An Upper and a Lower Bound for Tick Synchronization

*Marios Mavronicolas**
Aiken Computation Laboratory
Harvard University
Cambridge, MA 02138

Abstract

The tick synchronization problem is defined and studied in the semi-synchronous complete network with n processes. An algorithm for the tick synchronization problem enables each process to make an estimate of real time close enough to those of other processes. It is assumed that the (real) time for message delivery is at most d and the time between any two consecutive steps of any process is in the interval $[c, 1]$, where $0 < c \leq 1$.

We define the precision of a tick synchronization algorithm to be the maximum difference between estimates of real time made by different processes, and propose it as a worst-case performance measure. We show that no such algorithm can guarantee precision less than $\lfloor \frac{d-2}{2c} \rfloor$. We also present an algorithm which achieves a precision of $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$.

1 Introduction

Most existing distributed systems are modeled as a *communication network*—a collection of n processes arranged at the nodes of an undirected graph G and communicating by sending messages across links that correspond to the edges of G. Central to the programming of distributed systems are *synchronization problems,* where processes are required to obtain some common notion of time so as to perform a particular action simultaneously. How closely can they be guaranteed to perform such an action?

Such synchronization problems were first investigated by Lamport in [8], where a simple algorithm was presented allowing a system of asynchronous processes to maintain a discrete clock that remains consistent with the ordering of receipt of communication messages by the processes. Several researchers have considered a so called "partially synchronous" model of a distributed system in which processes have real-time clocks that run at the same rate as real time, but are arbitrarily offset from each other initially. In addition, there are known upper and lower bounds on message delay. The goal has been to prove limits on how closely clocks can be synchronized. In a completely connected network of n processes, Lundelius and Lynch ([9]) show a tight bound of $\eta(1 - \frac{1}{n})$ on how closely the clocks of n processes can be synchronized, where η is the difference between the bounds on the message delay. Their work was subsequently extended by Halpern, Megiddo and Munshi ([7]) to arbitrary networks. There has also been much work done on the problem of devising fault-tolerant algorithms to synchronize real-time clocks that drift slightly in the presence of variable message delay. (A good survey of this work on fault-taulerant clock synchronization, as well as of the general clock synchronization problem, appears in [17].)

In reality, however, each process acquires information about time from a local, inaccurate, *discrete* clock component that is available to it and operates at the rate at which it receives ticks from its local clock. Such information is necessarily imprecise since the time between successive ticks of any clock is not known exactly, but only within certain bounds. We model these *semi-synchronous* systems by assuming that there is an upper and a lower bound on the time between successive clock ticks that enable processes to estimate time. Such modelling was first introduced in [2], and subsequently received a lot of attention (see, e.g., [11, 3, 1, 15, 12, 16]).

We address the problem of achieving coordinated action in semi-synchronous networks by studying the *tick synchronization problem*, which is the problem of achieving as close as possible time estimates by different processes in a semi-synchronous network.

The tick synchronization problem is an abstraction of the synchronization needed for the execution of some tasks that arise in a distributed system, where

*Supported by ONR contract N00014-91-J-1981. Current address: Department of Computer Science, University of Crete, Iraklion, Crete, Greece.

separate components need to agree on as a close as possible common value of real time. Consider, for example, version management and concurrency control problems for database systems; solutions to such problems heavily rely on the ability to assign timestamps and version numbers to files or other entities. Also, some algorithms that use timeouts, such as communication protocols, are very much dependent on a value of real time common among processes.

Upon finishing the execution of a tick synchronization algorithm, a process enters a *synchronized* state. Informally, the *precision* that a tick synchronization algorithm achieves is the maximum, over all processes in the system, of the difference between the real-time estimate that a process is making at precisely the time at which it is entering a synchronized state and the real-time estimate that any other process in a synchronized state is making at the same time[1].

A synchronized state models the possibility that a process is in a position to make use of the estimate of real time it has obtained as a result of executing a tick synchronization algorithm. Clearly, after all processes enter a synchronized state, clock drifts may bring the system out of synchronization again; so, it makes sense to consider the behavior of the system prior to the time at which the last process enters a synchronized state. Multiple runs of a tick synchronization algorithm, appropriately scheduled, in a way, possibly, similar to [10], may reduce such future "desynchronizations".

Time is measured under the following assumptions on the system: Messages sent on a communication link incur a delay in the range $[0, d]$, where $d \geq 0$ is a (known) constant. The time between any two consecutive clock ticks (equivalently, consecutive computation steps of a process) is in the interval $[c, 1]$ for some parameter c such that $0 < c \leq 1$.

We show a lower bound of $\lfloor \frac{d-2}{2c} \rfloor$ on the precision achievable by any tick synchronization algorithm. Our proof follows a general technique of explicitly "shifting" and "shrinking" executions through retiming of events, reminiscent of a technique originally introduced in [9], which subsequently found applications in many different contexts (see, e.g., [2, 3, 4, 13]). Since, however, we are assuming that processes acquire information about time by receiving ticks from their inaccurate, discrete clocks, while [9] assumes processes have access to continuous clocks running at a perfect rate, that of real time, the precise details differ substantially. Thus, while the lower bound proof in [9] relies on message delivery time uncertainty, our lower bound proof focuses more on timing uncertainty. Clearly, our lower bound is interesting in the cases where $d > 2$.

We also present a simple algorithm that achieves a precision of $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$. This algorithm relies on explicit communication among the processes, so that each of them can estimate the difference between the local time estimate of every other process and its own, and add the average of these estimated differences to its local time estimate. This algorithm is a direct adaptation for the semi-synchronous model of one presented in [9]. Its analysis, however, is more intricate, since the timing assumptions in the semi-synchronous model are more "crisp" than the ones in [9], where clocks were assumed to run at a perfect rate.

The tick synchronization problem can be thought of as a "discrete analog" of the classical clock synchronization problem that has been extensively studied in the literature. We believe that discrete synchronization problems will play an important role in the development of a theory of real-time computing. To the best of our knowledge, the only synchronization problem that has been studied so far in the context of real-time distributed systems is the *session problem*, where a process is required to guarantee that all processes have performed a particular set of steps. Several combinatorial results on time bounds for the session problem in asynchronous and semi-synchronous models have been presented in [3, 12, 16]; our present results are similar in style to those.

The rest of this paper is organized as follows: Section 2 presents the system model, defines the tick synchronization problem and introduces some notation. Section 3 contains our lower bound result, and Section 4 contains our upper bound result. We conclude, in Section 5, with a discussion of our results and some open problems.

2 Definitions

In this section, we present the definitions for the underlying formal model[2], define what it means for an algorithm to solve the tick synchronization problem and introduce some notation.

2.1 The System Model

A *system* consists of n processes $p_1, \ldots, p_n$. Processes are located at the nodes of a complete graph

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

[2] These definitions are similar to those in [3] and could be expressed in terms of the general *timed automaton model* described in [2, 11, 14].

$G = (V, E)$, where $V = [n]$. For simplicity, we identify processes with the nodes they are located at and we refer to nodes and processes interchangeably. Each process p_i is modelled as a (possibly infinite) state machine with state set Q_i. The state set Q_i contains a distinguished *initial state* $q_{0,i}$. The state set Q_i also includes a subset S_i of *synchronized* states. We assume that any state of p_i includes a special component, $buffer_i$, which is p_i's message buffer. A *configuration* is a vector $C = (q_1, \ldots, q_n)$ where q_i is the local state of p_i; denote $state_i(C) = q_i$. The *initial configuration* is the vector $(q_{0,1}, \ldots, q_{0,n})$. Processes communicate by sending *messages*, taken from alphabet $\mathcal{M}$, to each other. A *send action* $send(j, m)$ represents the sending of message m to a neighboring process p_j. Let $\mathcal{S}_i$ denote the set of all send actions $send(j, m)$ for all $m \in \mathcal{M}$ and all $j \in [n]$, such that $(i, j) \in E$. That is, $\mathcal{S}_i$ includes the set of all the send actions possible for p_i.

We model computations of the system as sequences of *atomic events*, or simply *events*, for short. Each event is either a *computation event*, representing a computation step of a single process, or a *delivery event*, representing the delivery of a message to a process. Each *computation event* is specified by $comp(i)$ for some $i \in [n]$. In the computation step associated with event $comp(i)$, the process p_i, based on its local state, changes its local state and performs some set S of send actions, where S is a finite subset of $\mathcal{S}_i$. Each delivery event has the form $del(i, m)$ for some $m \in \mathcal{M}$. In a delivery step associated with the event $del(i, m)$, the message m is added to $buffer_i$, p_i's message buffer.[3]

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

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

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

satisfying the following conditions:

1. C_0 is the initial configuration;

2. If $\pi_j = del(i, m)$, then $state_i(C_j)$ is obtained by adding m to $buffer_i$.

3. If $\pi_j = comp(i)$, then $state_i(C_j)$ and $\mathcal{S}$ are obtained by applying $\mathcal{A}_i$ to $state_i(C_{j-1})$;

4. If π_j involves process i, then $state_k(C_{j-1}) = state_k(C_j)$ for every $k \neq i$;

5. For each $m \in \mathcal{M}$ and each process p_i, let $S(i, m)$ be the set of j such that π_j contains a $send(i, m)$ and let $D(i, m)$ be the set of j such that π_j is a delivery event $del(i, m)$. Then there exists a one-to-one onto mapping $\sigma_{i,m}$ from $S(i, m)$ to $D(i, m)$ such that $\sigma_{i,m}(j) > j$ for all $j \in S(i, m)$

That is, in an execution the changes in processes' states are according to the transition function, only a process which takes a step or to which a message is delivered changes its state, and each sending of a message is matched to a later message delivery and each message delivery to an earlier send. We adopt the convention that finite prefixes of an execution end with a configuration, and denote the last configuration in a finite execution prefix α by $last(\alpha)$. We say that $\pi_j = comp(i)$ is a *synchronized step* of the execution if $state_i(C_j) \in S_i$, i.e., it is taken from a synchronized state.

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

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

where the times are nondecreasing and unbounded.

Timed executions in this model are defined as follows. Fix real numbers c and d, where $0 < c \leq 1$ and $0 \leq d < \infty$.[4] Letting α be a timed sequence as above, we say that α is a *timed execution* of $\mathcal{A}$ provided that the following all hold:

1. $C_0, \pi_1, C_1, \ldots, \pi_j, C_j, \ldots$ is an execution of P;

2. (Synchronous start) There are computation events for all processes with time 0;

3. (Upper bound on step time) If the jth timed event is $(t_j, comp(i_j, S_j))$, then there exists a $k > j$ with $t_k \leq t_j + 1$ such that the kth timed event is $(t_k, comp(i_j, S_k))$;

4. (Lower bound on step time) If the jth timed event is $(t_j, comp(i_j, S_j))$, then there does not exist a $k > j$ with $t_k < t_j + c$ such that the kth timed event is $(t_k, comp(i_j, S_k))$;

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

[4] The *synchronous* model is a special case of the present model where $c = 1$ and $d < 1$.

5. (Upper bound on message delivery time) If message m is sent to p_i at the jth timed event, then there exists $k > j$ such that the kth timed event is the matching delivery $(t_k, del(i, m))$ (i.e., $\sigma_{i,m}(j) = k$) and $t_k \leq t_j + d$.

We say that α is an *execution fragment* of $\mathcal{A}$ if there is an execution α' of $\mathcal{A}$ of the form: $\alpha' = \beta\alpha\beta'$. This definition is extended to apply to timed executions in the obvious way. For a finite execution fragment $\alpha = C_0, (t_1, \pi_1), C_1, \ldots, (t_k, \pi_k), C_k$, we define $t_{start}(\alpha) = t_1$ and $t_{end}(\alpha) = t_k$. We say that a process p_j *receives the message* m *by time* t' (in a timed execution α) if, by time t', p_j has a computation event that is preceded in α by a delivery event $del(j, m)$. Note that if m is sent to p_j at time t, then p_j receives m by time $t+d+1$.

We say that a process p_i *enters a synchronized state* by time t' (in a timed execution α) if there exists a timed event (t_{j-1}, π_{j-1}) in α such that $t_{j-1} \leq t'$, $\pi_{j-1} = comp(i)$, and $state_i(C_j) \in S_i$.

2.2 The Tick Synchronization Problem

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

Intuitively, the *tick synchronization problem* is the problem of establishing synchronization among the processes, assuming that a process p_i can modify L_i during the execution of a synchronization algorithm in some way other than just incrementing it by one at the rate at which it receives its ticks. We assume that a process can start executing a tick synchronization algorithm either spontaneously or upon receipt of a message from a process that has already done so. Let t_i be the time at which p_i finishes executing its synchronization algorithm. We say that p_i is in a *synchronized* state at any time $t \geq t_i$. We will denote by $L_i(t_i+)$ the local, real-time estimate that p_i is making at t_i as a result of a run of a tick synchronization algorithm; that is, at time t_i, potentially, p_i updates $L_i(t_i)$ to $L_i(t_i+)$, based on knowledge gathered during the execution of the algorithm and enters a synchronized state. Clearly, it makes sense to compare $L_i(t)$ and $L_j(t)$ for $t \geq t_i, t_j$, when both p_i and p_j are guaranteed to be in a synchronized state. Furthermore, for a particular process p_i, it is most appropriate to compare $L_i(t)$ and $L_j(t)$, where p_j is any process that has also entered a synchronized state by t_i, at time *exactly* t_i, since further "asymmetry" in the rates at which p_i and p_j receive ticks can occur after t_i to separate L_i and L_j even more. Thus, we are somehow interested for the *worst* asynchronism that can possibly occur at the *best* moment for a process, which is the time at which the process is entering a synchronized state. This is in contrast to the situation investigated in [9], where once the clocks are brought into synchronization, synchronization is maintained for ever.

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

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

3 A Lower Bound

We show:

Theorem 3.1 *No clock synchronization algorithm can synchronize* P *within precision* γ *for any* $\gamma < \lfloor \frac{d-2}{2c} \rfloor$.

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

Consider a fast, synchronous infinite timed execution α of $\mathcal{A}$ in which all processes take steps at a rate of c in a round-robin order, starting with p_1, and start spontaneously and simultaneously executing their local protocols, and all messages are delivered after exactly $\frac{d}{2}$ delay. As a result of our assumptions, α will also be "symmetric" in the sense that all processes will undergo the same state changes in a synchronous fashion, enter a synchronized state simultaneously and make a common estimate of real time. Let $\alpha = \beta\beta'$, where β is the longest prefix of α such that some process is not in a synchronized state in $last(\beta)$, and β' is the remaining part of α. We reorder and retime events in α to construct an infinite timed execution α_1 of $\mathcal{A}$ which is equivalent to α in the sense that for each process p_i, events at p_i occur in the same order in α_1 as in α. This will guarantee that α and α_1 will be indistinguishable to the processes and, therefore, each process will undergo the same state changes and, therefore, make the same estimate of real time upon entering a synchronized state, as a result of a run of $\mathcal{A}$, for each of these executions.

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

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

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

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

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

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

Lemma 3.2 *Each receiving event is after the corresponding sending event in α_1.*

Proof: Consider the message sending event π_1 at node u_1 which occurs at time t_1 in α and let π_2 be the corresponding message delivery event at node u_2 which occurs at time t_2 in α. In α_1, let π_1 occur at time t'_1 and π_2 occur at time t'_2. We show, by case analysis, that the ordering of π_1 and π_2 is the same in α_1 as in α.

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

2. $u_1 = p_n$: In this case $t'_2 = t_2$; thus, we only need to consider the subcase where $t_1 \geq T_n - a$, since, otherwise, $t'_1 = t_1$, and the claim becomes trivial. We can also assume that $t_2 \leq T_2$, since, otherwise, π_2 occurs in β' and can be rescheduled to occur at a later time in β' without affecting the estimate of real time made by u_2 at T_2. Note that since:
$$t'_2 - t'_1 = t_2 - t'_1 = t_2 - t_1 - (t'_1 - t_1) = \frac{d}{2} - (t'_1 - t_1),$$
to show that $t'_2 \geq t'_1$, it suffices to show that $t'_1 - t_1 \leq \frac{d}{2}$. By our construction, the first computation step of u_1 that occurs at time $\geq T_n - a$ in α will occur at time $\lceil T_1 - a \rceil$ in α_1. Since there are at most $\lceil \frac{t_1-(T_n-a)}{c} \rceil$ computation steps of u_1 that occur in α at time t such that: $T_n - a \leq t \leq t_1$ and u_1 is a-retarded in α_1, we will have:
$$\begin{aligned}
t'_1 - t_1 &= \lceil T_n - a \rceil + (\lceil \frac{t_1 - (T_n - a)}{c} \rceil - 1) - t_1 \\
&\leq T_n - a + 1 + \frac{t_1 - (T_n - a)}{c} + 1 - 1 \\
&\quad -(T_n - a + t_1 - (T_n - a)) \\
&= 1 + \frac{t_1 - (T_n - a)}{c} - (t_1 - (T_n - a)) \\
&= 1 + (t_1 - (T_n - a))\frac{1-c}{c} \\
&\leq 1 + (T_n - (T_n - a))\frac{1-c}{c} \\
&\quad (\text{since } t_1 \leq T_n) \\
&= 1 + a\frac{1-c}{c} \\
&= 1 + (\frac{d}{2} - 1)\frac{c}{1-c}\frac{1-c}{c} \\
&= \frac{d}{2},
\end{aligned}$$
as needed.

3. $u_2 = p_n$: We only need to consider the subcase where $t_2 \geq T_2 - a$, since, otherwise, $t'_2 = t_2$, and the claim is trivial. It is obvious, however, that, by construction, we will then have: $t'_2 > t_2 \geq t_1 = t'_1$, as needed.

■

We next show:

Lemma 3.3 *The time between a message sending event and the corresponding message delivery event in α_1 is at most d.*

Proof: Consider the message sending event π_1 at node u_1 which occurs at time t_1 in α and let π_2 be the corresponding message delivery event at node u_2 which occurs at time t_2 in α. In α_1, let π_1 occur at time t'_1 and π_2 occur at time t'_2. We show, by case analysis, that: $t'_2 - t'_1 \leq d$.

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

2. $u_1 = p_n$: In this case, $t'_2 = t_2$, while, by construction, $t'_1 \geq t_1$. Thus: $t'_2 - t'_1 \leq t_2 - t_1 = \frac{d}{2} < d$.

3. $u_2 = p_n$: In this case, $t'_1 = t_1$. As in Lemma 3.2, we can show that: $t'_2 - t_2 \leq \frac{d}{2}$. Thus:
$$t'_2 - t'_1 = t'_2 - t_1 = t'_2 - t_2 + t_2 - t_1 \leq \frac{d}{2} + \frac{d}{2} = d,$$
as needed.

■

We can now show:

Lemma 3.4 *α_1 is a timed execution of $\mathcal{A}$.*

Proof: Obvious from Lemma 3.2, Lemma 3.3 and the fact that by construction, any two consecutive computation steps of any process are either c or 1 apart in α_1. ■

Thus, we have shown so far that α_1 is a timed execution of $\mathcal{A}$. Moreover, p_n makes precisely the same estimate about real time at the moment it is entering a synchronized state in each of α and α_1. Let T'_n be the (real) time at which p_n is entering a synchronized state in α_1. Let $L_n(T_n+)$ and $L_n(T'_n+)$ be the estimates of real time that p_n is making at real times T_n and T'_n in α and α_1, respectively. By our construction, $L_n(T_n+) = L_n(T'_n+)$. By symmetry, T_{n-1}, the time at which p_{n-1} is entering a synchronized state in α (in α_1, as well, since there are no changes for the times at which events at p_{n-1} occur in α_1) must equal T_n; by symmetry, also, $L_{n-1}(T_{n-1}+) = L_n(T_n+)$. We show a simple fact:

Claim 3.5 *The number of ticks that process p_{n-1} receives between T_n and T'_n is at least $\lfloor \frac{d-2}{2c} \rfloor$.*

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

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

Therefore:

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

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

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

■

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

$$\begin{aligned} \gamma + L_n(T_n+) &= \gamma + L_n(T'_n+) \\ &\quad \text{(since } \alpha \text{ and } \alpha_1 \text{ are equivalent)} \\ &\geq L_{n-1}(T'_n) \\ &\quad \text{(since } \mathcal{A} \text{ synchronizes } P \text{ within} \\ &\quad \text{precision } \gamma) \\ &= L_{n-1}(T_{n-1}+) + m \\ &= L_n(T_n+) + m \\ &\quad \text{(since } \alpha \text{ is symmetric with respect} \\ &\quad \text{to } p_n \text{ and } p_{n-1}) \\ &\geq L_n(T_n+) + \lfloor \frac{d-2}{2c} \rfloor \\ &\quad \text{(by Claim 3.5)} \end{aligned}$$

Therefore:

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

This completes our proof. ■

4 An Upper Bound

In this section, we prove the following theorem:

Theorem 4.1 *There exists an algorithm which synchronizes P within precision $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c}d + 1$.*

Proof: We describe an algorithm which is very similar to the one in [9]. Each process p can start executing the synchronization algorithm either spontaneously or upon receiving a message from a process that has already done so. As soon as it starts, it sends its local time in a message to the remaining processes and waits to receive a similar message from every other process.

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

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

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

We start by showing:

Lemma 4.2 $|L_p(t_q) - L_q(t_q-) - (R_q(t_q-) - R_p(t_p-))| \leq \frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2})$

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

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

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

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

Hence:

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

Next, we show some simple facts:

Claim 4.3 $\sum_{r \in P}|L_r(t) - L_q(t_q-) - D_{rq}| \leq (n-1)\lceil \frac{2d}{c} \rceil + \frac{d}{2}$

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

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

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

$$|L_r(t) - L_q(t_q-) - D_{rq}|$$

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

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

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

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

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

Therefore:

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

■

As in Claim 4.3, we can show:

Claim 4.4 $\sum_{r \in P} |L_r(t) - L_p(t_q-) - D_{rp}| \le (n-1)(\lceil \frac{2d}{c} \rceil + \frac{d}{2})$

The lemma follows from the last two claims. ■

We next show:

Lemma 4.5 $|T_p(t_q, t_p) - T_q(t_q, t_p)| \le \frac{1-c}{c} d + 1$

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

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

■

The theorem follows from Lemma 4.2 and Lemma 4.5. ■

5 Discussion and Future Research

In this paper, we defined the tick synchronization problem, a variant of the general synchronization problem, in semi-synchronous distributed networks and proposed the precision achieved by a tick synchronization algorithm as an appropriate worst-case measure of its performance. We showed that no algorithm can solve the tick synchronization problem and yet achieve precision less than $\lfloor \frac{d-2}{2c} \rfloor$. On the positive side, we presented a simple algorithm that achieves a precision of $\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c} d + 1$.

Neglecting round-offs and assuming that $c \ll 1$, the dominant terms in the expressions for our lower and upper bounds on precision will be the ones that are proportional to $\frac{d}{c}$. In this case, we get the following approximations:

$$\lfloor \frac{d-2}{2c} \rfloor \approx \frac{1}{2} \frac{d}{c} \ ,$$

and

$$\begin{aligned}&\frac{2(n-1)}{n}(\lceil \frac{2d}{c} \rceil + \frac{d}{2}) + \frac{1-c}{c} d + 1 \\ \approx\ & \frac{2(n-1)}{n} \frac{2d}{c} + \frac{d}{c} \\ =\ & (5 - \frac{4}{n}) \frac{d}{c}\end{aligned}$$

Thus, our lower bound is approximately within a factor of $2(5 - \frac{4}{n}) < 10$ of our upper bound under the assumption that $c \ll 1$. Although our bounds are, in general, not completely tight, we feel that our work substantially answers the question of how the precision achievable in a completely connected semi-synchronous network depends on the timing and message delay uncertainties, as measured by $\frac{d}{c}$.

There are several open problems directly related to the work in this paper. Most obviously, there is a gap remaining between our upper and lower bounds. We believe that an algorithm using more sophisticated averaging than the one we presented may exist and imply a better upper bound on precision. It would be interesting to consider the same problem in a model in which there is a nontrivial lower bound on the time for message delivery. While our upper bound proof still goes through in this model, the same is not true for our lower bound proof. Perhaps, the most intriguing open problem is the extension of this work to the case of a general communication network. We have some preliminary results towards this direction.

The work presented in this paper continues the study of time bounds in the presence of timing uncertainty within the framework of the semi-synchronous model ([1, 2, 3]). We believe that other related problems can also be studied using the models and techniques of this paper. One can define timing-based analogs of other problems that have been studied in the asynchronous setting, for example, other exclusion problems such as the *dining philosophers* problem, or the k-critical section problem of [6]. Another interesting direction is to study problems in the semi-synchronous model, assuming that processes communicate via shared-memory.

Besides the semi-synchronous model, there have been recently proposed many other models for concurrent computation that make different assumptions about the timing information that is available to the processes, for example, the *periodic* and *sporadic* models introduced in [16]. What precision can be achieved in these models?

References

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

[2] H. Attiya and N. Lynch, "Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty," in *Proceedings of the 10th IEEE Real-Time Systems Symposium,* pp. 268–284, December 1989.

[3] H. Attiya and M. Mavronicolas, "Efficiency of Semi-Synchronous vs. Asynchronous Networks," in *Proceedings of the 28th Annual Allerton Conference on Communication, Control and Computing,* pp. 578–587, October 1990. Expanded version: TR 21-90, Aiken Computation Laboratory, Harvard University, September 1990.

[4] H. Attiya and J. Welch, "Sequential Consistency versus Linearizability," in *Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures,* pp. 304–315, July 1991.

[5] J. Burns and N. Lynch, "The Byzantine Firing Squad Problem," *Advances in Computing Research,* vol. 4, pp. 147–161, 1987.

[6] M. Fischer, N. Lynch, J. Burns and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Space," *ACM Transactions on Programming Languages and Systems,* Vol. 11, No.1, pp. 90–114, January 1989.

[7] J. Halpern, N. Megiddo and A. Munshi, "Optimal Precision in the Presence of Uncertainty," *Journal of Complexity,* Vol. 1, No. 2, pp. 170–196, December 1985.

[8] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Communications of the ACM,* Vol. 21, No. 7, pp. 558–565, July 1978.

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

[10] J. Lundelius and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation,* Vol. 77, No. 1, pp. 1–36, April 1988.

[11] N. Lynch and H. Attiya, "Using Mappings to Prove Timing Properties," in *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing,* pp. 265–280, August 1990.

[12] M. Mavronicolas, *Efficiency of Semi-Synchronous versus Asynchronous Systems: Atomic Shared Memory,* Technical Report TR-03-92, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, 1992. To appear in *Computers and Mathematics with Applications.*

[13] M. Mavronicolas and D. Roth, "Efficient, Strongly Consistent Implementations of Shared Memory," to appear in the *6th International Workshop on Distributed Algorithms,* November 1992. Also: Technical Report TR-05-92, Center for research in Computing Technology, Aiken Computation Laboratory, Harvard University, 1992.

[14] M. Merritt, F. Modugno and M. Tuttle, "Time Constrained Automata," in *Proceedings of the 2nd International Conference on Concurrency,* pp. 408–423, Lecture Notes in Computer Science (Vol. 527), Springer-Verlag, August 1991.

[15] S. Ponzio, "Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures," in *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing,* pp. 125–138, August 1991.

[16] I. Rhee and J. Welch, "The Impact of Time on the Session Problem," to appear in the *11th Annual ACM Symposium on Principles of Distributed Computing,* August 1992.

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