Research Report 17/2007, Technische Universität Wien, Institut für Technische Informatik February 20, 2007

Towards A Real-Time Distributed Computing Model

Heinrich Moser
Vienna University of Technology
Embedded Computing Systems Group (E182/2)
A-1040 Vienna (Austria)
moser@ecs.tuwien.ac.at

February 20, 2007

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 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 drift- and failure-free 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, this is 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 algorithms and lower bounds.*

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

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 [Lyn96].

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 queuing phenomenons, which depend not only on the actual message arrival pattern but also on the queuing/scheduling discipline employed. The real-time systems community has established powerful techniques for analyzing such effects [SAA$^+$04], 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 [SLWL90, LL84b] 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 [LL84b], this is 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:[2]

(1) In Section 4, 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 ($\underline{\mathcal{M}}$) [LL84b] by just providing atomic computing steps of non-zero duration.

(2) In Section 6, 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 8, we revisit deterministic internal clock synchronization in synchronous systems [LL84b], 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 [NT93], 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 completely ignore queuing effects. Moreover, in contrast to our work, they restrict their attention to "internal problems" (essentially corresponding to our aj-problems) only.

Another example of a non-zero step time model is the the remote memory reference (RMR) model for shared-memory systems [AY96, AKH03] by Anderson et. al. It assumes computing step times which depend on 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 it is not applicable to message-passing systems, however, our results are not compareable.

Another branch of research where distributed computing and real-time systems issues are combined are modeling frameworks [AD94, LV95, LV96, MMT91, SGSAL98, KLSV03]. Such frameworks allow formal modeling and analysis of complex distributed real-time systems. A representative example are Timed I/O Automata (TIOA) [KLSV03], 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. How-

[2]Note that a prelimenary version of this paper has been published in [MS06a] and [MS06b].

ever, 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 [HLL02], 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 [LLS03, WLLS05, HW05] 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 considered by Basu and Punnekkat [BP03]. They propose simple variants of Srikanth & Toueg's clock synchronization algorithm [ST87] that can deal with scheduling latencies in heavily loaded real-time systems.

2 Preliminaries

Within this work, the notion of *causal dependency* will be used for various elements (actions, jobs, receive events, aj-events, st-events). Every such element x has an associated processor $proc(x)$. There can be two types of dependencies between these elements (cf. *happened before* relation, [Lam78]).

- *Message dependency ($x \xrightarrow{M} x'$):* One element x sends or inserts a message which is received or processed by x'. This is further formalized in the following sections.
- *Local dependency ($x \overset{L}{\longrightarrow}{}^{seq} x'$):* Both elements occur on the same processor and x appears before x' in the sequence seq, formally: $x \overset{L}{\longrightarrow}{}^{seq} x' :\Leftrightarrow proc(x) = proc(x') \wedge x \prec^{seq} x'$.

Causal dependency ($x \rightarrow^{seq} y$) is defined as the transitive closure of both types of dependency, i.e.

$$x \rightarrow^{seq} x' :\Leftrightarrow x \xrightarrow{M} x' \vee x \overset{L}{\longrightarrow}{}^{seq} x' \vee (\exists x^* : x \rightarrow^{seq} x^* \wedge x^* \rightarrow^{seq} x').$$

Definition 1. Some sequence *captures message causality* if the ordering of its elements ($\prec^{seq}$) is consistent with the message dependency relation, formally: $\forall x, x' \in seq : x \xrightarrow{M} x' \Rightarrow x \prec^{seq} x'$.

Let seq' be a reordering of some sequence seq. seq' is *causally consistent* with seq if the order of causally dependent elements is maintained, formally: $\forall x, x' \in seq : x \rightarrow^{seq} x' \Rightarrow x \prec^{seq'} x'$.

Observation 1. *If seq captures message causality, seq' is a reordering of seq and seq' is causally consistent with seq, seq is also causally consistent with seq'.*

3 Classic Computing Model

In clock synchronization research [LL84a, BW01, PSR94, AHR93, LL84b], system models are considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted "Partially Synchronous Reliable/Unreliable Models" in [SLWL90], 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 [LL84b]. Here it will be referred to as the *classic computing model.*

3.1 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. The hardware clock $HC_p : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ maps dense real-time[3] to dense clock-time; it can be read but not changed by its processor. HC_p is hence not part of the local state $state_p$, but considered separately.

The CPU is running an *algorithm*, which is specified as (a) a mapping from processor indices to a set of initial states and (b) a transition function. Processor p's set of *initial states* is denoted $Init_p$. The *transition function* takes the processor index p, one incoming message (taken from the current $intransit\_msgs$), receiver processor p's

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

current local state $oldstate$ and hardware clock reading HC_p as input, and yields a list of states and *messages to be sent*, e.g. $[oldstate, msg, int.st._1, int.st._2, newstate]$, as output. The intermediate states are usually neglected in the classic computing model, as the state transition from $oldstate$ to $newstate$ is instantaneous anyway. We explicitly 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 (by adding those to $intransit\_msgs$). Such a *computing step* (also called *message processing step*) will be called an *action* in the following. The complete action (message arrival, processing and sending messages) is performed instantly, i.e., in zero time.

Actions can actually be triggered by three different types of messages: Ordinary messages, timer messages and input 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.[4]

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.

Input messages arrive from outside the system. These messages are exempt from the requirement of having been sent by some processor in the system, and need not satisfy the delay bounds. (As the send time is unknown, this could not be verified anyway.) Usually, the problem specification (see Section 5.2.4) will define restrictions on input messages, for example, which types of input messages can arrive and their arrival pattern.

Booting 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 $t = 0$. Clearly, in our message-driven setting, at least one input message is required to trigger the first action in an execution. For simplicity, we assert that the *algorithm* may specify whether it requires only one such message or one message for each processor. We will assume that all of these *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 system-wide initial synchronization.

3.2 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 occurrence real-time $time(ac) = t$, the hardware clock value $HC(ac) = HC_p(t)$ and the state transition sequence $trans(ac) = [oldstate, \ldots, newstate]$ (including messages). 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 $states(ac)$.

As an execution is a *sequence* of actions, there is a well-defined total order $\prec^{ex}$ on actions. We will omit the superscript if it is clear from context. A message dependency ($ac \xrightarrow{M} ac'$) between two actions ac and ac' exists if $msg(ac') \in sent(ac)$.

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

- All state transitions and sent messages must be in accordance with the transition function defined in $\underline{\mathcal{A}}$.
- Processor states can only change during an action, i.e. if there are two actions $ac \prec ac'$ on the same processor p and there is no action on p between ac and ac', $newstate(ac) = oldstate(ac')$.
- The first action at every processor p must occur in an initial state of p and may—but need not—be triggered

[4] $\underline{\delta}^-$ and $\underline{\delta}^+$ are called μ and ν in [LL84b]. To disambiguate our notation, systems, parameters (like message delay bounds), and algorithms in the classic computing model are represented by underlined variables (usually $\underline{s}$, $\underline{\delta}^-$, $\underline{\delta}^+$, $\underline{\mathcal{A}}$).

by an init message. We will use $istate_p^{ex}$ to refer to the initial state of p in execution ex.

- The real-times of actions must be non-decreasing, i.e. $time(ac) < time(ac') \Rightarrow ac \prec ac'$.
- All hardware clock readings on the same processor must be consistent with the fact that hardware clock values are non-decreasing.
- 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 $HC(ac) = T$. For simplicity, we assume that algorithms do not set timers for some time less than the current hardware clock reading.
- 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 input messages (which includes init messages).

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

3.3 Systems and Admissible Executions

A *classic system* $\underline{s}$ is a system adhering to the classic computing model defined in Section 3.1, 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 2. Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. An execution is *$\underline{s}$-admissible*, if the execution comprises n processors and the message delay for each ordinary message stays within $[\underline{\delta}^-, \underline{\delta}^+]$. The execution must capture message causality.[5]

Claiming that an algorithm solves a certain 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 5). 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 hardware clocks 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.

[5]This additional condition is automatically satisfied if $\underline{\delta}^- > 0$.

4 Real-Time Computing Model

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, 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 actual 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 queuing, 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 queuing) 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.

4.1 Real-Time System Model

The system model in our real-time computing model is the same as in the classic computing model, except for the following change: A computing step in a real-time system is executed non-preemptively[6] 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 simple extension has far-reaching implications, which make the real-time computing model more realistic but also more complex. In particular, queuing 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* (i.e., currently executing a job). Since computing steps cannot be interrupted, a *queue* is needed to store ordinary, timer and input 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 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 than 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.[7] 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*

[6]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.

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

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 (recall Section 3), 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 (enqueuing) and actual message processing.

4.2 Real-time Runs

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

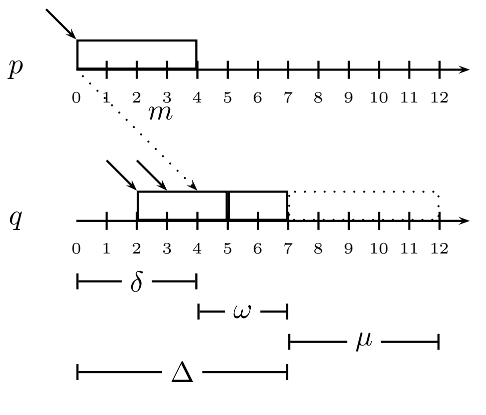
A *receive event* R for a message arriving at processor p at real-time t is a triple consisting of the processor index $proc(R) = p$, the message $msg(R)$, and the arrival real-time $time(R) = t$. Recall that t is the enqueuing time in Figure 1(b).

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 $trans(J) = [oldstate, \dots, newstate]$. $states(J)$, $sent(J)$, $oldstate(J)$ and $newstate(J)$ are abbreviations for parts of $trans(J)$ and defined analogously to the classic computing model (see Section 3.2). Let $end(J)$ be defined as $begin(J) + d(J)$.

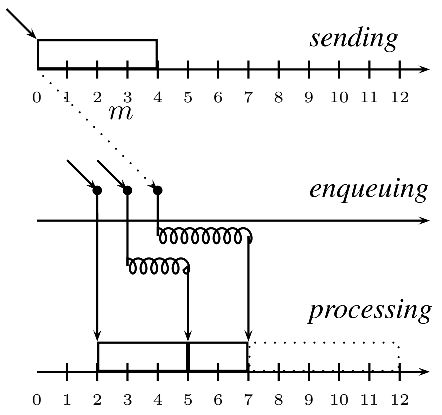
Figure 1 provides an example of a rt-run, containing three receive events and three jobs on the second processor. For example, the dashed job on the second processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \dots, newstate])$, with m being the message received during the receive event $(q, m, 4)$. 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 usually sufficient for algorithm and complexity analysis. The more detailed notion of *state transition traces* will be introduced later in Section 5.2.2.

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 $\mathcal{A}$. Processor states may only change during a job, i.e. $newstate(J) = oldstate(J')$, if $J \prec J'$, both jobs occur on the same processor p and there is no job on p in between. As in the classic computing model, the $oldstate$ of the first job on every processor must be some initial state $istate_p^{ru}$.
  - The begin times of jobs and the times of receive events must be non-decreasing, i.e. the rt-run is ordered by the (begin) times of jobs and receive events.
  - Hardware clock readings on the same processor must be non-decreasing.
  - 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 arrives when the hardware clock reads T, i.e., there is a receive event R with $HC_p(time(R)) = T$. Here, we assume that an algorithm does not set a timer during some job J for a hardware clock value less than $HC(J)$.
- *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 input messages, which are only received and processed). Receiving and processing must occur on the same processor in this order.



(a) Timing parameters for some message m



(b) Relation of message arrival and message processing, with enqueuing shown explicitly

Figure 1: Real-time computing model

A message dependency ($J \xrightarrow{M} R$) exists between a job J and a receive event R if $msg(R) \in sent(J)$. Clearly, the *global consistency* condition described above implies a local dependency between the receive event receiving a message and the job processing it. Thus there is a (transitive) causal dependency between a job sending a message and the job processing that message.

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*.

4.3 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 non-decreasing functions $\{0, \ldots, n-1\} \to \mathbb{R}^+$, representing a mapping from the number of destination processors to which ordinary messages are sent during that computing step to the actual message or processing delay bound.[8]

Example 1. 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 conditions:

- Intervals must be well-defined: $\forall \ell : \delta^-_{(\ell)} \leq \delta^+_{(\ell)} \wedge \mu^-_{(\ell)} \leq \mu^+_{(\ell)}$
- Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. This is equivalent[9] 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 non-decreasing 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 3. 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,

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

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

- *Message Delay:* The message delay (measured from $begin(J)$ to the corresponding 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 message causality.

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, hardware clocks 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 5.3].

5 Problems, Algorithms and Proofs

This section defines what it means to prove that some algorithm solves some given problem in the systems defined above. Ideally, it should be possible to specify a problem in the same way for the classic as well as for the real-time model. The following subsections present two suitable approaches.

5.1 aj-problems

Frequently, problems are specified as sets of executions. aj-problems (*action/job-based problems*) are a simple generalization of this technique. First, the data structures of actions and jobs are reduced to a common subset of attributes (called *aj-events*). A sequence of such aj-events, corresponding to an exectution or a rt-run, is called an *aj-trace*. Then, aj-problems can be specified easily as sets of aj-traces.

Definition 4 (aj-events[10])**.** The *aj-event* ev corresponding to action ac or to job J is a 4-tuple, consisting of the processor index $proc(ev) = proc(ac)/proc(J)$, the start real-time $begin(ev) = time(ac)/begin(J)$, the hardware clock value $HC(ev) = HC(ac)/HC(J)$ and the state transition sequence $trans(ev) = trans(ac)/trans(J)$.

The *action/job event trace* (*aj-trace)* of some execution or rt-run is just the sequence of aj-events corresponding to the actions/jobs. Within an aj-trace tr, there is a total ordering $\prec^{tr}$ on the aj-events, derived from the underlying execution or rt-run.

An *aj-problem* is a set of aj-traces, usually characterized by a predicate acting on some aj-trace tr. In addition, an aj-problem may specify a restriction on input messages.

Example 2 (Terminating Clock Synchronization)**.** Let $is\_lastevent(ev, p)$ be $true$ if ev is the last event on processor p, formally: $is\_lastevent(ev, p) :\Leftrightarrow proc(ev) = p \wedge \nexists ev' : ((ev \prec ev') \wedge (proc(ev') = p))$

- Precondition I: Hardware clocks do not drift.

  $\forall ev, ev' \in tr : (proc(ev) = proc(ev')) \Rightarrow$
  $(HC(ev) - HC(ev') = time(ev) - time(ev'))$

- Precondition II: Apart from the init messages, there are no input messages.

- *Termination:* All processors eventually terminate.

  $\forall p : \exists ev : is\_lastevent(ev, p)$

- *Agreement:* After all processors have terminated, all processors have adjusted clocks within γ of each other.

  $\forall p, q : \forall ev_p, ev_q \in tr :$
  $(is\_lastevent(ev_p, p) \wedge is\_lastevent(ev_q, q)) \Rightarrow$
  $|HC(ev_p) + newstate(ev_p).adj - begin(ev_p) -$
  $(HC(ev_q) + newstate(ev_q).adj - begin(ev_q))| \leq \gamma$

[10]Note that this defition of *aj-events* has nothing to do with *receive events* in rt-runs.

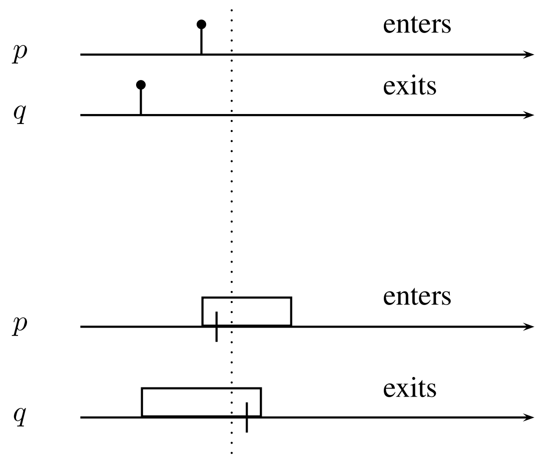


Figure 2: Example of a mutual exclusion violation in the real-time computing model (top: aj-trace, bottom: rt-run).

This example reveals that *aj-problem* specifications have some drawbacks: Predicates can only be defined for points in time where some event occurs; this is especially inconvenient for the definition of *drifting* clock synchronization (see Example 7 in Section 5.2.4). In addition, the usage of some distinguished state like $newstate$ is error-prone. Consider the following mutual exclusion condition, for example: A processor p may only enter the critical section during event ev, if $\forall q : newstate(last(q, ev)).in\_cs = false$, with $last(q, ev)$ being the last event on processor q before ev. In the classic computing model, this condition ensures mutual exclusion. In the real-time computing model, however, the situation depicted in Figure 2 can occur. While the aj-trace gives the impression that mutual exclusion is maintained, the rt-run shows that this is not always the case: As the actual state transitions can occur at any time during a job (marked as ticks in the figure), it may happen that, at a certain time (marked as a dotted vertical line), p has entered the critical section although q has not left yet.

5.2 st-problems

While aj-problems are an obvious approach for specifying problems in the models presented in this paper, they do not provide an easy way to specify predicates on "the global state of the system at time t". This is straightforward in classic models, where an action usually represents a single state transition. Actions and jobs presented in this paper, however, also involve intermediate states. This subsection presents a method to map executions to fine-grained state transition sequences. This method is general enough to be applicable to rt-runs in the real-time computing model as well, where the intermediate state transitions within a job do not necessarily occur at the same time.

5.2.1 Requirements

To provide an easy-to-apply tool for specifying problems, a model based on the global state should provide the following features:

Full time coverage To allow safety properties to be defined in a natural way, the system should be in a well-defined state at every time t, even if no state transition occurs at time t.

Full state coverage An obvious way to define a state model would be as a function $state(p, t)$ returning some well-defined (e.g. first or last) state of processor p at time t. While this approach is suitable for some types of problems, it turns out that it is not appropriate for the general case: Due to the fact that computing steps can take zero time (both in the classic computing model and in the real-time computing model if $\mu^- = 0$), multiple state transitions can occur at the same point in time. If $\underline{\delta}^-$ or $\delta^- = 0$, it is even possible for causally dependent state transitions on different processors to take place at the same real-time t. Therefore, the model should somehow support more than one global state at the same real-time t. Otherwise, information could be lost and certain properties not be satisfied anymore.

Consider, for example, an execution of a mutual exclusion algorithm in which processor p's state transitions (spread over multiple actions) *want to enter* $\rightarrow$ *enter* $\rightarrow$ *exit* $\rightarrow$ *want to enter* always occur within zero time, so that the first and the last state of p at every time t is always *want to enter*. A function $state(p, t)$ returning the first or last state of p at time t would always return *want to enter*. A liveness property ensuring that p eventually enters the critical section could never be proven correct, although the algorithm might satisfy this requirement.

Full causal coverage A function $state(p, t)$ returning the set of all possible states of p at time t would not suffice either. Consider the mutual exclusion example again and assume an execution where the following happens: p enters the critical section; p leaves the critical section and sends a message to q; upon receiving the message q enters the critical section; q leaves the critical section. All of this happens at the same time t. Clearly, without information about the causal dependency of the states at time t, it is impossible to determine whether or not the safety property that no two processors are inside the critical section simultaneously has been violated.

It might seem strange to devise a system model where "simultaneously" is more fine-grained than "at the same time t". However, being able to use 0 as the lower bound on message transmission delays and message processing times has shown to be a valuable tool in the analysis of distributed algorithms. Devising a model where such behavior is forbidden would invalidate such results and should hence be avoided.

5.2.2 State Transitions

As we will define formally in Section 5.2.3, the *global state* is composed of the local state of every processor $state_p$ and the set of not yet processed messages. We consider four distinct types of global state changes. Formally, each of these can be represented by a *state transition event* (short: *st-event*) ev with $type(ev) \in \{process, send, transition, input\}$.

- $(process : t, p, m)$:
  At time $time(ev) = t$, processor $proc(ev) = p$ starts processing message $msg(ev) = m$.

- $(send : t, p, m)$:
  At time $time(ev) = t$, processor $proc(ev) = p$ sends message $msg(ev) = m$.

- $(transition : t, p, s, s')$
  At time $time(ev) = t$, processor $proc(ev) = p$ changes its internal state from $oldstate(ev) = s$ to $newstate(ev) = s'$[11].

- $(input : t, m)$:
  At time $time(ev) = t$, input message $msg(ev) = m$ arrives from an external source.

[11]Although we will use $oldstate(ev)$ and $newstate(ev)$ to refer to the states of a $transition$ st-event, note that they do not necessarily match the $oldstate$ and $newstate$ of an action or job, as $oldstate$ and $newstate$ of a st-event might as well be intermediate states in an action or job.

In the classic computing model, every (execution, hardware clocks)-pair (ex, HC) can be mapped to a *state transition trace* (short: *st-trace*) tr, i.e., a sequence of st-events with associated hardware clocks HC_p^{tr} (the superscript is omitted if clear from context). A st-trace is created by following a simple transformation rule:

Definition 5. Each action ac at time t on processor p triggered by some message m is mapped to $(process : t, p, m)$, followed by $(send : t, p, m')$ or $(transition : t, p, s, s')$ for every message and every state transition in $trans(ac)$ (in the correct order). If m is an input message, there is a $(input : t, m)$ st-event immediately before the $process$ st-event, carrying the same time t.

A message dependency ($ev \xrightarrow{M} ev'$) between two events ev and ev' exists if $type(ev) \in \{send, input\}$, $type(ev') = process$ and $msg(ev) = msg(ev')$. As the order of the original execution is preserved, this definition implies that message causality is captured in the newly created st-trace if the execution captured message causality (by being admissible, for example).

In the real-time computing model, the mapping of a real-time run to a st-trace is similar:

Definition 6. Each job J starting at time t with duration d on processor p triggered by some message m is mapped to $(process : t, p, m)$, followed by $(send : t', p, m')$ or $(transition : t', p, s, s')$ for every message and every state transition in $trans(J)$ (in the correct order). The state transition and send times (t') must be within $[t, t + d]$ and nondecreasing.

Receive events are only mapped to the st-trace if they are caused by input messages. In that case, the receive event is mapped to $(input : t, m)$.

In the st-trace, the st-events are ordered by their time while preserving the original order of the rt-run as much as possible. The times of $send$ st-events (within $[t, t + d]$) must be chosen such that message causality is captured.[12]

Any st-events occurring at the same time t can be reordered as long as the reordering is causally consistent with the original st-trace (recall Section 2). Every such reordering results in another valid st-trace. Thus, for every execution, there is one unique set of st-events, which can be ordered into many st-traces. In the real-time computing model, however, the set of st-events corresponding to some

[12]This is automatically satisfied if $\forall \ell : \delta^-_{(\ell)} > \mu^+_{(\ell)}$.

real-time run ru is usually not unique, even if all jobs occur at different times, as the state transitions and message sends within some job can occur at different times within the job processing interval.

Example 3. Assume $\underline{\delta}^- = 0$, i.e., messages can be sent in zero time. Let ex be an execution consisting of two actions ac $(p, m_{init}, t, HC_p(t), [s_{old}, s_1, m, s_{new}])$ and ac' $(q, m, t, HC_q(t), [s'_{old}, s'_{new}])$. Figure 3 shows the st-traces corresponding to ex.

Note that rearranging these st-events is only possible because they all occur at the same real-time t. Due to the causal dependency between st-events on the same processor and between the *send* and *process* of messsage m, no other st-traces corresponding to ex exist.

5.2.3 Global States

Let the global state g be defined as a tuple $(t, s_1, \ldots, s_n, pending\_msgs)$ containing the time $time(g) = t$, the state of all processors $s_1(g) \ldots s_n(g)$ and the set of unprocessed messages $pending\_msgs(g)$ (i.e. messages in transit and messages that have been received but not processed yet). To achieve time coverage (see Section 5.2.1), we can annotate a st-trace by adding (at most countably many) sets of (either one or continuum many) global states:

- *At the beginning*:
  Insert a set $\{(t, istate_1, \ldots, istate_n, \{\}) : 0 \le t \le t'\}$, with t' being the time of the first st-event and $istate_p$ being the initial state of processor p.

- *Between every two consecutive st-events ev and ev'*:
  Insert a set $\{(t, s_1, \ldots, s_n, pending\_msgs) : time(ev) \le t \le time(ev')\}$ containing the global state after ev but before ev'. The effects of st-events on the global state are as follows:
  - $(process : t, p, m)$ removes m from $pending\_msgs$,
  - $(send : t, p, m)$ or $(input : t, m)$ adds m to $pending\_msgs$, and
  - $(transition : t, p, s, s')$ changes processor p's state to s'.

- *After the last st-event ev (if such an event exists)*:
  Insert a set $\{(t, s_1, \ldots, s_n, \{\}) : time(ev) \le t\}$ containing the global state after ev, i.e. the final state.

The state sets are totally ordered by time.

Example 4. Figure 4 shows the first st-trace presented in Figure 3, annotated by the generated state sets.

Note that this sequence of st-events alternating with global states bears a strong resemblance with the hybrid sequences of Timed I/O Automata [KLSV03]; still, the only trajectory is time t here.

Let $gstates(tr)$ denote the set of all global states appearing in the annotated st-trace tr. The annotated st-trace implies a total order $\prec^{tr+}$ on the set of all st-events and all global states, i.e. on the set $tr \cup gstates(tr)$. Note that $\prec^{tr+}$ is an extension of the order $\prec^{tr}$ defined in the previous subsection. We will again omit the superscript if it is clear from context.

5.2.4 Problem Definitions

A *state-based problem* (short: *st-problem*) is defined as a set of st-traces. Usually it is specified as a predicate on some st-trace tr and its associated hardware clocks HC_p^{tr} of the form "*preconditions* $\Rightarrow$ *safety* and *liveness* properties". An algorithm solves a given st-problem if all st-traces of all executions/rt-runs of this algorithm satisfy this predicate (see Section 5.3 for details).

The following definitions are helpful in the specification of st-problems. Let P be a predicate on st-events:

- $last(P, ev)$: the last st-event ev' satisfying P with $ev' \prec ev$ (or $\perp$, if no such st-event exists).

- $count(P, ev)$: the number of st-events ev' satisfying P with $ev' \prec ev \lor ev' = ev$.

Example 5 (Mutual Exclusion)**.** We define the predicate $is\_enter(ev) :\Leftrightarrow type(ev) = transition \land oldstate(ev).in\_cs = false \land newstate(ev).in\_cs = true$, with is_exit defined analogously. Likewise, we define $is\_want\_to\_enter(ev, p) :\Leftrightarrow type(ev) = input \land msg(ev).destination = p \land msg(ev).content =$ "want to enter", with $is\_want\_to\_exit(ev, p)$ defined analogously.

- Precondition: Apart from init messages, there are only "want to enter" and "want to exit" input messages.

  $\forall ev \in tr : (type(ev) = input) \Rightarrow (msg(ev).content =$ "init" or "want to enter" or "want to exit")

To ease presentation, the st-traces are presented in tabular form. For example, the first table corresponds to the following sequence: $(input : t, m_{init}), (process : t, p, m_{init}), (transition : t, p, s_{old}, s_1)$, $(send : t, p, m), (transition : t, p, s_1, s_{new}), (process : t, q, m), (transition : t, q, s'_{old}, s'_{new})$.

	input m_{init}						
p		process m_{init}	transition s_{old}, s_1	send m	transition s_1, s_{new}		
q						process m	transition s'_{old}, s'_{new}

	input m_{init}						
p		process m_{init}	transition s_{old}, s_1	send m		transition s_1, s_{new}	
q					process m		transition s'_{old}, s'_{new}

	input m_{init}						
p		process m_{init}	transition s_{old}, s_1	send m			transition s_1, s_{new}
q					process m	transition s'_{old}, s'_{new}	

Figure 3: Example of three st-traces.

$$\begin{gathered}
\left\{ \begin{pmatrix} 0 \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix}, \cdots, \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, (input : t, m_{init}), \left\{ \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{m_{init}\} \end{pmatrix} \right\}, (process : t, p, m_{init}), \left\{ \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, \\
(transition : t, p, s_{old}, s_1), \left\{ \begin{pmatrix} t \\ s_1 \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, (send : t, p, m), \left\{ \begin{pmatrix} t \\ s_1 \\ s'_{old} \\ \{m\} \end{pmatrix} \right\}, \\
(transition : t, p, s_1, s_{new}), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{old} \\ \{m\} \end{pmatrix} \right\}, (process : t, q, m), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, \\
(transition : t, q, s'_{old}, s'_{new}), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{new} \\ \{\} \end{pmatrix}, \cdots \right\}
\end{gathered}$$

Figure 4: Example of an annotated st-trace, containing both st-events and global states

- *Mutual Exclusion:* There is always at most one processor in the critical section.

  $\forall g \in gstates(tr) : |\{p : s_p(g).in\_cs = true\}| \leq 1$

- *Liveness I:* If a processor wants to enter the critical section, it will eventually be inside.

  $\forall p : \forall ev \in tr : is\_want\_to\_enter(ev, p) \Rightarrow (\exists g \succ ev : s_p(g).in\_cs = true)$

- *Liveness II:* If a processor wants to exit the critical section, it will eventually be outside.

  $\forall p : \forall ev \in tr : is\_want\_to\_exit(ev, p) \Rightarrow (\exists g \succ ev : s_p(g).in\_cs = false)$

- *Safety:* Do not enter or exit the critical section without a reason.
  $\forall ev \in tr : count(is\_enter, ev) \leq count(is\_want\_to\_enter, ev) \wedge count(is\_exit, ev) \leq count(is\_want\_to\_exit, ev)$

Example 6 (Terminating (Drift-Free) Clock Synchronization [LL84b]). We define $is\_finalstate(g) :\Leftrightarrow \forall g' \succ g : \forall p : s_p(g) = s_p(g')$. Let the *adjusted clock value* $AC_p(g)$ be defined as $HC_p^{tr}(time(g)) + s_p(g).adj$.

- Precondition I: Hardware clocks do not drift:

  $\forall p, t, t' : HC_p^{tr}(t) - HC_p^{tr}(t') = t - t'$

- Precondition II: Apart from the init messages, there are no input messages.

  $\forall ev \in tr : (type(ev) = input) \Rightarrow (msg(ev).content =$ "init"$)$

- *Termination:* All processors eventually terminate.

  $\exists g \in gstates(tr) : is\_finalstate(g)$

- *Agreement:* After all processors have terminated, all processors have adjusted clocks within γ of each other.

  $\forall g \in gstates(tr) : is\_finalstate(g) \Rightarrow (\forall p, q : |AC_p(g) - AC_q(g)| \leq \gamma)$

Example 7 (Drifting Clock Synchronization [AW04]). $AC_p(g)$ is defined as in the previous example.

- Precondition I: Adjusted clocks are initially synchronized within B.

  $\forall p, q : \forall g \in gstates(tr) : (\nexists g' : g' \prec g) \Rightarrow (|AC_p(g) - AC_q(g)| \leq B)$

- Precondition II: Hardware clock drift is bounded by ρ.

  $\forall p, t, t', t > t' : (t - t')/(1 + \rho) \leq HC_p^{tr}(t) - HC_p^{tr}(t') \leq (t - t')(1 + \rho)$

- Precondition III: All processors start processing at time 0.

  $\forall p : \exists ev \in tr : type(ev) = process \wedge time(ev) = 0 \wedge proc(ev) = p \wedge msg(ev).content =$ "init"

- Precondition IV: Apart from the init messages, there are no input messages.

  $\forall ev \in tr : (type(ev) = input) \Rightarrow (msg(ev).content =$ "init"$)$

- *Agreement:* All processors have adjusted clocks within γ of each other.

  $\forall p, q : \forall g \in gstates(tr) : |AC_p(g) - AC_q(g)| \leq \gamma$

- *Validity:* Adjusted clocks stay within a linear envelope (φ) of their hardware clocks.

  $\forall p, t : (HC_p^{tr}(t) - HC_p^{tr}(0))/(1 + \varphi) \leq AC_p(t) - AC_p(0) \leq (HC_p^{tr}(t) - HC_p^{tr}(0))(1 + \varphi)$

5.2.5 Relationship to aj-problems

Using the following algorithm, a st-trace tr^{st} can be reduced to an aj-trace tr^{aj}: Every $process$ st-event ev^{st} is mapped to an aj-event ev^{aj}, such that

- $proc(ev^{aj}) = proc(ev^{st})$
- $begin(ev^{aj}) = time(ev^{st})$
- $HC(ev^{aj}) = HC_{ev^{st}.processor}^{tr}(time(ev^{st}))$
- $trans(ev^{aj})$ can be derived from the sequence of $send$ and $transition$ st-events on this processor before the next $process$.

Thus, every aj-problem can also be specified as a st-problem containing exactly those st-traces that

- can be mapped to one aj-trace in the aj-problem and
- satisfy the input message restrictions specified in the aj-problem.

For this reason, all proofs in this paper will be conducted solely for st-problems.

5.3 Proofs

A *problem* $\mathcal{P}$ is either an aj-problem or a st-problem. We say that an execution/rt-run *satisfies* a problem if all aj-traces/all st-traces are $\in \mathcal{P}$, i.e. if all aj-traces/all st-traces satisfy the predicate that specifies the problem.

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

Definition 7 (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 8 (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 9 (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 10 (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 4, 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 11 (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 12 (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.

5.4 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 $(x_0, \dots, x_{n-1})$ 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 $(x_0, \dots, x_{n-1})$ 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.

We assume that, just like in an admissible rt-run, the receive events and jobs in a shifted rt-run are ordered by their occurrence 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[13], unless this is no longer possible (e.g. if messages travel backwards in time in the shifted rt-run).

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

5.5 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.

5.6 Time Complexity

The time complexity of some terminating 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.

6 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 into a real-time computing model algorithm (and vice versa) do exist. As a consequence, certain impossibility and lower bound results can also be translated.

One direction (Section 6.3), simulating a real-time system $(n, [\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+], [\mu^- = \underline{\mu}, \mu^+ = \underline{\mu}])$ on top of a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$, is quite straightforward: It suffices to implement an artificial processing delay $\underline{\mu}$, the queuing 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 $\delta^-_{(1)} \le \underline{\delta}^-$, $\delta^+_{(1)} \ge \underline{\delta}^+$ and $\mu^- \le \underline{\mu} \le \mu^+$ on top of it, thereby resulting in a correct classic computing model algorithm.

The other direction (Section 6.2), 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: First, 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 end-to-end delays $\Delta \in [\Delta^-, \Delta^+]$ (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 underlying 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 is "hidden" in the parameters of the classic computing model, but necessarily pops up when one tries to instantiate this model in a real system.

6.1 Problem Transformations

All the transformations introduced in this paper will guarantee an identical sequence of aj-events in the original algorithm and in the simulated algorithm (except for aj-events that are solely part of the simulation algorithm). It follows that any simulated rt-run corresponding to an execution (and vice versa) is equivalent w.r.t. any aj-problem. Consequently, our transformations inherently preserve all correctness and impossibility proofs w.r.t. aj-problems.

When running a real-time computing model algorithm in a classic system (Section 6.3), this also holds for st-problems (again, except for variables and messages solely used by the simulation algorithm). Unfortunately, this is not the case for transformations in the other direction, i.e. running a classic computing model algorithm in a real-time system (Section 6.2): The st-traces of a simulated execution are usually not the same as the st-traces of the corresponding rt-run. While all state transitions of some action ac at time t always occur at this time, the transitions of the corresponding job J take place at some arbitrary time between t and $t + d(J)$. Thus, there could be algorithms that solve some st-problem in the classic computing model, but fail to do so in the real-time computing model.

Fortunately, however, it is possible to show that if some algorithm $\underline{\mathcal{A}}$ solves some st-problem $\mathcal{P}$ in some classic system, $\mathcal{S}_{\underline{\mathcal{A}}}$ solves st-problem $\mathcal{P}^*_{\mu^+}$ in some corresponding real-time system. Conveniently, for some st-problems, it even holds that $\forall \mu^+ : \mathcal{P}^*_{\mu^+} = \mathcal{P}$. We will call such st-problems *shuffle-compatible problems*.

6.1.1 Shuffles

Definition 13. Let tr be a st-trace. A μ^+-*shuffle* of tr is constructed by:

1. moving $send$ or $transition$ st-events in tr at most μ^+ time units into the future (by increasing their time value and changing their position in the sequence, if needed). Every $send$ or $transition$ st-event may of course be shifted by a *different* value v, $0 \leq v \leq \mu^+$.

   If μ^+ is a function $\{0, \ldots, n-1\} \to \mathbb{R}$ rather than a number, a $send$ or $transition$ st-event ev may be moved by at most $\mu^+_{(\ell)}$ time units, with ℓ repesenting the number of $send$ st-events sending non-timer messages between the last $process$ st-event $\prec ev$ and the first $process$ st-event $\succ ev$. Intuitively, this corresponds to the number of non-timer messages sent by the action or job in the original execution.

2. moving $input$ st-events in tr arbitrarily far into the past[14].

None of these moving operations may violate causal dependency, i.e., the st-trace must be causally consistent with tr to be a valid μ^+-shuffle of tr. Causal dependency could be violated by changing the order of st-events occurring on the same processor or by causing messages to be processed before they have been sent. Since $gstates(tr)$ is a function of HC^{tr} and tr, $gstates(tr)$ changes during a shuffle. Note that HC^{tr} is not modified by the shuffling operations.

Let $shuffles(tr, \mu^+)$ be the set of all μ^+-shuffles of tr.

Observation 2. *The order of processor-local state transitions does not change, as otherwise causal dependency would be violated.*

Observation 3. *Let tr and $tr' \in shuffles(tr, \mu^+)$ be st-traces. Let g be a global state in $gstates(tr)$ and p be a processor. There is a global state g' in $gstates(tr')$ with $time(g) \leq time(g') \leq time(g) + \mu^+$ such that $s_p(g) = s_p(g')$. Informally, this means that if a processor is in a certain state in a st-trace, it will be in the same state in a shuffled st-trace, but this state might be delayed by up to μ^+ time units.*

The same holds the other way round: If a processor is in a certain state in $gstates(tr')$, it will be in the same state in $gstates(tr)$, but maybe up to μ^+ time units earlier.

Definition 14. Let $\mathcal{P}$ be a st-problem, i.e., a set of st-traces. Then $\mathcal{P}^*_{\mu^+}$ is defined as $\bigcup_{tr \in \mathcal{P}} shuffles(tr, \mu^+)$. Informally speaking, $\mathcal{P}^*_{\mu^+}$ is equivalent to $\mathcal{P}$ with the exception that the problem is still solved if an arbitrary number of message sends and state transitions may happen up to μ^+ time units later (without violating causality) and external inputs arrive earlier.

Note that, as $\mathcal{P}$ is a subset of $\mathcal{P}^*_{\mu^+}$, $\mathcal{P}^*_{\mu^+}$ is a weaker problem than $\mathcal{P}$, i.e., if some algorithm solves $\mathcal{P}$ in some system, it also solves $\mathcal{P}^*_{\mu^+}$ in the same system.

6.1.2 Simulation-Invariant Extensions

Sometimes, we will run an algorithm within some time-preserving simulation: The algorithm's state transitions are the same and occur at the same time, but the simulator needs to add its own variables. In addition, transmission of algorithm messages might be handled by the simulator instead (e.g. by wrapping them with additional information or receiving them earlier and queuing them). One such simulation will be presented in Section 6.3. We will hence restrict our attention to *simulation-compatible problems*, which do not impose any restrictions on messages (except the arrival of input messages) and that are only concerned with "their own" variables.

Let tr be a st-trace and $\mathcal{V}$ be a set of variable names. Formally, a *simulation-invariant $\mathcal{V}$-extension* of tr is constructed in the following way:

- Every state occurring in the st-trace, i.e., $oldstate$ and $newstate$ of every st-event, may be extended by variables from $\mathcal{V}$ (and their valuations).
- An arbitrary number of $process$, $send$ and $transition$ st-events may be inserted as long as they do not modify any variables other than those in $\mathcal{V}$.
- Messages appearing in $process$ and $send$ st-events may be replaced by other, arbitrary messages.

[14]For the purpose of the proof of Theorem 1, this condition can be weakened. Let ev' be the event starting the busy period (= period, where $process$ st-events are at most μ^+ time units apart). As our model assumes a non-idling scheduler, it suffices to allow the $input$ st-event ev to be moved back to any time in the interval $[time(ev'), time(ev)]$.

- The result must be a valid st-trace, i.e., every message sent must eventually be processed and every $newstate(ev)$ must correspond to $oldstate(ev')$ of the following st-event on the same processor.

A *simulation-invariant $\mathcal{V}$-extension* of some problem $\mathcal{P}$, denoted $\mathcal{P}^{>}_{\mathcal{V}}$, is defined as the set of all simulation-invariant $\mathcal{V}$-extensions of all st-traces in $\mathcal{P}$. For simplicity, we assume that $\mathcal{V}$ contains variables that are not already referenced explicitly in $\mathcal{P}$. A problem $\mathcal{P}$ where $\mathcal{P} = \mathcal{P}^{>}_{\mathcal{V}}$ for all $\mathcal{V}$ will be called *simulation-compatible*.

6.1.3 Examples

All examples in this section are simulation-compatible.

τ gap Mutual Exclusion Let $\mathcal{P}$ be the *3-second gap mutual exclusion* problem, defined by the properties in Section 5.2.4 and the additional requirement that all processors must have left the critical section for more than 3 seconds before the critical section can be entered again by some processor, i.e. $\forall ev, ev' \in tr : (is\_exit(ev) \wedge ev \prec ev' \wedge time(ev') \leq time(ev) + 3) \Rightarrow \neg is\_enter(ev')$

We claim that an algorithm solving $\mathcal{P}^*_{\mu^+}$ with $\mu^+ = 3$ seconds also solves *0-second gap mutual exclusion*. Looking ahead to Theorem 1, this means that a *3-second gap mutual exclusion* algorithm designed for a classic system can be used to solve the *0-second gap mutual exclusion* problem in some real-time system with $\mu^+ = 3$ and the other parameters determined by the feasible assignment (cf. Section 6.2).

Proof. We will show by contradiction that an algorithm solving $\mathcal{P}^*_{\mu^+}$ solves the 0-gap mutual exclusion problem. Assume a rt-run ru with st-trace tr' satisfying $\mathcal{P}^*_{\mu^+}$ where *mutual exclusion* is violated. Let g be the first global state in which two processors p and q are inside the critical section.

As ru satisfies $\mathcal{P}^*_{\mu^+}$, $tr' \in \mathcal{P}^*_{\mu^+}$. By the definition of $\mathcal{P}^*_{\mu^+}$, this means that tr' is a 3-shuffle of some st-trace $tr \in \mathcal{P}$. Thus, in tr, q is in the critical section at some time within $[t-3, t]$ and p is in the critical section at some (maybe other) time within $[t-3, t]$ (recall Observation 3). If p and q are in the critical section at the same time in tr, *mutual exclusion* is violated. Otherwise, one of them exits and the other one enters, causing the *3-second gap* condition to be violated. Both cases contradict the assumption that $\mathcal{P}$ solves *3-second gap mutual exclusion*.

Liveness I/II and *safety* in $\mathcal{P}^*_{\mu^+}$ follow directly from the same property in $\mathcal{P}$, as $enter$ and $exit$ st-events as well as local states are only moved forward w.r.t. tr (again, cf. Observation 3), whereas $want\_to\_enter$ and $want\_to\_exit$ st-events are only moved backwards w.r.t. tr. □

Causal Mutual Exclusion Let $\mathcal{P}$ be the *causal mutual exclusion* problem, defined by the properties in Section 5.2.4 and the additional requirement that every state transition in which a processor enters a critical section must causally depend on the last exit, formally $\forall ev, ev' \in tr : (ev = last(is\_exit, ev') \wedge is\_enter(ev')) \Rightarrow (ev \rightarrow ev')$.

In this case, $\mathcal{P}^*_{\mu^+} = \mathcal{P}$, i.e. causal mutual exclusion is a shuffle-compatible problem and the same algorithm used for some classic system can also be used in a real-time system with a feasible assignment.

Proof. As an algorithm solving $\mathcal{P}$ always solves $\mathcal{P}^*_{\mu^+}$, we just have to show the other direction, i.e. that an algorithm solving $\mathcal{P}^*_{\mu^+}$ solves causal mutual exclusion, to prove the equivalence. As in the previous example, *liveness I/II* and *safety* are always satisfied.

In $\mathcal{P}$, the new *exit-enter causality* and the *mutual exclusion* condition imply that there is a causal sequence $enter_p \rightarrow exit_p \rightarrow enter_q \rightarrow exit_q \rightarrow \cdots$ containing *all* enter and exit st-events. As shuffles must not violate causal dependencies, $enter_p \prec exit_p \prec enter_q \prec exit_q \prec \cdots$ still holds for all st-traces in $\mathcal{P}^*_{\mu^+}$, guaranteeing that neither the *mutual exclusion* nor the *exit-enter causality* condition is violated in $\mathcal{P}^*_{\mu^+}$. □

Terminating Clock Synchronization Let $\mathcal{P}$ be the *terminating clock synchronization* problem, defined by the conditions in Section 5.2.4. $\mathcal{P}$ is a shuffle-compatible problem.

Proof. As termination is guaranteed in every st-trace of $\mathcal{P}$, every μ^+-shuffle of that st-trace terminates at most μ^+ time units later.

Assume by contradiction that agreement is violated in some μ^+-shuffle tr' of a st-trace tr of $\mathcal{P}$. Let g be the first global state in which agreement between some processors p and q is violated. Clearly, g must be after termination. Thus, the adjustment values of p and q must be the same as the ones in all terminated states of tr. However, as both tr and tr' reference the same hardware clocks, this is a contradiction. $\square$

aj-problems Every aj-problem can be specified as a st-problem with restrictions solely on $process$ and $input$ st-events (cf. Section 5.2.5). As $process$ st-events are not changed by shuffles, every aj-problem whose input message restrictions are not violated by shifting $input$ st-events backwards in time is a shuffle-compatible problem.

6.2 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 solving some problem $\mathcal{P}$ into a real-time computing model algorithm solving $\mathcal{P}^*_{\mu^+}$ (cf. Section 6.1.1).

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 $\mathcal{S}_{\underline{\mathcal{A}}}$ 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, $\mathcal{S}_{\underline{\mathcal{A}}}$ can be expressed as given in Figure 5.

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 underlying 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 transformed 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 [SAA+04] for an overview. For the purpose of this paper, quite trivial considerations are sufficient: A trivial end-to-end delay lower bound Δ^- is $\delta^-_{(1)}$. 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 7 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 $[\underline{\delta}^-, \underline{\delta}^+]$. Additionally, the assignment must guarantee that not only the reception but also the processing of timer messages scheduled for some hardware clock value T starts at that time, i.e. that $HC(J) = T$ is satisfied for the job J processing the timer message.

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. Such a feasible assignment may not exist for some (algorithm, real-time system) pairs.

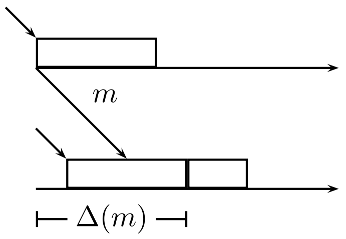
We should mention, however, that the adversary faced by $\underline{\mathcal{A}}$ when employed in $\mathcal{S}_{\underline{\mathcal{A}}}$ is somewhat restricted: The unrestricted adversary in the classic computing model can choose any value between $\underline{\delta}^-$ and $\underline{\delta}^+$, 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 algorithm designed for some classic system $\underline{s}$ solving some problem $\mathcal{P}$ can also solve problem $\mathcal{P}^*_{\mu^+}$ in s if a feasible assignment for $[\underline{\delta}^-, \underline{\delta}^+]$ exists. Figure 6 outlines the principle of our simulation.

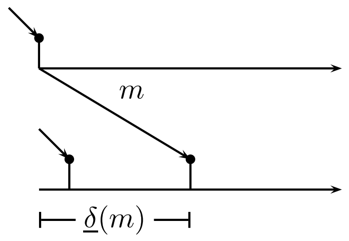
```
<global variables of $\underline{\mathcal{A}}$>

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

Figure 5: 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 $\underline{A}$ (satisfying $\mathcal{P}$).



- Every st-trace of ru is a μ^+-shuffle of a st-trace of $ex \Rightarrow ru$ satisfies $\mathcal{P}^*_{\mu^+}$.

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

Figure 6: Transformation outline (Theorem 1)

Theorem 1. *Let* $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ *be a real-time system and* $\mathcal{P}$ *be a problem. If there exists an algorithm* $\underline{A}$ *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 5 solves* $\mathcal{P}^*_{\mu^+}$ *in s.*

Proof. Assume such an algorithm $\underline{A}$ 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 $\Delta^- \geq \underline{\delta}^-$ and $\Delta^+ \leq \underline{\delta}^+$ since the assignment is feasible. We will now show that the implication ($\mathcal{S}_{\underline{A}}$ solves $\mathcal{P}^*_{\mu^+}$ in s) is justified. According to Definition 9 we have to show that every s-admissible rt-run of $\mathcal{S}_{\underline{A}}$ satisfies $\mathcal{P}^*_{\mu^+}$.

Corresponding execution: Let ru be such an s-admissible rt-run. We can now create a corresponding execution ex of $\underline{A}$ 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 occurrence 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. The feasible assignment also guarantees that timer messages are processed at their designated hardware clock time. Thus, ex is a valid execution: All requirements specified in Section 3.2 are met by ex.

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 input or timer message, it does not need to obey any bounds. If m is an ordinary 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 occurring at some real-time within $[t - \Delta^+, t - \Delta^-]$. Therefore, m's message delay in $\underline{s}$ ranges between $\Delta^- \geq \underline{\delta}^-$ and $\Delta^+ \leq \underline{\delta}^+$. As m obeys the $[\underline{\delta}^-, \underline{\delta}^+]$ bounds, $ex(i)$ is $\underline{s}$-admissible.

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

ru satisfies $\mathcal{P}^*_{\mu^+}$: To show that ru satisfies $\mathcal{P}^*_{\mu^+}$, we must show that every st-trace tr' of ru is a μ^+-shuffle of a st-trace tr of ex. We can construct tr from tr' as follows:

- Move the time of every $send$ and $transition$ st-event back to the time of their corresponding $process$ st-event. The $send$ and $transition$ st-events belonging to the same job should directly follow their $process$

st-event and the order of these $process$, $send$ and $transitions$ st-events must not change (of course, the order w.r.t. st-events of other jobs will change). tr is still causally consistent with tr' (see Sections 2 and 5.2.2), as the processor-local order of st-events is not changed, $process$ st-events are not moved and $send$ st-events are only moved backwards in time.

- Move the time of every $input$ st-event forward so that it has the same time as its corresponding $process$ st-event processing the input message. The $input$ st-event must directly precede the $process$ st-event. Clearly, this does not violate causal consistency with tr' either.

As these operations are an inverse subset of the μ^+-shuffle operations (see Definition 13), tr' is a μ^+-shuffle of tr. Still, we need to show that tr is a st-trace of ex:

- *Every action in ex is correctly mapped to st-events in tr:* Every job J in ru is mapped to an action ac in ex and a sequence of one $process$, multiple $send$/$transition$ and at most one $input$ st-event in tr. Following Definitions 5 and 6, there are two differences in the mapping of some job J to st-events and the corresponding action ac to st-events:
  - The $process$, $state$ and $transition$ st-events all occur at the same time $time(ac)$ when mapping an action. The construction of tr ensures that this is the case.
  - The $input$ st-event sending the message processed by the action occurs at the same time as the $process$ st-event processing it. This is also ensured by the construction of tr.

- *Every st-event in tr belongs to an action in ex:* Every st-event in tr' (and, thus, every corresponding st-event in tr) is based on either a job or an input message receive event in ru. By construction of ex, every job is mapped to one action, requiring the same amount of $process$, $send$ and $transition$ st-events. Every input message receive event in ru results in an $input$ st-event. By Definition 5, this $input$ st-event belongs to the action processing it.

- *Causal consistency:* Follows directly from the consistency of tr' and the fact that the construction of tr does not violate causal consistency.

Thus, we can conclude that tr is a st-trace of ex. As $\underline{\mathcal{A}}$ solves $\mathcal{P}$ in $\underline{s}$ and ex is an execution of $\underline{\mathcal{A}}$ in $\underline{s}$, $tr \in \mathcal{P}$. Similarly, tr' is a μ^+-shuffle of tr; therefore, tr' in $\mathcal{P}^*_{\mu^+}$. As this holds for every st-trace tr' of every s-admissible rt-run ru of $\mathcal{S}_{\underline{\mathcal{A}}}$, $\mathcal{S}_{\underline{\mathcal{A}}}$ solves $\mathcal{P}^*_{\mu^+}$ in s. $\square$

6.3 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 result for some classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ holds for all real-time systems $(n, [\delta^- \le \underline{\delta}^-, \delta^+ \ge \underline{\delta}^+], [\mu^-, \mu^+])$ for arbitrary μ^-, μ^+ as well, because the additional delay does not provide the algorithm with any useful information. For simulation-compatible problems (recall Section 6.1.2), this is true, and we will prove it by using yet another simulation, this time the other way round. Note that, contrary to the previous section, we do not require a scheduling analysis to obtain a feasible assignment here, and the problem transformation $\mathcal{P} \to \mathcal{P}^{>}_{\mathcal{V}}$ is much less restrictive than $\mathcal{P} \to \mathcal{P}^*_{\mu^+}$.

Figure 7 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 queuing, scheduling, and execution of real-time model computing steps (jobs) of duration $\underline{\mu}$, and can hence be parameterized with some function $\underline{\mu} : \{0, \ldots, n-1\} \to \mathbb{R}^+$ and some real-time computing model algorithm $\mathcal{A}$. It works as follows: At every point in time, the simulated processor is either *idle* (local variable $idle = true$) or *busy* ($idle = false$). Initially, the processor is idle. As soon as the first algorithm message arrives (line 13), the processor becomes busy and waits for $\underline{\mu}_{(\ell)}$ time units (ℓ being the number of ordinary messages sent during that computing

```
1   var queue ← {}
2   var idle ← $true$
3   < global variables of $\mathcal{A}$>
4
5   function $\underline{\mathcal{S}}_{\mu,\mathcal{A}}$−process_message(msg, time)
6     if msg = "finished-processing"
7       if queue is empty   /* type (d) */
8         idle ← $true$
9       else                /* type (c) */
10        process_next (time)
11    else
12      if idle             /* type (a) */
13        queue.add(msg)
14        process_next (time)
15      else                /* type (b) */
16        queue.add(msg)
17
18  function process_next (time)
19    var msg ← choose item from queue according
20           to some arbitrary scheduling policy
21    queue.remove(msg)
22    $\mathcal{A}$−process_message(msg, time)
23    idle ← $false$
24    $\ell$ ← number of ordinary messages sent by $\mathcal{A}$
25    set timer "finished-processing" for time + $\underline{\mu}_{(\ell)}$
```

Figure 7: Simulation algorithm (real-time computing model atop of classic computing model)

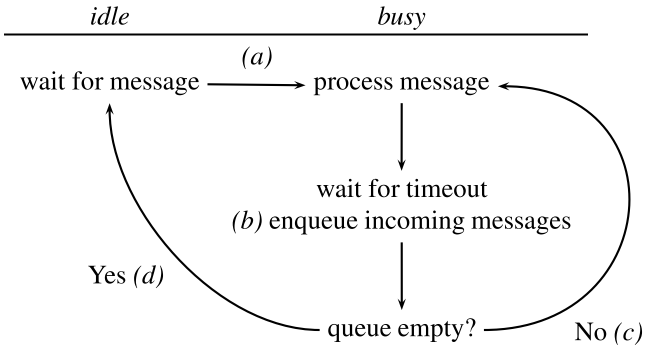


Figure 8: State diagram (algorithm in Figure 7)

step). All algorithm messages arriving during that time are enqueued (line 16). After these $\underline{\mu}_{(\ell)}$ time units have passed, the queue is checked. If it is empty, the processor returns to its idle state (line 8); otherwise, the next message is processed (line 10). The resulting state diagram is shown in Figure 8; Figure 9 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*

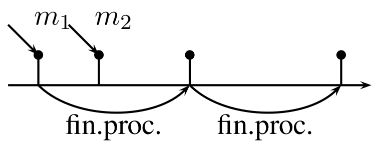
- $\forall \ell : \delta^-_{(\ell)} \le \underline{\delta}^-$ *and* $\delta^+_{(\ell)} \ge \underline{\delta}^+$,

$\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ given in Figure 7 solves $\mathcal{P}^{>}_{\mathcal{V}}$ in $\underline{s}$.

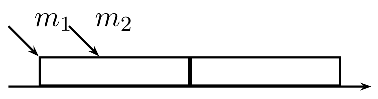
Proof. Assume such an algorithm $\mathcal{A}$ exists. We will show that the implication ($\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ solves $\mathcal{P}^{>}_{\mathcal{V}}$ in $\underline{s}$) is justified. According to Definition 7 we have to show that every $\underline{s}$-admissible execution of $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ satisfies $\mathcal{P}^{>}_{\mathcal{V}}$.

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. Fig-

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 $\mathcal{P}$).



- Every st-trace of ex is a simulation-invariant $\mathcal{V}$-extension of a st-trace of $ru \Rightarrow ex$ satisfies $\mathcal{P}^{>}_{\mathcal{V}}$.

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

Figure 9: Transformation outline (Theorem 2)

ure 8): (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 hardware clocks 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:

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

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

$$\begin{aligned} &proc(R) \leftarrow proc(ac) && time(R) \leftarrow time(ac) \\ &msg(R) \leftarrow msg(ac) && \end{aligned}$$

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

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

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

To illustrate this transformation, Figure 9 shows an example with actions of types (a), (b), (c) and (d) occurring 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 the empty set), 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 occurred 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 4.2) 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 $\underline{\mu}_{(\ell)} = \overline{\mu}_{(\ell)}$. 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 a rt-run and is s-admissible:

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

Proof. We will show the properties defined in Section 4.2 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, on every processor these calls occur in the same order 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 are still non-decreasing.
  - 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 occurring 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 ac at some time t with $HC(ac) = T$. 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 m received at time t_r. By design of the simulation algorithm, the action ac receiving m 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 m 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 occurring 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. □

- *Causality:* Follows directly from causality of ac.

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 9).

W.r.t. st-problems, let tr' be a st-trace of ex.

Lemma 6. *Let $\mathcal{V} = \{queue, idle\}$. tr' is a simulation-invariant $\mathcal{V}$-extension of a st-trace tr of ru (cf. Section 6.1.2).*

Proof. We can construct tr out of tr' by:

1. Remove the variables $queue$ and $idle$ from all states.
2. Remove any $transition$ st-events with $oldstate = newstate$, i.e. any $transition$ st-events that only manipulated $queue$ and/or $idle$.
3. Let ac and ac' be corresponding type (b) and (c) actions. Let ev and ev' be the $process$ st-events corresponding to ac and ac'. Remove ev and change $msg(ev')$ to $msg(ev)$, the message processed in ev, rather than the "finished-processing" message.
4. Remove all $process$ and $send$ st-events that are processing or sending "finished-processing" messages.

Note that tr is a valid st-trace: Only $transition$ st-events that have no effect have been removed. In step 3, receptions of algorithm messages are only moved forward in time, as $ac \prec^{ex} ac'$. Although $process$ st-events processing "finished-processing" messages are removed or replaced in steps 3 and 4, the corresponding $send$ st-events are removed as well (in step 4).

According to Section 6.1.2, this implies that tr' is a simulation-invariant $\mathcal{V}$-extension of tr. We now need to show that tr is a st-trace of ru.

- *Every job in ru is correctly mapped to st-events in tr:* Every job J in ru is based on either a type (a) or a type (c) action ac in ex. Following Definitions 5 and 6, the st-events produced by mapping ac are the same as the st-events produced by mapping J, with the following differences:
  - The st-events mapped by ac contain the simulation variables. However, they have been removed by the transformation from tr' to tr.
  - If ac is a type (c) action, its $process$ st-event processes a "finished-processing" message rather than the algorithm message received in the corresponding type (b) action. The creation of tr (step 3) also ensures that the correct message is used in tr.
  - If ac is a type (a) action and $msg(ac)$ is an input message, there is an additional $input$ st-event before the $process$ st-event. By construction of ru, however, there is a receive event at the time of the type (a) action corresponding to the $input$ st-event in tr.

- *Every input message receive event in ru is correctly mapped to an $input$ st-event in tr:* Every receive event in ru is based on either a type (a) or type (b) action. Both result in an $input$ st-event in tr' if the received message was an input message. By construction of tr, these $input$ st-events still exist in tr.

- *Every st-event in tr belongs to a job or input message receive event in ru:* Every st-event in tr' (and, thus, every st-event in tr) is based on an action ac in ex.
  - Type (a): The st-events in tr' contain the $send$ and the $transition$ st-events of $\mathcal{A}$`-process_message(msg, time)` and additional steps taken by the simulation algorithm. The transformation from tr' to tr ensures that these additional steps (and only these) are removed. Thus, the remaining st-events in tr correspond to the job J corresponding to ac. If the message received by ac was an input message, the $input$ st-event corresponds to the receive event created during the transformation $ex \rightarrow ru$.

- Type (b): This type of action only performs state transitions w.r.t. simulation variables. Thus, the only st-event left over after the transformation from tr' to tr ($send$ and $transition$ st-events removed during steps 1 and 2, $process$ st-event removed during step 3) is one $input$ st-event, if the received message was an input message. This $input$ st-event corresponds to the receive event created by the transformation from ex to ru.
- Type (c): As in type (a) actions, the transformation from tr' to tr ensures that only the $send$ and the $transition$ st-events of $\mathcal{A}$`-process_message(msg, time)` are left, with msg being the message received in the corresponding type (b) action. The transformation from tr' to tr ensures that the $process$ st-event in tr contains msg as the received message.
- Type (d): Only state transitions involving simulation variables are performed. All of these $transition$ st-events are lost during the creation of tr. As the $process$ st-event processes a "finished-processing" message, it is removed as well.

Thus, tr is a st-trace of ru. □

As $\mathcal{A}$ solves $\mathcal{P}$ in s and ru is a rt-run of $\mathcal{A}$ in s, $tr \in \mathcal{P}$. Similarly, tr' is a simulation-invariant $\mathcal{V}$-extension of tr; therefore, $tr' \in \mathcal{P}^{>}_{\mathcal{V}}$. As this holds for every st-trace tr' of every $\underline{s}$-admissible execution ex of $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$, it holds that $\underline{\mathcal{S}}_{\mu^-,\mathcal{A}}$ solves $\mathcal{P}^{>}_{\mathcal{V}}$ in s, which concludes our proof of Theorem 2. □

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* $\delta^+_{(1)} \geq \underline{\delta}^+$,

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

Proof. This proof and the detailed algorithm have been omitted, but an informal description is given as follows: First, note that $\delta^+_{(1)} \geq \underline{\delta}^+ \Leftrightarrow \forall l : \delta^+_{(\ell)} \geq \underline{\delta}^+$. Thus, the extended simulation algorithm mainly allows $\delta^-_{(\ell)}$ to be greater than $\underline{\delta}^-$ for $\ell > 1$. However, since $\varepsilon_{(\ell)} \geq \varepsilon_{(1)}$ (see Section 4.3), 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 by a job sending ℓ ordinary messages in total: 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 the value ℓ. □

7 First Results on Clock Synchronization

The remainder of this work will concentrate on the *terminating clock synchronization* problem in the drift- and failure-free case (cf. Section 5.2.4).

In the classic computing model, a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ has been proved in [LL84b] 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 to within $(1 - \frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ is possible.*

Proof. In the algorithm of [LL84b], 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 $\mathcal{S}_{\underline{A}}$ 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 $\underline{A}(n, \Delta^-, \Delta^+)$ will synchronize clocks to within $\gamma = (1-\frac{1}{n})(\Delta^+ - \Delta^-)$ in the classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$. As $\underline{A}$ 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 $\mathcal{S}_{\underline{A}}$ provides clock synchronization to 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. □

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 $O(()1)$, the worst-case time complexity in the real-time computing model is hence $\Theta(n)$.

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 $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ to within $\gamma < (1-\frac{1}{n})\varepsilon_{(1)}$. Applying Theorem 3 would imply that $\underline{\mathcal{S}'}_{\delta^-,\mu^-,\mathcal{A}}$ provides clock synchronization to 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 [LL84b]. □

8 Algorithms Achieving Optimal Precision

The comparison of Theorems 4 and 5 raises the obvious question of 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 [LL84b] 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.

8.1 Generalization of Existing Results

Two lemmata from [LL84b] can be generalized to our setting:

Lemma 7. *If q receives a timestamped message from p with end-to-end delay*[15] *uncertainty ε_Δ, q can estimate p's hardware clock value within an error of at most $\frac{\varepsilon_\Delta}{2}$.*

Proof. (Similar to Lemma 5 of [LL84b]). 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 q can calculate some E such that $|E - D| \leq \frac{\varepsilon_\Delta}{2}$. Let Δ^- and Δ^+ 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., $\Delta = \Delta^- + \frac{\varepsilon_\Delta}{2}$. 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.

[15] 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.

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

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

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 to within $(1-\frac{1}{n})err$ is possible.*

Proof. The proof can be obtained by a simple adaption of Theorem 7 of [LL84b] to a general err. □

8.2 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 7, 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 10, 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 synchronize 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 necessitate a second round of message exchanges.

Lemma 9. *No queuing occurs when running the algorithm in Figure 10.*

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

Proof. By 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 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 $t - \dot{\delta}^-$. Processor 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 10 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 10 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 to within $(1 - \frac{1}{n})\dot{\varepsilon}$.

With respect to message and time complexity, the algorithm obviously performs exactly n broadcasts, and the worst-case 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 $\delta^+_{(\ell)}$, $\varepsilon_{(\ell)}$ and $\mu^+_{(\ell)}$ w.r.t. ℓ. □

8.3 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 to 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 11):

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

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

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 $\{0, \ldots, i-1\}$ 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 $\{i+1, \ldots, n-1\}$ 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 occurring on i, initiating the send of $\text{TIME}_{i \to j}$) and WAIT_i (=

$\textsc{time}_{i-1\to i}$, because it initiates the wait phase). $begin(\ldots)$ denotes the beginning of the corresponding job processing the message. To ease analysis, we assume a "virtual" no-op job $\textsc{wait}_0$ with begin time $begin(\textsc{wait}_0) = begin(\textsc{send}_{0,\to 1}) - W$.

See Figure 12 for an example. Note that every processor sends exactly one $\textsc{time}$ message to every other processor.

Lemma 10. *The following invariant holds for all t when running the algorithm in Figure 11: All messages received up to time t on some processor i have been received in the following order: $\langle \textsc{time}_{0\to i}, \ldots, \textsc{time}_{i-1\to i} = \textsc{wait}_i, \textsc{send}_{i,\to i\oplus 1}, \ldots, \textsc{send}_{i,\to i\oplus(n-1)}, \textsc{time}_{i+1\to i}, \ldots, \textsc{time}_{n-1\to i}\rangle$. 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 $\textsc{send}$ jobs on the same processor are exactly $\dot{\mu}^+$ time units apart. $\textsc{send}_{i,\to i\oplus 1}$ arrives at $begin(\textsc{wait}_i) + W$.

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 $\textsc{time}_{j\to i}$ arrives at $t < begin(\textsc{time}_{j-1\to i}) + \dot{\mu}^+$. $\textsc{time}_{j\to i}$ has been sent no later than $t - \dot{\delta}^-$ by j's $\textsc{send}_{j,\to i}$ 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 $(\textsc{send}_{j,\to j\oplus 1}, \ldots, \textsc{send}_{j,\to i\ominus 1})$ and $\textsc{send}_{j,\to i}$ are exactly $\dot{\mu}^+$ time units apart, and $\textsc{wait}_j$ starts at least $\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+$ time units before the first send phase step. This means that

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

  $\textsc{wait}_j = \textsc{time}_{j-1\to j}$ has been sent during $j-1$'s $\textsc{send}_{j-1,\to j}$ job. Thus,

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

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

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

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

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

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

- *Wait $\to$ send phase*: Assume the $\textsc{send}_{i,\to i+1}$ timer message arrives at $t \ne begin(\textsc{wait}_i) + W$. As the $\textsc{send}_{i,\to i+1}$ timer is set in $\textsc{wait}_i$ to W, this is a contradiction.

- *Send phase*: Assume for $i \ne j$ and $i \ne j \oplus 1$ that $\textsc{send}_{i,\to j\oplus 1}$ arrives at $t \ne \textsc{send}_{i,\to j} + \dot{\mu}^+$. As the $\textsc{send}_{i,\to j\oplus 1}$ timer is set in $\textsc{send}_{i,\to j}$ to $\dot{\mu}^+$, this is a contradiction.

- *Send $\to$ second part receive phase*: Assume for $i < n - 1$ that $\textsc{time}_{i+1\to i}$ arrives at $t < begin(\textsc{send}_{i,\to i\oplus(n-1)}) + \dot{\mu}^+$ (= begin time of i's last send job $+\dot{\mu}^+$). $\textsc{time}_{i+1\to i}$ was sent during $\textsc{send}_{i+1,\to i} = \textsc{send}_{i+1,\to(i+1)\oplus(n-1)}$, which started no later than $t - \dot{\delta}^-$. As the invariant holds for all arrival times $< t$, $\textsc{send}_{i+1,\to(i+1)\oplus 1}$ started no later than $t - \dot{\delta}^- - (n-2)\dot{\mu}^+$. This means that $\textsc{wait}_{i+1} = \textsc{time}_{i\to i+1}$ started no later than $t - \dot{\delta}^- - (n-1)\dot{\mu}^+$ and $\textsc{time}_{i\to i+1}$ was sent (by job $\textsc{send}_{i,\to i+1}$) no later than

  $$begin(\textsc{send}_{i,\to i+1}) \le t - 2\dot{\delta}^- - (n-1)\dot{\mu}^+$$

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

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

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

□

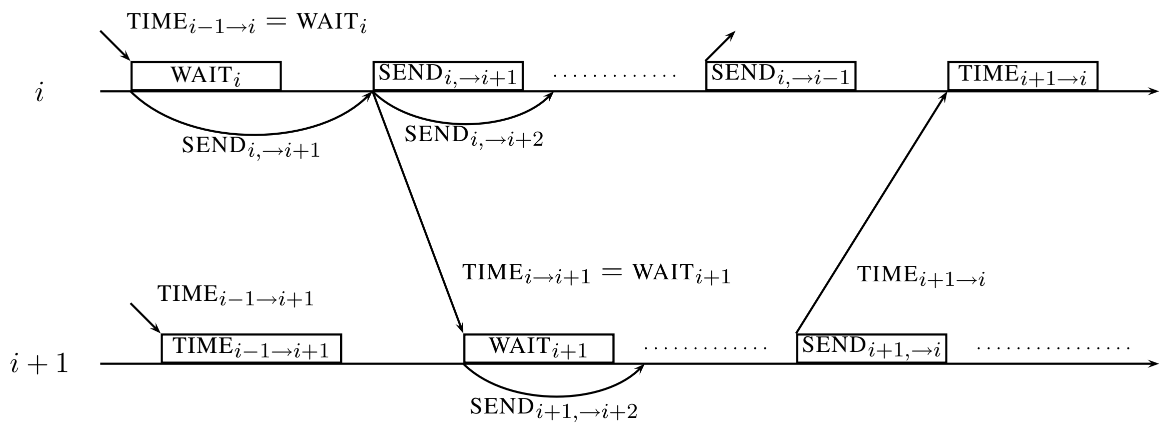

$\text{TIME}_{i-1\to i} = \text{WAIT}_i$
i
WAIT_i
$\text{SEND}_{i,\to i+1}$
$\text{SEND}_{i,\to i-1}$
$\text{TIME}_{i+1\to i}$
$\text{SEND}_{i,\to i+1}$
$\text{SEND}_{i,\to i+2}$
$\text{TIME}_{i\to i+1} = \text{WAIT}_{i+1}$
$\text{TIME}_{i+1\to i}$
$\text{TIME}_{i-1\to i+1}$
$i+1$
$\text{TIME}_{i-1\to i+1}$
WAIT_{i+1}
$\text{SEND}_{i+1,\to i}$
$\text{SEND}_{i+1,\to i+2}$

Figure 12: 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.

We can apply Lemma 7 to the algorithm of Figure 11 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 to within $(1-\frac{1}{n})\varepsilon_{(1)}$ can be achieved. As all job durations and message delays are independent of n this time ($\delta^+_{(1)}$ rather than $\delta^+_{(n-1)}$, etc.), the time complexity of this algorithm is $\mathrm{O}(n)$.

9 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 [BW01]. They have shown that no algorithm can achieve better precision than $\frac{1}{2}diam(G)$ for any communication network G, with $diam(G)$ 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 [LL84b] can no longer be achieved.

Unfortunately, the proof from [BW01] 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 [BW01]) can be used, however.

Additionally, we will show that the message and time complexity of clock synchronization to 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 synchronization to 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 ℓ messages is $\mu^+_{(\ell)}$.

9.1 Message Graph Diameter

Definition 15. 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 13 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, $|\Delta| \leq c \cdot \varepsilon_{(1)}$ and $|\Delta'| \leq c \cdot \varepsilon_{(1)}$.

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 $\leq c \cdot \varepsilon_{(1)}$, we have the following inequalities:

$$\begin{aligned} |\Delta| &\leq c \cdot \varepsilon_{(1)} \\ |\Delta + D \cdot \varepsilon_{(1)}/2| &\leq c \cdot \varepsilon_{(1)} \\ |\Delta - D \cdot \varepsilon_{(1)}/2| &\leq c \cdot \varepsilon_{(1)} \end{aligned}$$

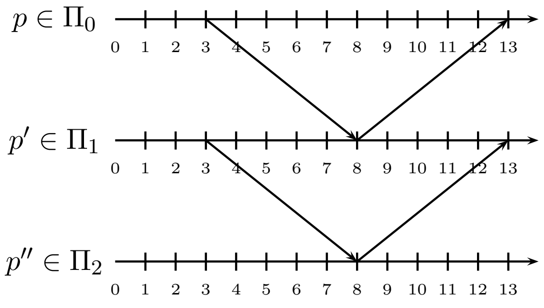
which imply that $c \geq D/2$ and provide the required contradiction to $D > 2c$. □

9.2 Message Complexity

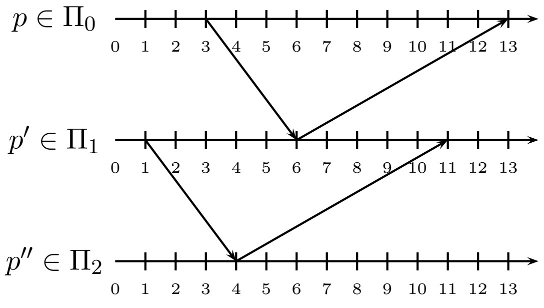
For clock synchronization to within some $\gamma < \varepsilon_{(1)}$ (i.e., $c < 1$), Lemma 11 implies that there exists a rt-run whose message graph has a diameter < 2, i.e., whose message graph is fully connected, and, therefore, has $\frac{n(n-1)}{2}$ edges. This leads to the following theorem:

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

Section 8 presented algorithms achieving optimal precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ with $n(n-1) = O(n^2)$ messages. Theorem 7 reveals that this bound is asymptotically tight.



(a) Original



(b) Shifted

Figure 13: Shifting by $d \cdot \varepsilon_{(1)}/2$ with $\varepsilon_{(1)} = 4$

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 $\geq \sqrt[D+1]{n}$.*[16]

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. □

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:

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

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 10.1 presents an algorithm synchronizing clocks to within $\varepsilon_{(1)}$ where every processor receives exactly one message. On the other hand, the algorithm in Section 10.2 also achieves this precision but bounds the number of sent messages per processor by 3.

9.3 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 4.3, the algorithm's time complexity must be at least $\min_{k=0}^{\lceil \sqrt[2c+1]{n} \rceil}(k \cdot \mu^+_{(0)} + \delta^+_{(n-k)})$.[17] Clearly, $k\mu^+_{(0)}$ is linear in k, so the interesting term is $\delta^+_{(n-k)}$, leading to the following corollary:

Corollary 2. *If multicasting a message in constant time is impossible, clock synchronization to 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 send and 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 8.3 is not only tight regarding precision but also has asymptotically optimal time complexity ($\mathrm{O}(n)$).

[17]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 14: Star Topology-based Clock Synchronization Algorithm

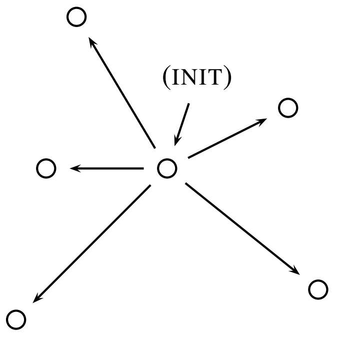


Figure 15: Principle of the Star Topology-based Clock Synchronization Algorithm

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

Sometimes, $\Theta(n^2)$ 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.

10.1 Algorithm With Least Number of Messages

Figure 14 is actually a simpler version of the algorithm presented in Section 8: 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 15). 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 to 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 $\delta^-_{(n-1)} + \delta^+_{(n-1)}$ in Line 9 with $\delta^-_{(1)} + \delta^+_{(1)}$.

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.

10.2 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 to 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 8 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 $\varepsilon_{(1)}$.

11 Conclusions and Future Work

We presented a real-time computing model, which just adds non-zero computing step times to the classic computing model. Since it explicitly incorporates queuing 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 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 to within a constant factor of the message delay uncertainty can be achieved in constant time only if a constant-time broadcast primitive is 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, examining 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

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AHR93] Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization un-

Constraint	**Consequences**	**Algorithm**
-	Best precision: $(1 - \frac{1}{n})\varepsilon_{(1)}$ *Proof: Corollary 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 9*	Section 8
Achieve best msg. complexity: $\mathrm{O}(n)$	Best precision: $\varepsilon_{(1)}$ *Proof: Theorem 7*	Section 10.1

Table 1: Tight Bounds and Algorithms

der different delay assumptions. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 109–120, New York, NY, USA, 1993. ACM Press.

[AKH03] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing*. John Wiley & Sons, 2nd edition, 2004.

[AY96] James H. Anderson and J.-H. Yang. Time/-contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[BP03] Dhruba Basu and Sasikumar Punnekkat. Clock synchronization algorithms and scheduling issues. In *Proceedings International workshop on Distributed Systems (IWDC'03), LNCS 2918*. Springer-Verlag, December 2003.

[BW01] Saâd Biaz and Jennifer L. Welch. Closed form bounds for clock synchronization under simple uncertainty assumptions. *Information Processing Letters*, 80(3):151–157, 2001.

[HLL02] Jean-François Hermant and Gérard Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.

[HW05] Jean-François Hermant and Josef 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)*, volume 3974 of *LNCS*, pages 334–350, Pisa, Italy, December 2005. Springer Verlag.

[KLSV03] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 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.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LL84a] Jennifer Lundelius and Nancy Lynch. A new fault-tolerant algorithm for clock synchronization. In *Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 75–88, August 1984.

[LL84b] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–240, 1984.

[LLS03] Gérard Le Lann and Ulrich Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003.

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[LV96] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[MMT91] Michael Merritt, Francesmary Modugno, and Marc R. Tuttle. Time-constrained automata (extended abstract). In *Proceedings of the 2nd International Conference on Concurrency Theory (CONCUR'91)*, pages 408–423, London, UK, 1991. Springer-Verlag.

[MS06a] Heinrich Moser and Ulrich Schmid. Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, LNCS 4305, pages 95–109, Bordeaux & Saint-Emilion, France, Dec 2006. Springer Verlag.

[MS06b] Heinrich Moser and Ulrich Schmid. Reconciling distributed computing models and real-time systems. In *Proceedings Work in Progress Session of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 73–76, Rio de Janeiro, Brazil, Dec 2006.

[NT93] Gil Neiger and Sam Toueg. Simulating synchronized clocks and common knowledge in distributed systems. *J. ACM*, 40(2):334–367, 1993.

[PSR94] Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization (extended abstract). In *STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 810–819, New York, NY, USA, 1994. ACM Press.

[SAA$^+$04] Lui Sha, Tarek Abdelzaher, Karl-Erik Arzen, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysious K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems Journal*, 28(2/3):101–155, 2004.

[SGSAL98] Roberto Segala, Rainer Gawlick, Jorgen F. Sogaard-Andersen, and Nancy A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, 1998.

[SLWL90] Barbara Simons, Jennifer Lundelius-Welch, and Nancy Lynch. An overview of clock synchronization. In Barbara Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, pages 84–96. Springer Verlag, 1990. (Lecture Notes on Computer Science 448).

[ST87] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[WLLS05] Josef Widder, Gérard Le Lann, and Ulrich Schmid. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 20–37, Budapest, Hungary, April 2005. Springer Verlag.