Optimal clock synchronization revisited: Upper and lower bounds in real-time systems

Heinrich Moser* and Ulrich Schmid
Technische Universität Wien
Embedded Computing Systems Group (E182/2)
A-1040 Vienna (Austria)
{moser,s}@ecs.tuwien.ac.at

July 25, 2006

Abstract: *This paper[1] introduces a simple real-time distributed computing model for message-passing systems, which reconciles the distributed computing and the real-time systems perspective: By just replacing instantaneous computing steps with computing steps of non-zero duration, we obtain a model that both facilitates real-time scheduling analysis and retains compatibility with classic distributed computing analysis techniques and results. We provide general simulations and validity conditions for transforming algorithms from the classic synchronous model (without clock drift) to our real-time model and vice versa, and investigate whether/which properties of real systems are inaccurately or even wrongly captured when resorting to zero step-time models. We revisit the well-studied problem of deterministic internal clock synchronization for this purpose, and show that no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. Since such an algorithm is known for the classic model, we have found an instance of a problem where the standard distributed computing analysis gives too optimistic results. We prove that optimal precision is only achievable with algorithms that take $\Omega(n)$ time in our model, and establish several additional lower bounds and algorithms.*

*Corresponding author.

[1]This work is part of our project *Theta* (*http://www.ecs.tuwien.ac.at/projects/Theta*), which is supported by the Austrian Science Foundation (FWF) under grant P17757.

1 Motivation

Executions of distributed algorithms are typically modeled as sequences of atomic computing steps that are executed in zero time. With this assumption, it does not make a difference, for example, whether messages arrive at a processor simultaneously or nicely staggered in time: The messages are processed instantaneously when they arrive. The zero step-time abstraction is hence very convenient for analysis, and a wealth of distributed algorithms, impossibility results and lower bounds have been developed for models that employ this assumption [15].

In real systems, however, computing steps are neither instantaneous nor arbitrarily preemptable: A computing step triggered by a message arriving in the middle of the execution of some other computing step is usually delayed until the current computation is finished. This results in queueing phenomenons, which depend not only on the actual message arrival pattern but also on the queueing/scheduling discipline employed. The real-time systems community has established powerful techniques for analyzing such effects [22], such that the resulting worst-case response times and end-to-end delays can be computed.

This paper introduces a real-time distributed computing model for message-passing systems, which reconciles the distributed computing and the real-time systems perspective: By just replacing the zero step-time assumption with non-zero step times, we obtain a real-time distributed computing model that admits real-time analysis without invalidating standard distributed computing analysis techniques and results: We show that a system adhering to the real-time model can simulate a system that adheres to the classic model and vice versa.

Apart from making distributed algorithms amenable to real-time analysis, our model also allows to address the interesting question whether/which properties of real systems are inaccurately or even wrongly captured when resorting to classic zero step-time models. In this paper, we revisit the well-studied problem of deterministic internal clock synchronization [23, 14] for this purpose. Clock synchronization is a particularly suitable choice here, since the achievable synchronization precision is known to depend on the end-to-end delay uncertainty (i.e., the difference between maximum and minimum end-to-end delay). Since non-zero computing step times are likely to affect end-to-end delays, one may expect that some results obtained under the classic model do not hold under the real-time model — if there are such effects at all.

Our analysis confirms that this is indeed the case: We show that no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. Since such an algorithm has been given for the classic model [14], we have found an instance of a problem where the standard distributed computing analysis gives too optimistic results. Actually, we show that optimal precision is only achievable with algorithms that take $\Omega(n)$ time, even if they are provided with a constant-time broadcast primitive.

Detailed major contributions:

(1) In Section 3, we define our real-time computing model ($\mathcal{M}$) for synchronous message-passing systems (both point-to-point and broadcast-based), which differs from the classic computing model ($\underline{\mathcal{M}}$) [14] by just providing atomic computing steps of non-zero duration.

(2) In Section 5, we provide transformations from the real-time computing model to the classic computing model (and vice versa): We show that a system adhering to some particular instance of $\mathcal{M}$ can simulate a system that adheres to some particular instance of $\underline{\mathcal{M}}$ (and vice versa). Consequently, certain distributed algorithms designed for a classic computing model can be run under the real-time computing model, for example.

(3) In Section 6, we revisit deterministic internal clock synchronization in synchronous systems [14], in the absence of failures and clock drift. It is known that the local clocks of n fully-connected processors cannot be synchronized with precision less than $(1-1/n)\underline{\varepsilon}$ when using messages with end-to-end delay uncertainty $\underline{\varepsilon}$. A constant time algorithm achieving this bound in the classic computing model also exists.
We show that this is not true in the real-time computing model: Optimal precision is only achievable with algorithms that take $\Omega(n)$ time. On the other hand, achieving a sub-optimal precision of O($\underline{\varepsilon}$) is achievable in constant time, if, and only if, a constant-time broadcast primitive is available.

Related Work: We are not aware of much existing work that is similar in spirit to our approach. Somewhat an exception is the work by Neiger & Toueg [19], which identified general problems and conditions that preserve the correctness of a solution based on perfectly synchronized clocks when logical clocks are used instead. The underlying model assumes non-zero step times but considers them sufficiently small to compeltely ignore queueing effects. Moreover, in contrast to our work, they restrict their attention to "internal problems" (essentially corresponding to our trace properties) only.

Another example of a non-zero step time model is the the remote memory reference (RMR) model for shared-memory systems [3, 2] by Anderson et. al. It assumes computing step times which depend upon the number of conflicting shared memory accesses. The RMR model has been used for deriving several algorithms e.g. for mutual exclusion and related lower bounds. Since is not applicable to message-passing systems, however, we cannot compare our results.

Another branch of research where distributed computing and real-time systems issues are combined are modeling frameworks [1, 16, 17, 18, 21, 10]. Such frameworks allow formal modeling and analysis of complex distributed real-time systems. A representative example are Timed I/O Automata (TIOA) [10], which can change state both via ordinary discrete transitions and via continuous trajectories. TIOAs facilitate hierarchical composition, abstraction, and proofs of safety and liveness properties. However, none of the above modeling frameworks supports non-zero step times and thus real-time scheduling analysis of distributed algorithms. By contrast, our work addresses exactly this issue.

Apart from those lines of research, we are not aware of too many distributed computing papers that incorporate real-time scheduling issues at all: In [8], for example, Hermant and Le Lann demonstrated the power of such an integrated approach by introducing fast failure detectors, which facilitate very fast detection times and thus quickly terminating asynchronous consensus algorithms. The Theta-Model proposed in [12, 25, 9] is an example of a (zero step-time) distributed computing model that takes advantage of real-time scheduling issues for bounding the ratio of minimal and maximal end-to-end delays. Clock synchronization under real-time scheduling is considerd by Basu and Punnekkat [5]. They propose simple variants of Srikanth & Toueg's clock synchronization algorithm [24] that can deal with scheduling latencies in heavily loaded real-time systems.

2 Classic Computing Model

In clock synchronization research [13, 6, 20, 4, 7, 14], system models are considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted "Partially Synchronous Reliable/Unreliable Models" in [23], such models are nowadays called (non-lockstep) synchronous models in literature. In order to solely investigate the effects of non-zero step-times, our real-time computing model will be based on the simple failure- and drift-free synchronous model introduced in [14]. It will be refered to as the *classic computing model* in our paper.

2.1 Classic System Model

We consider a network of n failure-free *processors*, which communicate by passing unique messages, using either a unicast, multicast or broadcast primitive. The system-wide set of messages in transit will be denoted $intransit\_msgs$. Each processor p is equipped with a CPU, some local memory, a hardware clock HC_p, and reliable, non-FIFO links to all other processors. Each hardware clock runs at the same rate as real time[2]; it can be read but not changed by its processor. HC_p is hence not part of the local state, but considered seperately.

The CPU is running an algorithm, specified as a mapping from processor indices to a set of initial states and a transition function. Processor p's set of *initial states* is denoted $Init_p$. The algorithm's *transition function* takes the processor index p, one incoming message (taken from the current $intransit\_msgs$), receiver processor p's current local state $oldstate$ and hardware clock reading HC_p as input, and yields a list $[oldstate, \dots, newstate]$ as output. $newstate$ is the new state after the transition and in between $oldstate$ and $newstate$ there may be an arbitrary number of *intermediate states* and *messages* to be sent. These intermediate states are usually neglected in the classic computing model, as the state transition from $oldstate$ to $newstate$ is instantaneous anyway. We explicitely model these states to retain compatibility with our real-time computing model, where they will become more important.

Every message arrival (also called message reception) simultaneously causes the message to be removed from $intransit\_msgs$ and the receiver processor to change its state and send out all messages according to the transition function (i.e., they are added to $intransit\_msgs$). Such a *computing step* (also called *message processing step*) will be called an *action* in the sequel. The complete action, i.e., message arrival, message processing and sending new messages, is performed instantly, i.e., in zero time.

Actions can actually be triggered by two different types of messages: Ordinary messages and timer messages. Ordinary messages are transmitted over the links. The *message delay* $\underline{\delta}$ is the difference between the real time of the action sending the message and the real time of the action receiving the message. There is a lower bound $\underline{\delta}^-$ and an upper bound $\underline{\delta}^+$ on the message delay of every ordinary message.[3]

Timer messages are used for modeling time(r)-driven execution in our message-driven setting: Typical clock synchronization algorithms setup one or more local timers in a computing step, the expiration of which triggers the execution of another computing step. A processor setting a timer is modeled as sending a timer message (to itself) in an action, and timer expiration is represented by the reception of a timer message. Note that timer messages do not need to obey the message delay bounds, since they are received when the hardware clock reaches (or has already reached) the time specified in the timer message.

We assume that every processor p in the system is in some initial state $istate_p \in Init_p$ right from the system start, at real-time $t = 0$. To trigger the first action of a processor in an execution, we allow one special *init message* to arrive at each processor from outside the system. These messages are of course exempt from the requirement of having been sent by some processor in the system, and need not satisfy the delay bounds. We will assume that all init messages arrive within a sufficiently short time interval, so that the initialization uncertainty does not significantly affect the time complexity of our algorithms. On the other hand, we consider the initialization uncertainty to be large enough to prohibit initial system-wide synchronization.

2.2 Executions

An execution in the classic computing model is a sequence of actions. An action ac occurring at real-time t at processor p is a 5-tuple, consisting of the processor index $proc(ac) = p$, the received message $msg(ac)$, the occurence real-time $time(ac) = t$, the hardware clock value $HC(ac) = HC_p(t)$ and the state transition sequence $trans(ac) = [oldstate, \dots, newstate]$. Let $states(ac)$ be defined as the list of all states and $sent(ac)$ as the list of all messages in $trans(ac)$. The abbreviations $oldstate(ac)$ and $newstate(ac)$ will be used for the first and the last entry in $states(ac)$.

A valid execution of an algorithm $\underline{\mathcal{A}}$ must satisfy the following properties:

[2]We assume that there is some Newtonian reference time, refered to as real-time, which is of course only available for analysis purposes.

[3]$\underline{\delta}^-$ and $\underline{\delta}^+$ are called μ and ν in [14]. To disambiguate our notation, systems, parameters like message delay bounds, and algorithms in the classic computing model are represented by underlined variables (usually $\underline{s}, \underline{\delta}^-, \underline{\delta}^+, \underline{\mathcal{A}}$).

- All state transitions and sent messages must be in accordance with the transition function defined in $\underline{\mathcal{A}}$.
- The first action at every processor p must occur in an initial state of p and may—but need not—be triggered by an external init message. We will use $istate_p^{ex}$ to refer to the initial state of p in execution ex.
- All hardware clock readings on the same processor must be consistent with the fact that hardware clocks run at the same rate as real time.
- If a timer message is sent for reception at time T, it arrives when the hardware clock reads T, i.e., triggers an action ac with $HC(ac) = T$. A timer message sent for some time T smaller than the current hardware clock reading $T' > T$ arrives immediately, i.e., triggers an action ac with $HC(ac) = T'$.
- There is a one-to-one correspondence between sent messages and message receptions in the obvious way: All sent messages are eventually received (exactly once), and all received messages have been sent (exactly once). The only exception are init messages.

As an execution is a *sequence* of actions, there is a well-defined total order $<^{ex}$ on actions. We will omit the superscript if it is clear from context.

2.3 Systems and Admissible Executions

A *classic system* $\underline{s}$ is a system adhering to the classic computing model defined in Section 2.1, parameterized by the system size n and the interval $[\underline{\delta}^-, \underline{\delta}^+]$ specifying the bounds on the message delay. The uncertainty $\underline{\varepsilon}$ is defined as $\underline{\delta}^+ - \underline{\delta}^-$.

Definition 1. Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. An execution is *$\underline{s}$-admissible*, if the execution contains exactly n processors, the message delay for each ordinary message stays within $[\underline{\delta}^-, \underline{\delta}^+]$ and the ordering of actions captures causality.

Note that in the classic computing model it is possible that two or more actions happen on the same processor at exactly the same time, for example, if an action sets a timer for the current hardware clock value or if two messages arrive at exactly the same time. *Capturing causality* means that (a) if an action ac' happens after[4] action ac on the same processor, $ac < ac'$, and (b) if an action ac sends a message received by some action ac', $ac < ac'$ (cf. *happened before* relation, [11]). Additionally, we require $time(ac) < time(ac') \Rightarrow ac < ac'$ for convenience.

Claiming that an algorithm solves some problem for a classic system $\underline{s}$ means that all possible $\underline{s}$-admissible executions of the algorithm must satisfy the required properties (see Section 4). The task of finding such an algorithm can be seen as providing a winning strategy to a player in an execution-creation game against an adversary, where the player provides the sets of initial states and the state transition function and the adversary chooses one initial state and the initial hardware clock values for every processor and controls the message delays (within the bounds $[\underline{\delta}^-, \underline{\delta}^+]$ provided by the system). Note carefully that it is the system/the adversary and not the algorithm that determines the actual delays in the classic computing model.

$intransit\_msgs(ac)$ denotes the set of messages in transit *after* action ac has sent all its messages but before any following action $ac' > ac$ in ex has had the opportunity to send or process messages.

2.4 Trajectories

The following definitions allow us to unify the definition of state properties in the classic computing model and in the real-time computing model. Given an execution, it defines the induced *local state trajectories* $t \rightarrow state_p(t)$.

Definition 2 (Classic computing model state trajectories)**.** Processor p's *local state* at time t in an execution is defined by a function $state_p(t)$, called the *local state trajectory*, which must satisfy the following requirements:

- If the first action on p occurs later than t, $state_p(t) = istate_p^{ex}$.
- If one or more actions $ac_1, \ldots, ac_k$ occur at time t on p, $state_p(t) \in \bigcup_{i=1}^{k} states(ac_i)$.
- Otherwise, $state_p(t) = newstate(ac)$ of the last action ac occurring on p before t.

Given an execution ex and a processor p, these requirements do not result in a unique trajectory: Even if there are no concurrent actions, $state_p(time(ac))$ for some action ac on p can be either $oldstate(ac)$, $newstate(ac)$ or any intermediate state. When analyzing state properties of an algorithm, we are usually interested in a *global state trajectory*, i.e., in a vector containing one state trajectory for every processor and the hardware clock values.

Definition 3. A *global state trajectory* of some execution ex is a mapping $t \rightarrow (state_1(t), HC_1(t), \ldots, state_n(t), HC_n(t))$ with

- $state_p(t)$ being a valid local state trajectory for processor p in ex and
- the global state trajectory being *causally consistent* w.r.t. message transmissions.

The second requirement above is only relevant if $\underline{\delta}^- = 0$ (otherwise, it is automatically satisfied): Assume some message m from processor p to processor q is sent *and*

[4]We assume that there is only a single CPU on every processor and, therefore, "happens after" is well-defined for actions on the same processor, even if $time(ac) = time(ac')$.

received at time t. Without this requirement it would be possible to have a global trajectory where $state_p(t)$ is the state before sending the message and $state_q(t)$ is the state after receiving the message. Thus, at t a message is received which has not been sent yet, violating causal consistency.

An algorithm claiming to satisfy some state property must do so for every possible global trajectory (see Section 4). $trajs^{ex}$ denotes the set of all valid global trajectories for execution ex.

3 Real-Time Computing Model

3.1 Motivation

Zero step-time computing models have good coverage in systems where message delays are much higher than message processing times. There are applications like high speed networks and SoCs (systems-on-chip), however, where this is not the case. Additionally, and more importantly, the zero step-time assumption inevitably ignores message queuing at the receiver: It is possible, even in case of large message delays, that multiple messages arrive at a single receiver at the same time. This causes the processing of some of these messages to be delayed until the processor is idle again. Common practice so far is to take this queuing delay into account by increasing the upper bound $\underline{\delta}^+$ on the message delay. This approach, however, has two disadvantages: First, a-priori information about the algorithm's message pattern is needed to determine a parameter of the system model, which creates cyclic dependencies. Second, in lower bound proofs, the adversary can choose an arbitrary message delay within $[\underline{\delta}^-, \underline{\delta}^+]$—even if this choice is not in accordance, i.e., not possible, with the current message arrival pattern. This could lead to overly pessimistic lower bounds.

It is of course not the goal of this paper to explicitly model all the phenomenons (receiver queuing, network queueing, scheduling overhead, ...) usually hidden within some adversary-controlled value. Rather, our aim was to find a suitable tradeoff between model complexity and model coverage. Explicitly modeling just non-zero step times and the resulting effects turned out to be an appropriate choice. Other effects, which depend more on the underlying hardware (e.g. network queueing) or which are unsuitable/too detailed for meaningful lower bounds (e.g. different processing times for different messages) are still abstracted away in (overly conservative) system parameters and thus subject to inappropriate exploitation by the adversary.

3.2 Real-Time System Model

The system model in our real-time computing model is the same as in the classic computing model (recall Section 2.1), except for the following change: A computing step in a real-time system is executed atomically and non-preemptively[5] within a system-wide lower bound μ^- and upper bound μ^+. Note that we allow the processing time and hence the bounds $[\mu^-, \mu^+]$ to depend on the number of messages sent in a computing step. In order to clearly distinguish a computing step in the real-time computing model from a zero-time action in the classic computing model, we will use the term *job* to refer to the former.

Interestingly, this seemingly simple extension has far-reaching implications, which make the real-time computing model more realistic but also more complex. In particular, queueing and scheduling effects must be taken into account:

- We must now distinguish two modes of a processor at any point in real-time t: *idle* and *busy* (= currently executing a job). Since computing steps cannot be interrupted, a *queue* is needed to store ordinary and timer messages arriving while the processor is busy. We assume that messages are stored in the queue in the order in which they have arrived.
- When and in which order messages collected in the queue are processed is specified by some *scheduling policy*, which is, in general, independent of the algorithm. Formally, a scheduling policy is specified as an arbitrary mapping from the current queue state (= a sequence of messages), the hardware clock reading, and the current local processor state onto a single message from that message sequence. The scheduling policy is used to select a new message from the queue whenever processing of a job has been completed.
  In this paper, we assume that the scheduling policy is *non-idling*: When the processor is idle, processing of an incoming message starts immediately. Similarly, when the processor finishes a job and the queue is non-empty, a message from the queue is taken and processing of the corresponding job starts without further delay.
- The delay of a message is measured from the real time of the *start of the job* sending the message to the arrival real time at the destination processor (where the message will be enqueued or, if the processor is idle, immediately causes the corresponding job to start). Like in the classic computing model, message delays of ordinary messages must be within a system-wide lower bound δ^- and an upper bound δ^+. The message delay and hence the bounds $[\delta^-, \delta^+]$ may again depend on

[5]If processing of a message has started, this computing step can neither be interrupted nor preempted. It is possible to simulate interruptable execution in our model, however, by splitting message processing into smaller non-interruptable steps connected by "continue_processing" timers.

the number of messages sent in the sending job.
It may seem counter-intuitive to measure the message delay from the beginning of the job rather then from the actual sending time, but this approach has several advantages: First, end-to-end delays (= message delay + queuing delay) of successive messages can just be added up to determine the duration of a message chain. Second, a-priori knowledge about the message sending pattern of the algorithm (e.g. always at the beginning/always at the end of the sending job) can still be encoded in the message delay bounds. And last but not least, no additional parameters in the system model or in the transition function are required.

- We assume that the hardware clock can only be read at the beginning of a job.[6] This restriction in conjunction with our definition of message delays will allow us to define transition functions in exactly the same way as in the classic computing model. After all, the transition function just defines the "logical" semantics of a transition, but not its timing.
- Contrary to the classic computing model, the state transitions $oldstate \to \ldots \to newstate$ in a single computing step need not happen at the same time: Typically, they occur at different times during the job, allowing an intermediate state to be valid on a processor for some non-zero duration.

Figure 1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. Part (a) shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dashed arrow. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not necessarily be known to the algorithm. Bounds on the queuing delay ω and the end-to-end delay Δ, however, are *not* parameters of the system model—in sharp contrast to the classic computing model, where the end-to-end delay always equals the message delay. Rather, those bounds (if they exist) must be derived from the system parameters (n, $[\delta^-, \delta^+]$, $[\mu^-, \mu^+]$) and the message pattern of the algorithm, by performing a real-time scheduling analysis.

Part (b) of Figure 1 shows the detailed relation between message arrival (enqueueing) and actual message processing.

[6]This models the fact that real clocks cannot usually be read arbitrarily fast, i.e., with zero access time.

3.3 Real-time Runs

This section formalizes the notion of a *real-time run* (*rt-run* for short), which corresponds to an execution in the classic computing model. A *rt-run* is just a sequence of receive events and jobs.

A *receive event* R for a message arriving at processor p at real-time t is a triple consisting of the processor index $proc(R) = p$, the message $msg(R)$, and the arrival real-time $time(R) = t$. Recall that t is the enqueueing time in Figure 1(b).

A *job* J starting at real-time t on processor p is a 6-tuple, consisting of the processor index $proc(J) = p$, the message being processed $msg(J)$, the start time $begin(J) = t$, the job processing time $d(J)$, the hardware clock reading $HC(J) = HC_p(t)$ and the state transition sequence $trans(J) = [oldstate, \ldots, newstate]$. $states(J)$, $sent(J)$, $oldstate(J)$ and $newstate(J)$ are abbreviations for parts of $trans(J)$ and defined analogously to the classic computing model (see Section 2.2). Let $end(J)$ be defined as $begin(J) + d(J)$.

Figure 1 provides an example of a rt-run, containing three receive events and three jobs on the second processor. For example, the dashed job on the second processor p consists of $(p, m, 7, 5, HC_p(7), [oldstate, \ldots, newstate])$, with m being the message received during the receive event $(p, m, 4)$. Note that neither the actual state transition times nor the actual sending times of the sent messages are recorded in a job. Measuring all message delays from the beginning of a job and knowing that the state transitions and the message sends occur in the listed order at arbitrary times during the job is sufficient for proving that a rt-run satisfies a given set of trace properties (see Section 4), as well as for performing time complexity analysis.

Clearly, not all sequences of receive events and jobs are valid real-time system runs. A rt-run of some algorithm $\mathcal{A}$ must satisfy the following properties:

- *Local Consistency:*
  - All state transitions and sent messages must be consistent with the transition function defined in $\mathcal{A}$. As in the classic computing model, the $oldstate$ of the first job on every processor must be some initial state $istate_p^{ru}$.
  - Hardware clocks must run at the same rate as real time, i.e., $HC(J) - time(J)$ must be the same for all jobs J on the same processor.
  - Jobs on the same processor must not overlap, i.e., there must not be two jobs J, J' with $proc(J) = proc(J')$ and $begin(J) \leq begin(J') < end(J)$.
  - If a timer message is sent for reception at time T, it

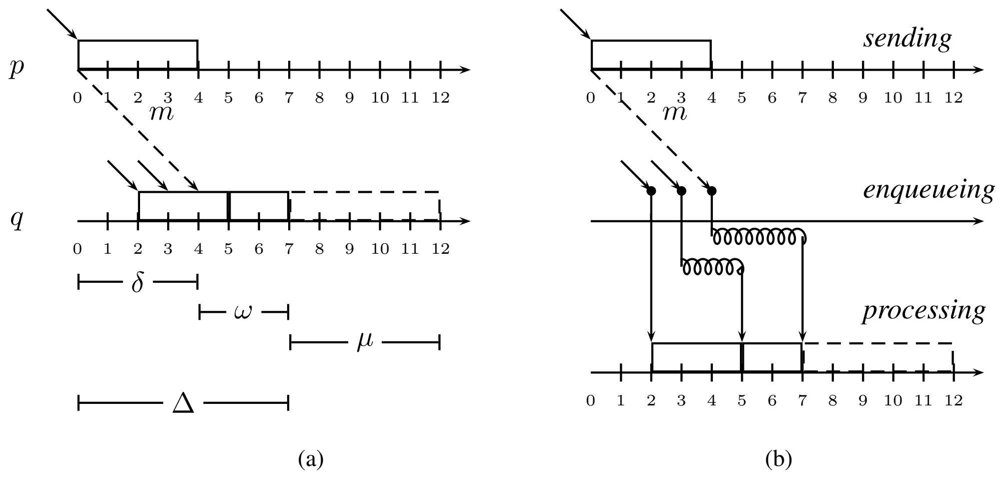


Figure 1: Real-time computing model: (a) Timing parameters for some message m, (b) Relation of message arrival and message processing, with enqueueing shown explicitly.

arrives when the hardware clock reads T, i.e., there is a receive event R with $HC_p(time(R)) = T$.[7]

- *Non-idling Scheduling:* Scheduling must be non-idling, i.e., as long as the queue is non-empty on some processor, there must always be a job executing on this processor. Of course, only a message from the queue (i.e., a message that has been received on that processor but has not been processed yet) can be chosen.
- *Global Consistency:* Every message is sent, received and processed exactly once (except for init messages). Receiving and processing must occur on the same processor.

Some processor p is *running* some job J at time t if $proc(J) = p$ and $begin(J) \leq t \leq end(J)$. Formally, a processor might run two jobs J, J' at the same time, if $end(J) = begin(J')$, or even more than two jobs in the (quite theoretical) setting where $\mu^- = 0$. This convention will be required for determining all the possible states the processor can be in at some time t.

A processor p is *busy* at time t if there is some job J such that $begin(J) \leq t < end(J)$. Otherwise, it is *idle*. Note that a processor can be running some job J but be idle at the same time t, if $t = end(J)$ and the queue is empty (or $\mu^- = 0$). We will use this definition to decide whether the processor enqueues an incoming message (*busy*) or immediately starts processing it (*idle*).

[7]A timer message sent during some job J for some time T smaller than $begin(J) + \mu^+$ arrives at $begin(J) + \mu^+$. This simplifying restriction makes the arrival of timer messages independent of the job duration and hence of the adversary.

3.4 Systems and Admissible Real-Time Runs

A real-time system s is defined by an integer n and two intervals $[\delta^-, \delta^+]$ and $[\mu^-, \mu^+]$.

Considering δ^-, δ^+, μ^- and μ^+ to be constants would give an unfair advantage to broadcast-based algorithms when comparing some algorithms' time complexity: Computation steps would take between μ^- and μ^+ time units, independently of the number of messages sent. This makes it impossible to derive a meaningful time complexity lower bound for systems in which a constant-time broadcast primitive is not available. Corollary 2 will show an example.

Therefore, the interval boundaries δ^-, δ^+, μ^- and μ^+ can be either constants or weakly monotonically increasing functions $\{0, \ldots, n-1\} \to \mathbb{R}^+$ from the number of destination processors[8] to which ordinary (= non-timer) messages are sent during that computing step to the actual message or processing delay bound.[9]

Example. During some job, messages to exactly three processors are sent. The duration of this job lies within $[\mu^-_{(3)}, \mu^+_{(3)}]$. Each of these messages has a message delay between $\delta^-_{(3)}$ and $\delta^+_{(3)}$. The delays of the three messages need not be the same.

To be useful, these functions must satisfy some condi-

[8]As message size is not bounded, we can make the simplifying assumption that at most one message is sent to every other processor during each job.

[9]Clearly, $\delta^-_{(0)}$ and $\delta^+_{(0)}$ are meaningless. They are assumed to be 0 because this allows some formulas to be written in a more concise form.

tions:

- Intervals must be well-defined: $\forall \ell : \delta^-_{(\ell)} \leq \delta^+_{(\ell)} \wedge \mu^-_{(\ell)} \leq \mu^+_{(\ell)}$
- Sending ℓ messages at once cannot be more costly than sending those messages in multiple steps. This is equivalent[10] to the requirement that sending ℓ messages at once cannot be more costly than sending those messages in two steps: $\forall i, j \geq 1 : f_{(i+j)} \leq f_{(i)} + f_{(j)}$ (for $f = \delta^-$, δ^+, μ^- and μ^+).

In addition, we assume that the message delay uncertainty $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ is also weakly montonically increasing and, therefore, $\varepsilon_{(1)}$ is the minimum uncertainty. This assumption is reasonable, as usually sending more messages increases the uncertainty rather than lowering it.

Definition 4. Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system. A rt-run is *s-admissible*, if the rt-run contains exactly n processors and satisfies the following timing properties: If ℓ is the number of messages sent during some job J,

- *Message Delay:* The message delay (measured from $begin(J)$ to the appropriate receive event) of every message in $sent(J)$ must be within $[\delta^-_{(\ell)}, \delta^+_{(\ell)}]$.
- *Job Duration:* The job duration $d(J)$ must be within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$.
- *Causality:* The ordering of receive events and jobs captures causality.

Similar to the classic computing model, *capturing causality* means that (a) if a job J' happens after job J on the same processor, $J < J'$, (b) if a job J sends a message received by some job J', $J < J'$, and (c) if a receive event R received some message which is processed by some job J, $R < J$. We also require that the rt-run, i.e., the sequence of receive events and jobs, is ordered by the occurence time of receive events and the begin times of jobs.

Similar to executions in the classic computing model, the creation of an s-admissible rt-run can be seen as a game of a player (the algorithm) against an adversary in the "arena" of a system s. The player provides sets of initial states and the state transition function, and the adversary can

- for every processor, choose an initial state from the set provided by the player, an initial hardware clock value and the time at which the init message will arrive,
- for every message, choose a value within $[\delta^-, \delta^+]$ representing the sum of
  - the time between the start of the job which sends the message and the actual sending time of the message, and
  - the actual transmission delay of the message (until the receive event occurs),
- for every job, choose a value within $[\mu^-, \mu^+]$ for its processing time and any associated overhead (scheduling etc.),
- define the scheduling policy [but see Section 4.4].

[10]Proof: $f(a+b+c) > f(a) + f(b) + f(c) \geq f(a) + f(b+c)$ contradicts $f(a + (b+c)) \leq f(a) + f(b+c)$.

3.5 Trajectories

In order to unify the definition of required properties/problems in the classic computing model and in the real-time computing model, we define local and global state trajectories for a rt-run analogous to Definitions 2 and 3:

Definition 5 (Real-time computing model state trajectories)**.** Processor p's *local state* at time t in a rt-run is defined by a function $state_p(t)$, called the *local state trajectory*, which must satisfy the following requirements:

- If the first job on p begins later than t, $state_p(t) = istate^{ru}_p$.
- If processor p runs one or more jobs $J_1, \ldots, J_k$ at time t, $state_p(t) \in \bigcup_{i=1}^k states(J_i)$.
- Otherwise, $state_p(t) = newstate(J)$ of the last job J running on p before t (= $oldstate(J')$ of the next job J' running on p after t, if such a job exists).
- The state changes inside a job must be causally consistent: Let $[istate^{ru}_p, \ldots, newstate(J')]$ be the concatenation of $states(J)$ from all jobs J running on p up to some time $t' = begin(J')$. The states in $state_p(t)$, $0 \leq t \leq t'$ must appear in exactly the same order, except that some states may be skipped in $state_p(t)$ as long as no other requirement is violated.

As in the classic computing model, there is usually no unique state trajectory.

Definition 6. A *global state trajectory* of some rt-run ru is a mapping $t \rightarrow (state_1(t), HC_1(t), \ldots, state_n(t), HC_n(t))$ with

- $state_p(t)$ being a valid local state trajectory for processor p in ru and
- the global state trajectory being *causally consistent* w.r.t. messages.

The second requirement above is only relevant if $\exists \ell : \delta^-_{(\ell)} \leq \mu^+_{(\ell)}$. Figure 2 shows an example: Processor p contains exactly one job with a state transition sequence $[oldstate, int.st., m, newstate]$ whereas q's job has a transition sequence of $[oldstate', newstate']$. This global

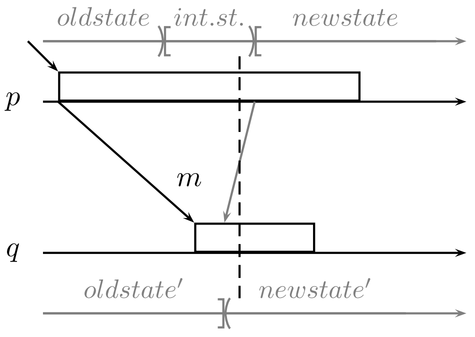


Figure 2: Example of an invalid global trajectory.

trajectory is invalid, because there is some time t (shown as a dashed vertical line), at which p is in a state before sending m, but q is in a state "knowing" about m: All states in a job J, except $oldstate(J)$, are of course assumed to know about the message which triggered the job. The fact that this global trajectory is invalid is illustrated by the gray arrow going backwards in time.

Figure 2 also demonstrates the two different concepts we use in the real-time computing model (and, in a simpler form, in the classic computing model): Rt-runs (shown black in the figure) provide a convenient tool for complexity analysis by modeling queuing effects and message delays in a simplified way by hiding the intricacies of *real* message transmissions and state changes. Trajectories (shown gray in the figure), on the other hand, incorporate additional assumptions we can safely make in a real system, such as the causal consistency of intermediate states and message transmissions, and explicity model state transitions, which is a necessity for proving state properties (see Section 4).

As in the classic computing model, $trajs^{ru}$ denotes the set of all valid global trajectories for some rt-run ru.

4 Problems, Algorithms and Proofs

This section defines what it means to prove that some algorithm solves some given problem in the systems defined above. A *problem* is represented as a set of *event properties*, which must be satisfied by the execution or rt-run, and a set of *state properties*, which must be satisfied by all possible *state trajectories* (see below) of the execution or rt-run. The following definitions will ensure that problems can be specified in the classic and in the real-time computing model in the same way.

4.1 Event Properties

Note that the notion of *events* introduced in Definition 7 has nothing to do with *receive events* in rt-runs.

Definition 7 (Events)**.** The *event* e corresponding to action ac or to job J is a 4-tuple, consisting of the processor index $proc(e) = proc(ac)/proc(J)$, the occurrence/start real-time $begin(e) = time(ac)/begin(J)$, the hardware clock value $HC(e) = HC(ac)/HC(J)$, and the old local state $oldstate(e) = oldstate(ac)/oldstate(J)$.

The motivation behind reducing the state of actions and jobs to $oldstate$ is that in the classic and in the real-time computing model it can be guaranteed that the state change to $oldstate(e)$ or some newer state occurs no later than $time(e)$. No such system parameter-independent guarantee can be given in the real-time computing model for the other states in $states(J)$. The restriction to $oldstate$ is done to avoid mistakes in this regard.

The *event trace* of some execution/rt-run is just the sequence of events corresponding to some relevant subset (the "external actions" in the notion of [15]) of the actions/jobs. A *trace property* is a set of event traces, usually characterized by a predicate E acting on event traces.

4.2 State Properties

Analogously, a *state property* is a set of global state trajectories (see Sections 2.4 and 3.5), characterized by a predicate P acting on global state trajectories.

Many properties can be specified as either event or state properties. It is up to the problem specification to choose which type is more appropriate.

4.3 Problems

A *problem* is a collection of trace properties and/or state properties. We say that an execution/rt-run *satisfies* a problem if all trace properties are satisfied in the (unique) corresponding event trace and all state properties are satisfied in all corresponding global state trajectories. What it means for an *algorithm* to *solve* a problem will be defined in the next subsection.

This paper deals with the terminating clock synchronization problem, where all clocks must be synchronized within some precision γ after termination. It is specified in Definition 8 using state properties only.

Definition 8 (Terminating Clock Synchronization Problem)**.** Let *adjusted time* at real-time t on processor p be defined as $HC_p(t) + state_p(t).adj$, where the local adjustment value adj can be changed by the processor (as it is part of the local state). Let $state_p(t).terminated$ be true when the algorithm has terminated at processor p by time t. The terminating clock synchronization problem with precision γ is specified by the following two state properties [14]:

- *Termination*
  $\exists t_{term} \forall t \geq t_{term}, \forall p : state_p(t).terminated = true$,
  i.e., all processors terminate by some finite real-time t_{term}.

- *Agreement*
  $\forall t : (\forall p : \mathit{state}_p(t).\mathit{terminated} = \mathit{true}) \Rightarrow \forall p, q : |HC_p(t) + \mathit{state}_p(t).\mathit{adj} - (HC_q(t) + \mathit{state}_q(t).\mathit{adj})| \leq \gamma$, i.e., after t_{term}, all processors have adjusted clocks within γ of each other.

If we assume infinite executions/rt-runs, i.e., that every processor continues to set timers and execute an infinite number of useless actions/jobs after termination, the terminating clock synchronization problem might as well be specified by trace properties:

Definition 9.

- *Termination*
  $\exists e_{term} : \forall e \geq e_{term} : \mathit{oldstate}(e).\mathit{terminated} = \mathit{true}$
- *Agreement*
  $\forall e_1, e_2 : (e_1 < e_2 \wedge \forall e \geq e_1 : \mathit{oldstate}(e).\mathit{terminated} = \mathit{true}) \Rightarrow |HC(e_1) + \mathit{oldstate}(e_1).\mathit{adj} - \mathit{time}(e_1) - (HC(e_2) + \mathit{oldstate}(e_2).\mathit{adj} - \mathit{time}(e_2))| \leq \gamma$.

Definition 9 requires that after termination the offsets from real time (= adjusted time $-$ real time) of two events are within γ. Although this definition is not semantically equivalent to Definition 8, it boils down to solving the same real-world problem.

4.4 Proofs

The notion of admissible executions/admissible rt-runs can be used to prove that some algorithm solves some problem $\mathcal{P}$. In the classic computing model, we can define correctness and impossibility in the usual way:

Definition 10 (Correctness)**.** An algorithm $\underline{\mathcal{A}}$ solves some problem $\mathcal{P}$ in some system $\underline{s}$ if, and only if, for every $\underline{s}$-admissible execution ex of $\underline{\mathcal{A}}$, ex satisfies $\mathcal{P}$.

Definition 11 (Impossibility)**.** A problem $\mathcal{P}$ is impossible to solve in some system $\underline{s}$ if, and only if, for every algorithm $\underline{\mathcal{A}}$ there exists an $\underline{s}$-admissible execution ex of $\underline{\mathcal{A}}$ violating $\mathcal{P}$.

The following definitions for the real-time computing model are completely analogous:

Definition 12 (Strong Correctness)**.** An algorithm $\mathcal{A}$ solves some problem $\mathcal{P}$ in some system s if, and only if, for every s-admissible rt-run ru of $\mathcal{A}$, ru satisfies $\mathcal{P}$.

Definition 13 (Weak Impossibility)**.** A problem $\mathcal{P}$ is impossible to solve in some system s if, and only if, for every algorithm $\mathcal{A}$ there exists an s-admissible rt-run ru of $\mathcal{A}$ violating $\mathcal{P}$.

The observant reader will have noticed that, in the real-time computing model of Section 3, the scheduling policies are *adversary-controlled*, meaning that, in the game between player and adversary, first the player chooses the algorithm and afterwards the adversary can choose the scheduling policy which is most unsuitable for the algorithm. Thus, correctness proofs are "strong" (as the algorithm can defend itself against the most vicious scheduling policy), but impossibility proofs are "weak" (because the adversary has the scheduling policy on its side).

However, sometimes algorithms are designed for particular, a-priori-known scheduling policies. To capture this notion of *algorithm-controlled* scheduling policies, we introduce the following definitions:

Definition 14 (Weak Correctness)**.** A pair (algorithm $\mathcal{A}$, scheduling policy pol) solves some problem $\mathcal{P}$ in some system s if, and only if, for every s-admissible rt-run ru of $\mathcal{A}$ conforming to pol, ru satisfies $\mathcal{P}$.

Definition 15 (Strong Impossibility)**.** A problem $\mathcal{P}$ is impossible to solve in some system s if, and only if, for every pair (algorithm $\mathcal{A}$, scheduling policy pol) there exists an s-admissible rt-run ru of $\mathcal{A}$ conforming to pol that violates $\mathcal{P}$.

All proofs in this paper show either strong correctness or strong impossibility for the real-time computing model.

4.5 Shifting

A common technique in the classic computing model for proving lower bounds for the clock synchronization problem is *shifting*. Shifting an execution ex of n processors by $(x_0, \ldots, x_{n-1})$ results in another execution ex', where

- actions on processor p_i happening at real-time t in ex happen at real-time $t - x_i$ in ex',
- the hardware clock of processor p_i is shifted such that all actions still have the same hardware clock reading as before, i.e. $HC'_{p_i}(t) := HC_{p_i}(t) + x_i$,

Note that this new execution might not be admissible, as messages could be received before they are sent.

The same technique can be applied to the real-time computing model: Shifting a rt-run ru of n processors by $(x_0, \ldots, x_{n-1})$ results in another rt-run ru', where

- receive events and jobs on processor p_i starting at real-time t in ru start at real-time $t - x_i$ in ru',
- the hardware clock of p_i is shifted such that all receive events and jobs still have the same hardware clock reading as before, i.e. $HC'_i(t) := HC_i(t) + x_i$.

Note that ru' is a valid rt-run, as the hardware clock readings of the receive events and the jobs do not change, and, therefore, consistency and scheduling properties are not violated. However, ru' might not be admissible as the message delay might have changed excessively. Moreover, the set of global state trajectories might be empty, since causal consistency may be impossible to achieve in ru'.

We assume that, just like in an admissible rt-run, the receive events and jobs in a shifted rt-run are ordered by their occurence time and begin time, respectively. Apart from that, the reordering must preserve the original ordering as much as possible, so that if the original rt-run captured causality (e.g. by being admissible), the shifted rt-run still captures causality[11], unless this is no longer possible (e.g. if messages travel backwards in time in the shifted rt-run).

[11]This is only relevant if at least two receive events/jobs occur/start at the same time in the shifted rt-run.

4.6 Notation for Specifying Algorithms

Recall that, in both system models, an action/a job consists of receiving a message (either from the messaging subsystem or from the queue), reading the hardware clock, performing state transitions and sending messages. Thus, the transition function and the initial state of some algorithm $\mathcal{A}$ can be thought of as a set of global variables (including their initial values) and some function $\mathcal{A}$-*process_message(msg, time)* carrying out the state transitions and sending the messages. msg contains the message to be processed and $time$ contains the hardware clock reading at the beginning of this action/job. If it is not obvious from the code, an informal description is given as to which operations are atomic, i.e., without an intermediate state, and which are not.

4.7 Time Complexity

The time complexity of some algorithm will be measured as the worst-case difference of the real time of arrival of the last init message to the real time when the last processor has terminated.

5 Transformations

In this section, we will show that the classic computing model and the real-time computing model are fairly equivalent from the perspective of solvability of problems: A real-time system can simulate some particular classic system (and vice versa), and conditions for transforming a classic computing model algorithm $\underline{\mathcal{A}}$ into a real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ (and vice versa) do exist. As a consequence, certain impossibility and lower bound results can also be translated.

One direction (Section 5.3), simulating a real-time system $(n, [\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+], [\mu^- = \mu, \mu^+ = \mu])$ on top of a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$, is quite straightforward: It suffices to implement an artifical processing delay μ, the queueing of messages arriving during such a simulated job, and the scheduling policy. This simulation allows to run any real-time computing model algorithm $\mathcal{A}$ designed for a system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^- \leq \underline{\delta}^-$, $\delta^+ \geq \underline{\delta}^+$, and $\mu^- \leq \mu \leq \mu^+$ on top of it, thereby resulting in a correct classic computing model algorithm.

The other direction (Section 5.2), simulating a classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$, is more tricky: The class of classic computing model algorithms $\underline{\mathcal{A}}$ that can be transformed into a real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ must be restricted (timer-free algorithms) and, because of the uncertainty regarding when a job's state transition is actually performed, the transformed algorithm solves a slightly different problem than the original algorithm. Second, and more importantly, a *real-time scheduling analysis* must be conducted in order to break the circular dependency of algorithm $\underline{\mathcal{A}}$ and simulated end-to-end delays $\Delta \in [\Delta^-, \Delta^+]$, which is normally "hidden" in the classic computing model, but pops up when one tries to instantiate this model in a real system.

5.1 Problem Transformations

All the transformations introduced in this paper will guarantee an identical sequence of events in the original algorithm and the simulated algorithm (w.r.t. Definition 7 and the problem-relevant subset of the state). It follows that any simulated rt-run corresponding to an execution (and vice versa) is equivalent w.r.t. any trace property. Consequently, our transformations inherently preserve all trace properties.

Unfortunately, this is not always the case for state properties: $state_p(t)$ in the simulated execution is usually not the same as $state_p(t)$ in the corresponding rt-run, since the actual state transitions occur at (slightly) different times. Consider the terminating clock synchronization problem given in Definition 8, for example: Whereas $state_p(t).terminated = true$ holds immediately after the terminating (zero-time) action has occurred in the classic computing model, it can only be guaranteed to hold for sure after the *end* of the terminating job in case of the real-time computing model. That is, the termination real-time t_{term} is different here. Nevertheless, we have defined the termination and agreement properties in Definition 8 in a way, which is not affected by this uncertainty: The condition $\forall p : state_p(t).terminated = true$ used as a guard in the agreement property becomes true only after the terminal state transitions have indeed occurred.

In order to extend our transformations to other problems, however, we need a generic way to ensure that all state properties hold true both in the simulated execution and in the corresponding rt-run. We will hence restrict our attention to a restricted class of problems, called *compatible problems*, which are specified in Definition 16. Basically, for a problem to be compatible, predicates corresponding to state properties must be invariant with respect to any shuffling of local states with previous states. If a state predicate satisfies this property, then it is not difficult to show that our transformations preserve the corresponding state property.

Definition 16 (Compatible Problem)**.** A problem is a *compatible problem*, if the predicate P for any of its state properties satisfies the following requirements:

- If P holds for some global state trajectory, then it continues to hold when an arbitrary number of local state trajectories $state_p(t), state_q(t), \dots$ in the global state trajectory are replaced by $state_p(t - x_p), state_q(t - x_q), \dots$ for arbitrary constants $x_p, x_q, \dots$.
- P is either a simple invariant ($\forall t : P'(state_1(t), HC_1(t), \dots)$) or an eventually holding invariant ($\exists t' : \forall t > t' : P'(state_1(t), HC_1(t), \dots)$). The domain of P' is a list of n states and hardware clock values rather than a function $t \to (state_1(t), HC_1(t), \dots)$.

Note that this transformation of trajectories is *not* analogous to shifting of executions/rt-runs as described in Section 4.5, as the hardware clock values are not modified this time. Intuitively, the second requirement is necessary because some more complex predicates might hold in all executions but not in rt-runs with $\mu^- > 0$. Take, for example, a simple state transition sequence $[oldstate, int.st._0, int.st._1, newstate]$, which occurs exactly once as an action/job in an execution/rt-run on processor p. In the execution, the state property $(\exists t : state_p(t) = int.st._0) \Rightarrow (\forall t : state_p(t) \neq int.st._1)$ holds for all global state trajectories, but this is not true for the rt-run.

The terminating clock synchronization problem is a compatible problem, for example, since there is an implicit third property in Definition 8: $state_p(t).adj$ does not change after $state_p(t).terminated$ becomes true. Hence, $state_p(t).adj = state_p(t - x_p).adj$ if $state_p(t - x_p).terminated = true$, which is secured by the termination validity condition.

5.2 Reusing Classic Computing Model Algorithms

In this section, we will show how to simulate a classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$, thereby providing a transformation of a classic computing model algorithm $\underline{\mathcal{A}}$ solving some compatible problem $\mathcal{P}$ into a real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ solving $\mathcal{P}$.

The major problem here is the circular dependency of the algorithm $\underline{\mathcal{A}}$ on the real end-to-end delays and vice versa: On one hand, the classic computing model algorithm $\underline{\mathcal{A}}$ run atop of the simulation might need to know the *simulated* message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, which are just the end-to-end delay bounds $[\Delta^-, \Delta^+]$ of the unterlying simulation. Those end-to-end delays, on the other hand, involve the queuing delay ω and are thus dependent on (the message pattern of) $\underline{\mathcal{A}}$ and hence on $[\underline{\delta}^-, \underline{\delta}^+]$.

This circular dependency can be broken as follows: Given some classic computing model algorithm $\underline{\mathcal{A}}$ with assumed message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, considered as unvalued parameters, a real-time scheduling analysis of the combined algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ must be conducted. This provides an equation for the resulting end-to-end delay bounds $[\Delta^-, \Delta^+]$ in terms of the real-time systems parameters $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ *and* the algorithm parameters $[\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+]$, i.e., a function F satisfying

$$[\Delta^-, \Delta^+] = F\big(n, [\delta^-, \delta^+], [\mu^-, \mu^+], [\Delta^-, \Delta^+]\big). \quad (1)$$

We do not want to embark on the intricacies of advanced real-time scheduling analysis techniques here, see [22] for an overview. For the purpose of this paper, quite trivial considerations are sufficient: A trivial end-to-end delay lower bound Δ^- is δ^-, as the first message is never queued. An upper bound Δ^+ can be obtained easily if, for example, there is an upper bound on the number of messages a processor receives in total, see Section 5.4 for a particular example.

Anyway, if eq. (1) provided by the real-time scheduling analysis can be solved for $[\Delta^-, \Delta^+]$, resulting in meaningful bounds $\Delta^- \leq \Delta^+$, they can be assigned to the algorithm parameters $[\underline{\delta}^-, \underline{\delta}^+]$. We will call such an assignment *feasible*. Any feasible assignment of $[\underline{\delta}^-, \underline{\delta}^+]$ results in a correct implementation of the real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$, since it ensures that both $\underline{\mathcal{A}}$ and the end-to-end delays are within their specifications.

We should mention, however, that the adversary faced by $\underline{\mathcal{A}}$ when employed in $\mathcal{S}_{\underline{\mathcal{A}}}$ is somewhat restricted: The unrestricted adversary in the classic computing model can choose any value between $\underline{\delta}^-$ and $\underline{\delta}^+$, for every message, whereas a large part of the end-to-end delay in the simulated setting is determined by the queue state (i.e., the message pattern). It cannot hence be chosen arbitrarily between $\underline{\delta}^-$ and $\underline{\delta}^+$ by the adversary.

In the remainder of this section, we will show that any timer-free[12] algorithm designed for some classic system $\underline{s}$

[12]Intuitively, this restriction is needed because in the real-time computing model we cannot guarantee that a timer set for some hardware clock time will be processed by that time (the system might be busy). This problem does not show up in the classic computing model where computing steps take zero time.

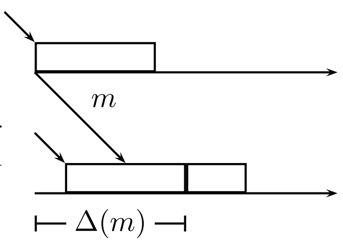
solving some compatible problem $\mathcal{P}$ can also solve that problem in s if a feasible assignment for $[\underline{\delta}^-, \underline{\delta}^+]$ exists.

The key to this transformation is a very simple simulation: Recall that an algorithm is specified as a mapping from processor indices to a set of initial states and a transition function, and that the transition function is defined identically for the classic and the real-time computing model. Let $\mathcal{S}_{\underline{A}}$ be an algorithm for the real-time computing model, comprising exactly the same initial states and transition function as a given classic computing model algorithm $\underline{\mathcal{A}}$. From a more practical point of view, $\mathcal{S}_{\underline{A}}$ can be expressed as given in Figure 3. Figure 4 outlines the principle of our simulation.

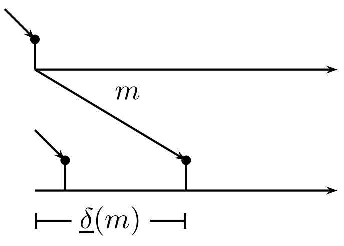
```
< global variables of $\underline{\mathcal{A}}$>

**function** $\mathcal{S}_{\underline{A}}$−process_message(msg, time)
  $\underline{\mathcal{A}}$−process_message(msg, time)
```

Figure 3: Simulation algorithm (classic computing model atop of real-time computing model)

For each s-admissible rt-run ru of $\mathcal{S}_{\underline{A}}$:



- create a corresponding $\underline{s}$-admissible execution ex of $\underline{\mathcal{A}}$ (satisfying $\mathcal{P}$).



- ru and ex have the same sequence of events $\Rightarrow$ ru satisfies trace properties of $\mathcal{P}$.
- $\mathcal{P}$ is a compatible problem and $state_p^{ru}(t)$ equals some $state_p^{ex}(t - x_p), 0 \le x_p \le \mu^+$. $\Rightarrow ru$ satisfies state properties of $\mathcal{P}$.

$\Rightarrow \mathcal{S}_{\underline{A}}$ solves $\mathcal{P}$ in s.

Figure 4: Transformation outline (Theorem 1)

Theorem 1. *Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system and $\mathcal{P}$ be a compatible problem. If there exists a timer-free algorithm $\underline{\mathcal{A}}$ for solving $\mathcal{P}$ in some classic system $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ with a feasible assignment, then $\mathcal{S}_{\underline{A}}$ given in Figure 3 solves $\mathcal{P}$ in s.*

Proof. Assume such an algorithm $\underline{\mathcal{A}}$ exists. Let Δ^- and Δ^+ be the minimum and maximum end-to-end delay of all messages of all rt-runs of $\mathcal{S}_{\underline{A}}$ in s, which satisfy $\Delta^- \ge \underline{\delta}^-$ and $\Delta^+ \le \underline{\delta}^+$ since the assignment is feasible. We will now show that the implication ($\mathcal{S}_{\underline{A}}$ solves $\mathcal{P}$ in s) is justified. According to Definition 12 we have to show that every s-admissible rt-run of $\mathcal{S}_{\underline{A}}$ satisfies $\mathcal{P}$.

Corresponding execution: Let ru be such an s-admissible rt-run. We can now create a corresponding execution ex of $\underline{\mathcal{A}}$ in $\underline{s}$ by mapping each job J on to an action ac in ex:

$$\begin{array}{ll} proc(ac) \leftarrow proc(J) & HC(ac) \leftarrow HC(J) \\ msg(ac) \leftarrow msg(J) & \\ time(ac) \leftarrow begin(J) & trans(ac) \leftarrow trans(J) \end{array}$$

Receive events are ignored. As start times of jobs are mapped to occurence times of actions and $HC(ac) = HC(J)$, the corresponding hardware clock readings in both systems are equal. As the actions in ex appear in the same order, process the same messages and read the same hardware clock values as the jobs in ru, they also perform the same state transitions (by design of the simulation algorithm) and send the same messages. Thus, ex is a valid execution: All requirements specified in Section 2.2 are met by ex (except for the requirement regarding timer messages, which is why we restricted this transformation to timer-free algorithms).

ex is $\underline{s}$-admissible: By induction, we can show that ex is $\underline{s}$-admissible: Let $ex(i)$ be the finite prefix of ex containing the first i actions. Trivially, $ex(1)$ containing the first message starting up the system is $\underline{s}$-admissible (as an init message, it is exempt from the requirement of having been sent and, thus, needs not to obey any delay bounds). Let $ex(i-1)$ be $\underline{s}$-admissible. The i-th action is caused by some message m received at real time t and corresponds to some job J in ru starting at the same real time and processing the same message. If m is an init message, it does not need to obey any bounds. If m is a regular message, we can exploit that fact that in s end-to-end delay bounds $[\Delta^-, \Delta^+]$ hold: This implies that m was sent in ru by some job starting at some real time within $[t - \Delta^+, t - \Delta^-]$. Thus, in ex, m was sent by some action occuring at some real time within $[t-\Delta^+, t-\Delta^-]$. Therefore, m's message delay in $\underline{s}$ ranges between $\Delta^- \ge \underline{\delta}^-$

and $\Delta^+ \leq \underline{\delta}^+$. As m obeys the $[\underline{\delta}^-, \underline{\delta}^+]$ bounds, $ex(i)$ is $\underline{s}$-admissible.

As $\underline{\mathcal{A}}$ is an algorithm solving $\mathcal{P}$ in $\underline{s}$ and ex is $\underline{s}$-admissible, ex satisfies $\mathcal{P}$.

ru satisfies trace properties of $\mathcal{P}$: As the sequence of events on each individual processor remains the same during the transformation, all trace properties that were satisified in ex are also satisfied in ru.

ru satisfies state properties of $\mathcal{P}$: Let $t \rightarrow (state_1^{ru}(t), \dots)$ be a global state trajectory and P be a state predicate on $trajs^{ru}$.

Case 1: $P \equiv \forall t' : P'(state_1^{run}(t'), \dots)$ As the predicate must hold for all global state trajectories of ex, we just have to show that, for every t, this vector of concrete states $(state_1^{run}(t), \dots)$ also exists at some time during any of ex's global trajectories. Fix some t'. We define $state_p^{ex}(t)$ as follows:

1. If p is not running any job at time t' in ru, let $state_p^{ex}(t)$ be any local state trajectory of p in ex. Clearly, $state_p^{ru}(t') = state_p^{ex}(t') =$ the $newstate$ of the last action/job before t' (or $istate_p^{ru}(= istate_p^{ex})$).
2. If p is running one or more jobs at time t', let J be one of these jobs with $state_p^{ru}(t') \in states(J)$. Let $state_p^{ex}(t)$ be any local state trajectory of p in ex in which $state_p^{ex}(begin(J)) = state_p^{ru}(t')$ This is a valid condition as $begin(J) = time(ac)$ of the action ac corresponding to job J and $trans(J) = trans(ac)$. Thus, $state_p^{ru}(t') = state_p^{ex}(t' - x_p)$, $0 \leq x_p \leq \mu^+_{(n-1)}$. As $\mathcal{P}$ is a compatible problem, this suffices.

Now all that is left is to show that it is possible to choose the $state_p^{ex}$ in accordance with the requirements just stated such that $t \rightarrow (state_1^{ex}(t), \dots)$ is a valid global state trajectory of ex. Assume by contradiction that this global state trajectory contains some causal inconsistency which cannot be fixed without violating the above requirements. Let m be the causal inconsistent message, sent by p at time t and received by q at time t, with $state_p(t)$ containing some state before sending the message and $state_q(t)$ containing some state after receiving the message. This implies that in ru there is a job on processor p starting at time t sending a message to processor q, which is received and starts processing also at time t. If it is impossible to change $state_p^{ex}(t)$ to some state after sending m without violating the second requirement specified above, $state_p^{ru}(t)$ must be in some state before sending m as well. Similarly, if it is impossible to change $state_q^{ex}(t)$ to some state before receiving m, $state_q^{ru}(t)$ must be in a state after receiving m as well. This contradicts the assumption that $t \rightarrow (state_1^{ru}(t), \dots)$ is a valid global state trajectory for ru.

Case 2: $P \equiv \exists t'' : \forall t' > t'' : P'(state_1^{run}(t'), \dots)$ As the predicate must hold for all global state trajectories of ex, let t^* be $\max(t'')$ over those trajectories. For the global state trajectory of ru, choose some $t'' > t^* + \mu^+_{(n-1)}$. We have already shown that invariants can be "mapped" onto a time window of $[t' - \mu^+_{(n-1)}, t']$ of a global state trajectory of ex, so the rest of the proof is completely analogous to the first case.

As $\mathcal{P}$ is a compatible problem, this suffices to show that ru satisfies all state properties that were satisfied in ex. Thus, we can conclude that $\mathcal{S}_{\underline{\mathcal{A}}}$ solves $\mathcal{P}$ in s. $\square$

5.3 Reusing Real-Time Computing Model Algorithms

As the real-time computing model is a generalization of the classic computing model, the set of systems covered by the classic computing model is a (strict) subset of the systems covered by the real-time computing model. More precisely, every system in the classic computing model $(n, [\underline{\delta}^-, \underline{\delta}^+])$ can be specified in terms of the real-time computing model $(n, [\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+], [\mu^- = 0, \mu^+ = 0])$. Thus, every result (correctness or impossibility) for some classic system also holds in the corresponding real-time system with the same message delay bounds and $\mu^-_{(\ell)} = \mu^+_{(\ell)} = 0$ for all ℓ.

Intuition tells us that impossibility results also hold for the general case, i.e., that an impossibility for some classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ holds for all real-time systems $(n, [\delta^- \leq \underline{\delta}^-, \delta^+ \geq \underline{\delta}^+], [\mu^-, \mu^+])$ for arbitrary μ^-, μ^+, because the additional delay does not provide the algorithm with any useful information. We will prove this by using yet another simulation, this time the other way round. Note that, contrary to the previous section, we also allow timer messages this time and do not restrict ourselves to compatible problems.

Figure 5 provides an algorithm $\underline{\mathcal{S}}_{\underline{\mu}, \mathcal{A}}$ designed for the classic computing model, which allows us to simulate a real-time system, and, thus, to use an algorithm $\mathcal{A}$ designed for the real-time computing model to solve problems in a classic system. The algorithm essentially simulates queueing, scheduling, and execution of real-time model computing steps (jobs) of duration $\underline{\mu}$, and can hence be parameterized with some function $\underline{\mu} : \{0, \dots, n-1\} \rightarrow \mathbb{R}^+$ and some real-time computing model algorithm $\mathcal{A}$. It works as follows: At every point in time, the processor is either *idle* (local variable $idle = true$) or *busy* ($idle = false$). Initially, the processor is idle. As soon as the first message arrives, the processor becomes busy and waits for $\underline{\mu}_{(\ell)}$ time units (ℓ being the number of ordinary messages sent during that computing step). All messages arriving during that time

```
var queue ← {}
var idle ← $true$
< global variables of $\mathcal{A}$>

function $\underline{\mathcal{S}}_{\mu,\mathcal{A}}$−process_message(msg, time)
  if msg = "finished-processing"
    next(time)
  else
    queue.add(msg)
    if idle
      next(time)

function next(time)
  if queue is empty
    idle ← $true$
  else
    var msg ← choose item from queue according
          to some arbitrary scheduling policy
    queue.remove(msg)
    $\mathcal{A}$−process_message(msg, time)
    idle ← $false$
    $\ell$ ← number of ordinary messages sent by $\mathcal{A}$
    set timer "finished-processing" for time + $\underline{\mu}_{(\ell)}$
```

Figure 5: Simulation algorithm (real-time computing model atop of classic computing model)

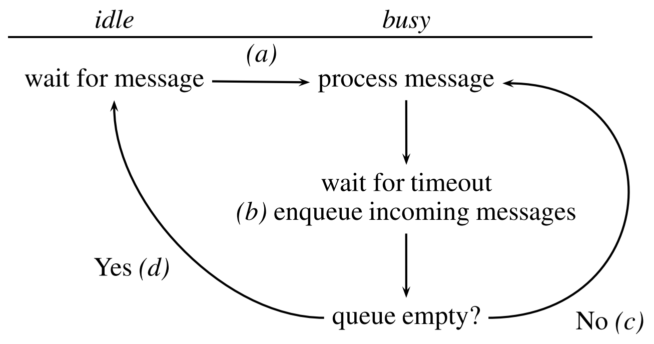
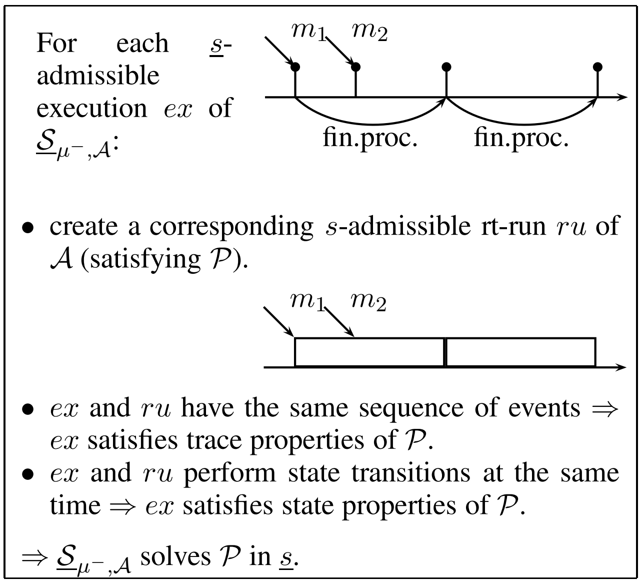


Figure 6: State diagram (algorithm in Figure 5)



For each $\underline{s}$-admissible execution ex of $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$:

- create a corresponding s-admissible rt-run ru of $\mathcal{A}$ (satisfying $\mathcal{P}$).

- ex and ru have the same sequence of events $\Rightarrow$ ex satisfies trace properties of $\mathcal{P}$.
- ex and ru perform state transitions at the same time $\Rightarrow$ ex satisfies state properties of $\mathcal{P}$.

$\Rightarrow \underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ solves $\mathcal{P}$ in $\underline{s}$.

Figure 7: Transformation outline (Theorem 2)

are enqueued. After these $\underline{\mu}_{(\ell)}$ time units have passed, the queue is checked. If it is empty, the processor returns to its idle state; otherwise, the next message is processed. The resulting state diagram is shown in Figure 6; Figure 7 outlines the principle of the simulation.

Theorem 2. *Let* $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ *be a classic system and* $\mathcal{P}$ *be a problem. If there exists an algorithm* $\mathcal{A}$ *for solving* $\mathcal{P}$ *in some real-time system* $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ *with*

- $\delta^-_{(\ell)} \le \underline{\delta}^-$ *and* $\forall \ell : \delta^+_{(\ell)} \ge \underline{\delta}^+$,

$\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ *given in Figure 5 solves* $\mathcal{P}$ *in* $\underline{s}$.

Proof. Assume such an algorithm $\mathcal{A}$ exists. We will show that the implication ($\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ solves $\mathcal{P}$ in $\underline{s}$) is justified. According to Definition 10 we have to show that every $\underline{s}$-admissible execution of $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ satisfies $\mathcal{P}$.

Let ex be such an $\underline{s}$-admissible execution of $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ in $\underline{s}$. Note that there are four kinds of actions in ex (cf. Figure 6): (a) algorithm message arriving, which is immediately processed, (b) algorithm message arriving which is enqueued, (c) "finished-processing" timer message arriving, causing some message from the queue to be processed, (d) "finished-processing" timer message arriving when no messages are in the queue. Every type (c) action has a corresponding type (b) action where the algorithm message being processed in (c) is enqueued.

We can now construct a rt-run ru in s with the same initial hardware clock values as ex. Let $trans^*(ac)$ contain $trans(ac)$ without (1) the simulation algorithm variables, (2) state transitions only involving simulation variables and (3) any "finished-processing" messages. Depending on the type of action, a corresponding receive event or job in ru is constructed for each action ac:

- Type (a): This action is mapped to a receive event R and a job J in ru. Let ℓ be the number of ordinary messages sent during ac:

  $proc(R) \leftarrow proc(ac)$
  $msg(R) \leftarrow msg(ac)$
  $time(R) \leftarrow time(ac)$
  $proc(J) \leftarrow proc(ac)$

$msg(J) \leftarrow msg(ac)$
$begin(J) \leftarrow time(ac)$
$d(J) \leftarrow \underline{\mu}_{(\ell)}$
$HC(J) \leftarrow HC(ac)$
$trans(J) \leftarrow trans^*(ac)$

- Type (b): This action is mapped to a receive event R in ru:

$proc(R) \leftarrow proc(ac)$ $\quad time(R) \leftarrow time(ac)$
$msg(R) \leftarrow msg(ac)$

- Type (c): This action is mapped to a job J in ru. Let msg be the algorithm message of the corresponding type (b) action and ℓ be the number of ordinary messages sent during ac.

$proc(J) \leftarrow proc(ac)$ $\quad d(J) \leftarrow \underline{\mu}_{(\ell)}$
$msg(J) \leftarrow msg$ $\quad HC(J) \leftarrow HC(ac)$
$begin(J) \leftarrow time(ac)$ $\quad trans(J) \leftarrow trans^*(ac)$

- Type (d): This action is not transferred to ru.

To illustrate this transformation, Figure 7 shows an example with actions of types (a), (b), (c) and (d) occuring in ex (in this order) and the resulting rt-run ru.

The following Lemmas 1–3 prove some useful invariants in ex:

Lemma 1. *Initially and directly after executing some action ac, the processor is in one of two well-defined states:*

1. $newstate(ac).idle = true$, $newstate(ac).queue = \{\}$*, there is no "finished-processing" timer message to p in* $intransit\_msgs(ac)$,
2. $newstate(ac).idle = false$*, there is exactly one "finished-processing" timer message to p in* $intransit\_msgs(ac)$.

Proof. By induction. Initially (replace $newstate(ac)$ with $istate_p^{ex}$ and $intransit\_msgs(ac)$ with $intransit\_msgs(t = 0)$), every processor is in state 1. If a message is received while the processor is in state 1, it is added to the queue, processed, $idle$ is set to $false$ and a "finished-processing" timer message is sent, i.e., the processor switches to state 2 [type (a) action]. If a message is received during state 2, one of two things can happen:

- The message is a "finished-processing" timer message: If the queue is empty, the processor switches to state 1 [type (d) action]. Otherwise, a new "finished-processing" timer message is generated. Thus, the processor stays in state 2 [type (c) action].
- The message is an algorithm message: The message is added to the queue and the processor stays in state 2 [type (b) action]. □

Lemma 2. *After a type (a) or (c) action sending ℓ ordinary messages occured at time t on processor p in ex, the next type (a), (c) or (d) action on p can occur no earlier than $t + \mu^-_{(\ell)}$.*

Proof. The "finished-processing" timer message sent by action ac of type (a) or (c) arrives no earlier than $t+\underline{\mu}_{(\ell)} = t+\mu^-_{(\ell)}$. As $newstate(ac).idle = false$ and this variable can only be changed by arrival of a "finished-processing" timer message, all other incoming messages during that time are enqueued, i.e., only type (b) actions can occur. □

Lemma 3. *For every processor p it holds that p is idle in ru at some time t (cf. Section 3.3) only if the last action ac on p in ex with $time(ac) \leq t$ had $newstate(ac).idle = true$.*

Proof. First, note that all jobs sending ℓ ordinary messages in ru have a duration of $\underline{\mu}_{(\ell)} = \mu^-_{(\ell)}$. Assume that some processor is idle in ru at time t although the last action ac on p in ex with $time(ac) \leq t$ had $newstate(ac).idle = false$. According to Lemma 1, the processor must then be in state 2, which means that one "finished-processing" timer message to p is in $intransit\_msgs(ac)$. As ac is the last action on p with $time(ac) \leq t$, this "finished-processing" message has not been received and processed yet by time t. As such a message is only sent during a type (a) or (c) action, let ac' be the last type (a) or (c) action on p before or at t. Let ℓ be the number of ordinary messages sent by ac'. As the "finished-processing" message has not been received yet by time t and, by design of the algorithm, there are exactly $\underline{\mu}_{(\ell)}$ time units between the action sending and the action receiving it, $time(ac')$ must be greater than $t-\underline{\mu}_{(\ell)}$. However, this implies that a job J' on p sending ℓ ordinary messages with $t-\underline{\mu}_{(\ell)} < begin(J') \leq t$ exists in ru, contradicting the assumption that the processor is idle in ru at time t. □

The next two lemmas will show that the constructed rt-run satisfies all the basic properties of an rt-run and is s-admissible:

Lemma 4. *ru is a valid rt-run.*

Proof. We will show the properties defined in Section 3.3 to be satisfied:

- *Local Consistency:*
  - We map only those actions from ex to jobs in ru where some algorithmic state transition is performed, i.e., where $\mathcal{A}$-process_message is called. By the design of the simulation algorithm and by the construction of ru, these calls occur at the same real time in ex and ru, and the same parameters (msg, $time$) are passed. Thus, $\mathcal{A}$'s transition function will yield the same result, which is consistent with the fact that the resulting action/job states in ex and ru (excluding the simulation variables $queue$ and $idle$) are equal.
  - Jobs start at the same time as the corresponding actions ($begin(J) = time(ac)$), and hardware clock readings are the same ($HC(J) = HC(ac)$). Thus, hardware clocks still run at the same rate.
  - Assume for a contradiction that two jobs J, J' overlap, i.e. their starting times are closer together than $\underline{\mu}_{(\ell)}$ (ℓ being the number of ordinary messages sent by J). The corresponding actions in ex must have been type (a) or (c) actions occuring at the same time as the start times of J and J'. This, however, contradicts Lemma 2.
  - Timer messages in ex sent for some hardware clock value T on some processor p cause a type (a) or (b) action at $t = HC_p^{-1}(T)$. As both types of action are mapped to receive events at t, and the hardware clocks are the same in ru and ex, timer messages arrive at the correct time in ru.
- *Non-idling Scheduling:* Let J be some job on processor p starting at time t_p processing some message msg received at time t_r. By design of the simulation algorithm, the action ac receiving msg at t_r had $newstate(ac).idle = false$. Assume for a contradiction that the processor has been idle at some time $t, t_r \leq t < t_p$. According to Lemma 3, this means that the last action ac' before or at t on p in ex had $newstate(ac').idle = true$. However, $idle$ is only set to $true$ if the queue is empty. As msg is added to the queue no later than t_r and leaves the queue no earlier than t_p, this is a contradiction.
- *Global Consistency:* Enqueuing a message (type (b)) corresponds to a receive event and removing a message from the queue (type (c)) corresponds to a job. As a type (c) action is always preceded by a corresponding type (b) action, a job processing a message is always preceded by a receive event receiving that message. Of course, type (a) actions map to both a receive event and a corresponding job; hence, they also satisfy this condition. □

Lemma 5. *ru is an s-admissible rt-run.*

Proof.

- *Message Delay:* All actions that receive algorithm messages (types (a) and (b)) are mapped to receive events occuring at the same real time. All actions sending messages (types (a) and (c)) are mapped to jobs starting at the same real time. Since $\delta^-_{(\ell)} \leq \underline{\delta}^-$ and $\delta^+_{(\ell)} \geq \underline{\delta}^+$ for all ℓ, the required delay condition for ru follows directly from the fact that ex is $\underline{s}$-admissible.
- *Job Duration:* Follows directly from the fact that $d(J) = \underline{\mu}_{(\ell)} = \mu^-_{(\ell)}$ for all jobs J sending ℓ ordinary messages. □

As $\mathcal{A}$ is an algorithm solving $\mathcal{P}$ in s and ru is an s-admissible rt-run of $\mathcal{A}$, ru satisfies $\mathcal{P}$ (by Definition 12). As the sequence of events (excluding type (b) and (d) actions, which do not affect the problem-relevant part of the state) on each individual processor remains the same during the transformation, trace properties satisfied in ru are also satisfied in ex.

W.r.t. state properties, let $t \to (state_1(t), HC_1(t), \dots)$ be a global state trajectory of ex.

Lemma 6. *$state_p(t)$ reduced to the problem-relevant subset of variables is also a local state trajectory of ru.*

Proof. We will show that the conditions set forth in Definition 5 w.r.t. ru are satisfied, based on the design of the simulation algorithm and on the fact that $state_p(t)$ satisfies the conditions for trajectories in the classic computing model w.r.t. ex.

First, note that due to the removal of simulation variables, type (b) or (d) actions do not have any effect on the trajectories, i.e., $state_p(t) = newstate(ac)$ of the last type (a) or (c) action ac occuring on p before t, if there are only type (b) or (d) actions at time t. Thus, we will use the term "action" in this proof to refer to type (a) and (c) actions only.

- *No job running at t in ru*: This implies that there is also no action occuring at t in ex. Thus, $state_p(t) = newstate(ac)$ of the last action in ex, which equals $newstate(J)$ of the last job in ru (or $istate_p^{ex/ru}$, if there is no such action/job).
- *One or more jobs $J_1, \dots, J_k$ running at time t*: Clearly, as $begin(J) \leq t$ for all $J \in \{J_1, \dots, J_k\}$, $time(ac) \leq t$ as well for all corresponding actions ac. If a subset $ac_1, \dots, ac_l$ of these corresponding actions have $time(ac) = t$, $state_p(t) \in states(ac)$ for one of these actions, which implies that $state_p(t) \in states(J)$ of the corresponding job. Otherwise, if, for all corresponding actions ac, $time(ac) < t$, let ac be the last of the corresponding actions. There is no other action

ac' occuring after ac but before or at t, due to the non-overlapping property of ru and the fact that its corresponding job is not in $J_1, \dots, J_k$. Thus, $state_p(t) = newstate(ac)$, which is also a valid value in a trajectory of ru, as the job corresponding to ac is running at time t. $\square$

The fact that the global state trajectory is causally consistent w.r.t ru follows directly from the fact that it is causally consistent w.r.t. ex. Thus, every global state trajectory of ex is also a global state trajectory of ru. As the state predicate P in question is guaranteed to be satisfied in every global state trajectory of ru, it is also satisfied in every global state trajectory of ex.

Thus, ex satisfies $\mathcal{P}$, and Definition 10 concludes our proof of Theorem 2. $\square$

We finally note that the bound $\delta^-_{(\ell)} \leq \underline{\delta}^-$ and $\delta^+_{(\ell)} \geq \underline{\delta}^+$ for all ℓ in Theorem 2 is overly conservative. The following bound suffices:

Theorem 3. *Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system and $\mathcal{P}$ be a problem. If there exists an algorithm $\mathcal{A}$ for solving $\mathcal{P}$ in some real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with*

- $\delta^-_{(1)} \leq \underline{\delta}^-$ *and* $\forall \ell : \delta^+_{(\ell)} \geq \underline{\delta}^+$,

then $\underline{S'}_{\delta^-,\mu^-,\mathcal{A}}$ solves $\mathcal{P}$ in $\underline{s}$.

Proof. This proof and the detailed algorithm have been omitted, but an informal description is as follows:

The extended simulation algorithm allows $\delta^-_{(\ell)}$ to be greater than $\underline{\delta}^-$ for $\ell > 1$. However, since $\varepsilon_{(\ell)} \geq \varepsilon_{(1)}$ (see Section 3.4), we can ensure that the simulated message delays lie within $\delta^-_{(\ell)}$ and $\delta^+_{(\ell)}$ although the real message delay might be smaller than $\delta^-_{(\ell)}$ by introducing an artificial, additional message delay of $\delta^-_{(\ell)} - \delta^-_{(1)}$ upon receiving a message sent together with $\ell - 1$ other messages in some computing step: The minimal simulated delay is $\underline{\delta}^- + \delta^-_{(\ell)} - \delta^-_{(1)} \geq \delta^-_{(\ell)}$, and the maximum simulated delay is $\underline{\delta}^+ + \delta^-_{(\ell)} - \delta^-_{(1)} \leq \delta^+_{(1)} + \delta^-_{(\ell)} - \delta^-_{(1)} = \delta^-_{(\ell)} + \varepsilon_{(1)} \leq \delta^-_{(\ell)} + \varepsilon_{(\ell)} = \delta^+_{(\ell)}$. Of course, being able to add this delay implies that the algorithm message is wrapped into a simulation message that also includes ℓ. $\square$

5.4 Clock Synchronization

In the classic computing model, a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ has been proved in [14] as the best achievable clock synchronization precision. In addition, an algorithm $\underline{\mathcal{A}}(n, \underline{\delta}^-, \underline{\delta}^+)$ has been given, which guarantees this optimal precision in every classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ with $\underline{\varepsilon} = \underline{\delta}^+ - \underline{\delta}^-$. The algorithm works by sending one timestamped message from every processor to every other processor, and then computing the average of the estimated clock differences as a correction value. Any processor broadcasts its message as soon as its init message arrives.

The transformations provided in the previous sections can be used to generalize these results to the real-time computing model, resulting in an upper bound of $(1 - \frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ and a lower bound of $(1 - \frac{1}{n})\varepsilon_{(1)}$ for the achievable precision:

Theorem 4. *In the real-time computing model, clock synchronization within $(1-\frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2)\cdot\mu^+_{(0)})$ is possible.*

Proof. In the algorithm of [14], every processor receives exactly one message from every other processor, and all messages are sent as broadcasts to $n - 1$ recipients. The worst-case scenario for the end-to-end delay hence occurs if all $n - 1$ messages plus the one init message arrive simultaneously: After delivery of these messages (taking $\delta^+_{(n-1)}$), the receiver's own broadcast send step (taking $\mu^+_{(n-1)}$) as well as $n-2$ receive steps ($\mu^+_{(0)}$) must complete before the last receive step can start. An upper bound on the end-to-end delay of running $\mathcal{S}_{\underline{\mathcal{A}}}$ in the real-time computing model is hence $\Delta^+ = \delta^+_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)}$.

Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system in which we want to synchronize clocks. We know that $\underline{\mathcal{A}}(n, \Delta^-, \Delta^+)$ will synchonize clocks to within $\gamma = (1 - \frac{1}{n})(\Delta^+ - \Delta^-)$ in the classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$. As $\underline{\mathcal{A}}$ is timer-free and all actions send either 0 or $n - 1$ messages, the smallest possible end-to-end delay is $\delta^-_{(n-1)}$, and Theorem 1 shows that $\mathcal{S}_{\underline{\mathcal{A}}}$ provides clock synchronization within $(1 - \frac{1}{n})(\Delta^+ - \Delta^-) = (1 - \frac{1}{n})(\delta^+_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)} - \delta^-_{(n-1)}) = (1 - \frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ in s. $\square$

As far as the time complexity of the above algorithm is concerned, we observe that at most $\delta^+_{(n-1)}$ time units after the last init message arrived, all processors have all $n - 1$ messages in their queue (or already processed). Due to non-idling scheduling, this implies a maximum time complexity of $\max(\delta^+_{(n-1)}, \mu^+_{(n-1)}) + (n-1) \cdot \mu^+_{(0)}$, which occurs if all processors' init messages arrive at the same time. In sharp contrast to the classic computing model, where the time complexity of this algorithm is $\Omega(1)$, the time complexity in the real-time computing model is hence $\Omega(n)$.

Likewise, we can use the other transformation to prove that clock synchronization closer than $(1-\frac{1}{n})\varepsilon_{(1)}$ is impossible.

Theorem 5. *In the real-time computing model, no algorithm can synchronize the clocks of a system closer than $(1-\frac{1}{n})\varepsilon_{(1)}$.*

Proof. Assume for a contradiction that there is some real-time computing model algorithm $\mathcal{A}$ which can provide clock synchronization for some real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ within $\gamma < (1-\frac{1}{n})\varepsilon_{(1)}$. Applying Theorem 3 would imply that $\underline{\mathcal{S}'}_{\delta^-,\mu^-,\mathcal{A}}$ provides clock synchronization within $\gamma < (1-\frac{1}{n})(\underline{\delta}^+ - \underline{\delta}^-)$ for some classic system $(n, [\underline{\delta}^- = \delta^-_{(1)}, \underline{\delta}^+ = \delta^+_{(1)}])$. This, however, contradicts the well-known lower bound result of [14]. □

6 Algorithms Achieving Optimal Precision

The comparison of Theorems 4 and 5 raises the obvious question whether the lower bound of $(1-\frac{1}{n})\varepsilon_{(1)}$ is tight in the real-time computing model? In this section, we will answer this in the affirmative: We show how the algorithm presented in [14] can be modified to avoid queuing effects and thus provides optimal precision in a real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$. We will first present an algorithm achieving a precision of $(1-\frac{1}{n})\varepsilon_{(n-1)}$ (which is $(1-\frac{1}{n})\varepsilon_{(1)}$ if a constant-time broadcast primitive is available) and then describe how to extend this algorithm so that it achieves $(1-\frac{1}{n})\varepsilon_{(1)}$ in the unicast case as well.

6.1 Generalization of Existing Results

Two lemmata from [14] can be generalized to our setting:

Lemma 7. *If q receives a timestamped message from p with end-to-end delay[13] uncertainty $\dot{\varepsilon}_\Delta$, q can estimate p's hardware clock value within an error of at most $\frac{\dot{\varepsilon}_\Delta}{2}$.*

Proof. (Similar to Lemma 5 of [14]). We define $D = HC_p(t) - HC_q(t)$ to be the actual difference between the hardware clocks of p and q (a constant, as clocks do not drift) and E to be the estimated difference, as estimated by q. Thus, we have to show that $\exists E : |E - D| \le \frac{\dot{\varepsilon}_\Delta}{2}$. Let $\dot{\Delta}^-$ and $\dot{\Delta}^+$ be the lower and upper bound on the end-to-end delay.

Let t be the time by which p sends its clock value (more precisely: the start time of the job in which p sends its clock value) and t' be the time by which q starts processing this message. Let Δ be the arithmetic mean between the lower and the upper bound on the end-to-end delay, i.e., $\Delta = \dot{\Delta}^- + \frac{\dot{\varepsilon}_\Delta}{2}$. Process q calculates the estimate as follows: $E = HC_p(t) - HC_q(t') + \Delta$, where $HC_p(t)$ is the timestamp in the message, $HC_q(t')$ is the hardware clock reading of the job processing the message and Δ must be known to the algorithm.

$$\begin{aligned}
|E - D| &= |HC_p(t) - HC_q(t') + \Delta - D| \\
&= |HC_q(t) - HC_q(t') + \Delta| \\
&\quad \text{(by Definition of } D) \\
&= |t - t' + \Delta| \\
&\quad \text{(Clocks run at the same rate as real time)} \\
&= |\Delta - (t' - t)|
\end{aligned}$$

As $t' - t$ ranges from $\dot{\Delta}^-$ to $\dot{\Delta}^+$, the expression $\Delta - (t' - t)$ ranges from $\Delta - \dot{\Delta}^+ = -\frac{\dot{\varepsilon}_\Delta}{2}$ to $\Delta - \dot{\Delta}^- = \frac{\dot{\varepsilon}_\Delta}{2}$. □

Lemma 8. *If every processor knows the difference between its own hardware clock and the hardware clock of every other processor within an error of at most $\frac{err}{2}$, clock synchronization within $(1-\frac{1}{n})err$ is possible.*

Proof. The proof can be obtained by a simple adaption of Theorem 7 of [14] to a general err. □

6.2 Optimality for Broadcast Systems

Note. As all jobs in this algorithm send either zero or $n-1$ messages, we will use the abbreviations $\dot{\delta}^-$, $\dot{\delta}^+$, $\dot{\mu}^-$, $\dot{\mu}^+$ and $\dot{\varepsilon}$ to refer to $\delta^-_{(n-1)}$, $\delta^+_{(n-1)}$, $\mu^-_{(n-1)}$, $\mu^+_{(n-1)}$ and $\varepsilon_{(n-1)}$, respectively.

In Section 5.4, the principle of the Lundelius-Lynch algorithm has been described. It can easily be modified to avoid queuing effects by "serializing" the information exchange, rather than sending all messages simultaneously.

The modified algorithm, depicted in Figure 8, works as follows: The n fully-connected processors have IDs $0, \ldots, n-1$. The first processor (0) sends its clock value to all other processors. Processor i waits until it has received the message from processor $i-1$, waits for another $\max(\dot{\varepsilon} - \dot{\delta}^- + \dot{\mu}^+, \dot{\mu}^+)$ time units and then broadcasts its own hardware clock value. That way, every processor receives the hardware clock values of all other processors with uncertainty $\dot{\varepsilon}$, provided that no queuing occurs (which will be shown below). This information suffices to synchonize clocks to within $(1-\frac{1}{n})\dot{\varepsilon}$. We assume here that only one init message is sent (only to processor 0), as additional init messages could cause unwanted queuing effects and would hence neccessiate a second round of message exchanges.

[13]Recall that the end-to-end delay is defined as the time between the start of the job sending the message and the start of the job processing the message.

```
1   var estimates ← {}
2   var adj
3
4   function process_message(msg, time)
5       /* start alg. by sending (SEND) to proc. 0 */
6       if msg = (SEND)
7           send (TIME, time) to all other processors
8       elseif msg = (TIME, remote_time)
9           estimates.add(remote_time − time + $\frac{\dot\delta^- + \dot\delta^+}{2}$)
10          if estimates.count = ID
11              send timer (SEND) for time + $\max(\dot\varepsilon - \dot\delta^- + \dot\mu^+, \dot\mu^+)$
12          if estimates.count = n−1
13              adj ← $(\sum estimates)/n$
```

Figure 8: Clock-synchronization algorithm to within $\varepsilon_{(n-1)}$, code for processor ID

Lemma 9. *No queueing occurs when running the algorithm in Figure 8.*

Proof. By the design of the algorithm, processor i only broadcasts its message after it has received exactly i messages. As processor 0 starts the algorithm and every processor broadcasts only once, this causes the processors to send their messages in the order of increasing processor number. For queuing to occur, some processor p must receive two messages within a time window smaller than $\dot\mu^+$. It can be shown, however, that the following invariant holds for all t: All receive events up to time t on the same processor i (a) occur in order of increasing (sending) processor number (including the timer message from i itself) and (b) are at least $\dot\mu^+$ time units apart.

Assume by contradiction that some message from processor $j > 0$ arrives on processor i at time t, although the message from processor $j-1$ has arrived (or will arrive) at time $t' > t - \dot\mu^+$. Choose t such that t is the first time the invariant is violated.

Case 1: $j = i$, i.e., the arriving message is i's timer message. This leads to a contradiction, as due to line 11, the timer message must not arrive earlier than $\dot\mu^+$ time units after $j-1$'s message, which has triggered the job sending the timer message.

Case 2: $j \neq i$. As j's broadcast arrived at t, it has been sent no later than $t - \dot\delta^-$. Process j's broadcast is triggered by a timer message sent by j's job starting $\max(\dot\varepsilon - \dot\delta^- + \dot\mu^+, \dot\mu^+)$ time units earlier, i.e., no later than $t - \dot\delta^- - (\dot\varepsilon - \dot\delta^- + \dot\mu^+) = t - \dot\varepsilon - \dot\mu^+$. The job sending the timer message has been triggered by the arrival of $j-1$'s broadcast, which must have been sent no later than $t - \dot\varepsilon - \dot\mu^+ - \dot\delta^-$. If $j - 1 = i$, we have the required contradiction, because i must have received its timer message at $t' \leq t - \dot\varepsilon - \dot\mu^+ - \dot\delta^-$ long ago. Otherwise, if $j - 1 \neq i$, process $j-1$'s broadcast arrived at i no later than $t - \dot\varepsilon - \dot\mu^+ - \dot\delta^- + \dot\delta^+ = t - \dot\mu^+$, also contradicting the assumption. □

Using this lemma, it is not difficult to show the following Theorem 6:

Theorem 6 (Optimal broadcasting algorithm)**.** *The algorithm of Figure 8 achieves a precision of* $(1 - \frac{1}{n})\dot\varepsilon$*, which is tight if communication is performed by a constant-time broadcast primitive, i.e., if* $\varepsilon_{(n-1)} = \varepsilon_{(1)}$. *It performs exactly* n *broadcasts and has a time complexity that is at least* $\Omega(n)$.

Proof. On each processor, the *estimates* set contains the estimated differences between the local hardware clock and the hardware clocks of the other processors. As no queuing occurs by Lemma 9, the end-to-end delays are just the message delays. Line 9 in the algorithm of Figure 8 ensures that the estimate is calculated as specified in the proof of Lemma 7. Thus, the estimates have a maximum error of $\frac{\dot\varepsilon}{2}$. According to Lemma 8, these estimates allow the algorithm to calculate an adjustment value in line 13 that guarantees clock-synchronization within $(1 - \frac{1}{n})\dot\varepsilon$.

With respect to message and time complexity, the algorithm oviously performs exactly n broadcasts, and the maximum time between two subsequent broadcasts is $\max(\dot\delta^+, 2\dot\varepsilon) + \dot\mu^+$ (= the timer delay plus one message delay). Thus, the time complexity is at least linear in n, and depends on the complexity of $\varepsilon_{(\ell)}$ and $\mu^+_{(\ell)}$ w.r.t. ℓ. □

6.3 Optimality for Unicast Systems

Note. As all computing steps in this algorithm send either zero or one messages, we will use the abbreviations $\dot\delta^-$, $\dot\delta^+$, $\dot\mu^-$, $\dot\mu^+$ and $\dot\varepsilon$ to refer to $\delta^-_{(1)}$, $\delta^+_{(1)}$, $\mu^-_{(1)}$, $\mu^+_{(1)}$ and $\varepsilon_{(1)}$, respectively.

$i \oplus j$ and $i \ominus j$ are defined as $(i + j \mod n)$ and $(i - j \mod n)$, respectively. These operations will be used for adding and subtracting processor indices.

The algorithm of the previous section provides clock synchronization within $(1 - \frac{1}{n})\varepsilon_{(n-1)}$. However, unless constant-time broadcast is available, $\varepsilon_{(1)}$ will usually be smaller than $\varepsilon_{(n-1)}$. The algorithm can be adapted to unicast sends as follows (see Figure 9):

Rather than sending all $n-1$ messages at once, they are sent in $n-1$ subsequent jobs connected by "send" timer messages, each sending only one message. These messages are timestamped with their corresponding HC value, e.g. the message sent during the second job will

```
1   var estimates ← {}
2   var adj
3
4   function process_message(msg, time)
5     /* start alg. by sending (SEND, 1) to proc. 0 */
6     if msg = (SEND, target)
7       send (TIME, time) to target
8       if target + 1 mod n ≠ ID
9         send timer (SEND, target + 1 mod n) for time + $\dot{\mu}^+$
10    elseif msg = (TIME, remote_time)
11      estimates.add(remote_time − time + $\frac{\dot{\delta}^- + \dot{\delta}^+}{2}$)
12      if estimates.count = ID
13        send timer (SEND, ID + 1) for
14              time + $\max(\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+, \dot{\mu}^+)$
15      if estimates.count = n−1
16        adj ← $(\sum estimates)/n$
```

Figure 9: Clock-synchronization algorithm to within $\varepsilon_{(1)}$, code for processor ID

be timestamped with the hardware clock reading of this second job.

By the design of the algorithm, every processor i goes though five phases. The only exception is processor 0, which starts at phase 3.

1. *First part receive phase*: i receives TIME messages from all processors $\{0, \dots, i-1\}$ in the order of increasing processor number.
2. *Wait phase*: After having received $i-1$'s TIME message, line 14 causes i to wait for $W := \max(\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+, \dot{\mu}^+)$ time units.
3. *Send phase*: i sends TIME messages to all processors (each in its own job, all jobs $\dot{\mu}^+$ time units apart).
4. *Second part receive phase*: i receives TIME messages from all processors $\{i+1, \dots, n-1\}$ in order of increasing processor number.
5. *Terminated phase*: No more messages are received; i has terminated.

We will use the following abbreviations to label messages and the corresponding receive events and jobs processing (not sending) them: $\text{TIME}_{i \to j}$ (TIME message from i to j), $\text{SEND}_{i, \to j}$ (SEND timer message occuring on i, initiating the send of $\text{TIME}_{i \to j}$) and WAIT_i (= $\text{TIME}_{i-1 \to i}$, because it initiates the wait phase). *begin*$(\dots)$ denotes the beginning of the corresponding job processing the message. To ease analysis, we assume a "virtual" no-op job WAIT_0 with begin time $begin(\text{WAIT}_0) = begin(\text{SEND}_{0,\to 1}) - W$.

See Figure 10 for an example. Note that every processor sends exactly one TIME message to every other processor.

Lemma 10. *The following invariant holds for all t when running the algorithm in Figure 9: All messages received up to time t on some processor i have been received in the following order: $\langle \text{TIME}_{0 \to i}, \dots, \text{TIME}_{i-1 \to i} = \text{WAIT}_i, \text{SEND}_{i, \to i \oplus 1}, \dots, \text{SEND}_{i, \to i \oplus (n-1)}, \text{TIME}_{i+1 \to i}, \dots, \text{TIME}_{n-1 \to i} \rangle$. All receive events up to time t on the same processor are at least $\dot{\mu}^+$ time units apart, which implies that no queuing occurs.*

The begin times of SEND jobs on the same processor are exactly $\dot{\mu}^+$ time units apart. $\text{SEND}_{i, \to i \oplus 1}$ arrives at $begin(\text{WAIT}_i) + W$.

Proof. By induction on the message arrival times in the rt-run. The following arrivals can happen at time t which could violate the invariant:

- *First/second part receive and wait phase:* Assume for $0 < j < i$ (first part receive phase/wait phase) or $i < j-1 < n-1$ (second part receive phase) that $\text{TIME}_{j \to i}$ arrives at $t < begin(\text{TIME}_{j-1 \to i}) + \dot{\mu}^+$. $\text{TIME}_{j \to i}$ has been sent no later than $t - \dot{\delta}^-$ by j's $\text{SEND}_{j, \to i}$ job. As the invariant holds for all arrivals before t, the begin times of the $((i \ominus 1) \ominus j)$ send phase steps of process j before $(\text{SEND}_{j, \to j \oplus 1}, \dots, \text{SEND}_{j, \to i \ominus 1})$ and $\text{SEND}_{j, \to i}$ are exactly $\dot{\mu}^+$ time units apart, and WAIT_j starts at least $\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+$ time units before the first send phase step. This means that

$$\begin{aligned} &begin(\text{WAIT}_j) \\ &\quad \le t - \dot{\delta}^- - ((i \ominus 1) \ominus j)\dot{\mu}^+ - (\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+) \\ &\quad = t - (i \ominus j)\dot{\mu}^+ - \dot{\varepsilon} - \dot{\mu}^+. \end{aligned}$$

  $\text{WAIT}_j = \text{TIME}_{j-1 \to j}$ has been sent during $j-1$'s $\text{SEND}_{j-1, \to j}$ job. Thus,

$$begin(\text{SEND}_{j-1, \to j}) \le t - (i \ominus j)\dot{\mu}^+ - \dot{\varepsilon} - \dot{\mu}^+ - \dot{\delta}^-.$$

  Clearly, $\text{SEND}_{j-1, \to j}$ refers to the first send job on $j-1$. $\text{TIME}_{j-1 \to i}$ is sent during $\text{SEND}_{j-1, \to i}$, which starts exactly $(i \ominus j)$ time units later:

$$\begin{aligned} &begin(\text{SEND}_{j-1, \to i}) \\ &\quad \le t - (i \ominus j)\dot{\mu}^+ - \dot{\varepsilon} - \dot{\mu}^+ - \dot{\delta}^- + (i \ominus j)\dot{\mu}^+ \\ &\quad = t - \dot{\varepsilon} - \dot{\mu}^+ - \dot{\delta}^-. \end{aligned}$$

  $\text{TIME}_{j-1 \to i}$ arrives at most $\dot{\delta}^+$ time units later,

$$begin(\text{TIME}_{j-1 \to i}) \le t - \dot{\varepsilon} - \dot{\mu}^+ - \dot{\delta}^- + \dot{\delta}^+ = t - \dot{\mu}^+,$$

  contradicting the assumption that $t < begin(\text{TIME}_{j-1 \to i}) + \dot{\mu}^+$.

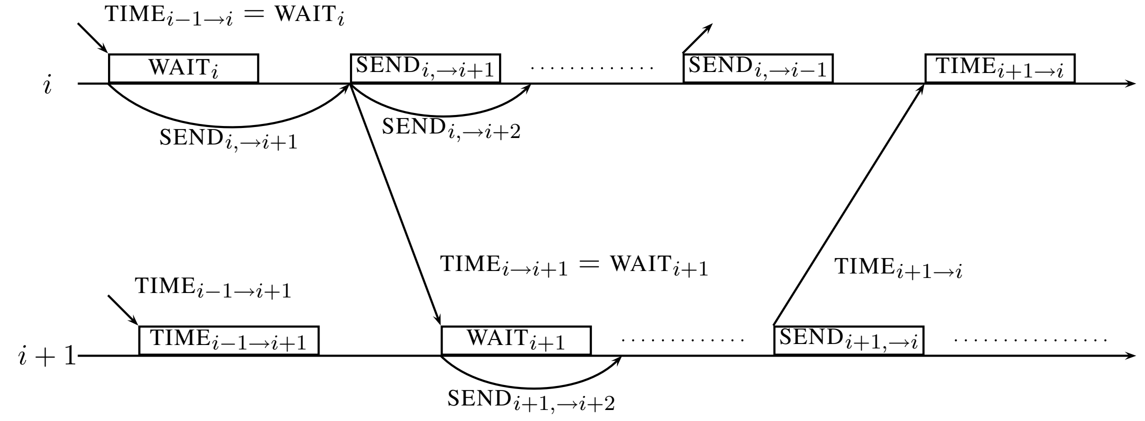


Figure 10: Processor i $(0 < i < n-2)$ switching from first part receive phase to wait, from wait to send, and from send to second part receive phase.

- *Wait $\to$ send phase*: Assume the SEND$_{i,\to i+1}$ timer message arrives at $t \neq \mathit{begin}(\text{WAIT}_i) + W$. As the SEND$_{i,\to i+1}$ timer is set in WAIT$_i$ to W, this is a contradiction.
- *Send phase*: Assume for $i \neq j$ and $i \neq j \oplus 1$ that SEND$_{i,\to j\oplus 1}$ arrives at $t \neq$ SEND$_{i,\to j} + \dot{\mu}^+$. As the SEND$_{i,\to j\oplus 1}$ timer is set in SEND$_{i,\to j}$ to $\dot{\mu}^+$, this is a contradiction.
- *Send $\to$ second part receive phase*: Assume for $i < n-1$ that TIME$_{i+1\to i}$ arrives at $t < \mathit{begin}(\text{SEND}_{i,\to i\oplus(n-1)}) + \dot{\mu}^+$ (= begin time of i's last send job $+\dot{\mu}^+$). TIME$_{i+1\to i}$ was sent during SEND$_{i+1,\to i}$ = SEND$_{i+1,\to(i+1)\oplus(n-1)}$, which started no later than $t - \dot{\delta}^-$. As the invariant holds for all arrival times $< t$, SEND$_{i+1,\to(i+1)\oplus 1}$ started no later than $t - \dot{\delta}^- - (n-2)\dot{\mu}^+$. This means that WAIT$_{i+1}$ = TIME$_{i\to i+1}$ started no later than $t - \dot{\delta}^- - (n-1)\dot{\mu}^+$ and TIME$_{i\to i+1}$ was sent (by job SEND$_{i,\to i+1}$) no later than

$$\mathit{begin}(\text{SEND}_{i,\to i+1}) \leq t - 2\dot{\delta}^- - (n-1)\dot{\mu}^+$$

As the SEND jobs are exactly $\dot{\mu}^+$ time units apart,

$$\begin{aligned}&\mathit{begin}(\text{SEND}_{i,\to i\oplus(n-1)})\\ &\quad\leq t - 2\dot{\delta}^- - (n-1)\dot{\mu}^+ + (n-2)\dot{\mu}^+\\ &\qquad\qquad = t - 2\dot{\delta}^- - \dot{\mu}^+\end{aligned}$$

which contradicts the assumption that $t < \mathit{begin}(\text{SEND}_{i,\to i\oplus(n-1)}) + \dot{\mu}^+$.

□

We can apply Lemma 7 to the algorithm of Figure 9 as well, resulting in estimates with a maximum error of $\frac{\varepsilon_{(1)}}{2}$ rather than $\frac{\varepsilon_{(n-1)}}{2}$. Thus, by Lemma 8, clock synchronization within $(1-\frac{1}{n})\varepsilon_{(1)}$ can be achieved. As all job durations and message delays are independent of n this time ($\dot{\delta}^+_{(1)}$ rather than $\dot{\delta}^+_{(n-1)}$, etc.), the time complexity of this algorithm is $\mathrm{O}(n)$.

7 Lower Bounds

In this section, we will establish lower bounds for message and time complexity of (close to) optimal precision clock synchronization algorithms.

In particular, for optimal precision, we will prove that at least $\frac{1}{2}n(n-1) = \Omega(n^2)$ messages must be exchanged, since at least one message must be sent over every link. This bound is tight, since it is matched by the algorithms from the previous section.

A strong indication for this result follows already from the work of Biaz and Welch [6]. They have shown that no algorithm can achieve better precision than $\frac{1}{2}\mathit{diam}(G)$ for any communication network G, with $\mathit{diam}(G)$ being the diameter of the graph when the edges are weighted with the uncertainties: In the classic computing model, a fully-connected network with equal link uncertainty $\underline{\varepsilon}$ can achieve no better precision than $\frac{1}{2}\underline{\varepsilon}$, whereas removing one link yields a lower bound of $\underline{\varepsilon}$. Thus, after removing one link, the optimal precision of $(1-\frac{1}{n})\underline{\varepsilon}$ shown by [14] can no longer be achieved.

Unfortunately, the proof from [6] cannot be used directly in our context to derive the message complexity bound mentioned above: While they show that $(1-\frac{1}{n})\underline{\varepsilon}$ cannot be achieved if the system forbids the algorithm to use one system-chosen link, we have to show that if the algorithm is presented with a fully-connected network and decides not to use one algorithm-chosen link (which can differ for each execution/rt-run) dynamically, this algorithm cannot achieve optimal precision. A shifting argument similar to the one used in their proof (Theorem 3 of [6]) can be used, however.

Additionally, we will show that the message and time complexity of clock synchronization within suboptimal precision also depends on the complexity of $\delta^+_{(\ell)}$ and $\mu^+_{(\ell)}$ w.r.t. ℓ.

Environment Let $c \in \mathbb{R}^+$ be a constant and $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system with $n > 2$. Assume that $\mathcal{A}$ is an algorithm providing clock synchroniziation within $c \cdot \varepsilon_{(1)}$ in s. Let ru be an s-admissible rt-run of $\mathcal{A}$ in s where the message delays of all messages are the arithmetic mean of the lower and upper bound. Thus, modifying the delay of any message by $\pm\varepsilon_{(1)}/2$ still results in a value within the system model bounds. The duration of all jobs sending m messages is $\mu^+_{(m)}$.

7.1 Message Graph Diameter

Definition 17. Let the *message graph* of a rt-run ru be defined as an undirected graph containing all processors as vertices and exactly those links as edges over which at least one message is sent in ru.

Lemma 11. *The message graph of ru has a diameter of $2c$ or less.*

Proof. Assume by contradiction that the message graph has a diameter $D > 2c$. Let p and q be two processors at distance D. Let Π_d be the set of processors at distance d from p. Let ru' be a new rt-run in which processors in Π_d are shifted by $d \cdot \varepsilon_{(1)}/2$, i.e., all receive events and jobs on some processor in Π_d happen $d \cdot \varepsilon_{(1)}/2$ time units earlier although with the same hardware clock readings (see Figure 11 for an example). As processors in Π_d only exchange messages with processors in Π_{d-1}, Π_d and Π_{d+1}, message delays are changed by $-\varepsilon_{(1)}/2$, 0 or $\varepsilon_{(1)}/2$. Thus, ru' is s-admissible.

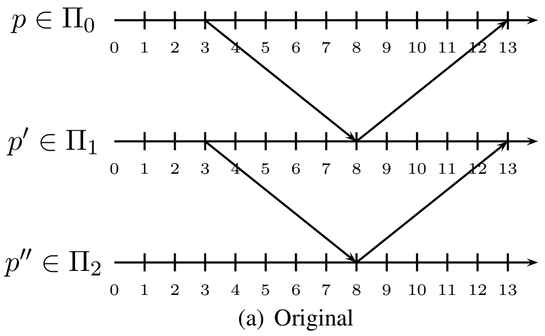
Let Δ and Δ' be the final (signed) differences between the adjusted clocks of p and q in ru and ru', respectively. As both rt-runs are s-admissible and $\mathcal{A}$ is assumed to be correct, $|\Delta| \le c \cdot \varepsilon_{(1)}$ and $|\Delta'| \le c \cdot \varepsilon_{(1)}$.

By definition of shifting, $HC'_p(t) = HC_p(t)$ and $HC'_q(t) = HC_q(t) + D \cdot \varepsilon_{(1)}/2$. Thus, $\Delta' = HC'_p(t) + adj_p - (HC'_q(t) + adj_q) = HC_p(t) + adj_p - (HC_q(t) + D \cdot \varepsilon_{(1)}/2 + adj_q) = \Delta - D \cdot \varepsilon_{(1)}/2$.

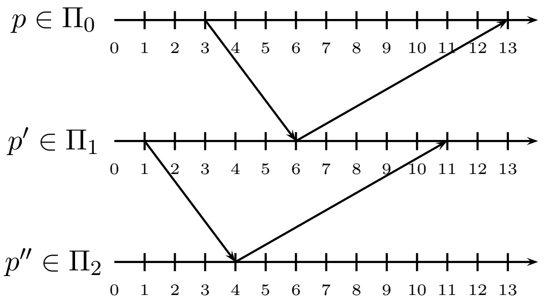
Let ru'' be ru shifted by $-d \cdot \varepsilon_{(1)}/2$. The same arguments hold, resulting in $\Delta'' = \Delta + D \cdot \varepsilon_{(1)}/2$. As $|\Delta|$, $|\Delta'|$ and $|\Delta''|$ must all be $\le c \cdot \varepsilon_{(1)}$, we have the following inequalities:

$$|\Delta| \le c \cdot \varepsilon_{(1)}$$
$$|\Delta + D \cdot \varepsilon_{(1)}/2| \le c \cdot \varepsilon_{(1)}$$
$$|\Delta - D \cdot \varepsilon_{(1)}/2| \le c \cdot \varepsilon_{(1)}$$

which imply that $c \ge D/2$ and provide the required contradiction to $D > 2c$. □



(a) Original



(b) Shifted

Figure 11: Shifting by $d \cdot \varepsilon_{(1)}/2$ with $\varepsilon_{(1)} = 4$

7.2 Message Complexity

For clock sychronization within some $\gamma < \varepsilon_{(1)}$ (i.e., $c < 1$), Lemma 11 implies that there exists a rt-run whose message graph has a diameter < 2, i.e., whose message graph is fully connected, and, therefore, has $\frac{n(n-1)}{2}$ edges. This leads to the following theorem:

Theorem 7. *Clock synchronization within $\gamma < \varepsilon_{(1)}$ has a worst-case message complexity of $\Omega(n^2)$.*

Section 6 presented algorithms achieving optimal precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ with $n(n-1) = \mathrm{O}(n^2)$ messages. Theorem 7 reveals that this bound is asymptotically tight.

A weaker lower bound can be given for suboptimal clock synchronization. We will use the following simple graph-theoretical lemma:

Lemma 12. *In an undirected graph with $n > 2$ nodes and diameter D or less, there is at least one node with degree $\geq \sqrt[D+1]{n}$.*[14]

Proof. Assume by contradiction that all nodes have a maximum degree of some non-negative integer $d < \sqrt[D+1]{n}$. As $n > 2$, $d = 0$ or $d = 1$ would cause the graph to be disconnected, thereby contradicting the assumption of bounded diameter. Thus, we can assume that $d > 1$.

Fix some node p. Clearly, after D hops, the maximum number of nodes reachable from p (including p at distance 0) is $\sum_{i=0}^{D} d^i = \frac{d^{D+1}-1}{d-1} \leq d^{D+1} < \sqrt[D+1]{n}^{D+1} = n$. As we cannot reach n nodes after D hops, we have the required contradiction. $\square$

Combining Lemmata 11 and 12 shows that there is at least one processor in ru which exchanges (= sends or receives) at least $\lceil \sqrt[2c+1]{n} \rceil$ messages. More general:

Theorem 8. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$ in some system real-time system s, there is at least one s-admissible rt-run in which at least one processor exchanges $\lceil \sqrt[2c+1]{n} \rceil$ messages.*

Corollary 1. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$, there is no constant upper bound on the number of messages exchanged per processor.*

It is, however, possible to either bound the number of received messages *or* the number of sent messages per processor: Section 8.1 presents an algorithm synchronizing clocks within $\varepsilon_{(1)}$ where every processor receives exactly one message. On the other hand, the algorithm in Section 8.2 also achieves this precision but bounds the number of sent messages per processor by 3.

7.3 Time Complexity

Theorem 8 immediately implies a lower bound on the worst-case time complexity of any algorithm that synchronizes clocks to within $c \cdot \varepsilon_{(1)}$: Some process p must exchange $\lceil \sqrt[2c+1]{n} \rceil$ messages, some k of which are received and the remaining ones are sent by p. Recalling $\delta^+_{(\ell)} \leq \ell\delta^+_{(1)}$ from Section 3.4, the algorithm's time complexity must be aat least $\min_{k=0}^{\lceil \sqrt[2c+1]{n} \rceil}(k \cdot \mu^+_{(0)} + \delta^+_{(n-k)})$.[15] Clearly, $k\mu^+_{(0)}$ is linear in k, so the interesting term is $\delta^+_{(n-k)}$, leading to the following corollary:

[14]A result with similar order of magnitude can be derived from the Moore bound.

[15]This bound cannot be reduced to the minimum of both extreme cases, counterexample: $\mu^+_{(0)} = 2, \delta^+_{(1,\ldots,6)} = \{3, 6, 6, 6, 9, 12\}$: $k = 2$ is smaller than $k = 0$ or $k = 6$.

```
var adj

function $s-$process_message(msg, time)
    /* start alg. by sending (INIT) to some proc. */
    if msg = (INIT)
        send time to all other processors
        adj ← 0
    else
        adj ← msg − time + $\frac{\delta^-_{(n-1)}+\delta^+_{(n-1)}}{2}$
```

Figure 12: Star Topology-based Clock Synchronization Algorithm

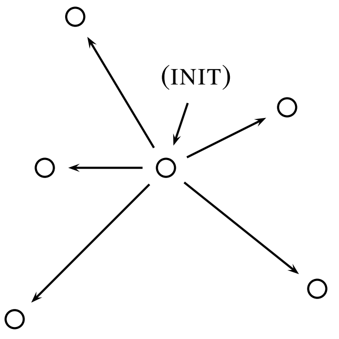


Figure 13: Principle of the Star Topology-based Clock Synchronization Algorithm

Corollary 2. *If multicasting a message in constant time is impossible, constant-time clock synchronization within a constant factor of the message delay uncertainty cannot be done in constant time.*

In the case of optimal precision, n processors need to process at least $\frac{n(n-1)}{2}$ messages, so no algorithm can achieve a run time better than $\frac{n-1}{2}\mu^+_{(0)}$ or better than $\delta^+_{(\frac{n-1}{2})}$ (assuming optimal parallelism). This shows that the algorithm presented in Section 6.3 is not only tight regarding precision but also has asymptotically optimal time complexity (O(n)).

8 Achievable Precision for $< \Omega(n^2)$ Messages

Sometimes, $\Omega(n^2)$ messages might be too costly if a precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ is not required. Clearly, every clock synchronization algorithm requires a minimum of $n - 1$ messages; otherwise, at least one processor would not participate. Interestingly, $n - 1$ messages (plus one external init message) already suffice to achieve a precision of $\varepsilon_{(1)}$ by using a simple star topology-based algorithm, presented in the following subsection.

8.1 Algorithm With Least Number of Messages

Figure 12 is actually a simpler version of the algorithm presented in Section 6: Rather than collecting the estimated differences to all other processors and then calculating the adjustment value, this algorithm just sets the adjustment value to the estimated difference to one designated master processor, the one receiving (INIT) (cf. Figure 13). Lemma 7 shows that the error of these estimates is bounded by $\frac{\varepsilon_{(n-1)}}{2}$. Thus, setting the adjustment value to the estimated difference causes all clocks to be synchronized within $\varepsilon_{(n-1)}$.

If δ^-, δ^+, μ^- and μ^+ are independent from n (i.e., if constant-time broadcasting is possible), $\varepsilon_{(n-1)} = \varepsilon_{(1)}$ and the algorithm achieves this precision in constant time (w.r.t. n). Otherwise, the following modification puts the precision down to $\varepsilon_{(1)}$ in the general case as well:

- Do not send all messages during the same job but during subsequent jobs on the "master" processor.
- Replace $\delta^-_{(n-1)}$ and $\delta^+_{(n-1)}$ in Line 9 with $\delta^-_{(1)}$ and $\delta^+_{(1)}$.

The algorithm still exchanges only $n-1$ messages and has linear time complexity w.r.t. n. As Theorem 7 has shown, $\varepsilon_{(1)}$ is the best precision that can be achieved with less than $\Omega(n^2)$ messages. As Corollary 2 has shown, this precision cannot be achieved in constant time in the general case.

8.2 Algorithm With Constant Bound on Number of Sent Messages per Processor

This is an informal description of a proof-of-concept algorithm showing that clock synchronization within $\varepsilon_{(1)}$ is possible with a constant bound (3 messages) on the number of messages sent per processor.

All processors send their current hardware clock reading to some designated processor q. This must be done in a serialized way to avoid queuing, and, thus, requires two sent messages per processor (one message to q and another message to the next processor). After this is done, q knows the difference between its own hardware clock and the hardware clock of any other processor to within $\varepsilon_{(1)}$. Section 6 showed that this estimate can be used to calculate an adjustment value for p, which, when applied, causes the clocks of p and q to be synchronized to within $\varepsilon_{(1)}/2$. To inform the other processors about their adjustment values, q sends the array of all adjustment values to some processor p, which passes them on the next processor and so on (requires one message per processor) until all processors have received their adjustment values. These values are finally applied, resulting in an overall clock synchronization precision of at most $\varepsilon_{(1)}$.

9 Conclusions and Future Work

We presented an real-time computing model, which just adds non-zero computing step times to the classic computing model. Since it explicitly incorporates queueing effects, our model makes distributed algorithms amenable to real-time scheduling analysis, without, however, invalidating classic algorithms, analysis techniques, and impossibility/lower bound results. General transformations based on simulations between both models were established for this purpose.

Revisiting the problem of optimal deterministic clock synchronization in the drift- and failure-free case, we showed that the best precision achievable in the real-time computing model is $(1 - \frac{1}{n})\varepsilon_{(1)}$. This matches the well-known result in the classic computing model; it turned out, however, that there is there is no constant-time algorithm achieving optimal precision in the real-time computing model. Since such an algorithm is known for the classic computing model, we have found an instance of a problem where the classic analysis gives too optimistic results. We also established algorithms and lower bounds for sub-optimal clock synchronization in the real-time computing model. For example, we showed that clock synchronization within a constant factor of the message delay uncertainty can be achieved in constant time only if a constant-time broadcast primitive ist available. Table 1 summarizes the bounds and the algorithms developed in this paper.

Part of our current research is devoted to extending our real-time computing model to failures and, in particular, drifting clocks. Clearly, all our lower bound results also hold for the drifting case. As time complexity influences the actual precision achievable with drifting clocks, however, a simpler, less precise algorithm might in fact yield some better overall precision than a more complex optimal algorithm, depending on the system parameters. Apart from this, we are looking out for problems and algorithms that involve more intricate real-time scheduling analysis techniques.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] J. H. Anderson, Y.-J. Kim, et al. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[3] J. H. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, Jan. 1996.

[4] H. Attiya, A. Herzberg, et al. Optimal clock synchronization under different delay assumptions. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pp. 109–120. ACM Press, New York, NY, USA, 1993. ISBN 0-89791-613-1.

Constraint	Consequences	Algorithm
-	Best precision: $(1 - \frac{1}{n})\varepsilon_{(1)}$ *Proof: Collorary 5*	
-	Overall message complexity: $\Omega(n)$ *Proof: obvious*	
-	$\exists$ one processor exchanging $\Omega(\sqrt[2(\gamma/\varepsilon_{(1)})+1]{n})$ msgs. *Proof: Theorem 8*	
Achieve best precision: $(1 - \frac{1}{n})\varepsilon_{(1)}$	Msg./time complexity: $\Omega(n^2)$, $\Omega(n)$ *Proof: Section 7*	Section 6
Achieve best msg. complexity: $\mathrm{O}(n)$	Best precision: $\varepsilon_{(1)}$ *Proof: Theorem 7*	Section 8.1

Table 1: Tight Bounds and Algorithms

[5] D. Basu and S. Punnekkat. Clock synchronization algorithms and scheduling issues. In *Proceedings International workshop on Distributed Systems (IWDC'03), LNCS 2918*. Springer-Verlag, December 2003.

[6] S. Biaz and J. L. Welch. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Information Processing Letters*, 80(3):151–157, 2001.

[7] R. Fan and N. Lynch. Gradient clock synchronization. In *Proceedings of the Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 320–327. Jul. 25-28, 2004.

[8] J.-F. Hermant and G. Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, Aug. 2002.

[9] J.-F. Hermant and J. Widder. Implementing reliable distributed real-time systems with the Θ-model. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, vol. 3974 of *LNCS*, pp. 334–350. Springer Verlag, Pisa, Italy, Dec. 2005.

[10] D. K. Kaynar, N. Lynch, et al. Timed i/o automata: A mathematical framework for modeling and analyzing real-time systems. *Proceedings 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 00:166, 2003.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. ISSN 0001-0782.

[12] G. Le Lann and U. Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Tech. Rep. 183/1-127, Department of Automation, Technische Universität Wien, January 2003.

[13] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 75–88. August 1984.

[14] —. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–240, 1984.

[15] N. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[16] N. Lynch and F. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, Sep. 1995.

[17] —. Forward and backward simulations, II: Timing-based systems. *Information and Computation*, 128(1):1–25, Jul. 1996.

[18] M. Merritt, F. Modugno, et al. Time-constrained automata (extended abstract). In *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, pp. 408–423. Springer-Verlag, London, UK, 1991. ISBN 3-540-54430-5.

[19] G. Neiger and S. Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334–367, 1993. ISSN 0004-5411.

[20] B. Patt-Shamir and S. Rajsbaum. A theory of clock synchronization (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pp. 810–819. ACM Press, New York, NY, USA, 1994. ISBN 0-89791-663-8.

[21] R. Segala, R. Gawlick, et al. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, 1998.

[22] L. Sha, T. Abdelzaher, et al. Real time scheduling theory: A historical perspective. *Real-Time Systems Journal*, 28(2/3):101–155, 2004.

[23] B. Simons, J. Lundelius-Welch, et al. An overview of clock synchronization. In B. Simons and A. Spector, eds., *Fault-Tolerant Distributed Computing*, pp. 84–96. Springer Verlag, 1990. (Lecture Notes on Computer Science 448).

[24] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, Jul. 1987.

[25] J. Widder, G. Le Lann, et al. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, vol. 3463 of *LNCS*, pp. 20–37. Springer Verlag, Budapest, Hungary, Apr. 2005.