EFFICIENCY OF PARTIAL SYNCHRONY, AND RESOURCE ALLOCATION IN DISTRIBUTED SYSTEMS

by

INJONG RHEE

A dissertation submitted to the faculty of
the University of North Carolina at Chapel Hill in partial
fulfillment of the requirements for the degree of Doctor of
Philosophy in the Department of Computer Science.

Chapel Hill, 1994

Approved by:

Dr. Jennifer L. Welch
(Advisor)

Dr. James H. Anderson
(Member)

Dr. Kevin Jeffay
(Member)

INJONG RHEE. Efficiency of Partial Synchrony, and Resource Allocation in Distributed Systems. (Under the direction of Dr. Jennifer L. Welch.)

ABSTRACT

This dissertation is in two parts, covering two distinct areas of distributed computing.

The first part concerns timing models in distributed systems that lie *between* the synchronous and asynchronous models in terms of their assumption on synchrony. We study their time complexity for solving distributed computing problems in shared memory (SM) and message-passing (MP) systems.

We consider four timing parameters: the upper and lower bounds on process step time and message delay. Timing models are obtained by considering independently whether each parameter is known (i.e., is hard-wired into processes' code) or unknown, giving rise to four SM models and 16 MP models. We also study other timing models that are not covered by this framework.

We show a general time complexity hierarchy indicating inherent time complexity gaps among the models. The time complexity gaps are introduced by the time complexity of *the session problem*, an abstraction of fundamental synchronization problems in distributed computing. The hierarchy can be valuable information for system designers in evaluating various timing models to build efficient, yet cost-effective, distributed systems.

The second part concerns resource allocation in distributed systems, i.e., the scheduling of accesses to resources shared by concurrent processes. Three different resource allocation problems with varying degrees of generality are considered including *the dining philosophers problem*.

A tight bound on the response time for the dining philosophers problem is obtained. We also prove that any (dining philosophers) algorithm with the optimal response is transformable in polynomial time into a sequential algorithm for an NP-complete problem. It suggests that an execution of any dining philosophers algorithm with the optimal response time may require a large (perhaps exponential) amount of local computation in acquiring resources.

We also present a modular resource allocation algorithm that uses any resource allocation algorithm as a subroutine. For appropriate choices of the subroutine, we obtain the fastest known algorithms in the worst case. Although they do not give the optimal performance, the algorithms require only a polynomial amount of computation in acquiring resources. We did simulation studies indicating that our algorithm performs faster than other known algorithms on average.

To My Wife, *Mina* and My Mother, *Kyung An Pak.*

ACKNOWLEDGMENTS

My deepest gratitude goes to my advisor, Jennifer L. Welch, for her continuous encouragement throughout this research, for her excellent criticisms, superb editing and meticulous proofreading. She suggested the topic of Part I and co-authored the research paper that became a basis of Part I. She also helped me iron out many difficult definitions in Part II. She always made time for me, listening to my (both technical and non-technical) problems and provided me with abundant wisdom. I wish to thank her for everything that she taught me and also for her generous financial support.

I would like to thank my readers, Kevin Jeffay and James Anderson, for reading the draft and offering many suggestions that improved the presentation of the dissertation immensely. I thank my other committee members, Peter Calingaert, Steve Weiss and Don Stanat. Dr. Calingaert kindly served as the chairman of my dissertation committee as a substitute for Dr. Stanat whose sudden illness made him unable to continue to serve as the chairman, and he also helped me overcome various scheduling and administrative obstacles for my dissertation defense. I thank Dr. Weiss for serving in the committee on a very short notice. I am also greatly indebted to Dr. Stanat for his continued encouragement.

Doubtlessly, my life as a graduate student was not so enjoyable without much help from my friends at UNC and Texas A&M University. My special thanks go to Paramasivam (P2) who distributed the dissertation to the committee before the defense and helped me turn in the dissertation in time to graduate school. I wish to thank J.J. Kim for her kindness for sending me various documents required to turn in my dissertation. I would like to thank Amitabh Varshney, Srinivas Konuri and Goopeel Chung for their valuable suggestions to improve my defense talk. Good company with Alok Pancholi and Shlomi Dolev at Texas A&M University made my short stay in Texas fairly tolerable. I thank Ms. Janet Jones for helping me overcome various bureaucratic hurdles to finish this research. I also would like to thank Dr. James Abello for letting me use his lab during my stay at Texas A&M University.

I also would like to thank my undergraduate advisors at Kyung Pook National University and other faculty at UNC for their continued encouragement and for everything that I learned from them.

Finally, love and support from my wife, Mina and my mother, Kyung-An Pak made all these possible.

This research was partially supported by an NSF PYI Award CCR-9158478, an IBM Faculty Development Award, and NSF grant CCR-9010730. Part of this work was conducted while the author was at the Department of Computer Science, Texas A&M University.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PART I

Efficiency of Partial Synchrony

This part concerns the *partially synchronous* timing models in distributed systems that lie *between* the synchronous and asynchronous models in terms of their assumption on synchrony. We study their relative strength in terms of time complexity for solving distributed computing problems in both shared memory (SM) systems and message-passing (MP) systems. Partially synchronous models are more realistic because in most distributed systems, processes operate neither in perfect synchrony (the synchronous model) nor at completely independent rates (the asynchronous model).

The timing parameters of interest are the upper and lower bounds on process step time and message delay. The partially synchronous timing models are obtained by considering independently whether each parameter is known (i.e., can be hard-wired into processes' code) or unknown, giving rise to four SM models and 16 MP models.

We also study timing models that are not covered by this framework, called *periodic* models. In periodic models, each process takes steps at a constant unknown rate; different processes can have different rates.

These timing models include those that were studied before and are of practical interest. Furthermore, because our framework assumes that in all models, the timing parameters (i.e., bounds) always exist even if they can be unknown, we have a uniform basis for comparing performance quantitatively.

The thesis of this part of the dissertation is that there exist some inherent time complexity gaps among the aforesaid timing models in solving distributed computing problems. That is, given two timing models M_1 and M_2, there exists a problem that can be solved faster in M_1 than in M_2, but no problem that can be solved faster in M_2 than in M_1.

We use the *session* problem as a tool to demonstrate the differences in the time needed to solve problems in the timing models. The session problem is an abstraction of fundamental synchronization problems in distributed systems.

Almost tight bounds are obtained on the time complexity of the session problem in these models and are used to show the general time complexity hierarchy among various timing models. In addition to providing new insights into this family of timing models, the results presented subsume some previously known results and close an existing open problem.

Our time complexity hierarchy and gaps introduced by the session problem can be valuable information for distributed system designers in evaluating and comparing various timing models and deciding what kind of the timing model they have to provide in order to build efficient, yet cost-effective, distributed systems.

This part is organized as follows. Chapter 1 discusses our objective, related previous work and our work in some detail. Chapter 2 gives the definition of the system model. Chapters 3 and 4 discuss our lower and upper bound results on the time complexity of solving the (s, n)-session problem. Chapter 5 describes our results for the periodic models. Chapter 6 concludes this part and indicates future directions.

CHAPTER 1

Introduction

1.1 Objective

Early work in distributed computing usually assumed one of two extreme timing models: either the completely synchronous model, in which processes operate in lock-step rounds of computation and a message sent in a round is delivered in the next round, or the completely asynchronous, in which there are no bounds on process step time or message delay. However, in most distributed systems, processes operate neither at lock-step nor at a completely independent rate. Furthermore, the asynchrony assumption makes it very difficult to design and verify distributed algorithms whereas the perfect synchrony assumption is very expensive, if not impossible, to implement in distributed systems.

Based on these observations, researchers (e.g., [ADLS91, AL89, AM90, CT90, CW90, DDS87, DLS88, Po91]) began to investigate the impact on distributed computing if those timing assumptions are relaxed or tightened to some extent in order to reflect more realistic system situations. The new timing models that are obtained by relaxing or tightening the two extreme timing assumptions are called *partially synchronous models* . In general, a partially synchronous model is a model that assumes some timing information about when certain events occur such as process steps and/or message deliveries.

Much work needs to be done in evaluating and comparing partially synchronous models for their usefulness in reasoning about important problems in distributed computing and perhaps in developing new models if these prove to be inadequate. This part of the dissertation is a part of this effort.

Our thesis is that there exist inherent time complexity gaps among partially synchronous models. Generally, one timing model is said to be *stronger* than another timing model if the timing information that the former has is a superset of the timing information that the latter has. Any two models where one model is stronger than the other are said to be *comparable*. Our goal is, given two comparable timing models M_1 and M_2, to show that there exists some problem that can be solved faster in M_1 than M_2, but no problem that can be solved faster in M_2 than in M_1.

1.2 Methodology

We use the (s, n)-*session* problem to find the time complexity hierarchy among various timing models. The (s, n)-session problem was first presented in [AFL83] and further studied in [AM90]. Informally, a *session* is a minimal-length computation fragment that involves at least one special "synchronization" step by every process in a distinguished set of n processes. An algorithm that solves the (s, n)-session problem must guarantee that in every computation there are at least s disjoint sessions and eventually all the n processes become idle. The (s, n)-session problem is an abstraction of the synchronization needed in many distributed systems (see Section 1.4 for more detail). As we shall see, the time complexity of the (s, n)-session problem is very sensitive to the underlying timing assumptions of the model. For these two reasons, it is an important tool for studying time complexity in distributed computing under various timing assumptions.

Given two models M_1 and M_2, if there exists an (s, n)-session algorithm in M_1 that runs faster than any (s, n)-session algorithm in M_2, we say that M_1 is *higher* in the time complexity hierarchy that M_2 *with respect to* the (s, n)-session problem. This can be shown by developing lower and upper bounds on the time complexity of the (s, n)-session problem in each model and comparing them, so that if the lower bound in one model is larger than the upper bound in the other model, the former is higher than the latter in the time complexity hierarchy with respect to the (s, n)-session problem.

This hierarchy of timing models with respect to the (s, n)-session problem has an important application in showing a more general time complexity hierarchy among comparable timing models *regardless of* the problem being solved. This is because any algorithm that works correctly in weaker models also works correctly, unchanged, with equal time complexity in stronger models but not vice versa. Therefore, if a model M_1 is higher in the hierarchy with respect to the (s, n)-session problem than M_2 and also M_1 is stronger than M_2, there exists some problem (e.g., the (s, n)-session problem) that can be solved in M_1 faster than M_2, but no problem that can be solved faster in M_2 than in M_1. So in that case, we say that M_1 is higher in the time complexity hierarchy than M_2 in general, not just with respect to the (s, n)-session problem.

1.3 Motivation

The motivation of our work lies on a fundamental tradeoff between the efficiency of the partially synchronous models and the cost of implementing the models. That is, it is harder to design and verify distributed algorithms in weaker models than in stronger models, whereas it is harder to implement the stronger models than the weaker models. This tradeoff is due to the fact that the stronger models carry more timing information than the weaker models.

This tradeoff concerns distributed (hardware or software) system designers in building an efficient and easy-to-use, yet inexpensive and easy-to-implement distributed system. To implement such a system, the system designers should evaluate each piece of timing information concerning the system (such as bounds on process running speed or communication delay) to find out whether it is worthwhile to support such a timing guarantee (e.g., processes run at a known speed) in the system. Specifically, the natural questions they ask in the evaluation are whether implementing the timing guarantee makes the system more efficient and if it is the case, how much more efficient the system can be.

The time complexity hierarchy that involves many possible timing models gives a general overview on the efficiency of the partially synchronous models. Specifically, proving a time complexity gap among models shows that one model with a certain timing guarantee is inherently more efficient than other models without the guarantee. The time complexity hierarchy with respect to the (s, n)-session problem also shows that the time complexity gaps can be fundamental because efficient solutions to the

(s, n)-session problem can be used in many synchronization problems in distributed computing. Furthermore, if these gaps are non-negligible with respect to various timing parameters (e.g., bounds on running speed of processes), the time complexity gaps can provide valuable information to the system designers in evaluating and comparing the various timing models and deciding what timing guarantees they have to provide or do not have to provide to build efficient, yet cost-effective, distributed systems.

Another important application of this work is related to the model simulation works by [Aw85, NT87, We87, CGL93], that transform an algorithm designed for a stronger model to work correctly in a weaker model. Some of these works (e.g., [Aw85]) may seem to suggest that *any* algorithm designed for stronger models can be transformed to work correctly in weaker models with only negligible time complexity overhead. However, our time complexity gaps imply that for some models, any such transformation entails a significant time complexity overhead in the transformed code. Therefore, an execution of the transformed code may carry this overhead.

1.4 The (s, n)-Session Problem

The (s, n)-session problem is an abstraction of the synchronization needed to solve many distributed computing problems, where separate processes are responsible for performing a small portion of a task. The (s, n)-session problem, like the mutual exclusion and dining philosophers problems, concerns possible ordering of sequence of process events (e.g., a process finishing its assigned task) rather than the computation of particular outputs. Examples of the problem can be found in many concurrent systems.

Consider, for example, *barrier synchronization* [GVW89], a fundamental mechanism in concurrent systems which guarantees that all processes have finished a specified task in their execution before any are allowed to proceed. Barrier synchronization is a special case of the (s, n)-session problem when $s = 2$, and solutions for the $(2, n)$-session problem can be used to construct barrier synchronization: after each process finishes its specified task, it keeps taking synchronization steps until it recognizes there is one session, and then it proceeds to make the second session.

Another example of the (s, n)-session problem can be found in a system which solves a set of equations by successive relaxation, where each process holds part of the data (cf. [Ba78]). Each process takes one synchronization step when it changes its data. Sufficient interleaving of synchronization steps by different processes ensures that a correct value was computed, since it implies sufficient interaction among the intermediate values computed by the processes. Solutions for the (s, n)-session problem can be used to solve the successive relaxation problem.

The (s, n)-session problem is also an abstraction of a simple message distribution system in which a sending process writes a sequence of s messages one at a time on a board (e.g., port or mailbox) visible to all and waits after each message until all n other processes have read the message. Each reading step by a process is one synchronization step of the process. Any protocol which ensures that the sender has waited sufficiently long will also solve the (s, n)-session problem.

Because the (s, n)-session problem is an abstraction of many fundamental synchronization problems, and as we shall see, the time complexity of the (s, n)-session problem is very sensitive to the underlying timing assumptions of the model, it is an important tool for studying time complexity in distributed computing under various timing assumptions.

1.5 Timing Models

1.5.1 Timing Parameters

We consider two different interprocess communication models: *shared memory* and *message passing*. In the shared memory model, processes communicate only by means of shared variables. Each variable is shared by no more than a processes, where a is some constant. In a distributed shared memory system, some part of the memory is *local* to a process so that accesses to this part incur less time overhead than to the other (non-local) part of the memory. In our shared memory model, accesses to the variables that a process doesn't have direct access to may involve more than one step – intuitively, in the worst case it may require $O(\log_a n)$ steps where n is the total number of processes (imagine a tree network of processes that relay information from a variable to a variable). Therefore, a is a constant that reflects communication bound in the shared memory model.

In the message passing model, communication is done by exchanging messages across a network. A process can broadcast a message at a step; the message is guaranteed to be delivered to every process after some finite time. *Process step time* is the amount of time between two consecutive steps of the same process and *message delay* is the amount of time between when a message is sent and when the message is received.

The relevant timing parameters of a model are the lower bound on process step time, c_1, the upper bound on process step time, c_2, and additionally, for the message passing model, the lower bound on message delay, d_1, and the upper bound on message delay, d_2.

We consider families of timing models for both shared memory (SM) and message-passing (MP) systems. The timing models are obtained by considering independently whether each parameter is known (i.e., can be hard-wired into processes' code) or unknown, giving rise to four SM models and 16 MP models.

We also study other timing models that are not covered by the above framework, called *periodic* models and also have unknown timing parameters. In the shared memory *periodic* model, for each process there exists an unknown constant such that the process makes one step at every period of the constant. In the message-passing variant, the lower and upper bounds on message delay (d_1 and d_2) are known.

The models with unknown parameters are a closer approximation to real concurrent systems than the asynchronous model because most distributed computing problems involve only finite-length computation (i.e., problems are solved in a finite time), and the bounds on step time and message delay can be reasonably bounded on a *per-computation* basis although it may or may not be possible for the model to have known bounds. Indeed, there can be an infinite number of such computations (and therefore, an infinite number of such bounds) in a system. We capture this notion by modeling each unknown bound model in such a way that it consists of infinitely many "similar" models with known bounds.

In more detail, we model a system in which a set of parameters is unknown by an infinite collection of constituent systems, each of which has a different known bound on the values of the unknown parameters, but has the same values for the known parameters. An algorithm is said to work in the system if it works, unchanged, in each of the constituent systems. The running time of an algorithm in the system is measured in terms of a function of unknown and known parameters, such that for any of its constituent model, the algorithm always finishes in time less than or equal to the value of the function given by the constituent model. A problem is said to have a certain lower bound in the system if there is no algorithm whose running time is less than the lower bound. Because in our models the timing parameters always exist even if they can be unknown, it provides a uniform basis for comparing performance quantitatively, namely *real time* (i.e., actual execution time of algorithms).

1.5.2 Motivations of Our Models

Distributed systems are generally modeled in terms of process running rate and communication delay. Step time and message delay models in our models reflect process running rate and communication delay. Because our framework covers many possible partially synchronous models including those studied before, it provides an insight into the relationship among partially synchronous models in general, and also it is theoretically elegant to consider the complete set of the timing models that fit our framework instead of only those models that might be of practical interests. In addition, our framework not only covers most of the models that were studied before, but also those that might be of practical interests.

For example, models with unknown bounds on step time are studied in [DLS88, AAT94]. These models provide a useful abstraction of timing constraints of systems as the models with non-existent bounds (e.g., the asynchronous model), yet is stronger than the models with non-existent bounds. A good description for the motivation of these models can be found in [AAT94].

> ...... the assumption of asynchrony is too weak, and many problems have been shown to be unsolvable in the asynchronous model. These impossibility results never seem to bother practitioners, which brings up the question whether such a model is the correct abstraction for modeling real systems. The appeal of the model lies in the fact that while it abstracts from implementation details, it is a better approximation of the real concurrent systems compared to the asynchronous model. Furthermore, it is stronger than the asynchronous model enabling us to design algorithms for problems that are unsolvable in the asynchronous model.

Models with known lower and upper bounds on step time are commonly called *semi-synchronous models* and have been previously studied in the literature (see [AM90, AAT94, ADLS91, AL89, AT92, CT90, LS92, Po91, RW92]); and these results are reviewed in more detail in Section 1.6. The semi-synchrony models systems where information about process step time is only approximately known, e.g, processes may have access to inaccurate clocks that operate at approximately, but not exactly, the same rate.

Models in which the lower bound on step time is known, but the upper bound on step time is unknown were studied in [RW92]. These models abstract event-driven processing such as responding to user inputs or non-periodic device interrupts. They model processes that can be blocked for an arbitrarily long (but finite) time waiting for a certain condition to be true or a certain event to occur, but that cannot take two consecutive steps faster than a certain lower bound.

The periodic timing constraints are inspired by constraints with the same name commonly used in many real-time problems, especially in scheduling of real time tasks for a uniprocessor [Je91, JSM91, LL73] where the period of task occurrences conforms to the constraints. In practice, as quoted in [JSM91], periodic timing constraints are used in applications such as avionics and process control when accurate control requires continual sampling and processing of data.

1.6 Background

In this section, we first review the previous works on the time complexity gaps among various timing models that include the partially synchronous models. These gaps are shown based on the time complexity of many different problems. We categorize these works by the problems they studied.

We also review some related model simulation works. Finally, we discuss some

Authors	Problem studied	System Model	Time Complexity Gap
[DDS87]	Consensus	MP	many (see text)
[DLS88]	Consensus	MP	Unknown bound vs. Asynchronous GST model vs. Asynchronous
[AL89]	Mutual Exclusion	MP	Semi-synchronous vs. Asynchronous Synchronous vs. Semi-synchronous
[AM90]	Session	MP	Semi-synchronous vs. Asynchronous Synchronous vs. Semi-synchronous
[ADLS91]	Consensus	MP	Semi-synchronous vs. Asynchronous Synchronous vs. Semi-synchronous
[AAT94]	Consensus	SM	Unknown bound vs. Asynchronous Synchronous vs. Unknown bound

The previous works on the time complexity gaps among various timing models.

TABLE 1

other works that study various distributed computing problems in the partially synchronous models. Although they are not relevant to the time complexity gaps we study, they are related to the motivation of our work in that they explore the usefulness of various partially synchronous models in many different applications.

1.6.1 Time Complexity Gaps

In this section, we review the previous works on the time complexity gaps among the partially synchronous models. These gaps are shown based on the time complexity of many different problems. We categorize these works by the problems they studied. Table 1 summarizes this section.

1.6.1.1 The (s,n)-Session Problem

Arjomandi, Fischer and Lynch [AFL83] studied the (s,n)-session problem in synchronous and asynchronous shared memory models. *Synchronous* means that the lower and upper bounds on step time are both equal to a known constant. *Asynchronous* means that the lower bound on step time is 0 and upper bound on step time is infinite. Their results showed a significant time complexity gap between the synchronous and asynchronous models, namely that s rounds are sufficient for the synchronous case but $(s-1)\lfloor \log_a n \rfloor$ rounds are necessary for the asynchronous case. Recall from Section 1.2 that a is the maximum number of processes that can access a shared variable and n is the size of the distinguished set of processes. The implication is that no communication is needed at all in the synchronous case, but it is needed for every session in the asynchronous case. (The $\lfloor \log_a n \rfloor$ factor is essentially the cost of communication when no more than a processes can access any shared variable.)

Attiya and Mavronicolas [AM90] addressed the (s,n)-session problem in semi-synchronous and asynchronous message passing systems. *Semi-synchronous* means that the lower bound on step time (c_1), the upper bound on step time (c_2) and the upper bound on message delay (d_2) are some known constants where $c_1 > 0$. They modeled the asynchronous system differently than [AFL83]: they let $c_1 = 0$ and the lower bound on message delay (d_1) be equal to 0, while c_2 and d_2 are some known constants. Their results showed that the asynchronous message passing model requires $(s-1)d_2$ time to solve the (s,n)-session problem (i.e., at least one message delay per session) while the semi-synchronous model requires at least $\min\{\frac{c_2}{2c_1}c_2, d_2\}(s-1)$

time to solve the (s, n)-session problem. They also show an almost matching upper bound $\min\{\frac{c_2}{c_1}c_2, d_2\}(s-1)$ for the semi-synchronous message passing model. They left the lower bound on time complexity for solving the session problem in the semi-synchronous shared memory model as an open problem. Their results indicated an important time separation between semi-synchronous and asynchronous networks and between synchronous and semi-synchronous networks.

1.6.1.2 The Consensus Problem

Byzantine Agreement [PSL80, LSP82] (or the consensus problem) is a distributed computing problem where processes have to agree on one bit in the presence of faulty processes. Dolev, Dwork and Stockmeyer [DDS87] investigated the minimal synchrony requirement for solving consensus. They considered five synchrony conditions: synchronous (lockstep) or asynchronous step time, synchronous (upper bound on message delay) or asynchronous message delay, synchronous (FIFO) or asynchronous message order, broadcast transmission or point-to-point transmission and atomic receive/send or separate receive and send. They identified four cases in which a consensus protocol tolerating $n-1$ process (crash) faults exists, but any weaking of the system by changing one parameter from favorable (e.g., synchronous) to unfavorable (e.g., asynchronous) is sufficient for proof that there is no such protocol. n is the total number of processes in the system. These cases are (1) synchronous step time and synchronous message delay, (2) synchronous step time and synchronous message order, (3) broadcast transmission and synchronous message order, (4) synchronous message delay, broadcast transmission and atomic receive/send. Among these four cases, the first case is most relevant to our work because we also consider step time and message delay. The first case implies the time complexity gaps between synchronous step time and message delay and synchronous step time and asynchronous message delay, and between asynchronous step time and synchronous message delay. Their models are different from ours in that we assume the (step and message delay) bounds exist, but they can be unknown.

Dwork, Lynch and Stockmeyer [DLS88] proposed two new timing models, called the *unknown bound model* and the *GST model*, which lie between the synchronous and asynchronous models. In the unknown bound model, fixed upper bounds on step time and message delay exist, but they are not known a priori. In the GST model, the bounds are known, but they are only guaranteed to hold after some unknown time T which is called *Global Stabilization Time*. They showed that consensus in the presence of various types of faults is solvable in these models. Contrast this result with those of [FLP85] and [DDS87], where it is shown that consensus is impossible in the asynchronous systems (i.e., no bounds exist for step time and message delay) even under the most benign failure (i.e., crash failure). This contrast shows the time complexity gaps between the asynchronous model and the unknown bound model, and between the asynchronous model and the GST model.

Attiya, Dwork, Lynch and Stockmeyer [ADLS91] studied lower and upper bounds on time complexity for the consensus problem under crash failures in the semi-synchronous model where the lower and upper bounds on step time are some constants (the ratio of the upper bound over the lower bound is $C > 1$) and the upper bound on message delay is some constant d. They proved a lower bound of $(f-1)d + Cd$ and an upper bound of $2(f+1)d + Cd$, where f is the maximum number of faulty processes. Their results imply that the ratio C doesn't have to apply to the number of faulty processes (f) in the time complexity of solving the consensus problem in the semi-synchronous model. This is significant because any synchronous consensus algorithm in the semi-synchronous system can be trivially simulated with its time complexity increased by a factor of C.

1.6.1.3 The Mutual Exclusion Problem

The mutual exclusion problem [Di65], in which processes have to synchronize to use a resource exclusively, has been extensively studied in asynchronous systems. Attiya and Lynch [AL89] developed a semi-synchronous mutual exclusion algorithm in a distributed network where each process has a local clock running at a known rate with a lower bound and an upper bound on clock tick time; there exists an upper bound on the message delay; there are upper bounds on the process step time and the time that a process uses a resource. Their model assumes that a process that finishes using a resource cannot notify other processes that it finished using it. In this model, the mutual exclusion problem in the asynchronous model cannot be solvable because there is no bound on the time period that a process uses a resource exists. But in the synchronous system, the obvious lower and upper bound is $\Theta(n)$ because there is a fixed bound on the time period and processes can take turns to use the resource in a fixed rate. Their result showed that in the semi-synchronous system, it takes at least $\Omega(n \cdot C)$ time where C is the ratio of the upper bound over the lower bound on clock tick time.

Alur, Attiya and Taubenfeld [AAT94] studied the time complexity of solving the consensus problem using only atomic read and write registers in the unknown bound shared memory model where the upper and lower bounds on memory access time are unknown and each process can delay taking a step explicitly for time period d by executing a statement $delay(d)$. They showed that it takes $\Theta(\Delta \cdot fac^{-1}(\Delta))$ time to solve the consensus problem in the unknown bound shared memory model where Δ is the (unknown) upper bound on memory access time and fac^{-1} is the inverse of the factorial function; note that $fac^{-1}(r) = \Theta(\frac{\log r}{\log \log r})$. This implies that the consensus problem in the presence of one process failure is solvable in the unknown bound shared memory model using only atomic read-write registers while it is impossible in the asynchronous model [LA92]. Note that in the synchronous shared memory model (where Δ is known), the problem can be solved in $\Theta(\Delta)$ time [AT92] using only atomic read and write registers.

1.6.2 Model Simulation

Our work is related to the simulation works by [Aw85, NT87, We87, CGL93], that transform an algorithm designed for a stronger model to work correctly in a weaker model. Whereas their works are pursuing the possibility of such simulation without incurring much overhead in the execution of the transformed algorithm, our work is pursuing the impossibility of such simulation without a non-negligible time overhead. We show that for some models, the transformed algorithm's running time can be increased by a significant factor.

Awerbuch [Aw85] introduced the concept of a *synchronizer* as a way to translate (or simulate) any algorithm designed for the synchronous message passing model to work correctly in the asynchronous message passing model.

Later, Welch [We87] and Neiger and Toueg [NT87] independently showed how a distributed system with synchronous processes and asynchronous communication can be simulated by a system with asynchronous processes and communication. [We87], which considers process failures, implies that the impossibility result of [DDS87] for consensus in partially synchronous systems follows from the impossibility result of [FLP85] in asynchronous systems.

Chaudhuri, Gawlick and Lynch [CGL93] provided some simulation techniques that show how to transform an algorithm designed for the simple programming model in which processes have perfectly accurate clocks to run correctly in a more realistic partially synchronous model in which processes have access to clocks whose divergence from real time is bounded.

1.6.3 Other Related Previous Works

Here we discuss some other works that study various distributed computing problems in partially synchronous models. Although they are not relevant to the time complexity gaps, they are related to our work in that they explored the usefulness of various partially synchronous models in many different applications.

Lamport [LA87] and Fischer [LA87] developed simple mutual exclusion algorithms in the semi-synchronous shared memory model. The model assumes there exist upper and lower bounds on step time. The algorithm ensures mutual exclusion and deadlock-freedom using only atomic registers, but it does not ensure the no-lockout property or fault-tolerance.

Lynch and Shavit [LS92] extended Fischer's algorithm such that the algorithm ensures mutual exclusion and deadlock-freedom if the timing constraints are met, but also it ensures the mutual exclusion if not. They also showed that it is possible to solve the mutual exclusion problem with $O(\log n)$ atomic registers in the semi-synchronous model. Contrast this result with that of Burns and Lynch [BL80] which showed that n is the lower bound on the number of atomic registers in solving the mutual exclusion problem in an asynchronous shared memory model.

Alur and Taubenfeld [AT92] also extended Fischer's algorithm and improved on the time performance by making the synchronization fast when there is no contention in the semi-synchronous model.

Ponzio [Po91] extended the work of [ADLS91] to tolerate message omission and Byzantine failure in the semi-synchronous model. He developed an algorithm with running time $4(f-1)d + Cd$ under omission failure and an algorithm with running time $2fCd$ under Byzantine failure.

Attiya and Djerassi-Shintel [AD93] studied the time complexity of solving the consensus problem in the semi-synchronous model under *timing failures* where processes may only fail by not obeying the timing requirements on their steps and on messages they send (i.e., processes are simply *late*). They showed a lower bound of $\Omega(\frac{n}{n-f}Cd)$ for solving the consensus problem in the model, where n is the total number of processes in the model. This result is significant because time C is required for every round of computation to solve the consensus problem or the renaming problem under timing failures in the semi-synchronous model in the presence of the timing failure.

Other works developing efficient algorithms for various problems in different timing models include clock synchronization (see [SWL88] for survey), leader election [CT90], renaming [AD93], transaction commit [CW90] and atomic broadcast [CASD86, CDS90].

1.7 Our Results

1.7.1 Known/Unknown Bound Models

Tables 2 and 3 summarize our results on the lower and upper bounds for the time complexity of solving the (s, n)-session problem. In Table 2, we denote shared memory models by a pair (c_1, c_2) where c_1 is either ?, meaning the lower bound on step time is unknown, or a positive real, meaning the lower bound on step time is c_1 and similarly for c_2 (for the upper bound on step time). Similarly in Table 3 we denote message passing models by a tuple (c_1, c_2, d_1, d_2) where d_1 and d_2 are lower and upper bounds on message delay, and they can be either ? or positive reals. (Note that some of the (time complexity) bounds involve a parameter that is unknown, e.g., both bounds for S_1 includes models in which the upper bound on step time is unknown.

Intuitively this means that in executions in which the upper bound in step time is c_2, the runing time is the given function of c_2. See Chapter 2 for formal details).

From now on, we say lower bound (resp., upper bound) to mean only the lower bound (resp., the upper bound) on time complexity of the (s, n)-session problem. When they are used with different meaning, we will make such distinction. Note that $s \cdot c_2$ is the the obvious lower bound for all models because processes have to take at least s steps to solve the (s, n)-session problem and each step can take up to c_2 time. We omit from the table the obvious lower bound $s \cdot c_2$ for all models and assume that all the lower bounds are bigger than $s \cdot c_2$. In the tables, we have grouped timing models in terms of lower and upper bounds. If two timing models are in the same group, they have the same lower and upper bounds.

In the shared memory case (see Table 2), our results indicate that if either the upper or lower bound on process step time or both is unknown (the group S_1), then the running time for the (s, n)-session problem is $c_2\Theta(\log_a n)(s-1)$, i.e., one communication cost $(\Theta(\log_a n)c_2)$ is required for each session. On the other hand, if both bounds are known (the group S_2), then the lower bound is $c_2 \min\{\frac{c_2}{2c_1}, \log_a n\}(s-1)$ and the upper bound is $c_2 \min\{\frac{c_2}{c_1}, O(\log_a n)\}(s-1)$. In words, in the (c_1, c_2) model, processes can use timing information about relative step times to count locally in order to determine when enough sessions have elapsed; however, if the gap between the upper and lower bounds on step time is sufficiently large, then it is more cost-effective to use explicit communication. These results are analogous to those of [AFL83]: intuitively, if either bound is unknown, then the system can be considered somewhat

Group	Lower bound	Upper bound
S_1	$(s-1)(\log_a n)c_2$	$(s-1)O(\log_a n)c_2$
S_2	$(s-1)\min\{\frac{c_2}{2c_1}c_2, (\log_a n)c_2\}$	$(s-1)\min\{\frac{c_2}{2c_1}c_2, O(\log_a n)c_2\}$

$$\begin{aligned} S_1 = \quad & \{(?,?),(?,c_2),(c_1,?)\} \\ S_2 = \quad & \{(c_1,c_2)\} \end{aligned}$$

Bounds for the session problem in shared memory models. c_1 and c_2 are the lower and upper bounds on step time.

TABLE 2

Group	Lower bound	Upper bound
M_1	$(s-1)d_2$	$(s-1)d_2$
M_2	$(s-1)\frac{d_2}{(d_2+d_1)}u$	$(s-1)\min\{\frac{d_2}{d_1}u, d_2)\}$
M_3	$(s-1)\frac{d_2}{(d_2+d_1)}u$	$(s-1)\min\{\frac{c_2}{c_1}u, \frac{d_2}{d_1}u, d_2)\}$
M_4	$(s-1)\frac{2d_2}{3d_1}c_2$	$(s-1)\min\{\frac{d_2}{d_1}c_2, d_2\}$
M_5	$(s-1)\min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{(d_2+d_1)}u\}$	$(s-1)\min\{\frac{d_2}{d_1}c_2, \frac{d_2}{d_1}u, d_2\}$
M_6	$(s-1)\min\{\frac{c_2}{2c_1}c_2, d_2\}$	$(s-1)\min\{\frac{c_2}{c_1}c_2, d_2\}$
M_7	$(s-1)\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2\}$	$(s-1)\min\{\frac{c_2}{c_1}c_2, \frac{d_2}{d_1}c_2, d_2\}$
M_8	$(s-1)\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{(d_2+d_1)}u\}$	$(s-1)\min\{\frac{c_2}{c_1}c_2, \frac{d_2}{d_1}c_2, \frac{c_2}{c_1}u, \frac{d_2}{d_1}u, d_2)\}$

$$\begin{aligned} M_1 = \quad & \{(?,?,?,?),(?,?,?,d_2),(?,?,d_1,?),(?,c_2,?,?), \\ & (?,c_2,?,d_2),(c_1,?,?,?),(c_1,?,?,d_2),(c_1,?,d_1,?)\} \\ M_2 = \quad & \{(?,?,d_1,d_2)\} \\ M_3 = \quad & \{(c_1,?,d_1,d_2)\} \\ M_4 = \quad & \{(?,c_2,d_1,?)\} \\ M_5 = \quad & \{(?,c_2,d_1,d_2)\} \\ M_6 = \quad & \{(c_1,c_2,?,?),(c_1,c_2,?,d_2)\} \\ M_7 = \quad & \{(c_1,c_2,d_1,?)\} \\ M_8 = \quad & \{(c_1,c_2,d_1,d_2)\} \end{aligned}$$

Bounds for the session problem in message passing models. c_1 and c_2 are the lower and upper bounds on step time, d_1 and d_2 are the lower and upper bounds on message delay and $u = d_2 - d_1$.

TABLE 3

"asynchronous", otherwise not. This closes an open problem in [AM90] of finding a lower bound for the (s, n)-session problem in the semi-synchronous shared memory model because S_2 is exactly the semi-synchronous shared memory model. Mavronicolas [Ma92] independently has also developed the same lower and upper bound for the model.

In the message-passing case (in Table 3), we discovered a pattern consisting of eight different groups of models. The bounds shown in the table are in a simplified form that are approximated from the original bounds with the assumption $c_2 \ll d_2 - d_1$. The precise results are discussed in Chapters 3 and 4.

Group M_1 contains all the models whose upper and lower bounds are $(s-1)d_2$, i.e., one communication delay (d_2) is required for each session. The models in M_1 behave the same way as in the asynchronous model of [AM90]. Our lower bound result for M_1 subsumes the result of [AM90] in the asynchronous model because M_1 involves a model (e.g., $(?, c_2, ?, d_2)$) that is stronger than the asynchronous model of [AM90].

When upper bound and lower bounds on message delay are known, a lower bound of $\frac{d_2}{d_2+d_1}u(s-1)$ is obtained for both M_2 and M_3 and upper bounds are $\min\{\frac{d_2}{d_1}u, d_2\}(s-1)$ in M_2 and $\min\{\frac{c_2}{c_1}u, \frac{d_2}{d_1}u, d_2\}(s-1)$ in M_3 where $u = d_2 - d_1$. These bounds suggest that when the difference between the upper bound and the lower bound on message delay is small, the system can be considered somewhat "synchronous", otherwise not because as $d_2 - d_1$ closes to zero, the upper bound is also close to zero.

The basic idea behind the algorithms for these models is that the difference ($u =$

$d_2 - d_1$) can be used to deduce the send time of messages. For example, when a process receives two messages at time t and $t + (d_2 - d_1)$ from two different processes respectively, it can deduce that the second message is sent after the first message is sent because of the bounds on message delay. Based on this idea, each process can broadcast a message at every step, and when it initially receives a message from every process, it recognizes there was a session. Then, when it receives another set of messages from every process after $d_2 - d_1$, it can recognize another session because the second set of messages must have been sent after the first session occurred. Time u can be measured, using the known lower bound (d_1 or c_1) on message delay or step time. M_2 and M_3 use d_1 to measure u and M_3 can also use c_1 to measure u.

Group M_4 consists of the model whose upper bound on step time and lower bound on message delay are known. In this model, it is possible to measure time c_2 through d_1. Intuitively, when a process receives a message from another process, it can recognize there were at least c_2/d_1 sessions since the message was sent because the minimum message delay (d_1) and the maximum step time (c_2) are known. The lower bound is $(\frac{2d_2}{3d_1}c_2(s-1))$ and the upper bound is $\min\{\frac{d_2}{d_1}c_2, d_2\}(s-1)$.

Group M_5 consists of the model whose upper bound on step time and lower and upper bounds on message delay and upper bound is known. The lower bound is $(s-1)\min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{(d_2+d_1)}u\}$ and the upper bound is $(s-1)\min\{\frac{d_2}{d_1}c_2, \frac{d_2}{d_1}u, d_2\}$. Since M_5 is stronger than M_2 and M_4, any algorithm for M_2 and M_4 can also be used in M_5. Thus, we can obtain the upper bound from the combination of those of M_2 and M_4.

Group M_6 includes the models whose lower and upper bounds on step time are known. This is similar to the known bound SM case (S_2): to know when enough sessions have elapsed, processes can count locally, depending on the relative timing between c_1 and c_2, or they can use explicit communication if the gap is too big. The lower bound is $\min\{\frac{c_2}{2c_1}c_2, d_2\}(s-1)$ and the upper bound is $\min\{\frac{c_2}{c_1}c_2, d_2\}(s-1)$. Our lower bound result for M_6 subsumes the result of [AM90] on the semi-synchronous model because the semi-synchronous model is weaker than the $(c_1, c_2, ?, d_2)$ model.

Group M_7 consists of the model whose lower and upper bound on the step time and lower bound on message delay are known. The lower bound is $(s-1)\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2\}$. The upper bound is $(s-1)\min\{\frac{c_2}{c_1}c_2, \frac{d_2}{d_1}c_2, d_2\}$. Since M_7 is stronger than M_4 and M_6, any algorithm for M_4 or M_6 will work correctly in this model. Thus, we can obtain the upper bound from the combination of those of M_4 and M_6.

Group M_8 consists of the model whose lower and upper bounds on the step time and lower and upper bounds on message delay are known. The lower bound is $(s-1)\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{(d_2+d_1)}u\}$ and the upper bound is $(s-1)\min\{\frac{c_2}{c_1}c_2, \frac{d_2}{d_1}c_2, \frac{c_2}{c_1}u, \frac{d_2}{d_1}u, d_2\}$. Since M_8 is stronger than M_1 through M_7, any algorithm that works in M_1 through M_7, also works correctly in M_8. Thus, we can obtain the upper bound from the combination of those of M_1 through M_7.

To analyze the relative strength among the models in terms of time complexity for the (s,n)-session problem, we define the following: a model A is *session-higher* than a model B, denoted $A >_{session} B$, if and only if the lower bound of A is larger than the upper bound of B. If A and B are comparable, then we say that a model

A is *higher* than B, denoted $A > B$. Note that $>_{session}$-relation and $>$-relation are closed under transitivity.

Figures 1 and 2 show the time complexity hierarchy among shared memory models and message passing models respectively. Each box represents a timing model and a line between two models represent that the model in the higher position is harder than the other in the lower position.

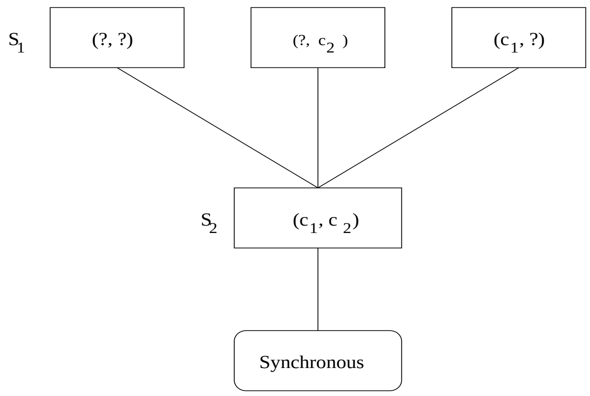

Fig. 1. A time complexity hierarchy of shared memory models.

In shared memory models , S_1 is session-harder than S_2 because the upper bound of S_2 is smaller than the lower bound of S_1. S_1 is session-harder than the synchronous model because in the synchronous model, the (s, n)-session problem can be solved in time $s \cdot c_2$. As $\log_a n$ gets bigger than $\frac{c_2}{2c_1}c_2$, the time complexity gap between S_1 and S_2 $((\log_a n - \frac{c_2}{2c_1})c_2(s-1))$ with respect to (s, n)-session problem gets closer to one communication delay per session $(\log_a nc_2)$. Since S_2 is stronger than S_1 and also session-harder than S_1, it is true that S_2 is harder than S_1, and the similar argument can be applied to the synchronous model and S_1 (see Figure 1).

For message passing models, because of some overlapping upper and lower bounds among several models, in order to compare their time complexity, we will make some assumptions on the values of relevant parameters. Clearly, M_1 is session-harder than all other models because $(s-1)d_2$ is the biggest term. The time complexity gap between M_1 and $\{M_2, M_3\}$ in solving the (s,n)-session problem can be shown to get closer to one communication delay (d_2) per session as d_1 gets closer to d_2. M_2 and M_3 can be shown session-harder than M_4 and M_5 when $c_2 < u$ $(u = d_2 - d_1)$, $c_2 < d_1$ and $c_2 < \frac{d_1 \cdot u}{d_2 + d_1}$. The time complexity gap between $\{M_2, M_3\}$ and $\{M_4, M_5\}$ with respect to the (s,n)-session problem gets closer to one communication delay (d_2) per session as $c_2 \ll d_1$ and $d_1 \ll d_2$. M_4 and M_5 can be shown session-harder than M_6, M_7 and M_8 when $c_2 < u$, $c_2 < d_1$, $c_2 < \frac{d_1 \cdot u}{d_2 + d_1}$ and $\frac{c_2}{c_2} < \frac{2d_2}{3d_1}$. The time complexity gap between $\{M_4, M_5\}$ and $\{M_6, M_7, M_8\}$ with respect to the (s,n)-session problem gets closer to $(\frac{d_2}{d_1} - \frac{c_2}{c_1})c_2$ per session as $\frac{c_2}{c_2} \ll \frac{d_2}{d_1}$. M_6, M_7 and M_8 are session-harder than the synchronous model because in the synchronous model, the (s,n)-session problem can be solved in time $s \cdot c_2$. The time complexity gap between $\{M_6, M_7, M_8\}$ and the synchronous model in solving the (s,n)-session problem can be shown to get closer to $(\frac{c_2}{c_1})c_2$ per session.

Therefore, under assumptions $c_2 < u$, $c_2 < d_1$, $c_2 < \frac{d_1 \cdot u}{d_2 + d_1}$ and $\frac{c_2}{c_2} < \frac{2d_2}{3d_1}$, it holds that $M_1 >_{session} \{M_2, M_3\} >_{session} \{M_4, M_5\} >_{session} \{M_6, M_7, M_8\} >_{session}$ the synchronous model. The time complexity hierarchy based on $>$-relation is obtained similarly as in shared memory cases (see Figure 2). For some values of parameters,

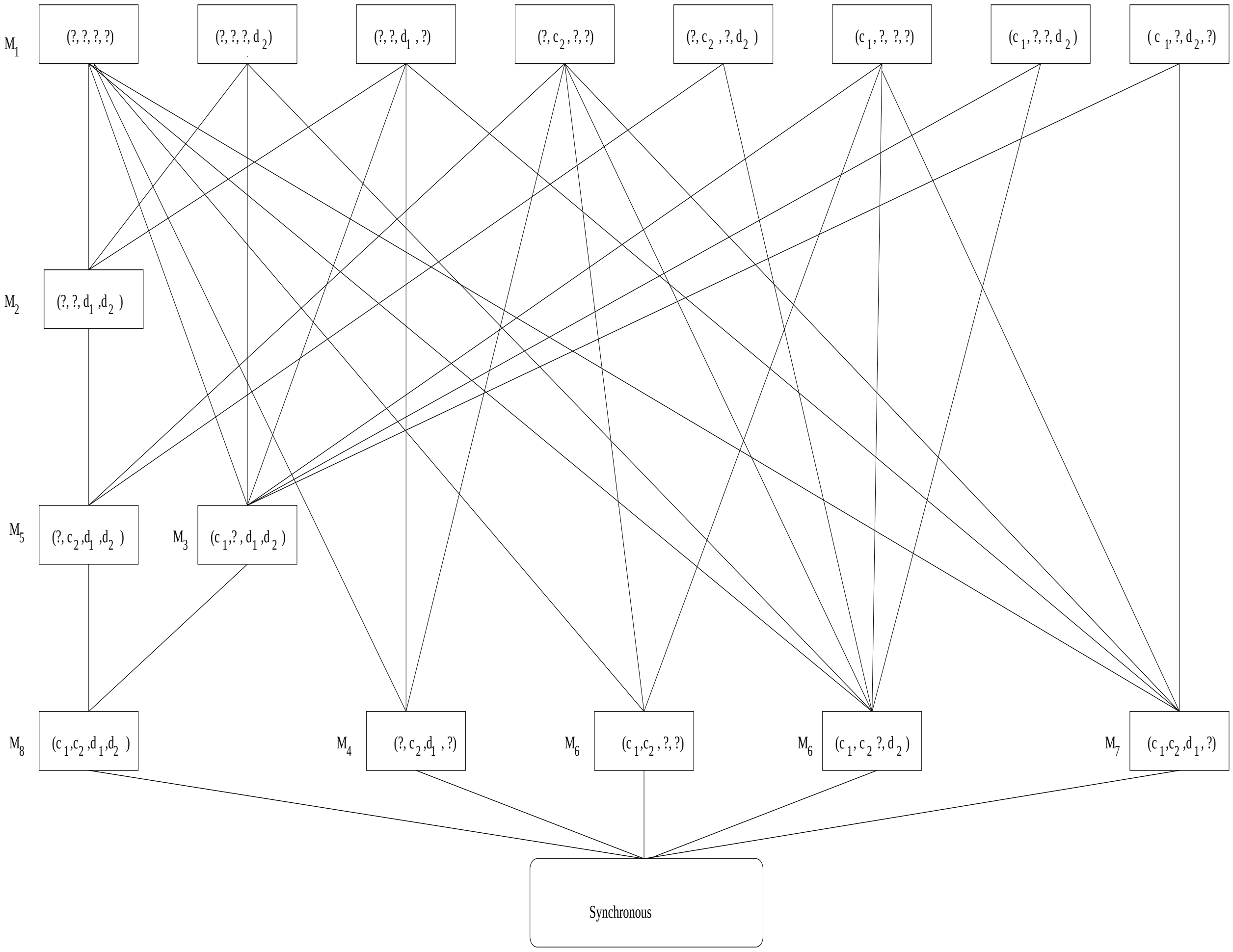

Fig. 2. A time complexity hierarchy of message passing models.

	Lower bound	Upper bound
Periodic SM model	$\max\{sc_2, \lfloor \log_{2a-1}(2n-1) \rfloor c_2\}$	$sc_2 + O(\log_a n)c_2$
Periodic MP model	$\max\{sc_2, d_2\}$	$sc_2 + d_2$

Bounds for the session problem in periodic timing models

TABLE 4

the above results indicates that the time complexity gaps among these models are not negligible and can be as large as one communication delay per session.

1.7.2 Periodic Models

In addition to the timing models considered above, we also consider *periodic models*, in which for each process there exists an unknown constant such that the process takes one step at every period of the constant. For the message passing variant, the lower and upper bounds on message delay are known. Note that the periodic shared memory model and the periodic message passing model are stronger than $(?, ?)$ and $(?, ?, d_1, d_2)$. The upper bounds for both the shared memory and message passing models are the time for the slowest process to take s steps plus the time for one communication (see Table 4). The lower bounds for both are the maximum of the time for the slowest process to take s steps and approximately the time for one communication. Our results indicate that the periodic model, which requires one communication, falls in between the synchronous model, which requires no communication, and $(?, ?, d_1, d_2)$ for the message passing variant and $(?, ?)$ for the shared memory variant, which require $s-1$ communications respectively.

CHAPTER 2

Definitions

2.1 Systems

The generalized system model definition for shared memory and message passing models is similar to that defined in [AFL83].

There are finite sets P of processes and X of shared variables. A *process* consists of a set of internal states, including an initial state. Each *shared variable* has a set of values that it can contain, including an initial value. A *global state* is a tuple of internal states of each process, and values of each shared variable. The *initial* global state contains the initial state for each process and the initial value for each shared variable. A *step* π consists of simultaneous changes to the state of some process p and the values of some number of variables $x_1, \ldots x_k$, depending on the current state of that process and current values of the variables. More formally, we represent the step π with a tuple $((s, p, r), (u_1, x_1, v_1), \ldots (u_k, x_k, v_k))$, where s and r are old and new states of a process $p \in P$; u_i and v_i are old and new values of a shared variable $x_i \in X$ for all i. We define $proc(\pi) = p$ and $var(\pi) = \{x_1, \ldots x_k\}$. We say that step π is *applicable* to a global state if p is in state s and x_i has value u_i for all i in the global state.

A *system* consists of P, X, and set Σ of possible steps. For all processes $p \in P$ and all global states g, there must exist some step in Σ involving process p that is applicable to global state g. This condition intuitively ensures that p never blocks. A *computation* of a system is a sequence of steps $\pi_1, \pi_2, \ldots$ such that: (1) π_1 is applicable to the initial global state, (2) each subsequent step is applicable to the global state resulting from the previous step, and (3) if the sequence is infinite, then every process takes an infinite number of steps. That is, there is no process failure. A *timed computation* of a system is a computation $\pi_1, \pi_2, \ldots$ together with a mapping T from positive integers to real numbers that associates a real time with each step in the computation. T must be nondecreasing and, if the computation is infinite, increase without bound.

2.1.1 Shared Memory Model (SMM)

We specialize the general system into the shared memory system in which processes communicate with each other by means of shared variables. Each step π involves only one shared variable. A process can both read and write a shared variable in a single atomic step, and we don't assume any upper bound on the size of the variables. A system is a-bounded if a is the maximum number of processes that access any single variable, in all the steps of the system, where a is some constant.

2.1.2 Message Passing Model (MPM)

We specialize the general system into the message passing system, in which processes communicate with each other by exchanging messages. P include a special process N, called the *network*. To distinguish other processes from the network, we call the rest of processes in P *regular* processes R. So, $P = R \cup \{N\}$. The network schedules the delivery of messages sent among the regular processes. $X = \{net\} \cup \{buf_p : p \in R\}$, where the values taken on by each variable are sets of messages. net models the state of the network, i.e., the set of messages in transit. buf_p holds the set of messages that have been delivered to p by the network but not yet received by p.

A step of a process p in R consists of p receiving the set M of messages in its buffer buf_p, and based solely on those message and its current state, changing its local state and sending out some message m to all the regular processes. More formally, the result of the step is to set buf_p to empty and to add (m, q) to net (i.e., receive a message), for all q in R and change state. So, the step involves two shared variables, buf_p and net. A step of N is to deliver some message of the form (m, q) in net to q. More formally, the result of the step is to remove (m, q) from net and add m to buf_q. We call this step the *delivery step* of m. Accordingly, the step also involves two shared variables, net and buf_q. We define $msg(\sigma)$ to be the message that is involved in a step σ of a process in P.

This definition of the MPM is an abstract model of a reliable strongly connected network with any topology (i.e., for every pair of processes, there exists a communication path between the two processes).

In a timed computation, each message has a *delay*, defined to be the difference between the time of the step that adds it to net and the time of the step that removes it from net. If the message is never removed, then it has infinite delay. The delay only counts the time in transit in the network and does not include the time that the recipient takes to receive the message. Note that after a message is delivered to a destination process p, p has to take at least one step to receive the message. That is, the time elapsed between the delivery step of a message m and the step of the destination process which finally removes m from the buffer is not counted toward the message delay, even if the message remains in the buffer for a long time before the recipient removes it from its buffer.

2.2 Timing Models

First we consider shared memory systems.

Let v and w be two positive real numbers with $v \leq w$. $M(v, w)$, called a *submodel*, is the set of all timed computations in which all step times are within $[v, w]$.

A shared memory model is specified by indicating whether the lower bound on step time is known; if so, what its value is; whether the upper bound on step time is known and if so, what its value is. Formally, a *shared memory model* is denoted $\mathcal{M}[c_1, c_2]$ where $c_i \in\, ? \cup \mathcal{R}^{+}$[1]; if the lower bound on step time is unknown, then $c_1 =?$; if the lower bound on step time is known, then c_1 is equal to a positive real; if the upper bound on step time is unknown, then $c_2 =?$; and if the upper bound on step time is known, then c_2 is equal to a positive real.

We define the four shared memory models of interest as follows.

$$\begin{aligned}
\mathcal{M}[c_1, c_2] &= \{M(c_1, c_2)\} \text{ if } c_1 \in \mathcal{R}^+ \text{ and if } c_2 \in \mathcal{R}^+. \\
\mathcal{M}[c_1, ?] &= \{M(c_1, w) : \forall w \text{ s.t. } w \geq c_1\} \text{ if } c_1 \in \mathcal{R}^+. \\
\mathcal{M}[?, c_2] &= \{M(v, c_2) : \forall v \text{ s.t. } v \leq c_2\} \text{ if } c_2 \in \mathcal{R}^+. \\
\mathcal{M}[?, ?] &= \{M(v, w) : \forall v, w \text{ s.t. } 0 < v \leq w\}.
\end{aligned}$$

[1] $\mathcal{R}^+$ is the set of positive reals.

We say that a timed computation α is *admissible* for a submodel $M(v,w)$ if and only if $\alpha \in M(v,w)$ and *admissible* for a model $\mathcal{M}[b_1,b_2]$ if and only if α is admissible for $M(v,w)$ for some v and w such that $M(v,w) \in \mathcal{M}[b_1,b_2]$.

We now consider message passing models.

Let v, w, x and y be four positive real numbers with $v \leq w$ and $x \leq y$. $M(v,w,x,y)$, called a *submodel*, is the set of all timed computations in which all step times are within $[v,w]$ and all message delays are within $[x,y]$.

A message passing model is specified by indicating whether the lower bound on step time is known and if so, what its value is; whether the upper bound on step time is known and if so, what its value is; and similarly for the lower and upper bounds on message delay. Formally, a message passing model is denoted $\mathcal{M}[c_1,c_2,d_1,d_2]$ where c_i and $d_i \in \{?\} \cup \mathcal{R}^+$; if the lower bound on step time is unknown, then $c_1 =?$; if the lower bound on step time is known, then c_1 is equal to a positive real; if the upper bound on step time is unknown, then $c_2 =?$; if the upper bound on step time is known, then c_2 is equal to a positive real; if the lower bound on message delay is unknown, then $d_1 =?$; if the lower bound on message delay is known, then d_1 is equal to a positive real; if the upper bound on message delay is unknown, then $d_2 =?$; and if the upper bound on message delay is known, then d_2 is equal to a positive real.

We define that the 16 message passing models of interest as follows.

$$
\begin{aligned}
\mathcal{M}[c_1,c_2,d_1,d_2] &= \{M(c_1,c_2,d_1,d_2)\} \text{ if } c_1,c_2,d_1 \text{ and } d_2 \in \cup\mathcal{R}^+. \\
\mathcal{M}[c_1,?,d_1,d_2] &= \{M(c_1,w,d_1,d_2) : \forall w \text{ s.t. } w \geq c_1\} \text{ if } c_1,d_1 \text{ and } d_2 \in \mathcal{R}^+. \\
\mathcal{M}[?,c_2,d_1,d_2] &= \{M(v,c_2,d_1,d_2) : \forall v \text{ s.t. } 0 < v \leq c_2\} \text{ if } c_2,d_1 \text{ and } d_2 \in \mathcal{R}^+. \\
\mathcal{M}[?,?,d_1,d_2] &= \{M(v,w,d_1,d_2) : \forall v,w \text{ s.t. } 0 < v \leq w\} \text{ if } d_1 \text{ and } d_2 \in \mathcal{R}^+. \\
\mathcal{M}[c_1,c_2,?,d_2] &= \{M(c_1,c_2,x,d_2) : \forall x \text{ s.t. } 0 < x \leq d_2\} \text{ if } c_1,c_2 \text{ and } d_2 \in \mathcal{R}^+. \\
\mathcal{M}[c_1,?,?,d_2] &= \{M(c_1,w,x,d_2) : \forall w,x \text{ s.t. } w \geq c_1 \wedge 0 < x \leq d_1\} \text{ if } c_1 \text{ and } d_2 \in \mathcal{R}^+.
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{M}[?, c_2, ?, d_2] &= \{M(v, c_2, x, d_2) : \forall v, x \text{ s.t. } 0 < v \leq c_2 \wedge 0 < x \leq d_1\} \text{ if } c_2, d_2 \in \mathcal{R}^+. \\
\mathcal{M}[?, ?, ?, d_2] &= \{M(v, w, x, d_2) : \forall v, w, x \text{ s.t. } 0 < v \leq w \wedge 0 < x \leq d_2\} \text{ if } d_2 \in \mathcal{R}^+. \\
\mathcal{M}[c_1, c_2, d_1, ?] &= \{M(c_1, c_2, d_1, y) : \forall y \text{ s.t. } y \geq d_1\} \text{ if } c_1, d_2 \text{ and } d_1 \in \mathcal{R}^+. \\
\mathcal{M}[c_1, ?, d_1, ?] &= \{M(c_1, w, d_1, y) : \forall w, y \text{ s.t. } w \geq c_1 \wedge y \geq d_1\} \text{ if } c_1 \text{ and } d_1 \in \mathcal{R}^+. \\
\mathcal{M}[?, c_2, d_1, ?] &= \{M(v, c_2, d_1, y) : \forall v, y \text{ s.t. } 0 < v \leq c_2 \wedge y \geq d_1\} \text{ if } c_2 \text{ and } d_1 \in \mathcal{R}^+. \\
\mathcal{M}[?, ?, d_1, ?] &= \{M(v, w, d_1, y) : \forall v, w, y \text{ s.t. } 0 < v \leq w \wedge y \geq d_1\} \text{ if } d_1 \in \mathcal{R}^+. \\
\mathcal{M}[c_1, c_2, ?, ?] &= \{M(c_1, c_2, x, y) : \forall x, y \text{ s.t. } 0 < x \leq y\} \text{ if } c_1 \text{ and } c_2 \in \mathcal{R}^+. \\
\mathcal{M}[c_1, ?, ?, ?] &= \{M(c_1, w, x, y) : \forall w, x, y \text{ s.t. } w \geq c_1 \wedge 0 < x \leq y\} \text{ if } c_1 \in \mathcal{R}^+. \\
\mathcal{M}[?, c_2, ?, ?] &= \{M(v, c_2, x, y) : \forall v, x, y \text{ s.t. } 0 < v \leq c_2 \wedge 0 < x \leq y\} \text{ if } c_2 \in \mathcal{R}^+. \\
\mathcal{M}[?, ?, ?, ?] &= \{M(v, w, x, y) : \forall v, w, y \text{ s.t. } 0 < v \leq w \wedge 0 < x \leq y\}.
\end{aligned}
$$

We say that a timed computation α is *admissible* for a submodel $M(v, w, x, y)$ if $\alpha \in M(v, w, x, y)$ and *admissible* for a model $\mathcal{M}[b_1, b_2, b_3, b_4]$ if α is admissible for $M(v, w, x, y)$ for some v, w, x and y such that $M(v, w, x, y) \in \mathcal{M}[b_1, b_2, b_3, b_4]$.

We now define the periodic timing models. A timed computation is *admissible for the shared memory periodic model* if for each process p, there exists a positive real constant c such that the time between any two consecutive step of p is exactly c.

Let d_1 and d_2 be positive real numbers. A timed computation is *admissible for the message passing periodic model* if for each process p, there exists a positive real constant c such that the time between any two consecutive step of p is exactly c and the delay of every message is between d_1 and d_2.

2.3 The (s, n)-Session Problem

We now state the conditions that must be satisfied for a system to *solve the (s, n)-session problem* for a model $\mathcal{M}$ (also called an (s, n)-session algorithm for $\mathcal{M}$).

(1) There is a distinguished set Y of n shared variables called *ports*; Y is a subset of X in the SMM and Y is the set of *buf*s in the MPM. There is a unique process in

P (in R for the MPM) corresponding to each port, which is called a *port process.*

(2) Let p be a port process which corresponds to a port y. A *port step* is any step involving p and y. A *session* is any minimal sequence of steps containing at least one port step for each port in Y.

(3) Each process in P (in R for the MPM) has a subset of special states, called *idle* states, which do not involve port. The set Σ of steps of the system guarantees that once a process is in an idle state, it always remains in an idle state.

A computation *performs s* sessions if it can be partitioned into s segments, each of which is one session. Every admissible computation for $\mathcal{M}$ must perform s sessions and all port processes are in idle states.

2.4 Time Complexity

An algorithm A in a submodel $M(x, y)$ has *running time t* if t is the maximal time, over all admissible computations of A for $M(x, y)$, until all port processes become idle. An algorithm in a model $\mathcal{M}[c_1, c_2]$ has running time t if the algorithm has running time t in its submodel $M(c_1, c_2)$. We say that a model $\mathcal{M}[c_1, c_2]$ has *upper bound t* on running time if there exists an algorithm whose running time is at most t. We say that a model $\mathcal{M}[c_1, c_2]$ has *lower bound t* if there exists no algorithm whose running time is less than t.

The running time of an algorithm in a model where the upper bound on step time is unknown is denoted in terms of a function of c_2. Intuitively, this means that in the executions in which the upper bounds on step time is c_2, the running time of the algorithm is the given function of c_2.

An algorithm in model $\mathcal{M}[c_1, ?]$ has running time $f(c_2)$ where f is a function $f : \mathcal{R}^+ \rightarrow \mathcal{R}^+$, if in every $M(c_1, y) \in \mathcal{M}[c_1, ?]$, the algorithm has running time $f(y)$. We say that model $\mathcal{M}[c_1, ?]$ has an *upper bound* $f(c_2)$ if there exists an algorithm whose running time is $f(c_2)$ (even though $\mathcal{M}$ has unknown upper bound on step time). We say that model $\mathcal{M}[c_1, ?]$ has a *lower bound* t if there exists no algorithm whose running time is $f'(c_2)$ such that for all values of c_1, $f'(c_2) < f(c_2)$.

An algorithm in model $\mathcal{M}[?, ?]$ has running time $f(c_1, c_2)$ where f is a function $f : \mathcal{R}^+ \times \mathcal{R}^+ \rightarrow \mathcal{R}^+$, if in every $M(x, y) \in \mathcal{M}[?, ?]$, the algorithm has running time $f(x, y)$. We say that model $\mathcal{M}[?, ?]$ has an *upper bound* $f(c_1, c_2)$ if there exists an algorithm whose running time is $f(c_1, c_2)$ (even though $\mathcal{M}$ has unknown upper and lower bounds on step time). We say that model $\mathcal{M}[?, ?]$ has a *lower bound* t if there exists no algorithm whose running time is $f'(c_1, c_2)$ such that for all values of c_1 and c_2, $f'(c_1, c_2) < f(c_1, c_2)$.

The time complexity definitions for message passing models are analogous to those for shared memory models.

It is clear that an upper bound (resp., a lower bound) on time complexity for a model M also applies to the models that are stronger (resp., weaker) than M. Therefore, an upper bound on time complexity for a model can be constructed from the minimum of all the running times of algorithms that work correctly in the weaker models, and a lower bound on time complexity for a model can be constructed from the maximum for all the lower bounds of the stronger models.

CHAPTER 3

The Lower Bounds

In this chapter, we address the lower bounds on the (s, n)-session problem in the shared memory timing models $\mathcal{M}[?, ?], \ldots, \mathcal{M}[c_1, c_2]$ and the message passing timing models $\mathcal{M}[?, ?, ?, ?], \ldots, \mathcal{M}[c_1, c_2, d_1, d_2]$. We give a single modular proof for all the timing models. Our lower bound proof is motivated by the proof in [AFL83] and [AM90]. We developed a general and modular lower bound proof technique for unknown bound models. Then we give corollaries for the specific models.

In the theorem, we give a sufficient condition on a set of parameters for a lower bound on the time complexity of the session problem to hold. This sufficient condition consists of a set of conditions relating to the parameters of the model (c_1, c_2, d_1 and d_2). The sufficient condition that there exist some real numbers and a function f that bear certain relationships to the parameters of the model. If this condition is true, then the lower bound on the time complexity for this model is at least f times $s - 1$. The corollaries for specific models are proved by first defining specific parameters and checking that the parameters satisfy the conditions using simple algebra.

3.1 Modular Lower Bound Proof

Theorem 3.1.1 *Let M be a timing model that satisfies the following.*

If $M = \mathcal{M}[c_1, c_2]$ is a shared memory model, then there exist positive real numbers

c, c_1', c_2' and a function f with $B = f(c_2')$ such that:

$$\begin{array}{llcll}
\text{S1.} & (B \leq (\log_a n)c_2') & \wedge & (c_1' \leq c_2') & \\
\text{S2.} & (c_1' \leq \frac{1}{2}c) & \wedge & (c_1' = c_1) & \textit{if } c_1 \neq ? \\
\text{S3.} & (c_2' \geq B\frac{c}{c_2'}) & \wedge & (c_2' = c_2) & \textit{if } c_2 \neq ?
\end{array}$$

If $M = \mathcal{M}[c_1, c_2, d_1, d_2]$ is a message passing model, then there exist positive real numbers c, c_1', c_2', d_1', d_2' and a function f with $B = f(c_2', d_2')$ such that:

$$\begin{array}{llcll}
\text{M1.} & (B \leq d_2') & \wedge & (c_1' \leq c_2') \wedge (d_1' \leq d_2') & \\
\text{M2.} & (c_1' \leq \frac{1}{2}c) & \wedge & (c_1' = c_1) & \textit{if } c_1 \neq ? \\
\text{M3.} & (c_2' \geq B\frac{c}{c_2'}) & \wedge & (c_2' = c_2) & \textit{if } c_2 \neq ? \\
\text{M4.} & (d_1' \leq (d_2' - B)\frac{c}{c_2'} + c) & \wedge & (d_1' = d_1) & \textit{if } d_1 \neq ? \\
\text{M5.} & (d_2' \geq (d_2' + B)\frac{c}{c_2'} - c) & \wedge & (d_2' = d_2) & \textit{if } d_2 \neq ?
\end{array}$$

The lower bound on time complexity of the (s, n)-session problem for M is $f(c_2)(s-1)$ if M is a shared memory model, and $f(c_2, d_2)(s-1)$ if M is a message passing model.

The basic idea of the lower bound proof is to show that if there is an algorithm that runs in time less than the claimed lower bound, the algorithm has a timed computation in which at least one process goes into an idle state even before there are s sessions. We do this by picking out a special admissible timed computation of the algorithm in which the lower bound on step is c_1' where c_1' is equal to c_1 if it is known; if it is not known, some positive real bigger than or equal to c_2'; and other bounds also are also set analogously with the lower bound on step time.

Conditions S1 and M1 specifies the relationship among parameters in obtaining the special computation. For example, because c_1', c_2', d_1' and d_2' are the lower and upper bounds on step time and message delay in the special computation, they have to obey the basic "safety" rules such that the lower bound is always less than or equal to the upper bound.

We retime and reorder this timed computation to obtain a new timed computation that contains fewer than s sessions and in which for some positive real c, the minimum step time is $\frac{1}{2}c$; the maximum step time is $B\frac{c}{c_2'}$; the minimum message delay is $(d_2' - B)\frac{c}{c_2'} + c)$; and the maximum message delay is $(d_2' + B)\frac{c}{c_2'} - c)$. To prove that this retimed computation is admissible for the model, the step times and message delays in the computation have to satisfy the timing constraints of the model. Condition S2 and M2 specify that the minimum step time should be bigger than or equal to the lower bound (c_1') on step time if the lower bound on step time is known. Condition S3 and M3 specify that the maximum step time should be less than or equal to the upper bound (c_2') on step time if the upper bound on step time is known. Condition M4 specifies that the minimum message delay should be bigger than or equal to the lower bound (d_1') on message delay if the lower bound on message delay is known. Condition M5 specifies that the maximum message delay should be less than or equal to the upper bound (d_2') on message delay if the upper bound on message delay is known.

Proof: Assume, by way of contradiction, that there exists such an algorithm A that solves the (s, n)-session problem in timing model M within time less than $f(c_2)(s-1)$ if M is a shared memory model or $f(c_2, d_2)(s-1)$ if M is a message passing model. We show that there exists an admissible timed computation of A for M which does not include s sessions.

First, we pick an admissible timed computation of A for M.

Let (α, T) be the timed computation in which regular processes take steps in round robin order and each process' ith step occurs at time $i \cdot c_2'$. Furthermore, if M is a message passing model, let all message delays be exactly d_2'.

From condition S2 and M3, $c_2' = c_2$ if $b_2 \neq ?$, which is equal to the upper bound on step time in the model; from condition M5, $d_2' = d_2$ if $b_4 \neq ?$, which is the upper bound on message delay in the model; from S1 and M1, $c_2' \geq c_1'$ and $d_2' \geq d_1'$;

We now argue that (α, T) is admissible for M. We show that step time c_2' is in the correct range. By S1 and M1, $c_1' \leq c_2'$; by S2 and M2, $c_1' = c_1$ if c_1 is known; by S3 and M3, $c_2' = c_2$ if c_2 is known. Thus c_2' is between c_1 and c_2 if c_1 and c_2 are known. If c_1 or c_2 (or both) is unknown, then there exists a submodel of M in which the lower bound and the upper bound on step time enclose c_2'.

Furthermore, if M is a message passing model, we must show that the message delay d_2' is in the correct range. By M1, $d_1' \leq d_2'$; by M4, $d_1' = d_1$ if d_1 is known; and by M5, $d_2' = d_2$ if d_2 is known; by M5, $d_2' = d_2$ if d_2 is known. Thus d_2' is between d_1 and d_2 if d_1 or d_2 (or both) is unknown, then there exists a submodel of M in which the lower and upper bounds on message delay enclose d_2'.

Let $\alpha = \beta\gamma$, where β contains all the steps that occur in (α, T) during time interval $[0, B(s-1))$ and γ contains the rest of α. Note that all the steps in α are idle steps because A solves the (s, n)-session problem in time less than $B(s-1)$.

<u>Case 1:</u> If $B \leq c_2'$, then β contains only $s-1$ sessions because each process takes

only $s-1$ steps in β. This is a contradiction since all regular processes are idle in γ.

Case 2: For the rest of the proof, assume that $B > c'_2$. For convenience of presentation, we assume that B is divisible by c'_2.[1] We will show that there is an admissible computation (β', T') for M, obtained by reordering β and retiming the steps, that results in the same global state as β, but that contains at most $s-1$ sessions. This is a contradiction.

To ensure that the retimed computation leads to the same global state, this retiming should not violate dependency of process steps. The dependency between two steps can happen if they are steps of the same processes; if one step reads a variable previously accessed by the other; or if it receives a message sent by the other. A more precise definition of dependency is given below.

We construct a partial order $\leq_\beta$ on the steps in β, representing dependency. Let $\sigma \leq_\beta \tau$ for every pair of steps σ and τ in β, and say τ is *dependent* on σ, if :

- $\sigma = \tau$, or
- $T(\sigma) < T(\tau)$ and $proc(\sigma) = proc(\tau) \neq N$, or
- $T(\sigma) < T(\tau)$ and $msg(\sigma) = msg(\tau)$ if M is a message passing model, or
- $T(\sigma) < T(\tau)$ and $var(\sigma) = var(\tau)$ if M is a shared memory model.

Close $\leq_\beta$ under transitivity.

Let $\beta = \beta_1 \ldots \beta_{s-1}$ such that each β_k, which we call *segment* k, $1 \leq k \leq s-1$, consists of all the steps in (α, T) during time interval $[(k-1)B, kB)$. Note that in a

[1]If B is not divisible by c'_2, the lower bound we obtain is $\lfloor \frac{B}{c'_2} \rfloor c'_2(s-1)$ instead of $B(s-1)$.

message passing model, no message sent in β_k is received in β_k since $B \leq d'_2$ by M1 and d'_2 is the message delay in (α, T).

We are now going to reorder steps in each segment β_k. The reordering of β_k will be done in such a way that the reordered segment contains at most one session and any two consecutive segments contain fewer than 2 sessions. We first pick one port variable for each segment in such a way that the same port variable is not picked for two consecutive segments. Let y_i be the port variable picked for segment β_i. Then, we reorder the steps of each segment, resulting in two "subsegments" such that the first subsegment doesn't contain any port event accessing y_i and the second subsegment doesn't contain any port event accessing y_{i+1}. Claim 3.1.4 proves that the reordered sequence contains only $s-1$ sessions. Figure 3 illustrates an example when $s = 4$. However, this reordering should not violate the $\leq_\beta$ relation within each

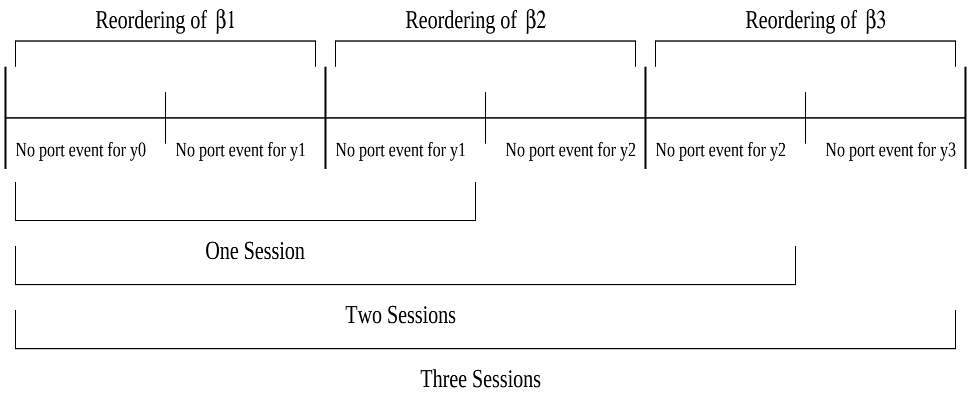

Fig. 3. An example of reordering when $s = 4$.

segment because the reordered sequence should be a computation that leads to the same state as in β. For example, in the retimed sequence, the last port step $sigma_i$

involving y_i is retimed to occur before the first port step tau_i involving y_{i-1} occurs. To prove that the reordered sequence doesn't violate the $\leq_\beta$ relation, we must show that tau_i is not dependent on $sigma_i$.

Claim 3.1.2 shows that there exists such y_i that tau_i is not dependent to $sigma_i$. Then, Claim 3.1.3 proves that the reordered sequence is consistent with the $\leq_\beta$ relation.

Claim 3.1.2 *Let y_0 be an arbitrary port in Y. For all k, $1 \leq k \leq s-1$, there exists a port variable y_k such that it is false that $\tau_k \leq_\beta \sigma_k$, where τ_k is the first step in β_k that accesses y_{k-1} and σ_k is the last step in β_k that accesses y_k.*

Proof: In a message passing model, no message sent in β_k is delivered in β_k because the size of each segment is less than d'_2 and d'_2 is the upper bound on message delay. Therefore, there is no $\leq_\beta$ relation between any two steps of different processes. Thus, any port variable that is different from y_{k-1} can be y_k.

In a shared memory model, part of the proof of Theorem 2 of [AFL83] [2] proves that there exists such y_k if each process takes fewer than $\log_a n$ steps in a segment. Because in β_k, each process takes fewer than $\log_a n$ steps, there exists such y_k in β_k.

■

Let us first assign times (a new mapping T'') to every step π in $\beta\gamma$, including all the steps of the network N if M is a message passing model, such that $T''(\pi) = T(\pi) \cdot \frac{c}{c'_2}$.

[2] The statement of Theorem 2 in [AFL83] does not explicitly include this result. For conciseness, we do not include a copy of the relevant part of the proof (the middle of pp. 453–455).

That is, every process (except N) takes a step at every c and each round occurs at every c. Since in a message passing model, the delivery steps of N are also remapped, the message delay is changed from d_2' to $d_2' \cdot \frac{c}{c_2'}$. Note that the assignment of T'' doesn't change the relative ordering of the steps in β because it only changes all the step times by the same proportion $(\frac{c}{c_2'})$.

We now reorder and assign new times (the mapping T') to every step in (β_k, T''). Intuitively, in order to obtain a timed computation as in Figure3, σ_k and all the steps that σ_k is dependent on have to move backward in time into the second half of (β_k, T'') and τ_k and all the steps that is dependent on τ_k have to move forward in time into the first half of (β_k, T'').

Let $t_k = k \cdot \frac{Bc}{c_2'}, 0 \leq k \leq s-1$. Thus, β_k occurs during $[t_{k-1}, t_k)$ under T''. Note that $t_k - t_{k-1} = \frac{Bc}{c_2'}$.

1. (Backward retiming) For each process p in P, let π be any step of p in β_k that σ_k is dependent on (in a message passing model, this means $proc(\pi) = proc(\sigma_k)$). Retime π such that $T'(\pi) = \frac{t_{k-1}+T''(\pi)}{2}$. The step is moved backward halfway to the beginning of β_k.

2. (Forward retiming) For each process p in P, let σ be any step of p in β_k that is dependent on τ_k (in a message passing model, this means $proc(\sigma) = proc(\tau_k)$). Retime σ such that $T'(\sigma) = \frac{T''(\sigma)+t_k-c}{2}$. The step is moved forward approximately halfway to the end of β_k.

3. (Stationary retiming) All other steps in β_k and γ are assigned the same times as they are under T''.

For all k, $1 \leq k \leq s-1$, let β''_k be any total ordering of β_k consistent with T', and let $\beta'' = \beta''_1\beta''_2 \ldots \beta''_{s-1}$.

If M is the message passing model, let β' be the result of changing the states of the network in β'' so that in each step of the network, the state of the network is consistent with all the send steps of regular processes and deliver steps of the network in β'' that have happened so far. If M is the shared memory model, let $\beta' = \beta''$.

Now, it is not difficult to prove the following claim:

Claim 3.1.3 *β' is a computation that leaves the system in the same global state as β does.*

Proof: We first prove that β' is consistent with $\leq_\beta$.

For any k, $1 \leq k \leq s-1$, pick any two steps, π and π' in β_k such that $\pi \leq_\beta \pi'$. Thus $T(\pi) \leq T(\pi')$ (recall that T is the original timing). We only need to prove that $T'(\pi) \leq T'(\pi')$.

Each of π and π' was retimed by one of the forward, backward or stationary retiming methods. Clearly the desired ordering is preserved:

(1) when π stays the same and π' either stays the same or moves forward, or

(2) when π' stays the same and π either stays the same or moves backward, or

(3) when π and π' both move in the same direction.

The remaining three cases cannot occur, since if π moves forward, then so does π', and if π' moves backward then so does π.

Since β' is consistent with $\leq_\beta$ and if M is a message passing model, all the states of *net* in β' are consistent with the steps of β'' by the definition of β', β' is a computation that results in the same global state as β. ■

Claim 3.1.4 *β' contains at most $s-1$ sessions.*

Proof: For all k, $1 \leq k \leq s-1$, let $h_k = T'(\sigma_k)$. We show, by induction on k, that time period $[0, h_k)$ in (β', T') contains at most $k-1$ sessions.

For the basis, $[0, h_1)$ contains no session because it does not contain any port event for y_0.

Assume, by way of induction, that $[0, h_{k-1})$ contains at most $k-2$ sessions. We prove that $[0, h_k)$ contains at most $k-1$ sessions. It is clear from the construction that $(h_{k-1}, t_{k-1}]$ contains no session because it doesn't contain any port event of y_{k-1} and similarly, $[t_{k-1}, h_k)$ doesn't contain any session because it contains no port event of y_{k-1}. Since p_{y-1} takes only one step at h_{k-1}, it is clear that $[h_{k-1}, h_k)$ contains at most one session. Therefore, $[0, h_k)$ contains at most $k-1$ sessions.

A similar argument shows that $[h_{s-1}, t_{s-1}]$ contains at most one session. Since $[0, h_{s-1})$ contains at most $s-2$ sessions, it follows that there are at most $s-1$ sessions in $[0, t_{s-1}]$. ■

Claim 3.1.5 *$(\beta'\gamma, T')$ is an admissible timed computation for M.*

Proof: By Claim 3.1.3, β' is a computation that leads to the same global state as β. Therefore, $\beta'\gamma$ is also a computation because $\beta\gamma$ is a computation. It remains to show that the timing T' conforms to the constraints of model M.

<u>Bounds on step time:</u>

Let π_i and π_{i+1} be consecutive steps of a single process, both in β_k for some k. $T''(\pi_i) = t_{k-1} + \Delta$ and $T''(\pi_{i+1}) = t_{k-1} + \Delta + c$, for some Δ.

The distance between the two steps is minimized when both steps are retimed by the same retiming method (it cannot be that π_i is retimed by the forward retiming while π_{i+1} is retimed by the backward retiming). If both π_i and π_{i+1} are retimed by the backward retiming, $T'(\pi_{i+1}) - T'(\pi_i) = c/2$. The same holds if they are retimed by the forward retiming.

Because none of the retiming methods retimes a step outside its original segment (for example, if a step is in segment β_k, after the retiming, it is in segment β'_k), neither π_i nor π_{i+1} is retimed outside the segment. Therefore, the distance between the two steps can never be larger than $t_k - t_{k-1} \leq B\frac{c}{c'_2}$.

For the step times of steps in different segments, let π_j and π_{j+1} be the consecutive steps of a process, each of which are in the different segments, say β_k and β_{k+1} respectively. $T''(\pi_j) = t_k$ and $T''(\pi_{j+1}) = t_k + c$ (this is because B is divisible by c'_2).

Since the two steps are in different segments, the distance between the two steps is minimized when π_j is retimed forward by the forward retiming while π_{j+1} is retimed backward by the backward retiming. However, neither of these retimings results in a change, i.e., $T''(\pi_j) = T'(\pi_j)$ and $T''(\pi_{j+1}) = T'(\pi_{j+1})$. Therefore, $T'(\pi_{i+1}) - T'(\pi_i) = c$.

Since the two steps are in different segment, the distance between the two steps is maximized when π_j is retimed backward by the backward retiming while π_{j+1} is retimed forward by the forward retiming. In this case, π_j moves to $(t_{k-1} + t_k - c)/2$ and π_{j+1} moves to $(t_k + t_{k+1} - c)/2$. Therefore, $T'(\pi_{i+1}) - T'(\pi_i) = t_{k+1} - t_{k-1} \leq B\frac{c}{c_2'}$.

We first check the lower bound.

If c_1 is unknown, then there exists a positive constant (namely $c/2$), that lower bounds the step time.

If c_1 is known, then we must show $c_1 \leq c/2$ because the minimum step time in β' is $c/2$ and it should be bigger than the lower bound on step time if the lower bound on step time is known. But this is true by S2 and M2 in the hypothesis of the theorem.

In either case, if c_2 is known, we must further show $c/2 \leq c_2$ because the minimum step time $(c/2)$ should be less than or equal to the upper bound on step time. By S3 (M3), in the hypothesis of the theorem, $B\frac{c}{c_2} \leq c_2$. By the assumption for Case 2, $B > c_2'$, which implies $c_2 > c$. Thus $c_2 \geq c/2$.

Now we check the upper bound.

If c_2 is unknown, then there exists a positive constant (namely $B\frac{c}{c_2'}$) that upper bounds the step time.

If c_2 is known, then we must show $c_2 \leq B\frac{c}{c_2'}$ because the maximum step time is $B\frac{c}{c_2'}$ and it should be less than or equal to the upper bound on step time if the upper bound is known. But this is true by S3 and M3.

In either case, if c_1 is known, we must further show $B\frac{c}{c_2'} \geq c_1$ because the maximum step time should be bigger than or equal to the lower bound on step time if the lower bound is known. Because $B > c_2'$ (assumption for Case 2), $B\frac{c}{c_2'} > c$. By S2 and M2, $c \geq c_1$. Thus it holds that $B\frac{c}{c_2'} \geq c_1$.

Bounds on message delay:

The step that moves the most by the backward retiming from T'' to T' is the step at the end of each segment. Let π_{last} be the step at the end of β_k. $T''(\pi_{last}) = t_k - c$. By the backward retiming, $T'(\pi_{last}) = (t_{k-1} + t_k - c)/2 = t_k - \frac{1}{2}B\frac{c}{c_2'} - c/2$. The step that moves the most by the forward retiming from T'' to T' is the steps at the start of each segment. Let π_{start} be the step at start of β_k. $T''(\pi_{start}) = t_{k-1}$. By the forward retiming, $T'(\pi_{start}) = (t_{k-1} + t_k - c)/2 = t_k - \frac{1}{2}B\frac{c}{c_2'} - c/2$. Therefore, the maximum distance that a step can move from T'' to T' is bounded by $\frac{1}{2}B\frac{c}{c_2'} - c/2$.

Let π_s and π_r be the send and receive steps of any message in β. Recall that under T'', all the message delays are $d_2'\frac{c}{c_2'}$. Thus, $T''(\pi_r) - T''(\pi_s) = d_2'\frac{c}{c_2'}$. Let $d' = d_2'\frac{c}{c_2'}$. Therefore, $d' - B\frac{c}{c_2} + c \leq T'(\pi_r) - T'(\pi_s) \leq d' + B\frac{c}{c_2} - c$ because both π_r and π_s can move $\frac{1}{2}B\frac{c}{c_2'} - c/2$ time from T' to T'', each in opposite directions.

We first check the lower bound. Note that $d' - B\frac{c}{c_2'} = (d_2' - B)\frac{c}{c_2'} > 0$ because of M1. Therefore, if d_1 is unknown, then there exists a positive constant that lower bounds the message delay because $T'(\pi_r) - T'(\pi_s) \geq d' - B\frac{c}{c_2} + c$.

If d_1 is known, then we must show $d_1 \leq d' - B\frac{c}{c_2'} + c$ because $T'(\pi_r) - T'(\pi_s)$ has to be bigger than or equal to the lower bound on message delay. But this is true by M4.

In either case, if d_2 is known, we must further show $d_2 \geq d' - B\frac{c}{c_2} + c$ because $T'(\pi_r) - T'(\pi_s)$ has to be less than or equal to the upper bound on message delay. By M5, $d_2 \geq d' + B\frac{c}{c_2'} - c$. Since $B > c_2'$ (assumption for Case 2), $B\frac{c}{c_2'} - c$ is a positive real. Thus $d_2 \geq d' + B\frac{c}{c_2'} - c > d' > d' - (B\frac{c}{c_2'} - c) = d' - B\frac{c}{c_2} + c$.

We now check the upper bound. If d_2 is unknown, then there exists a positive constant (namely $d' + B\frac{c}{c_2'} - c$) that upper bounds the message delay because $T'(\pi_r) - T'(\pi_s) \leq d' + B\frac{c}{c_2'} - c$.

If d_2 is known, then we must show $d_2 \geq d' + B\frac{c}{c_2'} - c$ because $T'(\pi_r) - T'(\pi_s) \leq d' + B\frac{c}{c_2'} - c$ and $T'(\pi_r) - T'(\pi_s)$ should be less than or equal to the upper bound on message delay (d_2). But this is true by M5.

In either case, if d_1 is known, we must further show $d_1 \leq d' + B\frac{c}{c_2'} - c$ because $T'(\pi_r) - T'(\pi_s)$ should be bigger than or equal to the lower bound on message delay (d_1). By M4, $d_1 \leq d' - B\frac{c}{c_2} + c$. As above, the result holds since $B > c_2'$. ∎

To finish the proof of the main theorem, $(\beta'\gamma, T')$ is an admissible timed computation with fewer than s sessions by Claims 3.1.4 and 3.1.5, contradicting the assumed correctness of A. ∎

3.2 SM Corollaries

For shared memory models, we prove corollaries for models $\mathcal{M}[?, c_2], \mathcal{M}[c_1, ?]$, $\mathcal{M}[c_1, c_2]$. Based on these corollaries, the lower bound for a model is derived using the fact that the lower bound for a model is also applicable to weaker models and the lower bound for a model is the maximum of all the applicable lower bounds for the model. The lower bounds for shared memory models are summarized in Table 5.

We first show the corollaries for models $\mathcal{M}[?, c_2]$, $\mathcal{M}[c_1, ?]$ and $\mathcal{M}[c_1, c_2]$.

Corollary 3.2.1 *For $M = \mathcal{M}[?, c_2]$, there exists no algorithm that solves the (s, n)-session problem within time less than $(s-1)(\log_a n)c_2$.*

Model	c_1	c_2	Lower bound
s_0	?	?	$\max\{ls_1, ls_2, ls_3\}$
s_1	?	c_2	$(s-1)(\log_a n)c_2$
s_2	c_1	?	$(s-1)(\log_a n)c_2$
s_3	c_1	c_2	$\min\{\frac{c_2}{2c_1}c_2, (\log_a n)c_2\}(s-1)$

Lower bounds for the session problem in shared memory models (ls_i means the lower bound of s_i).

TABLE 5

Proof: Clearly, f should be defined as $f(x) = x \log_a n$ (see the statement of Theorem 3.1.1). c'_2 must equal to c_2 because c_2 is known (see S3). We will define c, c'_1 and show that they satisfy the hypotheses of Theorem 3.1.1.

Let c'_1 be any constant $\leq c_2$ and $c = \frac{c_2}{\log_a n}$. Since $c'_1 \leq c_2$ and $B = f(c'_2) = c'_2 \log_a n$, S1 is satisfied. Since $B\frac{c}{c'_2} = c'_2$, S3 is satisfied. Since c_1 is unknown, we don't consider S2. ∎

Corollary 3.2.2 *For $M = \mathcal{M}[c_1, ?]$, there exists no algorithm that solves the (s,n)-session problem within time less than $(s-1)(\log_a n)c_2$.*

Proof: Clearly, f should be defined as $f(x) = x \log_a n$ and c'_1 must equal to c_1 because c_1 is known (see S2). We will define c and c'_2 and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c = 2c_1$ and c'_2 be any constant bigger than or equal to c_1. Thus, $B = f(c'_2) = (\log_a n)c'_2$ and S1 is satisfied. Since $\frac{1}{2}c = c_1$ and $c'_1 = c_1$, S2 is satisfied. Since c_2 is unknown, we don't consider S3. ∎

Corollary 3.2.3 *For* $M = \mathcal{M}[c_1, c_2]$*, there exists no algorithm that solves the* (s,n)*-session problem within time less than* $\min\{\frac{c_2}{2c_1}c_2, (\log_a n)c_2\}(s-1)$.

Proof: Clearly, f should be defined as $f(x) = \min\{\frac{x}{2c_1}c_2, (\log_a n)x\}$ and c_1' must equal to c_1 and c_2' must equal to c_2 because c_1 and c_2 are known (see S2 and S3). We will define c and show that it satisfies the hypotheses of Theorem 3.1.1.

Let $c = 2c_1$.

For S1,

$$\min\{B = f(c_2) = \frac{c_2}{2c_1}c_2, (\log_a n)c_2\} \leq (\log_a n)c_2$$

For S2,

$$c_1' = c_1 \leq \frac{c}{2}$$

For S3,

$$B\frac{c}{c_2'} = f(c_2) \cdot \frac{c}{c_2'} = \min\{\frac{c_2}{2c_1}c_2, (\log_a n)c_2\}\frac{2c_1}{c_2} \leq c_2 = c_2'$$

■

3.3 MP Corollaries

Now we show corollaries for message passing models. We prove corollaries for models $\mathcal{M}[?, c_2, ?, d_2]$, $\mathcal{M}[?, c_2, d_1, ?]$, $\mathcal{M}[?, c_2, d_1, d_2]$, $\mathcal{M}[c_1, ?, d_1, ?]$, $\mathcal{M}[c_1, ?, ?, d_2]$, $\mathcal{M}[c_1, ?, d_1, d_2]$, $\mathcal{M}[c_1, c_2, ?, d_2]$ and $\mathcal{M}[c_1, c_2, d_1, d_2]$. The lower bounds for the remaining models are obtained by taking the maximum of the lower bounds for the stronger models. The lower bounds for message passing models are summarized in Table 6.

Corollary 3.3.1 *For* $M = \mathcal{M}[?, c_2, ?, d_2]$*, there exists no algorithm that solves the* (s,n)*-session problem within time less than* $(s-1)d_2$.

	c_1	c_2	d_1	d_2	Lower bound
m_0	?	?	?	?	$\max\{lm_1, \ldots, lm_{15}\}$
m_1	?	?	?	d_2	$\max\{lm_3, lm_5, lm_7, lm_9, lm_{11}, lm_{13}, lm_{15}\}$
m_2	?	?	d_1	?	$\max\{lm_3, lm_6, lm_7, lm_{10}, lm_{11}, lm_{14}, lm_{15}\}$
m_3	?	?	d_1	d_2	$\max\{lm_7, lm_{11}, lm_{15}\}$
m_4	?	c_2	?	?	$\max\{lm_5, lm_6, lm_7, lm_{12}, lm_{13}, lm_{14}, lm_{15}\}$
m_5	?	c_2	?	d_2	$\max\{lm_7, lm_{13}, lm_{15}, (s-1)d_2\}$
m_6	?	c_2	d_1	?	$\max\{lm_7, lm_{14}, lm_{15}, (s-1)\frac{2d_2}{3d_1}c_2\}$
m_7	?	c_2	d_1	d_2	$\max\{lm_{15}, (s-1)\min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}\}$
m_8	c_1	?	?	?	$\max\{lm_9, \ldots, lm_{15}\}$
m_9	c_1	?	?	d_2	$\max\{lm_{11}, lm_{13}, lm_{14}, lm_{15}, (s-1)d_2\}$
m_{10}	c_1	?	d_1	?	$\max\{lm_{11}, lm_{14}, lm_{15}, (s-1)d_2\}$
m_{11}	c_1	?	d_1	d_2	$\max\{lm_{15}, (s-1)\frac{d_2}{d_2+d_1}u\}$
m_{12}	c_1	c_2	?	?	$\max\{lm_{13}, lm_{14}, lm_{15}\}$
m_{13}	c_1	c_2	?	d_2	$\max\{lm_{15}, (s-1)\min\{\frac{c_2}{2c_1}c_2, d_2\}\}$
m_{14}	c_1	c_2	d_1	?	$\max\{lm_{15}\}$
m_{15}	c_1	c_2	d_1	d_2	$(s-1)\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$

Lower bounds for the session problem in message passing models (lm_i means the lower bound of m_i).

TABLE 6

Proof: Clearly, f should be defined as $f(x,y)=y$. c_2' must equal to c_2 because c_2 is known (see M3) and d_2' must equal to d_2 because d_2 is known (see M5).

We will define c, c_1', d_1' and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c=\min\{\frac{c_2^2}{d_2},\frac{1}{2}c_2\}$; let c_1' be any constant less than or equal to c_2; and let d_1' be any constant less than or equal to d_2. Thus, $B=f(c_2',d_2')=d_2$. Clearly M1 is satisfied. M3 is satisfied because $B\frac{c}{c_2'}\le c_2$. M5 is satisfied because $(d_2'+B)\frac{c}{c_2'}-c< 2d_2\cdot\frac{\min\{\frac{c_2}{d_2},\frac{1}{2}c_2\}}{c_2}\le 2d_2\cdot\frac{\frac{1}{2}c_2^2}{c_2}=d_2$. Since c_1 is unknown and d_1 is unknown, we don't consider M2 and M4. ∎

Corollary 3.3.2 *For $M=\mathcal{M}[?,c_2,d_1,?]$, if $2c_2\le d_1$, then there exists no algorithm that solves the (s,n)-session problem within time less than $(s-1)\frac{2d_2}{3d_1}c_2$.*

Proof: Clearly f is $f(x,y)=\frac{2y}{3d_1}x$. c_2' must equal to c_2 because c_2 is known (see M3) and d_1' must equal to d_1 because d_1 is known (see M4).

We will define c, c_1', and d_2' and show that they satisfy the hypotheses of Theorem 3.1.1.

Let c_1' be any constant less than or equal to c_2; d_2' be any constant bigger than d_1; and $c=\frac{3d_1}{2d_2'}c_2$. Thus, $B=f(c_2',d_2')=\frac{2d_2'}{3d_1}c_2$. M1 is satisfied because $B=\frac{2d_2'}{3d_1}c_2<d_2'$ (since $2c_2\le d_1$ by hypothesis). M3 is satisfied because $B\frac{c}{c_2'}=\frac{2d_2}{3d_1}c_2\frac{\frac{3d_1}{2d_2}c_2}{c_2}=c_2$. For M4,

$$\begin{aligned}(d_2'-B)\tfrac{c}{c_2'}+c &= d_2'\tfrac{3d_1}{2d_2'}-c_2+c\\ &> \tfrac{3d_1}{2}-c_2\\ &> d_1 \text{because} 2c_2\le d_1.\end{aligned}$$

Since c_1 and d_2 are unknown, we don't consider M2 and M4. ∎

Corollary 3.3.3 *For* $M = \mathcal{M}[?, c_2, d_1, d_2]$, *if either* $d_2 \leq \frac{3d_1}{2}$ *or both* $d_2 \geq \frac{3d_1}{2} + c_2$ *and* $2c_1 \leq d_1$, *then there exists no algorithm that solves the* (s, n)*-session problem within time less than* $(s-1)\min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$, *where* $u = d_2 - d_1$.

Proof: Case 1: $d_2 \leq \frac{3}{2}d_1$.

Then $\min\{\frac{2y}{3d_1}x, \frac{y}{y+d_1}(y-d_1)\} \leq \frac{2d_2}{3d_1}c_2 = c_2$. The lower bound holds because a process has to take at least s steps to solve the (s, n) session problem.

Case 2: $d_2 \geq \frac{3d_1}{2} + c_2$ and $2c_1 \leq d_1$.

Clearly f should be defined as $f(x, y) = \min\{\frac{2y}{3d_1}x, \frac{y}{y+d_1}(y-d_1)\}$. c_2' must equal to c_2 because c_2 is known (see M3); d_1' must equal to d_1 because d_1 is known (see M4); and d_2' must equal to d_2 because d_2 is known (see M5).

If $d_2 \geq \frac{3}{2}d_1 + c_2$ and $d_1 \geq 2c_2$, we will define c, c_1' and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c = \frac{3d_1}{2d_2}c_2$ and c_1' be any constant less than or equal to c_2. Thus, $B = f(c_2', d_2') = \min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$.

For M1, $B \leq \frac{2d_2}{3d_1}c_2 < d_2$ because $2c_2 \leq d_1$.

For M3, $B\frac{c}{c_2}le\frac{2d_2}{3d_1}c_2\frac{c}{c_2} \leq c_2$.

For M4, $(d_2' - B)\frac{c}{c_2} + c > (d_2' - B)\frac{c}{c_2} \geq (d_2 - \frac{2d_2}{3d_1}c_2)\frac{c}{c_2}\frac{c}{c_2} \geq \frac{3}{2}d_1 - \frac{d_1}{2} = d_1$ because $d_1 \geq 2c_2$.

For M5, $(d_2' + B)\frac{c}{c_2} - c < (d_2' + B)\frac{c}{c_2} \leq (d_2' + \frac{2d_2}{3d_1}c_2)\frac{c}{c_2}\frac{c}{c_2} \leq d_2$ because $\frac{3}{2}d_1 + c_2 \leq d_2$.

Since c_1 is unknown, we don't consider M2. ∎

Corollary 3.3.4 *For* $M = \mathcal{M}[c_1, ?, d_1, ?]$, *there exists no algorithm that solves the* (s, n)*-session problem within time less than* $(s-1)d_2$.

Proof: Clearly f should be defined as $f(x,y)=y$. c_1' must equal to c_1 because c_1 is known (see M2) and d_1 must equal to d_1 because d_1 is known (see M4). We will define c, c_2', and d_2' and show that they satisfy the hypotheses of Theorem 3.1.1.

Let d_2' be any constant bigger than or equal to d_1; let c_2' be any constant bigger than or equal to c_1; and let $c=\max\{2c_1,d_1\}$. Thus, $B=f(c_2',d_2')=d_2'$. Clearly M1 is satisfied. Since $\frac{c}{2}=\frac{1}{2}\max\{2c_1,d_1\}>c_1$, M2 is satisfied. For M4, $(d_2'-B)\frac{c'}{c}+c>c\geq d_1$. Since c_2 and d_2 are unknown, we don't consider M3 and M5. ∎

Corollary 3.3.5 *For $M=\mathcal{M}[c_1,?,?,d_2]$, there exists no algorithm that solves the (s,n)-session problem within time less than $(s-1)d_2$.*

Proof: Clearly f should be defined as $f(x,y)=y$. c_1' must equal to c_1 because c_1 is known (see M2) and d_2' must equal to d_2 because d_2 is known (see M5).

We will define c, c_2' and d_2' and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c_2=4c_1$; let $c=2c_1$; and let d_1' be any constant less than or equal to d_2. Thus, $B=f(c_2',d_2')=d_2'$. Clearly $M1$ is satisfied. Since $\frac{c}{2}=c_1$, M2 is satisfied. For M5, $(d_2'+B)\frac{c}{c_2'}-c<(d_2'+B)\frac{c}{c_2'}=2d_2\frac{2c_1}{4c_1}=d_2$. Because c_2 and d_1 are unknown, we don't consider M3 and M4. ∎

Corollary 3.3.6 *For $M=\mathcal{M}[c_1,?,d_1,d_2]$, there exists no algorithm that solves the (s,n)-session problem within time less than $\frac{d_2}{d_2+d_1}u(s-1)$ where $u=d_2-d_1$.*

Proof: Clearly f should be defined as $f(x,y)=\frac{y}{y+d_1}(y-d_1)$. c_1' must equal to c_1 because c_1 is known (see M2); d_1' must equal to d_1 because d_1 is known (see M4); and

d'_2 must equal to d_2 because d_2 is known (see M5).

We will define c and c'_2 and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c_2 = \frac{4d_2c_1}{d_2+d_1}$ and let $c = 2c_1$. Thus, $B = f(c'_2, d'_2) = \frac{d'_2 u}{d'_2+d_1}$;. M1 is satisfied because $B < d_2$. Since $\frac{2}{2} = c_1$, M2 is satisfied.

For M4, $(d'_2 - B)\frac{c}{c'_2} + c = (d_2 - \frac{d_2 u}{d_2+d_1})\frac{2c_1}{\frac{4d_2c_1}{d_2+d_1}} + c = d_1 + c > d_1$.

For M5, $(d'_2 + B)\frac{c}{c'_2} - c = (d_2 + \frac{d_2 u}{d_2+d_1})\frac{2c_1}{\frac{4d_2c_1}{d_2+d_1}} - c = d_2 - c < d_2$.

Since c_2 is not known, we don't have to consider M3. ■

Corollary 3.3.7 *For $M = \mathcal{M}[c_1, c_2, ?, d_2]$, if $\neg(2c_1 < c_2 < 4c_1)$, then there exists no algorithm that solves the (s,n)-session problem within time less than* $\min\{\frac{c_2}{2c_1}c_2, d_2\}(s-1)$.

Proof: <u>Case 1</u>: $c_2 \leq 2c_1$.

The lower bound clearly holds because all processes have to take at least s steps to solve the (s,n)-session problem and c_2 is the upper bound on step time.

<u>Case 2</u>: $c_2 \geq 4c_1$.

Clearly, f should be defined as $f(x,y) = \min\{\frac{x}{2c_1}x, y\}$. c'_1 must equal to c_1 because c_1 is known (see M2); c'_2 must equal to c_2 because c_2 is known (see M3); and d'_2 must equal to d_2 because d_2 is known (see M2). We will define c and d'_1 and show that they satisfy the hypotheses of Theorem 3.1.1.

Let $c = 2c_1$ and let d'_1 be any constant less than or equal to d_2. Thus, $B = f(c'_2, d'_2) = \min\{\frac{c_2^2}{2c_1}, d'_2\}$. Since clearly $B \leq d_2$, M1 is satisfied. Since $\frac{c}{2} = c_1$, M2 is satisfied. Since $B\frac{c}{c'_2} = \min\{\frac{c_2^2}{2c_1}, d_2\}, \frac{2c_1}{c_2} \leq c_2$, M3 is satisfied.

For M5, there are two cases. When $\frac{c_2}{2c_1}c_2 \leq d_2$,

$$\begin{aligned}(d_2' + B)\tfrac{c}{c_2'} - c &\leq (d_2 + \tfrac{c_2}{2c_1}c_2)\tfrac{2c_1}{c_2} - 2c_1 \\ &= (2d_2)/2 - 2c_1 \;\text{ because } \tfrac{c_2}{2c_1}c_2 \leq d_2 \text{ and } c_2 \geq 4c_1 \\ &\leq d_2.\end{aligned}$$

When $\frac{c_2}{2c_1}c_2 > d_2$, it holds that $c_2 > \frac{d_2 2c_1}{c_2}$. Then, $(d_2' + B)\frac{c}{c_2'} - c < (2d_2)\frac{2c_1}{c_2} - 2c_1 < c_2 < d_2$. Thus, M5 is satisfied.

Because d_1 is unknown, we don't consider M4. ■

Corollary 3.3.8 *For $M = \mathcal{M}[c_1, c_2, d_1, d_2]$, if $\neg(\frac{3d_1}{2} < d_2 < \frac{3d_1}{2} + c_2)$ and $\neg(2c_1 < c_2 < 4c_1)$ and $2c_2 \leq d_1$, then there exists no algorithm that solves the (s,n)-session problem within time less than $\min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}(s-1)$.*

Proof: <u>Case 1</u>: $c_2 \geq \min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$.

The lower bound clearly holds because all processes have to take at least s steps to solve the (s,n)-session problem and c_2 is the upper bound on step time.

<u>Case 2</u>: $c_2 < \min\{\frac{c_2}{2c_1}c_2, \frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$.

Note that in this case, by the hypothesis of the corollary, the followings are true:

$$d_1 \geq 2c_2. \tag{3.1}$$

$$d_2 \geq \frac{3d_1}{2} + c_2. \tag{3.2}$$

$$c_2 \geq 4c_1. \tag{3.3}$$

To apply Theorem3.1.1, f should be defined as $f(x,y) = \min\{\frac{x}{2c_1}x, \frac{2y}{3d_1}x, \frac{y}{y+d_1}(y - d_1)\}$. c_1' must equal to c_1 because c_1 is known (see M2); c_2' must equal to c_2 because c_2 is known (see M3); d_1' must equal to d_1 because d_1 is known (see M4); and d_2' must equal to d_2 because d_2 is known (see M5). We will define c and show that it satisfies the hypothesis of Theorem 3.1.1.

Let $c = \max\{2c_1, \frac{3d_1}{2d_2}c_2\}$. Thus, $B = f(c_2', d_2') = \min\{\frac{c_2'}{2c_1}c_2', \frac{2d_2'}{3d_1}c_2', \frac{d_2'}{d_2'+d_1}d_2' - d_1\}$.

Case 2.A When $\frac{c_2}{2c_1}c_2 < \min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$, $B = \frac{c_2}{2c_1}c_2$ and $c = 2c_1$. As a result,

$$\frac{c_2}{2c_1} \leq \frac{2d_2}{3d_1}. \tag{3.4}$$

For M1, clearly $B = \frac{c_2}{2c_1}c_2 \leq \frac{d_2}{d_2+d_1}u < d_2$.

For M2, $\frac{c}{2} = c_1$.

For M3, $B\frac{c}{c_2'} = c_2$.

For M4,

$$\begin{aligned} (d_2' - B)\frac{c}{c_2'} + c &= d_2\frac{2c_1}{c_2} - c_2 + 2c_1 \\ &\geq d_2\frac{3d_1}{2d_2} - c_2 && \text{because of 3.4} \\ &\geq \frac{3d_1}{2} - \frac{d_1}{2} && \text{because of 3.1} \\ &= d_1. \end{aligned}$$

For M5,

$$\begin{aligned} (d_2' + B)\frac{c}{c_2'} - c &= d_2\frac{2c_1}{c_2} + c_2 - 2c_1 \\ &\leq d_2\frac{2c_1}{4c_1} + c_2 && \text{because of 3.3} \\ &< d_2/2 + \tfrac{1}{2}d_1 && \text{because of 3.1} \\ &< d_2. \end{aligned}$$

Case 2.B When $\frac{c_2}{2c_1}c_2 \geq \min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$, $B = \min\{\frac{2d_2}{3d_1}c_2, \frac{d_2}{d_2+d_1}u\}$ and $c = \frac{3d_1}{2d_2}c_2$. As a result,

$$\frac{c_2}{2c_1} > \frac{2d_2}{3d_1}. \tag{3.5}$$

For M1, $B \leq \frac{d_2}{d_2+d_1}u < d_2$.

For M2, $\frac{c}{2} = \frac{1}{2}\frac{3d_1}{2d_2}c_2 = c_1$ because of 3.5.

For M3, $B\frac{c}{c_2'} \leq \frac{2d_2}{3d_1}c_2\frac{\frac{3d_1}{2d_2}c_2}{c_2} = c_2$.

For M4,

$$\begin{aligned}
(d_2' - B)\tfrac{c}{c_2'} + c &\geq (d_2' - \tfrac{2d_2}{3d_1}c_2)\tfrac{c}{c_2'} + c \\
&= \tfrac{3d_1}{2} - c_2 + c \text{ because of 3.3} \\
&> \tfrac{3d_1}{2} - c_2 \\
&\geq \tfrac{3d_1}{2} - \tfrac{d_1}{2} \text{ because of 3.1} \\
&= d_1.
\end{aligned}$$

For M5,

$$\begin{aligned}
(d_2' + B)\tfrac{c}{c_2'} - c &\leq (d_2' + \tfrac{2d_2}{3d_1}c_2)\tfrac{c}{c_2'} - c \\
&= \tfrac{3d_1}{2} + c_2 - c \text{ because of 3.3} \\
&< \tfrac{3d_1}{2} + c_2 \\
&\leq d_2 \text{ because of 3.2.}
\end{aligned}$$

■

CHAPTER 4

The Upper Bounds

4.1 Communication in SMM

In describing our algorithms, we use a subroutine called *broadcast* as a generic operator for communication in both of the communication models.

In the MPM, the broadcasting of message m by process p is taken care of by the network. It takes at most $d_2 + c_2$ time for a message to be received by all processes in the MPM (d_2 time for a message to arrive at the destination's buffer and c_2 for the receiver to obtain the message from the buffer).

However, the communication in the SMM is constrained by the number of processes which can access a shared variable. Therefore, broadcasting in the SMM involves relaying messages from process to process by means of shared variables.

In an a-bounded shared memory system, we can build a tree network of processes and shared variables. This network can accomplish the necessary communication to propagate a piece of information originating from a process to all other processes in $O(\log_a n)$ steps.

Assuming an a-bounded shared memory system, the broadcasting can be accomplished as follows. The set of processes P contains, in addition to the port processes, some number of *relay* processes, whose job it is to read some variables and write the contents to other shared variables. These processes are configured in an $(a-2)$-ary tree of depth $\lfloor \log_{a-2} n \rfloor$ where n is the number of leaves. Every node except parents of leaves consists of a shared variable and a process. A process can access the shared variables of its parent and children as well as its own variable. The shared variables and the processes in the leaves of the tree are the port variables and the port processes respectively. The processes in non-terminal nodes are the relay processes. Each shared variable has two data fields called "in" and "out". Figure 4 shows an example.

A port process reads the *in* of its port and appends messages to its *out*; a port process only accesses one shared variable. All relay processes except the one in the root and the ones in the parents of port nodes read their *out*'s and append the contents to the *out*'s of their parents. Then they read the *in*'s of their parents and append the contents to their *in*'s. These processes include those at level 3 in Figure 4.

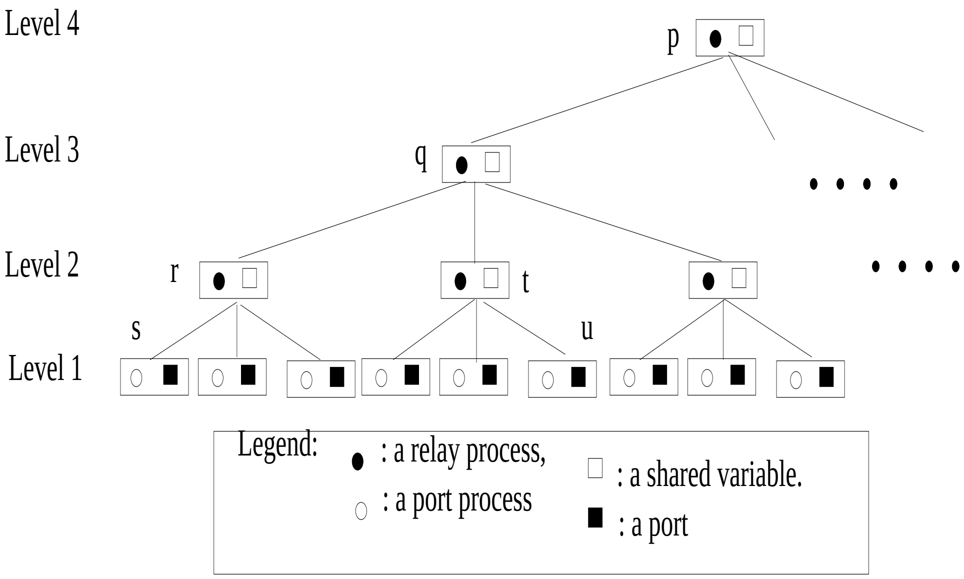

Fig. 4. An example of a tree network with depth 4 and $a = 5$

The relay processes in the parents of port nodes (which include those at level 2 in Figure 4) read the *out*'s of their child ports and append the contents to the *out*'s of their parent variables. Then they read the *in*'s of their parents and append the contents to the *in*'s of their child ports. The root process simply reads its *out* and writes the contents to its *in*. For example, in Figure 4, the information that process s writes to port is propagated to process u as follows. Process r reads the *out* of s and append the content to the *out* of process q. Then, q reads its *out* and append the content to the *out* of process p. Since p is the root of the tree, p reads its the *out* and writes the content to its *in*. Then, q reads the *in* of p and appends the content to its *in*. Process t, which is the parent of u reads the *in* of q and appends the content to *in* of t.

In the SMM, it requires $O(\log_a n)$ steps (therefore, $O(\log_a n)c_2$ time) for a piece of information (or a "message") originating from a process to be propagated to all other processes in the tree network.

In this chapter, when we say *broadcast* in the SMM, it implies all the interaction of processes in the tree network to accomplish the broadcasting. Throughout this chapter, we only describe the role of port processes in an algorithm and assume that broadcast encapsulates the interactions among port processes and other processes which participate in the tree-network communication. In addition, we use the term "step" interchangeably with "port step"; when necessary, we make the proper distinction.

4.2 Algorithms

This section provides algorithms for all the timing models discussed in this paper except the periodic models which are discussed in Chapter 5. We develop various techniques to solve the (s, n)-session problem when certain information about timing parameters for models are known. These techniques, depending on the known timing information about a model, take the form of either broadcasting certain information or counting the number of certain events such as process steps or message receptions. For example, given that a process knows when a session occurred, if an upper bound and a lower bound on the step time are known for the model in which the process runs, it is possible for the process to deduce when the next session has occurred by simply counting the number of steps that the process has taken since the last session.

We first present the various techniques for deducing whether there are s sessions in the execution of a system if a certain set of timing information for a model is known. The general strategy of each technique is, in every atomic step, to read its port variable (receive messages in message passing models); based on the values read (messages respectively), do some local computation and then write to the port variable (broadcast a message respectively).

Then, we obtain an (s, n)-session algorithm for a model simply by combining the techniques whose required information the model provides. The combination can be done in a straightforward way without increasing the time complexity of any of the techniques. One possible way of combining these techniques is to let each process simulate each step of all the techniques being combined in its single step. Suppose that there are N techniques being combined. To simulate N steps in its single step, each corresponding to one step of each technique, a process first reads its port (receive message in message passing models). Then it simulates the local computation of each step of the N techniques in a round robin order based on the previous local state of each technique and the values read (the local state of each technique can be maintained in some array). After that, it writes to its port all the information that each technique needs to write (in message passing models, broadcasts in one big message all the techniques need to broadcast in the step). This way, the process in one atomic step reads its port once; based on the read values; does some local computation and writes its port once. Clearly, this combination doesn't increase the time complexity of the techniques being combined: the amount of local computation is larger in the combined approach than in any individual technique, but the amount of local computation is ignored in measuring the time complexity (they are small in this case because there are only a small number of techniques).

We now describe our techniques. For convenience of presentation, a message is denoted $m(i, j, k)$, where i is the identifier of the sending process p_i, j is an integer in $[0, s-1]$ and k is an integer. We let $*$ be a *don't care* value for any of the fields. A port for a process i is denoted y_i. Recall that in a message passing model, y_i denotes p_i's buffer of incoming messages.

4.2.1 Technique *Asynchronous*

Technique *Asynchronous* (Figure 5), originally presented and analyzed in [AFL83] (for SM), doesn't require any timing information to solve the (s, n)-session problem. It can be used in any timing model including the asynchronous model because the correctness of the technique doesn't depend on specific step time or message delay.

```
session := 0;
while( session < s − 1 )
    msg_buf := msg_buf ∪ buf_i; /* read y_i in a shared memory model */
    if for all j ∈ [n], m(j, session, *)
    then
        session := session + 1;
    end if;
    broadcast m(i, session, *); /* write y_i in a shared memory model */
end while;
Enter an idle state.
```

Fig. 5. Technique *Asynchronous* for process i

This technique can be used for either a shared memory model or a message passing model. It solves the (s, n)-session problem in time $O(\log_a n)(s-1)$ in a shared memory model and in time $(s-1)d_2 + c_2$ in a message passing model.

The basic intuition for the technique is that since it is not using any timing information, a process has to rely only on communication with other processes at every session in order to find out whether they have taken at least one port step. In the technique, informally, each process executes a port event, broadcasts the fact to every process at each step and repeats the step until it hears that every process has executed one port event. Then it increments *session*. It repeats the above $s - 1$

```
B := c2/c1;
count := session := 0;
while( session < s - 1 )
   if (count ≥ B)
   then
        count := 0;
        session := session + 1;
   end if;
   count := count + 1;
   access y_i or buf_i;
end while;
Enter an idle state.
```

Fig. 6. Technique *Semi-synchronous* for process i

times. Then, after it executes one additional port event, it enters an idle state.

4.2.2 Technique *Semi-Synchronous*

Technique *Semi-Synchronous* (see Figure 6), originally presented and analyzed in [AM90] (for MP), requires information about an upper bound c_2 and a lower bound c_1 on step time as in the semi-synchronous models. For convenience of presentation, we assume that c_2 is divisible by c_1. T_2 can also be used for either a shared memory model or a message passing model. It solves the (s, n)-session problem in time $\frac{c_2}{c_1} c_2(s-1)+c_2$ in both a shared memory model and a message passing model.

In this technique, processes use timing information about relative step times to count locally in order to determine when enough sessions have elapsed. Informally, each process executes $\frac{c_2}{c_1}$ port events. Note that when a process takes $\frac{c_2}{c_1}$ steps, at least $\frac{c_2}{c_1} \cdot c_1$ time elapses because c_1 is the lower bound on step time. Since every process takes one port event in time c_2 (recall from Chapter 2 that c_2 is the upper bound on step time), there must be at least one session by the time that a process finishes $\frac{c_2}{c_1}$ steps. It repeats this $s-1$ times. Then after it executes one additional port event, it enters an idle state.

4.2.3 Technique *Sporadic*

So far, we have looked at the techniques that can be used for both shared memory models and message passing models. The next three techniques presented are used only for message passing models. These are new algorithms.

Technique *Sporadic* (see Figure 7) requires information about a lower bound on step time c_1, a lower bound on message delay d_1 and an upper bound on message delay d_2. This runs correctly in a model where processes run at "sporadic" rates (no upper bound on step time): processes run very slowly or fast sometimes, but they cannot run faster than a certain lower bound. For convenience of presentation, we assume that $u = (d_2 - d_1)$ is divisible by c_1.

The correctness of the technique relies on the following observation: If a process p_i receives a message m from a process p_j at time t, then the message must have been sent no later than $t - d_1$, because it takes at least d_1 time for a message to be delivered. All the messages received by p_i after $t + d_2 - d_1$ must have been sent after m was, because it takes at most d_2 time for a message to be delivered. Based on this idea, each process broadcasts a message at every step, and when it initially receives a message from every process, it recognizes there was a session because each process

```
B := u/c_1;
count := session := 0;
msg_buf := ∅;
while (session < s − 1)
    msg_buf := msg_buf ∪ buf_i;
    if (session = 0) or (count ≥ B)
    then
        if for all j ∈ [n], m(j, ∗, ∗) is in msg_buf
        then /∗ condition 1 ∗/
            count := 0;
            session := session + 1;
            msg_buf := ∅;
        end if;
    end if;
    broadcast m(i, ∗, ∗);
    count := count + 1;
end while;
Enter an idle state.
```

Fig. 7. Technique *Sporadic* for process i

must take at least one step to send the message. Then, when it receives another set of messages from every process after $d_2 - d_1$, it can recognize another session because the second set of messages must have been sent after the first session occurred. Time u can be measured, using the known lower bound (c_1) on step time (i.e., when a process takes $\frac{d_2-d_1}{c_1}$ local steps, at least u time is guaranteed to have elapsed).

Each process repeats the second procedure $s-2$ times. Then it enters an idle state after taking an additional step. The running time of *Sporadic* is $(\frac{c_2}{c_1}u + u + 2c_2)(s-2) + d_2 + 3c_2$. It takes at most $\frac{d_2-d_1}{c_1} \cdot c_2$ time to count $\frac{d_2-d_1}{c_1}$ steps; it takes at most $(u + 2c_2)$ time for a process to receive another set of message after it recognizes there was a session; it takes $(d_2 + 2c_2)$ time to receive the initial set of messages; and one more step to accomplish the last session.

Theorem 4.2.1 *Technique Sporadic solves the (s,n)-session problem within time $(\frac{c_2}{c_1}u + (u+2c_2))(s-2) + d_2 + 3c_2$ if c_1, d_1 and d_2 are known.*

Proof: Consider an arbitrary admissible timed computation C of Technique *Sporadic*. For each k, $o \le k \le s-1$, Let p_{i_k} be the first process that sets its *session* variable to k in C. To increment *session*, a process must receive a set of messages that satisfy condition 1 in Figure 7. Let M_k be the set of messages received by p_{i_k} that cause p_{i_k} to set $session_{i_k}$ to k (M_0 is the empty set); let m_k be the message which is sent last among M_k (if there is a tie, choose an arbitrary message among them).

Lemma 4.2.2 *Let π be the step which sends m_k. There are at least k sessions by the time that π occurs in C.*

Proof: We proceed by induction on k. For the basis, when $k=0$, it is always true that there are at least 0 sessions in C.

Inductively when $k>0$, assuming the lemma is true for $k-1$, we show that when π occurs, there are at least k sessions.

Let τ be the step that sent m_{k-1} and σ be the step in which $p_{i_{k-1}}$ sets $session_{i_{k-1}}$ to $k-1$. [1] For $p_{i_{k-1}}$ to update $session_{i_{k-1}}$, condition 1 in Figure 7 must hold.

Let t be the time when τ occurs and t' be the time when σ occurs. The message m_{k-1} must arrive at $buf_{i_{k-1}}$ at time between $[t+d_1, t+d_2]$ because of the bounds on

[1]Such a step exists because $session_{i_{k-1}}$ is always incremented by one.

message delay. Thus, $t' - t \geq d_1$ because σ occurs after $p_{i_{k-1}}$ receives m_{k-1}. Note that *count* in the algorithm is reset whenever *session* is updated. Let t'' be the time that p_{i_k} sets $session_{i_k}$ to k. From the code, because condition 1 should be true before *session* is updated, $count_{i_k}$ must equal to B at time t''. Thus, when $count_{i_k}$ is equal to B, at least $B \cdot c_1$ time has elapsed since time t' because t' is the time that *session* is updated to $k-1$ for the first time in computation C and $count_{i_k}$ must be reinitialized after time t. Thus, $t'' \geq t' + Bc_1 = t' + d_2 - d_1 \geq t + d_2$ because $t' - t \geq d_1$.

Therefore, the difference between times t and t'' is bigger than d_2 which is the upper bound on message delay. Thus, all messages received at time t'' or later must be sent after time t, at which time there were $k-1$ sessions by the inductive assumption. Since at least one message is sent by each process after time t, there must be at least one additional step by all processes between time t and the time π occurs. Therefore, there are at least k sessions by the time π occurs. ■

We define for each k, $2 \leq k \leq s-1$, $T_k = \max\{t : p_i \text{ sets } session_i \text{ to } k \text{ at time } t \text{ in } C \text{ for all } p_i \in R\}$. T_k is the latest time that a process sets *session* to k.

Lemma 4.2.3 *For each k, $2 \leq k \leq s-1$, $T_{k+1} \leq T_k + \frac{c_2}{c_1}u + u + 2c_2$.*

Proof: After a process p_i sets $session_i$ to k, it takes at most $\frac{u}{c_1}c_2$ time for *count* to be bigger than B. Then, it takes at most Δ time additionally for condition 1 to be true (i.e., for at least one message from every process to be received after *count* becomes bigger than B). We prove that $\Delta \leq u + 2c_2$ (see Figure 8).

Let m_1 be the message that is received from process p_j by p_i just before condition

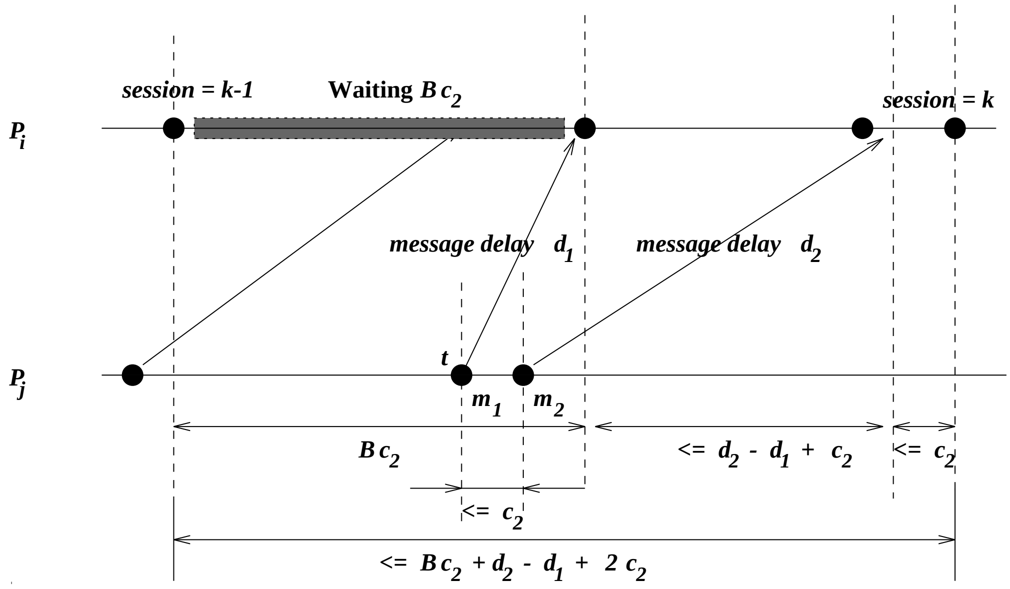

Fig. 8. Time complexity analysis for *Sporadic*.

1 becomes true in p_i (i.e., p_i has waited $\frac{u}{c_1}c_2$ time).[2] Let t be the time that m_1 is sent. p_j must broadcast another message m_2 within $t + c_2$ to process i because according to the code, all processes broadcast a message at every step. m_2 will be delivered to buf_i by time $t + c_2 + d_2$ (because it takes at most d_2 delay for a message to arrive at a buffer) and be received by time $t + 2c_2 + d_2$ (because it takes at most c_2 time for a process to take a step).

Process p_i will receive m_1 at time bigger than or equal to $t + d_1$ because it is sent at time t and it takes at least d_1 time delay for a message to arrive at a buffer. Since process i receives m_2 by time $t + 2c_2 + d_2$, the maximum time difference between the time that process i receives m_1 and the time that it receives m_2 is $(t + 2c_2 + d_2) - (t + d_1) = d_2 - d_1 + 2c_1 = u + 2c_2$. Therefore, $\Delta = u + 2c_2$. (See figure 8). ■

[2] m_1 exists because condition 1 becomes true only if there are enough messages in *msg_buf* which is emptied only after condition 1 becomes true.

From Lemmas 4.2.2 and 4.2.3, it follows that there are at least $s-1$ sessions at the time that m_{s-1} is sent. All processes will eventually set their *session*'s to $s-1$. Since all processes take additional steps after there are at least $s-1$ sessions (to receive a message), there are at least s sessions in C. By the algorithm, initially it takes at most d_2+2c_2 to receive at least one message from all processes in order to accomplish the first session. Therefore $T_1 = d_2 + 2c_1$. Using Lemma 4.2.3, T_{s-1}, which is the latest time that a process sets *session* to $s-1$, is $(\frac{c_2}{c_1}u + (u+2c_2))(s-2)+T_1$. After T_{s-1}, a process takes one step to complete s sessions. Therefore, a process enters the idle state by time $(\frac{c_2}{c_1}u + (u+2c_2))(s-2) + d_2 + 2c_2 + c_2$. ■

4.2.4 Technique *Sporadic2*

Technique *Sporadic2* (see Figure 9) requires information about a lower bound d_1 and an upper bound d_2 on message delay. As in Technique *Sporadic* after a process receives one message from every process initially and increments *session* to 1, it keeps taking steps until time $d_2 - d_1$ passes. In Technique *Sporadic*, time $u = d_2 - d_1$ is measured by counting local steps. This is possible because the lower bound c_1 on step time is known. However, Technique *Sporadic2* utilizes the lower bound d_1 on message delay to measure the time. For example, when a process p_i broadcasts a message at time t, it will be received by p_j no earlier than time $t + d_1$. In *Sporadic2*, each message carries the value of a variable called *count*, which is set to zero after *session* is incremented. When a process receives a message with *count* value equal to some integer that is bigger than its own *count* value, it sets its *count* to the value in the

```
B := u/d_1;
count := session := 0;
msg_buf := ∅;
while( session < s − 1 )
    msg_buf := msg_buf ∪ buf_i;
    if (session = 0) or (count ≥ B)
    then
        if for all j ∈ [n], m(j, ∗, ∗) is in msg_buf
        then /∗ condition 1 ∗/
            count := 0;
            session := session + 1;
            msg_buf := ∅;
        end if;
    end if;
    if there is any m(∗, session, ∗) in msg_buf
    then count := max{count, k : m(∗, session, k) ∈ msg_buf };
    broadcast m(i, session, count + 1);
end while;
Enter an idle state.
```

Fig. 9. Technique *Sporadic2* for process i

message and broadcasts another message with $count + 1$. This guarantees that when a process receives a message with *count* equal to k, at least time kd_1 has elapsed since the last update to *session*. In the code, it increments *count* until it becomes equal to u/d_1 to know whether time u has passed since the last time that *session* was incremented. Then when it receives another set of messages from every process, it can increment *session* again because the second set of messages must be sent after the previous session occurred (as in Technique *Sporadic*). The running time of *Sporadic2* is approximately $(\frac{d_2}{d_1}u)(s-1)$.

Theorem 4.2.4 *Technique Sporadic2 solves the (s,n)-session problem within time $(\frac{d_2}{d_1}u + u + 2c_2)(s-2) + d_2 + 2c_2$ if d_1 and d_2 are known.*

Proof: Technique *Sporadic2* differs from Technique *Sporadic* only in the way that *count* is incremented and in that B is set to u/d_1. The rest of the code is same. The correctness proof is similar to that of Technique *Sporadic*.

Since *count* and B affect condition $(count \geq B)$ in the code, we only need to prove that when $B < count$, at least time u has passed since the last time $session$ was incremented. *count* is incremented to k only when a process receives $m(j, session, k-1)$ for any j and for some value of $session$. When we apply this argument inductively, we prove that there must be a chain of processes $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ where p_{i_r} receives $m(i_{r-1}, session, r-1)$ from $p_{i_{r-1}}$. Thus, when *count* is B, at least time $Bd_1 = d_2 - d_1 = u$ has passed after p_{i_1} sent $m(i_1, session, 1)$ because it takes at least d_1 message delay for a message to be received after it is sent.

For time complexity, it takes at most time d_2+c_2 for p_{i_k} to receive $m(i_{k-1}, session, k-1)$ after $p_{i_{k-1}}$ receives $m(i_{k-2}, session, k-2)$. Therefore, for *count* to be bigger than or equal to B, it takes at most $B(d_2+c_2)$. After that, for condition 1 in the code to be true it takes at most $(u+2c_2)$ as it is proved in the proof of Lemma 4.2.3. Therefore, the running time of Technique *Sporadic2* is $(\frac{d_2+c_2}{d_1}u + u + 2c_2)(s-2) + d_2 + 2c_2$. ■

4.2.5 Technique *Inverse-sporadic*

Technique *Inverse-sporadic* (see Figure 10) can be used if an upper bound c_2 on step time and a lower bound d_1 on message delay are known. Since the lower bound

$B := \frac{c_2}{d_1}(s-1);$
$count := 0;$
$msg\_buf := \emptyset;$
while($count < B$)
  $msg\_buf := msg\_buf \cup buf_i;$
  $count := \max\{k, count : m(*,*,k) \in msg\_buf\ \};$
  $broadcast\ m(i,*,count+1);$
end while;
Enter an idle state.

Fig. 10. Technique *Inverse-sporadic* for process i

on message delay is known, we can deduce that when a process receives a message, at least d_1 time has passed after it is sent. That c_2 is known makes it possible to estimate how many steps processes take within time d_1 (at least $\frac{d_1}{c_2}$ steps). Therefore, we can deduce that if a message sent after a session has occurred is received by a process, it is true that at least $\frac{d_1}{c_2}$ sessions have occurred since the last session. The running time of *Inverse-sporadic* is approximately $\frac{d_2}{d_1}c_2(s-1)$ assuming $d_2 \gg u$.

Theorem 4.2.5 *Technique Inverse-sporadic solves the* (s,n)*-session problem in time* $\frac{d_2+c_2}{d_1}c_2(s-1)+c_2$ *if* c_2 *and* d_1 *are known.*

Proof: *count* is incremented to k only when a process receives $m(j,*,k-1)$ for any j. When we apply this argument inductively, we prove that there must be a chain of processes $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ where p_{i_r} receives $m(i_{r-1},*,r-1)$ from $p_{i_{r-1}}$. Thus, when *count* is B, at least time $Bd_1 = \frac{c_2}{d_1}(s-1)d_1$ has passed after p_{i_1} sent $m(i_1,*,1)$ because it takes at least d_1 message delay for a message to be received after it is sent. Thus, when $count > B$, at least time $c_2(s-1)$ has passed since the start of the computation.

Technique	Timing Information
Asynchronous	None
Semi-Synchronous	c_1, c_2
Sporadic	c_1, d_1, d_2
Sporadic2	d_1, d_2
Inverse-sporadic	c_2, d_1

Timing information that each technique assumes

TABLE 7

The theorem follows. ∎

4.2.6 Upper Bounds for Models

Table 7 lists the timing information that each technique assumes. As noted above, a technique that runs correctly in one model also runs correctly in stronger models (i.e., the models that provides more timing information than the model). We obtain an algorithm for the (s, n)-session problem in a model by combining the various techniques described in the first part of this section. An upper bound on time complexity of the (s, n)-session problem for a model is the minimum of the running times over all the combined techniques. Tables 8 and 9 summarize our upper bounds for all models.

Shared memory models $\mathcal{M}[?, ?]$, $\mathcal{M}[?, c_2]$, $\mathcal{M}[c_1, ?]$ can use Technique *Asynchronous* to solve the (s, n)-session problem in time $(s-1)O(\log_a n)c_2$ because Technique *Asynchronous* doesn't require any timing information. Shared memory model $\mathcal{M}[c_1, c_2]$ can use both Techniques *Asynchronous* and *Semi-Synchronous* because because Technique *Asynchronous* doesn't require any timing information and Technique *Semi-*

	c_1	c_2	Upper bounds
s_0	?	?	$(s-1)O(\log_a n)c_2$
s_1	?	c_2	u_0
s_2	c_1	?	u_0
s_3	c_1	c_2	$\min\{\frac{c_2}{c_1}c_2(s-1), u_0, u_1, u_2\}$

Upper bounds for the session problem in shared memory models (u_i means the upper bound of s_i).

TABLE 8

	c_1	c_2	d_1	d_2	Upper bounds
m_0	?	?	?	?	$(s-1)(d_2+c_2)+c_2$
m_1	?	?	?	d_2	u_0
m_2	?	?	d_1	?	u_0
m_3	?	?	d_1	d_2	$\min\{u_0, u_1, u_2, (\frac{d_2+c_2}{d_1}u + u + 2c_2)(s-2) + d_2 + 3c_2\}$
m_4	?	c_2	?	?	u_0
m_5	?	c_2	?	d_2	$\min\{u_0, u_1, u_4\}$
m_6	?	c_2	d_1	?	$\min\{u_0, u_2, (\frac{d_2+c_2}{d_1}c_2)(s-1)+c_2\}$
m_7	?	c_2	d_1	d_2	$\min\{u_0, \ldots, u_6\}$
m_8	c_1	?	?	?	u_0
m_9	c_1	?	?	d_2	$\min\{u_0, u_1, u_8\}$
m_{10}	c_1	?	d_1	?	$\min\{u_0, u_2, u_8\}$
m_{11}	c_1	?	d_1	d_2	$\min\{u_0, \ldots, u_3, u_8, u_9, u_{10}, (\frac{c_2}{c_1}u + u + 2c_2)(s-2) + d_2 + 3c_2\}$
m_{12}	c_1	c_2	?	?	$\min\{u_0, u_4, u_8, \frac{c_2}{c_1}c_2(s-1)+c_2\}$
m_{13}	c_1	c_2	?	d_2	$\min\{u_0, u_1, u_5, u_8, u_9, u_{12}\}$
m_{14}	c_1	c_2	d_1	?	$\min\{u_0, u_2, u_4, u_6, u_8, u_{10}, u_{12}\}$
m_{15}	c_1	c_2	d_1	d_2	$\min\{u_0, \ldots, u_{14}\}$

Upper bounds for the session problem in message passing models (u_i means the upper bound of m_i).

TABLE 9

Synchronous requires knowledge of c_1 and c_2. Thus, this model can solve the (s, n)-session problem in time $\min\{\frac{c_1}{c_2}c_2, O(\log_a n)\}(s-1) + c_2$.

Message passing models $\mathcal{M}[?, ?, ?, ?]$, $\mathcal{M}[?, ?, ?, d_2]$, $\mathcal{M}[?, ?, d_1, ?]$, $\mathcal{M}[?, c_2, ?, ?]$, $\mathcal{M}[?, c_2, ?, d_2]$, $\mathcal{M}[c_1, ?, ?, ?]$, $\mathcal{M}[c_1, ?, ?, d_2]$ and $\mathcal{M}[c_1, ?, d_1, ?]$ use only Technique *Asynchronous* to solve the (s, n)-session problem in time $(s-1)(d_2 + c_2) + c_2$.

Message passing model $\mathcal{M}[?, ?, d_1, d_2]$ uses techniques *Asynchronous* and *Sporadic2* to solve the (s, n)-session problem with the time complexity that is the minimum of those of Techniques *Asynchronous* and *Sporadic2*.

Message passing model $\mathcal{M}[?, c_2, d_1, ?]$ uses techniques *Asynchronous* and *Inverse-sporadic* to solve the (s, n)-session problem in time $\min\{\frac{d_2}{d_1}c_2, d_2 + c_2\} + c_2$.

The (s, n)-session algorithm for message passing model $\mathcal{M}[?, c_2, d_1, d_2]$ uses techniques *Asynchronous*, *Sporadic2* and *Inverse-sporadic* to have the time complexity that is the minimum of those of Techniques *Asynchronous*, *Sporadic2* and *Inverse-sporadic*.

Message passing model $\mathcal{M}[c_1, ?, d_1, d_2]$ can use techniques *Asynchronous*, *Sporadic* and *Sporadic2* to solve the (s, n)-session problem with the time complexity that is the minimum of those of Techniques *Asynchronous*, *Sporadic* and *Sporadic2*.

Message passing models $\mathcal{M}[c_1, c_2, ?, ?]$ and $\mathcal{M}[c_1, c_2, ?, d_2]$ use Techniques *Asynchronous* and *Semi-synchronous* to solve the (s, n)-session problem in time $\min\{\frac{c_2}{c_1}c_2, d_2 + c_2\}(s-1) + c_2$.

The (s, n)-session algorithm for message passing model $\mathcal{M}[c_1, c_2, d_1, ?]$ uses Techniques *Asynchronous*, *Semi-synchronous* and *Inverse-sporadic* to have the time complexity that is the minimum of those of Techniques *Asynchronous*, *Semi-synchronous* and *Inverse-sporadic*.

Message passing model $\mathcal{M}[c_1, c_2, d_1, d_2]$ can use Techniques *Asynchronous*, *Semi-synchronous*, *Sporadic*, *Sporadic2* and *Inverse-sporadic* to solve the (s, n)-session problem with the time complexity that is the minimum of those of Techniques *Asynchronous*, *Semi-synchronous*, *Sporadic*, *Sporadic2* and *Inverse-sporadic*.

CHAPTER 5

The Periodic Model

The periodic model and the synchronous model are similar in that a process takes steps at regular time intervals, yet they differ from each other in that different processes can take steps at different speeds and speeds are not known. We first present an algorithm A_{per} for the (s, n)-session problem in this model, whose running time is $sc_2 + d_2$ in MP and $sc_2 + O(\log_a n)c_2)$ in SM. We show that for all periodic algorithms that solve the (s, n)-session problem, there exists a computation which takes at least $\max\{s \cdot c_2, d_2\}$ time for the MP and $\max\{s, \lfloor \log_a n \rfloor\} \cdot c_2$ for the SM. In the periodic model c_2 denotes the maximum step time of all processes.

5.1 Algorithm

> Algorithm A_{per}: (This algorithm runs in the MPM and the SMM.) Each port process accesses its own port $s - 1$ times. At its $s - 1$th step, broadcasts the fact that it has finished its $(s - 1)$th step, and keeps taking port steps until it hears that all other processes have taken $s - 1$ steps. Then it takes one more port step and enters an idle state.

Theorem 5.1.1 *$A(p)$ solves the (s, n)-session problem in time $s \cdot c_2 + d_2$ in the MPM and time $s \cdot c_2 + O(\log_a n) \cdot c_2$ in the SMM.*

Proof: Let p_i be the port process with the maximum step time c_2. By the definition of the periodic model, it is clear that s sessions have occurred and no process is yet idle by the time that p_i takes its $s-1$ th step and broadcasts its message. It takes at most $s \cdot c_2$ time for p_i to take s steps; every process receives p_i's message and becomes idle by time $s \cdot c_2 + d_2$ in the MPM and time $s \cdot c_2 + O(\log_a n) \cdot c_2$ in the SMM.

∎

5.2 Lower Bounds

Theorem 5.2.1 *No MP periodic algorithm for the (s,n)-session problem runs in time less than* $\max\{s \cdot c_2, d_2\}$.

Proof: Suppose that $s \cdot c_2 \geq d_2$. Since all processes must take at least s steps to have s sessions, $s \cdot c_2$ is obviously the lower bound.

Suppose that $s \cdot c_2 < d_2$. Assume, by way of contradiction, that there is an algorithm A that solves the (s,n)-session problem in time less than d_2. We prove that there exists an infinite admissible computation of A that contains fewer than s sessions.

Let (α, T) be any admissible timed computation of A in which all the message delays are d_2. We will perturb (α, T) in order to get a new admissible timed computation (α', T'). We will prove that there exists at least one port process in (α', T') which enters an idle state before another port process takes any step, resulting in an admissible computation that contains fewer than s sessions.

Fix a port process p' and change the step time of p' to be d_2, i.e., steps of p' occur every d_2 time units. Run A with this modified process and the rest of processes to get a new timed admissible computation (α', T'). This is admissible because the upper bound on step time is unknown.

Since no process has received any message before time d_2 in α (because all message delays are d_2), all processes except p' will have the same local states as in α before time d_2. Since if a process enters an idle state, it remains in that state forever and all processes except p' must be in idle states before p' takes any step in (α', T'), (α', T) is an admissible timed computation that contains fewer than s sessions. This is a contradiction.

∎

Theorem 5.2.2 *No SM periodic algorithm for the (s, n)-session problem runs in time less than $max\{s \cdot c_2, \lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2\}$.*

Proof: Suppose that $s \cdot c_2 \geq \lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$. Since all processes must take at least s steps to have s sessions, $s \cdot c_2$ is obviously the lower bound.

Suppose that $s \cdot c_2 < \lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$. By way of contradiction we assume that there exists an algorithm A that solves the (s, n)-session problem in the periodic SMM in time Z strictly less than $\lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$, where Z is some constant. We prove that there exists an infinite admissible computation of A that contains fewer than s sessions.

Let (α, T) be the admissible timed computation in which processes take steps in round robin order and each process's ith step occurs at time $i \cdot c_2$. Each consecutive group of steps for p_1 through $p_{|P|}$ is a round. (Round i occurs at time $i \cdot c_2$ and consists of the i th step of each process.) Since all (port) processes should enter idle states by time Z in (α, T) and all the step time periods are equal to c_2 in (α, T), there are at most $r = \lfloor Z/c_2 \rfloor$ rounds required until termination in (α, T).

We will perturb (α, T) in order to get a new admissible timed computation (α', T'). We will prove that there exists at least one port process in (α', T') that enters an idle state before another port process takes any step, resulting in an admissible computation that contains fewer than s sessions.

Fix any port process p' and change the step time of p' to be $\lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$, i.e., the steps of p' occur every $\lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$ time units. Note that the step time here is bigger than Z. Thus, if we prove that there exists at least one port process in (α', T') that enters an idle state before p' takes a step (i.e., before time $\lfloor \log_{2a-1}(2n-1) \rfloor \cdot c_2$), we prove the theorem. Run A with this modified process and the rest of processes to get a new timed admissible computation (α', T'). It is admissible since the upper bound on step time is unknown.

Since the steps of p' occur at different times in (α', T') than they do in (α, T), there will be other steps that are influenced by the steps of p'. These steps will also influence others. We say that these steps are "contaminated" by the steps of p'. However, we prove that there is at least one process p whose steps are not contaminated through time Z. Since at time Z in (α', T'), all processes enter idle states, the process is also in an idle state at time Z in (α', T'). However, there has not been even a single step by time Z in (α', T'). because p' hasn't taken a step yet. This is a contradiction. We now formalize these ideas.

We define a *subround* to be a minimal computation fragment of α' that involves all processes except p'. A variable v is *contaminated* in subround k of α' if there exists $j \leq k$ and process $p \neq p'$ such that v's value in the global state of α' following p's step in subround j is not equal to v's value in the global state of α following p's step in round j. We define no variable to be contaminated in subround 0. A process p is *contaminated* in subround k of α' if $p \neq p'$ and there exists $j \leq k$ such that in subround j of α', p accesses a variable that is contaminated in subround j. We define no processes to be contaminated in subround 0.

Let $P(t)$ be the set of processes that are contaminated in subround t of α', and let $V(t)$ be the set of variables that are not contaminated in subround $t-1$ but are contaminated in subround t of α'. The sizes of $P(t)$ and $V(t)$ depend on the algorithm A, which has been chosen arbitrarily. Thus we define two other quantities, P_t and V_t, and analyze them. We show that P_t upper bounds $|P(t)|$ and V_t upper bounds $|V(t)|$. Define P_t and V_t to satisfy the recurrence equations:

$$P_0 = 0 \qquad V_0 = 0 \tag{5.1}$$

$$V_t = 2 \cdot P_{t-1} + 1 \tag{5.2}$$

$$P_t = (a-1) \cdot V_t + P_{t-1}. \tag{5.3}$$

Lemma 5.2.3 *$|P(t)| \leq P_t$ and $|V(t)| \leq V_t$ for $0 \leq t \leq r$, where $r = \lfloor Z/c_2 \rfloor$.*

Proof: We prove the lemma by induction on t. The key points are that p' contributes at most one variable to $V(t)$ in each subround, while each contaminated process contributes at most two. Also, in the worst case a process becomes contaminated as soon as possible, processes only become contaminated due to the variables that just become contaminated, and each variable contaminates at most $a-1$ other processes.

For the basis ($t = 0$), no process or variable is contaminated in subround 0 by definition.

Assume that the lemma is true for $t - 1 \geq 0$. Since p' does not take a step within subround t, p' contributes at most one variable to $V(t)$ (by failing to access a variable it was supposed to) and every process that is contaminated in subround $t-1$ contributes at most two variables to $V(t)$ (by failing to access a variable it was supposed to and accessing a variable it wasn't supposed to).

By the inductive hypothesis, there are at most P_{t-1} processes that are contaminated in subround $t-1$. Thus $|V(t)| \leq 2 \cdot P_{t-1} + 1 = V_t$.

In the worst case, a process becomes contaminated as soon as possible. Thus we assume that if a variable v becomes contaminated in subround i, for any i, then the remaining $a-1$ processes that can access v do so in subround i and become contaminated. As a result, processes only become contaminated in a subround due to variables that become contaminated in the same subround. So,

$$|P(t)| \leq |P(t-1)| + (a-1) \cdot |V(t)|.$$

By the inductive hypothesis and the above argument for $V(t)$,

$$|P(t)| \leq P_{t-1} + (a-1) \cdot V_t.$$

■

Substituting 5.2 into 5.3 yields

$$P_t = (2a - 1) \cdot P_{t-1} + a - 1. \tag{5.4}$$

The solution to 5.4 with initial condition 5.1 is

$$P_t = \frac{(2a-1)^t - 1}{2}$$

(This result can be obtained by the technique of characteristic equations or verified by induction on t.)

We now show that the total number of processes that are contaminated in subround r is at most $n - 1$.

$$\begin{aligned} P_r &= \frac{(2a-1)^r - 1}{2} \\ &< \frac{(2a-1)^{\lfloor \log_{2a-1}(2n-1) \rfloor} - 1}{2} \\ &\leq \frac{(2n-1) - 1}{2} \\ &= n - 1. \end{aligned}$$

Since fewer than n processes are contaminated in subround r, at least one port process $p \neq p'$ is in the same state at the end of subround r in α' as it is at the end of round r in α — an idle state. But p' has not taken a step yet. Thus (α', T') is an admissible timed computation that contains fewer than s sessions. Contradiction.

■

CHAPTER 6

Conclusion and Future Study

We showed a general time complexity hierarchy of various timing models for shared memory and message passing distributed systems. These timing models lie between the synchronous and asynchronous models. Each timing model is characterized by known or unknown lower and upper bounds on process step time, and, in the message passing model, message delay. This gave us four shared memory models and 16 message passing models. We also considered other timing models that are not covered by this framework, called periodic models.

The time complexity hierarchy was shown by developing upper and lower bounds on the time complexity for the (s, n)-session problem in each timing model. We obtained almost matching upper and lower bounds. We also presented a general and modular lower bound proof technique for the (s, n)-session problem in all the timing models. This proof identified sufficient conditions for a lower bound in a timing model to hold in solving the (s, n)-session problem.

The time complexity hierarchy among any two models implies that there exists some problem that can be solved faster in one model than in the other model, but no such problem exists for the reverse. This has applications in showing that any simulation of a stronger model with a weaker model requires some non-constant time complexity overhead.

For future study, there are some timing models, even though they are comparable, among which we are not able to show the time complexity hierarchy using the session problem. For example, the asynchronous model and the models in M_1 are comparable, but the time complexity for solving the session problem are the same. We are interested in finding any problem that can show the time complexity difference among those models.

For some timing models, we are not able to show the tight time complexity bound for solving the session problem using our lower and upper bound techniques. We are interested in seeing some works that can close such gaps. This will contribute to eliminating the assumptions we made to develop the general time complexity hierarchy.

We are also interested in applying our modular lower bound proof technique to other distributed computing problems such as the mutual exclusion problem and the dining philosophers problem. It might be used to show lower bounds on the time complexity for solving those problems in many different timing models.

PART II

Resource Allocation

This part concerns resource allocation in distributed message passing systems, i.e., the scheduling of accesses to resources shared by many concurrent processes. Three different kinds of the resource allocation problem with varying degrees of generality are considered: *the dining philosophers problem* [Di71], *the drinking philosophers problem* [CM84] and *the dynamic resource allocation problem* [AS90].

In this part of the dissertation, a tight bound on the response time for the dining and drinking philosophers problems is presented. These problems can be represented by a conflict graph in which a node represents a process and an edge represents a sharing of resources between two processes.

We show a lower bound of $\Omega(\chi \cdot (c + d))$ on the response time for the dining and drinking philosophers problems where χ is the chromatic number [1] of the given conflict graph, c is the maximum time that a user uses a resource and d is the maximum message delay. We prove that this bound is tight by giving a distributed dining philosophers algorithm with response time $\Omega(\chi \cdot (c + d))$. Unfortunately, this

[1]The minimum number of colors required to color the graph.

algorithm requires a large amount of local computation because it uses an optimal coloring algorithm as a subroutine.

However, we also show this cost is inevitable (unless P=NP). We prove that any dining philosophers algorithm with response time $\Omega(\chi \cdot (c + d))$ is transformable in polynomial time into a sequential algorithm for an NP-complete problem. This indicates the amount of computation (including the local computation) is very large (possibly exponential).

These results provide motivation for considering other distributed dining philosophers algorithms with non-optimal response time, but with a polynomial amount of local computation. We present an efficient modular resource allocation algorithm that uses any arbitrary resource allocation algorithm as a subroutine. It improves the performance of the subroutine by letting each process wait only for its currently conflicting processes, and therefore, allows more concurrency.

To be specific, consider an arbitrary resource allocation algorithm with response time $O(X \cdot (c + d))$ and message complexity $O(M)$ for some X and M (intuitively, X and M can be a function of the number conflicting process, e.g., δ). If we use the algorithm as a subroutine, the overall response time of the resulting algorithm is $O(\delta c + \max\{\delta^2, X\}d)$ and its message complexity is $\max\{M, O(r^2\delta)\}$, where r is the number of shared resources that a process requests and δ is the maximum degree of the conflict graph (i.e., the maximum number of processes that can be in conflict). [2].

Since c is only multiplied by δ, our algorithm reduces the impact of c on the worst

[2] δ, r, c and d are treated as constants here to measure the time complexity of algorithms, but in real systems they are variables.

case response time when X is bigger than $O(\delta)$. Note that c can often be very large depending on the applications. Reducing the impact of c on the response time can improve the performance of a system.

For appropriate choices of the subroutine, among all the known (distributed) resource allocation algorithms, our algorithms give the fastest worst case response time.

We also have done simulation studies that indicate our algorithm performs faster than other previously known algorithms on average, especially when the number of conflicting processes are large and the time period that a process uses a resource is big.

This part is organized as follows. Chapter 7 introduces the resource allocation problem and discusses previous results and our results concerning resource allocation in distributed systems. Chapter 8 gives the formal definition of the system model and the resource allocation problem. Chapter 9 presents our lower bound result on the response time of the dining and drinking philosophers problems. Chapter 10 describes our fast modular resource allocation algorithm (its informal description, the code, the correctness proof and performance analysis). Chapter 11 discusses our simulation result. Finally, Chapter 12 provides the conclusion and future direction of our work.

CHAPTER 7

Introduction

7.1 Problem

Distributed systems commonly include resources (CPUs, I/O channels, buffer space, data files, etc.) which must be managed so that no two processes access the resources at the same time, while avoiding starved or deadlocked processes. The scheduling of processes with various resource requirements in this type of system is generally known as *resource allocation.* We consider the problem in the message passing model. Typical efficiency metrics are the number of messages and the response time needed for a process to acquire all its required resources.

Examples of resource allocation can be found in many applications including distributed database systems and file systems. In distributed database systems, a transaction (i.e., process) accesses multiple data records during its operation and to maintain the consistency of the accessed data, a transaction usually locks the data records so that no other transactions can access them while it is accessing them. The locks on data records can be considered as resources to which a transaction needs to have exclusive access. In this system, efficient resource allocation schemes can be used to synchronize many concurrent transactions.

In many distributed file systems, file systems are replicated on many different sites to increase availability and fault-tolerance. A single update to a file entails the same update to every copy of the file in the different sites to make sure that all the sites contain the same copy of the file. Furthermore, a single update may contain updates of many different files (depending on the application that the file systems support). The updated files need to be locked while the updates are performed. The locks have to be managed efficiently without causing any deadlock or starvation. Efficient resource allocation schemes can be applied to synchronize many concurrent updates in this system.

Resource allocation in these kinds of systems has been formalized as three kinds: *the dining philosophers problem*, *the drinking philosophers problem* and *the dynamic resource allocation problem.* In all these problems, each process has a code segment singled out as *the critical region* where it uses the resources. A process enters its critical region after it acquires all its needed resources and then relinquishes them when it exits the region. If any two processes have overlapping resource requirements and they are either requesting the resources or using the resources, we say that the two processes are in *conflict.* No two conflicting processes can be in the critical region at the same time.

The dining philosophers problem [Di71] is a static resource allocation problem, where the process set is fixed and each process requests a fixed set of resources periodically. A more dynamic version of the dining philosophers, called drinking philosophers [CM84], allows a process to request a different subset of its maximum resource requirement. Any two processes whose current resource requirements do not overlap may execute at the same time even though their maximum resource requirements overlap. In the dining and drinking philosophers problems, each process knows its maximal set of conflicting processes. Using the information about the maximal sets of conflicting processes, we can formulate the dining and drinking philosophers problems in terms of a graph, called *the conflicting graph*, in which a node represents a process and an edge represents (possible) sharing of resource between two processes.

The dynamic resource allocation problem [AS90] is most general in that the process set may dynamically change over time and each process may need for its execution any set of resources and may need different sets of resources at different times (i.e., a process has no knowledge about its maximum resource requirement and its conflicting processes).

For convenience in measuring the response time of an algorithm, i.e., the time period between when a process requests resources and when it acquires them to enter the critical region, it is assumed that there are at most δ conflicting processes at any time during the execution of the algorithm [1] and c is the maximum time that a process is in the critical region. We also assume that d is the maximum message delay between any two processes. These parameters are used only for analyzing time complexity. They are not used in the code or to prove the correctness of algorithms; we only consider asynchronous systems.

The thesis of this part of the dissertation is that:

- $(\chi - 1)(c + d)$ is a tight bound for the response time of the dining philosophers

[1]In the dining and drinking philosophers problems, δ is the maximum degree of the conflict graph.

problem.

- Any dining philosophers algorithm with $(\chi - 1)(c + d))$ requires a large amount of local computation (unless P=NP).

- There exists a modular resource allocation algorithm that uses an arbitrary resource allocation algorithm as a subroutine and is also efficient, improving the response time of the subroutine.

7.2 Previous Results

We review the past relevant results on the resource allocation problems in chronological order in this section.

Dijkstra [Di71] first posed a resource allocation problem as a ring of 5 processes, called the dining philosophers, where each process shares a resource with each neighbor.

Later, Lynch [Ly81] generalized the problem to an conflict arbitrary graph. Lynch's solution uses an edge coloring algorithm to set a partial ordering on the shared resources, so that each process requests its needed resources in that order. The worst case response time is $O(c^{\delta}(c + d))$ where c is the number of colors used in the coloring. The number of colors represents the maximum length of the partial order among resources. The message complexity is $O(\delta)$.

Chandy and Misra [CM84] presented a dining philosophers algorithm using an acyclic directed version of the conflict graph. Processes (i.e., nodes) in the critical region have their edges always directed outward and processes which have all of their edges inward can enter the critical region. This ensures that the resulting conflicting graph is always acyclic assuming it is initially acyclic, and therefore, ensures no deadlock. They also first proposed the drinking philosophers problem and its solution which uses their dining philosophers solution as a subroutine. The worst case response time for both of the solutions is $O(n(c+d))$ and the message complexity is $O(\delta)$.

Both of the Chandy and Misra's algorithms are proven correct assuming a strong fairness property on the execution of actions. Murphy and Shankar [MS88] consider weaker fairness properties and show what modifications have to be made to preserve correctness.

Styer and Peterson [SP88] extended Lynch's algorithm to develop a dining philosophers algorithm with worst case response time $O(\delta^{\log\delta+1}(c+d))$ and message complexity $O(\delta^{\log\delta+1})$.

Ginat, Shankar and Agrawala [GSA89] proposed a drinking philosophers algorithm that solves the problem directly without using a dining philosophers subroutine, resulting in a more message efficient algorithm with the same time complexity as that of Chandy and Misra's algorithm.

Awerbuch and Saks [AS90] first defined and solved the dynamic resource allocation problem. In their model, they assume that processes have to know a priori the IDs of their conflicting processes.[2] They implemented a "distributed queue" where processes in the trying region select positions in such a way that no two conflicting processes select the same position. The processes at the head of the queue will enter the critical

[2]Note that our definition of the dynamic resource allocation problem is more general in that it doesn't assume a priori knowledge about the conflicting processes.

region. Process IDs are used to select such positions. The worst case response time of the algorithms is $O(\delta c + \delta^2(\log |\mathcal{U}|)d)$ where $\mathcal{U}$ is the the universal set from which process IDs are drawn, and the message complexity is $O(\delta^2 \log |\mathcal{U}|)$.

Weidman, Page and Pervin [WPP91] developed a dynamic resource allocation algorithm using Chandy and Misra's drinking philosophers algorithm as a subroutine without the assumption that [AS90] made – the assumption about a process having a priori knowledge about its conflicting processes. Their algorithm uses random numbers to break symmetry among conflicting processes, and allows a process to join any group of the conflicting processes by constructing a precedence graph which represents processes' priorities and the dynamic sharing patterns of processes. This graph is used as an input for the underlying Chandy and Misra drinking philosophers algorithm.

Choy and Singh [CS92] developed a dining philosophers algorithm that uses a node coloring scheme on the conflict graph to assign static priorities to processes to access the resources. Their algorithm used a concept of "double doors" where processes are blocked to enter each door if any conflicting processes are inside the door. Conflicts among processes inside the door are resolved by the static priorities among processes. Their algorithm has worst case response time $O(\delta^2(c + d))$ and message complexity $O(\delta)$. They also include a discussion of fault-tolerance.

Page, Jacob and Chern [PJC93] presented a dining philosophers algorithm using the conflict resolution technique of [CM84]. It has worst case response time $O(\delta c + nd)$ and message complexity $O(\delta^2)$. They have also conducted simulations of their algorithm and other algorithms including Chandy and Misra's algorithm [CM84] and Awerbuch and Saks' algorithm[AS90] Their simulation result showed that their algorithm performed better than Awerbuch and Saks' algorithm in terms of average response time, but worse than Chandy and Misra's.

Welch and Lynch [WL93] generalized the modular construction of Chandy and Misra's drinking philosophers algorithm to come up with a drinking philosophers algorithm which uses as a subroutine any dining philosophers algorithm. Its response time and message complexity are equal to those of the subroutine.

Bar-Ilan and Peleg [BP92] developed a *synchronous* algorithm that improves on Awerbuch and Saks' algorithm to have response time $O(\delta c + \delta(\log|\mathcal{U}|)d)$ in synchronous network systems.

Choy and Singh [CS93] developed a dynamic resource allocation algorithm with worst case response time $O(\delta^2 c + (\delta^3 + log^*|\mathcal{U}|)d)$ [3] with message complexity $O(\delta^2 + \delta\log^*|\mathcal{U}|)$ Their algorithm uses an idea of snapshots to make their dining philosophers algorithm [CS92] work in a dynamic system where processes are created or terminated dynamically. Snapshots of process states at an instance of time are used to find out the set of conflicting processes for each process, which changes over time.

Table 10 shows the worst case response time of the previous algorithms for dining/drinking philosopher problems and dynamic resource allocation problem. It does not contain the work by [WL93], [GSA89] and [MS88] because they are not concerned with the worst case response times of resource allocation algorithms. It also doesn't

[3] $\log^* n = \min\{i : \log^i n \leq 1\}$.

contain the work by [BP92] because their work is on the synchronous network.

Authors	Problem	Time	Message
Lynch [Ly81]	dining	$O(c^{\delta}(c+d))$	$O(\delta)$
Styer and Peterson [SP88]	dining	$O(\delta^{(\log\delta+1)}(c+d))$	$O(\delta^{(\log\delta+1)})$
Choy and Singh [CS92]	dining	$O(\delta^2(c+d))$	$O(\delta)$
Page et al. [PJC93]	dining	$O(\delta c+nd)$	$O(\delta^2)$
Chandy and Misra [CM84]	drinking	$O(n(c+d))$	$O(n)$
Awerbuch and Saks [AS90]	dynamic	$O(\delta c+\delta^2\log\lvert\mathcal{U}\rvert d)$	$O(\delta^2\log\lvert\mathcal{U}\rvert)$
Weidman et al. [WPP91]	dynamic	$O(nc+nd)$	$O(d)$
Choy and Singh [CS93]	dynamic	$O(\delta^2 c+(\delta^3+\delta\log^*\lvert\mathcal{U}\rvert)d)$	$O(\delta\log\lvert\mathcal{U}\rvert)$

Response time and message complexities of previously known algorithms (c is the time that a process uses a resource, d is the upper bound on the message delay, c is the number of colors used, δ is the number of conflicting processes, n is the total number of processes, $\mathcal{U}$ is the universal set from which process IDs are drawn.)

TABLE 10

7.3 Our Results

7.3.1 Lower Bound

Although there are many results about upper bounds on the response time for the resource allocation problems, there is little discussion about lower bounds on the response time. Only the obvious lower bound of $\Omega(\delta(c+d))$ is mentioned in the literature [AS90, BP92, CS92, PJC93].

The dining and drinking philosophers problems take as input the maximum resource requirements of all processes, from which we can construct the conflict graph (defined in Section 7.1). Note that δ is the maximum degree of the conflict graph. Based on characteristics of the conflict graph, we show a lower bound of $\Omega(\chi\cdot(c+d))$ on the response time for the dining and drinking philosophers problems, where χ is the chromatic number of the given conflict graph.

The previous lower bound $\Omega(\delta(c+d))$ for the dining (or drinking) philosophers problem states that for every algorithm, there exists an instance of the problem on which the algorithm has an execution with response time $\Omega(\delta(c+d))$, where δ is the maximum degree of the conflict graph for that instance. In contrast, we show that for every algorithm and *every* instance of the problem, the algorithm has an execution with response time $\Omega(\chi \cdot (c+d))$, where χ is the chromatic number of the conflict graph for that instance. The previous lower bound is still *correct*, since $\chi \leq \delta$ for every graph and there is a graph with $\chi = \delta$. However, our bound is more informative in that it has something to say about every instance.

We also give a dining philosophers algorithm with the matching response time and a drinking philosophers algorithm with almost matching response time (only differ by a constant factor). These algorithms use an optimal coloring algorithm as a subroutine, incurring a large amount of local computation. However, we show that "a large amount of local computation" is (probably) inevitable because any dining philosophers algorithm with the optimal response time is transformable in polynomial time into a sequential algorithm for an NP-complete problem.

A similar lower bound was presented by Barbosa and Gafni [BG87] on the maximum average number of times that a process runs in any given period, denoted by $\gamma^*(G)$, in a synchronous system where conflicting processes need to run in an alternating order. They proved that the inverse of $\gamma^*(G)$ is bounded in between the multi-chromatic number and the chromatic number of G. The NP-completeness of obtaining the optimal solution with respect to $\gamma^*(G)$ is also shown in [BG87]. Note that this result doesn't obviously imply our results. It might seem that obtaining an optimal response time would be easier than obtaining optimal $\gamma^*(G)$. Nonetheless, we prove that it is not the case.

7.3.2 A Fast Modular Resource Allocation Algorithm

Our results in Section 7.3.1 indicated it is hard, if not impossible, to have a dining philosophers algorithm with the optimal response time and with a polynomial amount of local computation. In this section, we consider distributed algorithms with non-optimal response time, but with a polynomial amount of local computation.

We present a modular algorithm that uses any arbitrary resource allocation algorithm as a subroutine. It can improve on the overall response time by letting each process wait, if it has to, only for its currently conflicting processes.

To be specific, consider an arbitrary resource allocation algorithm with response time $O(X \cdot (c + d))$ and message complexity $O(M)$ for some X and M. If we use the algorithm as a subroutine, the overall response time of the resulting algorithm is $O(\delta c + \max\{\delta^2, X\}d)$ and its message complexity is $\max\{M, O(r^2\delta)\}$, where r is the number of shared resources that a process requests. Since c is only applied to the δ term, our algorithm reduces the impact of c on the worst case response time when X is bigger than $O(\delta)$ (more precisely, when $X > 3$. See Section 10.3).

Reducing the impact of c on the response time can improve the performance of a system because c can often be large. For example, in such applications of resource allocation as database systems or distributed file systems, an access to each resource (i.e., records or files) in the critical region requires I/O operations because shared resources are typically in secondary storage. Especially it is more evident in a system such as a parallel processor system, where I/O latency is considerably larger than interprocessor communication latency.

The modularity of our algorithm provides an extra advantage that given any arbitrary dining philosophers algorithm as a subroutine, it behaves as a drinking philosophers algorithm, allowing more concurrency than the dining philosophers. Additionally, if the subroutine is a drinking philosophers algorithm or a dynamic resource allocation algorithm, so is the resulting algorithm. This is because our algorithm deals only with the current resource requirements of processes, but not with their maximums. So, when we use Choy and Singh's dining philosophers algorithm [CS92] as a subroutine, which has response time $O(\delta^2(c+d))$, we obtain a drinking philosophers algorithm with better response time $O(\delta c+\delta^2 d)$ and message complexity $O(r^2\delta)$. This is the fastest known drinking philosophers algorithm with the stated message complexity. When we use Choy and Singh's dynamic resource allocation algorithm [CS93] as a subroutine, which has response $O(\delta^2 c + (\delta^3 + \delta \log^* |\mathcal{U}|)d)$ [4] with message complexity $O(\delta^2 + \delta \log^* |\mathcal{U}|)$, we obtain a dynamic resource allocation algorithm with response time $O(\delta c + (\delta^3 + \delta \log^* |\mathcal{U}|)d)$ and message complexity $\max\{O(\delta^2 + \delta \log^* |\mathcal{U}|), O(r^2\delta)\}$. When $\log^* |\mathcal{U}|$ is less than δ, our dynamic algorithm is the fastest known dynamic resource allocation algorithm with the stated message

[4] $\log^* n = \min\{i : \log^i n \leq 1\}$.

complexity. A similar type of modularity was presented by Welch and Lynch [WL93], but one can get better performance by running the combined algorithm than by running the subroutine only unlike theirs.

The system structure of our solution is similar to those of Lynch [Ly81] and Weidman et al. [WPP91], in that there is one designated process for *each* resource, called a *resource manager*, that doles out resources to the requesting processes or maintains information about conflicting processes. In our solution, the resource managers also participate in scheduling accesses of processes to resources.

Our solution is distributed in that a (crash) failure of a single process (including a resource manager) doesn't necessarily affect the progress of the whole system, and resource managers do not act as a performance bottleneck. This is because each resource manager deals with resource requests only pertaining to its resource. Therefore, a crash of a resource manager doesn't necessarily affect the progress of the other processes that do not require the resource of the crashed resource manager. In fact, the effect of the failure of one resource manager can be less severe than that of the failure of one process (that is not a resource manager). To see this, imagine that a process crashes in the critical region while holding a set of resources, in which situation, all processes that require the resources cannot progress. This has the same effect as crashes of all the resource managers that manage the resources. Therefore, in terms of fault-tolerance, our solution is no less distributed than fully distributed solutions that do not use any centralized servers like resource managers.

We simulated our algorithm combined with Choy and Singh's algorithm (CSR) and with Chandy and Misra's algorithm (CMR), and compared the performance with Choy and Singh's algorithm, Chandy and Misra's algorithm and Awerbuch and Saks' algorithm. The simulation result indicated that the average performance of CMR and CSR is remarkably better than the other algorithms especially when resource contention among processes is high and the average time that a process remains in the critical region is bigger than the average message delay. We leave simulation of drinking philosophers and dynamic resource allocation algorithms for future work.

CHAPTER 8

Model and Problem Statements

8.1 System Model

There exists a (finite or infinite) set of processes $P = \{p_1, p_2, p_3, \ldots\}$. Processes communicate by passing messages. There are three types of process steps: *send*, *receive* and *local step*. Send and receive are communication primitives and a local step changes local variables of processes. Each process p_i is modeled by a finite state automaton with state set Q_i. The state set Q_i includes an initial state $q_{0,i}$.

The automaton for each process is specified by a single guarded command set $[B_1 \rightarrow A_1 \square B_2 \rightarrow A_2 \square \ldots \square B_m \rightarrow A_m]$. Each $B_i \rightarrow A_i$ is a guarded command, where a guard B_i is a boolean expression and/or a message reception (receive step), and a finite list of action statements A_i that consist of multiple local steps and/or one send step.

We model communication by a special process, called *network*, which schedules the delivery of messages sent among the other processes. Each process owns a buffer of infinite size. A buffer is an atomic read-modify variable, so that a process can add a message to the contents of the buffer atomically. A send step adds a set of messages to the buffer of the network. A receive step of a process removes some messages from the buffer of the process. The receive step of the network removes some messages from its buffer and the send step of the network adds the messages to the buffers of their destination processes. (In this case, we say that the message is *delivered* to the destination). A process can send the same message to a finite set of processes in one step, i.e., a send step can add multiple copies of the same message with different destinations to the buffer of the network in one step.

A *configuration* is a vector $C = \{q_1, q_2, \ldots\}$ where q_i is the local state of p_i for each $p_i \in P$ (including the network). A guarded command is *enabled* in a configuration if its associated boolean expression is true and associated receive, if any, can return nonempty messages, i.e., the messages specified in the receive are in the buffer of its process. An *execution of a guarded command* involves an atomic execution of all the steps in the action statements of the guarded command. It results in simultaneous changes to the state of the process of the guarded command based on the state of the process, and possibly the state of the network if the guarded command involves a send step. A guarded command enabled in a configuration C can be *applied* to C to yield a new configuration C' as a result of the execution of the guarded command.

A *system* is specified by describing P, an initial configuration $C_0 = (q_{0,1}, q_{0,2}, \ldots)$, and the automaton of all processes in P.

An *execution sequence* of a system is an infinite alternating sequence of configurations and enabled guarded commands $\alpha = C_0, \pi_1, C_1, \pi_2, \ldots C_i \pi_i \ldots$, where π_i is an enabled guarded command in C_{i-1}; and C_i is obtained by applying π_{i-1} to C_{i-1}.

We say that in an execution sequence, a guarded command is *continually enabled* from C_i to C_j, $i < j$ if the guarded command is enabled in every configuration from C_i to C_j and is not applied to any configuration in between C_i and C_{j-1}.

We also say that in an execution sequence σ, a guarded command g is enabled *before* a guarded command g' in C_k if there exists a sequence of configurations $C_i, \ldots, C_k$ in σ such that g is continually enabled from C_i to C_k, but g' is not continually enabled from C_i to C_j.

An *execution* is an execution sequence satisfying the following fairness conditions:

- If π_i is a guarded command of process p (that is not the network) and applied to C_i, then there is no guarded command of p that is enabled before π_i in C_i.
- All enabled guarded command will be executed eventually.
- Messages are received in the order that they are sent (i.e., FIFO).

A *timed execution* $(\alpha, T) = C_0, (\pi_1, t_1), \ldots, (\pi_j, t_j), \ldots$ satisfies the following:

- $\alpha = C_0, \pi_1, C_1, \pi_2, \ldots C_i \pi_i \ldots$ is an execution;
- T is a mapping from guarded commands to real numbers that associates a real time with each guarded command in the execution. The sequence $t_0, t_1, \ldots t_i \ldots$ is nondecreasing and unbounded.
- All messages sent are received in finite time.

If V is a state variable of a process and t is a real number, $V(t)$ denotes the value of V in the configuration C_j where $T(\pi_j) \leq t \leq T(\pi_{j+1})$, i.e., a configuration C_j represents the states of the system during time interval $[T(\pi_j), T(\pi_{j+1}))$.

Note that the system model here is completely asynchronous because there are no constraints on relative timing of process steps and message delays.

8.2 The Resource Allocation Problems

We now specialize the general system for resource allocation problem. Since we don't put any restriction on the behavior of resources and we treat them as passive entity[1], we don't model resources as physical objects in our system. We simply treat them as an additional system parameters. Let R be the set of all resource IDs in systems. There exists a set of processes U ($\subseteq P$) called *users* that need a subset of R for their execution at various times.

Each user's states are partitioned into four regions. In the *trying region*, the user requests its required resources. On acquiring the resources, the user enters the *critical region*. It remains in the region for a finite time using the resources. When the user is through with the resources, it enters the *exit region*, where it relinquishes the resources. Otherwise, the user is in the *remainder region*.

To specify this, we assume that each user has a local variable called *region* whose value is set to constant $Trying$, $Critical$, $Exit$, or $Remainder$ if and only if the user is in the trying region, the critical region, the exit region or the remainder region respectively. Initially, every user is in the remainder region.

Let $\mathcal{R}_i$ be a subset of R and denote *the maximum resource requirement* of a user i, and let R_i be a variable used to specify the current resource requirement of a user.

A user's code usually contains the statements that are not related to resource allocation such as the statements that are executed to use resources in the critical

[1]The exclusion property of a resource can also be modeled by restricting the behavior of processes.

region. A resource allocation algorithm is specified by only describing the statements that are related to resource allocation, so that it can be used as a subroutine for any users.

To formalize this concept, we define a mapping M_i for a user i that labels each guarded command in the code of user i to be either *resource allocation part* or *non-resource allocation part.* We call the set of guarded commands labeled *resource allocation the resource allocation part* of user i, and call the set of guarded commands labeled *non-resource allocation the non-resource allocation part* of user i.

A user code is *well-formed* if there exists M_i for any i in U such that:

1. All the guarded commands in the non-resource allocation part are executed only when user i is in the critical or remainder region.

2. *region* and R_i are the only variables that are used (i.e., read or written) both in the resource allocation part and in the non-resource allocation part.

3. In the resource allocation part, *region* is updated only either from $Trying$ to $Critical$ or from $Exit$ to $Remainder$.

4. In the non-resource allocation part, *region* is updated only either from $Critical$ to $Exit$ or from $Remainder$ to $Trying$.

5. R_i is updated only in the non-resource allocation part.

At some time t, if users i and j are in the trying or critical region, and $R_i(t) \cap R_j(t) \neq \emptyset$, then we say that user i *conflicts* with user j at time t.

The *dining philosophers* problem is a system with a finite and fixed set U. For each user i, there exists a fixed $\mathcal{R}_i$ and for all times t, $R_i(t) = \mathcal{R}_i$. Each user's code is well-formed and the algorithm satisfies the following two conditions even when the non-resource allocation part of user i (labeled by M_i) is replaced by any set of guarded commands:

- (*exclusion*) In any timed execution of the algorithm, if users i and j are both in their critical regions at time t, then i and j do not conflict with each other; this is the property that guarantees exclusive access to the resources.

- (*no-lockout*) In any timed execution of the algorithm, if a user is in the trying region or in the exit region, then it leaves its current region in finite time assuming no user remains in the critical region forever.

In a drinking philosophers algorithm and a dynamic resource allocation algorithm, if two users i and j are not conflicting currently even though they can potentially be conflicting (i.e., their maximum resource requirement overlaps) it should be possible for both to be in the critical region at the same time, allowing more concurrency. To formalize this concept, we define the following condition:

- (*concurrency)* The set of every timed execution of the algorithm does not include an execution where a user i enters the trying region at some time t; no conflicting user of i is in the trying region or the critical region at time t; after time t, no conflicting user of i enters the trying region; and user i remains in the trying region forever.

This condition ensures that as long as there are no conflicting users while user i is in the trying region, user i is never stuck in the trying region *even if other users are using other resources forever.* The same concurrency condition is given in [WL93].

The *drinking philosophers* problem is a system with a finite and fixed user set U. For each user i, there exists a fixed $\mathcal{R}_i$ and $R_i(t)$ is a priori unknown to all processes. Each user's code is well-formed and the algorithm satisfies the exclusion, no-lockout and concurrency conditions even when the non-resource allocation part of user i (labeled by M_i) is replaced by any set of guarded commands.

Based on the maximum resource requirements of users in the dining and drinking philosophers problem, we can construct a conflict graph where a node represents a user and an edge represents a sharing of resources between two users. Given $\mathcal{R}_i$ for all users $i \in U$ where $|U| = n$, let $G = (V, E)$, *the conflict graph* for U, be an undirected graph such that $V = \{v_1, v_2, \ldots v_n\}$ and $(v_i, v_j) \in E$ iff R_i intersects R_j. We use the conflict graph to prove a lower bound on the response time of the dining and drinking philosophers problems.

The *dynamic resource allocation* problem is a system with an infinite set U where for all users i, user i is well-formed, $\mathcal{R}_i$ and $R_i(t)$ are a priori unknown to all processes. Each user's code is well-formed and the algorithm satisfies the exclusion, no-lockout and concurrency conditions even when the non-resource allocation part of user i (labeled by M_i) is replaced by any set of guarded commands.

Note that any dynamic resource allocation algorithm is also a drinking philosophers algorithm, but not necessarily vice versa, and any drinking philosophers algorithm is also a dining philosophers algorithm, but not necessarily vice versa.

The *response time* of an algorithm is defined to be the maximum time period between when a user enters the trying region and when the user next enters the critical region for all timed executions of the algorithm. However, because there are no upper bounds on step time and message delay in our system model, it is not possible to prove an upper bound for the response time in real time in the model.

Therefore, when we measure the response time, we only consider the timed executions in which there exists an upper bound of one time unit on the period between when a process takes a step and when the process takes the next step; and there exists an upper bound of d on the *message delay* between any two processes. Message delay is the time period between when a process executed a send step to send a message and when the message is delivered to the destination process's buffer. Note that this approach of measuring time complexity in an asynchronous system was first proposed in [PF77].

In addition, the response time is dependent on the degree of resource contention (i.e., the number of conflicting users) and possibly the number of resources that a user request at a time. Therefore, we assume that there are at most δ conflicting users for any user at any time and r is the maximum number of shared resources that a user requests. We also assume that c is the maximum time that a user is in the critical region.

The existence of these parameters may seem contradictory to the description of the system model. However, these parameters are not used by the algorithms and are defined only for the *convenience* of the time complexity measure. So the algorithms should not depend on these parameters and should work in any environment where the description of the model fits.

We would like to measure the number of messages that are required for a user to enter the critical region. However, in any algorithm, it is possible that some messages are generated by a process to help another process enter the critical region or to respond to the request of processes that are trying to enter the critical region. These messages should be charged to the process entering the critical region. In order to count messages fairly, we need some way to map the generated messages to each entrance to the critical region.

Let f be a mapping that maps a message in an execution α of the algorithm to a (two dimensional) vector $(i, j)_\alpha$ which represents the jth entrance to the critical region by user i in α. Let M_f be the maximum number of messages that f maps to $(i, j)_\alpha$ for any i and j in all executions α of the algorithm. The message complexity is $\min\{M_f\}$ for all such f's.

CHAPTER 9

Lower Bound and NP-Completeness

In this chapter, we study the lower bound on the response time of the dining philosophers problem and the drinking philosophers problem. We also give a dining philosophers algorithm with the response time matching the lower bound and a drinking philosophers algorithm with the response time close to the lower bound (*only differ by a constant factor*). These algorithms use an optimal coloring algorithm as a subroutine, incurring a large amount of local computation. However, we show that "a large amount of local computation" is inevitable (unless P=NP) because any dining philosophers algorithm with the optimal response time is transformable in polynomial time into a sequential algorithm for an NP-complete problem.

9.1 Lower Bound on Response Time

Before we prove the lower bound, we first define a few terms. We define *a coloring assignment* C of an arbitrary graph $G = (V, E)$ to be a function $C : V \rightarrow \mathcal{N}$ [1] such that if $(v_i, v_j) \in E$, $C(v_i) \neq C(v_j)$. The *chromatic* number of a graph G is the minimum number of colors used for any coloring assignments of G. Finding a coloring assignment with the minimum number of colors for a given graph G is called *the coloring problem.*

[1] $\mathcal{N}$ is the set of natural numbers.

The lower bound studied here is based on the conflict graph G, which is given as input to the dining and drinking philosophers problems. Based on all resource requirements of users in dining (or drinking) philosophers problem, we construct the conflict graph G. Let χ be the chromatic number of G. In this section, we present the lower bound $\Omega(\chi \cdot (c+d))$ on the response time.

The structure of our lower bound proof is as follows. Given a dining or drinking philosophers algorithm, we define a specific execution of the algorithm in which all users enter the trying region at the start of the algorithm; every user stays in the critical region exactly c time; and once a user return to its remainder region, it stays there forever.

In this execution, we first prove that there exists a sequence of users $u_1, u_2, \ldots, u_\chi$ where for any i, u_i conflicts with u_{i+1} and u_i enters the critical region before u_{i+1} (see Lemma 9.1.1). Then, we prove that at least one communication delay (which is at most d time) elapses between when one user u_i leaves the critical region and when the next conflicting user u_{i+1} enters it (see Lemma 9.1.2). Thus, the worst case time period that u_χ is in the trying region is $(\chi - 1)(c + d)$. This proves the lower bound.

Let A be a dining (or drinking) philosophers algorithm for the given conflict graph G, and let (β, T), called *the synchronized execution*, be the timed execution of A where all users enter the trying region at time 0; after a user enters the remainder region, it remains in the region (i.e., no more requests for resources); all message delays are exactly d and all critical region times are exactly c where d and c are positive integers bigger than 1 and for every guarded command g in β, $T(g)$ is an integer.

Lemma 9.1.1 *In (β, T), there is a sequence $u_1, u_2, \ldots, u_k$ where $k \geq \chi$, such that for all j, $1 \leq j \leq k$, user u_j leaves the critical region before user u_{j+1} enters it and user u_j conflicts with user u_{j+1}.*

Proof: Assume by way of contradiction that there is no such sequence $u_1, \ldots, u_k$ with $k \geq \chi$ in β. We contradict the assumption by providing a coloring assignment of G with less than χ colors.

Let $t_c(j)$ and $t_e(j)$ be the times that user j enters the critical region and the exit region respectively in (β, T). Let DG be a directed graph (V, E') where $(v_i, v_j) \in E'$ if and only if $t_e(i) < t_c(j)$ and user i is in conflict with user j. Note that if any users i and j are in conflict, there is always an edge either from v_i to v_j or from v_i to v_j in DG because time periods $(t_c(i), t_e(i))$ and $(t_c(j), t_e(j))$ do not overlap by Lemma 9.1.2. Note also that DG is acyclic. Therefore, there must be at least one node such that all its edges are outgoing. Let V_1 be the set of all such nodes and define the *maximum distance* between V_1 and a node v_j to be the number of edges in the longest path between any node in V_1 and v_j. There must be at least one path from a node in V_1 to v_j because V_1 contains all the nodes with only outgoing edges. Let V_i be the set of all the nodes whose maximum distance from V_1 is $i-1$. It is true that $V_i \cap V_j = \emptyset$ for any i and j; $\cup_{\forall i} V_i = V$ and $\max\{i | V_i \neq \emptyset\} \leq k$ by the definition of the maximum distance.

Define a coloring assignment c of G as follows: for all $v_i \in V$, if $v_i \in V_j$, we color v_i with color j. No two adjacent nodes i and j have the same color because there must be an edge between them in DG, and thus, they cannot be in the same set V_l

for some l. Since $\max\{i|V_i \neq \emptyset\} \leq k < \chi(G)$, c is a coloring assignment with less than $\chi(G)$ colors. This is a contradiction. ■

Lemma 9.1.2 *In (β, T), if user i enters the exit region at time t and conflicting user j hasn't entered the critical region before time t, then user j enters the critical region only after $t + d$.*

Proof: By way of contradiction, we assume that j enters the critical region at time $t + x$ where $x < d$. Then, we prove that there exists an execution (β', T') where users i and j are in the critical region together. This contradicts the exclusion condition of A. We obtain execution (β', T') by retiming and reordering (β, T) in such a way that all the guarded commands of j executed during time interval $(t-1, t+x]$ in (β, T) happen during time interval $(t-1, t)$ in (β', T'). We also retime all the guarded commands of the network that deliver messages to j during time interval $(t-1, t+x]$ of (β, T) in the same way. The details follow.

Fix $\epsilon > 0$. For all guarded commands g in β,

$T'(g) = t - 1 + \frac{1}{x+1+\epsilon}(T(g) - (t-1))$	if $T(g)$ is in $(t-1, t+x]$ and g is either a guarded command of j or a guarded command of the network that adds messages to j's buffer.
$T'(g) = T(g)$	otherwise.

We reorder the guarded commands in β according to the new time assignment T'. Based on C_0 and the reordered sequence of guarded commands, it is easy to construct a new execution sequence β'. We prove that β' is an execution. Note that this retiming moves the guarded command g of j that enters the critical region to occur at some time in time interval $(t-1,t)$ because $T(g)=t+x$ and $T'(g)=t-1+\frac{(x+1)}{x+1+\epsilon}$. This means that user j enters the critical region in the interval $(t-1,t)$. Therefore, since user i is in the critical region at time between $(t-1,t)$, users i and j are in the critical region at the same time. This is a contradiction because i and j are conflicting.

Now we provide details. First, we show that β' is an execution.

Claim 9.1.3 *All messages received in β' are sent before they are received.*

Proof: Any messages sent to j at or after time t are received after time $t+x$ in (β,T) because $x<d$. Since all the guarded commands happening after time $t+x$ are not retimed, the messages sent to j at or after t are still received after time $t+x$ in (β',T'). Any messages sent to j at or before time $t-1$ in (β,T) are received after time $t-1$ (excluding time $t-1$) because $d\geq 1$. Since all the guarded commands executed after time $t-1$ are still retimed after time $t-1$ in (β',T'), those messages sent to j before $t-1$ are still received after time $t-1$. All the messages sent by j are received after they are sent in (β',T') because all the retimed guarded commands are moved only backward in time (since $\epsilon>0$). Therefore, all the messages received are sent before they are received in (β',T'). ∎

In β', the relative order between any two guarded commands of the same process (except the network) does not change because in the retiming, we only shrink or extend the time period between the execution of two consecutive guarded commands of the same process (although the relative order among guarded commands of different processes may change).

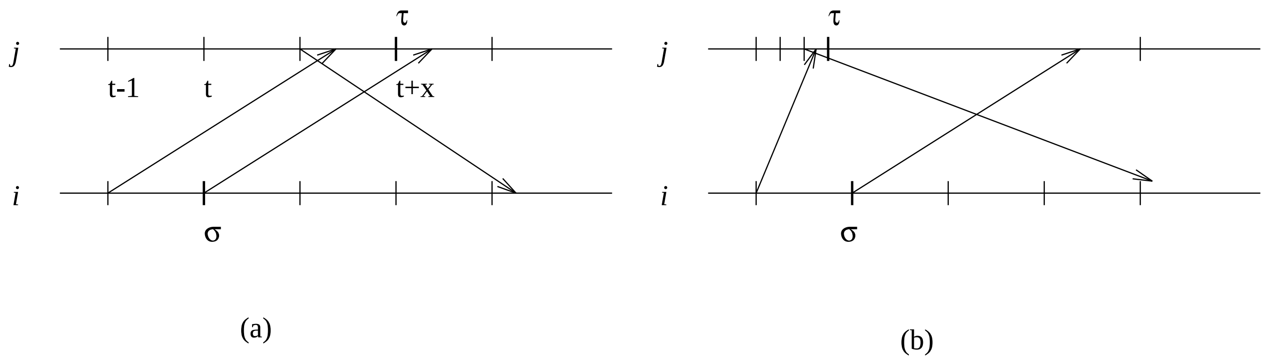

Fig. 11. An example of the retiming. (a) (β, T), (b) (β', T'). An arrow represents message delivery and delay; a tick mark represents an execution of a guarded command and time increases from left to right). In (β', T'), τ occurs before σ which means users i and j are in the critical region at the same time.

Therefore, since all the guarded commands of the same process are executed in the same order in β' as in β and all the messages are received after they are sent in β by Claim 9.1.3, all the processes go through the same order of state transitions (of course, except the network) as in β where all the guarded commands are enabled before they are executed. Therefore, in β' all the guarded commands are enabled before they are executed as in β. This is sufficient to prove β' is an execution sequence. Also the fact that all the processes goes through the same order of state transitions as in β implies that the fairness conditions are also satisfied in β' because the fairness conditions are satisfied in β. Therefore, β' is also an execution.

Let σ be the guarded command of user i at time t that leaves the critical region and τ be the guarded command of user j at time $t + x$ that enters the critical region in (β, T). By the retiming, $T'(\sigma) > T'(\tau)$ for any $\epsilon > 0$. Therefore, users i and j are in the critical region at the same time in (β', T'). This is a contradiction (see Figure 11). ■

Corollary 9.1.4 *The worst case response time of any algorithm for the dining philosophers problem and the drinking philosophers problem is* $(\chi - 1) \cdot (c + d)$.

Proof: Lemma 9.1.1 states that in β there is a sequence of χ users that enter the critical region consecutively. Recall that all users enter the trying region at time 0 by definition of β. Since each user spends c time in the critical region (definition of β) and at least d time elapses between when one user leaves the critical region and the next enters it (Claim 9.1.3), the result follows. ■

9.2 An Optimal Algorithm and NP-Completeness

The following theorem shows that $(\chi - 1) \cdot (c + d)$ is the optimal response time for the dining philosophers problem.

Theorem 9.2.1 *There exists a dining philosophers algorithm whose response time is* $(\chi - 1) \cdot (c + d)$.

We can prove theorem 9.2.1 by providing a simple algorithm. We only give an informal description of the algorithm here.

Algorithm: Given a conflict graph of a dining philosophers problem, we can color the conflict graph G with χ colors by using any known coloring algorithm. Therefore, each user knows the colors of all users. The basic idea of the algorithm is to use token-passing. Tokens are passed among users with different colors in such a way that if a user with color i holds a token, to pass the token to other users, it sends a message to all users with color $i + 1 \bmod \chi$. If a user with color $i + 1 \bmod \chi$ receives a message from each user with color $i \bmod \chi$, it now has a token. Initially, all users with color 1 have a token. If a user i is in the trying region while holding a token, it enters the critical region and when it enters the exit region, it passes the token. If not, it simply passes the token.

It is easy to see that the response time of the described algorithm is $(\chi-1)\cdot(c+d)$ because the users with color χ may have to wait for $\chi - 1$ users to enter the critical region and pass the token in series.

This algorithm can be easily converted to a drinking philosophers algorithm using Welch and Lynch's drinking philosophers algorithm [WL93]. Their algorithm has the property that, given any arbitrary dining philosophers algorithm as a subroutine, it behaves as a drinking philosophers algorithm with only a constant factor increase to the response time of the subroutine.

Obviously, this algorithm is very expensive because it uses an optimal coloring algorithm and all the known optimal coloring algorithms require exponential amount of (local) computation. However, we prove that this cost is inevitable (unless P=NP) by showing that any dining philosophers algorithm is transformable in polynomial time into a sequential algorithm for an NP-complete problem.

Theorem 9.2.2 *Any algorithm for the dining philosophers problem with response time less than or equal to $(\chi - 1)(c + d)$ is transformable in polynomial time into a sequential algorithm for an NP-complete problem.*

Proof: To prove the theorem, we first define the relevant NP-complete problem. Then we show how to transform a distributed dining philosophers algorithm with response time $(\chi-1)(c+d)$ into a sequential algorithm for the NP-complete problem.

Definition 9.2.1 *The optimal resource scheduling problem is, given a process set* $P = \{p_1, p_2, \ldots p_n\}$*, a resource set* $R = \{r_1, r_2, \ldots\}$ *and a resource requirement set* $Req = \{R'_1, R'_2, \ldots R'_n\}$*,* $R'_i \subset R$*, find a function* $T : P \rightarrow \mathcal{N}$ *such that if* R'_i *intersects* R'_j*,* $T(p_i) \neq T(p_j)$ *and* $\max_{\forall i}\{T(p_i)\}$ *is as small as possible.*

We call time mapping T a *schedule* and $\max_{\forall i}\{T(p_i)\}$ *the response time* of schedule T.

Lemma 9.2.3 *The optimal resource scheduling problem is NP-complete.*

Proof: We can easily show that the optimal resource scheduling problem is in NP. Since we know that finding a coloring assignment with the minimum number of colors is NP-complete, showing that the coloring problem is transformable to the optimal resource scheduling problem in polynomial time will be sufficient to prove the theorem.

Given a graph $G = (V, E)$, $V = \{v_1, v_2, \ldots, v_n\}$, we first construct P and R. Let $R'_i = \{r_{i,j} | (v_i, v_j) \in E\}$ and $P = \{p_1, p_2, \ldots, p_n\}$. It is easy to see this construction takes a polynomial time. Now, we run an optimal resource scheduling algorithm to obtain a schedule T with response time k for some k. Given this schedule, we can easily come up with a coloring assignment C with k colors: for all i, let $C(v_i) = T(p_i)$. Because no two conflicting processes p_i and p_j are assigned the same time by T and $(v_i, v_j) \in E$ by the construction, v_i and v_j must be assigned different colors. Therefore, C is a coloring assignment. We only need to prove that k is the minimum number of colors used by any coloring assignment of G. By way of contradiction, assume that there is another coloring assignment C' of G that uses less than k colors. Then we can come up with a schedule T' with response time less than k: for all i, let $T'(p_i) = C'(v_i)$. T' is a schedule because any two nodes v_i and v_j, $(v_i, v_j) \in E$, must have different colors so that any two conflicting processes are assigned different times. Thus, T' is a schedule with response time less than k. A contradiction. Therefore, k is the minimum number of colors. Since the coloring problem is transformable to the resource allocation problem in polynomial time, the optimal resource scheduling problem is NP-complete. ■

We now show how to transform any (distributed) algorithm with response time $(\chi - 1)(c + d)$ into a (sequential) optimal resource scheduling algorithm.

Transformation: First, since A is a dining philosophers algorithm, the user code of A must be well-formed and there exists a mapping M_i for each i that labels each guarded command either with resource allocation or with non-resource allocation satisfying the conditions specified in Section 8.2.

We replace the non-resource allocation part of the user code (labeled by M_i) by the following code:

<u>Remainder region:</u>

$\Box region = Remainder, \neg done \rightarrow$
$\quad region = Trying;$
$\Box region = Remainder, done \rightarrow$
$\quad done = true;$

<u>Critical region:</u>

$\Box region = Critical \rightarrow$
$\quad$ print $user\_id$;
$\quad region = Exit;$

done is a boolean variable and initially set to false, and *user_id* is ID of the user. The new algorithm is still a dining philosophers algorithm with the optimal response time because any dining philosophers algorithm should work correctly for any given remainder and critical region as long as $R_i = R_i(t)$ for a user i and any time t, and the resulting code is still well-formed.

The code states that all users enter the trying region as soon as they are in the remainder region (at time 0 because users is in the remainder region initially); in the critical region, they print out their ID and then they enter the remainder region (i.e., $c = 1$); and once entering the remainder region, users remain there forever.

Then, we simulate the new dining philosophers algorithm in a sequential machine as follows. Each process's local variables, buffer and state are maintained in a separate memory space. Each process in P is executed in round robin order, i.e., at each round, one enabled guarded command of each process, if there is any, is executed based on the local state of the process which is maintained in its memory space. At the end of each round, the network checks its buffer and delivers all the messages in the buffer to the destination process's buffer. This ensures that the message delay in this sequential execution is 0 (i.e., $d = 0$) because messages are delivered in the same round they are sent. It is easy to see that this transformation adds only polynomial time overhead (in fact, a factor of n if n is the total number of processes in P).

Let A_{seq} be this sequential version of the new algorithm. We can now use A_{seq} as a subroutine to solve the optimal resource scheduling problem. We feed Req, which is given for the optimal resource scheduling problem, as input to A_{seq} (i.e., let $R'_i = \mathcal{R}_i$) and run A_{seq}. A prints user ids that enters the critical region at each round (see the code of the critical region). All users must enter the critical region once by round χ

because $c = 1$ (because users takes one step in the critical region according to the code) and $d = 0$ in the sequential execution and the response time of the original dining philosophers algorithm is $(\chi - 1)(c + d)$. To come up with an optimal schedule T, for each user i, let $T(i)$ be the round that the user enters the critical region. It is easy to see that $\max_{\forall i}\{T(p_i)\} = \chi$ and it is the minimum because the dining philosophers algorithm has worst case response time $(\chi - 1)(c + d)$. This proves the theorem. ∎

Theorem 9.2.2 implies that any dining philosophers algorithm with the optimal response time must have a large (possibly exponential) amount of local steps. This justifies our use of an optimal coloring algorithm in our dining philosophers algorithm with the optimal response time. However, there are other disadvantages of the algorithm. The algorithm is message inefficient, i.e., its message complexity is $O(n^2)$ and is contention insensitive, i.e., the response time of the algorithm is always at most $(\chi - 1)d$, even if only one process is in the trying region. Finding a message efficient, and contention-sensitive distributed dining philosophers algorithm with the optimal response time is an open problem.

CHAPTER 10

A Modular Resource Allocation Algorithm

In this chapter, we describe our modular resource allocation algorithm. We first provide an informal description, present the code of the algorithm and then give a correctness proof and worst case performance analysis.

10.1 Informal Description

The basic idea of our algorithm is an implementation of a simple distributed queue, where each user has a position in the queue and enters the critical region in the order of position. The distributed queue runs under the following operational rules: (1) each user in the trying region occupies a position in the queue in such a way that no two conflicting users occupy the same position (note that this doesn't exclude the possibility that two non-conflicting users are at the same position); (2) users at the front of the queue enter the critical region and leave the queue when they enter the exit region; (3) when all users at the front of the queue leave the queue, all other users in the queue advance one position, preserving their relative order; (4) a new user joining the queue does not obstruct any advancing users in the queue.

It is trivial to see that the distributed queue with these operational rules guarantees no-lockout and exclusion (rules 1 and 2 ensure the exclusion condition and rules 3 and 4 ensure the no-lockout condition).

In our implementation, there is a group of special processes called *resource managers*, each assigned to one resource. Each resource manager rm_k maintains a queue q_k. The goal is to make these queues as a whole behave as the distributed queue by implementing the four operation rules. The following discusses our implementation.

We apply a modular approach to implement the first rule: we use another resource allocation algorithm (of any kind) as a subroutine. Each user i in the trying region at time t first runs the subroutine. When it enters the critical region of the subroutine, it selects a position to occupy in each q_k, $k \in R_i(t)$. The idea is that while a user i is in the critical region of the subroutine where user i is selecting a position in the queue, no other conflicting users are in the region. Therefore, user i can select a unique position in each q_k. In order to select a position in a queue, user i sends a *report* message to the resource manager of the queue and the resource managers acknowledge with a *marked* message that contains information about positions occupied by other users in the queue. After finishing the selection, the user informs all relevant resource managers of the selected positions with a *select* message. It is easy to see that it takes $O(d)$ time for a user to select the positions after entering the critical region of the subroutine. If the response time of the subroutine is $X(c+d)$ for any X, a user can select the positions in time $O(Xd)$ after it enters its trying region.

The position that a user selects affects the overall response time of the user. For example, a user may select the end position of each queue. But this can increase the response time. To see this, suppose that a user i needs resources a and b, q_a is occupied by 4 users and q_b is empty. If user i selects the end position of each queue, it will occupy position 5 in q_a and 1 in q_b. Suppose also that a user k requiring resource b subsequently selects position 2 in q_b. Now user k has to wait for user i to leave the queue which is in turn waiting for 4 other users to finish (see Figure 12). This waiting chain can grow up to $\Omega(n)$ users in length which will result in $\Omega(nc)$ worst case response time.

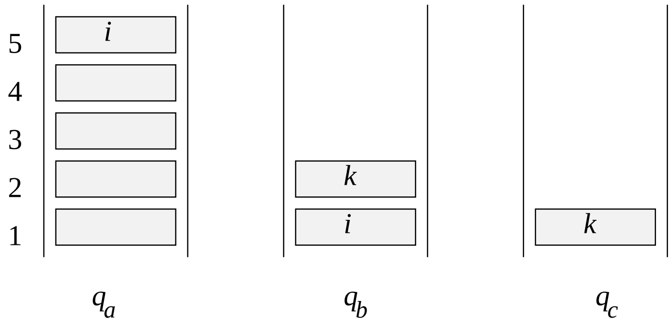

Fig. 12. An illustration of queues when a user selects the end position of queues: user k has to wait for the users that i is waiting for (occupied positions are shaded).

We propose that each user i selects the smallest position unoccupied in all q_k, $k \in R_i(t)$. That is, a user looks for a "hole" in all queues. In the above example, user i will select position 5 because it is the smallest position unoccupied in both q_a and q_b, and user k will select 1 because position 1 is unoccupied in q_b. Therefore, user k does not have to wait for user i to finish (see Figure 13). This way, each user, if it has to, waits only for its conflicting users.

To implement the second rule, each resource manager sends a *grant* message to the user at the front of its queue. A user i, when receiving the messages from all rm_k, $k \in R_i(t)$, enters its critical region. After it is through with its critical region, it sends a *release* message to all rm_k. This is the exit region of user i. Upon receiving a *release* message, a resource manager marks the front of the queue unoccupied.

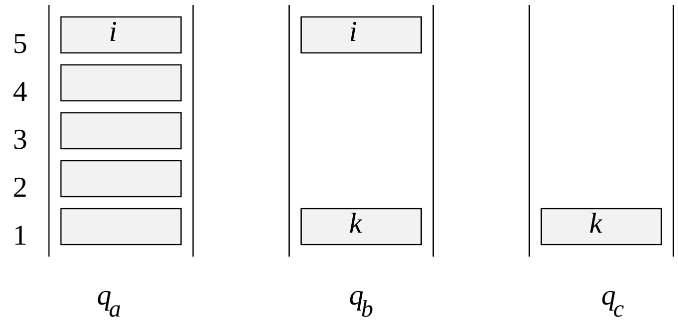

Fig. 13. An illustration of queues when a user selects a "hole": user k does not wait for the users that user i is waiting for.

If a position in a queue becomes unoccupied, any user occupying the immediately succeeding position advances to the position.

However, because of the asynchrony in the system, careless advancement of users in a queue may break the relative order among conflicting users. To illustrate this, suppose that user i selects a position p in resources a and b. Assume that all the users at the positions below position p in queue q_a finish executing the critical region and leave the queue before those in q_b. Suppose that after some time, the positions of user i becomes equal to, say, $p-4$ in q_a and p in q_b. It is possible that not all positions below position p in q_b are occupied. So, in the meanwhile, it may happen that some user j which also requires resources a and b selects, say, position $p-2$ in q_a and q_b. Now, because user j waits for i in q_a while i waits for j in q_b, there is no order between user i and j and this may cause deadlock (see Figure 14).

To implement the third rule without breaking the relative order among conflicting users, we let each resource manager maintain an integer counter for each occupied position in its queue. We denote by $C_k(p)$ the counter variable of position p in q_k.

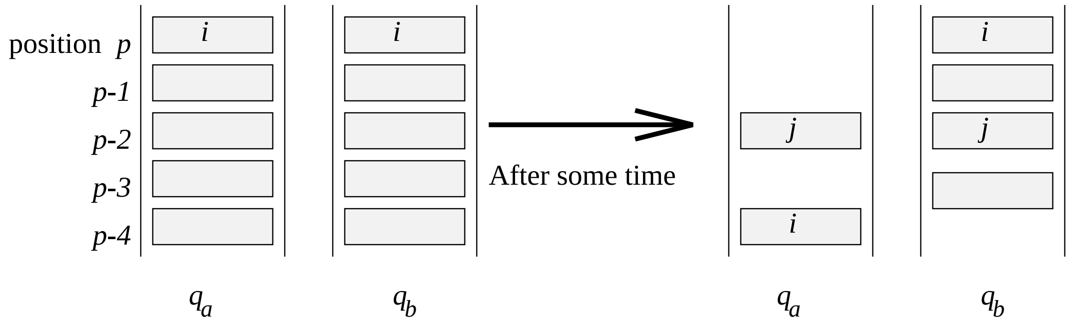

Fig. 14. An illustration of queues when a user i selects position p initially and then after some time user j selects $p-2$. This shows that a careless advancement of users (here user i in q_a) causes a deadlock.

When a user i occupies a new position p at time t, it initializes $C_k(p)$ to be $|R_i(t)|$. When position $p-1$ in q_k becomes unoccupied, rm_k decrements $C_k(p)$ by one and sends $dec(p)$ messages to all $rm_{k'}$, $k \neq k', k' \in R_i(t)$. Upon receiving a $dec(p)$, a $rm_{k'}$ decrements $C_{k'}(p)$ by one. When $C_k(p)$ becomes 0, user i advances to position $p-1$ in q_k.

It can be proved that when the above scheme is used, the difference between any two positions of user i is less than two, that is all the positions of user i in all q_k, $k \in R_i(t)$, advance almost in synchrony. This preserves the relative order among the conflicting users. For example, in the above example, user i will remain in position p or $p-1$ in both q_a and q_b. Therefore, user j cannot occupy a position in between the two positions of user i.

The fourth rule can be easily implemented by letting new users not consider the positions immediately preceding any occupied ones in the queues of their required resources.

We now discuss time performance. Because each user selects the minimum position that is not occupied by all its conflicting users and there are at most δ conflicting users, the initial position of a user is always $O(\delta)$. By induction on the position that a user occupies, it can be shown that a user advances one position in time $O(\delta d)$ whenever all position 1's in all queues are unoccupied (which will take at most time c). Therefore, since there are at most $O(\delta)$ positions in front of any user, a user will reach the front of the queue in time $O(\delta c + \delta^2 d)$. Since it takes time $O(Xd)$ to select an initial position, the overall response time of the algorithm is $O(\delta c + \delta^2 d + Xd)$.

We now discuss the message complexity. Before a user i advances to a new position in a q_k, rm_k sends only one dec message to all $rm_j, j \in R_i(t)$. Therefore, since $p \leq 2\delta$ and $|R_i(t)| \leq r$, the message complexity is $O(M + r^2\delta)$ if M is the message complexity of the subroutine.

10.2 The Code

As described in Chapter 8, in each user's code, *region* is set to *Trying*, *Critical*, *Exit* or *Remainder* if and only if a user is the trying region, the critical region, the exit region or the remainder region. The subroutine also has its own variable "*region*" which is also set to *Trying*, *Critical*, *Exit* or *Reminder*. To distinguish these two variables, we call *region* in the subroutine *sub-region*.

Given a subroutine, we take only the resource allocation part of the code and concatenate it to the user code described in Figure 15. All the occurrences of *region* in the resource allocation part of the subroutine are changed to *sub-region* and the variable R_i in the subroutine is also changed to $subR_i$. The resource allocation part of the subroutine should set *sub-region* to *Critical* and *Remainder* when it enters its critical region and its remainder region of the subroutine respectively and it should also enter the trying region and the exit region of the subroutine when *sub-region* is *Trying* and *Exit* respectively.

Figures 15 and 17 show the code of user and resource manager.

The following messages are used in the algorithm.

- $select(p, R_i)_{i,k}$: from user i to rm_k; indicates that i selected position p in q_k, and contains information about R_i.

- $report_{i,k}$: from user i to rm_k; requests information about the occupied positions in q_k.

- $release_{i,k}$: from user i to rm_k ; indicates that it has finished using the resource.

- $marked_{k,i}$: from rm_k to user i; contains information about all the occupied positions and their preceding positions of q_k.

- $grant_{k,i}$: from rm_k to user i; indicates that resource k is granted to the user.

- $dec(p)_{k,k'}$: from rm_k to $rm_{k'}$; indicates that position $p-1$ of q_k is unoccupied.

The following state variables are used in the user code (see Figure 15).

sub-region

- $\mathcal{R}_i$: the maximum resource requirement of user i.

- R_i: the current resource requirement of user i. Initially empty.

Protocol for user i :

$\Box$ $region = Trying$ and $sub\text{-}region = Remainder \rightarrow$
    $subR_i := R_i$;
    $sub\text{-}region := Trying$;

$\Box$ $sub\text{-}region = Critical$ and $req\text{-}report = false \rightarrow$
    $req\text{-}report := true$;
    for all $j \in R_i$: send $report_{i,j}$;

$\Box$ Receive $marked_{j,i}$, $\forall j \in R_i \rightarrow$
    $p := \min\{\mathcal{N} - \bigcup_{j \in R_i} marked_{j,i}\}$;
    for all $j \in R_i$: send $select(p, R_i)_{i,j}$;
    $req\text{-}report := false$;
    $sub\text{-}region := Exit$;

$\Box$ Receive $grant_{j,i}$, $\forall j \in R_i \rightarrow$
    $region := Critical$;

$\Box$ $region = Exit \rightarrow$
    $region := Remainder$;
    for all $j \in R_i$: send $release_{i,j}$;

Fig. 15. The Code of User i

- $subR_i$: the current resource requirement of the subroutine for user i. Initially empty.
- *req-report*: a boolean; true iff a user sent *report* messages, but hasn't received *marked* messages. Initially false.
- *region*: *Trying*, *Critical*, *Exit* and *Remainder* iff the user is in the trying region, the critical region, the exit region and the remainder region respectively. Initially *Remainder*.
- *sub-region*: *Trying*, *Critical*, *Exit* and *Remainder* iff the subroutine is in the trying region, the critical region, the exit region and the remainder region respectively. Initially *Remainder*.

Figure 16 shows the relationship among subroutines, users and resource managers in our modular algorithm.

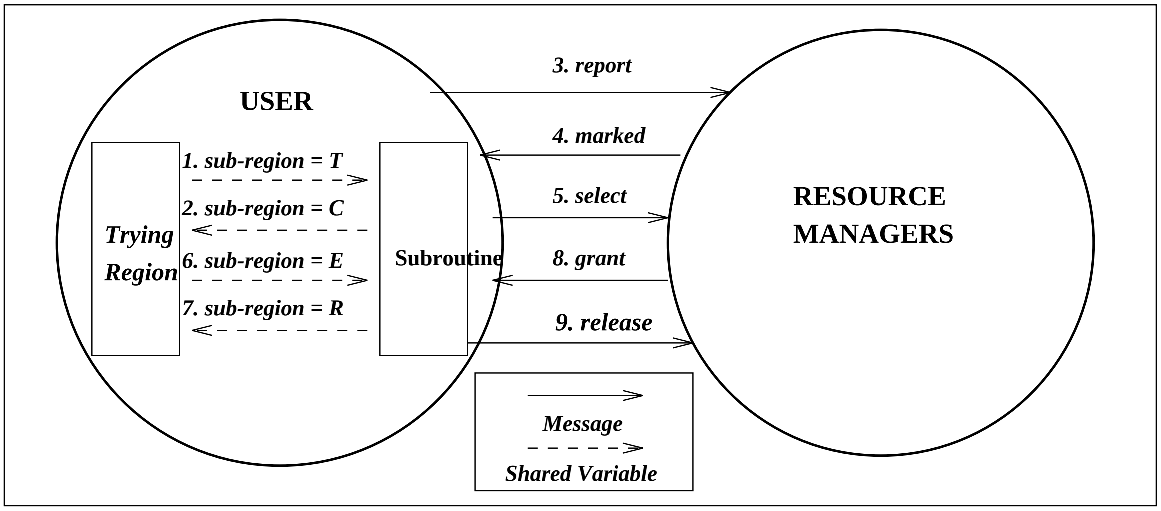

Fig. 16. The Algorithm Structure (The number represents the ordering of events). *Trying*, *Critical*, *Exit* and *Remainder* are abbreviated to T, C, E and R to save space. Dotted arrows represent using the value of shared variable *sub-region* as input to the entity toward which the arrow is directed. Solid arrows represent message passing.

The following state variables are used in the resource manager code (see Figure 17).

- $need_i$: equal to R_i for any i.
- $C_k(p)$: the counter for position p of q_k. Initially 0.
- $occupant_k(p)$: contains information about the ID of the user that occupies position p of q_k. 0 if the position is unoccupied. We assume there is no user with ID zero.
- $critical_k$: a boolean variable; true iff after rm_k received a *report* message from some i, it hasn't received any *select* message. Intuitively, when $critical_k$ is true, it means that the user is in the critical region of the subroutine. Initially false.

Protocol for a resource manager rm_k

$\square$ $\neg critical_k$, receive $report_{i,k}$ $\rightarrow$
  $marked_{k,i} := \{j-1, j : \forall j, occupant_k(j) \neq 0\}$;
  send $marked_{k,i}$;
  $critical_k := true$;
$\square$ $critical_k$, receive $select(p, R_i)_{i,k}$, $\rightarrow$
  $critical_k := false$;
  $occupant_k(p) := i$;
  $need_i := R_i$;
  $C_k(p) := C_k(p) + |need_i|$;
  $decremented_k(p) := false$;
  if $p = 1$ then send $grant_{k,i}$;
  else `adjust_queue(`p`);`
$\square$ $\neg critical_k$, receive $dec(p)_{i,k}$ $\rightarrow$
  $C_k(p) := C_k(p) - 1$;
  `adjust_queue(`p`);`
$\square$ $\neg critical_k$, receive $release_{i,k}$ $\rightarrow$
  $occupant_k(1) := 0$;
  `adjust_queue(2);`

PROCEDURE `adjust_queue(`p`)`
while $(occupant_k(p) \neq 0)$ and $(occupant_k(p-1) = 0)$
    if $(decremented_k(p) = false)$ then
      $C_k(p) := C_k(p) - 1$;
      $decremented_k(p) := true$;
      for all $k' \in need_j$: send $dec(p)_{k,k'}$;
    endif
    if $(C_k(p) = 0)$ then `advance_one_position(`p`);`
    $p := p + 1$;
end while

PROCEDURE `advance_one_position(`p`)`
  $occupant_k(p-1) := occupant_k(p)$;
  $decremented_k(p-1) := false$;
  $occupant_k(p) := 0$;
  $decremented_k(p) := false$;
  $C_k(p-1) := C_k(p-1) + \mid need_{occupant_k(p-1)} \mid$;
  if $(p - 1 = 1)$ then send $grant_{k,occupant_k(p-1)}$;
  if $(occupant_k(p-2) = 0)$ then
    $C_k(p-1) := C_k(p-1) - 1$;
    $decremented_k(p-1) = true$;
    for all $k' \in need_{occupant_k(p-1)}$: send $dec(p-1)_{k,k'}$;

Fig. 17. The Code of Resource Manager rm_k

- $decremented_k(p)$: a boolean variable: true iff $occupant_k(p) \neq 0$ and $occupant_k(p-1) = 0$. Initially false.

10.3 Correctness Proof and Performance Analysis

For convenience of presentation, we first define some terms. We denote by i_k a user i such that user i occupies a position in q_k, for any k. If a user i occupies or advances to a position p in q_k, for any p, k, then we say that i_k occupies or advances to p.

If rm_k sends a $dec(p)$ for any p while i occupies position p in q_k, then instead of saying that the resource manager sends the dec messages, we say more conveniently that i_k *sends* $dec(p)$. When rm_k receives a $dec(p)$ from the other resource manager and position p is occupied by i_k, we say more conveniently that i_k receives $dec(p)$.

Lemma 10.3.1 *In any timed execution of the algorithm, for any users i and j, i_k and j_k do not occupy the same position at the same time.*

Proof: Without loss of generality, we assume that user i selects a position in q_k after user j does (i.e., user i enters the critical region of the subroutine after user j does). Note that they cannot select a position at the same time because of the exclusion condition of the subroutine.

When user i selects a position in q_k, it receives a *marked* message from rm_k, which contains all the occupied positions in q_k. $critical_k$ in the resource manager's code is true when rm_k receives a *report* message from user i, which is sent only after the user enters the critical region of the subroutine. Since other conflicting users cannot be in the critical region at the same time, rm_k doesn't receive a *select* message until user i sends it. Thus, while user i is selecting a position in q_k, $critical_k$ is always true until $select_{i,k}$ is received, and therefore, the positions of other users in q_k do not change. This guarantees that i never selects the position that j_k selected since user i selects a position that are unoccupied in q_k.

When i_k advances to a new position only after its immediately preceding position is unoccupied (see statements 19 and 25), it never happens that i_k advances to the position that j_k occupies or vice versa. ∎

Theorem 10.3.2 *(Exclusion) In any timed execution of the algorithm, if users i and j are both in their critical regions at time t, then i and j do not conflict with each other at time t.*

Proof: Any two conflicting users select different positions at q_k if they share a resource k because of the exclusion property of the subroutine and because $critical_k$ is false while a user is selecting a position. By Lemma 10.3.1, i_k and j_k never occupy the same position at the same time. By the code, only the users at position 1 will be in the critical region. Thus, if users occupy different positions in a queue, there is no such case that they are in the critical region at the same time. ∎

We now prove the no-lockout condition. The structure of the proof is as follows. We first prove in Corollary 10.3.8 that a user i_k advances one position from a position p if and only if position $p-1$ in all $q_{k'}$, $k' \in R_i$, is unoccupied. This corollary implies that all i_k's are marching "almost" in synchrony whenever position $p-1$ in all $q_{k'}$, $k' \in R_i$, is unoccupied because initially every i_k occupies the same position in every q_k: the distance between any i_k and $i_{k'}$ is at most 1. Then we prove in Lemma 10.3.10 that if a user j_k occupies a position lower than i_k, it is always the case that $j_{k'}$ occupies a position lower than $i_{k'}$, i.e., there is no such situation that while a user i_k is at position 2 and a user j_k is at position 1, $i_{k'}$ is at 1 and $j_{k'}$ is at 2. Corollary 10.3.8 and Lemma 10.3.10 together guarantee that all users that occupy position 1 at any queue will eventually enter the critical region and leave the queues. Also Corollary 10.3.8 implies that after all users that occupy position 1 at any queue leave the queues, all other users in the queues advance one position. Inductively applying this argument, we can prove that all users in the queue eventually advance to position 1. This proves the no-lockout condition of the algorithm because all users in the trying region will eventually occupy a position in the queues of their required resources by the no-lockout condition of the subroutine, and they advance to position 1 where they are granted all of the required resources.

To prove Corollary 10.3.8, we need several other lemmas. In Lemma 10.3.3, we prove that i_k sends exactly one $dec(p)$ message only when position $p-1$ in q_k is unoccupied and p is occupied by i_k. message. In Lemma 10.3.4, we prove that every $dec(p)$ messages sent by $i_{k'}$, $k' \in R_i$ and $k' \neq k$, is received by rm_k while (i_k is at position p) or (i_k is at position $p+1$ and position p in q_k is unoccupied). According to the code, $|need_i|$ $(= R_i)$ is added to $C_k(p)$ when i_k advances to p and whenever rm_k receives $dec(p)$, $C_k(p)$ is decremented by one. Therefore, by Lemmas 10.3.8 and 10.3.4, it can be proved that $C_k(p)$ will become (eventually) zero when all $p-1$

positions in every q_k, $k \in R_i$, are unoccupied. This is proved in Lemma 10.3.5. In Lemma 10.3.7, we prove that i_k advances to position $p-1$ whenever $C_k(p) = 0$ and i_k occupies position p. This proves the corollary that when all all $p-1$ positions in every q_k, $k \in R_i$, are unoccupied and i_k occupies position p, i_k (eventually) advances to $p-1$.

In any timed execution of the algorithm, let t_1 be a time that i_k becames the occupant of position p and let $[t_1, t_2]$ the time interval that i_k occupies position p (in case that i_k never leaves p, which we proves doesn't happen, t_2 is ∞).

Lemma 10.3.3 *The followings are true:*

(1) either (not both) statement group (21, 22 and 23) or statement group (35, 36 and 37) in the code of resource manager rm_k is executed at most once in the interval $[t_1, t_2]$.

(2) either (not both) statement group (21, 22 and 23) or statement group (35, 36 and 37) in the code of resource manager rm_k is executed at least once in the interval $[t_1, t_2]$ if and only if there exists a (finite) time t, $t_1 \leq t \leq t_2$, that $occupant_k(p-1) = 0$ and $occupant_k(p) = i$.

Proof: (1) the statement is true because of the code. Either (not both) statement group (21, 22 and 23) or (35, 36 and 37) is executed *at most* once while user i_k is at position p because $decremented_k(p)$ is set to true at statements 22 or 36 at time t_1 and is not reset to false until another user occupies position p.

(2) (**Only if part**:) trivially true because of the conditions at statements 19 and 34.

(**If part**:) if there exists a (finite) time t, $t_1 \leq t \leq t_2$, that $occupant_k(p-1) = 0$ and $occupant_k(p) = i$, either (not both) statement group (21, 22 and 23) or statement group (35, 36 and 37) in the code of resource manager rm_k is executed at least once in the time interval $[t_1, t_2]$.

There are two possible cases: the first case is that $t = t_1$ (in words, position $p-1$ is already unoccupied when user i_k occupies position p at time t_1) and second, $t_1 < t$ (in words, position $p-1$ is occupied when user i_k occupies position p at time t_1, but later on position $p-1$ becomes unoccupied at time t).

Case 1: $t = t_1$. When user i_k occupies position p at time t_1 (either advances to position p or selects position p), if position $p-1$ is already unoccupied (i.e., it was unoccupied before time t_1), we prove that either statement group (21, 22 and 23) or (35, 36 and 37) is executed at least once in the interval $[t_1, t_2]$.

i_k gets to occupy position p when the guarded command at statement 5 is executed or the `advance_one_position`$(p+1)$ is executed at statement 25. When the guarded command at statement 5 is executed, since $p \geq 2$, `adjust_queue` is called, in which statement group (21, 22 and 23) is executed while $occupant_k(p) = i$ because the condition in the while loop at statement 19 is true if $p-1$ is unoccupied, and $decremented_k(p)$ must be false because of statement 10.

When the `advance_one_position`$(p+1)$ is executed at statement 25, if position $p-1$ is unoccupied, the condition 34 will be true. Thus statement group (35, 36 and 37) is executed while $occupant_k(p) = i$.

Case 2: $t_1 < t$. In case that position $p-1$ is occupied when user i_k occupies position p at time t_1, but later on position $p-1$ becomes unoccupied at time t, we prove statement group (21, 22 and 23) or (35, 36 and 37) is executed at least once after time t_1 (i.e., while $occupant_k(p) = i$).

When position $p-1$ becomes unoccupied, `advance_one_position(`$p-1$`)` at statement 25 is executed. After that, the condition in the while loop at statement 19 will be true because i_k is at position p and p is just unoccupied. Note that $decremented_k(p)$ must be false because it is set to false when i_k first occupies position p and $p-1$ have never been unoccupied since. Therefore, statement group (21, 22 and 23) is executed.

■

Lemma 10.3.4 *If a $dec(p)$ is received when either (i_k is at position p) or (i_k is at position $p+1$ and position p at q_k is unoccupied), the $dec(p)$ must be sent by $i_{k'}$, $k \neq k'$.*

Proof: Hypothesis:
Assume, by way of contradiction, that a message $m = dec(p)$ received when either (i_k is at p) or (i_k is at $p+1$ and p is unoccupied) is sent from $j_{k'}$, $i \neq j$ at position p in $q_{k'}$. Without loss of generality, assume that m is the first such message in the execution.

We first prove that if m is the first such message in the hypothesis and m is sent by $j_{k'}$, j_k cannot be in a position less than or equal to p and must be a position less than p in q_k. Then we also prove that j_k cannot be in a position bigger than p. This is a contradiction. Thus, m is not sent by $j_{k'}$.

When a user advances from a position p to $p-1$ in queue q_k, $C_k(p)$ must be zero (see statement 25). By the time that position p is occupied by another user, say, l, $C_k(p)$ is never incremented according to the code. When it is incremented, it is incremented by $|need_l|$. Therefore, for $C_k(p)$ to be zero again (at which time l_k advances to p), the total number of $dec(p')$ messages received while (l_k is at position p) and (l_k is at position $p+1$ and position p at q_k is unoccupied) should be $|need_l|-1$ ("-1" because according to Lemma 10.3.3, when position $p-1$ is unoccupied while l_k is at position p, $C_k(p)$ is decremented exactly once).

j_k cannot be at position less than p because of the following reason. j_k must once occupy position p in q_k because $j_{k'}$ should be at position p in $q_{k'}$ when it sends m, and initially all j_l's, $l \in R_i$, occupy the same position in all q_l. If j_k is at a position less than p, it must have received $|need_j| - 1$ $dec(p)$ messages. Since m sent by $j_{k'}$ is not received before j_k advances to $p-1$, there must be at least one message that is sent by some other user while (j_k is at position p) or (j_k is at position $p+1$ and position p at q_k is unoccupied). This means m is not the first such message in the hypothesis, which is a contradiction. Therefore, j_k cannot be at position less than p or leave q_k. This also implies that j_k should be in q_k at position bigger than or equal to p in q_k. However, we prove that j_k is not in q_k.

Since m is the first such message in the execution, before m is received, the claim must have been true.

Case 1: m is received when i_k is at p.

In this case, j_k must be at position bigger than or equal to $p+1$ because p is occupied by i_k. Because $j_{k'}$ must have been at position p in $q_{k'}$ to send m, it implies that $j_{k'}$ advanced from $p+1$ to p in $q_{k'}$. It also implies that $j_{k'}$ received a $dec(p)$ from

j_k because otherwise, m is not the first such message in the hypothesis. But because i_k occupies p, j_k cannot send the $dec(p)$ message. This is a contradiction.

Case 2: m is received when i_k is at $p+1$ and p at q_k is unoccupied.

The same argument can apply here. j_k must be at position at least $p+2$ because j_k cannot be at a position less than p; position p is unoccupied; and position $p+1$ is occupied by i_k. Because $j_{k'}$ must be at position p in $q_{k'}$ to send m, it implies that $j_{k'}$ advanced from $p+1$ to p in $q_{k'}$. It also implies that $j_{k'}$ received a $dec(p)$ from j_k because otherwise, m is not the first such message in the hypothesis. But because i_k occupies p, j_k cannot send the $dec(p)$ message. This is a contradiction. ∎

Lemma 10.3.5 *There exists a time t ($t_1 \leq t \leq t_2$) that $C_k(p) = 0$ if and only if there exists a time t' ($t_1 \leq t' \leq t_2$) that rm_k receives $|need_i| - 1$ $dec(p)$ messages, each message sent from a $i_{k'}$, $k \neq k'$, and $occupant_k(p-1) = 0$.*

Proof: Let t'' be the last time before time t_1 that p in q_k became unoccupied and let j_k be the user that occupied p just before t'' (if it has never been occupied before t, let $t'' = 0$).

Claim 10.3.6 *In time interval $[t'', t_2]$, rm_k does not receive a $dec(p)$ from any users other than $i_{k'}$, $k \neq k'$, $k' \in R_i$.*

Proof: Lemma 10.3.4 implies that in the interval $[t_1, t_2]$, i_k doesn't receive any $dec(p)$ sent by a user $j_{k'}$, $i \neq j$. We only need to prove that it does not receive a $dec(p)$ from $j_{k'}$ in time interval $[t'', t_1)$.

Suppose that i_k receives such a message in the interval $[t'', t_1)$ that was sent by $j_{k'}$, $i \neq j$. By Lemma 10.3.4, j_k must be either at position p or (j_k must be at position $p+1$ and p in q_k is unoccupied) at the time such $dec(p)$ was received.

j_k cannot be at position p at the time because i_k is at position p from time t_1; i_k and j_k cannot be at the same position by Lemma 10.3.1; and t'' is the last time position p becomes unoccupied before time t_1. So, j_k cannot be at position p in the interval $[t'', t_1)$.

For the second case, if j_k is at position $p+1$ and position p is unoccupied, then i_k is either at position at least $p+2$ or i_k is not in q_k at the time that j_k occupies $p+1$ because i_k is at position p at time t_1 and j_k is at $p+1$ in time interval $[t'', t_1)$.

Both imply that j_k once occupied p and then unoccupied p after time t'' because according to the code, all users in a queue advance one position at a time and no new user can occupy a position immediately preceding an occupied position. But, time t'' is the last time that p is unoccupied. This is a contradiction. So, it is not true that j_k is at position $p+1$ and p is unoccupied.

This contradicts Lemma 10.3.4 because rm_k receives a $dec(p)$ message from $j_{k'}$ in the interval $[t'', t_1)$. ■

$C_k(p)$ is not incremented at all in the interval $[t'', t_1)$ because it is only incremented when a user first occupies the position and no user occupies the position in the interval $[t'', t_1)$. $C_k(p)$ is added to $|need_i|$ when i_k occupies position p at time t. $C_k(p)$ is decremented only when a $dec(p)$ is received (statement 14) or when p in q_k is occupied and $p-1$ in q_k is unoccupied (by Lemma 10.3.3).

(**Only if part**): If there exists a time t ($t_1 \leq t \leq t_2$) that $C_k(p) = 0$, there exists a time t' ($t_1 \leq t' \leq t_2$) that rm_k receives $|need_i| - 1$ $dec(p)$ messages, each message sent from a $i_{k'}$, $k \neq k'$, and $occupant_k(p-1) = 0$.

By Claim 10.3.6, a $dec(p)$ received after time t'' and before i_k unoccupies p (i.e., before time t') must be sent by $i_{k'}$. Thus, the fact that $C_k(p) = 0$ at time t (which is after i_k occupies p) implies that rm_k receives $|need_i| - 1$ $dec(p)$ messages because each $i_{k'}$ sends only one $dec(p)$ message by the code, there are $|need_i| - 1$ such $i_{k'}$; $occupant_k(p) = i$; and $occupant_k(p-1) = 0$. Therefore, there exists such time t'.

(**If part**): There exists a time t ($t_1 \leq t \leq t_2$) that $C_k(p) = 0$, if there exists a time t' ($t_1 \leq t' \leq t_2$) that rm_k receives $|need_i| - 1$ $dec(p)$ messages, each message sent from a $i_{k'}$, $k \neq k'$, and $occupant_k(p-1) = 0$.

Note that at time t'', $C_k(p)$ must be zero because according to the code, only when $C_k(p) = 0$, p becomes unoccupied, and rm_k doesn't receive a $dec(p)$ from $i_{k'}$ before time t'' because of Lemma 10.3.4. Thus, if rm_k received $|need_i| - 1$ $dec(p)$ (each from a $i_{k'}$) and $occupant_k(p-1) = 0$ at time t', $C_k(p)$ is zero. ■

Lemma 10.3.7 *When i_k occupies position $p \geq 2$,* `advance_one_position(`p`)` *is executed if and only if $C_k(p) = 0$.*

Proof: (**Only if**): trivially true by the code. See statement 25.

(**If part**): if $C_k(p) = 0$, then statement 25 is executed.

Only within `adjust_queue()`, i_k can possibly advance to position $p-1$ after $C_k(p)$ is tested for zero (see statement 25). It is sufficient to prove that `adjust_queue(`p'`)`, $p' \leq p$, is called at all possible cases that $C_k(p)$ can be zero in the code because whenever `adjust_queue`(p') is called, position p can be checked in the while loop at statement 19 because $p' \leq p$.

We need to examine all the places in the code where changes to $C_k(p)$ are made. In the code, changes to $C_k(p)$ are made at statements 9,14,21 and 32.

<u>Case 1:</u> statement 9 is executed.

Since $p \geq 2$, `adjust_queue`(p) is called. At statement 19, the condition in the while loop is true because i_k occupies position p and by Lemma 10.3.5, if $C_k(p)$ is zero, $p-1$ must be unoccupied. Thus statement 25 is executed.

<u>Case 2:</u> statement 14 is executed.

`adjust_queue`(p) is called. The same argument as in the case of statement 9 applies to this case.

<u>Case 3:</u> statement 21 is executed.

Statement 25 is executed next.

<u>Case 4:</u> statement 32 is executed.

When statement 32 is executed (i.e., $C_k(p'-1) := C_k(p-1) + |need_{occupant_k(p-1)}|$ where $p' = p+1$), the value of $C_k(p)$ depends on the value of $need_i$. We consider two possible cases concerning the value of $need_i$.

Case 4.1: $|need_i| = 1$.

It is sufficient to prove that there is always some user at position $p-1$ when i_k advances to p (i.e., when statement 28 is executed) because in that case, $C_k(p)$ cannot be zero by Lemma 10.3.5.

When user i selects a position p' in q_k initially, there must be some user j_k at position $p' - 1$. Otherwise, user i can select $p' - 1$ instead of p'. Note that positions in q_k cannot change while user i is selecting because $critical_k$ is true while i is selecting. When j_k advances to $p' - 2$, i_k also advances to $p' - 1$ because when `advance_one_position(`$p' - 1$`)` is executed at statement 25, position p' is always checked for possible advancement of i_k because of the while loop at statement 19. Since $C_k(p' - 1)$ is equal to one (because it is occupied by i_k; $|need_i| = 1$; and by Lemma 10.3.4, no $dec(p)$ messages from other $j_{k'}$ can be received) and $decremented_k(p')$ must be false (because j_k was occupying position $p' - 1$), statement 21 will be executed which makes $C_k(p' - 1)$ zero. Thus, `advance_one_position(`p'`)` will be executed to advance i_k to position $p' - 1$. Therefore, j_k is always at a position immediately preceding i_k's position. By Lemma 10.3.3, if position $p - 1$ is occupied while i_k is at position p, $C_k(p)$ will never be decremented. Thus, $i_k(p)$ cannot be zero after statement 32 is executed.

Case 4.2: $|need_i| > 1$.

Since $|need_i| > 1$, it is possible that i_k receives a $dec(p)$ before it advances to position $p - 1$.

i_k advances to position p either when it receives a $dec(p+1)$ message sent by $i_{k'}$ for some k' (statement 13) or when it finds that position $p - 1$ is unoccupied (statement 21).

Suppose that i_k advances to position p when it receives a $dec(p+1)$ message sent by $i_{k'}$. It is possible that $i_{k'}$ send a $dec(p)$ after it advances to p in $q_{k'}$. But, if i_k advances to position p when it receives the $dec(p + 1)$, then the $dec(p)$ would not be received yet, because it has just received the $dec(p+1)$ from $i_{k'}$ and the communication is FIFO. Therefore, by Lemma 10.3.5, $C_k(p)$ cannot be zero because the $dec(p)$ from $i_{k'}$ hasn't been received.

Suppose that i_k advances to position p when it finds that position p is unoccupied at statement 21. i_k advances to position p, after it sends a $dec(p+1)$ at statement 23. By Lemma 10.3.4, $i_{k'}$ cannot be at position p because if it is, when the $dec(p+1)$ is received, $i_{k'}$ is at p contradicting Lemma 10.3.4. Therefore, $i_{k'}$ hasn't sent a $dec(p)$ yet when statment 23 is executed. Thus by Lemma 10.3.5, $C_k(p)$ cannot be zero. ■

Corollary 10.3.8 *If and only if there exists a time t ($t \geq t_1$) that for all l, $l \in R_i$, $occupant_l(p-1) = 0$ and $occupant_l(p) = i$, rm_k executes* `advance_one_position(`p`)` *at time t_2 and t_2 is a finite constant.*

Proof: (**If part**): if there exists a time t ($t \geq t_1$) that for all l, $l \in R_i$, $occupant_l(p-1) = 0$ and $occupant_l(p) = i$, then rm_k executes `advance_one_position(`p`)` at time t_2 and t_2 is a finite constant.

By Lemma 10.3.3, i_l will send exactly one $dec(p)$ message if position $p-1$ is unoccupied while i_l is at position p. By Lemma 10.3.5, $C_k(p)$ will be zero because rm_k will receive all $dec(p)$ messages sent by each $i_{l'}$, $l' \in R_i$ and $l' \neq k$ and position $p-1$ in q_k is unoccupied. By Lemma 10.3.7, if $C_k(p) = 0$, `advance_one_position(`p`)` is executed eventually.

(**Only if part**): There exists a time t ($t \geq t_1$) that for all l, $l \in R_i$, $occupant_l(p-1) = 0$ and $occupant_l(p) = i$ if rm_k executes `advance_one_position(`p`)` at time t_2 and t_2 is a finite constant.

If rm_k executes `advance_one_position`(p), by Lemma 10.3.7, $C_k(p) = 0$. If $C_k(p) = 0$, by Lemma 10.3.5, rm_k will receive all $dec(p)$ messages sent by each $i_{l'}$, $l' \in R_i$ and $l' \neq k$ and position $p-1$ in q_l is unoccupied. ∎

We define $P_k(i)$ to be the position of user i in q_k.

Corollary 10.3.9 *For any user i and any k and $k' \in R_i(t)$, where t is some time that i is in the trying region, $|P_k(i) - P_{k'}(i)| \leq 1$.*

Proof: When user i occupies a position initially in any queue, all i_k's, for $k \in R_i(t)$ occupy the same position in all q_k. By Lemmas 10.3.5 and 10.3.7, all i_k's advance one position from p to $p-1$ for some p only after $dec(p)$'s from all $i_{k'}$'s are received. Since i_k sent $dec(p)$ only once, if i_k is stuck at position p, $i_{k'}$ cannot advance to position $p-2$. Therefore, the corollary follows. ∎

For any k, i and j, $i \neq j$, if $P_k(i) > P_k(j)$, then we say that user i_k *waits for* user j_k, denoted by $i_k \rightarrow j_k$.

Lemma 10.3.10 *There is no cycle in the wait-for relation created by the execution of the algorithm.*

Proof: We show that if $i_k \rightarrow j_k$ at time t, $i_{k'} \rightarrow j_{k'}$, for any resources k and k' that users i and j share.

It is trivially true if $k = k'$. Assume $k \neq k'$. Because of the exclusion property of the subroutine algorithm, i and j are never scheduled at the same time. Without loss of generality, assume that i selects a position before j does.

If j_k occupies a position lower than $P_k(i)$, $j_{k'}$ should also occupy a position lower than $P_{k'}(i)$ because when j selects a position in q_k and $q_{k'}$ initially, from the code, it is true that $P(j_k) = P(j_{k'})$, and by Corollary 10.3.9, $\mid P(i_k) - P(i_{k'}) \mid \leq 1$. Therefore, if $i_k \rightarrow j_k$, then $i_{k'} \rightarrow j_{k'}$. It is now easy to show that the algorithm ensures the transitivity of the wait-for relation. This is sufficient to show there are no such cycles since a user cannot wait on itself.

■

Theorem 10.3.11 *(No-lockout) If a user is in the trying region or in the exit region, then it leaves its current region in finite time assuming no user remains in the critical region forever.*

Proof: By the no-lockout condition of the subroutine, the user will eventually select a position in all queues of the resources that it requires. By Corollary 10.3.9 and Lemma 10.3.10, and because no user remains in the critical region forever, all position 1's will be unoccupied eventually (if a user i_k is at position 1 at time t, for all $k' \in R_i(t)$, $i_{k'}$ will be eventually at position 1 and leave its queue in finite time).

Because positions immediately preceding all occupied positions will not be occupied by any new user, all users at position 2 will advance to position 1 by Corollary 10.3.8. If we apply this argument inductively, we prove that every user in every queue will advance at least one position after all position 1's become unoccupied. Since all users leave the critical region and leave their queues eventually, all users will advance to position 1 eventually. This proves that all users in the trying region leave the region in finite time.

Since in the exit region, users only send *grant* messages and then enter the remaining region immediately, users leave the exit region in finite time. ■

Theorem 10.3.12 *(Concurrency) The set of every timed execution of the algorithm does not include an execution where a user i enters the trying region at some time t; no conflicting user is in the trying region and the critical region at time t; after time t, no conflicting users enter the trying region; and user i remains in the trying region forever.*

Proof: If a user i is in the trying region, it will eventually enter the critical region of the subroutine due to its liveness condition, and since there is no conflicting user from time t on, all q_k, $k \in R_i(t)$, are empty and by the code, user i will select position 1 in all q_k and enter the critical region. ■

Theorem 10.3.13 *The algorithm is a drinking philosophers algorithm if the subroutine is a dining philosophers algorithm, a drinking philosophers algorithm or a dynamic resource allocation algorithm.*

Proof: The user code specified in Figure 15 is a resource allocation part and satisfies conditions 1 and 3 in the well-formedness definition in Section 8.2. The code can be used as a subroutine for any other code (non-resource allocation part) that does not violate the conditions 2 and 4 in the well-formedness definition. Therefore, the code is well-formed. If the subroutine is a dining philosophers algorithm, the resource requirement of the subroutine is always $\mathcal{R}_i$. Otherwise, it is set to the current resource requirement (R_i) of the user in the code. Therefore, the subroutine should work correctly by definition. Since the code satisfies the exclusion condition, the no-lockout condition and the concurrency condition, the algorithm is a drinking philosophers algorithm. ■

Theorem 10.3.14 *The algorithm is a dynamic resource allocation algorithm if the subroutine is a dynamic resource allocation algorithm.*

Proof: As proved in the proof of Theorem 10.3.13, the algorithm is well-formed. If the subroutine is a dynamic resource allocation algorithm, it doesn't require any prior information about $\mathcal{R}_i$. Since the code described in Figures 15 and 17 doesn't require any a priori information about $\mathcal{R}_i$, and the algorithm satisfies the exclusion condition, the no-lockout condition and the concurrency condition, the algorithm is a dynamic resource allocation algorithm. ■

To measure the response time of the algorithm, we define functions $Resp_A(c)$ and Msg_A to be the response time and the message complexity of a resource allocation algorithm A.

Theorem 10.3.15 *If a resource allocation algorithm A is used as a subroutine for our algorithm, then our algorithm has response time $O(\delta c + \delta^2 d + Resp_A(d))$ and message complexity $O(Msg_A + r^2\delta)$.*

Proof: For the response time, we only need to show that any user i that sends a *request* message at time t will receive *grant* messages from all resource managers rm_k, $k \in R_i(t)$ by time $t + O(\delta c + \delta^2 d + Resp_A(d))$.

Note that for a message to be received, it takes time $d+1$ in the worst case (the maximum message delay + one step to receive the message). The maximum time period that a user spends in the critical region of the subroutine is $3d+3$, i.e., sending a *report* message, receiving a *marked* message and sending a *select* message. Since the response time of the subroutine is $Resp_A(c)$ and in the subroutine, $c = 3d+3$, a user will enter the critical region of the subroutine within time $Resp_A(3d+3)$.

Let p be the initial position of user i. There are at most $\delta - 1$ other users in the queues of resources that user i requires. By Corollary 10.3.9, if a user j_k occupies a position p', either (all $j_{k'}$, $k' \neq k$, must occupy p or $p-1$) or (all $j_{k'}$, $k' \neq k$, must occupy p or $p+1$). Thus, no user occupies more than two positions. By the code, the position immediately preceding any occupied position is not considered for the selection. Therefore, $p \leq 3\delta$.

We now show that it takes at most $O(pc + p^2 d)$ time for any user i to leave a queue after it occupies position p.

It can be shown by an easy induction on p that if position 1's of all queues are unoccupied at time t', then all users at position $p > 1$ will advance to $p-1$ within time $t' + p(d+1)$. Assume, by way of induction, that all users at position $p-1$ advance one position to position $p-2$ within time $t' + (p-1)(d+1)$. When all users at $p-1$ advance, all users at position p send $dec(p)$ messages, which will be received within time $d+1$. All users at position p will advance to $p-1$ by time $t' + (p-1)(d+2) + d + 2$.

After all j_k, $k \in R_j$, occupy position 1 in all q_k, it takes $c + 2d + 2$ time for all

rm_k to receive a *release* message from user j, i.e., time $d+1$ for a *grant* message to be received by j; time c for j to be in the critical region; and time $d+1$ for a *release* message to be received. Thus, after a user i occupies position p, all users at position 1 will leave the queue within time $c+2d+2$. It follows that within time $t + Resp(3d+3) + c + 2d + 2 + p(d+1)$, all i_k occupy position $p-1$ or less.

Since there are $p-1$ positions to advance, within time $t + Resp(3d+3) + (p-1)(c+2d+2+p(d+1))$, user i advances to position 1. Since $p \leq 3\delta$, user i enters the critical region by time $t + Resp(3d+3) + (3\delta - 1)(c + 2d + 2 + 3\delta(d+1))$ plus the time to receive a *grant* message $(d+1)$. Therefore, user i enters the critical region by time $t + O(Resp(d) + \delta c + \delta^2 d)$.

For the message complexity, after a user occupies a position in a queue, it sends $O(r)$ *dec* messages in the position, where r is the the number of resources that a user requires at a time. Since a user occupies at most 2δ positions, it sends $O(r\delta)$ messages after it leaves the trying region of the subroutine A. A user sends at most $O(Msg_A + r^2\delta)$ before it actually uses its required resources in the critical region. ∎

CHAPTER 11

Simulation

We are interested in comparing the average response time of different resource allocation algorithms. The *average response time* of a resource allocation algorithm is the ratio of the total response time over the total number of entrances to the critical region in all executions of the resource allocation algorithm. Because it is difficult to develop a simple analytical model that covers all the different system models used by various algorithms, we only obtain estimated average response times of given resource allocation algorithms using discrete event simulation. *Discrete event simulation* is the modeling of systems in which the state variable changes only at a discrete set of points in time [BC84].

We simulated only dining philosophers algorithms. We didn't simulate drinking philosophers algorithms because most of drinking philosophers algorithms use a dining philosophers algorithm as a subroutine and therefore, we believe that our simulation of the dining philosophers algorithms brings some insight into the average performance of the drinking philosophers algorithms. We leave the simulation of dynamic resource allocation algorithms as future work. In this chapter, we present our simulation model in Section 11.1, implementation of the simulation in Section 11.2, the method used to analyze the simulation result in Section 11.3 and simulation results of various dining philosophers algorithms and compare their average response times in Section 11.4.

11.1 Simulation Model

We simplify the model in Chapter 2 to reduce the complexity of simulation and also to aid the measurement of the average response time. Each process is modeled in the same way as in Section 2. But, processes are executed synchronously so that at one time unit, each process executes one enabled guarded command (if no enabled guarded command, it takes an idle step). Message passing among processes is asynchronous so that each message has a random delay with uniform distribution, but is delivered in FIFO order.

Our simulation model has 100 users and 100 resources. The resource requirement of each process is set randomly prior to the execution of the simulation with an upper bound set by the control of the simulation, and the conflict graph is constructed based on the resource requirements of all users. We assume a uniform probability distribution of the resource requirements of users (i.e., the probability that a user requires a resource a is the same as that it requires a resource b).

Each user process takes idle steps in the remainder region for some random time before it enters the trying region. We call the time period *thinking time*. There is also the time period that a process uses a resource, which we call *resource service time*. The resource service time for each resource is set randomly before a user enters the critical region. The time period that a user stays in the critical region is determined accordingly by the sum of the resource service times of all resources that the user need to use (i.e., resources in the resource requirement of the user). We also assume that a user takes idle steps in the critical region and the probability distribution of the thinking time and the resource service time is uniform within a bound set prior to the execution of the simulation by the control of the simulation.

11.2 Implementation of the Simulation Model

The simulation model is implemented using SUN/OS *Light Weight Processing* package (LWP) and C programming language. Each process is implemented as a light weight process. Communication among light weight processes is done through globally shared variables. Each (light-weight) process has a shared variable called *message buffer* that can be shared with other process. The message passing between processes is implemented by having a separate (light weight) process, called *network*, handle messages. When a process sends a message, it writes the message to the buffer of the network. Each message contains information about its sender, receiver and message delay. The network reads the message delay of the message and writes the message to the buffer of its receiver after the specified message delay for the message has passed. To implement the FIFO delivery, when a message is deposited to the buffer of the network, the message delay of the message is compared with the delivery time of the message that was previously sent by the sender. If the current message has to be delivered earlier than the previous message according to the assigned message delay, the message delay of the current message is reset to one time after the delivery time of the previous message.

The synchronous execution of processes is implemented by having another process, called *scheduler*, manage the program control. The scheduler, which initially has the program control, passes the control to other processes (including the network) in round robin order. When a process has a program control, it executes one of enabled guarded commands according to the fairness conditions specified in Section 2 and returns the control back to the scheduler. If a process has to take an idle step, it returns the control immediately to the scheduler.

The scheduler also maintains the simulation clock. It increments the clock by one after each round of executions of processes. Other processes can read the clock, but only the scheduler can update it. For example, the network reads the clock to decide when a message needs to be delivered.

The randomness based on the uniform probability distribution is implemented using a random number generator in SUN/OS, called `random()`. `random()` returns a pseudo-random (integer) number in the range from 0 to $2^{15} - 1$ with the uniform probability distribution. A random number in the range from 0 to a given bound B w can be obtained as follows:

$$\frac{\texttt{random()}}{2^{15}-1} \cdot B$$

11.3 Output Analysis

We measure the response time of each user whenever the user finishes its critical region. After running the simulation for some time, we compute the arithmetic average response time of the algorithm based on all the response times collected by that time. Given sampled response times $r_1, r_2, \ldots, r_N$ where N is the total number of users that enter their critical region, the estimated mean response time $\overline{\theta}$ is :

$$\overline{\theta} = \frac{1}{N}\sum_{i=1}^{N} r_i$$

The common technique [BC84] to increase the accuracy of the estimated mean response times is to adjust the run length of each simulation in such a way that the number of the sampled response times that are within the 95% confidence interval of the mean response time is less than 5% of the total number of the subsampled response times. This has an impact of increasing the sample size (N) of response times (explain below). The 95% confidence interval of the mean response time θ can be computed as follows, assuming a normal distribution:

$$\theta - 1.96\frac{\sigma}{\sqrt{N}} \leq \theta \leq \theta + 1.96\frac{\sigma}{\sqrt{N}}$$

where σ and θ are *true* mean and standard deviation of the response time.

Since we don't know σ and θ, we estimate the confidence interval with the estimated mean response time $\hat{\theta}$ and standard deviation $\hat{\sigma}$ as follows:

$$\hat{\theta} - 1.96\frac{\hat{\sigma}}{\sqrt{N}} \leq \theta \leq \hat{\theta} + 1.96\frac{\hat{\sigma}}{\sqrt{N}}$$

The 95% confidence interval implies that the probability that the difference between the mean of any sampled data of size n and the true mean values θ is less than $1.96\frac{\sigma}{\sqrt{n}}$ is at least 95%. Therefore, when we increase the run-length and collect more response times, the confidence interval will get smaller and thus, the accuracy will increase. Since our termination condition will increase the run-length if the confidence interval is not small enough, it has the effect of increasing accuracy of the measured mean response time.

In addition, to bound the run-length of the simulation in some reasonable degree to prevent unduly prolonged simulation, we introduce another termination condition: if the standard error ($\hat{\sigma}/\sqrt{N}$) of the measured mean response time is less than 5% of the measured mean response time, then we terminate the simulation. The rationale for the condition is that when the standard error is small enough, then increasing the sample size of the response time doesn't contribute much to reducing the confidence interval and doesn't affect much the resulting estimated mean response time.

11.4 Simulation Results

We simulated Chandy and Misra's algorithm (CM), Choy and Singh's algorithm (CS), Awerbuch and Saks' algorithm (AS), our modular algorithm with Choy and Singh's algorithm as a subroutine (CSR) and our modular algorithm with Chandy and Misra's algorithm as a subroutine (CMR). Approximately 300 random conflict graphs were tested. Figures 18, 19, 20 and 21 show the result of our simulations, where the average response times of the above-mentioned algorithms are shown as the number of conflicting users increases. To see the impact of the ratio of message delay over resource service time, we simulated the algorithms with four different ratios. We fixed the average message delay to be 50 time units and varied the resource service time from 25 to 100 with 25 time unit increment. The average think time is set to 25.

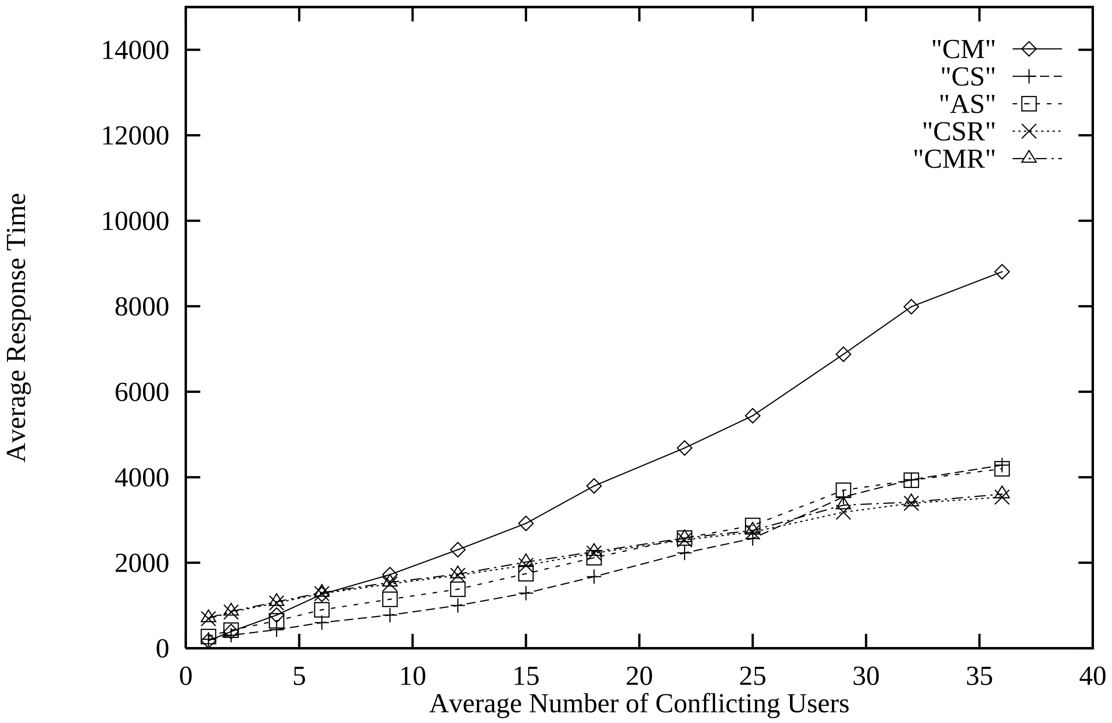

Fig. 18. Average response time of various dining philosophers algorithm when average resource service time = 25 and average message delay = 50.

When the ratio of the average response time over the average message delay is small (see Figure 18), all the algorithms except CM performed comparably well. CS performs better than all other algorithms tested when the average number of conflicting users is small. However, as the average number of conflicting users gets bigger, the average response time of CS grows more steeply than CSR, CMR and AS. This phenomenon becomes more evident when the ratio of the average resource service time over the average message delay gets bigger (compare Figure 18 with Figures 19, 20 and 21).

Our results show that AS, CSR and CMR perform clearly better than the other

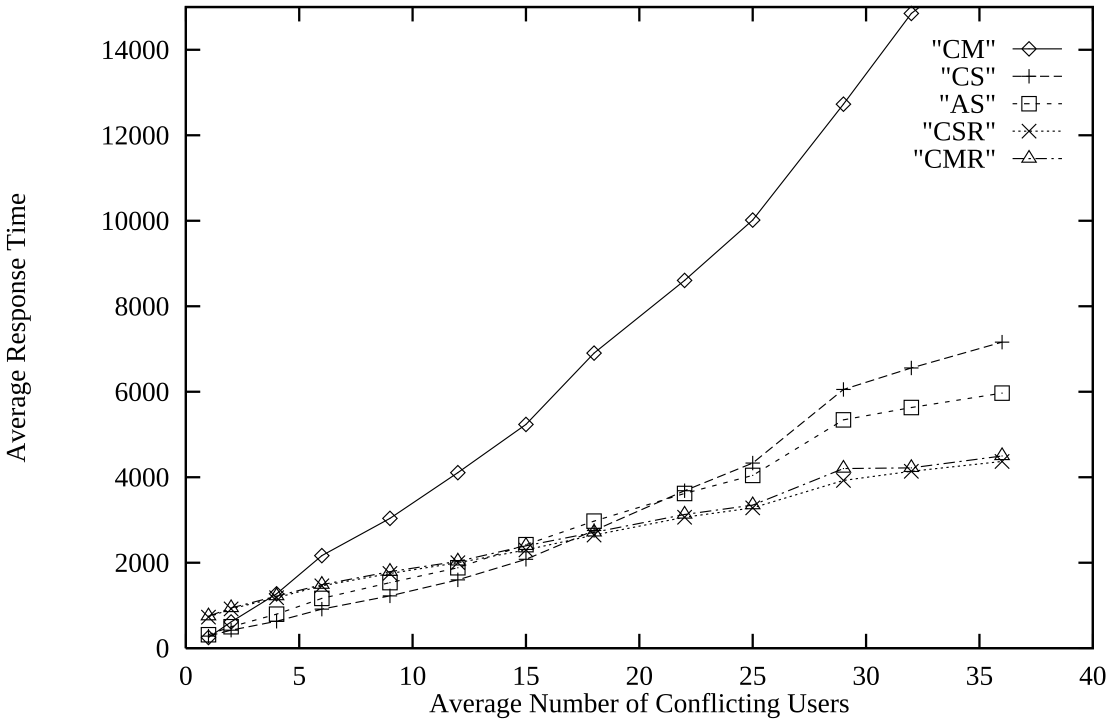

Fig. 19. Average response time of various dining philosophers algorithm when average resource service time = 50 and average message delay = 50.

algorithms (CS and CM) when the average number of conflicting users is big and the ratio of the average resource service time over the average message delay is high. These algorithms are similar in that their worst case response time has c applied only to δ. CMR and CSR show up to 60% improvement over CS and up to 80% improvement over CM.

CMR and CSR also show better average performance even than AS (up to 40%) as the ratio gets bigger. CMR works surprisingly better than CM in average.

Even though the ratio is small, if the average number of conflicting users is big, CMR and CSR show improvement over all the other algorithms (see Figures 18 and

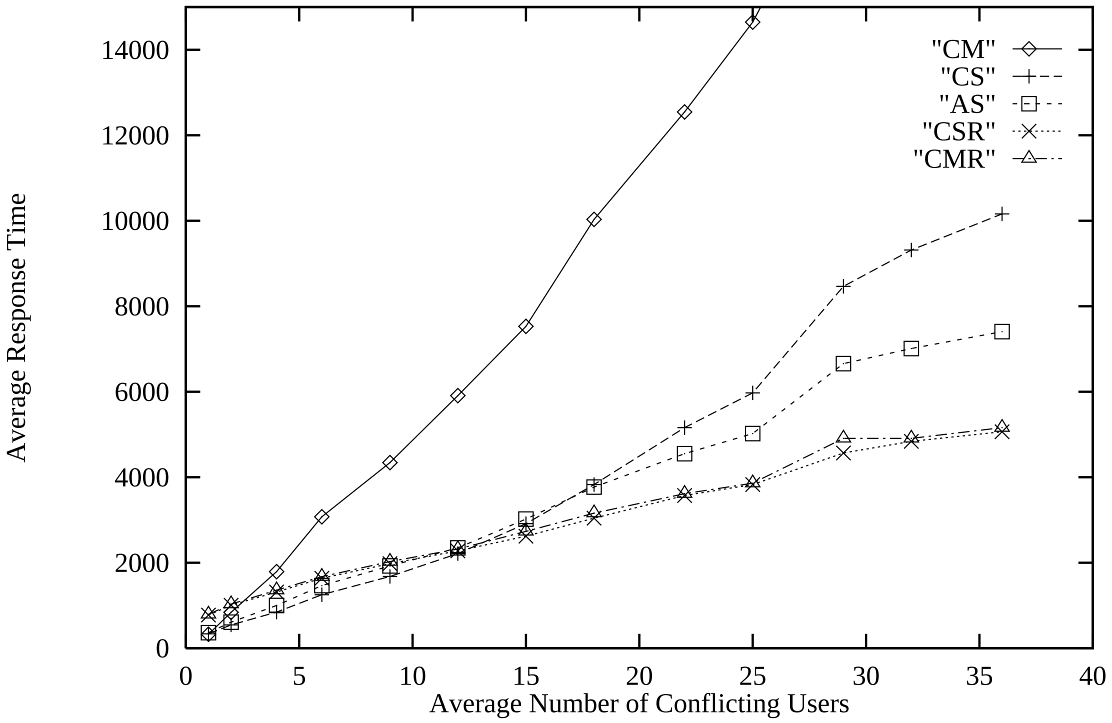

Fig. 20. Average response time of various dining philosophers algorithm when average resource service time = 75 and average message delay = 50.

19). Chandy and Misra's algorithm was worst among all in almost every case.

In summary, our simulation results clearly indicate that our modular algorithm can improve the overall average response time when there is high contention for resources and the message delay is smaller than the resource service time.

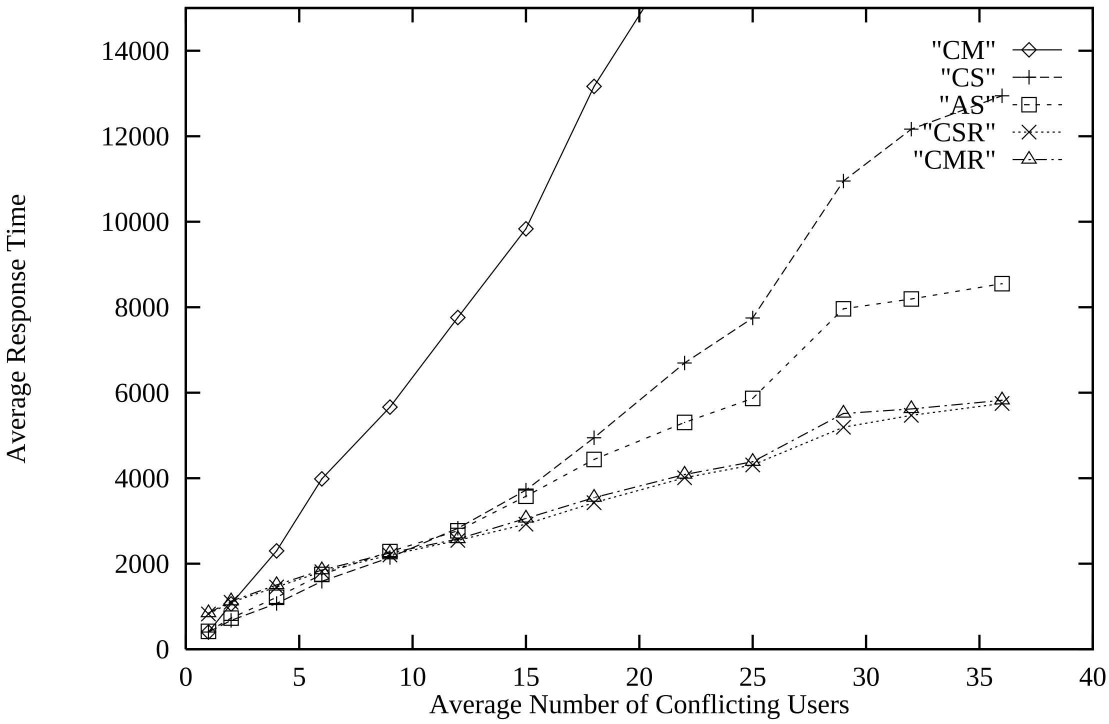

Fig. 21. Average response time of various dining philosophers algorithm when average resource service time = 100 and average message delay = 50.

CHAPTER 12

Conclusion and Future Study

In this part, we presented the tight bound of $\chi(c+d)$ on the response time of the dining and drinking philosophers problem. This bound suggests that the optimal response time dining and drinking philosophers problems are NP-hard. We also gave an efficient modular resource allocation algorithm that uses another resource allocation algorithm of any kind as a subroutine. When our algorithm uses Choy and Singh's dining philosophers algorithm [CS92], the combined algorithm gives worst case response time $O(\delta^2(c+d))$ and $O(r^2\delta)$, which is the fastest known dining philosophers algorithm with the stated message complexity. When Choy and Singh's dynamic resource allocation algorithm [CS93] is used, the combined dynamic resource allocation algorithm gives worst case response time $O(\delta c + (\delta^3 + \log^* |\mathcal{U}|)d)$ and message complexity $O(r^2\delta)$, which is the fastest known dining philosophers algorithm with the stated message complexity. We also simulated various dining philosophers algorithm using a discrete event simulation technique. Our simulation result indicates that our algorithm performs better than other algorithms in average especially when the average number of conflicting users per user is big and the ratio of the average time period that a user is in the critical region over the average message delay is high.

For future study, we leave as an open problem finding a dining or drinking philosophers algorithm with the optimal response time that is also message-efficient and contention-sensitive. We also leave the simulation of dynamic resource allocation algorithms and drinking philosopher algorithm for future study.

For a more broad future direction, we are interested in developing application areas for the resource allocation algorithms developed in this part of the dissertation. Although there has been much previous theoretical research in developing resource allocation algorithms, little work has been done in applying the developed techniques to real applications. Furthermore, new technologies (e.g., large-scale parallel and distributed systems, FDDI and ATM networks) and the new applications they support have engendered a new collection of problems. The application areas include distributed database systems, distributed file systems, parallel transaction systems, real-time transaction systems and hypermedia and collaborative systems. Each of these applications provides a different system environment for resource allocation and new correctness conditions and efficiency measures such as fault-tolerance, scalability, real-time execution (e.g., meeting a deadline), consistency conditions, deadlock and starvation freedom, etc. Therefore, the proposed algorithms have to be adapted to different applications. The performance evaluation of each algorithm through computer simulation should also be adapted to meet the different requirements of each application.

Specifically, we plan to investigate various theoretical models for user-interactive systems (e.g., hypermedia and collaborative systems), develop efficient deadlock handling algorithms for this application and conduct performance studies to evaluate various approaches.

In addition, we have interests in synchronization and resource allocation in parallel and distributed real-time systems. In real-time systems, meeting time deadlines of processes is of more importance than shortening response time of processes. It is only recently that researchers have studied real-time processor scheduling in conjunction with resource allocation [S92]. In fact, without providing an efficient way to synchronize accesses to resources, scheduling only processors among processes is not an effective solution for real-time applications because the resource accesses may cause processes to block, and therefore to miss deadlines. I would like to develop efficient distributed scheduling or synchronization algorithms for processes with real-time deadlines and resource requirements. Performance studies will be done to evaluate various approaches.

INDEX

REFERENCES

[Aw85] B. Awerbuch, "Complexity of Network Synchronization," *Journal of the ACM,* vol. 32, no 4, 1985, pp. 804–823.

[AAT94] R. Alur, H. Attiya and G. Taubenfeld, "Time-Adaptive Algorithms for Synchronization," to appear in *Proc. of ACM Symposium on Theory of Computing*, 1994.

[AD93] H. Attiya and T. Djerassi-Shintel, "Time Bounds for Decision Problems in the Presence of Timing Uncertainty and Failures," *Proc. of 7th International Workshop on Distributed Algorithms*, September 1993, pp. 204–218.

[ADLS91] H. Attiya, C. Dwork, N. Lynch and L. Stockmeyer, "Bounds on the Time to Reach Agreement in the Presence of Timing Uncertainty," *Proc. 23rd ACM Symp. on Theory of Computing,* 1991, pp. 359–369.

[AFL83] E. Arjomandi, M. Fischer and N. A. Lynch, "Efficiency of Synchronous versus Asynchronous Distributed Systems," *Journal of the ACM*, vol. 30, no. 3, 1983, pp. 449–456.

[AL89] H. Attiya and N. A. Lynch, "Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty," *Proc. 10th Real-Time Systems Symp.,* Santa Monica, CA, Dec. 1989, pp. 268–284.

[AM90] H. Attiya and M. Mavronicolas, "Efficiency of Semi-Synchronous versus

Asynchronous Networks," *Proc. 28th Allerton Conf. on Communications, Control and Computing*, Oct. 1990, pp. 578–587.

[AS90] B. Awerbuch and M. Saks, "A dining philosophers algorithm with polynomial response time," *Proc. 31st IEEE Symposium on Foundations of Computer Science,* St. Louis, MO, Oct. 1990, pp. 65–74.

[AT92] Rajeev Alur and Gadi Taubenfeld, "Results about Fast Mutual Exclusion," *Proceedings of the 13th Real-Time Systems Symposium*, Phoenix, AZ, pp. 12–21, 1992.

[Ba78] G. Baudet, "Asynchronous Interactive Methods for Multi-Processors," *Journal of the ACM*, vol. 32, no. 4, 1978, pp. 226–244.

[BP92] J. Bar-Ilan and D. Peleg, "Distributed resource allocation algorithms," *Proc. 6th International Workshop on Distributed Algorithms,* pp. 277–291.

[BG87] V. Barbosa and E. Gafni, "Concurrency in systems in heavily loaded neighborhood-constrained systems," *Proc. Conf. on Distributed Computing Systems,* 1987, pp. 448–455.

[BC84] J. Banks and J. Carson, "Discrete-Event System Simulation," Prentice Hall International Series in Industrial and Systems Engineering, 1984.

[BL80] J. Burns and N. Lynch, "Mutual Exclusion Using Indivisible Reads and Writes," *Proceedings of 18th Annual Allerton Conference on Communication, Control, and Computing*, 1980, pp. 833-842.

[CASD86] F. Cristian, H. Aghili, R. Strong and D. Dolev, "Atomic Broadcast: from simple message diffusion to Byzantine agreement," *Proceedings of 15th International Conference on Fault Tolerant Computing*, 1985, pp. 1–7.

[CDS90] F. Cristian, D. Dolev, H. Strong, "New Latency Bounds for Atomic Broadcast," *Proc. of 11th IEEE Real-Time Symposium*, Lake Buena Vista, FL, Dec. 1990, pp. 156–165.

[CGL93] Soma Chaudhuri, R. Gawlick and N. Lynch, "Designing Algorithms for Distributed Systems with Partially Synchronized Clocks," *Proc. of the 12th ACM Symposium on Principles of Distributed Computing*, August, 1993, pp. 121–132.

[CM84] K. Chandy and J. Misra, "The drinking philosophers problem," *ACM Transactions on Programming Languages and Systems,* vol. 6, no. 4, pp. 632–646, 1984

[CS92] M. Choy and A. Singh, "Efficient Fault–Tolerant Algorithms for Resource Allocation in Distributed Systems," *Proc. 24th Symposium on Theory of Computing,* May 1992, pp. 593–602.

[CS93] M. Choy and A. Singh, "Distributed Job Scheduling Using Snapshots," *Proc. of the 7th International Workshop on Distributed Algorithms,* Sept. 1993, pp. 145–159.

[CT90] B. A. Coan and G. Thomas, "Agreeing on a Leader in Real-Time," *Pro-*

ceedings of the 11th Real-Time Systems Symposium, Lake Buena Vista, FL, Dec. 1990, pp. 1–7.

[CW90] B. A. Coan and J. L. Welch, "Transaction Commit in a Realistic Timing Model," *Distributed Computing,* vol. 4, 1990, pp. 87–103.

[DDS87] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the ACM*, vol. 34, no. 1, Jan. 1987, pp. 77–97.

[Di65] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, vol. 8, September 1965, pp. 569.

[Di71] E. W. Dijkstra, "Hierarchical Ordering of Sequential Process," *Acta Informatica*, vol. 1, no. 2, 1971, pp. 115–138.

[DLS88] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *Journal of the ACM,* vol. 35, no. 2, 1988, pp. 288-323.

[FLP85] Michael Fischer, Nancy Lynch and Michael Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382.

[GSA89] D. Ginat, A. Shankar and A. Agrawala, "An efficient solution to the drinking philosophers problem and its extensions," *Proc. 3rd International Workshop on Distributed Algorithms,* LNCS 392, Springer–Verlag, September 1989, pp. 83–93.

[GVW89] J. Goodman, M. Vernon, and P. Woest, "Efficient Synchronization Primitives for Large-Scale Cache Coherent Multiprocessors," *Proc. of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems* 1989, pp. 64–75.

[Je91] K. Jeffay, "The Real-Time Producer/Consumer Paradigm: A Paradigm for the Efficient, Predictable Real-Time System," *Proc. 1993 ACM/SIGAPP Symposium on Applied Computing,* Indianapolis, IN, ACM Press, February 1993, pp. 796–804.

[JSM91] K. Jeffay, D. F. Stanat and C. U. Martel, "On Optimal, no.-Preemptive Scheduling of Periodic and Sporadic Tasks," *Proceedings of 12th IEEE Real-Time Systems Symposium*, San Antonio, TX, Dec. 1991, pp. 129–139.

[LA86] Leslie Lamport, "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Transactions on Programming Language and Systems*, vol. 6, no. 2, April 1984, pp. 254-280.

[LA87] Leslie Lamport, "A Fast Mutual Exclusion Algorithm," *ACM Transactions on Programming Language and Systems*, vol. 5, no. 1, 1987, pp. 1–11.

[Ly81] Nancy Lynch, "Upper Bounds for Static Resource Allocation in a Distributed System," *Journal of Computer and System Science*, vol. 23, 1981, pp. 254–278.

[LL73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, Jan. 1973, pp. 46–61.

[LA92] M. Loui and H. Abu-Amara, "Memory Requirements for Agreement among Unreliable Asynchronous Processes," *Advances in Computing Research*, 1987, vol 4, pp. 163–183.

[LS92] Nancy Lynch and Nir Shavit, "Timing-Based Mutual Exclusion," *Proceedings of the 13th Real-Time Systems Symposium*, San Antonio, TX, 1991, pp. 2–11.

[LSP82] L. Lamport, R. Shostak and M. Pease, "The Byzantine General Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 382-401.

[Ma92] M. Mavronicolas, "Efficiency of Semi-Synchronous versus Asynchronous Systems: Atomic Shared Memory," TR-03-92, Aiken Computation Lab., Harvard University, Jan. 1992.

[MS88] S. Murphy and A. Shankar, "A note on the drinking philosophers problem," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 2, 1988, pp. 178–188

[NT87] G. Neiger and S. Toueg, "Simulating Synchronized Clocks and Common Knowledge in Distributed Systems," *Journal of the ACM*, vol. 40, no. 3, April 1993, pp. 334-367.

[Po91] S. Ponzio, "Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures," *Proc. ACM Symp. on Principles of Distributed Computing,* Oct. 1991, pp. 125–137.

[PF77] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System," *Proc. 9th ACM Symp. on Theory of Computing,* 1977, pp. 91–97.

[PSL80] M. Pease, R. Shostak and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, vol. 27, no. 2, April 1980, pp. 228-234.

[RW92] I. Rhee and J. Welch, "The Impact of Time on the Session Problem," *Proc. of 11th ACM Sympo. on Principles of Distributed Computing,* Vancouver, Canada, August 1992, pp. 191–201.

[PJC93] I. Page, R. Jacob and S. Chern, "Fast algorithms for distributed resource allocation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, Feb, 1993, pp. 632–646.

[S92] J. Stankovic, "Editorial: Resource Allocation in Real-Time Systems," *Real Time Systems*, vol 5, pp. 1–6, 1992.

[SP88] E. Styer and G. Peterson, "Improved algorithms for distributed resource allocation," *Proc. 7th ACM Symposium on Principles of Distributed Computing*, Toronto, Canada, August 1988, pp. 105–116

[SWL88] B. Simons, J. Welch and N. Lynch, "An Overview of Clock Synchronization," IBM Computer Science Research Report RJ 6506 (63306), Oct. 1988.

[We87] J. Welch, "Simulating Synchronous Processors," *Information and Computation,* vol. 74, no. 2, August 1987, pp. 159-171.

[WL93] J. Welch and N. Lynch, "A modular drinking philosophers algorithm," Distributed Computing, vol. 6, 1993, pp. 233-244.

[WPP91] E. Weidman, I. Page, W. Pervin, "Explicit Dynamic Exclusion Algorithm," *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, December 1991, pp. 142–149.