Time-Adaptive Algorithms for Synchronization*

Rajeev Alur[†] Hagit Attiya[‡] Gadi Taubenfeld[§]

Abstract

We consider concurrent systems in which there is an unknown upper bound on memory access time. Such a model is inherently different from asynchronous model where no such bound exists, and also from timing-based models where such a bound exists and is known a priori. The appeal of our model lies in the fact that while it abstracts from implementation details, it is a better approximation of 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.

Two basic synchronization problems, consensus and mutual exclusion, are investigated in a shared memory environment that supports atomic read/write registers. We show that $\Theta(\Delta\frac{\log\Delta}{\log\log\Delta})$ is an upper and lower bound on the time complexity of consensus, where Δ is the (unknown) upper bound on memory access time. For the mutual exclusion problem, we design an efficient algorithm that takes advantage of the fact that some upper bound on memory access time exists. The solutions for both problems are even more efficient in the absence of contention, in which case their time complexity is a constant.

1 Introduction

The possibility and complexity of synchronization in a distributed environment depends heavily on timing assumptions. In the asynchronous model no timing assumptions are made about the relative speeds of the processes, while a timing-based model assumes known bounds on the speeds of the processes. Although the asynchronous model is weaker than the timing-based model, it provides a useful abstraction of the timing constraints, and algorithms designed for asynchronous model work correctly in all possible environments. However, sometimes 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.

*A preliminary version of this work appeared in *Proceedings of the Twenty-sixth Annual Symposium on Theory of Computing (STOC),* Montreal, Quebec, Canada, May 1994, pp.800-809.

†Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Murray Hill, NJ 07974 (email: alur@research.bell-labs.com).

‡Computer Science Department, Technion, Haifa 32000, Israel (email: hagit@cs.technion.ac.il). This work was performed while visiting at AT&T Bell Laboratories. This research was partially supported by grant No. 92-0233 from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel.

§Bell Laboratories and Isreal Open University.

We focus on an intermediate model which provides an alternative abstraction of the timing details of concurrent systems. We assume that there is an unknown upper bound on memory access time. This assumption is inherently different from the asynchronous model where no such bound exists, and from timing-based systems where such a bound exists and is known a priori. 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. The importance of a timing-based model with unknown bounds is also supported by an earlier work of Dwork et al. in the context of message-passing systems [DLS88].

We use a shared memory model where processes communicate with each other by reading and writing to shared registers. We assume that there is an upper bound, denoted by Δ, on time required for a single access to shared memory. There is no lower bound on time needed to execute a step, but a process can delay itself explicitly by executing a statement $delay(d)$, for some constant d. The resulting model is called the *known-delay* model or the *unknown-delay* model depending on whether or not this bound is known a priori. An algorithm in the unknown-delay model is required to be correct for all possible choices of Δ, and hence, cannot refer to Δ directly. We show that the unknown-delay model is inherently different from both the known-delay model and the asynchronous model by investigating two basic synchronization problems, consensus and mutual exclusion.

In the *consensus* problem, processes need to agree on a common output in the presence of possible failures [PSL80]. It has been proven that when even one process can fail, the consensus problem is not solvable in asynchronous systems [DDS87, FLP85, LA87]. In the known-delay model, there is an algorithm which tolerates any number of failures and terminates within time $O(\Delta)$ [AT96].

Our first result is a consensus algorithm that works in the unknown-delay model. The algorithm guarantees that, in every possible execution, processes never decide on conflicting values and the decision value is an input value of some process. If every step finishes within time Δ, then a process decides within time $O(\Delta \cdot fac^{-1}(\Delta))$ irrespective of the failures of other processes, where fac^{-1} is the inverse of the factorial function;[1] note that $fac^{-1}(r) = \Theta(\frac{\log r}{\log \log r})$. Furthermore, the algorithm is *fast*: in absence of contention, a process decides after a constant number of its own steps.

Our second result shows that the worst-case time complexity of any 2-process algorithm for consensus in the unknown-delay model is $\Omega(\Delta \cdot fac^{-1}(\Delta))$; this implies that our algorithm is time-optimal. The lower bound implies that not knowing Δ multiplies the time complexity by a factor of $fac^{-1}(\Delta)$.

The *mutual exclusion* problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. A mutual exclusion algorithm satisfies the *fast access property* if in absence of contention a process needs to execute only a constant number of steps in order to enter or exit its critical section. In [Lam87], Lamport presented a fast mutual exclusion algorithm that satisfies the fast access property. In the presence of contention, however small, the winning process in Lamport's algorithm may have to check the status of all other n processes before it is allowed to enter its critical section. In the known-delay model, there is an algorithm that

[1] For a real number $d > 0$, $fac^{-1}(d)$ is the smallest r such that $r! \geq d$.

satisfies the fast access property without requiring the winning process to check the status of all other n processes in the presence of contention [AT96]; other algorithms which satisfy the fast access property can be found in [CS93, MS93, Sty92, YA93].

Our third result is a mutual exclusion algorithm for the unknown-delay model where in the presence of contention, a process needs to delay itself for $2 \cdot \Delta$ time units before entering its critical section. The algorithm has a "warm-up" period during which the processes might have to access n registers before entering the critical section. The algorithm always provides fast access in the absence of contention.

For both problems, the knowledge of Δ is beneficial: in the case of consensus, the problem becomes solvable, and in the case of mutual exclusion, more efficient solutions can be obtained. Our results imply that these benefits can be achieved even when Δ is unknown.

Dwork et al. have studied the consensus problem in message-passing systems where there are unknown bounds on the time to deliver a message and on processes speed [DLS88]. Their work concentrates on the percentage of faulty processes (compared to the total number of processes) that can be tolerated. Our consensus algorithm is wait-free, that is, it can tolerate any number of crash failures.

Herzberg and Kutten [HK89] have studied a message-passing model where a priori upper bound on message delivery time is known, but is much larger than the actual message delay; this encourages the use of the (unknown) message delivery time in the algorithm. They considered the problem of detecting faulty processes.

In Section 2 a formal model is introduced and the issue of how to measure the time complexity is discussed. Section 3 is dedicated to the consensus problem; matching upper and lower bounds are presented for the worst-case time of a consensus algorithm. The fast algorithm for mutual exclusion appears in Section 4. We conclude with a discussion of our results and directions for future work.

2 A Timing-Based Model

In this section, we outline our model of distributed systems. Processes are modeled as (possibly infinite) state-machines communicating via shared memory consisting of registers that that support atomic reads and writes.

A *configuration* of the system includes the state of each process and the values of all shared registers. An *event* is a single step of some process, and is either a read of a shared register, or a write to a shared register, or simply an update of the internal state of a process. Processes can also execute a delay statement $delay(d)$, for a positive integer d, and its effect on a configuration is the same as a skip statement.

As in the standard interleaving semantics, an *execution* α of the system is an alternating sequence $s_0 \stackrel{e_0}{\to} s_1 \stackrel{e_1}{\to} \cdots$ of configurations s_i and events e_i such that (1) the initial configuration s_0 satisfies some initial conditions, and (2) every configuration s_{i+1} is derived from the previous configuration s_i by executing the event e_i.

We allow only crash failures: a failed process simply ceases to participate. Formally, a process p is *nonfaulty* in an execution α iff either α is finite or p takes infinitely many steps in α.

The notion of an execution captures only the asynchronous part of the system and not its timing requirements. Define the *explicit delay* of an event e, denoted by $d(e)$, to be n if e is the delay statement $delay(n)$, and 0 otherwise. A *time assignment* τ for an execution α is a mapping that assigns a real-valued occurrence time τ_i to each event e_i in α such that

1. the occurrence times are nondecreasing,

2. if α is infinite then the sequence of occurrence times is unbounded[2], and

3. whenever two events e_i and e_j are consecutive steps of the same process, then difference $\tau_j - \tau_i$ is greater than $d(e_i)$.

The last requirement captures the assumption regarding the lower bounds on execution speeds. A delay statement $delay(d)$ by a process p delays p for at least d time units before it can continue. For other statements, we simply require that a step of a process takes nonzero time [3] (note that adjacent events belonging to different processes can be assigned the same time). As an example, consider the following execution, where each event is labeled with the process it belongs to:

$$\alpha_{sample}: \; s_0 \xrightarrow{p:read(x)} s_1 \xrightarrow{q:write(y)} s_2 \xrightarrow{p:delay(4)} s_3 \xrightarrow{p:read(y)} s_4 \xrightarrow{q:read(x)} s_5.$$

A time assignment τ for the above execution α_{sample} is a sequence $\tau_0 \leq \tau_1 \leq \cdots \leq \tau_4$ such that

$$\tau_2 - \tau_0 > 0, \; \tau_3 - \tau_2 > 4, \; \tau_4 - \tau_1 > 0.$$

Thus, a possible time assignment is

$$\tau_{sample}: \; \tau_0 = 0, \; \tau_1 = 0, \; \tau_2 = 0.1, \; \tau_3 = 4.5, \; \tau_4 = 5.$$

The definition of explicit delay of an event is extended to finite executions also: for a finite execution $\alpha = s_0 \xrightarrow{e_0} \cdots s_n$, let $d(\alpha)$ denote the sum $\sum_{0 \leq i < n} d(e_i)$ of explicit delays of all the events in α. For instance, $d(\alpha_{sample})$ equals 4.

The assumption about the time needed to access shared memory is reflected in the following notion of admissibility. Let Δ be a positive real number.

> A timing assignment τ for an execution α is said to be Δ-*admissible* iff whenever two events e_i and e_j are consecutive steps of the same process, $\tau_j - \tau_i \leq \Delta + d(e_i)$.

Thus if the i-th step in an execution is a read or a write by process p, then the next step by process p must be within time Δ; if it is the delay statement $delay(d)$, then p's next step must be within time $\Delta + d$. If a process does not take the next step within this bounded time period, then it can never take a step, implying a crash failure.

Every execution has several time assignments, but may not have a Δ-admissible time assignment for a given Δ; this is because delay statements restrict the possible timing assignments. An execution α is Δ-*admissible* if there exists a Δ-admissible time assignment for α. For our sample execution α_{sample}, the timing assignment τ_{sample} is 5-admissible (in

[2] Our results do not depend on this second requirement.

[3] Our definition allows delays of read or write steps to be as small as we want. In a context where a non-negligible lower bound, say ϵ, on these steps is more appropriate, we can simply insert the statement $delay(\epsilon)$ after every step.

fact, it is Δ-admissible iff $\Delta \geq 5$). The execution α_{sample} itself is Δ-admissible for every $\Delta > 4$.

A problem such as consensus or mutual exclusion is usually specified by listing the properties to be satisfied by all the executions. An algorithm A satisfies a property ϕ in the asynchronous model if and only if all of its executions satisfy ϕ. While solving a problem in the timing-based model with an upper bound of Δ on the step-time, a key issue is whether the processes know the upper bound Δ.

In the *known-delay* model, we assume that individual processes know the upper bound Δ. Consequently, delay statements can refer directly to this value, and a process can enforce every other (nonfaulty) process to take at least one step by executing the statement $delay(\Delta)$. To solve a problem in this model, we want a family of algorithms $A(\Delta)$, parameterized by the upper bound Δ, such that for each Δ, all Δ-admissible executions of $A(\Delta)$ satisfy all the requirements of the problem.

In the *unknown-delay* model, we assume that some upper bound exists, but it is not known to individual processes a priori. In this model we want a single algorithm A that works for all possible values of Δ without referring to its actual value. We will say that an algorithm A satisfies a property ϕ in the unknown-delay model, if for every Δ, all Δ-admissible executions of A satisfy ϕ.

Before we consider specific problems, let us observe one property of algorithms in the unknown-delay model. A property ϕ is a *safety* property if the following holds: An infinite execution α satisfies ϕ if and only if all finite prefixes of α satisfy ϕ; that is, a safety property has to be *prefix-closed*. The unknown-delay model is the same as the asynchronous model as far as safety properties are concerned:

Lemma 1 *An algorithm A satisfies a safety property ϕ in the asynchronous model if and only if it satisfies ϕ in the unknown-delay model.*

Proof: Clearly, if A satisfies a property ϕ in the asynchronous model then it satisfies ϕ in the unknown-delay model. Suppose A does not satisfy ϕ in the asynchronous model. Then there is an execution α of A which violates ϕ. If ϕ is a safety property then there is a finite prefix α' of α such that α' violates ϕ. The finite execution α' is Δ-admissible for every $\Delta \geq d(\alpha')$. This implies the lemma. ■

Lemma 1 does not hold for liveness properties such as termination.

We now define our time complexity measures. Given an execution α and a time assignment τ for it, suppose $time(\alpha, \tau)$ is a measure of time taken according to τ. The exact definition of *time* depends upon the problem, and whether we are computing the worst-case complexity, or the contention-free complexity. For instance, in consensus, *time* may denote the maximum time spent by a process between its first step and its decision step in α. We denote by $time_\Delta(\alpha)$ the maximum of $time(\alpha, \tau)$ over all Δ-admissible time assignments τ for α. For an algorithm A, $time_\Delta(A)$ denotes the maximum of $time_\Delta(\alpha)$ over all Δ-admissible executions α of A.

Sometimes we will also need an estimate of how much time is spent due to explicit delay statements. For an execution α, let $min\text{-}time_\Delta(\alpha)$ denote the greatest lower bound on $time(\alpha, \tau)$ over all Δ-admissible time assignments τ for α; it gives the minimum time spent by a process in α. For instance, if we define $time(\alpha_{sample}, \tau)$ as $\tau_4 - \tau_0$, then for $\Delta > 4$, $time_\Delta(\alpha_{sample})$ is $2\Delta + 4$ and $min\text{-}time_\Delta(\alpha_{sample})$ is 4.

If *time* measures the *contention-free* complexity which is the time spent by a process when it executes by itself, then according to [Lam87, AT96], an algorithm is *fast* if and only if $time_\Delta(A)$ is $O(\Delta)$ and $min\text{-}time_\Delta(A)$ is zero. This implies that in absence of contention, a process executes only a constant number of steps and no explicit delay statements.

3 Time-Adaptive Consensus

In this section, we consider the problem of (binary) consensus in the unknown-delay model, and provide tight bounds for its worst-case time complexity.

The *consensus problem* is to design an algorithm in which all correct processes reach a common decision based on their initial inputs [PSL80]. Formally the problem is defined as follows. There are n processes, and each process p_i has an input value $in_i \in \{0, 1\}$. A process p_i *decides* on a value $v \in \{0, 1\}$ by executing the statement $decide(v)$. It may decide at most once. The consensus problem requires that

- *Agreement:* there exists a *decision value* $out \in \{0, 1\}$ such that if a process p_i decides on the value v then $v = out$, and
- *Validity:* if a process p_i decides on the value v then v equals the input value in_j for some process p_j.

Thus, no two processes decide on conflicting values, and if all input values are the same then that value must be the decision value. Apart from the above safety requirements, we want the correct processes to eventually decide

- *Wait Freedom:* each process p_i either takes only finitely many steps or decides on some value.

The requirement of wait-freedom means that one process cannot prevent another process from reaching a decision, and thus the algorithm must tolerate arbitrary number of process failures.

3.1 The Algorithm

In this section, we present a consensus algorithm. The algorithm always guarantees the safety requirements of agreement and validity. The liveness requirement is ensured using timing assumptions. If every step finishes within time Δ, then a process decides within time $O(\Delta \cdot fac^{-1}(\Delta))$ irrespective of the failures of other processes, where fac^{-1} is the inverse of the factorial function. Thus, the algorithm satisfies wait freedom in the unknown-delay model. Furthermore, the algorithm is fast: in absence of contention, a process decides after a constant number of steps without explicitly delaying itself.

Recall that there is no wait-free algorithm for consensus in the asynchronous model. In the known-delay model, a process can use its knowledge about the speeds of other processes by executing the statement $delay(\Delta)$, and it is possible to design a wait-free solution [AT96]. Let us see how such an algorithm can be constructed when the upper bound is not known.

Initially, each process starts with some estimate, say 1, for Δ. The algorithm proceeds in rounds. Each process has a preference for the decision value in each round; initially this preference is the input value of the process. In each round r, processes execute a timing based consensus algorithm with their current estimate of Δ, using their preferences for this round as inputs.[4] The algorithm guarantees that once processes have the same preference in some round, they will remain in agreement and will eventually decide. The timing based algorithm used in each round avoids conflicting decisions even if the current estimate for Δ is wrong. If no decision is made in a round then the processes advance to the next round, using a larger estimate for the time bound Δ. Eventually, processes either decide, or they end up using the correct estimate, in which case, the timing based algorithm guarantees that they will decide.

The code for the algorithm appears in Figure 1. The algorithm uses the following shared data structures: an infinite array $x[*, 0..1]$ of bits, and an infinite array $y[*]$; the possible values of each $y[i]$ are $\{\perp, 0, 1\}$. The decision value is written to the shared bit *out*. We only use atomic reads and writes to the shared registers. In addition, each process p_i has a local register v_i, containing its current preference and a local register r_i, containing its current round number. The estimate d_r used in round r is $r!$.

In round r, process p_i first flags its preference v by writing 1 to $x[r, v]$. Then, the process checks the lock on this round by reading $y[r]$, and writes its preference to $y[r]$, if $y[r]$ has still its initial value $\perp$. Process p_i then reads the flag for the other preference (denoted by $\bar{v}$). If $x[r, \bar{v}]$ is not set, then every process that reaches round r with the conflicting preference $\bar{v}$ will find $y[r]$ set to v. Consequently, process p_i can safely decide on v, and it writes the decision value to *out*. Otherwise, it waits for d_r (the estimate of Δ for the current round), and then sets its preference for the next round by reading $y[r]$.

Two processes with conflicting preferences for round r will not resolve the conflict only if both of them find $y[r] = \perp$ first, and one of them proceeds and chooses its preference for the next round before the other one finishes the assignment to $y[r]$. However, if each process is required to finish the assignment within time Δ, and the value of d_r exceeds Δ, then this cannot happen. Also notice that if all processes in a round have the same preference, then a decision is reached in that round. These observations, together with the fact that the sequence $d_1, d_2, \ldots$ increases without a bound, ensure termination. The next section includes a complete proof of correctness for this algorithm.

3.2 Correctness

We now present the correctness proof of the algorithm. We assume that a process keeps taking idling steps after it has decided. Thus an infinite execution contains infinitely many steps by every nonfaulty process.

Lemma 2 *If process p_i decides on a value v then $in_j = v$ for some process p_j.*

Proof: If there are two processes that have different inputs then the lemma holds trivially. Suppose all processes start with the same input in. Consider the following formula ϕ:

$$\forall i.\, v_i = in \ \wedge\ \forall r.\, y[r] \in \{\perp, in\} \ \wedge\ out \in \{\perp, in\}\ .$$

Initially ϕ holds. It is easy to check that each transition of the algorithm preserves ϕ. Thus ϕ is an invariant of the algorithm. The lemma follows immediately. ■

[4] The idea of using preferences for consensus was used previously, e.g., in [AH90].

```
Shared registers: initially: out =⊥, y[*] =⊥, x[*,*] = 0.
Local registers: initially: r_i = 1, v_i = in_i.
Constants: d_r = r! for all r.

1 while out =⊥ do
2     x[r_i, v_i] := 1;
3     if y[r_i] =⊥ then y[r_i] := v_i fi;
4     if x[r_i, v̄_i] = 0 then out := v_i
5                 else delay(d_{r_i});
6                      v_i := y[r_i];
7                      r_i := r_i + 1 fi
8 od;
9 decide(out).
```

Figure 1: Time-Adaptive Consensus – program for process i with input in_i.

Let $r \geq 1$ and $v \in \{0, 1\}$. Formally, a process p_i *reaches* round r, if it executes Statement 2 (see Figure 1) with $r_i = r$. A process p_i *prefers* the value v in round r, if $v_i = v$ when p_i reaches round r. A process p_i *commits* to the value v in round r, if it executes the assignment $out := v$ with $r_i = r$.

Lemma 3 *If all processes reaching round r have the same preference v for round r, then all nonfaulty processes reaching round r commit to v in round r.*

Proof: Suppose all processes reaching round r have the same preference v for round r. Thus, whenever some process p_i sets the bit $x[r, v_i]$ to 1, v_i equals v. Consequently, $x[r, \bar{v}] = 0$ is an invariant. Now consider a process p reaching round r. Assuming that p continues to take steps in round r, p will find $x[r, \bar{v}]$ unset at Statement 4, and commit to the value v. ■

Lemma 4 *If some process commits to v in round r then all processes reaching round $r+1$ prefer v in round $r+1$.*

Proof: Suppose some process p commits to v in round r. Since p finds $x[r, \bar{v}]$ unset at Statement 4, it follows that every process with preference $\bar{v}$ for round r, finds $y[r] \neq \perp$ at Statement 3. This implies that for a committed value v, $y[r] \neq \bar{v}$ is an invariant of the program. Since a process decides on its preference for round $r+1$ by reading $y[r]$, the lemma follows. ■

Lemma 5 *No two processes decide on conflicting values.*

Proof: Suppose two processes decide on conflicting values. This means that there exist nonfaulty processes p_0 and p_1 such that p_0 commits to 0 in round r and p_1 commits to 1 in round r'. We will obtain a contradiction.

First suppose that $r \neq r'$. Without loss of generality, let $r < r'$. Since p_0 commits to 0 in round r, from Lemma 4 all processes reaching round $r+1$ prefer 0 in round $r+1$, and consequently, from Lemma 3, if nonfaulty, commit to 0 in round $r+1$. Since p_1 reaches round $r+1$, and is nonfaulty, it follows that p_1 commits to 0 in round $r+1$; a contradiction.

Now suppose that $r = r'$. In round r, process p_0 prefers 0, and process p_1 prefers 1. If process p_0 finds $x[r,1]$ unset at Statement 4, then process p_1 must find $x[r,0]$ set at Statement 4, and vice versa. Consequently, it is not possible that both commit in round r. ■

The proof of termination relies only on the fact that the sequence of delays, $d_1, d_2, \ldots$ is unbounded.

The termination is guaranteed by the following lemma.

Lemma 6 *In a Δ-admissible execution, if $d_r \geq \Delta$ then all processes reaching round $r+1$ have the same preference in round $r+1$.*

Proof: Assume $d_r \geq \Delta$. Consider a Δ-admissible execution α, and a Δ-admissible time assignment τ for it. Let k be the smallest index such that the event e_k is the assignment $v_i := y[r]$ (at Statement 6) by some process p_i that reaches round $r+1$. Let the event e_l correspond to the delay statement $delay(d_r)$ (at Statement 5) by process p_i. We know that $\tau_k - \tau_l > d_r$, and hence, $\tau_k - \tau_l > \Delta$.

Process p_i, before it reaches the delay statement, either finds $y[r] \neq \perp$, or assigns its preference for round r to $y[r]$. Hence, in states s_m, for $m \geq l$, $y[r] \neq \perp$. Let $y[r] = v$ in state s_k. We want to prove that $y[r] = v$ in all states s_m for $m \geq k$. Suppose not. Let p_j, $j \neq i$, be a process that writes to $y[r]$ (at Statement 3) at step $k' > k$. Let $e_{l'}$ be the event that p_j tests the condition $y[r] = \perp$ (at Statement 3). Since $y[r] \neq \perp$ in all states s_m for $m \geq l$, we have $l' < l$. This implies $\tau_{k'} - \tau_{l'} \geq \tau_k - \tau_l > \Delta$. Since l' and k' are consecutive steps of p_j, this contradicts Δ-admissibility of τ. Thus $y[r] = v$ in all states s_m for $m \geq k$.

Since every process reaching round $r+1$ chooses its preference for round $r+1$ by reading $y[r]$ at some step $m \geq k$, the lemma follows. ■

If $d_r \geq \Delta$ then Lemma 6 and Lemma 3 imply that in a Δ-admissible execution, no process can reach round $r+2$, and every nonfaulty process decides in round $r+1$ or lower. If the sequence $d_1, d_2, \ldots$ is unbounded, then for every Δ, there is some r such that $d_r \geq \Delta$. Consequently, we get termination in each Δ-admissible execution. This implies the following theorem.

Theorem 1 (Correctness) *Algorithm of Figure 1 is a correct solution to wait-free consensus in the unknown-delay model.*

3.3 Time Complexity

Now let us analyze the time complexity of the algorithm. Recall that the worst-case time complexity of the algorithm is the maximum time after which a nonfaulty process decides.

Formally, given an execution α and a time assignment τ, let $time(\alpha, \tau)$ be the maximum difference $\tau_k - \tau_l$ such that both the events e_l and e_k are nonidling steps of the same process (recall that a process takes idling steps only after it has decided). The worst-case time complexity of the algorithm A when the upper bound is Δ, is then $time_\Delta(A)$ as defined in Section 2.

Lemma 7 *Let R be the smallest index such that $d_R \geq \Delta$. Then, the worst-case time complexity $time_\Delta(A)$ is at most $9(R+1)\Delta + d_R$.*

Proof: Let $d_R \geq \Delta$ and $d_{R-1} < \Delta$. Consider an execution α and a Δ-admissible time assignment τ. Let p be a process. Since $d_R \geq \Delta$, from Lemma 6 and Lemma 3, either p fails, or p decides in round $R+1$ or lower. For the worst case analysis, we assume that p decides in round $R+1$. For each round r, suppose p enters round r at the i_r-th step in α.

Let e_k be the last nonidling step of process p. The time taken by p is $\tau_k - \tau_{i_1}$. In each round r, p executes only a constant, at most 8, number of steps, possibly including a delay statement. By Δ-admissibility, we have $\tau_{i_{r+1}} - \tau_{i_r} \leq 8\Delta + d_r$, for any round r. For $r < R$, $d_r < \Delta$. Hence, $\tau_{i_{R+1}} - \tau_{i_1} \leq 9R\Delta + d_R$. In round $R+1$, p_i takes only a constant, at most 7, number of steps without executing the delay statement. Hence, $\tau_k - \tau_{i_{R+1}} \leq 7\Delta$. Hence, $\tau_k - \tau_{i_1} \leq 9(R+1)\Delta + d_R$. The lemma follows. ■

The time complexity of the algorithm depends on the choice of the sequence d_r. If the sequence is fast growing then the value of R will be small. But if the sequence grows too fast, then the value of d_R can be much larger than Δ itself. For instance, if we let $d_r = r$, then $R = \Delta = d_R$, and the time complexity is $O(\Delta^2)$. For $d_r = 2^r$, $R = O(\log \Delta)$, $d_R \leq 2{\cdot}\Delta$, and the time complexity is $O(\Delta{\cdot}\log\Delta)$. For $d_r = 2^{2^r}$, $R = O(\log\log\Delta)$, but $d_R \leq \Delta^2$, giving time complexity $O(\Delta^2)$. As we shall see (as part of the lower bound proof in the next section), the best sequence is $d_r = r!$.

Let fac^{-1} be the inverse of the factorial function, that is, $fac^{-1}(d)$ is the smallest integer r such that $r! \geq d$, for any real $d > 0$; note that $fac^{-1}(d) = \Theta(\frac{\log d}{\log\log d})$. In the case where $d_r = r!$ for all r, $R = fac^{-1}(\Delta)$, and $d_R = (fac^{-1}(\Delta))!$. Hence, $d_R \leq \Delta{\cdot}fac^{-1}(\Delta)$. This gives the overall complexity of $O(\Delta{\cdot}fac^{-1}(\Delta))$.

Theorem 2 (Time Complexity) *For the algorithm of Figure 1 with the sequence of delays $d_r = r!$, for every Δ, the worst-case time complexity $time_\Delta(A)$ is bounded by $10{\cdot}\Delta{\cdot}(fac^{-1}(\Delta)+1)$.*

Note that our algorithm uses unbounded space. In a Δ-admissible execution only the first $(fac^{-1}(\Delta)+1)$ elements of the arrays x and y are used. Since $fac^{-1}(\Delta)$ is small for any reasonable value of Δ, space is not a real problem. For instance, if our time unit is a second and the upper bound Δ is 1000 years, then $fac^{-1}(\Delta)$ is 14.

Finally let us consider the contention-free complexity of the algorithm. Informally, we want the contention-free complexity to indicate the time taken by a process when it executes by itself without interference from other processes. Formally, given an execution α and a time assignment τ, let $cf\text{-}time(\tau, \alpha)$ be the maximum difference $\tau_j - \tau_i$ such that both the events e_j and e_i are nonidling events of the same process p, and every other process q either has decided before step i or has not taken any step before step j.

The contention-free time complexity of the algorithm A for the upper bound Δ is then given by $cf\text{-}time_\Delta(A)$ and $min\text{-}cf\text{-}time_\Delta(A)$. Our algorithm has low contention-free complexity:

Theorem 3 (Fast decision in absence of contention) *For the algorithm of Figure 1, for every Δ, cf-time$_\Delta(A) = 7\cdot\Delta$, and min-cf-time$_\Delta(A) = 0$.*

Proof: Consider an execution α and indices i and j such that the event e_i is the first event of the process p, the event e_j is the decision event of p, and every other process q either has decided before step i or has not taken any step before step j. There are two cases to consider.

If some process q has decided before the process p starts, then the value of *out* is different from $\perp$ in state s_i, and p takes at most 2 steps before deciding, both of which have zero explicit delay. This implies that if τ is a Δ-admissible assignment for α then $\tau_j - \tau_i \leq 2\Delta$. Furthermore, since p does not execute any delay statement, the difference $\tau_j - \tau_i$ can be made as small as possible, implying that the infimum of $\tau_j - \tau_i$ over all Δ-admissible time assignments for α is 0.

If no process q has taken a step before event e_i, then p is the first process to start, and no process starts before p decides. In this case, p decides in the first round after taking at most 7 steps, again without executing any delay statement. In this case, the maximum of $\tau_j - \tau_i$ over all Δ-admissible time assignments for α is 7Δ, and the infimum is 0. ■

We point out that if some processes fail without deciding before a process p starts, then even if p runs by itself, it may execute for $O(\Delta\cdot fac^{-1}(\Delta))$ time. Thus failures of processes can lead to the worst-case time complexity.

3.4 Lower Bound on Time Complexity

For the algorithm of Section 3, if Δ is the upper bound on step-time, then a process decides within time $O(\Delta\cdot fac^{-1}(\Delta))$. If a process knew the value of Δ in advance, then it can execute the algorithm with $d_1 = \Delta$, ensuring termination in the second round, within time $O(\Delta)$. Thus, the lack of knowledge of the value of Δ multiplies the time complexity by a factor of $fac^{-1}(\Delta)$. In this section we prove this increase in cost to be inherent: we prove that any algorithm for solving 2-process consensus in the unknown-delay model has worst-case time complexity of $O(\Delta\cdot fac^{-1}(\Delta))$.

For proving the lower bound we restrict our attention to a system with two processes, p_1 and p_2. Let A be an algorithm for wait-free consensus in the unknown-delay model. Consider an execution $\alpha = s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots$. The execution can be partitioned into *blocks*, each containing a sequence of events by the same process. Formally, let $b_0, b_1, \ldots$ be an increasing sequence of integers with $b_0 = 0$, such that for each i, all the steps indexed from b_i to $b_{i+1} - 1$ are of the same process, and the step indexed b_{i+1} is of a different process. Thus the i'th block is the execution fragment $s_{b_i} \stackrel{e_{b_i}}{\rightarrow} s_{b_i+1} \cdots \stackrel{e_{b_{i+1}-1}}{\rightarrow} s_{b_{i+1}}$ consisting of steps of a single process.

By Lemma 1, A must guaranty safety also in the asynchronous shared-memory model. Therefore, from the proof of the impossibility of solving consensus in the shared memory asynchronous model (e.g., [LA87, Theorem 4.1]) we can deduce:

Lemma 8 *There exists an infinite sequence of executions $\alpha_0, \alpha_1, \ldots$ such that for all $k \geq 0$, (1) α_k is a finite execution with $k+1$ blocks, (2) α_{k+1} is an extension of α_k, and (3) no process has decided at the end of α_k.*

Let us recall the definition of time complexity of consensus. Given an execution α of A and a Δ-admissible time assignment τ for α, whenever e_i and e_j are the (nonidling) steps of the same process, we have $time_\Delta(A) \geq \tau_j - \tau_i$. Now we consider another definition needed for the proof. Given an execution α, define d_i to be the total sum of the delays in delay statements appearing in the ith block. With each execution we can associate a sequence $d_0, d_1, \ldots$, called the *sequence of block delays*.

Lemma 9 *Let α be a finite execution with $k+1$ blocks with $k \geq 1$ such that no process has decided at the end of α. Let $d_0, d_1, \ldots, d_k$ be the associated sequence of block delays, and let $\Delta \geq 1 + \sum_{i=0}^{k-1} d_i$. Then $time_{2\Delta}(A) \geq k \cdot \Delta + d_k$.*

Proof: Let α be $s_0 \xrightarrow{e_0} \cdots s_{n-1} \xrightarrow{e_n} s_n$ consisting of $k+1$ blocks starting at indices $b_0, \ldots b_k$. Without loss of generality assume that the last block corresponds to steps by process p_1.

Since no process has decided, and A satisfies wait freedom, we know that p_1 can take an additional step, e_{n+1}; let α' denote the extended execution. Now we construct a time assignment τ for α' as follows. Let $\Delta \geq 1 + \sum_{i=0}^{k-1} d_i$.

We want each block, except the last one, to take Δ time. For every $0 \leq j \leq n$,

- if e_j is the first step of the i-th block (i.e., $j = b_i$), for $i = 0, \ldots k$, then let τ_j be $i \cdot \Delta$; else
- if e_j is the last step of the i-th block (i.e., $j = b_{i+1} - 1$), for $i = 0, \ldots (k-1)$, then let τ_j be $(i+1) \cdot \Delta$; else
- let τ_j be $\tau_{j-1} + d(e_{j-1}) + 1/n$.

For each block i, the sum of the explicit delays of events in the block i is d_i, and the number of events in the block i is bounded by n. By the choice of Δ, it is clear that the sequence of values defined above is nondecreasing. Furthermore, let the time τ_{n+1} of the last step be $\tau_n + d(e_n) + \Delta$.

Now consider events e_j and $e_{j'}$ that are consecutive steps of the same process. There are two cases to consider.

1. Both e_j and $e_{j'}$ belong to the same block (i.e., $j' = j+1$). Then $\tau_{j'} - \tau_j > d(e_j)$. In each block, except possibly the last one, the difference between the time of the last step and the first step is Δ. In the last block, the time of the first step is $k \cdot \Delta$, and the times of the remaining steps are increased only when p_1 executes delay statements. Hence, $\tau_{j'} - \tau_j \leq \Delta + d(e_j)$.
2. The event e_j is the last event of a block, say i-th block, and $e_{j'}$ is the first event of the $(i+2)$-th block. In this case, $\tau_{j'} = (i+2) \cdot \Delta$. If the i-th block has only one event then $\tau_j = i \cdot \Delta$ else $\tau_j = (i+1) \cdot \Delta$. Thus, $\Delta \leq \tau_{j'} - \tau_j \leq 2\Delta$.

This implies that τ is a legal time assignment for α, and furthermore, it is 2Δ-admissible (i.e., we can choose 2Δ as the upper bound). Note that the total delay in the last block may be larger than 2Δ. Since only p_1 takes steps in the last block, this means that p_2 has failed if we put an upper bound of 2Δ on the step-times.

Finally, observe that the time of the last step τ_{n+1} is $(k+1)\cdot\Delta + d_k$. Time of the first step by p_1 is either 0 or Δ depending on whether the first or the second block corresponds to steps by p_1. This means that p_1 has executed for at least $k\cdot\Delta + d_k$ time without deciding. Hence, $time_{2\Delta}(A) \geq k\cdot\Delta + d_k$. ■

Lemma 10 *There exists an infinite nondecreasing sequence of values $\Delta_1, \Delta_2 \ldots$ such that for all $k \geq 1$, for all $\Delta \geq \Delta_k$, $2time_\Delta(A) \geq k\cdot\Delta + (\Delta_{k+1} - \Delta_k)$.*

Proof: Consider the infinite sequence of executions $\alpha_0, \alpha_1, \ldots$ of Lemma 8. Note that the lemma implies that α_{k+1} is obtained from α_k by adding one block. Let d_k be the delay of the last block of α_k. Let $\Delta_k = 2(1 + \sum_{i=0}^{k-1} d_i)$. For each $k \geq 1$, by applying Lemma 9 to α_k, we get that for all $\Delta \geq \Delta_k$, $time_\Delta(A) \geq k\cdot\Delta/2 + d_k$. The lemma follows since $d_k = (\Delta_{k+1} - \Delta_k)/2$. ■

To complete the proof of the lower bound, we show the following technical lemma:

Lemma 11 *For any nondecreasing sequence $\Delta_1, \Delta_2, \ldots$ of real numbers, for infinitely many indices k, $k\cdot\Delta_k + (\Delta_{k+1} - \Delta_k) \geq \Delta_k\cdot fac^{-1}(\Delta_k)$.*

Proof: The proof is by contradiction. Suppose there exists a nondecreasing sequence $\Delta_1, \Delta_2, \ldots$, and an index i, such that for all $k \geq i$, $k\cdot\Delta_k + (\Delta_{k+1} - \Delta_k) < \Delta_k\cdot fac^{-1}(\Delta_k)$. Define $m_k = fac^{-1}(\Delta_k)$. By definition, for all k,

$$(m_k - 1)! \;<\; \Delta_k \;\leq\; m_k!$$

The proof follows the following steps:

(1) We have, for all $k \geq i$, $k\cdot\Delta_k + (\Delta_{k+1} - \Delta_k) < \Delta_k\cdot m_k$. Also $\Delta_{k+1} - \Delta_k \geq 0$ for all k. Hence, $m_k > k$ for all $k \geq i$.

(2) Let $k \geq i$. We have

$$\Delta_{k+1} \;<\; (m_k - k + 1)\cdot\Delta_k \;\leq\; (m_k - k + 1)\cdot m_k! \;<\; (m_k + 1)!$$

Hence, $m_{k+1} \leq m_k + 1$. Thus the sequence of values $(m_k - k)$, for $k \geq i$, is nonincreasing. From (1), $(m_k - k)$ is positive for all $k \geq i$. That is, $(m_k - k)$, $k \geq i$, forms a nonincreasing infinite sequence of positive numbers. Hence, there exists a positive integer a and an index j such that $m_k - k = a$ for all $k \geq j$.

(3) Choose $k \geq j$ such that $k > (a+1)^2$. We have $\Delta_{k+1} < (a+1)\cdot\Delta_k$, and $\Delta_{k+2} < (a+1)\cdot\Delta_{k+1}$. Hence,

$$\Delta_{k+2} \;<\; (a+1)^2\cdot\Delta_k \;\leq\; (a+1)^2\cdot m_k! \;<\; k\cdot(a+k)! \;<\; (a+k+1)!$$

This implies $m_{k+2} \leq a + k + 1$. This contradicts the assertion $m_{k+2} = a + k + 2$ of (2), which completes the proof. ■

Lemma 11, together with Lemma 10, implies:

Theorem 4 (Lower Bound on Time Complexity) *For any algorithm A for solving 2-process wait-free consensus in the unknown-delay model, for every real number d, there exists $\Delta > d$ such that the worst-case time complexity $time_\Delta(A)$ is at least $(\Delta\cdot fac^{-1}(\Delta))/2$.*

Proof: Lemma 10 gives a nondecreasing sequence $\Delta_1, \Delta_2 \ldots$ such that for all $k \geq 1$, for all $\Delta \geq \Delta_k$, $2time_\Delta(A) \geq k \cdot \Delta + (\Delta_{k+1} - \Delta_k)$. There are two cases to consider:

1. The sequence $\Delta_1, \Delta_2 \ldots$ is unbounded: For every real value d, there is an index i such that for all $k \geq i$, $\Delta_k > d$. Using Lemma 11, there is an index $k \geq i$ such that $\Delta_k > d$ and $k \cdot \Delta_k + (\Delta_{k+1} - \Delta_k) \geq \Delta_k \cdot fac^{-1}(\Delta_k)$, and hence, $2time_{\Delta_k}(A) \geq \Delta_k \cdot fac^{-1}(\Delta_k)$.
2. The sequence $\Delta_1, \Delta_2 \ldots$ is bounded: There is a value d^* such that $\Delta_k < d^*$ for all k. Let $\Delta \geq d^*$. Then, by Lemma 10, for all k, $2time_\Delta(A) \geq k \cdot \Delta$. Choosing $k > fac^{-1}(\Delta)$ gives $2time_\Delta(A) > \Delta \cdot fac^{-1}(\Delta)$.

■

Notice that our lower bound implies that for every algorithm, there is an unbounded sequence of values Δ for which the worst-case time complexity is at least $(\Delta \cdot fac^{-1}(\Delta))/2$. This does not rule out the existence of a (different) unbounded sequence of values Δ for which the worst-case time complexity is less than $(\Delta \cdot fac^{-1}(\Delta))/2$. For the algorithm of Figure 1, if we choose the sequence $d_r = 2^{2^r}$, then setting $\Delta = d_r$ implies termination in $O(\log\log r)$ rounds, giving worst-case time complexity of $O(\Delta \cdot \log\log \Delta)$, however, if we set $\Delta = d_r + 1$, the worst-case time complexity is $O(2^\Delta)$.

4 Time-Adaptive Mutual Exclusion

The mutual exclusion problem is to design a protocol that guarantees mutually exclusive access to a critical section among a number of competing processes [Dij65]. A solution to the problem should satisfy the following two properties,

- *Mutual exclusion:* No two processes are in their critical section at the same time.
- *Deadlock freedom:* If some process p starts executing its algorithm, then eventually some process (possibly different from p) is in its critical section.

We assume that each of the potentially n contending processes has a unique identifier taken from the set $\{1, ..., n\}$. While deadlock-freedom is essential, starvation-freedom – any process that is trying to enter its critical section, eventually enters its critical section – is less important in systems where contention is rare. When contention is rare it is important to design algorithms satisfying the fast access property,

- *Fast access:* In absence of contention a process executes only a constant number of steps in order to enter its critical section, and only a constant number of steps to execute the exit code.

In [Lam87], Lamport has presented a mutual exclusion algorithm that satisfies the fast access property. However, in the presence of contention, however small, the winning process may have to check the status of all other n processes before it is allowed to enter its critical section. Alur and Taubenfeld overcome this limitation in the known-delay model [AT96] (see also [AT93]). Their algorithm satisfies the fast access property, and furthermore, in the presence of contention, a process does not have to check the status of all other n processes before it can enter its critical section, but may need to delay itself for $2 \cdot \Delta$ time units. In the next section, we describe an algorithm with similar properties for the unknown-delay model. Other algorithms which satisfy the fast access property can be found in [CS93, MS93, Sty92, YA93].

4.1 The Algorithm

We now present a fast mutual exclusion algorithm for the unknown-delay model. Since in this model a time bound on the speed exists but is not known, the processes keep an estimate of this time (stored in a shared register), and update it when it is noticed that the estimate is not accurate. An entry to the critical section which involves an update (of the estimate) is going to be much slower than an entry without an update. However, the algorithm has the property that at most Δ updates are necessary. As we show the algorithm is also time-efficient when there is contention. As is usually assumed when designing mutual exclusion algorithm, we also assume that process failures do not occur.

The precise code for the algorithm is given in Figure 2. Notice that the statement **await** *condition* is an abbreviation for **while** $\neg$*condition* **do** *skip* (and hence, may involve many accesses to the shared memory). The algorithm is composed of two basic algorithms. The first is Alur and Taubenfeld's algorithm (abbv. AT) for fast mutual exclusion using a timing assumption [AT96]. Statements 1–10 are the entry code of AT and Statements 29–31 are its exit code. We point out that in the original AT algorithm the register *bound* is initially set to Δ (which is assumed to be known). Furthermore, while the AT algorithm satisfies mutual exclusion only when *bound* $\geq \Delta$, the proof of deadlock freedom does not depend on the value of *bound*. We will exploit this property in our construction. The critical section of AT is now replaced by Lamport's fast mutual exclusion algorithm: Statements 11–28. These two algorithms are combined, together with a mechanism for estimating and updating the current bound. All references to the register *update* and the array *trying* belong to this mechanism, and are not part of the original AT and Lamport's algorithms.

Intuitively, the algorithm works as follows. First, each process executes the AT algorithm, using the current estimate *bound*. If the estimate is correct, or if there is no contention, only one process will proceed to the next stage (i.e., get to Statement 11). However, it is possible that the current estimate used by the processes is incorrect. In this case, more than one process may proceed to the next stage, and therefore, to guarantee mutual exclusion, we embed at this point Lamport's fast algorithm. If a process discovers contention while executing Lamport algorithm, it "knows" that the current estimate used is incorrect, and has to be increased. (Contention is discovered when either of conditions in Statements 14 and 18 evaluates *true*.)

To avoid complications only a process that enters its critical section (Statement 23) is allowed to update the register *bound*. This guarantees that no two processes try to update the estimate at the same time, and thus the value of *bound* never decreases.

The update is done as follows: First the process sets *update* to 1, signaling that it wants to make an update (Statement 22). Then, it waits until each active process returns to the beginning of its trying code and waits for *update* to become 0 (Statement 2). It is easy to check that once *update* is 1, eventually every process will test it. Once process i finds that *update* is 1, it returns to the beginning of its code, signals to the updating process that it is at the beginning by setting $trying[i]$ to 0, and waits (Statement 2). Once the updating process

Initially: $y = 0$, $yy = 0$, $z = 0$, $bound = 1$, $update = 0$, $trying[i] = 0$ and $b[i] = 0$ for all i.

```
start1:repeat
           if update = 1 then trying[i] := 0; await update = 0 fi;
           trying[i] := 1;
           x := i;
        until (y = 0);
        y := i;
        if x ≠ i then delay(2 · bound);
                      if y ≠ i then goto start1 fi;
                      await (z = 0) or (update = 1)
        else z := 1;

start2:if update = 1 then goto start1 fi;
        b[i] := 1;
        xx := i;
        if yy ≠ 0 then b[i] := 0;
                      await (yy = 0) or (update = 1);
                      goto start2 fi;
        yy := i;
        if xx ≠ i then b[i] := 0;
                      for j := 1 to n do await (b[j] = 0) or (update = 1) od;
                      if yy ≠ i then await (yy = 0) or (update = 1);
                                     goto start2
                      else update := 1 fi fi;                (* set lock *)
        critical section;
        trying[i] := 0;
        if update = 1 then for j := 1 to n do await trying[j] = 0 od;    (* wait *)
                      bound := bound + 1 fi;                 (* increment bound *)
        yy := 0;
        b[i] := 0;

        z := 0;
        if update = 1 then y := 0; update := 0
                      else if  y = i then y := 0 fi fi
```

Figure 2: Fast timing-adaptive algorithm – process i's program.

gets acknowledgements from all active processes, it safely increments *bound* (Statement 26), executes the exit code of both the algorithms, and releases the lock (Statement 30), which leaves the system in its initial configuration (except for the value of *bound*).

The fact that the processes return to the beginning of their code before *bound* is incremented, guarantees that the value of the *bound* will never be greater than Δ.

Once *bound* equals Δ the entry code of AT (Statements 1–10), ensures that no two processes execute Lamport algorithm (Statements 11–28) at the same time. Hence, from that point on, processes will always enter the their critical section along the fast path of Lamport's algorithm.

In summary, each process starts by checking if an update of *bound* is taking place, in which case it waits until the update is finished. Then the process performs the entry code for the timing-based mutual exclusion algorithm using the current estimate. If it gains access to the critical section (of AT), the process executes Lamport's fast mutual exclusion algorithm. However, in the algorithm, if a process enters its critical section via the slow path, it "knows" that the current estimate in use is incorrect, and should be increased. It does so, by first signaling other processes to go to the beginning of their code and, after they all do so, it increments the register *bound*.

4.2 Correctness

The design of the algorithm and its correctness proof are based on the following straightforward general observation.

Lemma 12 *Let A and B be mutual exclusion algorithms (with disjoint sets of shared registers), and let C be the algorithm obtained by replacing the critical section of A with the algorithm B.*[5]

1. *If both A and B are deadlock-free then C is deadlock-free.*
2. *If either A or B satisfies mutual exclusion then C satisfies mutual exclusion.*

Proof: The entry code of C is composed from the entry code of A, denoted by C_A, followed by that of B, denoted by C_B. Assume that both A and B are deadlock-free. If some process starts executing algorithm C, then, since A is deadlock-free, eventually some process will finish C_A and will proceed to C_B. Since B is deadlock-free, eventually some process will finish C_B and will enter its critical section. Thus, C is deadlock-free.

If A satisfies mutual exclusion, then no two processes can be at their C_B code at the same time. If B satisfies mutual exclusion, then no two processes can finish their C_B code at the same time. In either case, it implies that no two processes are in their critical section at the same time. ■

The correctness of the algorithm is based on Lemma 12 and the properties of Lamport and AT algorithms. Note that the algorithm satisfies the correctness requirements also in the asynchronous model.

As already explained, the algorithm is obtained by replacing the critical section of Alur and Taubenfeld's algorithm with Lamport's algorithm (Statements 11–28). These

[5] If the critical section of A has a label, then in C this label is associated with the first statement of B.

two algorithms are combined, together with a mechanism for estimating and updating the current time bound. All references to the register *update* and the array *trying* belong to this mechanism, and are not part of the original AT and Lamport's algorithms.

It is known that Lamport's algorithm satisfies mutual exclusion and deadlock-freedom, and that AT algorithm satisfies deadlock-freedom regardless of the value of *bound*.

Theorem 5 *The algorithm of Figure 2 satisfies deadlock-freedom in the asynchronous model.*

Proof: As long as the value of *update* is 0, executing Statements 1–10 or Statements 11–22, is the same as executing the entry code of AT algorithm or the entry code of Lamport's algorithm, respectively. Since both AT and Lamport's algorithms are deadlock-free (regardless of the value of *bound*), Lemma 12 implies that the algorithm cannot be deadlocked while the value of *update* is continuously 0.

We observe that if the value of *update* is 1, then there must be some process (called the *winner*) which is either in its critical section or in its exit code. Once *update* is 1, eventually every process (other than the winner) will test it. Once process i finds that *update* is 1, it returns to the beginning of its code, and signals to the winner that it is at the beginning by setting $trying[i]$ to 0. Thus, eventually the winner gets acknowledgements from all active processes, which implies that the winner cannot be blocked forever in the *for* loop of Statement 25, and will eventually set *update* back to 0. This implies that the system cannot be deadlocked, while the value of *update* is continuously 1.

Thus, a deadlock can occur in an infinite execution only if *update* changes values infinite number of times. However, each time *update* changes its value, some process either enters or exits its critical section. Therefore, in an execution where *update* changes values infinite number of times, no deadlock can occur. ■

Theorem 6 *The algorithm of Figure 2 satisfies mutual exclusion in the asynchronous model.*

Proof: As long as the value of *update* is 0, executing Statements 11–28, is the same as executing Lamport's algorithm. Since Lamport's algorithm satisfies mutual exclusion, by Lemma 12, the new algorithm must also satisfy mutual exclusion when the value of *update* is 0.

If the value of *update* is 1, then there must be some process (the *winner*) which is either in its critical section or in its exit code.

If some process has tested *update* before it was set to 1, then the value of *update* was 0 at this time, and (as already explained above) since Lamport's algorithm satisfies mutual exclusion this process will not enter its critical section, and eventually it will have to test *update* again.

Once the winner sets update to 1, no other process can enter the critical section until *update* is set back to 0. To see that, observe that once *update* is 1, eventually every process will test it, and once a process finds that *update* is 1, it returns to the beginning of its code, signals to the updating process that it is at the beginning by setting $trying[i]$ to 0, and waits (Statement 2) until *update* is set to 0. The winner sets *update* to 0 in its exit code, only after it gets acknowledgements from all active processes. Because the processes retreat to *start1* at the beginning of their code before *update* is reset, no process executes Lamport's (embedded) algorithm, which in turn guarantees that no two processes can enter their critical section as long as the value of *update* is not changed. ■

4.3 Time Complexity

Next we show that the register *bound* is updated at most Δ times.

Lemma 13 *bound* $\leq \Delta$ *is an invariant of the algorithm.*

Proof: Once *bound* reaches the correct Δ, the delay in Statement 7 is $2{\cdot}\Delta$. At this point all the processes that participate in the algorithm, except the one that is updating the value of *bound*, are at the beginning of their code, waiting for *update* to become 0. Thus, from that point on, this code (Statements 1–10) behaves exactly as the original AT algorithm.

This means that from now on only one process can be in Lamport's algorithm (Statements 11–28). When only one process is in Lamport's algorithm, the test in line 18 is always evaluated to false, and hence statement 22 will not be reached, the value of *update* will remain at 0, and no more updates to *bound* will occur. (Statement 22 is the only place where the value of *update* is changed from 0 to 1.) Thus, the number of times a winning process has to update *bound* after executing its critical section is bounded by Δ. ■

The next theorem shows that the algorithm is time-efficient. In the theorem, the time it takes for a process to enter its critical section is measured from the last time some process exited its critical section.

Theorem 7 *The algorithm has the following properties:*

1. *Fast access: In the absence of contention, a process executes only constant number of steps (13) from the location start1 to its critical section, and only constant number of steps (8) to execute the exit code. No delays are necessary.*

2. *In the presence of contention, a winning process which does not update the register bound executes a constant number of steps (14) and may need to delay itself for at most $2 \cdot \Delta$ time units, before entering its critical section.*

3. *In the presence of contention, a winning process which needs to update the register bound may execute $O(n)$ steps and may need to delay itself for at most $2 \cdot \Delta$ time units, before entering its critical section. This may happen at most Δ times.*

Proof: The first part is straightforward.

By Lemma 13, *bound* $\leq \Delta$. Thus, executing the delay in Statement 7 takes at most $2{\cdot}\Delta$ time units. Note that a winning process updates *bound*, if and only if it finds the condition in Statement 18 (i.e., $xx \neq i$), to be true. The second part of the theorem is easily verified by counting steps in the algorithm.

Only when a process finds the condition in Statement 18 to be true, it executes the *for* statement at Statement 19, in which it may need to execute $O(n)$ steps. This implies the third part of the theorem, and explains why the term $O(n)$ is added. ■

Observe that there is a tradeoff between the number of updates of the register *bound* and its maximum value. For example, if instead of incrementing it by 1, we double its value when it is updated, then we can show that: *bound* is updated at most $\log \Delta$ times, and $bound \leq 2 \cdot \Delta - 1$. Incrementing by 1 is the best strategy, since it gives the best amortized time complexity when the number of entries to the critical section is much bigger than Δ.

Notice that our algorithm uses $2n+5$ shared registers. It is possible to replace the arrays b and *trying*, each of n bits, with one array of n 3-valued registers. Lynch and Shavit have proved that, when the timing bounds are not known, n is a lower bound on the number of shared registers [LS92]. (The model used therein for algorithm design is the known-delay model, but their lower bound proof continues to hold also for the unknown-delay model.) In contrast, the algorithms for mutual exclusion in the known-delay model use only a constant number of registers [Lam87, AT96, LS92].

In our algorithm the value of the register *bound* can only be increased, and after it is updated Δ times, it will reach its maximum value Δ. In a dynamic system where processes are created and destroyed, the upper bound on the speed of the processes may change over time. Our algorithm adapts to an increase in Δ (that may be caused by adding slow processes). However, when Δ decreases (a slow process is destroyed) the value of *bound* may be too high leading to inefficient utilization. This may be resolved by periodically resetting *bound* to zero, and letting it adjust to reflect the current speed.

5 Discussion

We have defined the unknown-delay model, which formalizes systems in which there is an upper bound on memory access time, but this bound is not known. For the consensus problem, we have shown that $\Theta(\Delta \cdot fac^{-1}(\Delta))$ is an upper and lower bound on the time complexity of any algorithm, where fac^{-1} is the inverse of the factorial function. The algorithm that achieves this bound is fast in the absence of contention. Since consensus is universal [He91], our results imply that atomic reads and writes are universal in the unknown-delay model. For the mutual exclusion problem, we have presented an algorithm in which, in the presence of contention, a process needs only delay itself for $2 \cdot \Delta$ time units before entering the critical section, when no update of the time estimate is needed. This algorithm is also fast in the absence of contention.

The standard definitions of the consensus problem and the mutual exclusion problem differ in two ways. First, a mutual exclusion algorithm is invoked repeatedly, while a consensus algorithm is invoked only once. Second, in the consensus problem processes may fail, while in the mutual exclusion problem it is assumed that processes do not fail. Note however, that our algorithms for both problems are constructed in a similar manner, by combining an asynchronous algorithm that guarantees safety, a timing-based algorithm that converges when used with a correct estimate for Δ, and a mechanism for estimating Δ.

Acknowledgements: We wish to thank Yehuda Afek, Eli Gafni, Eyal Kushilevitz, and Michael Merritt for helpful discussions, and the anonymous referees for a thorough review of the manuscript.

References

[AT96] R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1), 1996.

[AT93] R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proc. of the 5th IEEE Symp. on Parallel and Distributed Processing*, pp. 470–477, 1993.

[AH90] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11 (Sept.):441–461, 1990.

[ADLS91] H. Attiya, C. Dwork, N. Lynch and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.

[CS93] M. Choy and A. Singh. Adaptive solution to the mutual exclusion problem. In *Proc. of the 12th ACM Symp. on Principles of Distributed Computing*, pp. 183–194, 1993.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[He91] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programming Languages and Systems*, 11(1):124–149, 1991.

[HK89] A. Herzberg and S. Kutten. Efficient detection of message forwarding faults. In *Proc. of the 8th ACM Symp. on Principles of Distributed Computing*, pp. 339–353, 1989.

[LA87] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Systems*, 5(1):1–11, 1987.

[LS92] N. Lynch and N. Shavit. Timing-based mutual exclusion. In *Proc. of the 13th IEEE Real-Time Systems Symp*, pp. 2–11, 1992.

[MS93] M. M. Michael and M. Scott. Fast mutual exclusion, even with contention. Technical Report 460, Department of Computer Science, University of Rochester, June 1993.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[Sty92] E. Styer. Improved fast mutual exclusion. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pp. 159–168, 1992.

[YA93] J-H. Yang and J.H. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proc. 12th ACM Symp. on Principles of Distributed Computing*, pp. 171–182, 1993.