Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty*

(SHORT VERSION)

Hagit Attiya and Nancy A. Lynch
Laboratory for Computer Science
MIT
Cambridge, MA 02139

Abstract

A timing-based variant of the *mutual exclusion* problem is considered. In this variant, only an upper-bound, m, on the time it takes to release the resource is known, and no explicit signal is sent when the resource is released; furthermore, the only mechanism to measure real time is an inaccurate clock, whose tick intervals take time between two constants, $c_1 \leq c_2$. When control is centralized it is proved that

$$n \cdot c_2 (\lfloor (m + l)/c_1 \rfloor + 1) + l$$

is an exact bound on the worst case response time for any such algorithm, where n is the number of contenders for the resource and l is an upper bound on process step time. On the other hand, when control is distributed among processes connected via communication lines with an upper bound, d, for message delivery time, it is proved that

$$n [c_2 (\lfloor (m + l)/c_1 \rfloor + 1) + d + c_2 + 2l]$$

is an upper bound. A new technique involving *shifting* and *shrinking* executions is combined with a careful analysis of the best allocation policy to prove a corresponding lower bound of

$$n \cdot c_2(m/c_1) + (n - 1)d.$$

These combinatorial results shed some light on modeling and verification issues related to real-time systems.

1 Introduction

An important area of computer applications is real-time process control, in which a computer system interacts with a real-world system in order to guarantee certain desirable real-world behavior. In most interesting cases, the real-world requirements involve timing properties, and so the behavior of the computer system is required to satisfy certain timing constraints. In order to be able to guarantee timing constraints, the computer system must satisfy some assumptions about time - for example, its various components should operate at known speeds.

It is clear that good theoretical work in the area of real-time systems is necessary. In the past few years, several researchers have proposed new frameworks for specifying requirements of such systems, describing implementations, and proving that the implementations satisfy the requirements. These frameworks are based on, among others, finite state machines ([D85]), weakest precondition methods ([H81]), first-order logic ([JM86, JM87]), temporal logic ([BH81]), Petri nets ([CR83, LS87, S77]), and process algebra ([HGR87, KSRGA88, ZLG89]). Work is still needed in evaluating and comparing the various models for their usefulness in reasoning about important problems in this area and perhaps in developing new models if these prove to be inadequate.

Work is also needed in developing the complexity theory of such systems; very little work has so far been done in this area. An example of the kind of work needed is provided by the theory of asynchronous concurrent systems. That theory contains many combinatorial results that show what can and cannot be accomplished by asynchronous systems; for tasks that can be accomplished, other combinatorial results determine the inherent costs. In addition to their individual importance, these results also provide a testbed for evaluating modeling decisions and a stimulus for the development of algorithm verification techniques. Similar results should be possible for real-time systems. Some examples of complexity results that have already been obtained for real-time systems are the many results on clock synchronization, including [DHS86, HMM85, L78, LL84, WL88] (see [SWL88] for a survey).

In this paper, we embark on a study of com-

*This work was supported by ONR contract N0014-85-K-0168, by NSF contract CCR-8611442, and by DARPA contract N00014-83-K-0125.

plexity results for real-time systems. We begin this study by considering timing-based variations of certain problems that have previously been studied in asynchronous concurrent systems. In particular, in this paper, we study a variant of the *mutual exclusion problem*. This problem is one of the fundamental problems in distributed computing; it serves as an abstraction of a large class of *hazard avoidance* problems. We note that this particular problem appears in the real-time computing literature (cf. [JM87]) as the "nuclear reactor problem". There, operators push different buttons to request the motion of different control rods in the same nuclear reactor. It is undesirable to have more than one control rod moving at the same time, presumably since in that case the nuclear reaction might be slowed down too much.

More specifically, we consider a system consisting of some number, n, of identical moving parts (e.g., control rods), no two of which are supposed to move at the same time. An operator associated with each moving part can request permission for the associated part to move by pushing a button that sends a *REQUEST* signal to the computer system. The system responds with *GRANT* signals; each *GRANT* signal gives permission to the designated moving part to move, but such motion is expected to be finished no more than a fixed time, m, later. The system is only supposed to issue a *GRANT* signal when it knows that it is safe to move the corresponding moving part, i.e., at least real time m has elapsed since the last *GRANT* signal. We assume, for simplicity, that a *REQUEST* signal is only issued by a particular operator if any preceding *REQUEST* by that operator has already been satisfied (by a corresponding *GRANT* signal). Our goal is to minimize the worst-case time between a *REQUEST* signal and the corresponding *GRANT* signal, i.e., the *worst-case response time*.

The computer system might consist of a single process running on a dedicated processor or might be a distributed system running on separate processors communicating over a message system. Solving the problem efficiently requires the computer system to make accurate estimates of the elapsed time since the last *GRANT* signal; the difficulty, however, is that the computer system only has inaccurate information about time, as given by inaccurate clock components within the system and by estimates of the time required for certain events. Specifically, the only information about time that the computer system has is the following:

1. the knowledge that a moving part will stop moving within time m after a *GRANT* signal,
2. the knowledge that the time between successive ticks of any clock is always in the interval $[c_1, c_2]$, for known constants c_1 and c_2, where $0 < c_1 \leq c_2$,
3. the knowledge that the time between successive steps of any process within the computer system is always in the interval $[0, l]$, for a known constant $l, 0 \leq l$, and
4. (if the system is distributed) the knowledge that the time to deliver the oldest message in each channel is no greater than a known constant $d, 0 \leq d$.

In the cases we have in mind, we suppose that $l << c_1 < c_2 << d << m$, but we state explicitly any assumptions that we require about relative sizes of the various constants.

One way in which our problem differs from the mutual exclusion problem usually studied in asynchronous systems is that we do not assume that an explicit signal is conveyed to the computer system when a moving part stops moving; the only information the system has about the completion of the critical activity is based on its estimates of the elapsed time. It is fairly typical for real-time systems to use time estimates in order to make deductions about real-world behavior. The results of this paper indicate some of the costs that result from using such estimates.

We obtain the following results. First, we consider a centralized computer system, consisting of just a single process with a local clock. For that case, we show that

$$n \cdot c_2 (\lfloor (m + l)/c_1 \rfloor + 1) + l$$

is an *exact* bound on the worst-case response time for the timing-based mutual exclusion problem. The upper bound result arises from a careful analysis of a simple FIFO queue algorithm, while the matching lower bound result arises from explicitly constructing and "retiming" executions to obtain a contradiction.

We then consider the distributed case, which is substantially more complicated. For that case, we obtain very close (but not exact) bounds: an upper bound of

$$n [c_2 (\lfloor (m + l)/c_1 \rfloor + 1) + d + c_2 + 2l]$$

and a lower bound of

$$n \cdot c_2(m/c_1) + (n - 1)d .$$

Assuming that the parameters have the relative sizes described earlier, e.g., that d is much larger than l, c_1 and c_2, the gap between these two bounds is just slightly more than a single message delay time. The

upper bound arises from a simple token-passing algorithm, while the lower bound proof employs a new technique of shifting some of the events happening at a process while carefully retiming other events.

The model that we use for proving our results is the I/O automaton model [LT87], which has been extended recently to include timing [MMT88]. As noted earlier, many people are working on the development of other models and frameworks for reasoning about real-time systems. The most popular way of evaluating such frameworks involves their application to the specification and verification of substantial examples of practical utility. This paper, however, suggests a complementary approach. Since a framework for real-time processing should allow proof of combinatorial upper and lower bound and impossibility results, in addition to allowing specification and verification of systems, careful proofs of combinatorial results such as those in this paper should teach us a good deal about the appropriateness of a model for real-time processing.

The rest of this paper is organized as follows. Section 2 presents the timed I/O automaton model. Section 3 contains the general statement of the problem to be solved. Section 4 contains our results for the centralized case, Section 5 contains our results for the distributed case, and Section 6 contains some discussion and open problems. Many of the proofs are omitted in this version and can be found in the full version ([AL89]).

2 Model and Definitions

2.1 I/O Automata

An *I/O automaton* consists of the following components: a set of *actions*, classified as *output*, *input* and *internal*, a set of *states*, including a distinguished subset called the *start states*, a set of (*state, action, state*) triples called *steps*, and a *partition* of the *locally controlled* (output and internal) actions into equivalence classes. An action π is said to be *enabled* in a state s' provided that there is a step of the form (s', π, s). An automaton is required to be *input enabled*, which means that every input action must be enabled in every state. The partition groups actions together that are to be thought of as under the control of the same underlying process.

Concurrent systems are modeled by compositions of I/O automata, as defined in [LT87]. In order to be composed, automata must be *strongly compatible*; this means that no action can be an output of more than one component, that internal actions of one component are not shared by any other component, and that no action is shared by infinitely many components. The result of such a composition is another I/O automaton. The *hiding* operator can be applied to reclassify output actions as *internal* actions.

We refer the reader to [LT87] for a complete presentation of the model and its properties.

2.2 Timed Automata

We augment the I/O automaton model as in [MMT88] to allow discussion of timing properties. Namely, a *timed I/O automaton* is an I/O automaton with an additional component called a *boundmap*. The boundmap associates a closed subinterval of $[0, \infty]$ with each class in the automaton's partition; to avoid certain boundary cases we assume that the lower bound of each interval is not ∞ and the upper bound is nonzero. This interval represents the range of possible differences between successive times at which the given class gets a chance to perform an action. We sometimes use the notation $b_\ell(C)$ to denote the lower bound assigned by boundmap b to class C, and $b_u(C)$ for the corresponding upper bound.

A *timed sequence* is a sequence of alternating states and (action,time) pairs:

$$s_0, (\pi_1, t_1), s_1, (\pi_2, t_2) \ldots$$

Define $t_0 = 0$. The times are required to be nondecreasing, i.e., for any $i \geq 1$ for which t_i is defined, $t_i \geq t_{i-1}$, and if the sequence is infinite then the times are also required to be unbounded. For any finite timed sequence α define $t_{end}(\alpha)$ to be the time of the last event in α, if α is nonempty, or 0, if α is empty; for an infinite timed sequence α, $t_{end}(\alpha) = \infty$.

A timed sequence is said to be a *timed execution* of a timed automaton A with boundmap b provided that when the time components are removed, the resulting sequence is an execution of the I/O automaton underlying A, and it satisfies the following conditions for each class C of the partition of A and every i:

1. Suppose $b_u(C) < \infty$. If some action in C is enabled in s_i and one of the following holds: either $i = 0$ or no action in C is enabled in s_{i-1} or π_i is in C, then there exists $j > i$ with $t_j \leq t_i + b_u(C)$ such that either π_j is in C or no action of C is enabled in s_j.

2. If some action in C is enabled in s_i and either $i = 0$ or no action in C is enabled in s_{i-1} or π_i is in C, then there does not exist $j > i$ with $t_j < t_i + b_\ell(C)$ and π_j in C

The first condition says that, starting from when an action in C occurs or first becomes enabled, within

time $b_u(C)$ either some action in C occurs or there is a point at which no such action is enabled. The second condition says that, again starting from when an action in C occurs or first becomes enabled, no action in C can occur before time $b_\ell(C)$ has elapsed. The third condition merely requires that the steps taken by the automaton are indeed legal.

Note that the definition of a timed execution includes a liveness condition (in 1.) in addition to safety conditions (in both 1. and 2.). For finite timed sequences, it is sometimes interesting to consider only the safety properties. Thus, we define a weaker notion, as follows. A finite timed sequence is said to be a *timed semi-execution* provided that when the time components are removed, the resulting sequence is an execution of the I/O automaton underlying A, and it satisfies the following conditions, for every class C and i.

1. Suppose $b_u(C) < \infty$. If some action in C is enabled in s_i and one of the following holds: either $i = 0$ or no action in C is enabled in s_{i-1} or π_i is in C, then either $t_{end}(\alpha) \leq t_i + b_u(C)$ or there exists $j > i$ with $t_j \leq t_i + b_u(C)$ such that either π_j is in C or no action of C is enabled in s_j.
2. Condition 2. above.

Intuitively, timed semi-executions represent sequences in which the safety conditions described by the boundmap are not violated. The following lemmas say that such a sequence can be extended to a timed execution in which the liveness conditions described by the boundmap are also satisfied.

Lemma 2.1 *If α is a timed semi-execution of a timed automaton A and no locally controlled action of A is enabled in the final state of α, then α is a timed execution of A.*

Lemma 2.2 *Let $\{\alpha_i\}_{i=1}^{\infty}$ be a sequence of timed semi-executions of a timed automaton A such that*

1. *for any $i \geq 1$, α_i is a prefix of α_{i+1}, and*
2. *$\lim_{i\to\infty} t_{end}(\alpha_i) = \infty$.*

Then there exists an infinite timed execution α of A such that for any $i \geq 1$, α_i is a prefix of α.

Lemma 2.3 *Let A be a timed automaton having finitely many classes in its partition, and let α be a timed semi-execution of A. Then there is a timed execution α' of A that extends α, such that only events from classes with finite upper bound occur in α' after α.*

For any timed execution or semi-execution α we define $sched(\alpha)$ to be the sequence of (action,time) pairs occurring in α, i.e., α with the states removed. We say that a sequence of (action,time) pairs is a *timed schedule* of A if it is $sched(\alpha)$, where α is a timed execution of A. We also define $beh(\alpha)$ to be the subsequence of $sched(\alpha)$ consisting of external (input and output) actions and associated times, and say that a sequence of (action,time) pairs is a *timed behavior* of A if it is $beh(\alpha)$, where α is a timed execution of A.

Definitions for composing timed automata to yield another timed automaton, analogous to those for I/O automata, are developed in [MMT88]. We model real-time systems as compositions of timed automata. (Real-time systems were also modeled in this way in [L88].)

2.3 Adding Time Information to the States

We would like to use standard proof techniques such as invariant assertions to reason about timed automata. In order to do this, we find it convenient to define an ordinary I/O automaton $time(A)$ corresponding to a given timed automaton A. This new automaton has the timing restrictions of A built into its state, in the form of predictions about when the next event in each class will occur. Thus, given any timed I/O automaton A having boundmap b, the *ordinary* I/O automaton $time(A)$ is defined as follows.

The automaton $time(A)$ has actions of the form (π,t), where π is an action of A and t is a nonnegative real number. Each of its states consists of a state of A, augmented with a time called $Ctime$ and, for each class C of the partition, two times, $Ftime(C)$ and $Ltime(C)$. $Ctime$ (the "current time") represents the time of the last preceding event, initially 0. The $Ftime(C)$ and $Ltime(C)$ components represent, respectively, the first and last times at which an action in class C is scheduled to be performed (assuming some action in C stays enabled). (We use record notation to denote the various components of the state of $time(A)$; for instance, $s.Astate$ denotes the state of A included in state s of $time(A)$.) More precisely, each initial state of $time(A)$ consists of an initial state s of A, plus $Ctime = 0$, plus values of $Ftime(C)$ and $Ltime(C)$ with the following properties. If there is an action in C enabled in s, then $Ftime(C) = b_\ell(C)$ and $Ltime(C) = b_u(C)$. Otherwise, $Ftime(C) = 0$ and $Ltime(C) = \infty$.

If (π,t) is an action of $time(A)$, then $(s',(\pi,t),s)$ is a step of $time(A)$ exactly if the following conditions hold.

1. $(s'.Astate, \pi, s.Astate)$ is a step of A.

2. $s'.Ctime \leq t = s.Ctime$.

3. If π is a locally controlled action of A in class C, then

   (a) $s'.Ftime(C) \leq t \leq s'.Ltime(C)$,

   (b) if some action in C is enabled in $s.Astate$, then $s.Ftime(C) = t + b_\ell(C)$ and $s.Ltime(C) = t + b_u(C)$, and

   (c) if no action in C is enabled in $s.Astate$, then $s.Ftime(C) = 0$ and $s.Ltime(C) = \infty$.

4. For all classes D such that π is not in class D,

   (a) $t \leq s'.Ltime(D)$,

   (b) if some action in D is enabled in $s.Astate$ and some action in D is enabled in $s'.Astate$ then $s.Ftime(D) = s'.Ftime(D)$ and $s.Ltime(D) = s'.Ltime(D)$.

   (c) if some action in D is enabled in $s.Astate$ and no action in D is enabled in $s'.Astate$ then $s.Ftime(D) = t + b_\ell(D)$ and $s.Ltime(D) = t + b_u(D)$, and

   (d) if no action in D is enabled in $s.Astate$, then $s.Ftime(D) = 0$ and $s.Ltime(D) = \infty$.

Note that property 4(a) ensures that an action does not occur if any other class has an action that must be scheduled first. The partition classes of $time(A)$ are derived one-for-one from the classes of A (although we will not need them in this paper).

The finite executions of $time(A)$, when the states are projected onto their $Astate$ components, are exactly the same as the *finite* prefixes of the timed executions of A. This implies that safety properties of a timed automaton A can be proved by proving them for $time(A)$, e.g., using invariant assertions.

3 Problem Statement

For either the centralized or distributed case, we assume that there are n modules called *moving parts*, n modules called *operators*, plus some modules comprising the *computer system*. The actions of the complete system, exclusive of any internal actions of the computer system, are $REQUEST(i)$, $GRANT(i)$ and $FINISH(i)$, for $0 \leq i \leq n-1$. Each $operator(i)$ has input action $GRANT(i)$ and output action $REQUEST(i)$. Each $movingpart(i)$ has input action $GRANT(i)$ and output action $FINISH(i)$. The *computer system* has input actions $REQUEST(i)$ for all i and output actions $GRANT(i)$ for all i. See Figure 1.

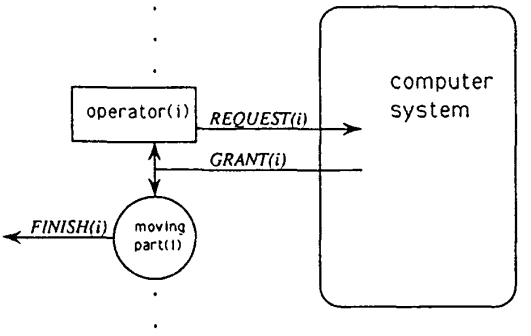

Figure 1: The system architecture.

Let $movingpart(i)$ be a particular timed automaton with the given signature, having a state consisting of one component, GRANTED, a Boolean variable, initially *false*. The input action $GRANT$ sets the variable GRANTED to be *true*; the output action $FINISH$ is enabled only when GRANTED = *true*, and it sets GRANTED to be *false*. There is only one class in the partition for $movingpart(i)$, a singleton containing the one action $FINISH(i)$. The boundmap associates the interval $[0, m]$ with this class. As described in the Introduction, the timed executions of this timed automaton have the property that, within time m after a $GRANT(i)$ occurs, a $FINISH(i)$ must also occur - that is, $movingpart(i)$ "stops moving".

Now consider $operator(i)$. It is described as an automaton with the maximum amount of freedom we want to allow to the operator. Let $operator(i)$ be the timed automaton with the appropriate signature, having a state consisting of one component, PUSHED, a Boolean variable, initially *false*. The output action $REQUEST$ is enabled only when PUSHED = *false*, and it sets PUSHED to be *true*; The input action $GRANT$ sets the variable PUSHED to be *false*. Again, there is only one (singleton) class in the partition for $operator(i)$. We do not want to insist that the operator push the button within a particular amount of time after a $GRANT$. (It may never do so, in fact.) Thus, we define the boundmap to assign the interval $[0, \infty]$ to this one class.

The requirement for the computer system is that when it is composed with the given operators and moving parts, the resulting system has all its behaviors satisfying the following conditions:

1. *Request well-formedness:* For any $0 \leq i \leq n-1$, $REQUEST(i)$ and $GRANT(i)$ actions alternate, starting with a $REQUEST(i)$.

2. *Moving part well-formedness:* For any $0 \leq i \leq$

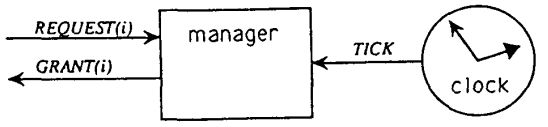

Figure 2: The architecture of the centralized control system.

$n-1$, $GRANT(i)$ and $FINISH(i)$ actions alternate, starting with $GRANT(i)$.

3. *Mutual exclusion:* There are never two consecutive $GRANT$ events without an intervening $FINISH$ event.

4. *Eventual granting:* Any $REQUEST(i)$ event has a following $GRANT(i)$ event.

We measure the *performance* of the system by the *worst case response time*, i.e., the longest time between $REQUEST(i)$ and the next subsequent $GRANT(i)$ in any timed behavior.

4 A Centralized System

We first consider the case of a "centralized" computer system to solve this exclusion problem. In this case, the architecture is as follows. There are two modules (timed I/O automata), the *manager* and the *clock*. The *clock* has only one action, the output $TICK$, which is always enabled, and has no effect on the clock's state. The boundmap associates the interval $[c_1, c_2]$ with the single class of the partition. This means that successive $TICK$ events will occur with intervening times in the given interval.

The *manager* has input actions $TICK$ and $REQUEST(i)$ for all i, and output actions $GRANT(i)$. It is an arbitrary automaton, subject to the restriction that it has only a single class in its partition. (This says that it is really a sequential process – it cannot be running several processes in parallel.) We associate the boundmap $[0, l]$ with the single class of locally controlled actions. This means that successive locally-controlled steps of the manager are done within the given intervals (if there are any enabled).

The computer system is the composition of the manager and the clock, (with the I/O automaton hiding operator applied to hide the $TICK$ actions). See Figure 2.

Note that the timed automaton model forces us to model the step time of the manager process explicitly. Other models (e.g., the one used for clock synchronization in [LL84]) might avoid this level of detail by hypothesizing that the manager's steps are triggered only by input events such as clock ticks or requests. We regard such a model (informally) as a limiting case of our model, as the upper bound on manager step time approaches zero.

4.1 Upper Bound

4.1.1 The Algorithm

The following simple algorithm for the manager process solves the problem. The manager simply puts requests on a FIFO queue. If there is a pending request, the manager issues a $GRANT$ signal to the node whose request is first on the queue, and sets a timer to measure the time until the moving part stops moving. When the timer goes off, the manager repeats.

There is some subtlety in determining the minimum number of clock ticks that guarantee that time m has elapsed since the $GRANT$. At first glance, one might be tempted to count $\lfloor m/c_1 \rfloor + 1$ ticks, but a careful examination shows that this might cause a violation of the exclusion property, if a $TICK$ happens immediately after the $GRANT$, and the next $GRANT$ happens immediately after the last $TICK$. Waiting for $\lfloor m/c_1 \rfloor + 2$ suffices to overcome this difficulty, but the lower bound presented in Subsection 4.2 suggests that this might not be optimal. In order to achieve the best possible timing performance, the algorithm only grants immediately after a clock tick, and the timer is set to $\lfloor (m+l)/c_1 \rfloor + 1$ clock ticks.

In addition to the $REQUEST$ and $TICK$ inputs and $GRANT$ outputs already specified, the manager has an internal action $ELSE$. This action is enabled exactly when no output action is enabled; this has the effect of ensuring that locally controlled steps of the manager occur at (approximately) regular intervals, as determined by the manager's boundmap.

The manager's state is divided into components:

TICKED holding a boolean value, initially *true*;
QUEUE holding a queue of indices $i \in [0..n-1]$, initially empty;
TIMER holding an integer, initially 0;

The manager's algorithm is as follows:
$REQUEST(i)$, $0 \leq i \leq n-1$
Effect:
    add i to QUEUE

$TICK$
Effect:
    TIMER := TIMER -1

TICKED := *true*

GRANT(i), $0 \leq i \leq n-1$
Precondition:
i is first on QUEUE
TIMER ≤ 0
TICKED = *true*
Effect:
remove i from front of QUEUE
TIMER := $\lfloor (m+l)/c_1 \rfloor + 1$
TICKED := *false*

ELSE
Precondition:
QUEUE is empty
or TIMER > 0 or TICKED = *false*
Effect:
TICKED := *false*

4.1.2 Correctness Proof

Let A be the composition of the four given kinds of timed automata - operators, moving parts, manager and clock. This subsection is devoted to proving the following theorem.

Theorem 4.1 *Algorithm A is a correct centralized resource allocation algorithm.*

We prove correctness using automaton *time*(A), as defined above. In this case, the system state is augmented with the variable *Ctime*, plus the variables *Ftime* and *Ltime*, for the following partition classes:

1. *REQUEST*(i) for each i, which contains the single action *REQUEST*(i),
2. *FINISH*(i) for each i, which contains the single action *FINISH*(i),
3. *TICK*, which contains the single action *TICK*, and
4. *LOCAL*, the locally controlled actions, which contains all the actions *GRANT*(i), $0 \leq i \leq n-1$ and the *ELSE* action.

Initially, we have *Ftime*(*REQUEST*(i)) = 0, *Ltime*(*REQUEST*(i)) = ∞, *Ftime*(*FINISH*(i)) = 0 and *Ltime*(*FINISH*(i)) = ∞, *Ftime*(*TICK*) = c_1, *Ltime*(*TICK*) = c_2, *Ftime*(*LOCAL*) = 0 and *Ltime*(*LOCAL*) = l.

The proof of mutual exclusion rests on the following invariant for *time*(A).

Lemma 4.2 *Let s be a reachable state of time(A). Then the following all hold:*

1. *If FINISH(i) is enabled in s.Astate, then*
   (a) $s.TIMER > 0$,
   (b) $s.Ftime(TICK) + (s.TIMER - 1)c_1 > s.Ltime(FINISH(i))$, *and*
   (c) *FINISH(j) is not enabled in s.Astate, for any $j \neq i$.*
2. *If $s.TICKED$ then $s.Ftime(TICK) \geq s.Ltime(LOCAL) + c_1 - l$.*

Thus, if a part is moving, the manager's TIMER is positive. Moreover, the TIMER is large enough so that waiting that number of ticks would cause enough time to elapse so that the part would be guaranteed to have stopped moving. Property 1(c) implies mutual exclusion, while property 2 guarantees a lower bound on the time till the next *TICK*, if no *LOCAL* step has occurred since the previous *TICK*.

The proof of correctness is done in careful detail and can be found in [AL89].

Proof: (of Theorem 4.1) Lemma 4.2 implies mutual exclusion. Moving part well-formedness follows easily from the same lemma and the definition of the moving part. Request well-formedness follows from the definitions of the operators and the manager. The remaining condition, eventual granting, can be argued from the queue-like behavior of the manager and the fact that the clock keeps ticking. (This latter property also follows from the formal proof of the upper bound on response time in the following subsection.) ■

4.1.3 Response Time

Now we prove our upper bound on response time for the given algorithm A.

Theorem 4.3 *Assume that $l < c_1$. The worst case response time for algorithm A is at most*

$$n\,[c_2\,(\lfloor (m+l)/c_1 \rfloor + 1)] + l.$$

The proof of this theorem requires several lemmas.

Lemma 4.4 *In any reachable state there are at most n entries in QUEUE.*

Lemma 4.5 *In any reachable state s, $s.TIMER \leq \lfloor (m+l)/c_1 \rfloor + 1$.*

Lemma 4.6 *Let s be any state occurring in a timed execution, in which $s.TIMER \leq k$, for $k \geq 1$. Then (at least) one of the following two conditions holds.*

1. *$s.TIMER \leq 0$ and $s.TICKED$ = true, or*

2. *the time from the given occurrence of s until a later TICK event resulting in TIMER ≤ 0 is bounded above by $c_2 \cdot k$.*

Proof: (of Theorem 4.3) When a request arrives, it is at worst in position n on the QUEUE, by Lemma 4.4. By Lemmas 4.5 and 4.6, either TIMER ≤ 0 and TICKED = *true* at the time when the request arrives, or else within time $c_2(\lfloor (m+l)/c_1 \rfloor + 1)$ a *TICK* event (call it π_1) occurs which sets TIMER to 0. In the former case, there must be a *TICK* event occurring prior to the request that sets TIMER ≤ 0, with no intervening local events; let π_1 denote this *TICK* event. In either case, within time l after π_1 (but after the request) the first entry gets its request granted and gets removed from the QUEUE, and TIMER is set to

$$\lfloor (m+l)/c_1 \rfloor + 1.$$

Since $l < c_1$, within time c_2 after π_1, another *TICK* event φ_1 occurs, this one decreasing TIMER to $(\lfloor (m+l)/c_1 \rfloor)$.

Immediately after φ_1, either TIMER $= 0$, or $\lfloor (m+l)/c_1 \rfloor \geq 1$; in this latter case, by Lemma 4.6, within at most time $c_2\,(\lfloor (m+l)/c_1 \rfloor)$ after φ_1, a *TICK* event occurs that sets TIMER ≤ 0. Thus, in either case, from event π_1 until another *TICK* event π_2 that sets TIMER ≤ 0, at most

$$c_2\,(\lfloor (m+l)/c_1 \rfloor + 1)$$

time elapses. The next entry in the queue is enabled immediately after π_2. In this manner, we can construct a sequence of *TICK* events, $\pi_1, \ldots, \pi_n$, such that the time between π_i and π_{i+1}, for each $i, 1 \leq i < n$, is at most

$$c_2\,(\lfloor (m+l)/c_1 \rfloor + 1)\,,$$

and for any $1 \leq i \leq n$, the i'th entry on the original queue (if there is any) is enabled after π_i. Hence, within time

$$n\,[c_2\,(\lfloor (m+l)/c_1 \rfloor + 1)]\,,$$

the enabling condition is satisfied for the given request. Then within time at most l afterwards, the request is granted. This completes the proof of the upper bound on response time. ■

Note that this proof requires the assumption that $l < c_1$; in case this assumption is not made, an analysis similar to the one in the proof above yields a slightly higher upper bound of

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + l]\,.$$

Also, note that the limit of the given upper bound, as l approaches 0, is $n \cdot c_2(\lfloor m/c_1 \rfloor + 1)$. We think of this as an upper bound for this algorithm when it is run on an interrupt-driven model.

It follows from the lower bound in Section 4.2 that algorithm A has optimal response time. This seems to imply that the best policy is to issue a *GRANT* right after a *TICK*. This is apparently because a time estimate done immediately after a clock *TICK* is the most accurate.

Although this proof is currently written in terms of executions, it seems that the invariant assertion techniques for time-augmented automata developed above could be extended to handle response time analysis; preliminary results in that direction appear in [LA].

4.2 Lower Bound

Now we turn to proving lower bounds. We begin with a fairly simple lower bound result that is quite close to the upper bound proved in the preceding subsection, but does not match exactly. The gap between this lower bound and the upper bound depends on the manager's step time and the roundoffs. Since we consider these to be very small, for practical purposes one might be satisfied with this simpler lower bound. However, it is interesting theoretically to note that in this case, we can obtain a tight bound by a related but somewhat more difficult argument.

Theorem 4.7 *The worst case response time of any centralized resource allocation algorithm is at least*

$$n \cdot m(c_2/c_1).$$

In order to see why this is so, define a timed execution or timed semi-execution to be *slow* if the times between successive *TICK* events (and the time of the first *TICK* event) are exactly c_2. We have:

Lemma 4.8 *Let α be a slow timed execution of a correct centralized resource allocation algorithm. Then the time between any two consecutive GRANT events in α is strictly greater than*

$$m(c_2/c_1).$$

Now we present the more delicate arguments needed to prove a lower bound that matches the upper bound given in Section 4.1. Note that the only differences between the lower bound to be proved and the one already proved in Theorem 4.7 are the presence of the l terms describing bounds on the manager's step time and the careful treatment of roundoff. Still, it is interesting that the bound can be improved in these ways to match the upper bound exactly.

Theorem 4.9 *Assume that $l \leq c_1$.*[1] *Then the worst case response time of any centralized resource allocation algorithm is at least*

$$n\left[c_2\left(\lfloor (m+l)/c_1 \rfloor + 1\right)\right] + l.$$

An I/O automaton is called *active* if in every state there is a locally-controlled action enabled. (Recall, for example, that the manager in the algorithm of the preceding subsection was made active by the inclusion of the *ELSE* action.) Before proceeding with the proof of the theorem, it is useful to prove the following lemma, which claims that there is no loss of generality in assuming that the manager is active. As in the previous subsection, denote by *LOCAL* the class of *all* the actions that are locally controlled by the manager (including $GRANT(i)$, for all i).

Lemma 4.10 *Suppose that A is a centralized resource allocation algorithm with response time $\leq b$, for a real number b. Then there is another such algorithm A', with response time $\leq b$, in which the manager is active.*

Now we return to the task of proving Theorem 4.9. The proof will proceed by iterative construction of a particular slow timed execution. A major step in the construction is forcing a *GRANT* event to happen only in certain situations, as specified and proved in the following technical lemma.

If i is an index with $0 \leq i \leq n-1$, we say that i is *unfulfilled* in a timed semi-execution α if the number of $REQUEST_i$ events in α is strictly greater than the number of $GRANT_i$ events in α. We say that a timed execution or timed semi-execution α is *heavily loaded starting from time t* if for all times $t \leq t' < t_{end}(\alpha)$, all indices are unfulfilled in the prefix of α consisting of all the events occurring up to and including time t'. We say that an action is an *ELSE* action if it is a locally controlled action of the manager other than a *GRANT*; *ELSE* events and steps are defined similarly.

Lemma 4.11 *Let A be a centralized resource allocation algorithm with an active manager, and let α be a slow timed semi-execution of A. Assume that there are unfulfilled indices in α, and LOCAL and TICK events occur in α at time $t_{end}(\alpha)$. Then there exists a slow timed semi-execution β extending α, such that for some i, $0 \leq i \leq n-1$,*

$$\begin{aligned} sched(\beta) = {} & sched(\alpha\sigma)\ (GRANT(i), t) \\ & (REQUEST(i), t)\ (FINISH(i), t), \end{aligned}$$

where $t = t_{end}(\alpha\sigma)$, LOCAL and TICK events occur in $\alpha\sigma$ at time t, and there are no REQUEST or GRANT events in σ.

Notice that if α is a heavily loaded starting from time t then β is also heavily loaded starting from time t.

Now we are ready to present the main proof.

Proof: (of Theorem 4.9) Assume that we have a particular centralized resource allocation algorithm. By Lemma 4.10, we may assume without loss of generality that the manager is active. We explicitly construct a (slow) timed execution in which the response time for a particular grant is at least

$$n\left(\lfloor (m+l)/c_1 \rfloor + 1\right) c_2 + l.$$

We first construct an initial section, β_0. We begin by allowing some *LOCAL* events to occur (at arbitrary allowable times), ending with both a *LOCAL* event and a *TICK* event occurring at exactly time c_2, in that order. Notice that by the *grant well-formedness* property these *LOCAL* events must be *ELSE* events. We let

$$REQUEST(0), \ldots, REQUEST(n-1)$$

events happen immediately after these *ELSE* and *TICK* events, also at time c_2. Formally, let β_0 be a timed semi-execution that extends another timed semi-execution δ containing only *ELSE* events, such that

$$\begin{aligned} sched(\beta_0) = {} & sched(\delta)\ (\pi, c_2)\ (TICK, c_2) \\ & (REQUEST(0), c_2)\ \ldots \\ & (REQUEST(n-1), c_2), \end{aligned}$$

where π is an *ELSE* event. Note that $0, \ldots, n-1$ are unfulfilled indices in β_0, and that *LOCAL* and *TICK* events occur in β_0 at time $c_2 = t_{end}(\beta_0)$; furthermore, note that β_0 is heavily loaded starting from time $t_0 = t_{end}(\beta_0) = c_2$.

Starting from β_0, we construct successive proper extensions $\beta_1, \ldots, \beta_k, \ldots$, such that for each $k \geq 1$, β_k is a slow timed semi-execution of the form $\beta_{k-1}\gamma_k$ that ends at time $t_k = t_{end}(\beta_k)$, that is heavily loaded starting from time t_0, and that has the following properties:

1. β_k ends with $GRANT(j_k)$, $REQUEST(j_k)$ and $FINISH(j_k)$ events, occurring in that order at time t_k.
2. There are no other *REQUEST* or *GRANT* events in γ_k.
3. A *LOCAL* event (other than the $GRANT(j_k)$) and a *TICK* event occur in β_k at time t_k.

[1] Notice that a non-strict inequality is used in this assumption, whereas a corresponding assumption for Theorem 4.3 uses a strict inequality. This reflects the difference in the kinds of reasoning needed for lower and upper bound results.

The construction is done inductively; the base case is the construction of β_1. Since β_0 has a *LOCAL* and a *TICK* event at time $t_{end}(\beta)$, and there are unfulfilled indices in β_0, we can apply Lemma 4.11 to get an execution β_1 with the properties above.

For the inductive step, assume we have constructed a slow timed semi-execution β_{k-1}, for $k > 1$, with the above properties; we show how to construct β_k. Since β_{k-1} is heavily loaded starting at time t_0, and *LOCAL* and *TICK* events occur in β_{k-1} at time t_{k-1}, we can apply Lemma 4.11 to β_{k-1}, and get a slow timed semi-execution β_k that extends β_{k-1} such that

$$sched(\beta_k) = sched(\beta_{k-1}\sigma_k)\ (GRANT(j_k), t_k)\ (REQUEST(j_k), t_k)\ (FINISH(j_k), t_k),$$

where $t_k = t_{end}(\beta_{k-1}\sigma_k)$, *LOCAL* and *TICK* events occur in $\beta_{k-1}\sigma_k$ at time t_k, and there are no *REQUEST* or *GRANT* events in σ_k. Let γ_k be such that

$$\beta_k = \beta_{k-1}\gamma_k \ .$$

Clearly, β_k has the required properties.

Claim 4.12 *For any $k > 1$, there are at least*

$$\lfloor (m+l)/c_1 \rfloor + 1$$

ticks in segment γ_k of β_k.

The claim implies that

$$t_{k+1} - t_k \geq c_2(\lfloor (m+l)/c_1 \rfloor + 1) \ ,$$

for any $k \geq 1$, becuase β_{k+1} is slow.

We continue the proof of Theorem 4.9. Since for every $k \geq 1$, β_k is heavily loaded starting from time t_0 and the algorithm satisfies the *eventual granting* property, there exists k' such that for every i, $0 \leq i \leq n-1$ at least one $GRANT(i)$ event appears in $\beta_{k'}$ at or after time t_1. By the same reasoning, there exists $k'' > k'$ such that for every i, $0 \leq i \leq n-1$ at least one $GRANT(i)$ event appears in $\beta_{k''}$ after time $t_{k'}$. It follows that there is some i, $0 \leq i \leq n-1$ for which there are two consecutive $GRANT(i)$ events in $\beta_{k''}$ having at least $n-1$ intervening $GRANT(j)$ events for $j \neq i$. Suppose that the first of these $GRANT(i)$ events occurs at time t_{k_1}, and the second at time t_{k_2}; it must be that $k_2 - k_1 \geq n$. Note that the $REQUEST(i)$ event corresponding to the second of these $GRANT(i)$ events occurs at time t_{k_1}. By the remark after Claim 4.12 the total amount of time from time t_{k_1} in β_{k_2}, when $REQUEST(i)$ occurs, until the corresponding $GRANT(i)$ occurs, at time t_{k_2} is at least

$$n\left[c_2\left(\lfloor (m+l)/c_1 \rfloor + 1\right)\right] \ .$$

We now construct from β_{k_2} a timed semi-execution δ in which the $GRANT(j_{k_2})$ event occurs at time $t_{k_2} + l$, retiming later events as necessary to maintain monotonicity. The timed sequence δ is a timed semi-execution since $l \leq c_2$, and since there is a *LOCAL* event preceding $GRANT(j_{k_2})$ at time t_{k_2} in β_{k_2}. It follows that the total amount of time from time t_{k_1} in δ, when $REQUEST(i)$ occurs, until the corresponding $GRANT(i)$ occurs at time $t_{k_2} + l$, is at least

$$n\left[c_2\left(\lfloor (m+l)/c_1 \rfloor + 1\right)\right] + l \ .$$

Since δ can be extended to a timed execution (By Lemma 2.3) the Theorem follows. ■

We note that Theorem 4.7 seems quite robust in that it can be extended to any reasonable model, including those in which the manager takes steps only in response to inputs. However, the better lower bound in Theorem 4.9 depends more heavily on the features of the timed automaton model. Note that the limiting case of the lower bound in Theorem 4.9 is

$$n[\lfloor m/c_1 \rfloor + 1]c_2 \ ,$$

which is slightly better than the lower bound given by Theorem 4.7.

5 A Distributed System

Now we consider the case where the computer system is distributed. We assume that the events concerning the different moving parts occur at separate manager processes p_i, $0 \leq i \leq n-1$, which communicate over unidirectional channels. More precisely, for each ordered pair (i,j), $i \neq j$, we assume that there is a channel automaton $channel(i,j)$ representing a channel from p_i to p_j, having *SEND* events as inputs and *RECEIVE* events as outputs. The channel operates as a FIFO queue; when the queue is nonempty, the channel is always enabled to deliver the first item. All *RECEIVE* actions are in the same partition class, with associated bounds $[0,d]$; this means that the channel will deliver the first item on the queue within time d. Also, we assume that there is a separate clock, $clock(i)$, for each process p_i. It is similar to the centralized clock described earlier, with output action $TICK(i)$ that is an input to p_i, and with associated bounds $[c_1, c_2]$. See Figure 3.

If the clocks are perfectly accurate, i.e., $c_1 = c_2$, then since all processes start at the same time, there is a very simple algorithm that assigns to each process a periodic predetermined "time slice" and whose worst case response time is $n \cdot m$ (plus some terms involving

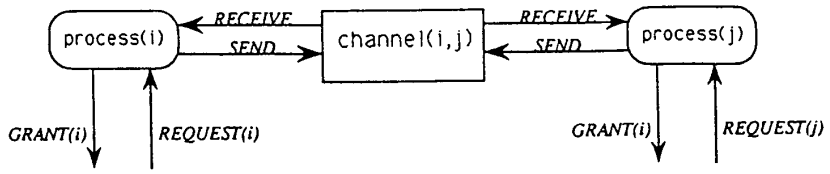

Figure 3: The architecture of the distributed control system.

and c_2 and l). This is optimal.[2] So, for our lower bound we will assume that $c_1 < c_2$.

5.1 The Upper Bound

5.1.1 The Algorithm

The following algorithm implements a round-robin granting policy: The processes issue grants when they are in possession of a token that circulates on a ring.

Assume processes are numbered $0, \ldots, n-1$ in clockwise order, and interpret $i+1$ to be $i+1$ mod n. Each process p_i has input actions *REQUEST*(i), *TICK*(i) and *RECEIVE-TOKEN*(i), output actions *GRANT*(i) and *SEND-TOKEN*(i), and internal action *ELSE*(i). The state of process i is divided into components:

REQUESTED	holding a Boolean value, initially *false*;
TIMER	holding an integer, initially 0;
TICKED	holding a Boolean value, initially *true*;
TOKEN	holding a value in {*not_here*, *available*, *used*}, initially *used* for p_0, *not_here* for the other processes.

Process p_i executes the following algorithm:

REQUEST(i)
Effect:
REQUESTED := *true*

TICK(i)
Effect:
TIMER := TIMER -1
TICKED : = *true*

GRANT(i)
Precondition:
REQUESTED = *true*
TOKEN = *available*
TICKED = *true*
Effect:
REQUESTED := *false*
TOKEN := *used*
TIMER := $\lfloor (m+l)/c_1 \rfloor + 1$
TICKED := *false*

SEND-TOKEN(i) /* to process p_{i+1} */
Precondition:
TOKEN = *used*
TIMER ≤ 0
Effect:
TOKEN := *not_here*
TICKED := *false*

ELSE(i)
Precondition:
neither GRANT(i)
nor *SEND-TOKEN*(i) is enabled
Effect:
TICKED := *false*

RECEIVE-TOKEN(i)
Effect:
if REQUESTED then TOKEN := *available*
else TOKEN := *used*

[2] In fact, even if we deviate from the model by allowing accurate clocks with non-synchronized starts, there is an algorithm which selects synchronization points so that its worst case response time is at most $n \cdot (m + (d/2))$ (plus some terms involving and c_2 and l). A corresponding lower bound can also be proved. A formal treatment of these results requires several changes to our model, and we prefer not to present it here. The clock synchronization algorithm of [LL84] yields synchronization points that can be used by a distributed allocation algorithm whose response time is at most $n \cdot m + (n-1)d$. Since the lower bound of [LL84] implies that this clock synchronization algorithm is optimal, it does not appear that a naive use of clock synchronization produces optimal resource allocation algorithms.

5.1.2 Correctness Proof

Now let B be the composition of all the given timed automata: operators, moving parts, processes, channels and clocks. This subsection is devoted to proving the following theorem.

Theorem 5.1 *Algorithm B is a correct distributed resource allocation algorithm.*

As in the proof of the centralized algorithm, we construct the I/O automaton $time(B)$. This time, the new state components are $Ctime$, plus, for each i, $Ftime$ and $Ltime$ for the following partition classes:

1. $REQUEST(i)$, which contains the single action $REQUEST(i)$,
2. $FINISH(i)$, which contains the single action $FINISH(i)$,
3. $TICK(i)$, which contains the single action $TICK(i)$, and
4. $LOCAL(i)$, the class of locally controlled actions of process i, which contains all the actions $GRANT(i)$, $SEND\text{-}TOKEN(i)$ and $ELSE(i)$.

Initially, we have $Ftime(REQUEST(i)) = 0$, $Ltime(REQUEST(i)) = \infty$, $Ftime(FINISH(i)) = 0$ and $Ltime(FINISH(i)) = \infty$, $Ftime(TICK(i)) = c_1$, $Ltime(TICK(i)) = c_2$, $Ftime(LOCAL(i)) = 0$ and $Ltime(LOCAL(i)) = l$.

Let $\#tokens(i)$ be the length of the queue in $channel(i, i+1)$. We first prove a lemma giving an invariant for $time(B)$; this invariant happens not to involve any of the state components that encode time information. The proof can be found in [AL89].

Lemma 5.2 *Let s be a reachable state of $time(B)$. Then the total number of processes at which TOKEN $\neq$ not_here plus the sum of $\#tokens(i)$, over $0 \leq i < n$, is exactly 1.*

We now prove another invariant, this one involving the timing information. The result is similar to Lemma 4.2. The proof can be found in [AL89].

Lemma 5.3 *Let s be a reachable state of $time(B)$, and let $0 \leq i \leq n-1$. Then the following all hold:*

1. *If $FINISH(i)$ is enabled in $s.Astate$, then*
   (a) $s.TIMER(i) > 0$,
   (b) $s.Ftime(TICK(i)) + (s.TIMER(i) - 1)c_1 > s.Ltime(FINISH(i))$, *and*
   (c) $s.TOKEN(i) = used$.
2. *If $s.TICKED(i) = true$ then $s.Ftime(TICK(i)) \geq s.Ltime(LOCAL(i)) + c_1 - l$.*

The following corollary implies that mutual exclusion is maintained by the algorithm.

Corollary 5.4 *In any reachable state s of B, if $FINISH(i)$ is enabled, for some i, then $FINISH(j)$ is not enabled for all $j \neq i$.*

Proof: (of Theorem 5.1) Corollary 5.4 implies mutual exclusion. Moving part well-formedness follows from the same corollary and the definition of the moving part. Request well-formedness follows from the definitions of the operators and the processes. Eventual granting can be argued from the round-robin behavior of the processes; it also follows from the upper bound on response time proved formally in the following subsection. ■

5.2 Response Time

Now we prove the upper bound on response time for the given distributed algorithm B.

Theorem 5.5 *The worst case response time for algorithm B is at most*

$$n[c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l].$$

We use the following lemmas.

Lemma 5.6 *In any reachable state s, and for any i,*

$$s.TIMER(i) \leq \lfloor (m+l)/c_1 \rfloor + 1.$$

Lemma 5.7 *Let s be any state occurring in a timed execution, in which $s.TIMER(i) \leq k$, for $k \geq 1$. Then (at least) one of the following two conditions holds.*

1. *$s.TIMER(i) \leq 0$ and $s.TICKED(i) = true$, or*
2. *the time from the given occurrence of s until a later $TICK(i)$ event resulting in $TIMER(i) \leq 0$ is bounded above by $c_2 \cdot k$.*

Say that process p_i is *operative* in state s if $s.\text{TOKEN}(i) = used$. By Lemma 5.2 at any time there is at most one operative process.

Lemma 5.8 *If process p_i is operative, then the time until process p_{i+1} becomes operative is at most*

$$c_2(\lfloor (m+l)/c_1 \rfloor + 1) + d + c_2 + 2l\ .$$

Define the *distance* from process p_i to process p_j to be the distance between them along the ring (in the clockwise direction); if $i = j$ we define the distance to be n.

Proof: (of Theorem 5.5) Consider the point in the timed execution at which a request arrives, say at process p_j. We consider cases (one of which must hold, by Lemma 5.2).

1. There is some operative process, p_i, when the request arrives (where it is possible that $i = j$). Then the distance from p_i to p_j is at most n. Applying Lemma 5.8 repeatedly (at most n times) yields the claimed bound.

2. The value of TOKEN(i) = *available* for some i. If $i = j$, then the request will be granted within time $c_2 + l$. If $i \neq j$, then within time $c_2 + l$, process p_i becomes operative. Applying Lemma 5.8 repeatedly (at most $n - 1$ times) yields the claimed bound.

3. There is a message in one of the channels, say *channel*$(i-1, i)$. If $i = j$, then the request will be granted within time $d + c_2 + l$. If $i \neq j$, then within time $d + c_2 + l$, process p_i becomes operative. Applying Lemma 5.8 repeatedly (at most $n - 1$ times) yields the claimed bound.

■

Again, we note that the limiting case of the upper bound as l approaches 0, is

$$n\left[c_2\left(\lfloor m/c_1 \rfloor + 1\right) + d + c_2\right] .$$

5.3 Lower Bound

Now we prove our lower bound on worst case response time for arbitrary distributed resource allocation algorithms. This proof is similar to that of the simple lower bound for centralized algorithms (Theorem 4.7) rather than the more complicated tight bound (Theorem 4.9) in that we do not concern ourselves with process step time or with roundoffs. As a result, this proof seems sufficiently robust to extend to other reasonable models for timing-based computation.

Note that the gap between our upper and lower bounds for the distributed case does not *only* involve process step times and roundoffs, but also involves additive terms of d and of $n \cdot c_2$.

In order to prove this lower bound we must make the assumption that the moving time is much larger than the message delivery time, more precisely, that $(n-1) \cdot d \leq m(c_2/c_1)$.

Theorem 5.9 *Assume that $c_1 < c_2$ and that $(n-1) \cdot d \leq m \cdot (c_2/c_1)$. Then the worst case response time of any distributed resource allocation algorithm is at least*

$$n \cdot c_2(m/c_1) + (n-1) \cdot d .$$

The lower bound is proved under the assumption that every message is delivered within time d. This is a stronger assumption than the one used for the upper bound; there, we only insist that this upper bound hold for the *first* message on any link. Since the present assumption is stronger, it only serves to strengthen the lower bound.

In the proof we first show that the round-robin granting policy used by the algorithm of Section 5.1 is optimal in the following sense: for any "efficient" algorithm, in any execution in which requests arrive continuously, the order in which requests are first granted must be repeated in a round-robin fashion.

Once such an order has been established, we extend the execution while fixing a particular pattern of message delays. After doing this for a sufficiently long time, we retime parts of the execution by carefully "shifting" certain events, while appropriately retiming other events, to get the desired time bound.

Recall the definition of a *heavily loaded* timed execution or timed semi-execution from Section 4.2. In a manner similar to the centralized case, we define a timed execution or timed semi-execution to be *slow* if, for each i, the times between successive $TICK(i)$ events (and the time of the first $TICK(i)$ event) are exactly c_2. The following lemma is the distributed version of Lemma 4.8.

Lemma 5.10 *Let α be a slow timed execution of a correct distributed resource allocation algorithm. Then the time between any two consecutive GRANT events in α is strictly greater than*

$$c_2(m/c_1) .$$

The next lemma shows that if an execution is heavily loaded, the best policy (for an "efficient" algorithm) is to grant the resource in a round robin manner, because changing the granting order will cause the response time to exceed a bound higher than the one we are attempting to prove as a lower bound.

Lemma 5.11 *Let B be a distributed resource allocation algorithm with response time at most $(n+1) \cdot c_2(m/c_1)$. Let α be a slow timed execution of B that is heavily loaded starting from time t. Then there exists some permutation, ρ, of $\{0, \ldots, n-1\}$ such that the subsequence of all GRANT events that occur in α after time t is of the form*

$GRANT(\rho_0), \ldots, GRANT(\rho_{n-1})$,
$GRANT(\rho_0), \ldots, GRANT(\rho_{n-1}), \ldots$

Proof: (of Theorem 5.9) Assume by way of contradiction that there is some algorithm that always responds within time

$$n \cdot c_2(m/c_1) + (n-1)d \ .$$

By assumption

$$(n-1)d \leq m(c_2/c_1) \ ,$$

which implies that

$$n \cdot c_2(m/c_1) + (n-1) \leq (n+1) \cdot c_2(m/c_1) \ .$$

Thus, the response time for the algorithm is at most

$$(n+1) \cdot c_2(m/c_1) \ .$$

We will construct a slow timed execution of the algorithm that either exceeds the claimed bound on response time or violates the *mutual exclusion* property. We begin by considering a slow timed execution α' that is heavily loaded starting from some time t, and letting α be the shortest prefix of this timed execution that ends just after exactly n $GRANT$ events have occurred after time t. Lemma 5.11 implies that there is some permutation ρ, such that all $GRANT$ events that appear in α' after time t occur in the order $\rho_0, \ldots, \rho_{n-1}, \rho_0, \ldots$ In fact, Lemma 5.11 implies that $GRANT$ events that occur after time t in any timed semi-execution that extends α and is heavily loaded starting from time t, appear in the order $\rho_0, \ldots, \rho_{n-1}$. We sometimes abuse notation and write $p_{\rho_i} < p_{\rho_j}$ when $i < j$, that is p_{ρ_i} precedes p_{ρ_j} in the the order established by ρ.

We now consider the "ring" of processes formed by the round-robin order defined above. We extend the execution in such a way that messages are delivered with maximum delay when sent from lower numbered processes to higher numbered processes (in the order established by ρ), while messages going the other way are delivered immediately. Intuitively, this enables us to "postpone" notification of the granting as long as possible.

More formally, we extend α to get a slow timed execution $\alpha\beta'$ which is heavily loaded starting from time t and such that the message delivery times for messages sent in β' are as follows:

- If $i < j$, then a message from p_{ρ_i} to p_{ρ_j} takes exactly time d.
- If $i > j$, then a message from p_{ρ_i} to p_{ρ_j} takes exactly time 0.

Let $\alpha\beta$ be a "sufficiently long" prefix of $\alpha\beta'$, specifically, one for which

$$\frac{c_1}{c_2} \leq \frac{t_{end}(\alpha\beta) - t_{end}(\alpha) - d}{t_{end}(\alpha\beta) - t_{end}(\alpha)}.$$

This can be easily done since, by assumption, $c_1/c_2 < 1$. Let $r_1 = t_{end}(\alpha)$ and $r_2 = t_{end}(\alpha\beta)$.

Let γ be such that $\alpha\beta\gamma = \alpha\beta'$. We know that γ contains a subsequence of $n+1$ consecutive $GRANT$ events, in order

$GRANT(\rho_0), GRANT(\rho_1), \ldots,$
$GRANT(\rho_{n-1}), GRANT(\rho_0).$

Now divide γ into $n+2$ *segments*, $\gamma_0, \ldots, \gamma_{n+1}$, where

1. γ_0 ends with the first of these $GRANT(\rho_0)$ events,
2. for each $i, 1 \leq i \leq n-1$, γ_i starts just after $GRANT(\rho_{i-1})$ and ends with $GRANT(\rho_i)$,
3. γ_n starts just after $GRANT(\rho_{n-1})$ and ends with the second $GRANT(\rho_0)$, and
4. γ_{n+1} includes the rest of γ.

For each $i, 0 \leq i \leq n+1$, let $t_i = t_{end}(\alpha\beta\gamma_0 \ldots \gamma_i)$. For any $1 \leq i \leq n$, define the *length* of any segment γ_i, to be $\ell_i = t_i - t_{i-1}$. Intuitively, ℓ_i is the amount of time that passes during γ_i.

We now prove a key lemma that provides a lower bound for the length of each segment $\gamma_1, \ldots, \gamma_{n-1}$.

Lemma 5.12 *For any* $i, 1 \leq i \leq n-1$,

$$\ell_i > c_2(m/c_1) + d.$$

Proof: Assume by way of contradiction that

$$\ell_i \leq c_2(m/c_1) + d$$

for some particular $i, 1 \leq i \leq n-1$.

From $\alpha\beta\gamma$ we construct a new timed execution, $\alpha\delta$, in which the *mutual exclusion* property is violated. We first construct an intermediate timed execution $\alpha\delta'$ in which we "shift" back in time the events occurring at processes $p_{\rho_i}, \ldots, p_{\rho_{n-1}}$, in the following way:

1. Each event occurring at any of the processes $p_{\rho_0}, \cdots, p_{\rho_{i-1}}$ that occurs in $\beta\gamma$ at time u, also occurs in δ' at time u.
2. Each event occurring at any of the processes $p_{\rho_i}, \cdots, p_{\rho_{n-1}}$ that occurs in $\beta\gamma$ at time u, occurs in δ' at time u' where:

(a) If $u > r_2$ then $u' = u - d$.

(b) If $r_1 \leq u \leq r_2$ then

$$u' = r_1 + \frac{r_2 - r_1 - d}{r_2 - r_1} \cdot (u - r_1).$$

I.e., $\frac{u'-r_1}{u-r_1} = \frac{r_2-r_1-d}{r_2-r_1}$.

That is, the events occurring at processes $\geq p_{\rho_i}$ at times $> r_2$ are moved d earlier; notice that events occurring in α (at times $\leq r_1$) are not moved. All the intermediate events are shifted back proportionally.

The resulting sequences of timed events must be merged into a single sequence consistently with the order of the times; events occurring at different processes at the same time can be merged in arbitrary order, except that a *SEND* event that corresponds to a *RECEIVE* event in $\alpha\beta\gamma$ must precede it in $\alpha\delta'$.

Claim 5.13 *$\alpha\delta'$ is a timed execution of the system.*

Now we resume the proof of Lemma 5.12. Note the following additional properties of $\alpha\delta'$:

- Any clock tick interval at a process $\leq p_{\rho_{i-1}}$ takes time exactly c_2.
- Any clock tick interval at a process $\geq p_{\rho_i}$ that begins at a time $\geq r_2 - d$ takes time exactly c_2.
- Any clock tick interval at a process $\geq p_{\rho_i}$ that begins at a time $\leq r_2 - d$ and ends at a time $u > r_2$ takes time at least $u - r_2 + (c_2 - (u - r_2))(c_1/c_2)$.
- The length of the new segment corresponding to γ_i is at most $c_2(m/c_1)$.

Now to get $\alpha\delta$ from $\alpha\delta'$, we "shrink" the portion of $\alpha\delta'$ after time r_2 by the ratio (c_1/c_2) and move the $FINISH(\rho_{i-1})$ event (of segment γ_i) after the $GRANT(\rho_i)$ event (at the end of segment γ_i), thus creating a violation of the *mutual exclusion* property. More precisely, if an event happens at time u' in $\alpha\delta'$, then the corresponding event happens at time u in $\alpha\delta$, where:

1. If $u < r_2$, then $u' = u$.
2. If $u \geq r_2$, then $u' = r_2 + (c_1/c_2)(u - r_2)$.

Claim 5.14 *$\alpha\delta$ is a timed execution of the system.*

To complete the proof of Lemma 5.12, we need only observe that $\alpha\delta$ is a timed execution of the system in which the *mutual exclusion* property is violated, a contradiction. ■

To complete the proof of Theorem 5.9, consider the execution $\alpha\beta\gamma$ and consider the $REQUEST(\rho_0)$ that occurs just after the first of the designated $GRANT(\rho_0)$ events in γ. From Lemma 5.10 it follows that

$$\ell_n > c_2(m/c_1).$$

Together with Lemma 5.12 this implies that the total time from that $REQUEST(\rho_0)$ event until the corresponding $GRANT(\rho_0)$ event is strictly greater than

$$\begin{aligned}&(n-1)(c_2(m/c_1) + d) + c_2(m/c_1)\\ &= n \cdot c_2(m/c_1) + (n-1)d\,,\end{aligned}$$

as claimed. ■

6 Discussion and Open Problems

In this paper, we have defined a timing-based variant of the mutual exclusion problem, and have considered both centralized and distributed solutions to this problem. We have proved upper bounds for both cases, based on simple algorithms; these bounds are fairly complicated functions of clock time, manager or process step time, moving time for the moving parts, and (in the distributed case) message delivery time.

We also have proved corresponding lower bounds for both cases. In the centralized case, the lower bound exactly matches the upper bound, even when the manager step time and the roundoffs are considered. In the more complicated distributed setting, the lower bound is very close to the upper bound, but does not match it exactly.

The bounds are all proved using the *timed automaton* model for timing-based concurrent systems. It is interesting to ask how dependent the results are on this choice of model. The timed automaton model differs from some others in modeling process steps explicitly (rather than assuming the algorithms are interrupt-driven); thus, our results involving this process step time would not be expected to extend immediately to such interrupt-driven models (except possibly in the limit, as this step time approaches zero). However, some of our results - most notably, the lower bound for the distributed case - do not involve process step times and thus appear to be quite model-independent. An alternative approach would be to use a general model that describes interrupt-driven computation, but we do not yet know (in general) how to define such model.

There are several open questions directly related to the work presented in this paper. First, there is a gap

remaining between the upper and lower bound results for the distributed resource allocation problem. Even neglecting process step time, there is a difference of an additive terms of d, the upper bound on message delivery time, and $n \cdot c_2$, then number of processes times the upper bound on the clock tick time. Preliminary results suggest that under certain assumptions about the relative sizes of the parameters, the upper bound can be reduced by approximately d. However, we do not yet have a general result about this.

Our lower bound for the distributed resource allocation problem assumes that $(n-1) \cdot d \leq m \cdot (c_2/c_1)$. It would be interesting to see if this assumption can be removed.

It would also be interesting to consider the same problem in a model in which there are nontrivial lower bounds on the time for message delivery (and perhaps for process steps). While our upper bound proofs still work in this situation, the same is not true for our lower bound proofs. The strategy of shrinking and shifting timed executions to produce other timed executions becomes much more delicate when lower bounds on these various kinds of events must also be respected.

Our results imply that the ratio c_2/c_1 has a significant impact on the response time of the system. It would also be interesting to consider the case where a process has more than one clock, say an additional clock with bounds $[c_1', c_2']$. We would like to understand how the results depend on the four parameters c_1, c_2, c_1' and c_2'.

Other related problems can also be studied using the models and techniques of this paper. One could define timing-based analogs of other problems besides mutual exclusion that have been studied in the asynchronous setting (for example, other exclusion problems such as the *dining philosophers* problem, distributed consensus problems, or synchronization problems such as the *session problem* of [AFL81]); it should be possible to obtain combinatorial results about them in the style of the results of this paper. In addition to defining variants of asynchronous problems, one can also extract prototypical problems from practical real-time systems research and use them as a basis for combinatorial work.

In another direction, the algorithm proofs presented here suggests general approaches to verification of real-time systems. As mentioned in Section 4.1.3, we believe that there may be a unified method for treating *correctness* and *performance analysis* of timing-based algorithms, and are currently exploring this possibility in [LA].

Work of the sort presented here (and the extensions proposed above) should provide an excellent basis for evaluating the timed automaton model as a general model for reasoning about timing-based systems (and comparing it with alternative models for timing-based computation).

Acknowledgements

We would like to thank Nancy Leveson for providing us with background information on real-time systems, and for suggestions and encouragement in the early stages of this work. Thanks are also due to Jennifer Welch for discussions about clock synchronization and for reading the paper and providing us with very valuable comments. We would also like to thank Michael Merritt and Mark Tuttle for discussions about modeling time and John Keen and Steve Ponzio for comments on earlier versions of this paper.

References

[AL89] H. Attiya and N. A. Lynch, "Time bounds for real-time processcontrol in the presence of timing uncertainty," Technical Report MIT/LCS/TR-403, Laboratory for Computer Science, MIT, July 1989.

[AFL81] E. Arjomandi, M. J. Fischer and N. Lynch, "Efficiency of synchronous versus asynchronous distributed systems," *Journal of the ACM*, Vol. 30, No. 3 (July 1983), pp. 449–456.

[BH81] A. Bernstein and P. Harter, Jr. "Proving real-time properties of programs with temporal logic," *Proc. 8th Symp. on Operating System Principles*, Operating Systems Review, Vol. 15, No. 5 (December 1981), pp. 1–11.

[CR83] J. E. Coolahan and N. Roussopoulus, "Timing requirements for time-driven systems using augmented Petri nets," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5 (September 1983), pp. 603–616.

[D85] B. Dasarathy, "Timing constraints of real-time systems: Constructs for expressing them, methods for validating them," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1 (January 1985), pp. 80–86.

[DHS86] D. Dolev, J. Halpern and H. R. Strong, "On the possibility and impossibility of achieving clock synchronization." *Journal of Computer and Systems Sciences*, Vol. 32, No. 2 (1986) pp. 230–250.

[HMM85] J. Halpern, N. Megiddo and A. A. Munshi, "Optimal precision in the presence of uncertainty." *Journal of Complexity*, Vol. 1 (1985), pp. 170–196.

[H81] V. H. Hasse, "Real-time behavior of programs," *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5 (September 1981), pp. 494–501.

[HGR87] C. Huizing, R. Gerth, and W. P. deRoever, "Full abstraction of a real-time denotational semantics for an OCCAM-like language," in *Proc. 14th ACM Symp. on Principles of Programming Languages*, 1987, pp. 223–237.

[JM86] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 9 (September 1986), pp. 890–904.

[JM87] F. Jahanian and A. Mok, "A graph-theoretic approach for timing analysis and its implementation," *IEEE Transactions on Computers*, Vol. C-36, No. 8 (August 1987), pp. 961–975.

[KSRGA88] R. Koymans, R. K. Shyamasundar, W. P. deRoever, R. Gerth, and S. Arun-Kumar, "Compositional semantics for real-time distributed computing," *Information and Computation*, Vol. 79, No. 3 (December 1988), pp. 210–256.

[L78] L. Lamport, "Time, clocks and the ordering of events in distributed systems." *Communications of the ACM*, Vol. 21, No. 7 (July 1978), pp. 558-565.

[LS87] N. Leveson and J. Stolzy, "Safety analysis using Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 3 (March 1987), pp. 386–397.

[LL84] J. Lundelius and N. Lynch, "An upper and lower bound for clock synchronization," *Information and Control*, Vol. 62, Nos. 2/3 (August/September 1984), pp. 190–204.

[L88] N. Lynch, "Modelling real-time systems," in *Foundations of Real-Time Computing Research Initiative*, ONR Kickoff Workshop, November 1988, pp. 1–16.

[LA] N. Lynch and H. Attiya, "Assertional Proofs for Timing Properties," in progress.

[LT87] N. Lynch and M. Tuttle, "Hierarchical Correctness Proofs for Distributed Algorithms," in *Proc. 7th ACM symp. on Principles of Distributed Computing*, August 1987, pp. 137–151.

Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, MIT, April 1987.

[MMT88] M. Merritt, F. Modugno and M. Tuttle, "Time constrained automata," manuscript, November 1988.

[S77] J. Sifakis, "Petri nets for performance evaluation, in Measuring, Modeling and Evaluating Computer Systems," in *Proc. 3rd Symp. IFIP Working Group 7.3*, H. Beilner and E. Gelenbe (eds.), Amsterdam, The Netherlands, North-Holland, 1977, pp. 75–93.

[SWL88] B. Simons, J. L. Welch and N. Lynch, "An overview of clock synchronization," IBM Technical Report RJ 6505, October 1988.

[WL88] J. L. Welch and N. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Information and Computation*, Vol. 77, No. 1 (April 1988), pp. 1–36.

[ZLG89] A. Zwarico, I. Lee and R. Gerber, "A complete axiomatization of real-time processes," Submitted for publication.