Optimal Clock Synchronization Revisited: Upper and Lower Bounds in Real-Time Systems*

Heinrich Moser** and Ulrich Schmid

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

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. As a by-product, it also allows us to investigate whether/which properties of real systems are inaccurately or even wrongly captured when resorting to zero step-time models. We revisit the well-studied problem of deterministic internal clock synchronization for this purpose, and show that, contrary to the classic model, no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. We prove that optimal precision is only achievable with algorithms that take $\Omega(n)$ time in our model, and establish several additional lower bounds and algorithms.

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 [1].

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,

** Corresponding author.

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

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 [2], 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.

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 [3, 4] 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. 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 [4], we have found an instance of a problem where the standard distributed computing analysis gives too optimistic results. Actually, we show that optimal precision is only achievable with algorithms that take $\Omega(n)$ time, even if they are provided with a constant-time broadcast primitive.

Lacking space does not allow us to present all our results and derivations here, which can be found in [5].

2 Classic Computing Model

In clock synchronization research [6–9, 4], system models are considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted "Partially Synchronous Reliable/Unreliable Models" in [3], 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 [4]. Here it will be referred to as the *classic computing model*.

We consider a network of n failure-free *processors*, which communicate by passing unique messages. Each processor p is equipped with a CPU, some local memory, a hardware clock $HC_p(t)$ running at the same rate as real time, and reliable, non-FIFO links to all other processors.

The CPU is running an algorithm, specified as a mapping from processor indices to a set of initial states and a transition function.

The *transition function* takes the processor index p, one incoming message, receiver processor current local state and hardware clock reading as input, and yields a list of *states* and *messages to be sent*, e.g. $[oldstate, int.st._1, int.st._2, \text{msg. } m \text{ to } q, \text{msg. } m' \text{ to } q', int.st._3, newstate]$, as output. A "message to be sent" is specified as a pair consisting of the message itself and the destination processor the message will be sent to. The intermediate states are usually neglected in the classic computing model, as the state transition from *oldstate* to *newstate* is instantaneous. We explicitly model these states to retain compatibility with our real-time computing model, where they will become important.

Every message reception immediately causes the receiver processor to change its state and send out all messages according to the transition function. Such a *computing step* will be called an *action* in the following. The complete action (message arrival, processing and sending messages) is performed in zero time.

Actions can actually be triggered by ordinary messages and timer messages: Ordinary messages are transmitted over the links. The *message delay*[2] $\underline{\delta}$ 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.

Timer messages are used for modeling time(r)-driven execution in our message-driven setting: 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.

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 to be sent). A valid execution must satisfy obvious properties such as conformance with the transition function, timer messages arriving on time, and reliable message transmission. To trigger the first action of a processor in an execution, we allow one special *init message* to arrive at each processor from outside the system.

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

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 $[\underline{\delta}^-, \underline{\delta}^+]$.

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

3 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 execution of their predecessors has been completed. Common practice so far is to take this queuing delay into account by increasing the upper bound $\underline{\delta}^+$ on the message delay. This approach, however, has two disadvantages: First, a-priori information about the algorithm's message pattern is needed to determine a parameter of the system model, which creates cyclic dependencies. Second, in lower bound proofs, the adversary can choose an arbitrary message delay within $[\underline{\delta}^-, \underline{\delta}^+]$ – even if this choice is not in accordance, i.e., not possible, with the current message arrival pattern. This could lead to overly pessimistic lower bounds.

3.1 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 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 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*. Since computing steps cannot be interrupted, a *queue* is needed to store messages arriving while the processor is busy.
- 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. To ensure liveness, we assume that the scheduling policy is *non-idling*.
- 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). Analogous to the classic computing

model, message delays of ordinary messages must be within a system-wide lower bound δ^- and an upper bound δ^+. Like the processing delay, the message delay and hence the bounds $[\delta^-, \delta^+]$ may depend on the number of messages sent in the sending job.
- We assume that the hardware clock can only be read at the beginning of a job. 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 $oldstate \rightarrow \cdots \rightarrow newstate$ in a single computing step need not happen at the same time: Typically, they occur at different times during the job, allowing an intermediate state to be valid on a processor for some non-zero duration.

Figure 1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. Part (a) shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dotted arrow. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not 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 (see Sect. 2), 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 Fig. 1 shows the detailed relation between message arrival (enqueuing) and actual message processing.

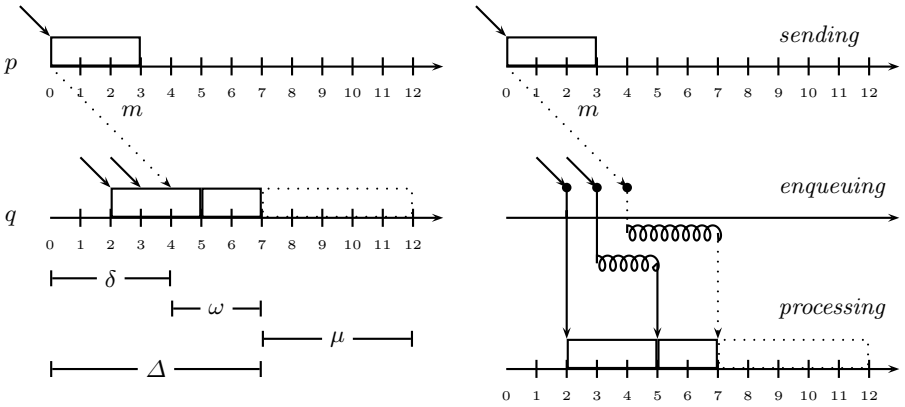
3.2 Real-time Runs

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

A *receive event* R for a message arriving at 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 Fig. 1(b).

A *job* J starting at real-time t on p is a 6-tuple, consisting of the processor index $proc(J) = p$, the message being processed $msg(J)$, the start time $begin(J) = t$, the job processing time $d(J)$, the hardware clock reading $HC(J) = HC_p(t)$, and the state transition sequence $trans(J) = [oldstate, \ldots, newstate]$.

Figure 1 provides an example of a rt-run, containing three receive events and three jobs on the second processor. For example, the dotted job on the second processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \ldots, newstate])$, with m being the message received during the receive event $(q, m, 4)$. 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



(a) Timing parameters for some msg. m (b) Relation of msg. arrival and msg. processing, with enqueuing shown explicitly

Fig. 1. Real-time computing model

that the state transitions and the message sends occur in the listed order at arbitrary times during the job is sufficient for proving that a rt-run satisfies a given set of properties, as well as for performing time complexity analysis.

In addition to the properties required for valid executions, a valid rt-run must also satisfy additional consistency constraints: Jobs on the same processor must not overlap, and the execution must conform to some arbitrary non-idling scheduling policy.

3.3 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 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 15 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.[3]

Example: During some job, messages to exactly three processors are sent. The duration of this job lies within $[\mu^-_{(3)}, \mu^+_{(3)}]$. Each of these messages has a

[3] 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.

message delay between $\delta^-_{(3)}$ and $\delta^+_{(3)}$. The delays of the three messages need not be the same.

Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. 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 1. *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,*

- *The message delay (measured from $begin(J)$ to the corresponding receive event) of every message in $trans(J)$ must be within $[\delta^-_{(\ell)}, \delta^+_{(\ell)}]$.*
- *The job duration $d(J)$ must be within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$.*

3.4 Correctness and Impossibility

In the model presented so far, the scheduling policies are *adversary-controlled*, meaning that, in the game between player and adversary, the player chooses the algorithm and afterwards the adversary can decide on a scheduling policy that 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* (as the adversary has the scheduling policy on its side).

However, sometimes algorithms are designed for particular, a-priori-known scheduling policies, or the algorithm designer has the freedom to choose the scheduling policy which is most convenient for the algorithm. Thus, it is useful to define as *weak correctness* the correctness of an ⟨algorithm, scheduling policy⟩ pair and, analogously, as *strong impossibility* the absence of any such pair. In this paper, we only consider strong correctness and strong impossibility.

3.5 Transformations

The classic computing model and the real-time computing model are fairly equivalent from the perspective of solvability of problems. In [5], we present two transformations: One direction, simulating a real-time system 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 $\delta^-_{(1)} \leq \underline{\delta}^-$, $\delta^+_{(1)} \geq \underline{\delta}^+$ and $\mu^- \leq \underline{\mu} \leq \mu^+$ on top of it, thereby resulting in a correct classic computing model algorithm. From this result (Theorem 3 in [5]), the following lemma can be derived directly:

Lemma 2. *Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. If there exists an algorithm for solving some problem $\mathcal{P}$ in some real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^-_{(1)} \leq \underline{\delta}^-$ and $\delta^+_{(1)} \geq \underline{\delta}^+$, then $\mathcal{P}$ can be solved in $\underline{s}$.*

The other direction, 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, the class of classic computing model algorithms $\underline{\mathcal{A}}$ that can be transformed into a real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ must be somewhat restricted. 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.

In our setting, this circularity can be broken as follows: Given some classic computing model algorithm $\underline{\mathcal{A}}$ with assumed message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, considered as unvalued parameters, a real-time scheduling analysis of the combined algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ ($\underline{\mathcal{A}}$ and simulation algorithm) 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

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

We do not want to embark on the intricacies of advanced real-time scheduling analysis techniques here, see [2] for an overview. For the purpose of simple problems like terminating clock synchronization (see below), 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.

Anyway, if (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 $[\underline{\delta}^-, \underline{\delta}^+]$. Our transformation in fact guarantees that any timer-free algorithm designed for some classic system $\underline{s}$ solving some suitable problem $\mathcal{P}$ can also solve that problem in the corresponding real-time system s if such a feasible assignment for $[\underline{\delta}^-, \underline{\delta}^+]$ exists.

Clock Synchronization: In the classic computing model, a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ has been proved in [4] as the best achievable clock synchronization precision. We can use Lemma 2 to show that clock synchronization closer than $(1 - \frac{1}{n})\varepsilon_{(1)}$ is impossible in the real-time computing model.

Theorem 3. *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 algorithm that can provide clock synchronization for some real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ to within $\gamma < (1 - \frac{1}{n})\varepsilon_{(1)}$. Applying Lemma 2 would imply that clock synchronization to within $\gamma < (1 - \frac{1}{n})(\underline{\delta}^+ - \underline{\delta}^-)$ can be provided for some classic system $(n, [\underline{\delta}^- = \delta^-_{(1)}, \underline{\delta}^+ = \delta^+_{(1)}])$. This, however, contradicts a well-known lower bound result [4].

4 Algorithms Achieving Optimal Precision

Theorem 3 raises the question whether the lower bound of $(1 - \frac{1}{n})\varepsilon_{(1)}$ is tight in the real-time computing model. In this section, we will answer this in the affirmative: We show how the algorithm presented in [4] can be modified to avoid queuing effects and thus provide optimal precision in a real-time system s. 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. Two lemmata from [4] can be generalized to our setting:

Lemma 4. *If q receives a timestamped message from p with end-to-end delay uncertainty ε_Δ, q can estimate p's hardware clock value within an error of at most $\frac{\varepsilon_\Delta}{2}$. (Proof: See Lemma 5 of [4] or see [5].)*

Lemma 5. *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: See Theorem 7 of [4].)*

4.1 Optimality for Broadcast Systems

Note: *To ease presentation, we 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 $\varepsilon_{(n-1)}$ in this subsection.*

The Lundelius-Lynch clock synchronization algorithm [4] 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 initialization message arrives. This algorithm can easily be modified to avoid queuing effects by "serializing" the information exchange, rather than sending all messages (possibly) simultaneously.

The modified algorithm, depicted in Fig. 2, 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

```
1   var estimates ← {}
2
3   function process_message(msg, time)
4     /* start alg. by sending (SEND) to proc. 0 */
5     if msg = (SEND)
6       send (TIME, time) to all other processors
7     elseif msg = (TIME, remote_time)
8       estimates.add(remote_time − time + (δ̇⁻ + δ̇⁺)/2)
9       if estimates.count = ID
10        send timer (SEND) for time + max(ε̇ − δ̇⁻ + μ̇⁺, μ̇⁺)
11      if estimates.count = n−1
12        set adjustment value to (∑ estimates)/n
```

Fig. 2. Clock-synchronization algorithm to within $\varepsilon_{(n-1)}$, code for processor ID

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 $(1 - \frac{1}{n})\dot{\varepsilon}$.

Lemma 6. *No queuing occurs when running the algorithm of Fig. 2.*

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

Assume by contradiction that some message from processor $j > 0$ arrives on processor i at time t, although the message from processor $j - 1$ has arrived (or will arrive) at time $t' > t - \dot{\mu^+}$. Choose t to be 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 10, this message must not arrive earlier than $\dot{\mu^+}$ time units after $j - 1$'s message, which has triggered the job sending the timer message.

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

ago. Otherwise, if $j-1 \neq i$, process $j-1$'s broadcast arrived at i no later than $t - \dot{\varepsilon} - \dot{\mu^+} - \dot{\delta^-} + \dot{\delta^+} = t - \dot{\mu^+}$, also contradicting our assumption.

Theorem 7 (Optimal broadcast algorithm). *The algorithm of Fig. 2 achieves a precision of* $(1 - \frac{1}{n})\varepsilon_{(n-1)}$*, which matches the lower bound in Theorem 3 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 set *estimates* contains the estimated differences between the local hardware clock and the hardware clocks of the other processes. As no queuing occurs by Lemma 6, the end-to-end delays are just the message delays. Line 8 in the algorithm of Fig. 2 ensures that the estimates are calculated as specified in the proof of Lemma 4. Thus, the estimates have a maximum error of $\frac{\dot{\varepsilon}}{2}$. According to Lemma 5, this allows the algorithm to calculate an adjustment value (in line 12) ensuring 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 message delay $\max(\dot{\varepsilon} - \dot{\delta^-} + \dot{\mu^+}, \dot{\mu^+})$ plus one message delay $\dot{\delta^+}$). 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.

4.2 Optimality for Unicast Systems

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 shown in Fig. 3, however: 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.

Theorem 8 (Optimal unicast algorithm). *The algorithm of Fig. 3 achieves a precision of* $(1 - \frac{1}{n})\varepsilon_{(1)}$*, which matches the lower bound given in Theorem 3. It sends exactly* $n(n-1) = \mathrm{O}(n^2)$ *messages system-wide and has* $\mathrm{O}(n)$ *time complexity.*

Proof. Omitted due to size limitations; see [5].

5 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

```
1   var estimates ← {}
2
3   function process_message(msg, time)
4     /* start alg. by sending (SEND, 1) to proc. 0 */
5     if msg = (SEND, target)
6       send (TIME, time) to target
7       if target + 1 mod n ≠ ID
8         send timer (SEND, target + 1 mod n) for time + μ⁺₍₁₎
9     elseif msg = (TIME, remote_time)
10      estimates.add(remote_time − time + (δ⁻₍₁₎ + δ⁺₍₁₎)/2)
11      if estimates.count = ID
12        send timer (SEND, ID + 1) for
13              time + max(ε₍₁₎ − δ⁻₍₁₎ + 2μ⁺₍₁₎, μ⁺₍₁₎)
14      if estimates.count = n−1
15        set adjustment value to (∑ estimates)/n
```

Fig. 3. Clock-synchronization algorithm to within $\varepsilon_{(1)}$, code for processor ID

every link. This bound is asymptotically tight, since it is matched by Theorem 8. A strong indication for this result follows already from the work of Biaz and Welch [7]. They have shown that no algorithm can achieve a precision better than $\frac{1}{2}diam(G)$ for any communication network G, with $diam(G)$ being the diameter of the graph where 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 [4] can no longer be achieved.

Unfortunately, the proof from [7] cannot be used directly in our context: 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 the proof of Theorem 3 of [7] can be used, however. *Shifting* is a common technique in the classic computing model for proving clock synchronization lower bounds. Analogously, shifting a rt-run ru of n processors by $(x_0, \ldots, x_{n-1})$ results in another rt-run ru', where

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

Environment: Let $c \in \mathbb{R}^+$ be a constant and $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system with $n > 2$. Let $\mathcal{A}$ be 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 delay of all messages is 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 $\mu^+_{(\ell)}$.

Lemma 9. *The message graph of ru has a diameter of* $2c$ *or less.*

Proof. 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. 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. 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 inequalities $|\Delta| \leq c \cdot \varepsilon_{(1)}$, $|\Delta + D \cdot \varepsilon_{(1)}/2| \leq c \cdot \varepsilon_{(1)}$ and $|\Delta - D \cdot \varepsilon_{(1)}/2| \leq c \cdot \varepsilon_{(1)}$, which imply $c \geq D/2$ and hence contradict $D > 2c$.

5.1 Message Complexity

For clock synchronization to within some $\gamma < \varepsilon_{(1)}$, Lemma 9 implies that there is a rt-run with message graph 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 10. *Clock synchronization to within* $\gamma < \varepsilon_{(1)}$ *has a worst-case message complexity of at least* $\frac{n(n-1)}{2} = \Omega(n^2)$.

Section 4 presented an algorithm achieving optimal precision of $(1 - \frac{1}{n})\varepsilon_{(1)}$ with $n(n-1) = \mathrm{O}(n^2)$ messages, cp. Theorem 8. Theorem 10 reveals that this bound is asymptotically tight. Obviously, a smaller lower bound can be given for suboptimal clock synchronization. We will use the following simple graph-theoretical lemma:

Lemma 11. *In an undirected graph with* $n > 2$ *nodes and diameter* D *or less, there is at least one node with degree* $\geq \sqrt[D+1]{n}$.[4]

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

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 are done.

Combining Lemmata 9 and 11 shows that there is at least one processor in ru which exchanges (= sends or receives) at least $\lceil \sqrt[2c+1]{n} \rceil$ messages. This implies the following results:

Theorem 12. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$ in some real-time system s, there is an s-admissible rt-run in which at least one processor exchanges $\lceil \sqrt[2c+1]{n} \rceil$ messages.*

Corollary 13. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$, there is no constant upper bound on the number of messages exchanged per processor.*

Note, however, that it is possible to constantly bound either the number of received *or* the number of sent messages per processor (but not both): Section 6 presents an algorithm synchronizing clocks to within $\varepsilon_{(1)}$ where every processor receives exactly one message, and an algorithm provided in [5] achieves this precision with sending just 3 messages per processor.

5.2 Time Complexity

In the case of optimal precision, the following Theorem 14 states a time complexity lower bound of $\Omega(n)$. This bound is asymptotically tight, since it is matched by the optimal algorithm underlying Theorem 8.

Theorem 14. *Clock synchronization to within $\gamma < \varepsilon_{(1)}$ has a worst-case time complexity of at least $\Omega(n)$.*

Proof. Theorem 10 revealed that n processors need to exchange at least $\frac{n(n-1)}{2}$ messages. Therefore, no algorithm can achieve a run time better than $\max\big(\frac{n-1}{2}\mu^+_{(0)}, \delta^+_{(\frac{n-1}{2})}\big)$, assuming optimal concurrency. This implies a time complexity lower bound of $\Omega(n)$ as asserted.

On the other hand, Theorem 12 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 Sect. 3.3, the algorithm's time complexity must be at least $\min_{k=0}^{\lceil \sqrt[2c+1]{n} \rceil}(k \cdot \mu^+_{(0)} + \delta^+_{(n-k)})$. Clearly, $k\mu^+_{(0)}$ is linear in k, so the interesting term is $\delta^+_{(n-k)}$. Consequently:

Corollary 15. *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.*

```
1  function s−process_message(msg, time)
2     /* start alg. by sending (INIT) to some proc. */
3     if msg = (INIT)
4        send time to all other processors
5        set adjustment value to 0
6     else
7        set adjustment value to (msg − time + (δ⁻_(n−1) + δ⁺_(n−1))/2)
```

Fig. 4. Star Topology-based Clock Synchronization Algorithm

6 Achievable Precision for less than $\Omega(n^2)$ Messages

Sometimes, $\Omega(n^2)$ messages may 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 to be sent system-wide; 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: Figure 4 is actually a simpler version of the algorithm presented in Sect. 4. 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). Lemma 4 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 of n (i.e., if a constant-time broadcasting primitive is available), $\varepsilon_{(n-1)} = \varepsilon_{(1)}$ 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 unicast case as well:

- Do not send all messages during the same job but during subsequent jobs on the "master" processor.
- Replace $\delta^-_{(n-1)}$ and $\delta^+_{(n-1)}$ in Line 7 with $\delta^-_{(1)}$ and $\delta^+_{(1)}$.

The algorithm still exchanges only $n-1$ messages and has linear time complexity w.r.t. n. As Theorem 10 has shown, $\varepsilon_{(1)}$ is the best precision that can be achieved with less than $\Omega(n^2)$ messages. As Corollary 15 has shown, this precision cannot be achieved in constant time in the unicast case.

7 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 model, this is 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.

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

References

1. Lynch, N.: Distributed Algorithms. Morgan Kaufman (1996)
2. Sha, L., Abdelzaher, T., Arzen, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Systems Journal **28**(2/3) (2004) 101–155
3. Simons, B., Lundelius-Welch, J., Lynch, N.: An overview of clock synchronization. In Simons, B., Spector, A., eds.: Fault-Tolerant Distributed Computing, Springer Verlag (1990) 84–96 (Lecture Notes on Computer Science 448).
4. Lundelius, J., Lynch, N.: An upper and lower bound for clock synchronization. Information and Control **62** (1984) 190–240
5. Moser, H., Schmid, U.: Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. Research Report 71/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2006) `http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1973&viewmode=paper&year=2006`.
6. Lundelius, J., Lynch, N.: A new fault-tolerant algorithm for clock synchronization. In: Proc. 3rd ACM Symposium on Principles of Distributed Computing (PODC). (1984) 75–88
7. Biaz, S., Welch, J.L.: Closed form bounds for clock synchronization under simple uncertainty assumptions. Information Processing Letters **80**(3) (2001) 151–157
8. Patt-Shamir, B., Rajsbaum, S.: A theory of clock synchronization (extended abstract). In: STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1994) 810–819
9. Attiya, H., Herzberg, A., Rajsbaum, S.: Optimal clock synchronization under different delay assumptions. In: PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (1993) 109–120