) \in Pend_i(X)\}$;
  $Return_i(X, X_i)$;
end procedure $\mathsf{Read}_i(X)$;

Procedure $\mathsf{Write}_i(X, v)$; (/* executed upon a $Write_i(X, v)$ event */)
  waitfor $\text{fmod}(Cl_i, 3u + \epsilon) > 3u$;
  $broadcast_i((update(X, v), Cl_i))$;
  $SetTimer_i(d)$;
  $Pend_i(X) := Pend_i(X) \cup \{(v, Cl_i - d)\}$;
  $t_{max} := \max\{t' : (v', t') \in Pend_i(X)\}$
  $Pend_i(X) := \{(v', t') : (v', t') \in Pend_i(X) \text{ and } t_{max} - t' \leq 2u\}$
  $Ack_i(X)$;
end procedure $\mathsf{Write}_i(X, v)$;

Procedure $\mathsf{Update}_i(X)$ (/* executed upon receipt of $update((X, v), t)$ */)
  $Pend_i(X) := Pend_i(X) \cup \{(v, t)\}$;
  $t_{max} := \max\{t' : (v', t') \in Pend_i(X)\}$;
  $Pend_i(X) := \{(v', t') : (v', t') \in Pend_i(X) \text{ and } t_{max} - t' \leq 2u\}$;
  $timer_i(X) := 0$;
end procedure $\mathsf{Update}_i(X)$;

Figure 3-2: The Algorithm $\mathcal{A}_1^{imp}$—code for process p_i

---

Theorem 3.3 *$\mathcal{A}_1^{imp}$ is a linearizable implementation of read/write objects, which achieves: $|R| < 4u + \epsilon$, $|W| = d + 3u$.*

Proof: Fix any execution σ of $\mathcal{A}_1^{imp}$.

We first present a timing analysis to show the claimed bounds on $|R|$ and $|W|$.

The bound on $|W|$ is obvious since, by the algorithm, each process p_i waits till $\mathrm{fmod}(Cl_i, 3u+\epsilon)$ holds, and returns after an additional time d elapses. We proceed to show that $|R| < 4u+\epsilon$. Consider a call event $Read_i(X)$ occurring at time t in σ. We show that p_i responds to $Read_i(X)$ by time $t' < t + 4u + \epsilon$.

Since on delivery of an update message involving X to p_i, p_i sets $timer_i(X)$ to 0 and procedure $\mathsf{Read}_i(X)$ is prevented from immediate termination, it seems as if a deadlock may occur due to successive deliveries of update messages to p_i; the "time-slicing" technique, however, rules this out. We show that there exists a family of "quiet" ("update-free") time intervals, $quiet_i(k)$, one for each integer k, with the following properties:

- p_i receives no update messages in each time interval $quiet_i(k)$;
- Each time interval $quiet_i(k)$ has length at least u;
- Any two such consecutive time intervals, $quiet_i(k)$ and $quiet_i(k+1)$, are separated by a time interval, $gap_i(k) = (\lceil quiet_i(k) \rceil, \lfloor quiet_i(k+1) \rfloor)$ of length $\leq 2u + \epsilon$.

Formally established in the next two claims, these properties will imply that any read operation will return strictly before time $4u + \epsilon$ elapses from its initiation.

Claim 3.3 *For each integer k, there exists a time interval $quiet_i(k)$ such that p_i receives no update messages in $quiet_i(k)$. Furthermore, $|quiet_i(k)| \geq u$.*

Proof: Consider any $j \in [n]$. Any update message sent by p_j to p_i while $Cl_j < k(3u+\epsilon)$ will be delivered to p_i while $Cl_j < k(3u+\epsilon)+d$; any update message sent by p_j to p_i while $Cl_j > k(3u+\epsilon)+3u$ is delivered to p_i while $Cl_j > k(3u+\epsilon)+3u+d-u = k(3u+\epsilon)+d+2u$. (Recall that, by the algorithm, p_j cannot send any update messages while $k(3u+\epsilon) \leq Cl_j \leq k(3u+\epsilon)+3u$.) Thus, no update message from p_j is delivered to p_i while $k(3u+\epsilon)+d \leq Cl_j \leq k(3u+\epsilon)+d+2u$.

For each $j \in [n]$, let $t_j(k)$ be the (real) time at which $Cl_j = k(3u+\epsilon)$. It follows that for each $j \in [n]$, no update message from p_j is delivered to p_i in the time interval $[t_j(k)+d, t_j(k)+d+2u]$. Hence, no message from any process is delivered to p_i in the time

interval $quiet_i(k)$, where:

$$quiet_i(k) = \bigcap_{j\in[n]} [t_j(k)+d, t_j(k)+d+2u] = [\max_{j\in[n]} t_j(k)+d, \min_{j\in[n]} t_j(k)+d+2u] \ .$$

We have:

$$|quiet_i(k)| = \min_{j\in[n]} t_j(k)+d+2u - \max_{j\in[n]} t_j(k) - d = \min_{j\in[n]} t_j(k) - \max_{j\in[n]} t_j(k) + 2u \ .$$

It follows, however, from Theorem 3.2 that:

$$\max_{j\in[n]} t_j(k) - \min_{j\in[n]} t_j(k) \leq u \ .$$

This implies: $|quiet_i(k)| \geq -u+2u = u$, as needed. ■

We now show an upper bound on the "gap" between consecutive "quiet" time intervals*:

Claim 3.4 *For each integer k, $|gap_i(k)| \leq 2u+\epsilon$.*

Proof: Using the notation of Claim 3.3, we have:

$$\begin{aligned}
|gap_i(k)| &= \lfloor quiet_i(k+1)\rfloor - \lceil quiet_i(k)\rceil \\
&= \max_{j\in[n]} t_j(k+1) + d - \min_{j\in[n]} t_j(k) - d - 2u \\
&= \max_{j\in[n]} t_j(k) + 3u + \epsilon - \min_{j\in[n]} t_j(k) - 2u \\
&= \max_{j\in[n]} t_j(k) - \min_{j\in[n]} t_j(k) + u + \epsilon \\
&\leq u + u + \epsilon \qquad \text{(by Theorem 3.2)} \\
&= 2u+\epsilon \ ,
\end{aligned}$$

as needed. ■

Clearly, in the worst-case, $Read_i(X)$ occurs (at time t) within some time interval $quiet_i(k)$, for some integer k, but p_i enters the time interval $gap_i(k)$ and receives some update message

*For a real interval $x=[x_1,x_2]$, $\lfloor x\rfloor = x_1$ and $\lceil x\rceil = x_2$.

involving X before it may respond to $Read_i(X)$. Such an update message must be delivered to p_i by time $< t + u$, since, otherwise, $timer_i(X)$ would have attained the value u. By Claim 3.4, p_i will enter $quiet_i(k+1)$ by time $< t+u+2u+\epsilon = t+3u+\epsilon$. Since, by Claim 3.3, $|quiet_i(k+1)| \geq u$, $timer_i(X)$ does attain the value u within $quiet_i(k+1)$, and p_i issues $Return(X, X_i)$ by time $< t + 3u + \epsilon + u = t + 4u + \epsilon$. Hence, $|R| < 4u + \epsilon$, as claimed.

We turn to show that $\mathcal{A}_1^{imp}$ is a linearizable implementation.

We construct a legal linearization τ of σ such that, for each MCS process p_j, $\tau|j = \sigma|j$. Our construction proceeds in two phases.

In the first phase, we "serialize" each read and write operation in σ to occur at the time of its response in σ, breaking ties by ordering all write operations before read ones that occur simultaneously and then using $<_{\mathcal{V}}$. Formally, let the operation sequence τ' be the serialization of σ with respect to t_r.

Clearly, if $op_1 \xrightarrow{\sigma} op_2$, then $t_r(op_1) < t_r(op_2)$ and, by construction of τ' (Definition 3.0.1, (1)), $op_1 \xrightarrow{\sigma} op_2$. Hence, τ' is a linearization of σ such that, for each MCS process p_j, $\tau'|j = \sigma|j$.

τ' might, however, be not legal, and the objective of the second phase of our construction is to "perturb" τ', through reordering operations and while still preserving the properties of τ', in order to obtain an operation sequence τ which is, in addition, legal.

We continue with a simple Claim (reminiscent of Claim 3.1) which we will use in constructing τ from τ'. To state it, we define (as in Theorem 3.1) a relation, $\overset{\sigma}{\rightsquigarrow}$, between write and read operations in σ as follows: for any write and read operations wop and rop, respectively, on object X in γ, $wop \overset{\sigma}{\rightsquigarrow} rop$ if $val(wop) = val(rop)$ and the element of the pending set of the reading process with value component $val(wop)$ (chosen to be returned) was added to the pending set on receipt of an update message from the writing process. For wop and rop such that $wop \overset{\sigma}{\rightsquigarrow} rop$, denote by $t_{br}(wop)$ and $t_{del}(wop)$ the real time at which the writing process broadcasts the update message and the real time at which the reading process receives it, respectively. We have:

Claim 3.5 *Let* $wop = [Write_i(X, v), Ack_i(X)]$ *and* $rop = [Read_k(X), Return_k(X, v)]$ *be*

write and read operations, respectively, for indices $i, k \in [n]$, *a read/write object* X *and a value* $v \in \mathcal{V}$, *such that* $wop \overset{\sigma}{\rightsquigarrow} rop$. *Then,* $wop \overset{\tau'}{\longrightarrow} rop$.

Proof: Since every message delay is $\geq d - u$, $t_{del}(wop) - t_{br}(wop) \geq d - u$. Since, by the algorithm, $t_r(rop) \geq t_{del}(wop) + u$, it follows that: $t_r(rop) \geq t_{br}(wop) + d = t_r(wop)$. Thus, by construction, $wop \overset{\tau}{\longrightarrow} rop$, as needed. ■

Notice that Claim 3.5 implies that if a read operation in τ' returns a value "out of order," i.e., other than that of the immediately preceding (in τ') write operation on the same object, then this read operation is related (through $\overset{\sigma}{\rightsquigarrow}$) to a write operation that precedes it in τ'. Thus, Claim 3.5 restricts the way in which τ' may violate legality.

In the second phase, we scan τ' and trace and fix each legality violation in it by "locally" reordering operations, while still preserving its properties. We show that the length of the maximal prefix of τ' which is a legal operation sequence strictly grows after each fix as we proceed; thus, inductively, this results in a legal operation sequence τ which is a linearization of σ such that, for each MCS process p_l, $\sigma|l = \tau|l$.

Formally, let $rop_1 = [Read_k(X), Return_k(X, v_1)]$ be the earliest read operation in τ', for some index $k \in [n]$, read/write object X, value $v_1 \in \mathcal{V}$, for which there exist write operations $wop_1 = [Write_i(X, v_1), Ack_i(X)], wop_2 = [Write_j(X, v_2), Ack_j(X)]$, for some indices $i, j \in [n]$, and value $v_2 \in \mathcal{V}$, such that wop_1, wop_2, rop_1 is a subsequence of τ', $wop_1 \overset{\sigma}{\rightsquigarrow} rop_1$, and wop_2 is the latest write operation on X in τ' such that $wop_2 \overset{\tau'}{\longrightarrow} rop_1$. Call rop_1 an *illegal* read operation. Reorder wop_2 to immediately precede wop_1 in τ'. Iterate till no illegal read operation exists. Let τ be the resulting operation sequence.

We start by showing:

Lemma 3.5 τ *is a legal operation sequence.*

Proof: We proceed inductively and show that, after each reordering, the prefix of τ' ending with an illegal read operation rop_1 is a legal operation sequence.

We start with a simple Claim establishing that every process "hears" about a write operation before its termination. Since each process receives update message for wop by time

$\leq t_{br}(wop) + d$ and, by the algorithm, $t_r(wop) = t_{br}(wop) + d$, we immediately have:

Claim 3.6 *For any write operation wop in σ, every process receives update message for wop by time $t_r(wop)$.*

Our next Claim establishes that the local broadcasting times of wop_1 and wop_2 are "close".

Claim 3.7 $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| \leq 2u$.

Proof: Assume, by way of contradiction, that: $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| > 2u$. We proceed by case analysis.

1. Take $Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2)) > 2u$. Clearly, it must be that $t_{br}(wop_1) > t_{br}(wop_2)$, since, otherwise,

$$\begin{aligned} Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2)) &= Cl_i(t_{br}wop_2) - (t_{br}(wop_2) - t_{br}(wop_1)) - Cl_j(t_{br}(wop_2)) \\ &\leq Cl_i(t_{br}(wop_2)) - Cl_j(t_{br}(wop_2)) \\ &\leq u \ , \end{aligned}$$

   by Theorem 3.2. By the algorithm, $t_r(wop_i) = t_{br}(wop_i) + d$, for $i \in \{1, 2\}$. Hence, $t_r(wop_1) > t_r(wop_2)$. By (1) in Definition 3.0.1, this implies: $wop_2 \xrightarrow{\tau'} wop_1$. A contradiction.

2. Take $Cl_j(t_{br}(wop_2)) - Cl_i(t_{br}(wop_1)) > 2u$. By Claim 3.6, an update message for v_2 is delivered to p_k by time $\leq t_r(wop_2) \leq t_r(wop_1)$, since $wop_2 \xrightarrow{\tau'} rop_1$. Let t_{max} be the maximal time component of elements in $Pend_k(X)$ at time $t_r(rop_1)$. We have:

$$t_{max} - Cl_i(t_{br}(wop_1)) \geq Cl_j(t_{br}(wop_2)) - Cl_i(t_{br}(wop_1)) > 2u \ .$$

   Hence, $(v_1, Cl_i(t_{br}(wop_1)))$ is removed from $Pend_k(X)$ at time $t_r(rop_1)$. A contradiction.

■

We next use properties of the time slicing technique to show that Claim 3.7 implies that the local broadacsting times fall within the same time slice of the writing processes. Formally, assume $3uk_1 - \epsilon < Cl_i(t_{br}(wop_1)) \leq 3uk_1$ and $3uk_2 - \epsilon < Cl_j(t_{br}(wop_2)) \leq 3uk_2$, for some positive integers k_1 and k_2. We have:

Claim 3.8 $k_1 = k_2$.

Proof: Assume, by way of contradiction, that $k_1 \neq k_2$. Without loss of generality, let $k_1 > k_2$. This implies:

$$\begin{aligned} |Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| &= Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2)) \\ &> k_1(3u+\epsilon) - (k_2(3u+\epsilon) - \epsilon) \\ &= (k_1 - k_2)(3u+\epsilon) + \epsilon \\ &\geq 3u + 2\epsilon \\ &> 2u \ , \end{aligned}$$

contradicting Claim 3.7. ■

We continue by showing a simple fact about τ'.

Claim 3.9 *Let wop and wop' be any write operations in τ' such that there is no write operation in $(wop, wop')_{\tau'}$. Then, every read operation in $(wop, wop')_{\tau'}$ returns the same value.*

Proof: Consider any read operation in $(wop, wop')_{\tau'}$. By Definition 3.0.1 used in constructing τ', $t_r(wop) \leq t_r(rop) < t_r(wop')$. By Claim 3.6, the reading process receives update messages for wop and all earlier (in τ') write operations on X by time $\leq t_r(wop)$. Moreover, it must be that the reading process receives an update message for wop' later than time $t_r(rop)$, since, otherwise, the algorithm implies that $t_r(rop) \geq t_r(wop)$. Hence, the pending set of the reading process at time $t_r(rop)$ may only contain elements corresponding to write operations completed by time $t_r(wop)$. By the algorithm, every reading process returns identically. ■

By Claim 3.9, we assume, without loss of generality, that at most one read operation may be completed between any two successsive completions of write operations in τ'.

Our key Claim follows:

Claim 3.10 *There is no read operation rop_2 on X in τ' such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$.*

Proof: Assume, by way of contradiction, that there exists a read operation rop_2 on X in τ' such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$. We proceed by case analysis:

1. Assume, first, that $t_r(rop_2) \geq t_r(rop_1)$. By assumption, there is no write operation on X in $(wop_2, rop_1)_{\tau'}$. Hence, by Claim 3.9, there must be at least one write operation on X in $(rop_1, rop_2)_{\tau'}$; let wop_3 be the one with the maximal local broadcasting time. We consider the intervals: $(wop_1, rop_1)_{\tau'}$ and $(wop_2, rop_2)_{\tau'}$ in τ'; it follows from Claims 3.7 and 3.8 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 $t_r(wop_2)$, 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 the argument for the previous case to the intervals $(wop_1, rop_2)_{\tau'}$ and $(wop_1, rop_1)_{\tau'}$. Let t^1_{max} and t^2_{max} be the maximal time components of elements of the pending sets for X of the processes performing rop_1 and rop_2, respectively, at the time of response. As already argued, by time $t_r(rop_2)$, each process modifies its pending set for X as a result of a write operation on X completed by time $t_r(rop_2)$. Thus, any modification of this set at time $> t_r(rop_2)$ is due to a write operation returning at time $> t_r(rop_2)$; thus, 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 the pending set for X of any process can only increase t^2_{max}. Hence, $t^1_{max} \geq t^2_{max}$. Clearly, $t^2_{max} - Cl_j(t_r(wop_2)) \leq 2u$ and $t^1_{max} - Cl_i(t_r(wop_1)) \leq 2u$. This implies: $t^1_{max} - t_r(wop_2) \leq 2u$. By the algorithm and the way wop_1 returns, $v_1 <_{\mathcal{V}} v_2$. Hence, by the way wop_2 returns, it must be that:

$t^2_{max} - t_r(wop_1) > 2u$. A contradiction.

■

Clearly, Claim 3.10 implies that, after the reordering, the prefix of τ' ending with rop_1 is a legal operation sequence. ■

We continue by showing:

Lemma 3.6 *τ is a linearization of σ.*

Proof: It suffices to show that the reordered operations wop_1 and wop_2 "overlap" in σ, i.e., that $t_c(wop_2) \leq t_r(wop_1)$. By Claim 3.8, the local broadcasting times of wop_1 and wop_2 fall in the same time slice. Notice that, by Theorem 3.2, this implies: $|t_{br}(wop_1) - t_{br}(wop_2)| \leq u + \epsilon$. Since, by assumption, ϵ is arbitrarily small, we may choose ϵ so that $\epsilon \leq d - u$. (Recall that, by assumption, $d > u$.) Thus, we have:

$$t_c(wop_2) \leq t_{br}(wop_2) \leq t_{br}(wop_1) + \epsilon + u = t_r(wop_1) - d + \epsilon + u \leq t_r(wop_1),$$

as needed. ■

Recall that for τ', $\tau'|l = \sigma|l$ for each MCS process p_l. In permuting τ' to obtain τ, we reordered only wop_1 and wop_2 and showed that neither $wop_1 \xrightarrow{\sigma} wop_2$ nor $wop_2 \xrightarrow{\sigma} wop_1$. Hence:

Lemma 3.7 *For each MCS process p_l, $\tau|l = \sigma|l$.*

It follows from Lemmas 3.5, 3.6 and 3.7 that $\mathcal{A}_1^{imp}$ is a linearizable implementation of read/write objects. ■

3.2.3 Second Implementation

In this Subsection, we show how to slightly modify $\mathcal{A}_1^{imp}$ and obtain a linearizable implementation of read/write objects, $\mathcal{A}_2^{imp}$, which achieves worst-case response times of $< \mu d + 4u + \epsilon$

and $(1-\mu)d+3u$ for read and write operations, respectively, where $\epsilon > 0$ is an arbitrarily small constant (as in $\mathcal{A}_1^{imp}$).

$\mathcal{A}_2^{imp}$ differs from $\mathcal{A}_1^{imp}$ only with respect to the "timings" of read and write operations:

- Upon a $Read_i(X)$ event, p_i sets a timer to expire at time μd thereafter, for some parameter μ such that $0 \leq \mu < \frac{d-u}{d}$; after the timer expires, p_i runs as in $\mathcal{A}_1^{imp}$: it returns at the earliest possible time, provided that time $\geq u$ elapsed since the previous update of X.

- Upon a $Write_i(X,v)$ event and when in the appropriate time slice, p_i broadcasts an $update(X,v)$ message and waits for an additional time $(1-\mu)d$ to set X_i to v and issue $Ack_i(X)$.

p_i selects a value to return in a read operation as in $\mathcal{A}_1^{imp}$. The code for process p_i appears in detail in Figure 3-3, using the same conventions as in Figures 3-1 and 3-2. Formally, we show:

Theorem 3.4 *For each μ, $0 \leq \mu < \frac{d-u}{d}$, $\mathcal{A}_2^{imp}$ is a linearizable implementation of read/write objects, which achieves: $|R| < \mu d + 4u + \epsilon$, $|W| = (1-\mu)d + 3u$.*

Proof: Fix any execution σ of $\mathcal{A}_2^{imp}$.

We first present a timing analysis to show the claimed bounds on $|R|$ and $|W|$.

The bound on $|W|$ is obvious since, by the algorithm, each process p_i waits till $\text{fmod}(Cl_i, 3u+\epsilon)$ holds, and returns after an additional time $(1-\mu)d$ elapses. We proceed to show that $|R| < \mu d + 4u + \epsilon$. Since, for a read operation, a process first waits for time μd and, then, it runs as in $\mathcal{A}_1^{imp}$, the proof of Theorem 3.3 immediately applies to yield: $|R| < \mu d + 4u + \epsilon$.

We turn to show that $\mathcal{A}_2^{imp}$ is a linearizable implementation.

We construct a legal linearization τ of σ such that, for each MCS process p_j, $\tau|j = \sigma|j$. Our construction proceeds in two phases.

The first phase is exactly identical to that in the proof of Theorem 3.3. Let τ' be the operation sequence resulting from the first phase. As in the proof of Theorem 3.3, τ' is a

Algorithm $\mathcal{A}_2^{imp}$: code for process p_i

Procedure $\mathsf{Read}_i(X)$; (/* executed upon a $Read_i(X)$ event */)
  $SetTimer_i(\mu d)$;
  if $timer_i(X) \neq \perp$ **then waitfor** $timer_i(X) \geq u$;
    $timer_i(X) := \perp$;
  endif;
  $X_i := \max_{<_V}\{v : (v,t) \in Pend_i(X)\}$;
  $Return_i(X, X_i)$;
end procedure $\mathsf{Read}_i(X)$;

Procedure $\mathsf{Write}_i(X, v)$; (/* executed upon a $Write_i(X, v)$ event */)
  waitfor $\mathrm{fmod}(Cl_i, 3u + b) > 3u$;
  $broadcast_i((update(X, v), Cl_i))$;
  $SetTimer_i((1-\mu)d)$;
  $Pend_i(X) := Pend_i(X) \cup \{(v, Cl_i - (1-\mu)d)\}$;
  $t_{max} := \max\{t' : (v', t') \in Pend_i(X)\}$;
  $Pend_i(X) := \{(v', t') : (v', t') \in Pend_i(X) \text{ and } t_{max} - t' \leq 2u\}$;
  $Ack_i(X)$;
end procedure $\mathsf{Write}_i(X, v)$;

Procedure $\mathsf{Update}_i(X)$ (/* executed upon receipt of $update((X, v), t)$ */)
  $Pend_i(X) := Pend_i(X) \cup \{(v, t)\}$;
  $t_{max} := \max\{t' : (v', t') \in Pend_i(X)\}$;
  $Pend_i(X) := \{(v', t') : (v', t') \in Pend_i(X) \text{ and } t_{max} - t' \leq 2u\}$;
  $timer_i(X) := 0$;
end procedure $\mathsf{Update}_i(X)$;

Figure 3-3: The Algorithm $\mathcal{A}_2^{imp}$—code for process p_i

linearization of σ such that for each MCS process p_j, $\tau'|j = \sigma|j$. Note that Claim 3.5 still applies, where $\overset{\sigma}{\rightsquigarrow}$ is identically defined.

In the second phase, we scan τ' and trace and fix each legality violation in it by "locally" reordering operations, while still preserving its properties. We show that the length of the maximal prefix of τ' which is a legal operation sequence strictly grows after each fix as we proceed; thus, inductively this results in a legal operation sequence τ which is a linearization of σ such that for each MCS process p_l, $\tau|l = \sigma|l$.

Formally, let $rop_1 = [Read_k(X), Return_k(X, v_1)]$ be the earliest read operation in τ', for some index $k \in [n]$, read/write object X, value $v_1 \in \mathcal{V}$, for which there exist operations $wop_1 = [Write_i(X, v_1), Ack_i(X)]$, $wop_2 = [Write_j(X, v_2), Ack_j(X)]$, $rop_1 = [Read_k(X), Return_k(X, v_1)]$, for some indices $i, j \in [n]$, and value $v_2 \in \mathcal{V}$, such that wop_1, wop_2, rop_1 is a subsequence of τ', $wop_1 \overset{\sigma}{\rightsquigarrow} rop_1$, and wop_2 is the latest write operation on X in τ' such that $wop_2 \xrightarrow{\tau'} rop_1$. Call rop_1 an *illegal* read operation. We consider two cases:

- Take $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| > 2u$, i.e., the local broadcasting times of wop_1 and wop_2 do not fall in the same time slice. Reorder rop_1 to immediately precede wop_2 in τ'.

- Take $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| \leq 2u$, i.e., the local broadcasting times of wop_1 and wop_2 fall in the same time slice. If there is no later read operation rop_2 in τ' such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$, then reorder wop_2 to immediately precede wop_1 in τ'; else, reorder rop_1 to immediately precede wop_2 in τ'. (We will soon provide justification for our construction.)

Iterate till no illegal read operation exists. Let τ be the resulting operation sequence.

We start by showing:

Lemma 3.8 *τ is a legal operation sequence.*

Proof: We proceed inductively and show that, after each reordering, the prefix of τ' ending with an illegal read operation rop_1 is a legal operation sequence. We consider separately each of the two cases in our construction.

Assume, first, $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| > 2u$. Clearly, by the definition of an illegal read operation, the prefix of τ' ending with rop_1 is a legal operation sequence.

Assume, next, $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| \leq 2u$. Notice that in this case it may be possible for a read operation invoked sufficiently "late" (after wop_1 and wop_2 terminate) to return v_1; hence, reordering rop_1 to precede wop_2 (as in the previous case) might not guarantee legality. We consider separately each of the two cases we considered in our construction.

Assume, first, that there is no later read operation, rop_2, on X in τ' such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$. As in Claim 3.10(2), we can show:

Claim 3.11 *There is no read operation rop_2 on X in $(wop_2, rop_1)_{\tau'}$ such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$. v_2.*

Hence, it follows that there is no read operation, rop_2, on X in τ' such that $wop_2 \xrightarrow{\sigma} rop_2$, and, after the reordering, the prefix of τ' ending with rop_1 is trivially legal.

Assume, now, that there is a read operation rop_2 on X in τ' such that $wop_2 \xrightarrow{\sigma} rop_2$. Notice that Claim 3.10 still applies and implies that there is no read operation rop_1' in τ' such that $wop_1 \overset{\sigma}{\rightsquigarrow} rop_1$ and $rop_2 \xrightarrow{\tau'} rop_1$. Hence, our reordering does not introduce any new legality violation.

Hence, after the reordering, the prefix of τ' ending with rop_1 is a legal operation sequence.

■

We continue by showing:

Lemma 3.9 *τ is a linearization of σ.*

Proof: We proceed by case analysis.

Assume, first, that $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| > 2u$, so that rop_1 is reordered to immediately precede wop_2 in τ'. It suffices to show that the reordered operations, rop_1 and wop_2, "overlap":

Claim 3.12 $t_r(wop_2) < t_c(rop_1)$.

Proof: Assume, by way of contradiction, that $t_r(wop_2) \geq t_c(rop_1)$. We have:

$$\begin{aligned} t_r(rop_1) - t_{br}(wop_2) &\geq t_r(rop_1) - (t_c(rop_1) - t_r(wop_2)) - t_{br}(wop_2) \\ &\geq \mu d + (1-\mu)d = d \ . \end{aligned}$$

Thus, the reading process receives $update(X, v_2)$ before time $t_r(rop_1)$, whence it returns v_1. A contradiction. ■

Assume, now, that $|Cl_i(t_{br}(wop_1)) - Cl_j(t_{br}(wop_2))| \leq 2u$, so that either wop_2 is reordered to immediately precede wop_1, or rop_1 is reordered to immediately precede wop_2 in case there is read operation, rop_2, on X in τ' such that $wop_2 \xrightarrow{\sigma} rop_2$. In the next two Claims, we establish that the corresponding reordered operations "overlap" in σ:

Claim 3.13 $t_c(wop_2) < t_r(wop_1)$.

Proof: Since the local broadcasting times of wop_1 and wop_2 fall in the same time slice, by Claim 3.8, $|t_{br}(wop_1) - t_{br}(wop_2)| \leq u + \epsilon$. Since, by assumption, $b \ll u$, we may assume that: $\epsilon < (1-\mu)d - u$. (Note that $(1-\mu)d - u > 0$ since, $\mu < \frac{d-u}{d}$.) Thus, we have:

$$t_c(wop_2) \leq t_{br}(wop_2) < t_{br}(wop_1) + u + b = t_r(wop_1) - (1-\beta)d + u + b < t_r(wop_1)$$

as needed. ■

Finally, we show:

Claim 3.14 *Assume there is a read operation rop_2 in σ such that $wop_2 \overset{\sigma}{\rightsquigarrow} rop_2$. Then, $t_c(rop_1) \leq t_r(wop_2)$.*

Proof: Assume, by way of contradiction, that $t_r(wop_2) < t_c(rop_1)$. We get:

$$t_r(rop_1) - t_{br}(wop_2) > t_r(rop_1) - (t_c(rop_1) - t_r(wop_2)) - t_{br}(wop_2) \geq \mu d + (1-\mu)d = d$$

Thus, $update(X, v_2)$ is received by p_k at time $< t_r(rop_1)$; hence, by the algorithm, $v_2 <_{\mathcal{V}} v_1$. But, the process performing rop_2 also receives $update(X, v_1)$ before it returns, since, by assumption, $t_r(rop_2) \geq t_r(rop_1)$; hence, $v_1 <_{\mathcal{V}} v_2$. A contradiction. ■

The Lemma follows from Claims 3.13 and 3.14. ■

Recall that for τ', $\tau'|l = \sigma|l$ for each MCS process p_l. In permuting τ' to obtain τ, for each case of an illegal read operation, we showed that the reordered operations "overlap". Hence, they may not be performed by the same process, implying:

Lemma 3.10 *For each MCS process p_l, $\tau|l = \sigma|l$.*

It follows from Lemmas 3.8, 3.9, and 3.10 that $\mathcal{A}_2^{imp}$ is a linearizable implementation of read/write objects. ■

3.3 Imperfect Clocks: Lower Bounds

This Section is organized as follows. In Subsection 3.3.1, we present a lower bound on $|R|+|W|$ for a certain class of sequentially consistent implementations, implying a corresponding lower bound for linearizable implementations. In Subsection 3.3.2, we present a lower bound on $|R|$ for linearizable implementations.

3.3.1 Lower Bounds on $|R| + |W|$

Our lower bounds on $|R| + |W|$ apply to a certain class of implementations of read/write objects, called *object-separable* and *object-symmetric*, which, roughly speaking, satisfy the following conditions:

- Each process acts on an interrupt event involving a certain read/write object independently of activity it previously performed on 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.

- Each process action on an interrupt event is symmetric with respect to the object involved in the event.

Formally, we define:

Definition 3.3.1 (Object separability, symmetry) *An implementation $\mathcal{A}$ of read/write objects is object-separable if, for each process p_i, every state s of p_i, includes $|\mathcal{X}|$ components, $s_1, s_2, \ldots, s_{|\mathcal{X}|}$, one for each read/write object, so that if an interrupt event e_k involves object X_k and (s, e_k, s', R, S, T) is a computation step of the process p_i, then $s'_l = s_l$ for every $l \neq k$. If, in addition, $s'_k = s'_l$ for any pair of interrupt events e_k and e_l involving objects X_k and X_l, respectively, then $\mathcal{A}$ is object-symmetric.*

We first present two technical lemmas which will be used in the proof of our main lower bound on $|R| + |W|$. These lemmas establish simple properties of sequentially consistent, object-separable and object-symmetric implementations, which are of independent interest.

Let $\mathcal{A}$ be any sequentially consistent, object-separable and object-symmetric implementation of read/write objects.

First Property

Roughly speaking, our first lemma asserts that, in any execution of $\mathcal{A}$, objects written identically by processes respond identically to read operations. This lemma is inspired by and generalizes Theorem 1 in [35], shown there for the special case where $u = d$.

Formally, consider objects X and Y, both holding the value v_0 at time 0, and construct an execution σ_1 of $\mathcal{A}$ consisting of the following call and response events at processes p_i and p_j, where $h_k(ops(\sigma_1))$, $k \in [n]$, denotes the sequence of ordered pairs of call and response events at p_k and their corresponding times in σ_1:

$$\begin{aligned} h_i(ops(\sigma_1)) \;=\; & (Write_i(Y, v), 0), (Ack_i(Y), t_{i,1}), \\ & (Read_i(X), t_{i,1}), (Return_i(X, v_i), t_{i,2}), \end{aligned}$$

$$\begin{aligned} h_j(ops(\sigma_1)) &= (Write_j(X,v),\frac{u}{2}),(Ack_j(X),t_{j,1}),\\ &\quad (Read_j(Y),t_{j,1}),(Return_j(X,v_j),t_{j,2}) \ . \end{aligned}$$

Furthermore, assume that in σ_1 delays of messages from any process to p_j are equal to d, delays of messages from p_j to any process are equal to $d-u$, and all other message delays are equal to $d-\frac{u}{2}$. Assume, also, that in σ_1, $Cl_k(0)=0$ for any $k \in [n]$, $k \neq j$, while $Cl_j(0)=-\frac{u}{2}$. We show:

Lemma 3.11 *Read operations by p_i and p_j in σ_1 must both return v.*

Proof: We start with an informal outline of our proof. By "perturbing" σ_1, we construct an execution σ_1', "symmetric" with respect to the objects X and Y, with the following properties: *(i)* each process "sees" the same event happening at the same (local) time in both σ_1 and σ_1', and *(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 read operations by p_i and p_j in σ_1' both return v, which follows from *(ii)* and object-symmetry.

Formally, we obtain σ_1' by retiming events and changing local clock times in σ_1 as follows:

- Each event at process p_j that occurs at (real) time t in σ_1 occurs at (real) time $t-\frac{u}{2}$ in σ_1', while each event at any other process occurs at the same (real) time in both σ_1 and σ_1'.
- The (local) clock time of p_j at (real) time t in σ_1' is equal to that in σ_1 plus $\frac{u}{2}$, while (local) clock times of all other processes do not change.

Thus, we have:

$$\begin{aligned} h_i(ops(\sigma_1')) &= (Write_i(Y,v),0),(Ack_i(Y),t_{i,1}),\\ &\quad (Read_i(X),t_{i,1}),(Return_i(X,v_i'),t_{i,2}),\\ h_j(ops(\sigma_1')) &= (Write_j(X,v),0),(Ack_j(X),t_{j,1}-\frac{u}{2}),\\ &\quad (Read_j(Y),t_{j,1}-\frac{u}{2}),(Return_j(X,v_j'),t_{j,2}-\frac{u}{2}) \ , \end{aligned}$$

where v_i' and v_j' are determined by $\mathcal{A}$. We show that σ_1' is an execution of $\mathcal{A}$ by proving:

Claim 3.15 *In σ_1', the time between a message-send event and the corresponding message-delivery event is equal to $d - \frac{u}{2}$.*

Proof: The only cases of interest involve message-send or message-delivery events at p_j:

1. Each message-send event at p_j, occurring at time t in σ_1, occurs at time $t - \frac{u}{2}$ in σ_1'. In σ_1', the corresponding message-delivery event occurs at time $t + d - u$, incurring a delay of $(t + d - u) - (t - \frac{u}{2}) = d - \frac{u}{2}$.

2. Each message-delivery event at p_j, occurring at time t in σ_1, occurs at time $t - \frac{u}{2}$ in σ_1'. In σ_1', the corresponding message-send event occurs at time $t - d$, incurring a delay of $(t - \frac{u}{2}) - (t - d) = d - \frac{u}{2}$.

■

Furthermore, consider an event at process p_j that happens at real time t and local clock time $Cl_j(t)$ in σ_1. In σ_1', this event occurs at real time $t - \frac{u}{2}$ and local clock time $Cl_j(t - \frac{u}{2}) + \frac{u}{2} = Cl_j(t) - \frac{u}{2} + \frac{u}{2} = Cl_j(t)$. Thus, p_j undergoes the same state changes in σ_1' as in σ_1, and so, $v_j' = v_j$. Also, by construction, p_i undergoes the same state changes in σ_1' as in σ_1, and so, $v_i' = v_i$. Therefore, it suffices to show that $v_i' = v_j' = v$.

Notice that in σ_1', by construction, $Cl_i(0) = 0$, while $Cl_j(0) = -\frac{u}{2} + \frac{u}{2} = 0$. Thus, local clocks of p_i and p_j are synchronized in σ_1'. Since all message delays are equal, object symmetry implies that $v_i' = v_j'$. Notice that $v_i' = v_j' = v_0$ contradicts sequential consistency. Therefore, $v_i' = v_j' = v$, as needed. ■

Second Property

Roughly speaking, our second lemma establishes that in any execution of $\mathcal{A}$ with "conflicting" write operations on some object, read operations on this object performed sufficiently "late" by different processes, that is, after these processes "hear" about the write operations, must return the same value.

Formally, consider an object X, holding the value x_0 at time 0, and construct an execution σ_1 of $\mathcal{A}$ consisting of the following call and response events at processes p_i, p_j, p_k and p_l:

$$\begin{aligned}
h_i(ops(\sigma_1)) &= (Write_i(X, v_1), 0), (Ack_i(X), t_i), \\
h_j(ops(\sigma_1)) &= (Write_j(X, v_2), 0), (Ack_j(X), t_j), \\
h_k(ops(\sigma_1)) &= (Read_k(X), t), (Return_k(X, v_k), t_k), \\
h_l(ops(\sigma_1)) &= (Read_l(X), t), (Return_l(X, v_l), t_l),
\end{aligned}$$

where $t > d + |W|$, so that any message sent by p_i or p_j is delivered before the read operations are invoked. Furthermore, assume that in σ_1 delays of all messages are equal and all clocks are initially synchronized. We show:

Lemma 3.12 *In σ_1, read operations by p_k and p_l return the same value.*

Proof: Assume, by way of contradiction, that $v_k \neq v_l$. We construct an execution σ_1' of $\mathcal{A}$ which is not sequentially consistent.

We start with an informal outline of our proof. By "augmenting" σ_1, we obtain an execution σ_1' 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 symmetry to argue that the operations on Y and Z must be "interleaved" in any legal serialization of σ_1'. This will prevent all read operations on X by one of p_k and p_l to precede all such of the other. Since σ_1' is an "augmentation" of σ_1, early read operations on X in σ_1' must return different values, as in σ_1. We use object-separability to argue that each later read operation on X returns the same value as the corresponding earlier one 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, holding the values y_0 and z_0, respectively, at time 0. By the serial specification of X, Y and Z, there exists an execution σ_1' of $\mathcal{A}$ consisting of the following

call and response events at processes p_i, p_j, p_k and p_l:

$$\begin{aligned}
h_i(ops(\sigma_1')) &= h_i(ops(\sigma_1)),\\
h_j(ops(\sigma_1')) &= h_j(ops(\sigma_1)),\\
h_k(ops(\sigma_1')) &= (Read_k(X), t), (Return_k(X, v_k'), t_k),\\
&\quad (Write_k(Y, y_1), t_k), (Ack_k(Y), t_{k,1}),\\
&\quad (Read_k(Z), t_{k,1}), (Return_k(Z, z_1'), t_{k,2}),\\
&\quad (Read_k(X), t_{k,2}), (Return_k(X, v_k''), t_{k,3}),\\
h_l(ops(\sigma_1')) &= (Read_l(X), t), (Return_l(X, v_l'), t_l),\\
&\quad (Write_l(Z, z_1), t_l), (Ack_l(Z), t_{l,1}),\\
&\quad (Read_l(Y), t_{l,1}), (Return_l(Y, y_1'), t_{l,2}),\\
&\quad (Read_l(X), t_{l,2}), (Return_l(X, v_l''), t_{l,3}),
\end{aligned}$$

Furthermore, assume that, in σ_1', delays of all messages are equal and all clocks are initially synchronized.

By object-separability, $v_k'' = v_k'$ and $v_l'' = v_l'$. Since all message delays are equal, object-symmetry implies that either $y_1' = y_1$ and $z_1' = z_1$, or $y_1' = y_0$ and $z_1' = z_0$. Notice, however, that $y_1' = y_0$ and $z_1' = z_0$ contradicts sequential consistency. Hence, $y_1' = y_1$ and $z_1' = z_1$.

Since σ_1' is sequentially consistent, there exists a legal serialization τ of σ_1', respecting the order of events at each process. In τ, either the second read operation on X by p_k precedes the first read operation on X by p_l, or the second read operation on X by p_l precedes the first read operation on X by p_k; assume, without loss of generality, the former. Note, however, that the first read operation on X by p_l precedes the write operation on Z by p_l (since $\tau|l = \sigma_1'|l$), which precedes the read operation on Z by p_k (by the serial specification of Z), which precedes the second write operation on X by p_k (since $\tau|k = \sigma_1'|k$). A contradiction. ■

We now present our main lower bound result:

Theorem 3.5 *In any sequentially consistent, object-separable and object-symmetric implementation of at least three objects, accessed by at least four processes,* $|R| + |W| \geq d + \frac{u}{2}$.

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 $|R|+|W| < d+\frac{u}{2}$. We construct an execution of $\mathcal{A}$ which is not sequentially consistent.

Informally, our proof proceeds as follows. We construct an execution σ of $\mathcal{A}$ in which two processes, p_i and $p_{i'}$, each perform an early and a late read operation on an object X; we use symmetry to "force" p_i and $p_{i'}$ 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_j and p'_j. We appropriately choose message delay times in σ so that, under the assumption $|R| + |W| < d + \frac{u}{2}$, p_i "gathers" fast information about the write operation by p_j, but cannot "hear" about the write operation by p'_j till late. (The roles of delays of messages from p_j and p'_j are reversed for $p_{i'}$.) Thus, by symmetry, read operations by p_i and $p_{i'}$ return values in different order, establishing the contradiction. We now present the details of the formal proof.

Consider objects X and Y, both holding the value v_0 at time 0, and construct an execution σ of $\mathcal{A}$ consisting of the following call and response events at processes p_i, p_j, $p_{i'}$ and $p_{j'}$:

$$\begin{aligned}
h_i(ops(\sigma)) &= (Write_i(Y, v_1), 0), (Ack_i(Y), t_{i,1}), \\
&\quad (Read_i(X), t_{i,1}), (Return_i(X, x_i), t_{i,2}), \\
&\quad (Read_i(X), t_{i,3}), (Return_i(X, x'_i), t_{i,4}), \\
h_{i'}(ops(\sigma)) &= (Write_{i'}(Y, v_2), 0), (Ack_{i'}(Y), t_{i',1}), \\
&\quad (Read_{i'}(X), t_{i',1}), (Return_{i'}(X, x_{i'}), t_{i',2}), \\
&\quad (Read_{i'}(X), t_{i',3}), (Return_{i'}(X, x_{i''}), t_{i',4}), \\
h_j(ops(\sigma)) &= (Write_j(X, v_1), \frac{u}{2}), (Ack_j(X), t_{j,1}), \\
&\quad (Read_j(Y), t_{j,1}), (Return_j(Y, y_j), t_{j,2}),
\end{aligned}$$

$$\begin{aligned} h_{j'}(ops(\sigma)) &= (Write_{j'}(X, v_2), \frac{u}{2}), (Ack_{j'}(X), t_{j',1}), \\ & \quad (Read_{j'}(Y), t_{j',1}), (Return_{j'}(Y, y_{j'}), t_{j',2}) \ . \end{aligned}$$

Furthermore, assume that, in σ, delays of messages from p_j to any process p_k, $k \in [n]$, are equal to $d - u$ if $k \neq i'$, and d if $k = i'$, delays of messages from p'_j to any process p_k, $k \in [n]$, are equal to $d - u$ if $k \neq i$ and d if $k = i$, and all other message delays are equal to $d - \frac{u}{2}$. Assume, also, that, in σ, $Cl_k = 0$ for any $k \in [n]$, $k \neq j, j'$, while $Cl_j(0) = Cl_{j'}(0) = -\frac{u}{2}$. Let also $t_{i,3}, t_{i',3} > \frac{u}{2} + |W|$, so that any message sent by p_j or $p_{j'}$ while performing write operations on X is delivered before the late read operations on X by p_i and $p_{i'}$ are invoked.

Since, by assumption, $|R| + |W| < d + \frac{u}{2}$, it follows that $t_{i,2} < d + \frac{u}{2}$; hence, the assumed message delays imply that p_i may not receive a message from $p_{j'}$ till after time $t_{i,2}$. Thus, Lemma 3.11 applies on the prefices of $\sigma|i$ and $\sigma|j$ consisting of all events at p_i and p_j occurring at time $\leq t_{i,2}$ in σ to yield: $x_i = v_1$. A symmetric argument yields: $x_{i'} = v_2$.

By the symmetry in delays of messages sent by the processes writing X, p_j and $p_{j'}$ to p_i and $p_{i'}$, there are two possibilities: either $x'_i = v_2$ and $x'_{i'} = v_1$, or $x'_i = v_1$ and $x'_{i'} = v_2$. Clearly, the first possibility immediately contradicts sequential consistency. On the other hand, the second possibility contradicts, by object-separability, Claim 3.12. ■

We remark that although, apparently, the assumption of at least three objects is not explicitly used in the Proof of Theorem 3.5, this assumption is necessary since it is made for the proof of Lemma 3.12.

Since linearizability implies sequential consistency, it immediately follows:

Corollary 3.1 *In any linearizable, object-separable and object-symmetric implementation of at least three objects, using at least four processes,* $|R| + |W| \geq d + \frac{u}{2}$.

3.3.2 Lower Bound on $|R|$

We show:

Theorem 3.6 *In any linearizable implementation of read/write objects accessed by at least three processes,* $|R| \geq \frac{u}{2}$.

Proof: Assume, by way of contradiction, that there exists a linearizable implementation $\mathcal{A}$ of such objects for which $|R| < u/2$. We construct an execution of $\mathcal{A}$ which is not linearizable.

Consider an object X, holding the value x_0 at time 0, and let p_i and p_j be two processes that read X, and p_k be a process that writes X.

Informally, our proof proceeds as follows: We start with an execution in which p_i reads x_0 from X, p_i and p_j alternate reading from X while p_k writes x_1 to X, and finally p_j reads x_1 from X. Thus, there exists a read operation, rop_0, say by p_i, that returns x_0 and is immediately followed by a read operation, rop_1, by p_j that returns x_1. If p_i's history is shifted later by $\frac{u}{2}$, while p_j's history is shifted earlier by $\frac{u}{2}$, there results an execution in which rop_1 precedes rop_0. Since rop_1 returns x_1, while rop_0 returns x_0, this contradicts linearizability. We now present the details of the formal proof.

Let $b = \lceil \frac{|W(X)|}{u} \rceil$. By the serial specification of X, there exists an execution σ_1 of $\mathcal{A}$ consisting of the following call and response events at processes p_i, p_j and p_k:

$$\begin{aligned}
h_i(ops(\sigma)) &= (Read_i(X), 0), (Return_i(X, v_0), t_0), \\
&\quad (Read_i(X), u), (Return_i(X, v_2), t_2), \ldots, \\
&\quad (Read_i(X), bu), (Return_i(X, v_{2b}), t_{2b}), \\
h_j(ops(\sigma)) &= (Read_j(X), \frac{u}{2}), (Return_j(X, v_1), t_1), \\
&\quad (Read_j(X), \frac{3u}{2}), (Return_j(X, v_3), t_3), \ldots, \\
&\quad (Read_j(X), bu + \frac{u}{2}), (Return_j(X, v_{2b+1}), t_{2b+1}), \\
h_k(ops(\sigma)) &= (Write_k(X, x_1), \frac{u}{2}), (Ack_k(X), t_r),
\end{aligned}$$

Furthermore, we assume that the message delays in σ_1 are as follows: Each message from p_i to p_l, $l \neq i$, incurs of delay of either d if $l = j$ or $d - \frac{u}{2}$ if $l \neq j$; each message from p_j to p_l, $l \neq j$, incurs a delay of either $d - u$ if $l = i$ or $d - \frac{u}{2}$ if $l \neq i$.

Since, by the definition of b,

$$t_r \leq \frac{u}{2} + |W(X)| < \frac{u}{2} + bu,$$

and σ_1 is linearizable, it follows that: $v_{2b+1} = x_1$. Note also that for $0 \leq l \leq 2b+1$:

$$t_l \leq l\frac{u}{2} + |R(X)| < l\frac{u}{2} + \frac{u}{2},$$

which, for $l = 0$, implies, by linearizability, that $v_0 = x_0$. It also follows by linearizability that there exists an index r, $0 \leq r \leq 2b$ such that $v_r = x_0$ and $v_{r+1} = x_1$. We assume, without loss of generality, that r is even, so that v_r is the result of a read operation by p_j.

We now show how to "perturb" σ_1 to obtain another execution σ_2 of $\mathcal{A}$ that is not linearizable. We assign times to events in σ_2: Each event at process p_i that happens at (real) time t in σ_1 will occur at (real) time $t + \frac{u}{2}$ in σ_2, while each event at process p_j that happens at (real) time t' in σ_1 will occur at (real) time $t' - \frac{u}{2}$ in σ_2. Let $Cl_i(t)$ and $Cl_j(t)$ be the (local) clock times of processes p_i and p_j, respectively, at (real) time t in σ_1. The (local) clock times of p_i and p_j at time t in γ_2 will be $Cl_i(t) - \frac{u}{2}$ and $Cl_j(t) + \frac{u}{2}$, respectively. Times of events and (local) clock times of any other process do not change.

We start by showing:

Lemma 3.13 *In σ_2, the time between a message-send event and the corresponding message-delivery event is in the range $[d-u, d]$.*

Proof: We proceed by case analysis:

1. Each message that is sent by p_i at (real) time t and delivered to p_j at (real) time $t+d$ in σ_1 will be sent and delivered at (real) times $t + \frac{u}{2}$ and $t + d - \frac{u}{2}$, respectively, in σ_2, incurring a delay of $t + d - \frac{u}{2} - (t + \frac{u}{2}) = d - u$.

2. Each message that is sent by p_j at (real) time t and delivered to p_i at (real) time $t+d-u$ in σ_1 will be sent and delivered at (real) times $t - \frac{u}{2}$ and $t + d - u + \frac{u}{2}$, respectively, in σ_2, incurring a delay of $t + d - u + \frac{u}{2} - (t - \frac{u}{2}) = d$.

3. Each message that is sent by p_i at (real) time t and delivered at a process other than p_j at (real) time $t + d - \frac{u}{2}$ in σ_1 will be sent and delivered at (real) times $t + \frac{u}{2}$ and $t + d - \frac{u}{2}$, respectively, in σ_2, incurring a delay of $t + d - \frac{u}{2} - (t + \frac{u}{2}) = d - u$.

4. Each message that is sent by p_j at (real) time t and delivered at a process other than p_i at (real) time $t + d - \frac{u}{2}$ in α_1 will be sent and delivered at (real) times $t - \frac{u}{2}$ and $t + d - \frac{u}{2}$, respectively, in σ_2, incurring a delay of $t + d - \frac{u}{2} - (t - \frac{u}{2}) = d$.

■

We next argue that each process "sees" the same event happening at the same local clock time in both σ_1 and σ_2: The only non-trivial cases are when this event involves either p_i or p_j. Consider an event involving p_i that occurs at real time t in σ_1. Let $Cl_i(t)$ be the local clock time of p_i at real time t in σ_1. By construction, this event will occur at (real) time $t + \frac{u}{2}$ in γ_1 when the local clock time of p_i will be: $Cl_i(t + \frac{u}{2}) - \frac{u}{2} = Cl_i(t) + \frac{u}{2} - \frac{u}{2} = Cl_i(t)$, as needed. The case where the event involves p_j is similar. Thus, σ_2 is a collection of process histories and, therefore, by Lemma 3.13, an execution of $\mathcal{A}$.

Finally, note that in the execution σ_2, the order of the values returned by read operations on X performed by p_i and p_j is: $v_1, v_0, v_3, v_2, \ldots, v_{r+1}, v_r, \ldots$. Thus, $v_{r+1} = x_1$ is being read before $v_r = x_0$, which contradicts linearizability. ■

We remark that the general outline of the lower bound proof follows [13]. Our improvement over [13] is achieved by carefully choosing message delays in the construction of σ_1.

Chapter 4

Semi-Synchrony versus Asynchrony

In this Chapter, we compare and contrast the *asynchronous* and *semi-synchronous* models of distributed computation by presenting upper and lower bounds for the time complexity of solving the s-session problem.

This Chapter is organized as follows. Section 4.1 includes our bounds for *network* models, where interprocess communication is achieved through point-to-point message passing between processes. Section 4.2 includes our bounds for *shared memory*, where interprocess communication is achieved through a collection of *shared variables* that may be read and written by processes.

4.1 Networks

This Section is organized as follows. Subsection 4.1.1 includes our upper bounds for both the asynchronous and semi-synchronous models, while Subsection 4.1.2 includes our lower bounds for both models. In Subsection 4.1.3, we consider networks with non-uniform delays and state the corresponding upper and lower bounds. In Subsection 4.1.4, the uninitialized case is considered, and upper and lower bounds for both models are presented.

4.1.1 Upper Bounds

The Asynchronous Model

We start with a simple asynchronous algorithm in which processes communicate in order to learn about completion of a session before advancing to the next session. Each process maintains as part of its state a variable that gives its current session number; upon hearing that every other process has reached its current session, it increments its session number by one and notifies all other processes. Notification is done by sending messages along a shortest-path tree rooted at it. The process enters an idle state when its session number is set to s. We prove:

Theorem 4.1 *Let G be any graph. There exists an asynchronous algorithm, $\mathcal{A}^{as}$, that solves the s-session problem on G within time $diam(G)D(s-1)$.*

Proof: We describe an asynchronous algorithm, $\mathcal{A}^{as}$, that solves the s-session problem on G within time $diam(G)D(s-1)$; that is, in any execution of $\mathcal{A}^{as}$ there are at least s sessions and all processes enter an idle state no later than time $diam(G)D(s-1)$. The algorithm is described here informally; this description can be easily translated into a state transition function.

For each $i \in [n]$, the state of p_i consists of the following components: *buffer* — a buffer, an unordered set of elements of $\mathcal{M}$, initially $\emptyset$; *session* — a nonnegative integer, initially 1. The message alphabet, $\mathcal{M}$, consists of the pairs (i,k), where $i \in [n]$ and $1 \leq k \leq s-1$. The initial state of p_i is non-idle.

The algorithm is as follows. Upon taking its first computation step, p_i broadcasts $(i,1)$. If for all $j \in [n]$, $(j, session_i) \in buffer_i$, p_i increments *session* by 1. If $session_i = s$, p_i enters an idle state and remains in this state forever. Otherwise, p_i broadcasts $(i, session_i)$.

We assume that messages from a process are flooded on a *shortest path tree* rooted at this process. That is, $\mathcal{A}^{as}$ uses a routing algorithm by which, for any nodes $u, v \in G$, a message from u to v is routed through exactly $dist(u,v)$ communication links in G. The details of how this is done are not discussed here; the reader is referred to, e.g., [54].

If $session_i = k$, we say that p_i is in its kth session. The message (i,k) can be interpreted as "process i executed a step in the kth session".

We start by showing that in any execution of $\mathcal{A}^{as}$ there are at least s sessions. Fix an arbitrary timed execution α of $\mathcal{A}^{as}$. Clearly, each process p_i receives $(j,1)$ for all $j \in [n]$ and sets $session_i$ to 2. By induction, it is simple to show that for each k, $1 \leq k \leq s-1$, p_i sets $session_i$ to k in α. For each k, $1 \leq k \leq s$, define α_k to be the longest prefix of α that does not include a configuration in which, for some $i \in [n]$, $session_i \geq k$, i.e., no process has passed its kth session. Note that $\alpha_1 = \lambda$, the *empty sequence*, and that for each k, $1 \leq k \leq s-1$, α_k is a prefix of α_{k+1}. For each k, $1 \leq k \leq s-1$, let β_k be such that $\alpha_{k+1} = \alpha_k \beta_k$; let β_s be such that $\alpha = \alpha_s \beta_s$.

Lemma 4.1 *For each k, $1 \leq k \leq s-1$, there is a session in β_k.*

Proof: Let p_i be a process which sets $session_i$ to $k+1$. By definition, this event is not in β_k. By the algorithm, this implies that for each j, $j \in [n]$ and $j \neq i$, p_i has received a (j,k) message. However, by definition, no process p_j has $session_j \geq k$ in α_k. Thus, by the algorithm, no process p_j sends a (j,k) message in α_k. Hence, there is a step by every process, and, therefore, a session in β_k. ■

In addition, there is a session in β_s, since, for every $i \in [n]$, a computation step is included in β_s at which p_i sets $session_i$ to s. (Note that, by the definition of α_s, such a step cannot be included in α_s.) This implies that there are at least s sessions in α. Since α was chosen arbitrarily, this implies the correctness of $\mathcal{A}^{as}$. We now analyze the time complexity of $\mathcal{A}^{as}$.

Informally, the next definition captures the latest time at which the kth session can be completed. For each k, $1 \leq k \leq s-1$, define

$$T_k = \max_{i \in V} \{t : p_i \text{ sets } session_i \text{ to } k \text{ at time } t \text{ in } \alpha\} .$$

By the algorithm, $T_1 = 0$. We have:

Lemma 4.2 *For each k, $1 \leq k < s$, $T_{k+1} \leq T_k + diam(G)D$.*

Proof: Fix some process p_i, and let t be the time at which p_i broadcasts (i,k); note that, by definition, $t \leq T_k$. Clearly, for every process p_j, the event $del(j,(i,k))$, delivering the message (i,k) to p_j, will occur at time $\leq t + (diam(G)-1)D + d$. Thus, by time $t + diam(G)D$, p_i has a computation step in which (i,k) is in the buffer. Thus, by time $T_k + diam(G)D$ every process has a computation step in which (i,k) is in the buffer, for every $i \in [n]$. By the algorithm, at this step the process sets its *session* variable to $k+1$. The claim follows. ■

Since $T_1 = 0$, it follows that $T_s \leq diam(G)D(s-1)$. Hence, every process enters an idle state after setting *session* to s, no later than time $diam(G)D(s-1)$. Thus, $\mathcal{A}^{as}$ solves the s-session problem on G within time $diam(G)D(s-1)$. ■

The Semi-Synchronous Model

In the semi-synchronous model, we can slightly improve $\mathcal{A}^{as}$ by taking advantage of the available initial synchronization; specifically, each process operates exactly as in $\mathcal{A}^{as}$, except that it does not wait to hear that every other process has completed its first session, but passes directly to the second one upon taking its second step. We prove:

Theorem 4.2 *Let G be any graph. There exists a semi-synchronous algorithm, $\mathcal{A}_1^{ss}$, that solves the s-session problem on G within time $1 + diam(G)D(s-2)$.*

Proof: We describe a semi-synchronous algorithm, $\mathcal{A}_1^{ss}$, which is very similar to $\mathcal{A}^{as}$ and solves the s-session problem on G within time $1 + diam(G)D(s-2)$; that is, in any execution of $\mathcal{A}^{as}$ there are at least s sessions and all processes enter an idle state no later than time $1 + diam(G)D(s-2)$.

For each $i \in [n]$, the state of p_i consists of the following components: *buffer* — a buffer, an unordered set of elements of $\mathcal{M}$, initially $\emptyset$; *session* — a nonnegative integer, initially 1. The message alphabet, $\mathcal{M}$, consists of the pairs (i,k) where $i \in [n]$ and $2 \leq k \leq s-1$. The initial state of p_i is non-idle.

Upon taking its second computation step, p_i increments $session_i$ to 2 and broadcasts $(i,2)$. If for all $j \in [n]$, $(j, session_i) \in buffer_i$, p_i increments $session_i$ by 1. If $session_i = s$, p_i

enters an idle state and remains in this state for ever. Otherwise, p_i broadcasts $(i, session_i)$. As in $\mathcal{A}^{as}$, we assume that messages from a process are flooded on a shortest path tree rooted at this process. We say that p_i is in its kth session if $session_i = k$ and we interpret the message (i, k) as "process i executed a step in the kth session".

We start by showing that in any execution of $\mathcal{A}_1^{ss}$ there are at least s sessions. Fix an arbitrary execution α of $\mathcal{A}_1^{ss}$. For each k, $1 \leq k \leq s$, define α_k to be the longest prefix of α that does not include a configuration in which, for some $i \in [n]$, $session_i \geq k$, i.e., no process has passed its kth session. Note that $\alpha_1 = \lambda$, and that for each k, $1 \leq k \leq s-1$, α_k is a prefix of α_{k+1}. For each k, $1 \leq k \leq s-1$, let β_k be such that $\alpha_{k+1} = \alpha_k \beta_k$; let β_s be such that $\alpha = \alpha_s \beta_s$.

Lemma 4.3 *There is a session in β_1.*

Proof: Note that $\beta_1 = \alpha_2$, since $\alpha_1 = \lambda$. For every process p_i, the steps of p_i that are included in α_2 are exactly those that occur at time 0. Since every process has a step at time 0, there is a session in $\alpha_2 = \beta_1$. ■

As in Lemma 4.1, we can prove:

Lemma 4.4 *For each k, $1 \leq k \leq s-1$, there is a session in β_k.*

In addition, there is a session in β_s. This implies that there are at least s sessions in α. Since α was chosen arbitrarily, this implies the correctness of $\mathcal{A}_1^{ss}$.

We now analyze the time complexity of $\mathcal{A}_1^{ss}$. For each k, $2 \leq k \leq s-1$, we define:

$$T_k = \max_{i \in V} \{t : p_i \text{ sets } session_i \text{ to } k \text{ at time } t \text{ in } \alpha\} .$$

Note that $T_2 \leq 1$. In addition, as in Lemma 4.2, we have:

Lemma 4.5 *For each k, $2 \leq k < s$, $T_{k+1} \leq T_k + diam(G)D$.*

Since $T_2 \leq 1$, it follows that $T_s \leq 1 + diam(G)D(s-2)$. Every process enters an idle state after setting *session* to s, no later than time $1 + diam(G)D(s-2)$. Thus, $\mathcal{A}_1^{ss}$ solves the s-session problem on G within time $1 + diam(G)D(s-2)$. ■

We next show that the timing information available in the semi-synchronous model can be exploited to obtain a bound which is sometimes better than the previous bound. This algorithm uses no communication; intuitively, this means that no process state transition can result in a send action. Formally, an algorithm $\mathcal{A}$ *uses no communication* if for every i, $i \in [n]$, for every $q \in Q_i$, $\mathcal{A}_i(q) = (q', \emptyset)$ for some $q' \in Q_i$. We prove:

Theorem 4.3 *Let G be any graph. There exists a semi-synchronous algorithm, $\mathcal{A}_2^{ss}$, which solves the s-session problem on G within time $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$. Furthermore, $\mathcal{A}_2^{ss}$ uses no communication.*

Proof: We describe a semi-synchronous algorithm, $\mathcal{A}_2^{ss}$, which solves the s-session problem on G within time $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$. For each $i \in [n]$, the state of p_i consists of a *counter*, an integer, initially -1. The initial state of p_i is non-idle. At each computation event, p_i increments $counter_i$ by 1; p_i enters an idle state when $counter_i$ is equal to $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$.

We start by showing that in any execution of $\mathcal{A}_2^{ss}$ there are at least s sessions. Consider an arbitrary execution α of $\mathcal{A}_2^{ss}$. We partition α into execution fragments, $\alpha = \alpha_0 \alpha_1 \ldots \alpha_{s-1}$, such that: *(i)* α_0 consists only of the computation steps at time 0, and *(ii)* for each k, $1 \leq k \leq s-2$, $\alpha_0 \ldots \alpha_k$ is the shortest prefix of α that includes a configuration in which, for some $i \in [n]$, $counter_i = k(\lfloor \frac{1}{c} \rfloor + 1)$. We have:

Lemma 4.6 *For each k, $1 \leq k \leq s-2$, there is a session in α_k.*

Proof: Let p_i be the first process to set $counter_i$ to $k(\lfloor \frac{1}{c} \rfloor + 1)$ in α. By the definition of α_k, the steps at which $counter_i$ is equal to $(k-1)(\lfloor \frac{1}{c} \rfloor + 1) + j$, for $1 \leq j \leq \lfloor \frac{1}{c} \rfloor + 1$, are included in α_k. Thus, there are at least $\lfloor \frac{1}{c} \rfloor + 1$ steps by p_i in α_k. These steps take time at least $c(\lfloor \frac{1}{c} \rfloor + 1) > c\frac{1}{c} = 1$; thus, there exists a computation step by every process and, therefore, a session in α_k. ■

In addition, there is a session in α_0 since every process takes a step at time 0.

There is also a session in α_{s-1}, since, by the definition of α_{s-1}, each process sets its counter to $1 + (s-2)(\lfloor \frac{1}{c} \rfloor + 1)$ at its last non-idle step. Together with Lemma 4.6, this implies that there are at least s sessions in any timed execution of $\mathcal{A}_2^{ss}$.

Each process will enter an idle state no later than time $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$, since for any process the time between successive computation steps is at most 1. Thus, $\mathcal{A}_2^{ss}$ solves the s-session problem on G within time $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$. ■

When d and $diam(G)$ are known, it is possible to calculate in advance which of the algorithms of Theorems 4.2 and 4.3 is faster, and run it. Moreover, even if d and $diam(G)$ are not known, it is possible to run these algorithms "side by side", halting when the first of them does. In both cases, we get:

Theorem 4.4 *Let G be any graph. There exists a semi-synchronous algorithm, $\mathcal{A}^{ss}$, which solves the s-session problem on G within time $1 + \min\{\lfloor \frac{1}{c} \rfloor + 1, diam(G)D\}(s-2)$.*

4.1.2 Lower Bounds

In all of our lower bound proofs, we use an infinite timed execution in which processes take steps in round-robin order, starting with p_1, with step time close to 1, and all messages incur a delay of exactly d. It is called a *slow, synchronous* timed execution.

The Asynchronous Model

We start by showing that for the asynchronous model, the algorithm presented in Theorem 4.1 is optimal. The proof of the following theorem is based on delaying information propagation and then perturbing an execution to obtain an execution of the algorithm which does not include s sessions.

Theorem 4.5 *Let G be any graph. There does not exist an asynchronous algorithm which solves the s-session problem on G within time strictly less than $diam(G)d(s-1)$.*

Proof: Assume, by way of contradiction, that there exists an asynchronous algorithm, $\mathcal{A}$, which solves the s-session problem on G within time strictly less than $diam(G)d(s-1)$. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

The following is an informal outline of the proof. We start with a slow, synchronous timed execution of $\mathcal{A}$ and partition it into $s-1$ execution fragments each of which is completed within

time $< diam(G)d$. Since communication is slow, there is no communication between any pair of antipodal nodes within each fragment. By "retiming" we will perturb each fragment to get a new execution fragment in which there is a "fast" peripheral node which takes all of its steps before a "slow" antipodal node takes any of its steps. Our construction will have the "slow" node of each execution fragment be identical to the "fast" node of the next execution fragment. In each execution fragment, a session can be completed as soon as the "slow" peripheral node takes its first computation step; since the "fast" peripheral node does not take any more computation steps, no more sessions can be completed in this execution fragment. This will guarantee that at most one session is contained in each execution fragment; thus, the total number of sessions in the "retimed" execution is at most $s-1$, contradicting the correctness of $\mathcal{A}$.

We now present the details of the formal proof.

Pick some ε such that $0 < \varepsilon \leq (diam(G)d(s-1)+1)^{-1}$. Consider a slow, synchronous timed execution $\gamma = \alpha\alpha'$ of $\mathcal{A}$, with step time $1-\varepsilon$, where α is the shortest prefix of γ such that all processes are in an idle state in $last(\alpha)$ and α' is the remaining part of γ. We perturb $\alpha\alpha'$ to obtain another timed execution $\beta\beta'$ that does not include s sessions.

We first show how to modify α to obtain β. By assumption, $t_{end}(\alpha) < diam(G)d(s-1)$. Write $\alpha = \alpha_0\alpha_1 \ldots \alpha_{s-1}$, where $\alpha_0 = \lambda$ and for each k, $1 \leq k \leq s-1$, $t_{end}(\alpha_k) - t_{end}(\alpha_{k-1}) < diam(G)d(1-\varepsilon) < diam(G)d$. (We adopt the convention that $t_{end}(\alpha_0) = 0$.) For some sequence $i_0, \ldots, i_{s-1}$ of peripheral nodes, we construct from each execution fragment α_k an execution fragment $\beta_k = \rho_k\sigma_k$, such that:

(1) ρ_k contains no computation step of $p_{i_{k-1}}$, and

(2) σ_k contains no computation step of p_{i_k}.

In this construction, i_{k-1} is the "fast" node which takes all of its steps in the execution fragment ρ_k, before the "slow" node i_k takes any of its steps. (All the steps of i_k are in σ_k.) Our construction uses peripheral nodes since they maximize the time to transfer information to other nodes, which is roughly $diam(G)d$. In particular, i_{k-1} will be antipodal to i_k.

We now show, for each k, $1 \leq k \leq s-1$, how to construct β_k, by induction on k. For the base case, let i_0 be an arbitrary peripheral node of G, and take β_0 to be λ.

Assume we have picked $i_0, \ldots, i_{k-1}$ and constructed $\beta_0, \ldots, \beta_{k-1}$. Let i_k be some node that is antipodal to i_{k-1}, i.e., $dist(i_{k-1}, i_k) = diam(G)$; note that i_k is also peripheral. We now show how to construct β_k.

For any node u, ρ_k includes all events at u that occur at time $< t_{end}(\alpha_{k-1}) + dist(u, i_{k-1})d$ in α_k; σ_k includes all events at u that occur at time $\geq t_{end}(\alpha_{k-1}) + dist(u, i_{k-1}d)$ in α_k. Events at each process occur in the same order as in α_k and all occur at time 0, in both ρ_k and σ_k. In addition, ordering of events across different processes that occur at the same time in α_k is preserved within each of ρ_k and σ_k. Since

$$t_{end}(\alpha_{k-1}) + dist(i_k, i_{k-1})d = t_{end}(\alpha_{k-1}) + diam(G)d > t_{end}(\alpha_k) \ ,$$

and all events at i_k occur at time $\leq t_{end}(\alpha_k)$ in α_k, this implies that all events at i_k will appear in ρ_k. On the other hand, since

$$t_{end}(\alpha_{k-1}) + dist(i_{k-1}, i_{k-1})d = t_{end}(\alpha_{k-1}) \ ,$$

and all events at i_{k-1} in α_k occur at time $\geq t_{end}(\alpha_{k-1})$, all events at i_{k-1} in α_k will appear in σ_k. Thus, $\beta_k = \rho_k \sigma_k$ has properties (1) and (2) above.

Let $\beta = \beta_0 \beta_1 \ldots \beta_{s-1}$.

By construction, events at each process p_i, $i \in [n]$, occur in the same order in β as in α. Hence, p_i undergoes the same state changes in β as in α, and, therefore, $state_i(last(\beta)) = state_i(last(\alpha))$.

We now modify α' to obtain β'. The first computation step of any process in β' will occur at time 1 and all later computation steps of it are 1 time unit apart. Any message delivery event at a process will occur at time d after the corresponding message sending event.

We next establish that $\beta\beta'$ is a timed execution of $\mathcal{A}$. We start by showing:

Lemma 4.7 *Each receive event is after the corresponding send event in $\beta\beta'$.*

Proof: Consider the message send event π_1 at node u_1 which occurs at time t_1 in $\alpha\alpha'$ and let π_2 be the corresponding message delivery event at node u_2 which occurs at time t_2 in $\alpha\alpha'$. Note that u_1 and u_2 are neighboring processes, i.e., $dist(u_1, u_2) = 1$. Hence, $dist(u_1, i_{k-1}) + 1 \geq dist(u_2, i_{k-1})$. The only non-trivial case is when π_1 and π_2 occur in the same α_k, for some k, $1 \leq k \leq s-1$. We show that the ordering of π_1 and π_2 is the same in β_k as in α_k.

The only case of interest is when π_1 occurs in σ_k, while π_2 occurs in ρ_k. In this case, $t_1 \geq t_{end}(\alpha_{k-1}) + dist(u_1, i_{k-1})d$, while $t_2 < t_{end}(\alpha_{k-1}) + dist(u_2, i_{k-1})d$. Then,

$$\begin{aligned} t_2 &= t_1 + d \\ &\geq t_{end}(\alpha_{k-1}) + dist(u_1, i_{k-1})d + d \qquad \text{(since } \pi_1 \text{ occurs in } \sigma_k\text{)} \\ &= t_{end}(\alpha_{k-1}) + (dist(u_1, i_{k-1}) + 1)d \\ &\geq t_{end}(\alpha_{k-1}) + dist(u_2, i_{k-1})d \ , \end{aligned}$$

a contradiction. ■

All events in β occur at time 0; in β', computation steps occur at step time 1 and all messages incur a delay of exactly d. Since there are no lower bounds on either process step time or message delivery time in the asynchronous model, we have:

Lemma 4.8 *Lower and upper bounds on step time are preserved in $\beta\beta'$.*

Lemma 4.9 *Lower and upper bounds on message delay time are preserved in $\beta\beta'$.*

To derive a contradiction, we prove:

Lemma 4.10 *There are at most $s-1$ sessions in β.*

Proof: We show, by induction on k, that $\beta_0 \ldots \beta_{k-1}\rho_k.$ does not contain k sessions, for $1 \leq k \leq s-1$.

For the base case, note that, by construction, $\beta_0 = \lambda$ and ρ_1 does not include a computation step of p_{i_0}. Thus, $\beta_0\rho_1$ cannot contain one session.

For the induction step, assume that the claim holds for $k-1$, i.e., $\beta_0 \ldots \beta_{k-2}\rho_{k-1}$ does not contain $k-1$ sessions, for $1 \leq k \leq s$. Hence, the kth session does not start within $\beta_0 \ldots \beta_{k-2}\rho_{k-1}$. Since neither σ_{k-1} nor ρ_k contains a computation step of $p_{i_{k-1}}$, $\sigma_{k-1}\rho_k$ does not contain a session. Thus, $\beta_0 \ldots \beta_{k-1}\rho_k$ does not contain k sessions.

To complete the proof, note that σ_{s-1} does not contain a session since, by construction, it does not contain a computation step of $p_{i_{s-1}}$. ■

Thus, there are strictly less than s sessions in β; however, in β' no process takes a non-idle step, so there cannot be an additional session in β'. A contradiction. ■

We remark that the general outline of this lower bound proof follows [4, 38]. However, while the proofs in [4, 38] use causality arguments to reorder the events in the execution, our proof presents an explicit reordering and retiming of the events. We do so because this provides a basis for the retiming arguments used to show the lower bound for the semi-synchronous model. Our improvement over [38] is achieved by carefully choosing only peripheral nodes in the construction of β.

The Semi-Synchronous Model

In Subsection 4.1.1, we have seen two algorithms that solve the s-session problem in the semi-synchronous model. The first of them, $\mathcal{A}_1^{ss}$, solves the s-session problem on G within time $1 + diam(G)D(s-2)$. Designed for the asynchronous model, $\mathcal{A}_1^{ss}$ has the interesting property that processes do not use any timing information. Loosely speaking, the lower bound proved in Theorem 4.5 says that if processes have no timing information, then $diam(G)d(s-1)$ is a lower bound for any asynchronous algorithm which solves the s-session problem on G.

Recall, also, that $\mathcal{A}_2^{ss}$ uses no communication, but relies only on timing information to achieve an upper bound of $1 + (\lfloor \frac{1}{c} \rfloor + 1)(s-2)$. We first show that this upper bound is close to optimal in the absence of communication:

Theorem 4.6 *Let G be any graph. There does not exist a semi-synchronous algorithm which solves the s-session problem on G within time strictly less than $\lfloor \frac{1}{c}(s-2) \rfloor$ and uses no communication.*

Proof: Assume, by way of contradiction, that there exists a semi-synchronous algorithm, $\mathcal{A}$, which solves the s-session problem on G within time strictly less than $\lfloor \frac{1}{c}(s-2) \rfloor$, and uses no communication. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

Let α be a slow, synchronous timed execution of $\mathcal{A}$. Assume, without loss of generality, that p_n is the last process to enter an idle state in α. Let $\alpha = \alpha_0 \alpha'$, where α_0 includes events at time 0, while α' is the remaining part of α. Let m be the number of non-idle steps taken by any process in α'. It must be that $m < \lfloor \frac{1}{c}(s-2) \rfloor$, since $\mathcal{A}$ solves the s-session problem within time $< \lfloor \frac{1}{c}(s-2) \rfloor$, and α is slow.

Now modify α' to get a new timed execution fragment β in which all processes except p_n, operate with fastest step time, i.e., c. This can be done since there are no receive events in α.

In β, all processes but p_n enter an idle state at time $cm < c\lfloor \frac{1}{c}(s-2) \rfloor \leq c\frac{1}{c}(s-2) = s-2$. Thus, in β, p_n performs strictly less than $s-2$ steps when all other processes are not in an idle state. Therefore, at most $s-2$ sessions can be completed in β; hence, at most $s-1$ sessions can be completed in $\alpha_0\beta$. A contradiction. ■

We show next that communication and timing information *cannot* be combined to get an upper bound that is significantly better than the upper bound achieved in Theorem 4.4. We prove:

Theorem 4.7 *Let G be any graph and assume that $d \geq \frac{d}{\min\{\lfloor 1/2c \rfloor, diam(G)d\}} + 2$. There does not exist a semi-synchronous algorithm which solves the s-session problem on G within time strictly less than $1 + \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-2)$.*

Proof: Assume, by way of contradiction, that there exists a semi-synchronous algorithm, $\mathcal{A}$, which solves the s-session problem on G within time strictly less than $1+\min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-2)$. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

The general structure of our lower bound proof closely follows that of Theorem 4.5, though there are several complications: First, the early events of the execution, happening at time < 1 and including processes' steps occurring at time 0, are handled separately (unlike the proof of Theorem 4.5. Second, the additional timing requirements placed in the semi-synchronous model require more careful arguing to show the correctness of the construction.

We start with a slow, synchronous timed execution of $\mathcal{A}$ and partition it into an execution fragment containing the events at time 0 and $s-2$ execution fragments each of which is completed within time $< \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}$. Since communication is slow, there is no communication between any pair of antipodal nodes during a fragment. Furthermore, since the execution is slow, a process takes, roughly, at most $\frac{1}{2c}$ steps, so it is possible to have these all steps occur at the same time another process takes only one step. By "retiming", we will perturb each fragment to get a new execution fragment in which there is a "fast" peripheral node which takes all of its steps before a "slow" antipodal node takes any of its steps. The part of the proof that shows that the "retimed" execution preserves the timing constraints of the semi-synchronous model requires substantially more careful arguments than the corresponding part in the proof of Theorem 4.5. In particular, we need to choose the execution fragments to take time $< \lfloor \frac{1}{2c} \rfloor$, so that it will be possible for a process not to have a computation step during a large part of the execution fragment. Our construction will have the "slow" node of each execution fragment be identical to the "fast" node of the next execution fragment. Arguing as in Theorem 4.5, this will guarantee that at most one session is contained in each execution fragment. Thus, the total number of sessions in the "retimed" execution is at most $s-2$, contradicting the correctness of $\mathcal{A}$.

We now present the details of the formal proof.

Denote $e = \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}$.

If $e \leq 1$, then the lower bound we are trying to prove is $\leq 1 + 1(s-2) = s-1$. Since s steps of each process are necessary if s sessions are to occur and they can occur 1 time unit apart, it follows that $s-1$ is a lower bound. Thus, we assume, without loss of generality, that $e > 1$. It follows that $c < \frac{1}{2}$. Note that, by assumption, $d \geq \frac{d}{e} + 2$, i.e., $d \geq \frac{2e}{e-1}$. Since $e > 1$, it follows that $d > 1$.

Let γ be a slow, synchronous timed execution of $\mathcal{A}$ with step time 1. Assume $\gamma = \beta_0 \alpha \alpha'$, where β_0 contains only events that occur at time < 1, and $\beta_0 \alpha$ is the shortest prefix of γ such that all processes are in an idle state in $last(\beta_0 \alpha)$ (α' is the remaining part of γ). Denote $T = t_{end}(\beta_0 \alpha)$. Since γ is slow and s steps of each process are necessary to guarantee

s sessions, $T \geq s-1$. Since $\mathcal{A}$ solves the s-session problem within time strictly less than $1+e(s-2)$, it follows that $T < 1+e(s-2)$. Note that, by construction, $t_{start}(\alpha) = 1$. Thus, $t_{end}(\alpha) - t_{start}(\alpha) = T-1 < e(s-2)$, and hence $\lceil \frac{T-1}{e} \rceil \leq (s-2)$. Denote $s' = \lceil \frac{T-1}{e} \rceil$; it follows that $s' \leq s-2$.

We write $\alpha = \alpha_1 \alpha_2 \ldots \alpha_{s'}$, where:

- For each k, $1 \leq k < s'$, α_k contains all events that occur at time t, where $1+(k-1)e \leq t < 1+ke$, and
- $\alpha_{s'}$ contains all events occurring at time t, where $1+(s'-1)e \leq t \leq T$.

That is, we partition α into execution fragments, each taking time $< e$.

Figure 4-1 depicts the timed execution $\beta_0 \alpha \alpha'$. Each horizontal line represents events happening at one process. We use the symbol $\bullet$ to mark non-idle process steps; similarly, we use the symbol $\times$ to mark idle process steps. Arrows show typical message delay times between pairs of processes; dashed vertical lines mark time points that are used in the proof.

We reorder and retime events in α to obtain a timed sequence β and reorder and retime events in α' to obtain a timed sequence α', such that $\beta_0 \beta \beta'$ is a timed execution of $\mathcal{A}$ that does not include s sessions.

We first show how to modify α to obtain an execution fragment $\beta = \beta_1 \beta_2 \ldots \beta_{s'}$ that includes at most $s' \leq s-2$ sessions. For some sequence $i_0, \ldots, i_{s'}$ of peripheral nodes, we construct from each execution fragment α_k an execution fragment $\beta_k = \rho_k \sigma_k$, such that:

(1) ρ_k contains no computation step of $p_{i_{k-1}}$, and

(2) σ_k contains no computation step of p_{i_k}.

For each k, $1 \leq k \leq s'$, we show how to construct β_k inductively. For the base case, let i_0 be an arbitrary peripheral node of G.

Assume we have picked $i_0, \ldots, i_{k-1}$ and constructed $\beta_1, \ldots, \beta_{k-1}$. Let i_k be some node that is antipodal to i_{k-1}, i.e., $dist(i_{k-1}, i_k) = diam(G)$; note that i_k is also peripheral. We now show how to construct β_k. For any node u, ρ_k includes all events at u that occur at

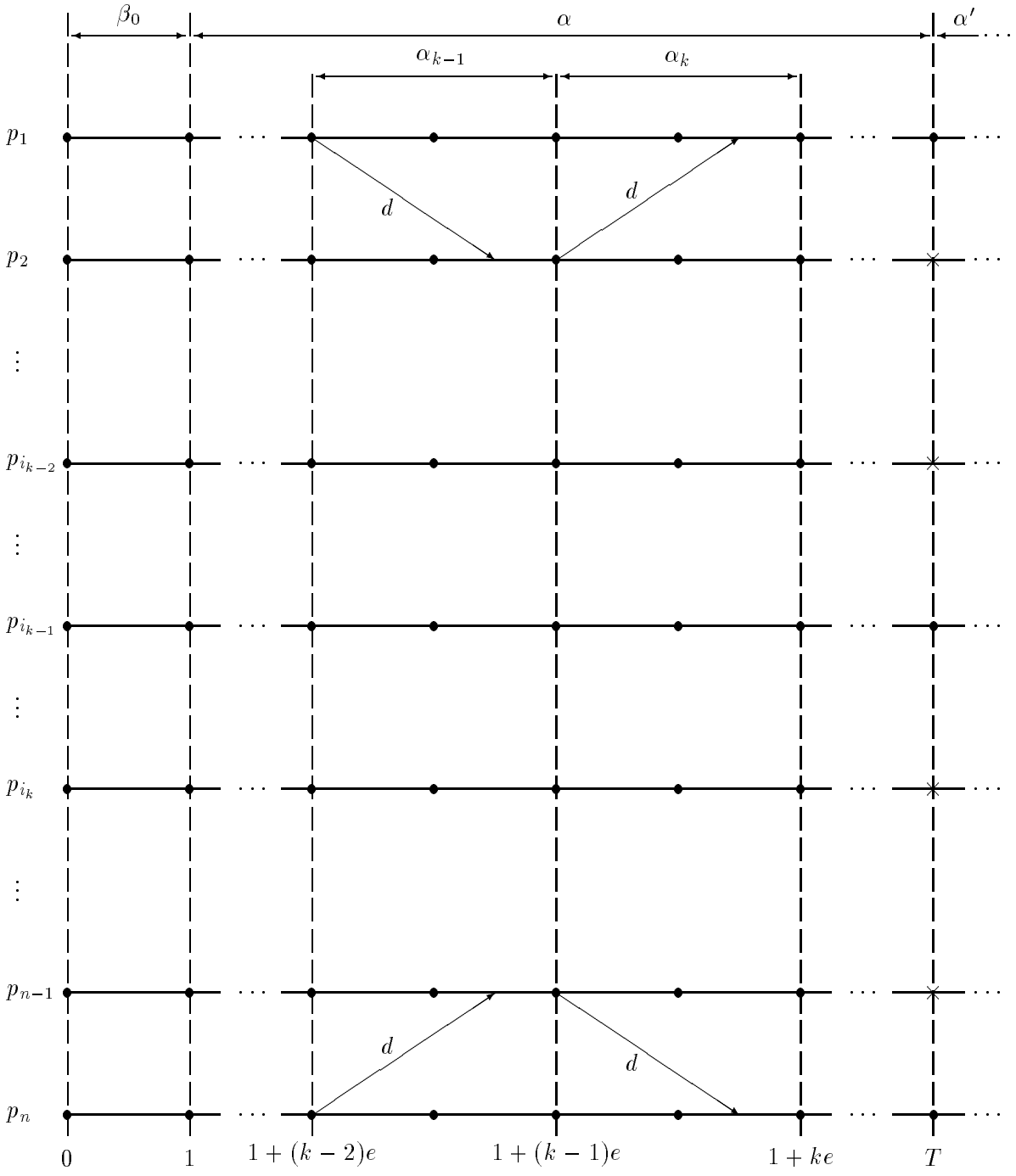

Figure 4-1: The timed execution $\beta_0\alpha\alpha'$

time $< 1 + (k-1)e + dist(u, i_{k-1})d$ in α_k; σ_k includes all events at u that occur at time $\geq 1 + (k-1)e + dist(u, i_{k-1})d$ in α_k. Events at each process occur in the same order as in α_k and all occur at step time of c, in both ρ_k and σ_k. In addition, ordering of events across different processes that occur at the same time in α_k is preserved within each of ρ_k and σ_k. Since

$$1+(k-1)e+dist(i_k, i_{k-1})d = 1+(k-1)e+diam(G)d \geq 1+(k-1)e+e = 1+ke > t_{end}(\alpha_k) ,$$

and all events at i_k occur at time $\leq t_{end}(\alpha_k)$ in α_k, this implies that all events at i_k will appear in ρ_k. On the other hand, since

$$1 + (k-1)e + dist(i_{k-1}, i_{k-1})d = 1 + (k-1)e \leq t_{start}(\alpha_k) ,$$

and all events at i_{k-1} in α_k occur at time $\geq t_{end}(\alpha_{k-1})$, all events at i_{k-1} in α_k will appear in σ_k. Thus, $\beta_k = \rho_k\sigma_k$ has properties (1) and (2) above.

To complete our construction, we assign times to events in β_k. Let $t_{start}(\rho_1) = c$. The first and last computation steps of i_k in ρ_k occur at times $t_{start}(\rho_k) = t_{end}(\sigma_{k-1}) + c$ and $t_{end}(\rho_k)$, respectively. Similarly, the first and last computation steps of i_{k-1} in σ_k occur at times $t_{start}(\sigma_k) = t_{end}(\rho_k)$ and $t_{end}(\sigma_k)$, respectively. Steps are taken c time units apart. For each process p_j, we schedule each computation step π_j of p_j in ρ_k to occur simultaneously with a computation step π_{i_k} of i_k such that π_j and π_{i_k} occurred at the same time in α_k. Similarly, for each process p_j, we schedule each computation step π_j of p_j in σ_k to occur simultaneously with a computation step $\pi_{i_{k-1}}$ of i_{k-1} such that π_j and $\pi_{i_{k-1}}$ occurred at the same time in α_k. Any message delivery event at a process will occur right after and at exactly the same time as the computation step of the process which immediately precedes the delivery event in α_k. We will shortly show that assigning times in this manner is consistent with the requirements for a timed execution.

We now modify α' to obtain β'. The first computation step of any process in β' will occur at time c after its last computation step in β and all later computation steps of it will

occur c time units apart in β'. Any delivery event at a process will occur at time d after the corresponding send event.

Figure 4-2 depicts the timed execution $\beta_0\beta\beta'$ using the same conventions as in Figure 4-1.

We remark that what allowed us to "separate" the steps at i_{k-1} from those at i_k in each of the execution fragments was the assumption that the length of each execution fragment is less than $diam(G)d$ which is the time needed for a communication between an antipodal pair of nodes to be established.

We first show that $\beta_0\beta\beta'$ is a timed execution of $\mathcal{A}$. By Lemma 4.10, since $s' \leq s-2$ and β_0 contains exactly one session, we derive a contradiction.

By the same arguments as in Lemma 4.7, we prove:

Lemma 4.11 *Each receive event is after the corresponding send event in $\beta_0\beta\beta'$.*

Before showing that the timing constraints are preserved in $\beta_0\beta\beta'$, we prove the following simple fact:

Claim 4.1 *(1) For any k, $1 \leq k \leq s'-1$, $t_{end}(\rho_{k+1}) - t_{end}(\rho_k) \leq 1-c$.*
(2) For any k, $1 \leq k \leq s'$, $t_{end}(\beta_k) - t_{end}(\beta_{k-1}) \leq 1-c$.

Proof: We first show that for any k, $1 \leq k \leq s'-1$, $t_{end}(\rho_{k+1}) - t_{end}(\beta_k) \leq \frac{1}{2}$, and for any k, $1 \leq k \leq s'$, $t_{end}(\beta_k) - t_{end}(\rho_k) \leq \frac{1}{2} - c$.

Fix some k, $1 \leq k \leq s'$. By construction,

$$t_{start}(\alpha_k) \geq 1 + (k-1)e,$$

while

$$t_{end}(\alpha_k) < 1 + ke.$$

Thus

$$t_{end}(\alpha_k) - t_{start}(\alpha_k) < 1 + ke - 1 - (k-1)e = e \leq \lfloor \frac{1}{2c} \rfloor.$$

Let m be the maximum number of steps over all processes that some process takes within α_k.

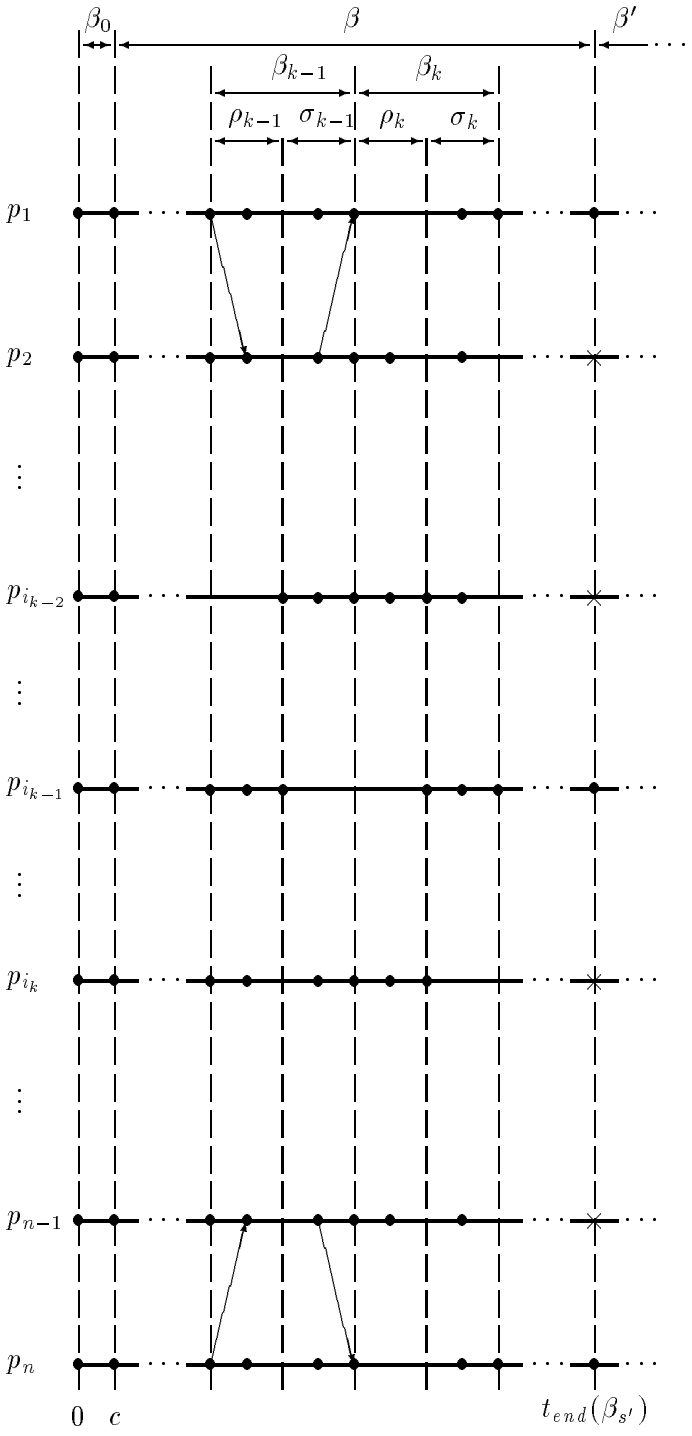

Figure 4-2: The timed execution $\beta_0\beta\beta'$

If both $t_{start}(\alpha_k)$ and $t_{end}(\alpha_k)$ are integral, $t_{end}(\alpha_k) - t_{start}(\alpha_k) \leq \lfloor \frac{1}{2c} \rfloor - 1$; then, since α is a slow execution,

$$m \leq t_{end}(\alpha_k) - t_{start}(\alpha_k) + 1 \leq \lfloor \frac{1}{2c} \rfloor \leq \frac{1}{2c}.$$

If at least one of $t_{start}(\alpha_k)$ and $t_{end}(\alpha_k)$ is not integral, then, since α is a slow execution,

$$m \leq \lceil t_{end}(\alpha_k) - t_{start}(\alpha_k) \rceil \leq \lceil \lfloor \frac{1}{2c} \rfloor \rceil = \lfloor \frac{1}{2c} \rfloor \leq \frac{1}{2c}.$$

Thus, in any case, $m \leq \frac{1}{2c}$.

Let n_k be the number of computation steps of process $p_{i_{k-1}}$ in α_k and n_{k+1} be the number of computation steps of process $p_{i_{k+1}}$ in α_{k+1}. (Recall that, by construction, in β_k, $p_{i_{k-1}}$ will have all of its steps in σ_k, while in β_{k+1}, $p_{i_{k+1}}$ will have all of its steps in ρ_{k+1}.) Thus,

$$t_{end}(\rho_{k+1}) - t_{end}(\beta_k) = n_{k+1}c \leq mc \leq \frac{1}{2c}c = \frac{1}{2}.$$

Also, since $p_{i_{k-1}}$ takes n_k steps in σ_k with the first occurring at time $t_{start}(\sigma_k) = t_{end}(\rho_k)$, and the last occurring at time $t_{end}(\sigma_k) = t_{end}(\beta_k)$, we have:

$$t_{end}(\beta_k) - t_{end}(\rho_k) = (n_k - 1)c \leq (m-1)c \leq (\frac{1}{2c} - 1)c = \frac{1}{2} - c \ .$$

Now, we have

$$t_{end}(\rho_{k+1}) - t_{end}(\rho_k) = t_{end}(\rho_{k+1}) - t_{end}(\beta_k) + t_{end}(\beta_k) - t_{end}(\rho_k) \leq \frac{1}{2} + \frac{1}{2} - c = 1 - c,$$

which proves (1). Also,

$$t_{end}(\beta_k) - t_{end}(\beta_{k-1}) = t_{end}(\beta_k) - t_{end}(\rho_k) + t_{end}(\rho_k) - t_{end}(\rho_{k-1}) \leq \frac{1}{2} - c + \frac{1}{2} = 1 - c,$$

which proves (2). ■

We next show:

Lemma 4.12 *Lower and upper bounds on step time are preserved in $\beta_0\beta\beta'$.*

Proof: By construction, no two computation steps are closer than c in $\beta_0\beta\beta'$; so, the lower bound on step time is preserved. Note also that the difference between consecutive computation steps of a process is maximized when the process is a peripheral node, i_k, for some k such that $1 \leq k \leq s' - 1$, that has no computation steps in either σ_k or ρ_{k+1}. By Claim 4.1(1), this is less than or equal to 1. ■

To complete the proof that $\beta_0\beta\beta'$ is a timed execution we show that:

Lemma 4.13 *The time between a send event and the corresponding receive event in $\beta_0\beta\beta'$ is at most d.*

Proof: Let π_1 be a computation step at node u_1 which occurs at time t_1 in $\beta_0\alpha\alpha'$, in which a message is sent; let π_2 be the corresponding delivery event at node u_2 occurring at time t_2 in $\beta_0\alpha\alpha'$. Assume π_1 and π_2 are scheduled to occur at times t_1' and t_2', respectively, in $\beta_0\beta\beta'$.

If π_2 occurs in α' then, by construction, $t_2' - t_1' = d$, in $\beta_0\beta\beta'$. So assume π_1 and π_2 occur in $\beta_0\alpha$. We first consider the case where both π_1 and π_2 occur in α. Assume π_1 appears in α_{k_1} and π_2 appears in α_{k_2}, where $1 \leq k_1 \leq k_2 \leq s'$. Clearly, in β, π_1 appears in β_{k_1} and π_2 appears in β_{k_2}. Note that, by construction, $d = t_2 - t_1 > (k_2 - 1)e - k_1 e = (k_2 - k_1 - 1)e$, i.e., $k_2 - k_1 - 1 < \frac{d}{e}$. It follows that:

$$\begin{aligned}
t_2' - t_1' &\leq t_{end}(\beta_{k_2}) - t_{start}(\beta_{k_1}) \\
&\leq t_{end}(\beta_{k_2}) - t_{end}(\beta_{k_1 - 1}) \\
&= \sum_{j=k_1}^{k_2} t_{end}(\beta_j) - t_{end}(\beta_{j-1}) \\
&\leq (k_2 - k_1 + 1) \qquad \text{(by Claim 4.1(2))} \\
&\leq \frac{d}{b} + 2 \leq d \qquad \text{(by assumption)} ,
\end{aligned}$$

as needed.

Finally, we consider the case where π_1 occurs in β_0, i.e., $t_1 = 0$. Asume that π_2 appears in α_{k_2}. By construction, $d = t_2 - t_1 = t_2 > (k_2 - 1)e$, i.e., $k_2 - 1 < \frac{d}{e}$. Reasoning as in the

previous case, we get:

$$t'_2 - t'_1 \leq t_{end}(\beta_{k_2}) = \sum_{j=1}^{k_2}(t_{end}(\beta_j) - t_{end}(\beta_{j-1})) \leq k_2 < \frac{d}{e} + 1 < d,$$

as needed. ■

Lemma 4.10 implies that β contains at most $s' \leq s-2$ sessions; also, β_0 contains exactly one session. Therefore, there are at most $s-1$ sessions in $\beta_0\beta$. Since in β' no process takes a non-idle step, there is no additional session in β'. Thus, there are at most $s-1$ sessions in $\beta_0\beta\beta'$. A contradiction. ■

4.1.3 The Non-Uniform Case

In this Subsection, we consider the case problem where delays on communication links are not uniform. Specifically, we assume that for each $(i,j) \in E$, the delay of any message along (i,j) is in the interval $[0, d(i,j)]$ for some $d(i,j)$ such that $0 \leq d(i,j) < \infty$.

We first develop some notation that is necessary for stating our results. Let p be a path from node v_0 to node v_k in G, i.e., a sequence of nodes $v_0, v_1, \ldots, v_k$ such that for each i, $1 \leq i < k$, $(v_i, v_{i+1}) \in E$. Denote by $l(p)$ the *length*, k, of p. We define the *delay on* p, $d(p)$, to be the sum of the delay on its edges, i.e.,

$$d(p) = \sum_{i=0}^{k-1} d(v_i, v_{i+1}) \ .$$

We define the *delay from node* i *to node* j, $del(i,j)$, to be the minimum of $d(p)$ over all paths p between i and j. Naturally, the *delay on* G, $\hat{d}(G)$, is the maximum of the delay from one node of G to another, over all pairs of nodes in G, i.e.,

$$\hat{d}(G) = \max_{i,j \in V} del(i,j) \ .$$

Intuitively, $\hat{d}(G)$ is the worst-case delay that a message between a pair of nodes may incur along a "shortest-delay" path from i to j in G. However, because of local processing time a

message that is sent along a path p can effectively incur a delay of up to $d(p)+l(p)$, since each process in the path can incur a local processing delay of at most 1 and postpone forwarding the message until its next computation step. Thus, we define the *effective delay on* G to be

$$\hat{D}(G) = \max_{i,j \in V} \min_{p \text{ a path from } i \text{ to } j} (d(p) + l(p)) .$$

Clearly, in the uniform case, when all delays are equal to d, $\hat{d}(G)$ and $\hat{D}(G)$ are equal to $diam(G)d$ and $diam(G)D$, respectively. Also,

$$\hat{d}(G) \leq \hat{D}(G) \leq \hat{d}(G) + diam(G) .$$

Denote $d_{\min} = \min_{(i,j) \in E} d(i,j)$.

To obtain bounds for the non-uniform case, we observe that $\hat{D}(G)$ naturally replaces $diam(G)D$ in the upper bounds for the uniform analogs, while $\hat{d}(G)$ naturally replaces $diam(G)d$ in the corresponding lower bounds.

Also, for the lower bounds for the semi-synchronous model, let $e' = \min\{\lfloor \frac{1}{2c} \rfloor, \hat{d}(G)\}$ and assume, as in the proof of Theorem 4.7, that $e' > 1$. Note that if the condition $d \geq \frac{d}{e'} + 2$ holds with $d_{\min}$ for d, then it also holds with $d(i,j)$, for any $(i,j) \in E$, for d. This implies that the non-uniform analog of the condition $d \geq \frac{d}{e} + 2$ is $d_{\min} \geq \frac{d_{\min}}{e'} + 2$. We next state our upper and lower bound results for the non-uniform case. Their proofs exactly follow those of their uniform analogs and are omitted.

Theorem 4.8 *Let G be any graph. There exists an asynchronous algorithm $\mathcal{A}_w^{as}$ which solves the s-session problem on G within time $\hat{D}(G)(s-1)$.*

Theorem 4.9 *Let G be any graph. There exists a semi-synchronous algorithm $\mathcal{A}_w^{ss}$ which solves the s-session problem on G within time $1 + \min\{\lfloor \frac{1}{c} \rfloor + 1, \hat{D}(G)\}(s-2)$.*

Theorem 4.10 *Let G be any graph. There does not exist an asynchronous algorithm which solves the s-session problem on G within time strictly less than $\hat{d}(G)(s-1)$.*

Theorem 4.11 *Let G be any graph and assume that $d_{\min} \geq \frac{d_{\min}}{\min\{\lfloor 1/2c \rfloor, \hat{d}(G)\}} + 2$. There does not exist a semi-synchronous algorithm which solves the s-session problem on G within time strictly less than $1 + \min\{\lfloor \frac{1}{2c} \rfloor, \hat{d}(G)\}(s-2)$.*

4.1.4 The Uninitialized Case

In this Subsection, we consider the *uninitialized* case of the s-session problem. The states of a process are now partitioned into three disjoint sets: *quiescent*, *non-idle* and *idle*. At time 0 all processes except one, the *initiator*, are in a *quiescent* state. A process may enter a non-idle state only upon receiving a message. In all other aspects, the behavior of the system in the uninitialized case is as described in Section 2.2. The time associated with an execution of an algorithm is the time until the last process enters an idle state.

Upper Bounds

We present algorithms that solve the s-session problem in the uninitialized case.

We start with the asynchronous model and present an algorithm which follows the style of $\mathcal{A}^{as}$. As soon as a process receives the first message, it communicates in order to learn about completion of a session before advancing to the next session. As in $\mathcal{A}^{as}$, each process maintains as part of its state a variable that gives its current session number; upon hearing that every other process has reached its current session, it increments its session number by one and notifies all other processes. The process enters an idle state when its session number is set to s. We prove:

Theorem 4.12 *Let G be any graph. There exists an uninitialized, asynchronous algorithm, $\mathcal{B}^{as}$, which solves the s-session problem on G within time $diam(G)Ds$.*

Proof: We describe an uninitialized, asynchronous algorithm, $\mathcal{B}^{as}$, which solves the s-session problem on G within time $diam(G)Ds$; that is, in any execution of $\mathcal{B}^{as}$ there are at least s sessions and all processes enter an idle state no later than time $diam(G)Ds$. The algorithm is described here informally; this description can be easily translated into a state transition function.

For each $i \in [n]$, each non-quiescent state of p_i consists of the following components: *buffer* — a buffer, an unordered set of elements of $\mathcal{M}$, initially $\emptyset$; *session* — a nonnegative integer, initially 0. The message alphabet, $\mathcal{M}$, consists of the pairs (i,k), where $i \in [n]$ and $0 \leq k \leq s-1$.

The algorithm is as follows: The initiator, p_{i_0} sends a message $(i_0,0)$ to all processes (including itself). For each $i \in [n]$, when p_i receives the message $(i_0,0)$, it enters a non-idle state, sets $session_i$ to 1, and broadcasts $(i,1)$. It then continues as in $\mathcal{A}^{as}$. If for all $j \in [n]$, $(j, session_i) \in buffer_i$, p_i increments $session_i$ by 1. If $session_i = s$, p_i enters an idle state and remains in this state forever. Otherwise, p_i broadcasts $(i, session_i)$. As in previous algorithms, we assume that messages from a process are flooded on a shortest-paths tree rooted at this process. We say that p_i is in its kth session if $session_i = k$, and we interpret the message (i,k) as "process i executed a step in the kth session".

We start by showing that in any execution of $\mathcal{B}^{as}$ there are at least s sessions. Fix an arbitrary timed execution α of $\mathcal{B}^{as}$ and let p_{i_0} be the initiator process for that execution. For any k, $1 \leq k \leq s$, define α_k to be the longest prefix of α that does not include a configuration in which for some $i \in [n]$, $session_i \geq k$. Note that for each k, $1 \leq k \leq s-1$, α_k is a prefix of α_{k+1}. For each k, $1 \leq k \leq s-1$, let β_k be such that $\alpha_{k+1} = \alpha_k \beta_k$; let β_s be such that $\alpha = \alpha_s \beta_s$. As in Lemma 4.1, we can prove:

Lemma 4.14 *For each k, $1 \leq k \leq s-1$, there is a session in β_k.*

In addition, there is a session in β_s, since, for every $i \in [n]$, a computation step is included in β_s at which p_i sets $session_i$ to s. (Note that, by the definition of α_s, such a step cannot be included in α_s.) This implies that there are at least s sessions in α. Since α was chosen arbitrarily, this implies the correctness of $\mathcal{B}^{as}$.

We now analyze the time complexity of $\mathcal{B}^{as}$. For each k, $1 \leq k \leq s$, we define:

$$T_k = \max_{i \in V} \{t : \ p_i \text{ sets } session_i \text{ to } k \text{ at time } t \text{ in } \alpha\}$$

Since every process p_i is guaranteed to receives a $(i_0,0)$ message by time $\leq diam(G)D$, it

follows that $T_1 \leq diam(G)D$. As in Lemma 4.2, we can prove:

Lemma 4.15 *For each k, $1 \leq k < s$, $T_{k+1} \leq T_k + diam(G)D$.*

Thus, it follows that $T_s \leq diam(G)Ds$. Hence, every process enters an idle state after setting *session* to s, no later than time $diam(G)Ds$. Thus, $\mathcal{B}^{as}$ solves the s-session problem on G within time $diam(G)Ds$. ■

We continue with algorithms for the semi-synchronous model. Clearly, the algorithm in Theorem 4.12 works also in the semi-synchronous model. We remark, however, that unlike the case were processes start simultaneously, in the uninitialized case, timing information cannot be exploited to improve the time complexity of $diam(G)Ds$ achieved by $\mathcal{B}^{as}$. We next present a simple, semi-synchronous algorithm that relies on the timing information that is available in the semi-synchronous model.

Theorem 4.13 *Let G be any graph. There exists an uninitialized, semi-synchronous algorithm $\mathcal{B}^{ss}$ which solves the s-session problem on G within time $3diam(G)D + (\lfloor \frac{1}{c} \rfloor + 1)(s-1)$.*

Proof: We describe an uninitialized semi-synchronous algorithm, $\mathcal{B}^{ss}$, which solves the s-session problem on G within time $3diam(G)D + (\lfloor \frac{1}{c} \rfloor + 1)(s-1)$. For each $i \in [n]$, the state of process p_i consists of a *counter*, an integer, initially -1. The message alphabet is $\{wake, ack, ack^2\}$. Let p_{i_0} be the initiator. At time 0, p_{i_0} sends *wake* to all processes and remains in a non-idle state. Once it receives *ack* from all processes, it sends ack^2 to all processes and sets $counter_{i_0}$ to 0. At each of its next computation steps, it increments $counter_{i_0}$ by 1. All other processes, upon receiving *wake*, enter a non-idle state and send *ack* to p_{i_0}. When a non-initiator process, p_i, receives an ack^2 message it sets $counter_i$ to 0. At each of its next computation steps, it increments $counter_i$ by 1. Each process p_i (including the initiator) enters an idle state when $counter_i = (\lfloor \frac{1}{c} \rfloor + 1)(s-1)$.

We start by showing that in any execution of $\mathcal{B}^{ss}$ there are at least s sessions. Fix an arbitrary timed execution γ of $\mathcal{B}^{ss}$. Let $\gamma = \alpha_0\alpha_1\alpha\alpha'$, where: α_0 is the shortest prefix of γ such that every process is in a non-quiescent state in $last(\alpha_0)$; $\alpha_0\alpha_1$ is the shortest prefix of γ

such that $counter_{i_0} = 0$ in $last(\alpha_0\alpha_1)$; $\alpha_0\alpha_1\alpha$ is the longest prefix of γ such that all processes are in a a non-idle state in $last(\alpha_0\alpha_1\alpha)$ (α' is the remaining part of γ).

We remark that, by construction, all processes are in a non-idle state in every configuration of α. Partition α into execution fragments as $\alpha = \alpha_1\alpha_2\ldots\alpha_{s-1}$, such that, for each k, $1 \leq k \leq s-1$, $\alpha_1\ldots\alpha_k$ is the shortest prefix of α that includes a configuration for which, for some $i \in [n]$, $counter_i = k(\lfloor\frac{1}{c}\rfloor + 1)$. Clearly:

Lemma 4.16 *There is a session in α_0.*

As in Lemma 4.6, we have:

Lemma 4.17 *For each k, $1 \leq k \leq s-1$, there is a session in α_k.*

Thus, there are at least $s-1$ sessions in α and, therefore, at least s sessions in γ. Since γ was chosen arbitrarily, this implies the correctness of $\mathcal{B}^{ss}$. We now analyze the time complexity of $\mathcal{B}^{ss}$.

Let $\gamma = \beta_1\beta\beta'$, where β_1 is the shortest prefix of γ such that, for some $i \in [n]$, $counter_i = 0$ in $last(\beta_1)$, and $\beta_1\beta$ is the longest prefix of γ such that some process is not in an idle state in $last(\beta_1\beta)$ (β' is the remaining part of γ). It suffices to show that

$$t_{end}(\beta) \leq 3diam(G)D + (\lfloor\frac{1}{c}\rfloor + 1)(s-1) \ .$$

Clearly, $t_{end}(\beta_1) \leq 3diam(G)D$. Also,

$$t_{end}(\beta) - t_{end}(\beta_1) \leq (\lfloor\frac{1}{c}\rfloor + 1)(s-1) \ .$$

This implies that,

$$t_{end}(\beta) \leq t_{end}(\beta_1) + (\lfloor\frac{1}{c}\rfloor + 1)(s-1) \leq 3diam(G)D + (\lfloor\frac{1}{c}\rfloor + 1)(s-1) \ ,$$

as needed. Thus, every process enters an idle state no later than time $3diam(G)D + (\lfloor\frac{1}{c}\rfloor + 1)(s-1)$ in γ. Since γ was chosen arbitrarily, this implies that $\mathcal{B}^{ss}$ solves the s-session problem

on G within time $3diam(G)D + (\lfloor \frac{1}{c} \rfloor + 1)(s-1)$. ■

As in the initialized case, when d and $diam(G)$ are known, it is possible to calculate in advance which of the the algorithms of Theorem 4.12 and Theorem 4.13 is faster and run it. Moreover, even if d and $diam(G)$ are not known, it is possible to run these algorithms "side by side" and halt when the first of them does.

Roughly speaking, each process starts executing the algorithm of Theorem 4.13. After hearing from the initiator, p_{i_0}, for the first time, it starts executing also the algorithm of Theorem 4.12. Specifically, it identifies the *wake* message with the $(i_0, 0)$ message.

Omitting some details and noting that:

$$\min\{3diam(G)D+(\lfloor\frac{1}{c}\rfloor+1)(s-1), diam(G)Ds\} \leq 3diam(G)D+\min\{(\lfloor\frac{1}{c}\rfloor+1), diam(G)D\}(s-1)\,,$$

we can show:

Theorem 4.14 *Let G be any graph. There exists an uninitialized, semi-synchronous algorithm, $\mathcal{B}^{ss}$, which solves the s-session problem on G within time $3diam(G)D + \min\{\lfloor \frac{1}{c} \rfloor + 1, diam(G)D\}(s-1)$.*

Lower Bounds

We start by showing that for the asynchronous model, the uninitialized algorithm presented in Theorem 4.12 is almost optimal. We will use an infinite timed execution in which processes take steps with step time close to 1 and all messages are delivered after exactly d delay; we will call it a *slow* timed execution. The proof of the following theorem is based on delaying information propagation and then perturbing an execution to obtain one which does not include s sessions. The general structure of our proof closely follows that of Theorem 4.5.

Theorem 4.15 *Let G be any graph. There does not exist an uninitialized, asynchronous algorithm which solves the s-session problem on G within time strictly less than $diam(G)ds$.*

Proof: Assume, by way of contradiction, that there exists an uninitialized, asynchronous algorithm, $\mathcal{A}$, which solves the s-session problem on G within time strictly less than $diam(G)ds$. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

Pick some ε such that $0 < \varepsilon \leq (diam(G)d(s-1)+1)^{-1}$. Fix a peripheral process p_1 and let γ be a slow timed execution of $\mathcal{A}$ with step time $1-\varepsilon$, in which p_1 is the initiator. Let p_2 be a process which is antipodal to p_1. Let $\gamma = \alpha_0\alpha\alpha'$, where: α_0 is the shortest prefix of γ such that p_2 is in a non-quiescent state in $last(\alpha_0)$; $\alpha_0\alpha$ is the shortest prefix of γ such that all processes are in an idle state in $last(\alpha_0\alpha)$ (α' is the remaining part of γ). We perturb $\alpha_0\alpha$ to obtain another timed execution $\beta\alpha'$ that does not include s sessions.

We first show how to modify $\alpha_0\alpha$ to obtain β. Since γ is slow, $t_{end}(\alpha_0) \geq diam(G)d$, while, by assumption, $t_{end}(\alpha_0\alpha) < diam(G)ds$. This implies that:

$$t_{end}(\alpha_0\alpha) - t_{end}(\alpha_0) < diam(G)ds - diam(G)d = diam(G)d(s-1)$$

Write $\alpha = \alpha_1\alpha_2\ldots\alpha_{s-1}$, where for each k, $1 \leq k \leq s-1$, $t_{end}(\alpha_k) - t_{end}(\alpha_{k-1}) < diam(G)d(1-\varepsilon) < diam(G)d$. Thus, $\alpha_0\alpha = \alpha_0\alpha_1\ldots\alpha_{s-1} = \alpha'_1\alpha'_2\ldots\alpha'_{s-1}$, where: $\alpha'_1 = \alpha_0\alpha_1$ and for each k, $2 \leq k \leq s-1$, $\alpha'_k = \alpha_k$. Also denote $\alpha'_0 = \alpha_0$. For some sequence $i_0, \ldots, i_{s-1}$ of peripheral nodes, we construct from each execution fragment α'_k, an execution fragment $\beta_k = \rho_k\sigma_k$ such that:

(1) ρ_k contains no computation step of $p_{i_{k-1}}$, and

(2) σ_k contains no computation step of p_{i_k}.

We now show, for each k, $1 \leq k \leq s-1$, how to construct β_k, by induction on k. For the base case, let $i_0 = p_2$.

Assume we have picked $i_0, \ldots, i_{k-1}$ and constructed $\beta_1, \ldots, \beta_{k-1}$. Let i_k be some node that is antipodal to i_{k-1}, i.e., $dist(i_{k-1}, i_k) = diam(G)$; note that i_k is also peripheral. We now show how to construct β_k.

For any node u, ρ_k includes all events at u that occur at time $< t_{end}(\alpha_{k-1}) + dist(u, i_{k-1})d$ in α'_k; σ_k includes all events at u that occur at time $\geq t_{end}(\alpha_{k-1}) + dist(u, i_{k-1})$ in α'_k. Events

at each process occur in the same order as in α'_k, and all occur at time 0 in both ρ_k and σ_k. In addition, ordering of events across different processes that occur at the same time in α'_k is preserved within each of ρ_k and σ_k. Since

$$t_{end}(\alpha'_{k-1}) + dist(i_k, i_{k-1})d = t_{end}(\alpha'_{k-1}) + diam(G)d > t_{end}(\alpha'_k) \ ,$$

and all events at i_k occur at time $\leq t_{end}(\alpha'_k)$ in α'_k, this implies that all events at i_k will appear in ρ_k. On the other hand, since

$$t_{end}(\alpha'_{k-1}) + dist(i_{k-1}, i_{k-1})d = t_{end}(\alpha_{k-1}),$$

and all events at i_{k-1} in α'_k occur at time $\geq t_{end}(\alpha_{k-1})$, all events at i_{k-1} will appear in σ_k. Thus, $\beta_k = \rho_k\sigma_k$ has properties (1) and (2) above.

Let $\beta = \beta_1 \ldots \beta_{s-1}$.

By construction, events at each process p_i, $i \in [n]$, occur in the same order in β as in $\beta_0\alpha$. Thus, p_i undergoes the same state changes in β as in $\beta_0\alpha$, and, therefore, $state_i(last(\beta)) = state_i(last(\alpha_0\alpha))$.

We now modify α' to obtain β'. The first computation step of any process in β' will occur at time 1 after its last computation step in β and all later computation steps of it will occur 1 time unit apart in β'. Any message delivery event at a process will occur at time d after the corresponding message send event.

We next establish that $\beta\beta'$ is a timed execution of $\mathcal{A}$. In a way similar to the proof of Lemma 4.7 we can show:

Lemma 4.18 *Each receive event is after the corresponding send event in $\beta\beta'$.*

All events in β occur at time 0; in β', steps occur with step time 1 and all messages are delivered after time exactly d. Since there are no lower bounds on either process step time or message delivery time in the asynchronous model, we have:

Lemma 4.19 *Lower and upper bounds on step time are preserved in $\beta\beta'$.*

Lemma 4.20 *Lower and upper bounds on message delay time are preserved in $\beta\beta'$.*

As in Lemma 4.10, we can show:

Lemma 4.21 *There are at most $s-1$ sessions in β.*

Thus, there are strictly less than s sessions in β; however, in β' no process takes a non-idle step, so there cannot be an additional session in β'. A contradiction. ■

We continue by showing that, as in the initialized case, there are limitations on combining communication with the timing information available in the semi-synchronous model.

We will use an infinite timed execution in which processes that are in a non-quiescent state take steps synchronously, in round-robin order and with step time equal to 1, and all messages are delivered after exactly d delay, except for the initial messages from the initiator to all other processes which are delivered after $\lfloor d \rfloor$ delay. We will call it a *slow, synchronous* timed execution. We show:

Theorem 4.16 *Let G be any graph and assume that $d \geq \frac{d}{\min\{2,\lfloor 1/2c \rfloor, diam(G)d\}} + 2$. There does not exist an uninitialized, semi-synchronous algorithm which solves the s-session problem on G within time strictly less than $diam(G)\lfloor d \rfloor + \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-1)$.*

Proof: Assume, by way of contradiction, that there exists an uninitialized, semi-synchronous algorithm, $\mathcal{A}$, which solves the s-session problem on G within time strictly less than $diam(G)\lfloor d \rfloor + \min\{\lfloor \frac{1}{2c} \rfloor, diam(G)d\}(s-1)$. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

The general structure of our lower bound proof closely follows that of Theorem 4.7. We next present the details of the formal proof.

Denote $e = \min\{2, \lfloor \frac{1}{2c} \rfloor, diam(G)d\}$.

If $e \leq 1$, then the lower bound to prove is $\leq diam(G)\lfloor d \rfloor + s - 1$. In this case, the proof is trivial. Consider a slow, synchronous timed execution where the initiator, p_1, is peripheral. Since all initial messages take time $\lfloor d \rfloor$ to be delivered, there is some peripheral process, p_2,

that enters a non-idle state no earlier than time $diam(G)\lfloor d \rfloor$. Since p_2 takes its steps at a rate of 1, and s steps of p_2 are necessary if s sessions are to occur, it follows that these steps will need $s-1$ time to be completed. Thus, p_2 enters an idle state no earlier than time $diam(G)\lfloor d \rfloor + s - 1$. Thus, we assume, without loss of generality, that $e > 1$. As in Theorem 4.7, it follows that $c < \frac{1}{2}$.

Fix a peripheral process p_1 and let γ be a slow, synchronous timed execution of $\mathcal{A}$ in which p_1 is the initiator. Let $\gamma = \alpha_0\alpha\alpha'$, where: α_0 is the longest prefix of γ such that there is some peripheral process, p_2, which is in a quiescent state in $last(\alpha_0)$; $\alpha_0\alpha$ is the shortest prefix of γ such that all processes are in an idle state in $last(\alpha_0\alpha)$ (α' is the remaining part of γ).

Since γ is slow, $t_{start}(\alpha) \geq diam(G)\lfloor d \rfloor$. Denote $T = t_{end}(\alpha_0\alpha)$. Since $\mathcal{A}$ solves the s-session problem on G within time strictly less than $diam(G)\lfloor d \rfloor + e(s-1)$, it follows that $T < diam(G)\lfloor d \rfloor + e(s-1)$. Thus, $t_{end}(\alpha) - t_{start}(\alpha) \leq T - diam(G)\lfloor d \rfloor < e(s-1)$; hence, $\lceil \frac{T - diam(G)\lfloor d \rfloor}{e} \rceil \leq s-1$. Denote $s' = \lceil \frac{T - diam(G)\lfloor d \rfloor}{e} \rceil$; it follows that $s' \leq s-1$.

We write $\alpha = \alpha_1\alpha_2 \ldots \alpha_{s'}$, where:

- For each k, $1 \leq k \leq s'$, α_k contains all events that occur at time t, where $t_{start}(\alpha) + (k-1)e \leq t < t_{start}(\alpha) + ke$, and
- $\alpha_{s'}$ contains all events occurring at time t, where $t_{start}(\alpha) + (s'-1)e \leq t \leq T$.

That is, we partition α into execution fragments each taking time $< e$.

We reorder and retime events in α_0, α and α' to obtain timed sequences β_0, β and β', respectively, such that $\beta_0\beta\beta'$ is a timed execution of $\mathcal{A}$ that does not include s sessions.

We first show how to modify α_0 to obtain β_0. Any event at a process which occurs at time t in α_0 will occur at time tc in β_0. This implies that events at each process occur in the same order as in α_0 with step time c. In addition, ordering of events across different processes that occur at the same time in α_0 is preserved within β_0.

We now show how to modify α to obtain an execution fragment $\beta = \beta_1\beta_2 \ldots \beta_{s'}$ that includes at most $s' \leq s-1$ sessions. For some sequence $i_0, \ldots, i_{s'}$ of peripheral nodes, we construct from each execution fragment α_k an execution fragment $\beta_k = \rho_k\sigma_k$, such that:

(1) ρ_k contains no computation step of $p_{i_{k-1}}$, and

(2) σ_k contains no computation step of p_{i_k}.

For each k, $1 \leq k \leq s'$, we show how to construct β_k inductively. For the base case, let i_0 be p_2.

Assume we have picked $i_0, \ldots, i_{k-1}$ and constructed $\beta_1, \ldots, \beta_{k-1}$. Let i_k be some node that is antipodal to i_{k-1}, i.e., $dist(i_{k-1}, i_k) = diam(G)$; note that i_k is also peripheral. We now show how to construct β_k. For any node u, ρ_k includes all events at u that occur at time $< t_{start}(\alpha) + (k-1)e + dist(u, i_{k-1})d$ in α_k; σ_k includes all events at u that occur at time $\geq t_{start}(\alpha) + (k-1)e + dist(u, i_{k-1})d$ in α_k. Events at each process occur in the same order as in α_k and all occur with step time c, in both ρ_k and σ_k. In addition, ordering of events across different processes that occur at the same time in α_k is preserved within each of ρ_k and σ_k. Since

$$t_{start}(\alpha)+(k-1)e+dist(i_k, i_{k-1}) = t_{start}(\alpha)+(k-1)e+diam(G)d \geq t_{start}(\alpha)+ke > t_{end}(\alpha_k) ,$$

and all events at i_k occur at time $\leq t_{end}(\alpha_k)$ in α_k, this implies that all events at i_k will appear in ρ_k. On the other hand, since

$$t_{start}(\alpha_k) + (k-1)e + dist(i_{k-1}, i_{k-1})d = t_{start}(\alpha) + (k-1)e \leq t_{start}(\alpha_k) ,$$

and all events at i_{k-1} in α_k occur at time $\geq t_{end}(\alpha_{k-1})$, this implies that all events at i_{k-1} will appear in σ_k. Thus, $\beta_k = \rho_k\sigma_k$ has properties (1) and (2) above.

To complete our construction, we assign times to events in β_k. Let $t_{start}(\rho_1) = t_{end}(\beta_0)+c$. The first and last computation steps of i_k in ρ_k occur at times $t_{start}(\rho_k) = t_{end}(\alpha_{k-1})+c$ and $t_{end}(\rho_k)$, respectively. Similarly, the first and last computation steps of i_{k-1} in σ_k occur at times $t_{start}(\sigma_k) = t_{end}(\rho_k)$ and $t_{end}(\sigma_k)$, respectively. Steps are taken c time units apart. For each process p_j, we schedule each computation step π_j of p_j in ρ_k to occur simultaneously with a computation step, π_{i_k}, of i_k which is such that π_j and π_{i_k} occurred at the same time in α_k. Similarly, for each process p_j, we schedule each computation step π_j of p_j in σ_k to

occur simultaneously with a computation step, $\pi_{i_{k-1}}$, of i_{k-1} which is such that π_j and $\pi_{i_{k-1}}$ occurred at the same time in α_k. Any message delivery event at a process will occur right after and at exactly the same time as the computation step of the process which immediately precedes it in α_k. We shall shortly show that assigning times in this manner is consistent with the requirements for a timed execution.

We finally modify α' to obtain β'. The first computation step of any process in β' will occur at time c after its last computation step in β and all later computation steps of it will occur c time units apart in β'. Any message delivery event at a process will occur at time d after the corresponding message send event.

We first show that $\beta_0\beta\beta'$ is a timed execution of $\mathcal{A}$. By Lemma 4.10, since $s' \leq s-1$ and β_0 contains no session, we derive a contradiction.

By the same arguments as in Lemma 4.7, we prove:

Lemma 4.22 *Each receive event is after the corresponding send event in $\beta_0\beta\beta'$.*

As in Lemma 4.12, we can show:

Lemma 4.23 *Lower and upper bounds on step time are preserved in $\beta_0\beta\beta'$.*

We finally show:

Lemma 4.24 *The time between a message send event and the corresponding message delivery event in $\beta_0\beta\beta'$ is at most d.*

Proof: Let π_1 be a computation step at node u_1, occurring at time t_1 in $\alpha_0\alpha\alpha'$, in which a message is sent; let π_2 be the corresponding delivery event at node u_2, occurring at time t_2 in $\alpha_0\alpha\alpha'$. Assume π_1 and π_2 are scheduled to occur at times t_1' and t_2', respectively, in $\beta_0\beta\beta'$.

If π_2 occurs in α', then, by construction, $t_2' - t_1' = d$ in $\beta_0\beta\beta'$. So assume π_1 and π_2 occur in $\alpha_0\alpha$. Since we constructed β exactly as in Theorem 4.7, the case where both π_1 and π_2 occur in α is handled as in Lemma 4.13. Thus, it only remains to consider the case where π_1 occurs in α_0. Assume that π_2 occurs in α_k for some k such that $1 \leq k \leq s'$. Note that, by

construction, $t_2 - t_{end}(\alpha_0) > (k-1)e$, i.e., $k < \frac{t_2 - t_{end}(\alpha_0)}{e} + 1$. It follows that:

$$\begin{aligned}
t'_2 - t'_1 &= t'_2 - t_{end}(\beta_0) + t_{end}(\beta_0) - t'_1 \\
&= t'_2 - t_{end}(\beta_0) + (t_{end}(\alpha_0) - t_1)c && \text{(by construction)} \\
&\leq t_{end}(\beta_k) - t_{end}(\beta_0) + \frac{t_{end}(\alpha_0) - t_1}{2} && \text{(since } c < \tfrac{1}{2}\text{)} \\
&< k + \frac{t_{end}(\alpha_0) - t_1}{2} && \text{(by Claim 4.1 (2))} \\
&< \frac{t_2 - t_{end}(\alpha_0)}{e} + 1 + \frac{t_{end}(\alpha_0) - t_1}{2} \\
&\leq \frac{t_2 - t_1}{e} + 1 = \frac{d}{e} + 1 \leq d && \text{(by assumption)} ,
\end{aligned}$$

as needed. ■

By construction, there is no session in β_0. Also, Lemma 4.10 implies that there are at most $s' < s$ sessions in $\beta_0\beta$. Since in β' no process takes a non-idle step, there is no additional session in β'. Thus, there are at most $s-1$ sessions in $\beta_0\beta\beta'$. A contradiction. ■

4.2 Shared Memory

This Section is organized as follows. Subsection 4.2.1 surveys some simple bounds, deducible from either previous work (cf. [4]) or results in Section 4.1, that also hold for our asynchronous and semi-synchronous shared-memory models. Subsection 4.2.2 includes our main lower bound.

4.2.1 Simple Bounds

For the asynchronous model, where there is no lower bound on processes' step time, the lower bound proof in [4], relying on the ability to schedule many steps by the same process at the same time, still works to yield a lower bound of $\Omega(s \log_b n)$*. Also, the "tree network"

*Note that in [4], the asynchronous model is defined in a slightly different way than ours, more specifically by having all infinite admissible computations be allowable, and puts no restriction on the number of steps a process takes at a time.

algorithm sketched in [4] (Section 4) still works in our model. The "tree network" algorithm relies entirely on explicit communication between processes to ensure that the needed steps have occurred and does not use any timing information. Roughly speaking, this algorithm consists of building up a "tree"out of b-atomic registers whose leaves are the n processes. Neglecting roundoffs, this network has depth $\log_b n$. Processes communicate through this network in order to learn about completion of a session before advancing to the next session. Thus, the necessary communication for one session can be accomplished in time $O(\log_b n)$ and the total time for all processes to enter an idle state after performing s sessions is $O(s \log_b n)$ in both the asynchronous and the semi-synchronous models.

On the other hand, the algorithm of Theorem 4.3 still works for the semi-synchronous shared memory model; this algorithm does not use any communication[†], but relies entirely on and exploits the timing information available in the semi-synchronous model to obtain a bound which is sometimes better than the bound of the "tree network" algorithm. Roughly speaking, in this algorithm each process takes about $s\lfloor\frac{1}{c}\rfloor$ computation steps before entering an idle state.

It is possible to run the two previous algorithms "side by side," halting when the first of them does, and get a bound of $O(\min\{\frac{1}{c}, \log_b n\}s)$ for the s-session problem in the semi-synchronous shared-memory model. Note that, by an appropriate choice of the various parameters, this upper bound and the $\Theta(s \log_b n)$ tight bound for the asynchronous model together imply a time separation between semi-synchronous and asynchronous shared-memory models.

Note also that the lower bound of $\lfloor\frac{1}{c}(s-2)\rfloor$, shown in Theorem 4.6, also holds for the semi-synchronous shared memory model in the absence of communication.

4.2.2 Main Lower Bound

We show that, for the semi-synchronous model, communication and timing information *cannot* be combined to yield an upper bound that is significantly better than the $O(\min\{\frac{1}{c}, \log_b n\}s)$ upper bound discussed in Subsection 4.2.1.

[†]This means that no state transition can result in an operation on a shared variable.

In our lower bound proof, we use an infinite timed execution in which processes take steps in round-robin order, starting with p_1, with step time equal to 1. It is a called a *slow, synchronous* timed execution. We have:

Theorem 4.17 *There does not exist a semi-synchronous algorithm which solves the s-session problem within time strictly less than* $1 + \min\{\lfloor \frac{1}{2c} \rfloor, \lfloor \log_b(n-1) - 1 \rfloor\}(s-2)$.

Proof: Assume, by way of contradiction, that there exists a semi-synchronous algorithm, $\mathcal{A}$, which solves the s-session problem within time strictly less than $1 + \min\{\lfloor \frac{1}{2c} \rfloor, \lfloor \log_k(n-1) - 1 \rfloor\}(s-2)$. We construct a timed execution of $\mathcal{A}$ which does not include s sessions.

We start with a slow, synchronous timed execution of $\mathcal{A}$ and partition it into an execution fragment containing the events at time 0 and at most $s-2$ other execution fragments each of which is completed within time $< \min\{\lfloor \frac{1}{2c} \rfloor, \lfloor \log_b(n-1) - 1 \rfloor\}$. We use causality and fan-out arguments to argue that there is no communication through shared memory between a certain pair of processes within each fragment. Furthermore, since the execution is slow, a process takes, roughly, at most $\frac{1}{2c}$ steps in each fragment, so it is possible to have all these steps occur while another process takes only one step. By "retiming", we will perturb each fragment to get a new one in which there is a "fast" process which takes all of its steps before a "slow" process takes any of its steps. The part of the proof that shows that the "retimed" execution preserves the timing constraints of the semi-synchronous model requires to choose the execution fragments to take time $< \lfloor \frac{1}{2c} \rfloor$, so that it will be possible for a process to not take a computation step during a large part of the execution. Our construction will have the "fast" process of each execution fragment be identical to the "slow" process of the next execution fragment. This will guarantee that at most one session is completed in each execution fragment. Thus, the total number of sessions in the "retimed" execution is at most $s-1$, contradicting the correctness of $\mathcal{A}$.

We now present the details of the formal proof.

Denote $e = \min\{\lfloor \frac{1}{2c} \rfloor, \lfloor \log_b(n-1) - 1 \rfloor\}$.

If $e \leq 1$, then the lower bound we are trying to prove is $\leq 1 + 1(s-2) = s-1$. Since s steps of each process are necessary if s sessions are to occur and they can occur 1 time unit

apart, it follows that $s-1$ is a lower bound. Thus, we assume, without loss of generality, that $e > 1$.

Let γ be a slow, synchronous timed execution of $\mathcal{A}$. Assume $\gamma = \alpha_0\alpha\alpha'$, where α_0 contains only events that occur at time < 1, $\alpha_0\alpha$ is the shortest prefix of γ such that all processes are in an idle state in $last(\alpha_0\alpha)$, and α' is the remaining part of γ. Denote $T = t_{end}(\alpha_0\alpha)$. Since γ is slow and s steps of each process are necessary to guarantee s sessions, $T \geq s-1$. Since $\mathcal{A}$ solves the s-session problem within time strictly less than $1+e(s-2)$, it follows that $T < 1+e(s-2)$. Note that, by construction, $t_{start}(\alpha) = 1$. Thus, $t_{end}(\alpha) - t_{start}(\alpha) = T-1 < e(s-2)$. Denote $s' = \lceil \frac{T-1}{e} \rceil$; it follows that $s' \leq s-2$.

We write $\alpha = \alpha_1\alpha_2 \ldots \alpha_{s'}$, where:

- For each k, $1 \leq k < s'$, α_k contains all events that occur at time t, where $1+(k-1)e \leq t < 1+ke$, and
- $\alpha_{s'}$ contains all events occurring at time t, where $1+(s'-1)e \leq t \leq T$.

That is, we partition α into execution fragments, each taking time $< e$.

Figure 4-3 depicts the timed execution $\alpha_0\alpha\alpha'$. Each horizontal line represents events happening at one process. We use the symbol $\bullet$ to mark non-idle process steps; similarly, we use the symbol $\times$ to mark idle process steps. Dashed vertical lines mark time points that are used in the proof.

We reorder and retime events in α to obtain a timed sequence β and reorder and retime events in α' to obtain a timed sequence β', such that $\alpha_0\beta\beta'$ is a timed execution of $\mathcal{A}$ that does not include s sessions.

In our construction, we will use a partial order $\leq_\alpha$, representing "dependency", on the computation steps that processes take in α. We start by defining $\leq_\alpha$. For every pair of steps π_1, π_2 in α, we let $\pi_1 \leq_\alpha \pi_2$ if $\pi_1 = \pi_2$ or if π_1 precedes π_2 in α and either π_1 and π_2 are steps taken by the same process or by different processes, but on the same shared variable. Close $\leq_\alpha$ under transitivity. $\leq_\alpha$ is a partial order, and every total order of computation steps in α consistent with $\leq_\alpha$ represents a computation which leaves the system in the same

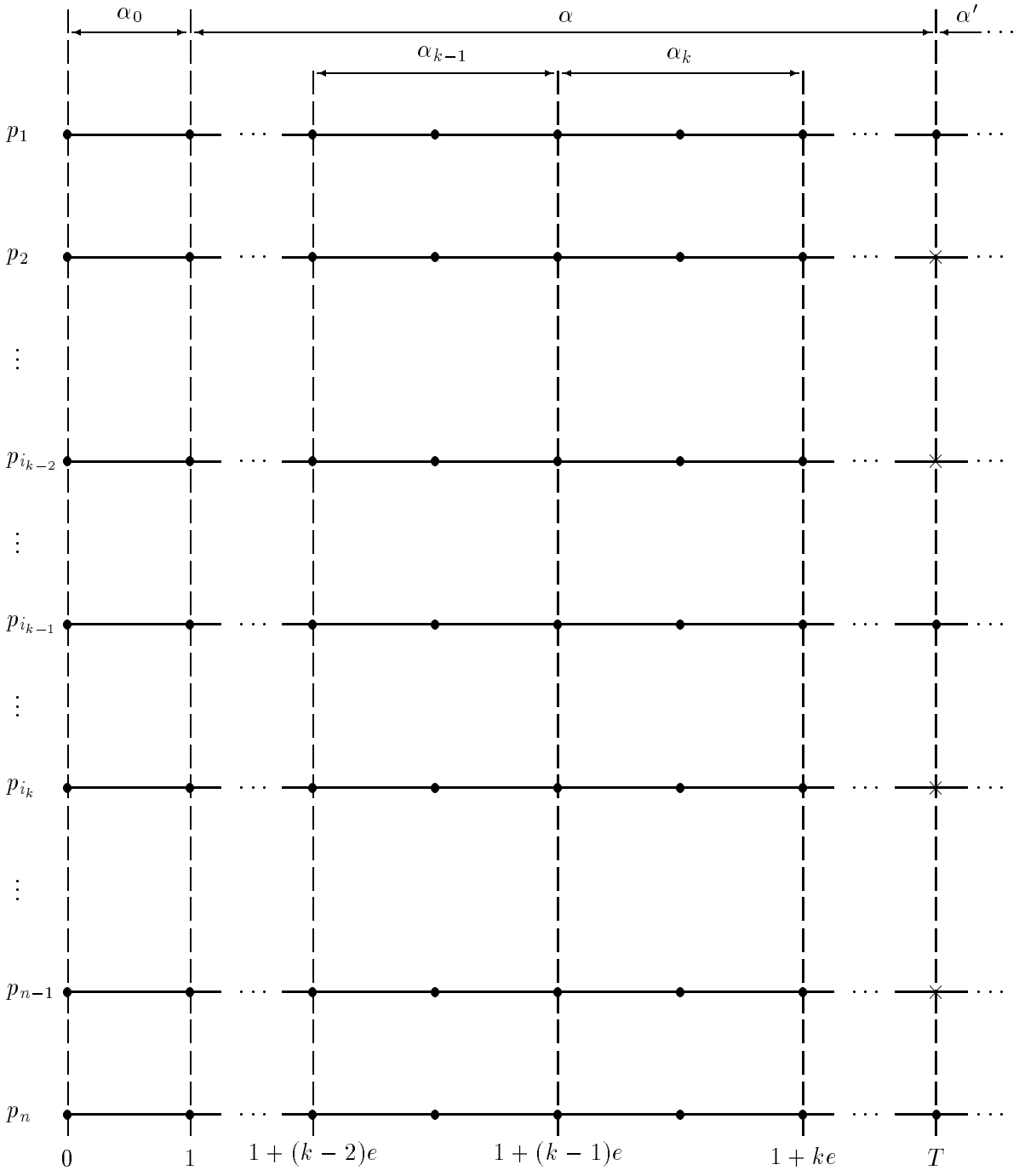

Figure 4-3: The timed execution $\alpha_0\alpha\alpha'$

configuration as α. (Clearly, α itself provides such a total order.)

We first show how to modify α to obtain an execution fragment $\beta = \beta_1\beta_2\ldots\beta_{s'}$ that includes at most $s' \leq s-2$ sessions. For some sequence $p_{i_0},\ldots,p_{i_{s'}}$ of processes, we construct from each execution fragment α_k an execution fragment $\beta_k = \rho_k\sigma_k$, such that:

(1) ρ_k contains no computation step of $p_{i_{k-1}}$, and

(2) σ_k contains no computation step of p_{i_k}.

In this construction, p_{i_k} is the "fast" process which takes all its steps in ρ_k, before the "slow" process $p_{i_{k-1}}$ takes any of its steps. (All the steps of $p_{i_{k-1}}$ are in σ_k.)

For each k, $1 \leq k \leq s'$, we show how to construct β_k inductively. For the base case, let p_{i_0} be an arbitrary process.

Assume we have picked $p_{i_0},\ldots,p_{i_{k-1}}$ and constructed $\beta_1,\ldots,\beta_{k-1}$. We first show that there exists some process such that a communication between it and $p_{i_{k-1}}$ cannot be established in α_k.

Lemma 4.25 *Let π_1 be the first step of $p_{i_{k-1}}$ in α_k. There is some process of which there is no computation step σ in α_k such that $\pi_1 \leq_\alpha \sigma$.*

Proof: Clearly, it suffices to show that the number of steps τ in α_k such that $\pi \leq_\alpha \tau$, where π is any step of $p_{i_{k-1}}$ in α_k, is at most $n-1$. We proceed to count the number of such steps.

By construction,

$$t_{end}(\alpha_k) - t_{start}(\alpha_k) < 1 + ke - 1 - (k-1)e = e.$$

Let m be the maximum number of steps over all processes that some process takes within α_k. Since α is a slow execution,

$$m \leq \lceil t_{end}(\alpha_k) - t_{start}(\alpha_k)\rceil \leq \lceil e\rceil \leq \lceil\lfloor \log_b(n-1) - 1\rfloor\rceil = \lfloor\log_b(n-1) - 1\rfloor.$$

Clearly, the number of steps τ taken by any process in α_k such that $\pi_i \leq_\alpha \tau$, where π_i is the ith step of $p_{i_{k-1}}$ in α is at most k^{m-i+1}. Thus, the number of steps τ in α_k such that $\pi \leq_\alpha \tau$,

where π is any step of $\pi_{i_{k-1}}$ is at most:

$$\sum_{i=1}^{m} b^{m-i+1} = b \sum_{i=0}^{m-1} b^i = b\frac{b^m - 1}{b-1} \leq b^{m+1} \leq b^{\lfloor \log_b(n-1)-1 \rfloor + 1} \leq b^{\log_b(n-1)} = n-1.$$

The claim follows. ■

Fix p_{i_k} to be any process such that a communication between $p_{i_{k-1}}$ and p_{i_k} is not established in α_k. We now show how to construct β_k. For any process u, σ_k includes all steps τ of u in α_k such that $\pi \leq_\alpha \tau$, where π is any step of $\pi_{i_{k-1}}$ in α_k; ρ_k includes all remaining steps of u in α_k. Steps at each process occur in the same order as in α_k and all occur at step time of c, in both ρ_k and σ_k. In addition, ordering of steps by different processes that occur at the same time in α_k is preserved within each of ρ_k and σ_k. By Lemma 4.25, there is no step σ of p_{i_k} in α_k such that, for some step π of $p_{i_{k-1}}$ in α_k, $\pi \leq_\alpha \sigma$. This implies that all steps of p_{i_k} in α_k will appear in ρ_k. On the other hand, since $\pi \leq_\alpha \pi$ for any step π of $p_{i_{k-1}}$ in α_k, all steps of $p_{i_{k-1}}$ in α_k will appear in σ_k. Thus, $\beta_k = \rho_k\sigma_k$ has properties (1) and (2) above.

To complete our construction, we assign times to steps in β_k. Let $t_{start}(\rho_1) = c$. The first and last steps of p_{i_k} in ρ_k occur at times $t_{start}(\rho_k) = t_{end}(\sigma_{k-1}) + c$ and $t_{end}(\rho_k)$, respectively. Similarly, the first and last steps of $p_{i_{k-1}}$ in σ_k occur at times $t_{start}(\sigma_k) = t_{end}(\rho_k)$ and $t_{end}(\sigma_k)$, respectively. Steps are taken c time units apart. For each process p_j, we schedule each step π_j of p_j in ρ_k to occur simultaneously with a step, π_{i_k}, of p_{i_k} which is such that π_j and π_{i_k} occurred at the same time in α_k. Similarly, for each process p_j, we schedule each step π_j of p_j in σ_k to occur simultaneously with a step, $\pi_{i_{k-1}}$, of $p_{i_{k-1}}$ which is such that π_j and $\pi_{i_{k-1}}$ occurred at the same time in α_k. We will shortly show that assigning times in this manner is consistent with the requirements for a timed execution.

We now modify α' to obtain β'. The first computation step of any process in β' will occur at time c after its last computation step in β and all later computation steps of it will occur c time units apart in β'.

Figure 4-4 depicts the timed execution $\alpha_0\beta\beta'$ using the same conventions as in Figure 4-3.

We remark that what allowed us to "separate" the steps of $p_{i_{k-1}}$ from those of p_{i_k} in each

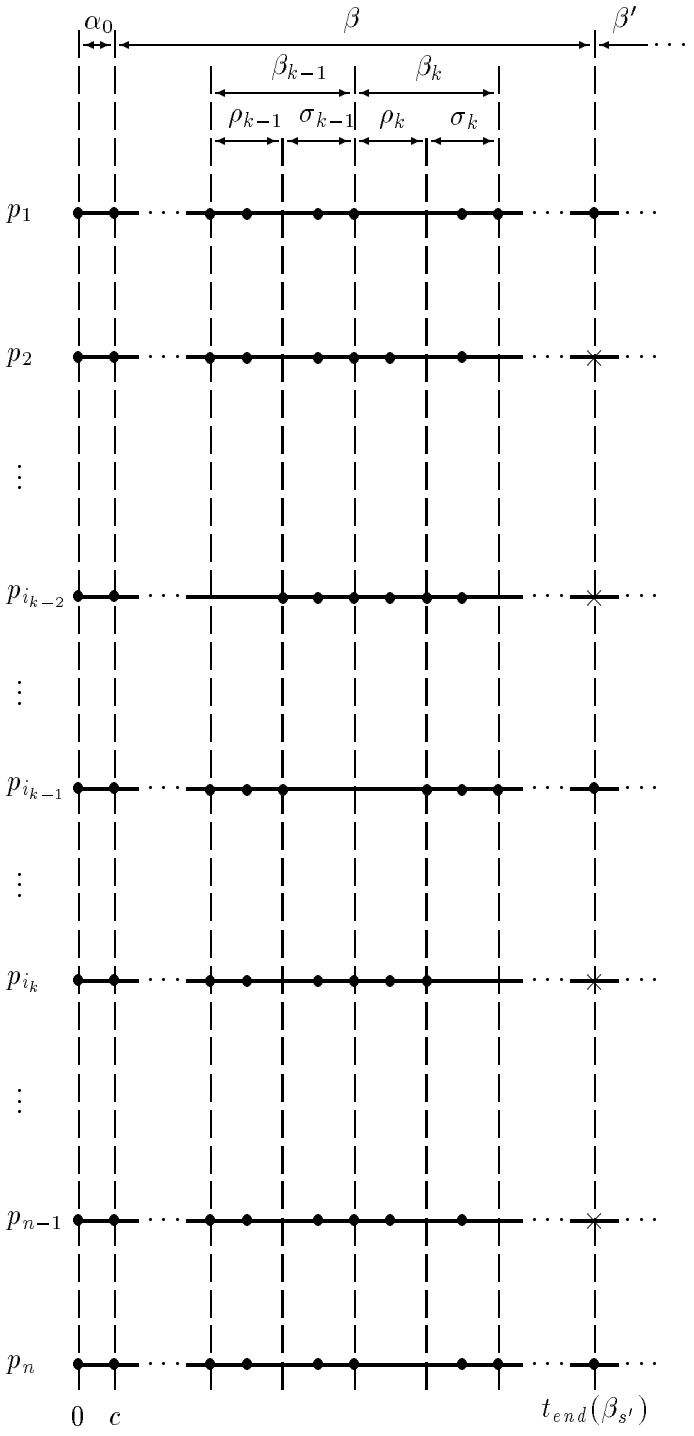

Figure 4-4: The timed execution $\alpha_0\beta\beta'$

of the execution fragments was the assumption that the length of each execution fragment is less than $\lfloor \log_b(n-1) - 1 \rfloor$ which, due to the communication limitations of the model, is not enough to guarantee that a process can "affect" at least one step of every other process.

We next establish that $\alpha_0\beta\beta'$ is a timed execution of $\mathcal{A}$. We start by showing:

Lemma 4.26 *Ordering of computation steps operating on the same shared variable is preserved in $\alpha_0\beta\beta'$.*

Proof: Let π_1 and π_2 be computation steps operating on the same shared variable in α_k, such that $\pi_1 \leq_\alpha \pi_2$. The only non-trivial case is when π_1 and π_2 occur in the same α_k, for some k, $1 \leq k \leq s'$. We show that the ordering of π_1 and π_2 is the same in β_k as in α_k.

The only case of interest is when π_1 occurs in σ_k, while π_2 occurs in ρ_k. By construction, there is some step π_1' of $p_{i_{k-1}}$ in α_k such that $\pi_1' \leq_\alpha \pi_1$, while there is no step π_2' of $p_{i_{k-1}}$ in α_k such that $\pi_2' \leq_\alpha \pi_2$. But, from $\pi_1' \leq_\alpha \pi_1$ and $\pi_1 \leq_\alpha \pi_2$, it follows, by transitivity, that $\pi_1' \leq_\alpha \pi_2$. A contradiction. ■

In a way similar to Claim 4.1(1), we can prove the following simple fact:

Claim 4.2 *For any k, $1 \leq k \leq s'-1$, $t_{end}(\rho_{k+1}) - t_{end}(\rho_k) \leq 1 - c$.*

We next show:

Lemma 4.27 *Lower and upper bounds on step time are preserved in $\alpha_0\beta\beta'$.*

Proof: By construction, no two computation steps are closer than c in $\alpha_0\beta\beta'$; so, the lower bound on step time is preserved. Note also that the time difference between consecutive computation steps of a process is maximized when the process is some p_{i_k}, for some k such that $1 \leq k \leq s'-1$, that has no computation steps in either σ_k or ρ_{k+1}. By Claim 4.2, this time difference is less than or equal to 1. ■

This completes the proof that $\alpha_0\beta\beta'$ is a timed execution. Lemma 4.10 implies that β contains at most $s' \leq s-2$ sessions; also, α_0 contains exactly one session. Therefore, there are at most $s-1$ sessions in $\alpha_0\beta$. Since in β' no process takes a non-idle step, there is no additional session in β'. Thus, there are strictly less than s sessions in $\alpha_0\beta\beta'$. A contradiction. ■

We remark that the general structure of our proof closely follows [4] and that of the proof of the lower bound for semi-synchronous networks in Theorem 4.7. Specifically, we used causality arguments as in [4] to reorder the steps in the execution, and presented an explicit retiming of them as in the proof of Theorem 4.7.

Chapter 5

Semi-Synchrony versus Real Time

In this Chapter, we show how the timing assumptions made for the semi-synchronous model may be exploited to enable processes acquire close estimates of real time.

This Chapter is organized as follows. Sections 5.1 and 5.2 include our lower and upper bounds, respectively, on the achievable precision.

5.1 A Lower Bound

We show:

Theorem 5.1 *No clock synchronization algorithm can synchronize P within precision Δ for any $\Delta < \lfloor \frac{d-2}{2c} \rfloor$.*

Proof: Fix any tick synchronization algorithm $\mathcal{A}$ which synchronizes P within precision Δ. We will show that $\Delta \geq \lfloor \frac{d-2}{2c} \rfloor$.

Consider a fast, synchronous infinite timed execution α of $\mathcal{A}$ in which all processes take steps at a rate of c in a rouund-robin order, starting with p_1, and start spontaneously and simultaneously executing their local protocols, and all messages are delivered after exactly $\frac{d}{2}$ delay. As a result of our assumptions, α will also be "symmetric" in the sense that all processes will undergo the same state changes in a synchronous fashion, enter a synchronized state simultaneously and make a common estimate of real time. Let $\alpha = \beta\beta'$, where β is

the longest prefix of α such that some process is not in a synchronized state in $last(\beta)$, and β' is the remaining part of α. We reorder and retime events in α to construct an infinite timed execution α_1 of $\mathcal{A}$ which is equivalent to α in the sense that for each process p_i, events at p_i occur in the same order in α_1 as in α. This will guarantee that α and α_1 will be indistinguishable to the processes and, therefore, each process will undergo the same state changes and, therefore, make the same estimate of real time upon entering a synchronized state, as a result of a run of $\mathcal{A}$, for each of these executions.

To facilitate the description of the technical details of our construction, we introduce the following definition: for each process p_i, we denote by T_i the time at which p_i enters a synchronized state in α and we say that p_i gets a-retarded in α if events at p_i are retimed so that the following two conditions are met:

1. Ordering of events at p_i which occur in β is maintained.

2. All computation steps of process p_i that occur at time $\geq T_i - a$ in α are rescheduled to occur at a rate of 1, with the first of them occurring at the same time as in α.

3. Each message delivery event at process p_i which occurs at time $\geq T_i - a$ in α is rescheduled to occur at exactly the same time as the computation step of p_i that immediately precedes it.

Our construction for obtaining α_1 consists merely of a-retarding p_n in α, where $a = \frac{d}{2}\frac{c}{1-c}$.

We next eastablish that α_1 is a timed execution of $\mathcal{A}$. We start by showing:

Lemma 5.1 *Each receiving event is after the corresponding sending event in* α_1.

Proof: Consider the message sending event π_1 at node u_1 which occurs at time t_1 in α and let π_2 be the corresponding message delivery event at node u_2 which occurs at time t_2 in α. In α_1, let π_1 occur at time t_1' and π_2 occur at time t_2'. We show, by case analysis, that the ordering of π_1 and π_2 is the same in α_1 as in α.

1. None of u_1 and u_2 is a-retarded in α_1: Obvious.

2. $u_1 = p_n$: In this case $t_2' = t_2$; thus, we only need to consider the subcase where $t_1 \geq T_n - a$, since, otherwise, $t_1' = t_1$, and the claim becomes trivial. We can also assume that $t_2 \leq T_2$, since, otherwise, π_2 occurs in β' and can be rescheduled to occur at a later time in β' without affecting the estimate of real time made by u_2 at T_2. Note that since:

$$t_2' - t_1' = t_2 - t_1' = t_2 - t_1 - (t_1' - t_1) = \frac{d}{2} - (t_1' - t_1),$$

to show that $t_2' \geq t_1'$, it suffices to show that $t_1' - t_1 \leq \frac{d}{2}$. By our construction, the first computation step of u_1 that occurs at time $\geq T_n - a$ in α will occur at time $\lceil T_1 - a \rceil$ in α_1. Since there are at most $\lceil \frac{t_1 - (T_n - a)}{c} \rceil$ computation steps of u_1 that occur in α at time t such that: $T_n - a \leq t \leq t_1$ and u_1 is a-retarded in α_1, we will have:

$$\begin{aligned}
t_1' - t_1 &= \lceil T_n - a \rceil + (\lceil \frac{t_1 - (T_n - a)}{c} \rceil - 1) - t_1 \\
&\leq T_n - a + 1 + \frac{t_1 - (T_n - a)}{c} + 1 - 1 - (T_n - a + t_1 - (T_n - a)) \\
&= 1 + \frac{t_1 - (T_n - a)}{c} - (t_1 - (T_n - a)) \\
&= 1 + (t_1 - (T_n - a))\frac{1 - c}{c} \\
&\leq 1 + (T_n - (T_n - a))\frac{1 - c}{c} \\
&\quad \text{(since } t_1 \leq T_n\text{)} \\
&= 1 + a\frac{1 - c}{c} \\
&= 1 + (\frac{d}{2} - 1)\frac{c}{1 - c}\frac{1 - c}{c} \\
&= \frac{d}{2},
\end{aligned}$$

as needed.

3. $u_2 = p_n$: We only need to consider the subcase where $t_2 \geq T_2 - a$, since, otherwise, $t_2' = t_2$, and the claim is trivial. It is obvious, however, that, by construction, we will then have: $t_2' > t_2 \geq t_1 = t_1'$, as needed.

■

We next show:

Lemma 5.2 *The time between a message-send event and the corresponding message-delivery event in α_1 is at most d.*

Proof: Consider the message sending event π_1 at node u_1 which occurs at time t_1 in α and let π_2 be the corresponding message delivery event at node u_2 which occurs at time t_2 in α. In α_1, let π_1 occur at time t_1' and π_2 occur at time t_2'. We show, by case analysis, that: $t_2' - t_1' \leq d$.

1. None of u_1 and u_2 is a-retarded in α_1: Obvious.

2. $u_1 = p_n$: In this case, $t_2' = t_2$, while, by construction, $t_1' \geq t_1$. Thus: $t_2' - t_1' \leq t_2 - t_1 = \frac{d}{2} < d$.

3. $u_2 = p_n$: In this case, $t_1' = t_1$. As in Lemma 5.1, we can show that: $t_2' - t_2 \leq \frac{d}{2}$. Thus:

$$t_2' - t_1' = t_2' - t_1 = t_2' - t_2 + t_2 - t_1 \leq \frac{d}{2} + \frac{d}{2} = d,$$

   as needed.

■

We can now show:

Lemma 5.3 *α_1 is a timed execution of $\mathcal{A}$.*

Proof: Obvious from Lemma 5.1, Lemma 5.2 and the fact that by construction, any two consecutive computation steps of any process are either c or 1 apart in α_1. ■

Thus, we have shown so far that α_1 is a timed execution of $\mathcal{A}$. Moreover, p_n makes precisely the same estimate about real time at the moment it is entering a synchronized state in each of α and α_1. Let T_n be the (real) time at which p_n is entering a synchronized state in α_1. Let $L_n(T_n+)$ and $L_n(T_n'+)$ be the estimates of real time that p_n is making at T_n and T_n', in

α and α_1, respectively. By our construction, $L_n(T_n+) = L_n(T'_n+)$. By symmetry, T_{n-1}, the time at which p_{n-1} is entering a synchronized state in α (in α_1, as well, since there are no changes for the times at which events at p_{n-1} occur in α_1) must equal T_n; by symmetry, also, $L_{n-1}(T_{n-1}+) = L_n(T_n+)$. We show a simple fact:

Claim 5.1 *The number of ticks that process p_{n-1} receives between T_n and T'_n is at least $\lfloor \frac{d-2}{2c} \rfloor$.*

Proof: Since process p_n takes its computation steps at a rate of c in α, it will have $\lceil \frac{a}{c} \rceil$ computation steps that occur in α at time t such that $T_n - a \leq t \leq T_n$. In α_1, these computation steps will be taken at a rate of 1 and require time $\geq \lceil \frac{a}{c} \rceil - 1$ to be completed; since they are completed at time T'_n, this implies that:

$$T'_n - (T_n - a) \geq \lceil \frac{a}{c} \rceil - 1$$

Therefore:

$$T'_n - T_n = (T'_n - (T_n - a)) - (T_n - (T_n - a)) \geq \lceil \frac{a}{c} \rceil - a$$

In view of the above, the number of ticks, m, that p_{n-1} receives between T_n and T'_n must satisfy:

$$\begin{aligned} m &\geq \lfloor \frac{T'_n - T_n}{c} \rfloor \\ &\geq \lfloor \frac{\lfloor \frac{a}{c} \rfloor - a}{c} \rfloor \\ &= \lfloor \frac{\lfloor \frac{d}{2} \frac{1}{1-c} \rfloor - \frac{d}{2} \frac{c}{1-c}}{c} \rfloor \\ &\geq \lfloor \frac{\frac{d}{2} \frac{1}{1-c} - 1 - \frac{d}{2} \frac{c}{1-c}}{c} \rfloor \\ &\geq \lfloor \frac{d-2}{2c} \rfloor \end{aligned}$$

■

We are now ready to present the main argument of our proof. We have:

$$\begin{aligned}
\Delta + L_n(T_n+) &= \Delta + L_n(T'_n+) \\
&\quad \text{(since } \alpha \text{ and } \alpha_1 \text{ are equivalent)} \\
&\geq L_{n-1}(T_{n'}) \\
&\quad \text{(since } \mathcal{A} \text{ synchronizes } P \text{ within precision } \Delta) \\
&= L_{n-1}(T_{n-1}+) + m \\
&= L_n(T_n+) + m \\
&\quad \text{(since } \alpha \text{ is symmetric with respect to } p_n \text{ and } p_{n-1}) \\
&\geq L_n(T_n+) + \lfloor \frac{d-2}{2c} \rfloor \\
&\quad \text{(by Claim 5.1)}
\end{aligned}$$

Therefore:

$$\Delta \geq \lfloor \frac{d-2}{2c} \rfloor \ ,$$

as needed. ■

5.2 An Upper Bound

We show:

Theorem 5.2 *There exists an algorithm which synchronizes P within precision $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$.*

Proof: We describe an algorithm which is very similar to the one in [36]. Each process p can start executing the synchronization algorithm either spontaneously or upon receiving a message from a process that has already done so. As soon as it starts, it sends its local time in a message to the remaining processes and waits to receive a similar message from every other process.

We start with an informal description of $\mathcal{A}$. Each process p keeps a special register R_p; as for local time, a piece-wise continuous function of (real) time t, $R_p(t)$, can be defined. If p receives a message from q saying that q's local time is L_q, at its next computation step, when the local time of it is, say, L_p, it estimates the difference between its local time with that of q to be $L_q + \frac{d}{2} - L_p$ and adds this value to R_p. After receiving local times from all other processes, it sets R_p to the average of the estimated differences (including 0 for the difference between p and itself) by simply dividing R_p by n; next, p sets L_p to $L_p + R_p$, i.e. it adds R_p to the current value of L_p. Finally, it sets R_p back to 0 and passes to a synchronized state, having completed its synchronization algorithm.

We analyze the precision achieved by the above algorithm. Consider the real time t_p at which process p enters a synchronized state and let q be a process that entered a synchronized state at $t_q < t_p$. Let $L_p(t-)$ and $L_p(t+)$ be the values that L_p attains right before and right after, respectively, the last computation step of p. (Note that, according to the definition of synchronization we have proposed, $L_p(t_p+)$ is what is really important and should be compared to $L_q(t_p)$; we can consider $L_p(t_p-)$ as, merely, an intermediate value.) Let, also, $R_p(t_p-)$ be the average of the estimated (by p) differences of its local time with those of the other processes and $R_p(t_p+)$ be 0. By the algorithm, $L_p(t_p+) = L_p(t_p-) + R_p(t_p-)$. We can define the corresponding quantities: $L_q(t_q-), L_q(t_q+), R_q(t_q-)$ and $R_q(t_q+) = 0$ for the process q. For any i, $1 \leq i \leq n$, and any t_1, t_2, $t_1 < t_2$, we denote by $T_i(t_1, t_2)$ the number of physical ticks that process p_i received from its local clock between the real times t_1 and t_2. We have:

$$\begin{aligned}
|L_p(t_p+) - L_q(t_p)| &= |L_p(t_p-) + R_p(t_p-) - (L_q(t_q+) + T_q(t_q, t_p))| \\
&= |L_p(t_q) + T_p(t_q, t_p) + R_p(t_p-) - (L_q(t_q-) + R_q(t_q-) + T_q(t_q, t_p))| \\
&= |L_p(t_q) + T_p(t_q, t_p) + R_p(t_p-) - L_q(t_q-) - R_q(t_q-) - T_q(t_q, t_p)| \\
&\leq |L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-))| + |T_p(t_q, t_p) - T_q(t_q, t_p)|
\end{aligned}$$

We start by showing:

Lemma 5.4 $|L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-))| \leq \frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2})$

Proof: For each $r \in P$, let D_{rq} be the difference of the local times of processes r and q, as estimated by the process q. Also, let D_{rp} be the difference of the local times of processes r and p, as estimated by the process p. By the algorithm, $R_q(t_q-) = \frac{1}{n}\sum_{r\in P} D_{rq}$ and $R_p(t_p-) = \frac{1}{n}\sum_{r\in P} D_{rp}$. We have:

$$\begin{aligned}
& L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-)) \\
= \;& L_p(t_q) - L_q(t_q-) - (\frac{1}{n}\sum_{r\in P} D_{rq} - \frac{1}{n}\sum_{r\in P} D_{rp}) \\
= \;& \frac{1}{n}(n(L_p(t_q) - L_q(t_q-)) - (\sum_{r\in P} D_{rq} - \sum_{r\in P} D_{rp})) \\
= \;& \frac{1}{n}\sum_{r\in P}(L_p(t_q) - L_q(t_q-) - (D_{rq} - D_{rp}))
\end{aligned}$$

For any process r, $r \in P$, let $t = \min\{t_r, t_q\}$. (For notational simplicity, we hide the fact that t is, actually, dependent on r.) We add and subtract $L_r(t)$ in the right side of the above equation to get:

$$\begin{aligned}
& L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-) \\
= \;& \frac{1}{n}\sum_{r\in P}((L_p(t_q) - L_r(t)) - (L_q(t_q-) - L_r(t)) - (D_{rq} - D_{rp})) \\
= \;& \frac{1}{n}\sum_{r\in P}((L_r(t) - L_q(t_q-) - D_{rq}) - (L_r(t) - L_p(t_q) - D_{rp}))
\end{aligned}$$

Hence:

$$\begin{aligned}
& |L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-))| \\
\leq \;& \frac{1}{n}\sum_{r\in P}|(L_r(t) - L_q(t_q-) - D_{rq}) - (L_r(t) - L_p(t_q) - D_{rp})| \\
\leq \;& \frac{1}{n}\sum_{r\in P}(|L_r(t) - L_q(t_q-) - D_{rq}| + |L_r(t) - L_p(t_q) - D_{rp}|) \\
= \;& \frac{1}{n}(\sum_{r\in P}|L_r(t) - L_q(t_q-) - D_{rq}| + \sum_{r\in P}|L_r(t) - L_p(t_q) - D_{rp}|)
\end{aligned}$$

Next, we show some simple facts:

Claim 5.2 $\sum_{r \in P} |L_r(t) - L_q(t_q-) - D_{rq}| \leq \lceil \frac{2d}{c} \rceil + \frac{d}{2}$

Proof: Notice that for $r = q$, $t = t_q$ and $L_r(t) = L_q(t_q-)$, so that: $|L_r(t) - L_q(t_q-) - D_{rq}| = |L_q(t_q-) - L_q(t_q-) - D_{rr}| = |0 - 0| = 0$. For $r \neq q$, let t_1 be the (real) time at which process r sends its local time, $L_r(t_1)$, to every other process and let t_2 be the (real) time at which process q receives it, or, rather, the (real) time at which process q takes a computation step at which it estimates the difference in local times between process r and itself. (Again, for notational simplicity, we hide the fact that t_1 and t_2 are, actually, dependent on r.) We have:

$$|L_r(t) - L_q(t_q-) - D_{rq}| = |L_r(t) - L_q(t_q-) - (L_r(t_1) + \frac{d}{2} - L_q(t_2))$$

Note, however, that since, by definition, $t_2 \leq t_q$, and process q can only *increase* L_q in the interval $[t_2, t_q-]$ by incrementing its value by one every time it receives a tick, it follows that: $L_q(t_2) \leq L_q(t_q-)$. Hence:

$$\begin{aligned} |L_r(t) - L_q(t_q-) - D_{rq}| &\leq |L_r(t) - L_q(t_q-) - (L_r(t_1) + \frac{d}{2} - L_q(t_q-))| \\ &= |L_r(t) - L_r(t_1) - \frac{d}{2}| \\ &\leq |L_r(t) - L_r(t_1)| + \frac{d}{2} \end{aligned}$$

Note, however, that since, by definition, $t \leq t_r$, process r can only increase L_r in the interval $[t_1, t]$ by incrementing its value by one every time it receives a physical tick. Thus:

$$|L_r(t) - L_r(t_1)| \leq \lceil \frac{t - t_1}{c} \rceil.$$

But, $t - t_1 \leq t_r - t \leq 2d$, since a communication between process r and any other process can take time up to $2d$. So, combining the above, we get:

$$|L_r(t) - L_q(t_q-) - D_{rq}| \leq \lceil \frac{2d}{c} \rceil + \frac{d}{2}$$

Therefore:

$$\begin{aligned}\sum_{r\in P}|L_r(t)-L_q(t_q-)-D_{rq}| &\leq (n-1)\max_{r\in P}|L_r(t)-L_q(t_q-)-D_{rq}| \\ &\leq (n-1)(\lceil\frac{2d}{c}\rceil+\frac{d}{2})\end{aligned}$$

■

As in Claim 5.2, we can show:

Claim 5.3 $\sum_{r\in P}|L_r(t)-L_p(t_q-)-D_{rp}| \leq (n-1)(\lceil\frac{2d}{c}\rceil+\frac{d}{2})$

The lemma follows from the last two claims. ■

We next show:

Lemma 5.5 $|T_p(t_q,t_p)-T_q(t_q,t_p)| \leq \frac{1-c}{c}d+1$

Proof: Clearly, $\lceil t_p-t_q\rceil \leq T_p(t_q,t_p) \leq \lceil\frac{t_p-t_q}{c}\rceil$ and $\lceil t_p-t_q\rceil \leq T_q(t_q,t_p) \leq \lceil\frac{t_p-t_q}{c}\rceil$. Hence: $|T_p(t_q,t_p)-T_q(t'_q,t_p)| \leq \lceil\frac{t_p-t_q}{c}\rceil - \lceil t_p-t_q\rceil$. Note, however, that: $0 < t_p-t_q \leq d$, since every process is alive at t_q (otherwise, q could not have heard from all of them and go to a synchronized state at t') and a message from any process to p must reach p within time d from t_q. Thus, we have:

$$\begin{aligned}T_p(t_q,t_p)-T_q(t_q,t_p) &\leq \lceil\frac{t_p-t_q}{c}\rceil - \lceil t_p-t_q\rceil \\ &\leq \frac{t_p-t_q}{c}+1-(t_p-t_q) \\ &\leq \frac{1-c}{c}d+1\end{aligned}$$

■

The theorem follows from Lemma 5.4 and Lemma 5.5. ■

Chapter 6

Discussion and Directions for Future Research

In this Chapter, we summarize and discuss the results in this thesis and possible extensions of them, and suggest some general directions for future research in the area of timing-based, distributed computation.

Our results for the *continuous-time model* are discussed in Section 6.1. Section 6.2 surveys our results for the *discrete-time* model. We conclude, in Section 6.3, with a general discussion and some important open problems.

6.1 Continuous-Time Model

We presented strongly timing-dependent, full caching linearizable implementations of shared memory consisting of read/write objects in perfect and imperfect clock models. We also presented lower bound results to support optimality of our implementations. These results indicate that the goal of designing efficient, linearizable implementations of shared memory whose logical correctness is timing-independent may be too hard to achieve.

Although there is a gap between our lower bound of $d + \frac{u}{2}$ and upper bound of $d + 7u$ on $|R| + |W|$ in the imperfect clocks model, we feel we have substantially answered the question

of how $|R| + |W|$ depends on the parameters d and u. In particular, our upper bounds show that only a single "long communication" (i.e., a communication requiring time d) is needed, which cannot be avoided (cf. [35]). As our model approaches the synchronous model, i.e., as the message delay uncertainty u becomes smaller, the additive $\Theta(u)$ term in our lower bounds becomes smaller, and our upper bounds get "closer" to optimal.

As usual, it is necessary to be cautious in making inferences about the performance of real systems from theoretical negative results, for the theoretical results are often based on (unrealistic) assumptions that might be weakened in practice. In the case of our lower bound results, however, we believe that such practical inferences about the performance of (real) cache consistency protocols can be safely drawn since our theoretical assumptions are quite minimal. We feel that our lower bound results still apply for real protocols which are more complex, since they have to deal with issues such as cache misses and network or bus contentions. (Note that the executions used in our lower bound proofs are "mild" and do not rely on any serious network congestion that is non-tolerable in practice.)

Our work continues the study of the cost of implementing memory objects, under various correctness conditions and timing assumptions, for shared-memory multiprocessor systems, initiated in [13, 5, 35]. Although our model ignores several important practical issues, like, e.g., 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.

We expect that our synchronization schemes, in particular, the "time- slicing" technique, will be applicable to other problems in distributed computing, in particular, to more general broadcasting and deadlock resolution problems.

Our results assume that clocks are available to processes; what if processes have no timing information at all and computations are totally asynchronous and message-driven? What is the tightest coefficient of d in $|R| + |W|$ 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 snap-

Step time range	Upper bound	Lower bound
$[0,1]$	$diam(G)D(s-1)$	$diam(G)d(s-1)$
$(0,1]$	$1+diam(G)D(s-2)$	$1+diam(G)d(s-2)$
$[c,1]$ $(0<c\leq 1)$	$1+\min\{\lfloor\frac{1}{c}\rfloor+1, diam(G)D\}(s-2)$	$\lfloor\frac{1}{c}(s-2)\rfloor$, if no communication is used
		$1+\min\{\lfloor\frac{1}{2c}\rfloor, diam(G)d\}(s-2)$, if $d\geq\frac{d}{\min\{\frac{1}{2c}, diam(G)d\}}+2$

Figure 6-1: Summary of our main results for networks

shots (cf. [1]) under sequential consistency and linearizability. How does strengthening of the shared memory primitives affect the worst-case response times? In the opposite direction, is it possible to obtain better bounds on response times by relaxing the correctness conditions to weaker ones like, e.g., causal memory (cf. [2]), or slow memory (cf. [29])? (Results in this direction have already been obtained in [8].) We leave all of these as a subject for future work.

6.2 Discrete-Time Model

6.2.1 Semisynchrony versus Asynchrony

Assuming that $1 \ll d$, i.e., that $D \approx d$, we have almost matching upper and lower bounds of $diam(G)D(s-1)$ for the asynchronous network model. For the semi-synchronous network model, we showed an upper bound of $1+\min\{\lfloor\frac{1}{c}\rfloor+1, diam(G)D\}(s-2)$ and a lower bound of $1+\min\{\frac{1}{2c}, diam(G)D\}(s-2)$. Neglecting roundoffs, the upper bound is within a factor of 2 of the lower bound. Similar results were proved for the cases where message delays are not uniform. The case where processes do not start simultaneously was also studied, and our techniques were extended to yield similar, although less tight results for this case. We summarize our main results for networks in Figure 6.2.1.

We also showed a lower bound of $1+\min\{\lfloor\frac{1}{2c}\rfloor, \lfloor\log_b(n-1)-1\rfloor\}(s-1)$ on the time complexity of the s-session problem in a realistic semi-synchronous, shared-memory model.

Neglecting round-offs, this lower bound is no less than 1/2 of a simple (combined) upper bound described in Section 4.2.1.

Our lower bound results show the inherent limitations on using timing information in systems where communication is achieved through either network or shared memory.

Our work continues the study of time bounds in the presence of timing uncertainty within the framework of the semi-synchronous model (cf. [9, 7, 52, 24]). Our results give a time separation between semi-synchronous (in particular, synchronous) and asynchronous systems achieving communication through either network or shared memory. Unlike previous separation results ([4, 38]), our results do not rely on the ability to schedule several steps by the same process at the same real time.

Our work leaves open several interesting problems. An obvious open problem is to close the gap between the lower and the upper bounds for the semi-synchronous case (in both the network and shared memory models). It will be interesting to relax the assumption $d \geq \frac{d}{\min\{\lfloor 1/2c \rfloor, diam(G)d\}} + 2$ used in to prove the lower bound for the semi-synchronous network model. The definition of a session does not require processes to be "aware" of a session's end; how do the bounds change if this requirement is imposed?

Our results show that there are some synchronous algorithms that *cannot* be simulated by asynchronous time algorithms without significant time overhead (e.g., algorithms for the s-session problem). In contrast, the results of Awerbuch ([14]) indicate that there are some synchronous algorithms which *can* be simulated by asynchronous algorithms with only constant time overhead. Perhaps the most interesting extension of our research is to characterize the synchronous algorithms which can (respectively, cannot) be efficiently simulated by asynchronous algorithms.

6.2.2 From Semi-Synchrony to Real Time

We defined the tick synchronization problem, a variant of the general synchronization problem, in semi-synchronous distributed networks and proposed the precision achieved by a tick synchronization algorithm as an appropriate worst-case measure of its performance. We showed

that no algorithm can solve the tick synchronization problem and yet achieve precision less than $\lfloor \frac{d-2}{2c} \rfloor$. On the positive side, we presented a simple algorithm that achieves a precision of $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$. Neglecting round-offs and considering only terms proportional to $\frac{d}{c}$ as the dominant ones in the expressions for our lower and upper bounds on precision, our upper bound can be easily seen to be within a factor of about 8 of the lower bound. We believe that an algorithm using more sophisticated averaging than the one we presented may exist and imply a better upper bound on precision. We are currently investigating this possibility.

There are several open problems directly related to our work on tick synchronization. Most obviously, there is a gap remaining between our upper and lower bounds. It would be interesting to consider the same problem in a model in which there is a nontrivial lower bound on the time for message delivery. While our upper bound proof still goes through in this model, the same is not true for our lower bound proof. Perhaps, the most intriguing open problem is the extension of our work to the case of a general communication network.

6.3 Open Problems

We conclude our thesis with a list of important open problems in the area of timing-based distributed computation.

- Develop a framework model, perhaps as a sufficiently expressive extension of that of the *timed automaton model* (cf. [47, 9]), to host assertional reasoning proofs of timing properties for algorithms subject to *probabilistic* timing assumptions. Predictable performance, in the form, e.g., of safety properties to hold with overwhelming probability, is often a desirable characteristic of real-time systems. Such a framework model could use the timed automaton model as a starting point, but, instead, use the *boundmap* for a formal description of the probabilistic timing assumptions for the components of the system. It would be very interesting to explore probabilistic analogs of the *progress functions* developed in [39]. Also, allowing such algorithms to make use of randomization

techniques presents a new class of verification problems.

- Extract prototypical problems from practical real-time systems research and use them as a basis for combinatorial work. Good candidates for such problems are several variants of *hazard avoidance* problems (cf. [30]).

- Derive the exact time and space complexities of variants of the fundamental problem of *reaching agreement*, under various assumptions on possible faults (e.g., *fail-stop, omission, Byzantine,* etc.), in timing-based models of distributed computation like, e.g., the semi-synchronous model. It would be extremely interesting to derive bounds on costs for solving agreement problems in the shared-memory semi-synchronous model studied in our thesis. The *approximate agreement* problem (see, e.g., [23, 11]), closely related to the important problem of synchronizing local clocks in a distributed system, naturally lends itself to rigorous time complexity analysis. It would be interesting to know how relaxing exactness affects the time bounds in [7, 52], more specifically how these bounds will depend on the *approximation constant* ϵ. The *renaming problem* and its variants (cf. [6]) is an important problem that should be studied in timing-based models; can the timing assumptions of the semi-synchronous model reduce the inherent space complexities that this problem has in asynchronous models? Probabilistic versions of the agreement problem should also be formulated and studied in timing-based models.

- Obtain complexity results on the effect of the strength of correctness conditions, such as serializability (cf. [49]), for highly available replicated databases on the costs of supporting them. Such a cost might be the worst-case completion time, the amount of communication or the local storage needed for performing a transaction. Such results will naturally provide intuition for defining new, perhaps more appropriate correctness conditions, possibly by strengthening the ones in [40] while still sacrifying serializability, that are tailored towards specific applications. We expect that precise trade-offs between these costs and the strength of a correctness condition can be shown in a way similar to those for distributed implementations of concurrent data structures ([13, 46]).

- Develop the complexity theory of concurrent data structures. In particular, investigate the possible advantages of supplying timing information to processes on the time complexity of data accessing, and also the inherent costs of building *fault-tolerant* concurrent data structures. A fundamental problem in concurrent computation is that of implementing stronger shared-memory primitives (*registers*) by weaker ones; the costs of such implementations, expressed as the number of *physical* registers used in implementing one *logical* register, or the number of operations performed on a physical register per operation on the logical register, are of ultimate importance. Previous work on these costs (see, e.g., [21]) assumed that no timing information was available to processes. We believe that providing such information to processes can significantly improve the costs that are inherent in its absence.

- The area of communication protocols should be a good arena for timing-dependent algorithms. It would be worth studying, for example, the design of efficient timer-based protocols for the transport layer of communication networks. As for dynamic networks, one could study the basic capabilities of algorithms that assume a certain amount of reliability, quantified at a given moment as the time since the link last recovered (cf. [16]). It has been demonstrated in [16] that algorithms for broadcasting, end-to-end communication and topology update gain in efficiency by assuming a certain amount of link reliability; similar results should be possible for other dynamic network algorithms.

- Investigate in depth the capabilities and the limitations of the *approximately synchronized clocks* model introduced in [57]. In this model, each process obtains timing information from a local clock which runs "within an envelope" of real time. More precisely, for any pair or real times t_1 and t_2, $t_1 < t_2$, the difference in clock values at t_2 and t_1 is at least $a_1(t_2 - t_1) - a_3$ and at most $a_2(t_2 - t_1) + a_3$ for some positive constants a_1, a_2 and a_3 such that $0 \leq a_1 \leq 1 \leq a_2 \leq \infty$. thus, the quantity $A = \frac{a_2}{a_1}$, called *accuracy*, is a measure of the timing uncertainty in this model. (Preliminary steps in this direction appear in [43], where the session problem serves as a convenient vehicle for showing time separation results between variants of the approximately synchronized

clocks model that are analogs of the asynchronous and the semi-synchronous models.)

Glossary of Notation

α	timed execution, p. 33
$\mathcal{A}$	algorithm (protocol), pp. 31, 32
$\mathcal{A}_i$	local algorithm of process p_i, pp. 31, 32
$Ack_i(X)$	response event of a write operation, p. 25
$Access(\mathcal{R})$	set of processes that can access $\mathcal{R}$, p. 31
b	fan-in, p. 31
$buffer_i$	message buffer of process p_i, p. 31
c	lower bound on process step time, p. 33
C	configuration (vector of local states), p. 30
$\tilde{C}$	extended configuration (vector of local states and shared variables), p. 31
c_i	local clock parameter at node i, p. 25
Cl_i	local clock at node i, p. 25
$comp(i, \mathcal{R})$	computation step of process p_i (shared memory), p. 31
$comp(i, S)$	computation step of process p_i (network), p. 30
d	upper bound on message delivery time, pp. 27, 34
D	$d + 1$, p. 34
δ	accuracy, p. 43
Δ	precision, p. 36
$del_i(j, m)$	message delivery event, pp. 25, 30
$diam(G)$	diameter of G, p. 35

$D(i,m)$	set of indices of delivery events, p. 32
$dist(i,j)$	distance of i and j (in G), p. 34
E	edge set, p. 30
ϵ	arbitrary small constant, p. 45
γ	execution in discrete-time models, p. 32
G	graph, p. 30
h_i	history for MCS process p_i, p. 26
I_i	subset (of Q_i) of idle states, p. 30
L_i	real-time register of process p_i, p. 35
m	message, pp. 25, 30
$\mathcal{M}$	message alphabet, pp. 25, 30
μ	tunable parameter, p. 38
n	number of nodes and processes, pp. 24, 30
op	operation, p. 27
$Op(X)$	set of operations on X ($X \in \mathcal{X}$), p. 27
$ops(\sigma)$	sequence of call and response events in σ, p. 28
P	collection of nodes (processes), pp. 24, 30
π	(atomic) event, p. 32
p_i	MCS process at node i, pp. 25, 30
P_i	application program at node i, p. 25
q	state, pp. 26, 30
Q_i	state set of process p_i, p. 30
$q_{0,i}$	initial state of process p_i, p. 30
R	set of response events, p. 26
$\mathcal{R}$	shared variable, p. 31
$\|R\|$	maximum, over all X, of $\|R(X)\|$, p. 29
$\Re$	real-time domain, p. 26

$Read_i(X)$	call event of a read operation, p. 25
$Return_i(X, v)$	response event of a read operation, p. 25
rop	read operation, p. 28
$\|R(X)\|$	maximum time for a read operation on X, p. 29
s	number of sessions, p. 35
S	set of message-send events, or send actions, pp. 26, 30
σ	execution in the continuous-time model, p. 26
$send_i(j, m)$	message-send event, or send action, pp. 25, 30
$\sigma_{i,m}$	one-to-one onto mapping from $S(i, m)$ to $D(i, m)$, p. 32
$SetTimer_i(t)$	timer-set event at MCS process p_i, p. 25
$\mathcal{S}_i$	send actions possible for p_i, p. 30
$S(i, m)$	set of indices of send events, p. 32
$state_i(C)$	ith state component of C (or $\tilde{C}$), pp. 30, 31
t	real time, p. 25
t_i	synchronization time of process p_i, p. 36
T	set of timer-set events, p. 26
τ	operation sequence, p. 28
$\tau\|i$	restriction of τ to operations at p_j, p. 28
$\tau\|X$	restriction of τ to operations on X, p. 28
u	message delay uncertainty, p. 27
V	vertex set, p. 30
$\mathcal{V}$	domain (set of values), pp. 24, 31
$val(op)$	value associated with operation op, p. 27
$value_k(\tilde{C})$	kth value component of $\tilde{C}$, p. 31
$\|W\|$	maximum, over all X, of $\|W(X)\|$, p. 29
wop	write operation, p. 28
$Write_i(X, v)$	call event of a write operation, p. 25

$\|W(X)\|$	maximum time for a write operation on X, p. 29
X	read/write object, p. 24
$\mathcal{X}$	collection of read/write objects, p. 24
$\perp$	"undefined" value, pp. 24, 31
$<_{\mathcal{V}}$	total order on $\mathcal{V}$, p. 24
$\xrightarrow{\sigma}$	partial order specified by σ, p. 28

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 1–14, August 1990.

[2] J. Ahamad, P. Hutto, and R. John, *Implementing and Programming Causal Distributed Shared Memory,* Technical Report TR GIT-CC-90-49, College of Computing, Georgia Institute of Technology, 1990.

[3] R. Alur, T. Feder, and T. Henzinger, "The Benefits of Relaxing Punctuality," in *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pp. 139–152, August 1991.

[4] E. Arjomandi, M. Fischer, and N. Lynch, "Efficiency of Synchronous versus Asynchronous Distributed Systems," *Journal of the ACM,* Vol. 30, No. 3, pp. 449–456, July 1983.

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

[6] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an Asynchronous Environment," *Journal of the ACM*, Vol. 37, No. 3, pp. 524–548, July 1990.

[7] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, "Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty," in *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pp. 359–369, May 1991.

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

[9] H. Attiya and N. Lynch, "Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty," in *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pp. 268–284, December 1989.

[10] H. Attiya and N. Lynch, "Theory of Real-Time Systems–Project Survey," in *Foundations of Real-Time Computing: Formal Specifications and Methods* (A.M. van Tilborg and G.M. Koob, eds.), pp. 111–138, Kluwer Academic Publishers, 1991.

[11] H. Attiya, N. Lynch, and N. Shavit, "Are Wait-Free Algorithms Fast?," in *Proceedings of the 31st IEEE Annual Symposium on Foundations of Computer Science*, pp. 55–64, October 1990.

[12] H. Attiya and M. Mavronicolas, "Efficiency of Semi-Synchronous versus Asynchronous Networks," in *Proceedings of the 28th Annual Allerton Conference on Communication, Control and Computing*, pp. 578–587, October 1990.

[13] H. Attiya and J. Welch, "Sequential Consistency versus Linearizability," in *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pp. 304–315, July 1991.

[14] B. Awerbuch, "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, No. 4, pp. 804–823, October 1985.

[15] B. Awerbuch, A. Baratz, and D. Peleg, "Cost-Sensitive Analysis of Communication Protocols," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 177–187, August 1990.

[16] B. Awerbuch, O. Goldreich, and A. Herzberg, "A Quantitative Approach to Dynamic Networks," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 189–203, August 1990.

[17] J.C.M. Baeten and J.A. Bergstra, *Real-Time Process Algebra,* Technical Report P8916b, University of Amsterdam, 1990.

[18] G. M. Baudet, "Asynchronous Iterative Methods for Multi-Processors," *Journal of the ACM*, Vol. 25, No. 2, pp. 226–244, April 1978.

[19] A. Bernstein and P. Harter, "Proving Real-Time Properties of Programs with Temporal Logic," in *Proceedings of the 8th Symposium on Operating Systems Principles*, Vol. 15, No. 5, pp. 1–11, Operating Systems Review, December 1981.

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

[21] S. Chaudhuri and J. Welch, "Bounds on the Costs of Register Implementations," in *Proceedings of the 4th International Workshop on Distributed Algorithms (WDAG '90),* (J. van Leeuwen and N. Santoro, eds.), pp. 402–421, Lecture Notes in Computer Science (Vol. 486), Springer-Verlag, September 1990.

[22] J.E. Coolahan and N. Roussopoulus, "Timing Requirements for Time-Driven Systems using Augmented Petri Nets," *IEEE Transactions on Software Engineering,* Vol. SE-9, No. 1, pp. 603–616, September 1983.

[23] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching Approximate Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 33, No. 3, pp. 499–516, July 1986.

[24] C. Dwork and L. Stockmeyer, "Bounds on the Time to Reach Agreement as a Function of Message Delay," Technical Report RJ 8181 (75140), IBM, June 1991.

[25] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with one Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374–382, April 1985.

[26] J. Halpern, N. Megiddo, and A. Munshi, "Optimal Precision in the Presence of Uncertainty," *Journal of Complexity*, Vol. 1, No. 2, pp. 170–196, December 1985.

[27] T. Henzinger, "Half-Order Modal Logic: How to Prove Real-Time Properties," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 281–296, August 1990.

[28] 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.

[29] P. Hutto and M. Ahamad, *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories,* Technical Report GIT-ICS-89/39, College of Computing, Georgia Institute of Technology, October 1989.

[30] F. Jahanian and A. Mok, "A Graph-Theoretic Approach for Timing Analysis and its Implementation," *IEEE Transactions on Computers*, Vol. C-36, pp. 961–975, August 1987.

[31] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565, July 1978.

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

[33] H.R. Lewis, *Finite State Analysis of Asynchronous Circuits with Bounded Temporal Uncertainty,* Technical Report TR-15-89, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, 1989.

[34] H.R. Lewis, "A Logic of Concrete Time Intervals," in *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pp. 380–389, June 1990.

[35] R. Lipton and J. Sandberg, *A Scalable Shared Memory,* Technical Report CS-TR-180-88, Department of Computer Science, Princeton University, September 1988.

[36] 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.

[37] J. Lundelius and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, Vol. 77, No. 1, pp. 1–36, April 1988.

[38] N. Lynch, *Asynchronous versus Synchronous Computation,* Lecture notes for MIT 18.438: Network Protocols and Distributed Graph Algorithms, Spring 1989.

[39] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 265–280, August 1990.

[40] N. Lynch, B. Blaustein, and M. Siegel, "Correctness Conditions for Highly Available Replicated Databases," in *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pp. 11–28, August 1986.

[41] N. Lynch and M. Fischer, "On Describing the Behavior and Implementation of Distributed Systems," *Theoretical Computer Science*, Vol. 13, No. 1, pp. 17–43, January 1981.

[42] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pp. 137–151, August 1987.

[43] M. Mavronicolas, *Accuracy and Real Time,* in preparation, Harvard University, 1992.

[44] M. Mavronicolas, "Efficiency of Semi-Synchronous versus Asynchronous Systems: Atomic Shared Memory," Technical Report TR-03-92, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, January 1992. To appear in *Computers and Mathematics with Applications.*

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

[46] M. Mavronicolas and D. Roth, *Efficient, Strongly Consistent Implementations of Shared Memory,* Technical Report TR-05-92, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, February 1992. Accepted for publication in *6th International Workshop on Distributed Algorithms (WDAG'92),* Haifa, Israel, 1992.

[47] M. Merritt, F. Modugno, and M. Tuttle, "Time Constrained Automata," in *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR '91)* (J.C.M. Baeten and J.F. Groote, eds.), pp. 408–423, Lecture Notes in Computer Science (Vol. 527), Springer-Verlag, August 1991.

[48] S. Moran and Y. Wolfstahl, "Extended Impossibility Results for Asynchronous Complete Networks," *Information Processing Letters*, Vol. 26, No. 3, pp. 145–151, November 1987.

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

[50] G. Peterson, "Time-Space Trade-offs for Asynchronous Parallel Models: Reducibilities and Equivalences," in *Proceedings of the 7th Annual ACM Symposium on Theory of Computing*, pp. 224–230, May 1979.

[51] G. Peterson and M. Fischer, "Economical Solutions to the Critical Section Problem in a Distributed System," in *Proceedings of the 9th Annual ACM Symposium on Theory of Computing*, pp. 91–97, May 1977.

[52] S. Ponzio, *The Real-Time Cost of Timing Uncertainty: Consensus and Failure Detection,* Technical Report MIT/LCS/TR-518, Laboratory for Computer Science, MIT, October 1991.

[53] I. Rhee and J. Welch, "The Impact of Time on the Session Problem," to appear in the *11th Annual ACM Symposium on Principles of Distributed Computing*, August 1992.

[54] A. Segall, "Distributed Network Protocols," *IEEE Transactions on Information Theory*, Vol. IT-29, No. 1, pp. 23–35, January 1983.

[55] A.U. Shankar and S.L. Lam, "Time-Dependent Distributed Systems: Proving Safety, Liveness and Safety Properties," *Distributed Computing*, Vol. 2, No. 2, pp. 61–79, 1987.

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

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

[58] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware," in *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science*, pp. 233–243, October 1986.

[59] A. Zwarico, *Timed-Acceptance: an Algebra of Time-Dependent Computing,* Ph.D. Thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.