

Optimal Deterministic Remote Clock Estimation 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. In an OPODIS'06 paper, we laid down the foundations of a real-time distributed computing model (RT-Model) with non-zero duration computing steps, which reconciles correctness proofs and real-time schedulability analysis of distributed algorithms. By applying the RT-Model to the well-known drift-free internal clock synchronization problem, we proved that classic zero step-time analysis sometimes provides too optimistic results. The present paper provides a first step towards worst-case optimal deterministic clock synchronization with drifting clocks in real-time systems, which is an open problem even in classic distributed computing. We define and prove correct an optimal remote clock estimation algorithm, which is a pivotal function in both external and internal clock synchronization, and determine a matching lower bound for the achievable maximum clock reading error in the RT-Model. Moreover, we show how to combine our optimal clock estimation algorithm with existing clock synchronization algorithms.

Keywords. Distributed algorithms, real-time systems, computing models, optimal clock synchronization, remote clock estimation.

1 Introduction

Clock synchronization [1–3] is an important and well-studied problem in distributed systems, where processors are equipped with imperfect hardware clocks and connected through a message-passing network. The problem is parameterized by the achievable worst-case synchronization *precision* and comes in two flavors: The goal of *external clock synchronization* is to synchronize all clocks to the clock of a dedicated source processor, whereas *internal clock synchronization* aims at mutually synchronizing all the clocks to each other.

Our research aims at optimal deterministic clock synchronization in distributed *real-time* systems. It differs from traditional distributed computing research by not abstracting away queueing phenomena and scheduling, but rather quantifying their impact on “classic” results. As a first step towards this

* Our research is supported by the Austrian Science Foundation (FWF) under grants P17757 and P20529.

goal, we revisited the well-known drift- and failure-free internal clock synchronization problem [4] in [5, 6]: Utilizing a novel distributed computing model (RT-Model) based on non-zero-duration computing steps, which both facilitates real-time schedulability analysis and retains compatibility with classic distributed computing analysis techniques and results, we proved that the best precision achievable in the real-time computing model is the same as in the classic computing model, namely, $(1 - \frac{1}{n})\varepsilon$, where n is the number of processors and ε is the message delay uncertainty. It turned out, however, that any optimal clock synchronization algorithm has a time complexity of $\Omega(n)$ in the real-time computing model. Since a $O(1)$ algorithm achieving optimal precision is known for the classic computing model [4], it became apparent that distributed computing results are sometimes too optimistic in the real-time systems context.

The present paper provides a first step towards deterministic optimal-precision¹ clock synchronization in real-time systems with clocks that have non-zero drift, i.e., that do not progress exactly as real-time does. More specifically, we restrict our attention to a (deceptively simple) subproblem of clock synchronization, namely, *remote clock estimation*, in the real-time computing model. Informally, a remote clock estimation algorithm allows processor p to estimate the local clock at processor q at some real-time t , with a known maximum clock reading error. In fact, it is well-known that any existing clock synchronization algorithm can be reviewed in terms of a generic structure [11], which consists of (1) detecting the need for resynchronization, (2) estimating the remote clock values, (3) computing a (fault-tolerant) clock adjustment value, and (4) adjusting the local clock accordingly. Our results on (continuous) remote clock estimation are hence pivotal building blocks for finding and analyzing optimal algorithms for both external and internal clock synchronization in real-time systems.

Related work: Optimal-precision clock synchronization with drifting clocks is an open problem even in classic distributed computing: No optimal algorithms, and only trivial or quite specialized lower bounds are known by now. For example, [12] provides a lower bound for the restricted class of function-based internal clock synchronization algorithms, and [13] provides a lower bound for the precision achievable in a “single-shot” version of clock synchronization (“tick synchronization”) in the semi-synchronous model, with inter-step-times $\in [c, 1]$ that can be thought of as being caused by drifting clocks. Optimal results are only available with respect to given message patterns (“passive” clock synchronization) [14, 15]; unfortunately, optimal message patterns and hence optimal “active” clock synchronization algorithms cannot be inferred from this research.

Remote clock estimation itself is also handled/analyzed sub-optimally or abstracted away entirely in the wealth of existing research on clock synchronization:

¹ We restrict our attention to worst-case optimality throughout this paper, i.e., we only care about minimizing the maximum precision in the worst execution (rather than in every execution). This is appropriate in the hard real-time systems context, where only the worst case performance matters. For the same reason, we consider deterministic algorithms only; probabilistic clock synchronization [7, 8], statistically optimal estimations [9, 10] and similar topics are hence out of scope.

Most papers on clock synchronization employ trivial clock estimation algorithms only, based on a one-way or round-trip time-transfer via messages [9], and provide a fairly coarse analysis that (at best) incorporates clock drift [7] and clock granularity [16]. Alternatively, as in [17, 12], remote clock estimation is considered an implementation issue and just incorporated via the a-priori given maximum clock reading error. Hence, to the best of our knowledge, optimal deterministic clock estimation has not been addressed in the existing literature.

Contributions: The present paper aims at optimal deterministic clock estimation and related lower bounds in the real-time systems context, where the issue of queueing and scheduling is added to the picture: Utilizing an extension of the real-time computing model (Sect. 2) introduced in [18], which allows to model executions in real-time systems with drifting clocks, we provide an optimal solution for the problem of how to continuously estimate a source processor’s clock (Sect. 3). The algorithm is complemented by a matching lower bound on the achievable maximum clock reading error (Sect. 4). Our results precisely quantify the effect of system parameters such as clock drift, message delay uncertainty and step duration on optimal clock estimation. Finally, we show how to incorporate our optimal clock estimation algorithm in existing clock synchronization algorithms (Sect. 5). Some conclusions (Sect. 6) eventually complete our paper.

2 The Real-Time Computing Model

Our system model is based on an extension of the simple real-time distributed computing model introduced in [5], which reconciled the distributed computing and the real-time systems perspective by just replacing instantaneous computing steps with computing steps of non-zero duration. The extended version of the real-time model, introduced in [18], is based on a “microscopic view” of executions termed *state-transition traces*, which allows to define a total order on the *global states* taken on during an execution; this feature is mandatory for properly modeling drifting clocks. Due to lack of space, we will only restate the most important definitions and properties of the extended real-time computing model here; consult [18] for all the details (including a relation to existing computing models).

2.1 Jobs and Messages

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 $HC_p(t)$, and reliable 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, the receiver processor’s current local state and hardware clock reading as input, and yields a sequence of *states and messages to be sent* (termed *state transition sequence* in the sequel), e.g. $[oldstate, int.st.1, int.st.2, msg. m \text{ to } q, msg. m' \text{ to } q', int.st.3, newstate]$, as

output. Note that a message to be sent is specified as a pair consisting of the message itself and the destination processor. Since more than one atomic state transition might be required in response to a message, the example above contains three *intermediate states*.

Every message reception causes the receiver processor to change its state and send out all messages according to the state transition sequence provided by the transition function. Such a *computing step* will be called a *job* in the following, and is executed non-preemptively within a system-wide lower bound $\mu^- \geq 0$ and upper bound $\mu^+ < \infty$. We assume that the hardware clock can only be read at the beginning of a job. Note that these assumptions are not overly restrictive, since a job models a (possibly complex) computing step rather than a task in the real-time computing model.

Jobs can be triggered by ordinary messages, timer messages and input messages: *Ordinary messages* are transmitted over the links. The *message delay* δ is the difference between the real time of the *start of the job* sending the message and the real time of the arrival of the message at the receiver. There is a lower bound $\delta^- \geq 0$ and an upper bound $\delta^+ < \infty$ on the message delay of every ordinary message. Since the message delay *uncertainty* is a frequently used value, we will abbreviate it with $\varepsilon := \delta^+ - \delta^-$.

To capture timing differences between sending a single message versus (single-hop) multicasting or broadcasting, the interval boundaries δ^- , δ^+ , μ^- and μ^+ can be either constants (e.g. in the case of hardware broadcast) or non-decreasing functions $\{0, \dots, n-1\} \rightarrow \mathbb{R}^+$, representing a mapping from the number of destination processors to which ordinary messages are sent during a job to the actual message or processing delay bound. The following example shall clarify this: If a job sends ℓ messages to ℓ recipients (with the same or with different content),² that job's duration lies between $\mu_{(\ell)}^-$ and $\mu_{(\ell)}^+$ time units. Each of the ℓ messages takes between $\delta_{(\ell)}^-$ and $\delta_{(\ell)}^+$ time units to arrive at the recipient. The delays of these messages need not be the same, however.

Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. Formally, $\forall i, j \geq 1 : f_{(i+j)} \leq f_{(i)} + f_{(j)}$ (for $f = \delta^-, \delta^+, \mu^-$ 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 sending more messages usually increases the uncertainty rather than lowering it.

Timer messages are used for modeling time(r)-driven executions in our message-driven setting: A processor setting a timer is modeled as sending a timer message (to itself), 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.

² As the message size is not bounded, we can assume that at most one message is sent to the same processor in a job. Hence, there is a one-to-one correspondence between messages and destination processors in each job.

Input messages arrive from outside the system. In this paper, input messages are used solely for booting up the system, i.e., for triggering the first job in an execution.

Messages arriving while the processor is busy with some job are *queued*. 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 a 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. We ensure liveness by assuming that the scheduling policy is *non-idling*. To make our results as strong as possible, we will allow the adversary to control the scheduling policy in the algorithm proofs, but we will assume an algorithm-controlled scheduler in the lower bound proof.

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. It 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 arrows. 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. 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 schedulability analysis, cp. Sect. 3.4.

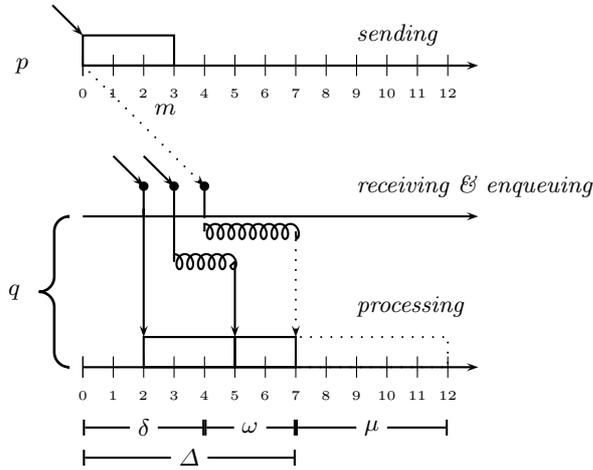


Fig. 1. Timing parameters for some message m

2.2 Hardware Clocks

The hardware clock of any processor p starts with some arbitrary initial value $HC_p(0)$ and then progresses with a bounded drift rate of ρ_p , i.e., t real-time units correspond to at least $(1 - \rho_p)t$ and at most $(1 + \rho_p)t$ clock-time units. Formally, for all $p, t > t' \geq 0$:

$$(t - t')(1 - \rho_p) \leq HC_p(t) - HC_p(t') \leq (t - t')(1 + \rho_p)$$

When talking about *time units* in this paper, we mean *real-time units*, unless otherwise noted.

2.3 Real-time Runs

A *real-time run* (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 $time(R) = t$. Note that t is the receiving/enqueuing time in Fig. 1.

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 $duration(J)$, the hardware clock reading $HC(J) = HC_p(t)$ when the job starts, and the state transition sequence $trans(J) = [oldstate, \dots, newstate]$. Let $end(J) = begin(J) + duration(J)$.

Figure 1 provides an example of an rt-run, containing three receive events and three jobs on the second processor q . For example, the dotted job on processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \dots, newstate])$, with m being the message received during the receive event $(q, m, 4)$. An rt-run is called *admissible*, if all its message delays (measured from the start of the job to the corresponding receive event) stay within $[\delta_{(\ell)}^-, \delta_{(\ell)}^+]$, the duration of all jobs sending ℓ messages is within $[\mu_{(\ell)}^-, \mu_{(\ell)}^+]$, and the ordering of receive events and jobs does not violate causality (cf. the well-known *happens-before* relation [19]).

If a timer is set during some job J for some time $T < HC_{proc(J)}(end(J))$, the timer message will arrive at time $end(J)$, when J has completed.

2.4 State Transition Traces

The *global state* of a system is composed of the local state s_p of every processor p and the set of not yet processed messages. Rt-runs do not allow a well-defined notion of global states, since they do not fix the exact time of state transitions in a job. This problem is fixed by introducing the “microscopic view” of *state-transition traces* (st-traces) [18], which assign real times to all atomic state transitions.

The following example should provide the required background for understanding the usage of this method, see Appendix A for more information.

Example 1. Let J with $trans(J) = [oldstate, msg. m \text{ to } q, int.st.1, newstate]$ be a job in a real-time run ru . If tr is an st-trace of ru , it contains the following *state transition events* (st-events) ev' , ev'' and ev''' :

- $ev' = (send : t', p, m)$
- $ev'' = (transition : t'', p, oldstate, int.st.1)$
- $ev''' = (transition : t''', p, int.st.1, newstate)$

with $begin(J) \leq t' \leq t'' \leq t''' \leq end(J)$. For every time t , there is at least one global state g in tr . Note carefully that tr may contain more than one g with $time(g) = t$. For example, if $t'' = t'''$ in the previous example, three different global states at time t'' would be present in the st-trace, with $s_p(g)$ representing p 's state as $oldstate$, $int.st.1$ or $newstate$. Nevertheless, in every st-trace, all st-events and global states are totally ordered by some relation \prec .

The time t of a *send* st-event must be such that message causality is not violated, i.e., $t \leq begin(J)$, with J being the job processing the message in ru . Recall, however, that the message delay δ is measured from the start of the job sending the message to the receive event in ru . This convention allows a complete schedulability analysis to be done on top of the rt-run, without the need to consider the actual time of state transitions.

3 Optimal Remote Clock Estimation

In this and the following section, we assume a failure-free two-processor system with drifting clocks. Note that remote clock reading is trivially unsolvable in case of just a single crash failure.

3.1 Interval-Based Notation

Often, remote clock estimations are represented by a tuple $(value, margin)$, with $value$ representing the expected value of the remote clock and $margin$ the absolute deviation from the remote clock's real value, i.e., $remote_clock \in [value - margin, value + margin]$. With non-drifting clocks, this works well [4, 5]. However, consider the two cases in Fig. 2, in which p tries to guess src 's value at time t_r by evaluating a timestamped message with delay $\in [\delta^-, \delta^+]$ and clocks with maximum drift ρ_{src} and ρ_p .

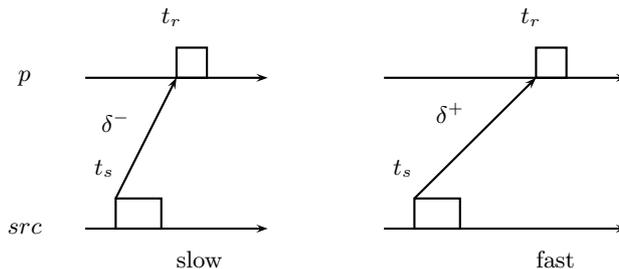


Fig. 2. p receiving a timestamped message from src .

In the first case, src is a processor with a slow clock and the message is fast; in the second case, src 's clock is fast but the message is slow. Thus, at time t_r , src 's hardware clock reads $HC_{src}(t_s) + \delta^-(1 - \rho_{src})$ in the first and $HC_{src}(t_s) + \delta^+(1 + \rho_{src})$ in the second case. In the drift-free case ($\rho_{src} = 0$), p can assume that src 's clock progressed by $\frac{\delta^- + \delta^+}{2} = \delta^- + \frac{\varepsilon}{2}$ and add this value to $HC_{src}(t_s)$, which is contained in the message. This results in a good estimation of $HC_{src}(t_r)$: It matches the expected value of $HC_{src}(t_r)$, provided message delays are uniformly distributed, with a maximum error margin of $\pm\varepsilon/2$.

In the drifting case, the arithmetic mean of $\delta^-(1 - \rho_{src})$ (= the progress of src in the first case) and $\delta^+(1 + \rho_{src})$ (in the second case) is $\delta^- + \frac{\varepsilon}{2}(1 + \rho_{src})$, which is larger than $\delta^- + \frac{\varepsilon}{2}$. Thus, p can either estimate src 's clock to be

- $HC_{src}(t_s) + \delta^- + \frac{\varepsilon}{2}(1 + \rho_{src})$, which makes for a nice symmetric error margin of $\pm(\delta^- \rho_{src} + \frac{\varepsilon}{2}(1 + \rho_{src}))$, or
- $HC_{src}(t_s) + \delta^- + \frac{\varepsilon}{2}$, which is the expected value, but which has asymmetric error margins $[-(\frac{\varepsilon}{2} + \delta^- \rho_{src}), +(\frac{\varepsilon}{2} + \delta^+ \rho_{src})]$.

To avoid this problem, we assume that p outputs two values est^- and est^+ , such that src 's real value is guaranteed to be $\in [est^-, est^+]$. Since we want to prove invariants on $[est^-, est^+]$, although there might not be a computation event at every time t , we define $est_p^-(g)$ and $est_p^+(g)$ at some global state g on processor p as functions of the current hardware clock reading, $HC_p(time(g))$, and the current local state $s_p(g)$ of p . Hence, the remote clock estimation problem is formally defined as follows:

Definition 1 (Continuous clock estimation within Γ). *Let src (source) and p be processors. Eventually, p must continuously estimate the hardware clock value of src with a maximum clock reading error Γ . Formally, for all st-traces tr :*

$$\exists ev_{stable} \in tr : \forall g \succ ev_{stable} : \\ HC_{src}(time(g)) \in [est_p^-(g), est_p^+(g)] \wedge est_p^+(g) - est_p^-(g) \leq \Gamma$$

3.2 System Model

The clock estimation algorithm in Sect. 3.3 will continuously send messages from src to p as fast as possible. The following parameters specify the underlying system:

- $[\delta^-, \delta^+]$: Bounds on the message delay.
- $[\mu_{(0)}^-, \mu_{(0)}^+]$: Bounds on the length of a job processing an incoming message, without sending any (non-timer) messages. In the algorithm of Sect. 3.3, all jobs on p fall into this category.
- $[\mu_{(1)}^-, \mu_{(1)}^+]$: Bounds on the length of a job processing an incoming message and sending one message to the other processor. In our algorithm, all jobs on src fall into this category; any such job is triggered by a timer message (or an input message, in case of the first job).

- ρ_p and ρ_{src} : Bounds on the drift of p and src , respectively. We assume $0 \leq \rho < 1$, for both $\rho = \rho_p$ and $\rho = \rho_{src}$.

To circumvent pathological cases, we need to assume that

$$\mu_{(1)}^- \geq \mu_{(0)}^+ . \quad (1)$$

Otherwise, the adversary could create an rt-run in which the “receiving” computing steps at p take longer than the “sending” computing steps at src , causing p ’s message queue to grow without bound. Note that (1) can also be interpreted as a bandwidth requirement: The maximum data rate of src must not exceed the available processing bandwidth at p (including communication).

3.3 The Algorithm

Processor src	
1	upon booting or upon receiving “send now”:
2	var my_hc = current_hc() /* can only be read at the beginning of the job */
3	send my_hc to p
4	set “send now” timer for my_hc /* will arrive at $end(current_job)$ */
Processor p	
1	var rcv_hc $\leftarrow -\infty$ /* local time of reception */
2	var send_hc $\leftarrow -\infty$ /* remote time of sending */
3	
4	function age = current_hc() – rcv_hc
5	public function $est^- = send_hc + (1 - \rho_{src})(\delta^- + age/(1 + \rho_p))$
6	public function $est^+ = send_hc + (1 + \rho_{src})(\delta^+ + \min\{\mu_{(0)}^+ + \mu_{(1)}^+, age/(1 - \rho_p)\})$
7	
8	upon receiving a HC value HC_{src} from src :
9	var my_hc = current_hc()
10	if $HC_{src} > send_hc$:
11	rcv_hc $\leftarrow my_hc$; send_hc $\leftarrow HC_{src}$ /* one atomic step */

Fig. 3. Remote clock estimation algorithm

Consider the algorithm in Fig. 3, which lets src send timestamped messages to p as fast as possible. Processor p determines an estimate for src ’s clock by using the most recent message from src : While the formula used for the lower error margin est^- is straightforward (est^- increases with a factor ≤ 1 due to p ’s drift), the fact that the upper error margin est^+ stays constant as soon as the last message from src is older than $(\mu_{(0)}^+ + \mu_{(1)}^+)(1 - \rho_p)$ might seem counter-intuitive, because it means that, as the last message from src gets older, the clock reading error $est^+ - est^-$ of the estimate becomes *smaller* than it was immediately after receiving the message.

The explanation for this phenomenon is that, in a system with reliable links, a lot of information can be gained from *not* receiving a message. As we will show in the next section, the end-to-end delay Δ , i.e., the message delay plus the queuing delay, of every *relevant* message is $\in [\delta^-, \delta^+]$ in the model of Sect. 3.2. If the last message m from *src* is $\mu_{(1)}^+ + x$ time units old (for some $x > 0$, plus $\mu_{(0)}^+$ for processing on the receiver side, plus some drift factor), we know that this message cannot have had an end-to-end delay Δ_m of δ^+ . Otherwise, the next message m' from *src* should have arrived by now. Actually, we know that Δ_m must be within $[\delta^-, \delta^+ - x]$, which is much more accurate than our original assumption of $[\delta^-, \delta^+]$. Clearly, the better p 's estimate for Δ_m is, the better p 's estimate of *src*'s hardware clock can be.

As can be inferred from Fig. 4, the maximum clock reading error is reached when the message is $(\mu_{(0)}^+ + \mu_{(1)}^+)(1 - \rho_p)$ hardware clock time units old:

$$\Gamma = \max\{est^+ - est^-\} = (1 + \rho_{src}) \left(\delta^+ + (\mu_{(0)}^+ + \mu_{(1)}^+)(1 - \rho_p)/(1 - \rho_p) \right) - (1 - \rho_{src}) \left(\delta^- + (\mu_{(0)}^+ + \mu_{(1)}^+)(1 - \rho_p)/(1 + \rho_p) \right)$$

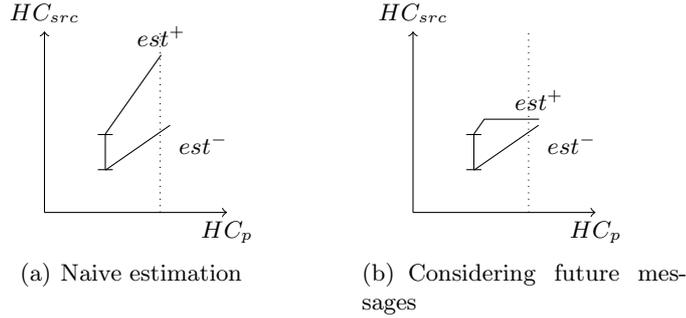


Fig. 4. p 's estimate of *src*'s hardware clock

Note that $(\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1 - \rho_p}{1 + \rho_p} (1 - \rho_{src})$ can be rewritten as $(\mu_{(0)}^+ + \mu_{(1)}^+) (1 - \rho_{src} - 2\rho_p) + \nu$, with

$$\nu = 2(\mu_{(0)}^+ + \mu_{(1)}^+) \rho_p \frac{\rho_p + \rho_{src}}{1 + \rho_p} \quad (2)$$

denoting a very small term in the order of $O(\mu^+ \rho^2)$,³ which is usually negligible. Thus, we have a maximum clock reading error of

$$\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu . \quad (3)$$

³ We use $\mu^+ = \max\{\mu_{(0)}^+, \mu_{(1)}^+\}$ and $\rho = \max\{\rho_{src}, \rho_p\}$ as abbreviations here.

3.4 Schedulability Analysis

Applying the system model restrictions from Sect. 3.2 to the algorithm allows us to make some general observations:

Observation 1. *Every timer set during some job starts processing at the end of that job. Formally, for all timer messages m : $(m \in \text{trans}(J)) \Rightarrow \exists J' : (\text{begin}(J') = \text{end}(J) \wedge \text{msg}(J') = m)$.*

Observation 2. *src sends an infinite number of messages to p . The begin times of the jobs sending these messages are between $\mu_{(1)}^-$ and $\mu_{(1)}^+$ time units apart.*

Given only FIFO links and a FIFO scheduling policy, a simple analysis would show that the end-to-end delay Δ_m , i.e., message (transmission) delay plus queuing delay, is within $[\delta^-, \delta^+]$, for all ordinary messages m . However, in the general setting with non-FIFO links and an arbitrary scheduling policy, it could, for example, be the case that a slow (message delay δ^+) message is “overtaken” by a fast message that was sent later but arrives earlier. If this fast message causes the slow one to be queued, the bound of δ^+ is exceeded. We can, however, solve this problem by filtering (line 10 of the algorithm) “irrelevant” messages, which have been overtaken by faster messages and, thus, might have had a longer end-to-end delay than δ^+ .

Of course, one obvious solution would be to put that “filter” inside the scheduler, preventing these irrelevant messages from being enqueued and allowing us to derive the bound $\Delta_m \in [\delta^-, \delta^+]$ by some very simple observations. However, the remainder of this section will demonstrate that this is not necessary: By showing that the bound is satisfied even if every message gets queued and filtering is done within the algorithm, we increase the coverage of our result to systems without low-level admission control.

Let $i \geq 1$ denote the i -th non-timer message sent from src (to p). We will show, by induction on i , that a certain bound holds for all messages. This generic bound will allow us to derive the upper bound of δ^+ for the end-to-end delay of relevant messages. First, we need a few definitions:

- J_i : The sending job of message i (on processor src).
- J'_i : The processing job of message i (on processor p).
- $\mathcal{F}_i := \{r : \text{begin}(J'_r) < \text{begin}(J'_i) \wedge r > i\}$: The set of “fast” messages $r > i$, that were processed (on p) before i . Informally speaking, this is the set of messages that have overtaken message i . Note that these messages are not necessarily *received* earlier than i , but *processed* earlier.
- $f(i) := \text{begin}(J_i) + \delta^+ + \sum_{j \in \mathcal{F}_i \cup \{i\}} \mu_{(0)}^j$. This is an upper bound on the “finishing” real time by which all messages $\leq i$ have been processed. $\mu_{(0)}^j$ denotes the actual processing time $\in [\mu_{(0)}^-, \mu_{(0)}^+]$ of message j ($= \text{duration}(J'_j)$).

Observe that $f(i) \geq f(i-1)$, since $\text{begin}(J_i)$ increases by at least $\mu_{(1)}^-$, whereas at most one message (whose processing takes at most $\mu_{(0)}^+$) “leaves” the set $\mathcal{F}_i \cup \{i\}$.

Lemma 1. *For all i holds: No later than $f(i)$, all J'_j , $1 \leq j \leq i$, finished processing; formally, $end(J'_j) \leq f(i)$.*

Proof. By induction. For the induction start $i = 0$, the statement is void since there is no job to complete ($f(0)$ can be defined arbitrarily). For the induction step, we can assume that the condition holds for $i - 1 \geq 0$, i.e., that

$$\forall 1 \leq j \leq i - 1 : end(J'_j) \leq f(i - 1) . \quad (4)$$

Assume by contradiction that the condition does not hold for i , i.e., that there is some $j \leq i$ such that $end(J'_j) > f(i)$. Since $f(i) \geq f(i - 1)$, choosing some $j < i$ immediately leads to a contradiction with (4). Thus,

$$end(J'_i) > f(i) . \quad (5)$$

Assume that $begin(J'_i) \leq begin(J'_{i-1})$. Since $end(J'_i) \leq end(J'_{i-1}) \leq f(i - 1) \leq f(i)$ by (4), this leads to a contradiction with (5). Thus, $begin(J'_i) > begin(J'_{i-1})$.

Since J'_i starts later than J'_{i-1} , $\mathcal{F}_{i-1} \subseteq \mathcal{F}_i$ (since $i \notin \mathcal{F}_{i-1}$, and, thus, all $r \in \mathcal{F}_{i-1}$, $r > i - 1$, are also in \mathcal{F}_i). Partition \mathcal{F}_i into $\mathcal{F}^{old} = \mathcal{F}_{i-1}$ and $\mathcal{F}^{new} = \mathcal{F}_i \setminus \mathcal{F}_{i-1}$. Note that $f(i) \geq f(i - 1) + \mu_{(1)}^- + \mu_{(0)}^i - \mu_{(0)}^{i-1} + \sum_{j \in \mathcal{F}^{new}} \mu_{(0)}^j$.

Let $t = f(i) - \mu_{(0)}^i - \sum_{j \in \mathcal{F}^{new}} \mu_{(0)}^j$. Note that $t \geq f(i - 1)$, which means that all messages J'_j , $j < i$, and all messages $\in \mathcal{F}^{old}$ have been processed by that time, and that $t \geq begin(J_i) + \delta^+$, which means that message i has arrived by time t . There are two cases, both contradicting (5):

1. There is some idle period in between t and $f(i)$: Since i has arrived by time t , this means that i has already been processed by time $f(i)$, due to our non-idling scheduler.
2. There is no idle period in between t and $f(i)$. Thus, we have a busy period of length $f(i) - t = \mu_{(0)}^i + \sum_{j \in \mathcal{F}^{new}} \mu_{(0)}^j$, which is only used to process messages from \mathcal{F}^{new} and message i (all other messages are done by $f(i - 1)$ due to the induction assumption). This also implies that i gets processed by $f(i)$. \square

We call i a “relevant” message if $\mathcal{F}_i = \emptyset$. Thus, the following lemma follows immediately:

Lemma 2. *The end-to-end delay Δ of all relevant messages is $\in [\delta^-, \delta^+]$.*

3.5 Proof of Correctness

Fix some rt-run ru and st-trace tr and let ev_{stable} be the *transition* st-event of the first relevant message from src to p . It will be shown that after ev_{stable} , src 's hardware clock stays within p 's values of est^- and est^+ .

Fix some global state $g \succ ev_{stable}$: Let m be the last relevant message from src to p fully processed before g , i.e., whose *transition* st-event $\prec g$, with t_j being the time that the job processing the message starts and t_s being the starting time

of the sending job of that message. Since $g \succ ev_{stable}$, such a message m must exist. Observe that line 10 in the algorithm ensures that only relevant messages cause a state transition in p .

Let $t = time(g)$ and $\Delta_m = t_j - t_s$ (cf. Fig. 5). Note that Δ_m corresponds to δ_m , the message delay, plus any queuing delay m may experience. (For simplicity, Fig. 5 shows a case without queuing.) Due to Lemma 2, we know that Δ_m is bounded by $[\delta^-, \delta^+]$. In addition, we define the following *drift factors*:

$$dr_p = \frac{HC_p(t) - HC_p(t_j)}{t - t_j} \quad (6a)$$

$$dr_{src} = \frac{HC_{src}(t) - HC_{src}(t_s)}{t - t_s} \quad (6b)$$

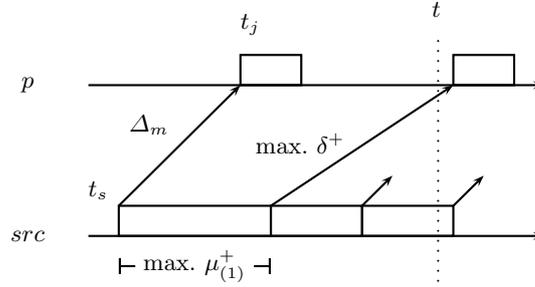


Fig. 5. Two consecutive messages from src to p

Clearly, $dr_{src} \in [1 - \rho_{src}, 1 + \rho_{src}]$ and $dr_p \in [1 - \rho_p, 1 + \rho_p]$. These definitions allow us to derive the following by applying (6a) and the definition of Δ_m :

$$\begin{aligned} HC_{src}(t) &= HC_{src}(t_s) + (t - t_s)dr_{src} = HC_{src}(t_s) + ((t - t_j) + (t_j - t_s))dr_{src} \\ &= HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{dr_p} + \Delta_m \right) dr_{src} . \end{aligned}$$

Since $HC_{src}(t)$ can never become less than the minimum of this expression,

$$\begin{aligned} HC_{src}(t) &\geq \min_{\substack{dr_p \\ dr_{src} \\ \Delta_m}} \left\{ HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{dr_p} + \Delta_m \right) dr_{src} \right\} \\ &= HC_{src}(t_s) + \left(\frac{HC_p(t) - HC_p(t_j)}{1 + \rho_p} + \delta^- \right) (1 - \rho_{src}) \\ &= s_p(g).send_{hc} + (age_p(g)/(1 + \rho_p) + \delta^-) (1 - \rho_{src}) . \end{aligned}$$

Hence, we have:

Lemma 3. $HC_{src}(t) \geq est_p^-(g)$.

Doing the same for the maximum of the above expression yields a similar result:

Lemma 4. $HC_{src}(t) \leq s_p(g).send_hc + (age_p(g)/(1 - \rho_p) + \delta^+) (1 + \rho_{src})$.

This value is still greater than est^+ . Thus, we have to use another approach to prove our upper bound on HC_{src} : First, we note that the real time between t_s and t is bounded:

Lemma 5. $t - t_s \leq \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+$.

Proof. We will again use the numbering of messages as in Sect. 3.4. Recall Fig. 5 and assume by contradiction that $i = m$ was sent earlier, i.e., that $t_s < t - \delta^+ - \mu_{(0)}^+ - \mu_{(1)}^+$. Since the (real-time) delay between two consecutive messages sent from src to p is at most $\mu_{(1)}^+$ (cf. Observation 2), $t'_s < t - \delta^+ - \mu_{(0)}^+$ holds for t'_s , the begin time of the job sending $i + 1$. Since i is a relevant message, $i + 1$ must be processed later than i .

Consider \mathcal{F}_{i+1} , the set of messages sent after message $i + 1$ but processed earlier; \mathcal{F}_{i+1} might also be \emptyset , if $i + 1$ is a relevant message. Let J'_{i+1} be the job processing message $i + 1$ and let $\mathcal{J} = \mathcal{F}_{i+1} \cup \{i + 1\}$. By Lemma 1 we know that $end(J'_{i+1}) \leq f(i + 1) = t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu_{(0)}^j$.

Let x be the first message $\in \mathcal{J}$ that will be processed at p . Clearly, x must be a relevant message. Otherwise, there would be some $y > x \geq i + 1$ such that J'_y is processed before J'_x . However, if $begin(J'_y) < begin(J'_x) \leq begin(J'_{i+1})$, then $y \in \mathcal{J}$, contradicting the assumption that x is the first message $\in \mathcal{J}$ that will be processed.

We know that all of \mathcal{J} have been processed before $end(J'_{i+1}) \leq t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu_{(0)}^j$ and that processing all of \mathcal{J} takes at least $\sum_{j \in \mathcal{J}} \mu_{(0)}^j$ time units. Thus, at $t'_s + \delta^+$, at least one of \mathcal{J} starts processing, is currently being processed or has already been processed. Since J'_x is the first such job, $begin(J'_x) \leq t'_s + \delta^+$.

Recalling $t'_s < t - \delta^+ - \mu_{(0)}^+$ from the beginning of the proof leads to $begin(J'_x) < t - \mu_{(0)}^+$. Since x is a relevant message and processing x takes at most $\mu_{(0)}^+$ time units, its *transition* st-event is no later than at $t' < t$. This contradicts our assumption that $m = i$ is the last relevant message from src fully processed by p before g . \square

Combining Lemma 5 with (6b) results in

$$\frac{HC_{src}(t) - HC_{src}(t_s)}{dr_{src}} \leq \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+$$

and hence

$$HC_{src}(t) \leq HC_{src}(t_s) + (\delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) dr_{src}$$

$$\begin{aligned}
&\leq \max_{dr_{src}} \left\{ HC_{src}(t_s) + (\delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) dr_{src} \right\} \\
&= HC_{src}(t_s) + (\delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) (1 + \rho_{src}) \\
&= s_p(g).send_hc + (\delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) (1 + \rho_{src})
\end{aligned}$$

which, combined with Lemma 4 and the definition of est^+ , yields the following result:

Lemma 6. $HC_{src}(t) \leq est_p^+(g)$.

Combining Lemma 3 and 6 finally yields Theorem 1, which proves that the algorithm in Fig. 3 indeed solves the remote clock estimation problem according to Definition 1.

Theorem 1 (Correctness). *For all global states $g \succ ev_{stable}$, where ev_{stable} is the transition st-event of the first message from src to p , it holds that $HC_{src}(time(g)) \in [est_p^-(g), est_p^+(g)]$. The maximum clock reading error $\Gamma = \max\{est^+ - est^-\}$ is*

$$\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu,$$

with the usually negligible term $\nu = O(\rho^2\mu)$ given by (2).

4 Lower Bound

In this section, we will show that the upper bound on Γ determined in Theorem 1 is tight, i.e., that the algorithm in Fig. 3 is optimal w.r.t. the maximum clock reading error.

4.1 System Model

For the lower bound proof, we require that $\delta^+(1 - \rho) \geq \delta^-(1 + \rho)$ and that $\mu_{(\ell)}^+(1 - \rho) \geq \mu_{(\ell)}^-(1 + \rho)$, for $\ell \in \{0, 1\}$ and $\rho \in \{\rho_{src}, \rho_p\}$. These lower bounds on the message and processing delay uncertainties prevent the processors from using their communication subsystems or their schedulers to simulate a clock that has a lower drift rate than their hardware clocks.

To simplify the presentation, we will make three additional assumptions. In Sect. 4.3, we will briefly discuss the consequences of dropping these.

1. $\delta^- \geq \mu_{(0)}^+$. This allows the adversary to choose a scenario where no *send* and/or *transition* st-event in a job occurs earlier than $\mu_{(0)}^+(1 - \rho)$ hardware clock time units after the beginning of the job.
2. We assume that the algorithm knows when it has stabilized, i.e., that p switches a Boolean register *stable* (initially false) when the algorithm has stabilized. In the algorithm in Fig. 3, p would set its *stable* register after completing the processing of the first relevant message from src .
3. There is at least one message from src arriving at p after p has set its *stable* register.

4.2 Proof

Assume by contradiction that there exists some deterministic algorithm \mathcal{A} that allows processor p to continuously estimate processor src 's hardware clock with a maximum clock reading error $\max\{est^+ - est^-\} < \Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu$. Using an adaption of the well-known shifting and drift scaling techniques to st-traces, which is technically quite intricate due to the multiple state transitions involved in a job, we show that there are indistinguishable rt-runs of \mathcal{A} that cause a clock reading error of at least Γ .

Definition 2. *Since our proof uses an indistinguishability argument, we will use the notation $p : tr[ev_A, ev_\Omega] \approx tr'[ev'_A, ev'_\Omega]$ to denote that, for processor p , st-trace tr from st-event ev_A to ev_Ω is indistinguishable from st-trace tr' from st-event ev'_A to ev'_Ω , where ev_A, ev_Ω, ev'_A and ev'_Ω all occur on processor p . Intuitively, this means that p cannot detect a difference between the two st-trace segments.*

Let $(ev_1, ev_2, \dots, ev_\eta)$ and $(ev'_1, ev'_2, \dots, ev'_{\eta'})$ be the restrictions of st-traces tr and tr' to send and transition st-events occurring on processor p , beginning with $ev_A = ev_1$ and $ev'_A = ev'_1$, and ending with $ev_\Omega = ev_\eta$ and $ev'_\Omega = ev'_{\eta'}$. Indistinguishability means that $\eta = \eta'$ and $ev_i = ev'_i$ for all $i, 1 \leq i \leq \eta$, except for the real time of the events, i.e., $time(ev_i) = time(ev'_i)$ is not required. In fact, indistinguishability is even possible if the st-trace segments are of different real time length, i.e., if $time(ev_\Omega) - time(ev_A) \neq time(ev'_\Omega) - time(ev'_A)$. However, $HC_p^{tr}(time(ev_i)) = HC_p^{tr'}(time(ev'_i))$ must of course be satisfied, i.e., the hardware clock values of all matching st-events must be equal.

The notations $tr[t_A, ev_\Omega]$, $tr[ev_A, t_\Omega]$ and $tr[t_A, t_\Omega]$ are sometimes used as short forms for $tr[ev_A, ev_\Omega]$, with ev_A being the first st-event with $time(ev_A) \geq t_A$ and ev_Ω being the last st-event with $time(ev_\Omega) \leq t_\Omega$. Parenthesis are used to denote $<$ instead of \leq , e.g. $tr[0, t_\Omega)$.

Likewise, global states are sometimes used as boundaries: $tr[g_A, \dots]$ and $tr[\dots, g_\Omega]$ actually mean the first st-event on p succeeding g_A and the last st-event on p preceding g_Ω . Clearly, $s_p(g_\Omega) = s_p(g'_\Omega)$ if $p : tr[\dots, g_\Omega] \approx tr'[\dots, g'_\Omega]$.

Note: Since $est^{-/+}$ can be a function of the state of p and the current hardware clock value, it does not suffice to show $s_p(g_1) = s_p(g_2)$ in some indistinguishable st-traces tr_1 and tr_2 . If we want to prove that $est^{-/+}$ is equal in both st-traces, we also need to show that $HC_p^{tr_1}(time(g_1)) = HC_p^{tr_2}(time(g_2))$, which is more difficult in our setting than in a drift-free environment.

Let tr_1 be an st-trace of some rt-run ru_1 of \mathcal{A} where the adversary makes the following choices:

- Both processors boot (i.e., receive an initial input message, if required) at time $t = 0$.
- $HC_p(0) = 0$, $HC_{src}(0) = 0$.
- Every message from src takes δ^+ time units.
- Every message to src takes δ^- time units.

- Every job sending ℓ message takes $\mu_{(\ell)}^+$ time units.
- No *transition* or *send* st-event occurs earlier than $\mu_{(0)}^+(1 - \rho)$ hardware clock time units after the beginning of the job ($\rho = \rho_p$ for p and $\rho = \rho_{src}$ for src).
- src 's clock has a drift factor of $1 + \rho_{src}$
- p 's clock has a drift factor of $1 - \rho_p$.

Since \mathcal{A} is a correct algorithm, the execution will eventually become stable. Let $ev_{sta,1}$ be the *transition* st-event at which p switches its *stable* register in tr_1 . Let m be an arbitrary message from src to p , sent by a job starting at time t_s and arriving through a receive event at time t_r , with $t_r > time(ev_{sta,1})$. By assumption (cf. Sect. 4.1), such a message exists.

Let tr_2 be an st-trace of another rt-run ru_2 of \mathcal{A} where the adversary behaves exactly as specified for tr_1 (using the same scheduling policy), with the following differences (cf. Fig. 6):

- src boots at time $t = \varepsilon$ (instead of 0).
- $HC_{src}(\varepsilon) = 0$ (instead of $HC_{src}(0) = 0$).
- Every message from src takes δ^- time units (instead of δ^+).
- Every message to src takes δ^+ time units (instead of δ^-).
- After $t_s + \varepsilon$, src 's clock has a drift factor of $1 - \rho_{src}$.
- After t_r , p 's clock has a drift factor of $1 + \rho_p$.
- After t_r , on p , every job sending ℓ messages takes $\mu_{(\ell)}^+ \frac{1 - \rho_p}{1 + \rho_p}$ time units (instead of $\mu_{(\ell)}^+$). Note that $\mu_{(\ell)}^+ \frac{1 - \rho_p}{1 + \rho_p} \in [\mu_{(\ell)}^-, \mu_{(\ell)}^+]$ (cf. Sect. 4.1). Likewise, *send* and *transition* st-events occur no earlier than $\mu_{(0)}^+ \frac{1 - \rho_p}{1 + \rho_p}$ time units (and hence no earlier than $\mu_{(0)}^+(1 - \rho_p)$ hardware clock time units, as in tr_1) after the beginning of their job.⁴

Lemma 7. $p : tr_1[0, t_r] \approx tr_2[0, t_r]$ and $src : tr_1[0, t_s] \approx tr_2[\varepsilon, t_s + \varepsilon]$.

Proof. The lemma follows directly from the following observations:

- The initial states are the same in ru_1 and ru_2 .
- All st-events within that time occur at the same hardware clock time and in the same order (on each processor).

A formal proof can be obtained by induction on the st-events of ru_1 or ru_2 , using these properties, or by adapting any of the well-known “shifting argument” proofs. \square

Since $time(ev_{sta,1}) \leq t_r$, this lemma also implies⁵ the existence of a corresponding st-event $ev_{sta,2}$ in tr_2 , in which p sets its *stable* register.

⁴ If there is a job J starting before but ending after t_r , its duration is weighted proportionally, i.e., $duration(J) = (\mu_{(\ell)}^+ - x) + x \frac{1 - \rho_p}{1 + \rho_p}$, with $x = end(J) - t_r$. The same is done with the minimum offset for *send* and *transition* st-events in a job.

⁵ This could not be inferred that easily if the algorithm did not know when it had stabilized.

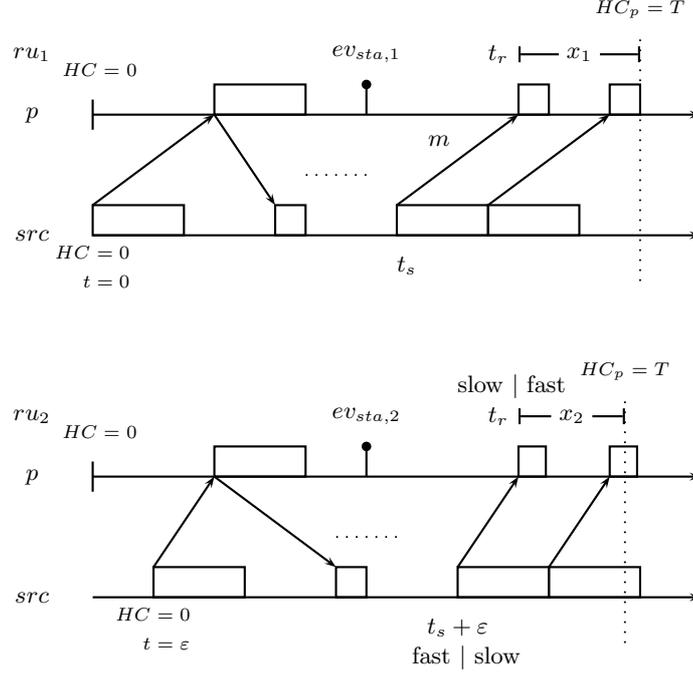


Fig. 6. ru_1 and ru_2 (timer messages not shown); $x_1 = \mu_{(0)}^+ + \mu_{(1)}^+$; $x_2 = x_1 \frac{1-\rho_p}{1+\rho_p}$

Lemma 8. For all $t_1, t_2 \geq t_r$: $HC_p^{tr_1}(t_1) = HC_p^{tr_2}(t_2) \Leftrightarrow t_2 = (t_1 - t_r) \frac{1-\rho_p}{1+\rho_p} + t_r$.

Proof. The proof follows directly from the drift factors of p in tr_1 and tr_2 , i.e., for all $t_1, t_2 \geq t_r$: $HC_p^{tr_1}(t_1) = t_1(1-\rho_p)$ and $HC_p^{tr_2}(t_2) = t_r(1-\rho_p) + (t_2 - t_r)(1+\rho_p)$.

Let g_1 and g_2 be defined as follows:

- g_1 is the first global state in tr_1 at time $t_r + \mu_{(0)}^+ + \mu_{(1)}^+$, i.e., the global state preceding the first st-event (if any) happening at $t_r + \mu_{(0)}^+ + \mu_{(1)}^+$.
- g_2 is the first global state in tr_2 at time $t_r + (\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p}$.

Clearly, by Lemma 8, p 's hardware clock values at g_1 and g_2 are equal (denoted T and represented by the dotted line in Fig. 6).

Lemma 9. $p : tr_1[0, g_1] \approx tr_2[0, g_2]$

Proof. By Lemma 7, tr_1 and tr_2 are indistinguishable for src until t_s and $t_s + \epsilon$, respectively. Since src starts a job of duration $\mu_{(1)}^+$ in ru_1 at time t_s , a corresponding job is started in ru_2 at time $t_s + \epsilon$. Both jobs send the same message m to p . Since our system model does not allow preemption, src 's next message to p can be sent no earlier than $t_s + \mu_{(1)}^+$ (tr_1) and $t_s + \epsilon + \mu_{(1)}^+$ (tr_2). Thus, by the definition of message (transmission) delays in ru_1 and ru_2 , the earliest

time that p can receive another message from src (after the reception of m) is $t_r + \mu_{(1)}^+$ (in both tr_1 and tr_2 , cf. Fig. 6).

Thus, the only jobs occurring at p in ru_1 and ru_2 after the reception of m (at time t_r) and before $t_r + \mu_{(1)}^+$ are jobs caused by timer messages, by message m or by messages that have been received earlier. These messages, however, cannot “break” the indistinguishability: Since (a) p ’s hardware clock is speeded up and (b) the processing time of jobs on p are slowed down by the same factor ($\frac{1-\rho_p}{1+\rho_p}$), the hardware clock times of all jobs (starting and ending times) as well as all state transitions are equal in tr_1 and tr_2 , as long as no new external message reaches p . Since this does not happen before $t_r + \mu_{(1)}^+$, we can conclude that tr_1 and tr_2 are indistinguishable until hardware clock time $T' := HC_p^{tr_1}(t_r + \mu_{(1)}^+)$, at which a message might arrive in ru_1 that did not yet arrive in ru_2 (since, in ru_2 , only $t_r + \mu_{(1)}^+ \frac{1-\rho_p}{1+\rho_p}$ real time units have passed yet at T'). Thus, $p : tr_1[0, t_r + \mu_{(1)}^+] \approx tr_2[0, t_r + \mu_{(1)}^+ \frac{1-\rho_p}{1+\rho_p}]$.

If a job (J_1 in tr_1 , J_2 in tr_2) is currently running at hardware clock time T' , a message reception does not change any (future) state transitions of that job, due to no-preemption. Thus, the indistinguishability continues until $T'' := HC_p^{tr_1}(end(J_1)) = HC_p^{tr_2}(end(J_2))$. (If no job was running at hardware clock time T' , let $T'' := T'$, cp. Fig. 6.) At hardware clock time T'' , the schedulers of ru_1 and ru_2 might choose different jobs to be executed next (since the message from src arrived at different hardware clock times in ru_1 and ru_2). However, due to our assumption that the adversary causes all state transitions to occur no earlier than $\mu_{(0)}^+(1 - \rho_p)$ hardware clock time units after the beginning of the job, the state of p is still equal in ru_1 and ru_2 until hardware clock time $T'' + \mu_{(0)}^+(1 - \rho_p)$. As $T'' \geq T'$, this corresponds to some real time of at least $t_r + \mu_{(0)}^+ + \mu_{(1)}^+$ in tr_1 and at least $t_r + (\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p}$ in tr_2 . Since g_1 and g_2 are, by definition, the first global states at these real times, no state transition breaking the indistinguishability can have occurred yet. \square

Lemma 10. $HC_{src}^{tr_1}(time(g_1)) - HC_{src}^{tr_2}(time(g_2)) = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu$

Proof.

$$\begin{aligned}
HC_{src}^{tr_1}(time(g_1)) &= HC_{src}^{tr_1}(t_r + \mu_{(0)}^+ + \mu_{(1)}^+) = HC_{src}^{tr_1}(t_s + \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) \\
&= (t_s + \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+)(1 + \rho_{src}) \\
&= t_s + \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+ + \rho_{src}(t_s + \delta^+ + \mu_{(0)}^+ + \mu_{(1)}^+) \\
HC_{src}^{tr_2}(time(g_2)) &= HC_{src}^{tr_2}\left(t_r + (\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p}\right) \\
&= HC_{src}^{tr_2}\left(t_s + \varepsilon + \delta^- + (\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p}\right) \\
&= HC_{src}^{tr_2}(\varepsilon) + t_s(1 + \rho_{src}) + \left(\delta^- + (\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p}\right)(1 - \rho_{src})
\end{aligned}$$

Again, $(\mu_{(0)}^+ + \mu_{(1)}^+) \frac{1-\rho_p}{1+\rho_p} (1 - \rho_{src})$ can be rewritten as $(\mu_{(0)}^+ + \mu_{(1)}^+) (1 - \rho_{src} - 2\rho_p) + \nu$, with ν , defined in (2), denoting a small term in the order of $O(\mu^+ \rho^2)$. Thus,

$$HC_{src}^{tr_2}(time(g_2)) = t_s + \delta^- + \mu_{(0)}^+ + \mu_{(1)}^+ + \rho_{src}(t_s - \delta^- - \mu_{(0)}^+ - \mu_{(1)}^+) - 2\rho_p(\mu_{(0)}^+ + \mu_{(1)}^+) + \nu . \quad \square$$

We can now prove our lower bound theorem:

Theorem 2. *There is no clock estimation algorithm \mathcal{A} that allows processor p to estimate processor src 's clock with a maximum clock reading error of less than $\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu$.*

Proof. Lemmas 8 and 9 have shown that $s_p(g_1) = s_p(g_2)$ and $HC_{src}^{tr_1}(time(g_1)) = HC_{src}^{tr_2}(time(g_2))$. Since $est^{-/+}$ on p is a function of the local state and the hardware clock time, it holds that $est_p^-(g_1) = est_p^-(g_2)$ and $est_p^+(g_1) = est_p^+(g_2)$. By the assumption that \mathcal{A} is a correct algorithm which allows p to estimate src 's hardware clock with a maximum clock reading error $< \Gamma$, the following condition must hold: \mathcal{A} always maintains two values est^- and est^+ , such that

$$HC_{src}^{tr_1}(time(g_1)) \in [est^-, est^+] \quad \text{and} \quad HC_{src}^{tr_2}(time(g_2)) \in [est^-, est^+]$$

with $est^+ - est^- < \Gamma$.

Lemma 10 reveals, however, that $HC_{src}^{tr_1}(time(g_1)) - HC_{src}^{tr_2}(time(g_2)) = \Gamma$, which provides the required contradiction. \square

4.3 The System Model Revisited

In Sect. 4.1, three assumptions were introduced, which simplify the lower bound proof. In this section, we will briefly discuss the consequences of dropping these assumptions.

1. If we replace the assumption $\delta^- \geq \mu_{(0)}^+$ by the weaker criterion that no *send* or *transition* st-event occurs before $x \cdot (1 - \rho)$ hardware clock time units, with $0 \leq x < \mu_{(0)}^+$ (thereby restricting the adversary's power in tr_1 and tr_2), the precision lower bound is decreased to $\varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(x + \mu_{(1)}^+) - \nu'$, i.e., $\mu_{(0)}^+$ gets replaced by x . Analogously, ν' equals ν with $\mu_{(0)}^+$ replaced by x .
2. If the algorithm need not know when it has stabilized, we must prove that one can always find two st-traces tr_1 and tr_2 where p has stabilized before t_r , recall Fig. 6. Informally, this can be guaranteed due to the fact that even eventual properties are always satisfied within bounded time in a closed model like the RT-Model (where all delays are bounded), see e.g. [21].

3. *There is at least one message from src arriving at p after p has set its stable register.* If this condition is not satisfied, we have two cases:

Case 1: After p has set its *stable* register, no more messages are exchanged between p and src . In that case, it is trivial to create an indistinguishable rt-run in which p has a different drift rate. Since no messages are exchanged, neither p nor src ever detects a difference between the two rt-runs and we can choose a global state g arbitrarily far in the future to create an arbitrarily large discrepancy between p 's estimate and src 's hardware clock.

Case 2: After p has set its *stable* register, only messages from p to src are sent. In that case, the proof is quite similar to the one in Sect. 4.2. Since only src receives messages here, only src can detect a difference between two rt-runs with different drift rates. Consider Fig. 6 with the labels p and src reversed. In complete analogy to Lemma 9, we can argue that src cannot detect a difference until m' , the second message from p , has arrived. For p to change its estimate, this information needs to be transmitted back to p .⁶ Therefore we have an additional δ^- for the message transmission plus $\mu_{(0)}^-$ (or x , see Assumption 1) required by p until a state transition in response to this message can be performed. Thus, detecting a change in this case takes at least δ^- time units longer than in the case analyzed in Sect. 4.2, finally leading to the same contradiction.

5 Synchronizing Clocks

In this section, we will move from the two-processor clock estimation problem to its application in external and internal clock synchronization.

Since the problems analyzed in this section involve more than two processors, a job may send (non-timer) messages to more than one recipient. Thus, we will also use subscripts (ℓ) on the message delay bounds $\delta_{(\ell)}^-$ and $\delta_{(\ell)}^+$ here, which give the number of recipients to which the sending job sends a message. As detailed in Sect. 2.1, $\delta_{(\ell)}^-$, $\delta_{(\ell)}^+$ as well as $\varepsilon_{(\ell)} := \delta_{(\ell)}^+ - \delta_{(\ell)}^-$ are assumed to be non-decreasing w.r.t. ℓ .

5.1 Admission Control

In the classic zero-step-time computing model usually employed in the analysis of distributed algorithms, a Byzantine faulty processor can send an arbitrary number of messages with arbitrary content to all other processors. This “arbitrary number”, which is not an issue when assuming zero step times, could cause problems in the real-time model: It would allow a malicious processor to create a huge number of jobs at any of its peers. Consequently, we must ensure that messages from faulty processors do not endanger the liveness of the algorithm at correct processors.

⁶ Since src detected a difference, the rt-runs are no longer indistinguishable. Thus, messages from src to p are possible in this (shifted) rt-run.

In the following sections, we assume the presence of an *admission control* component in the scheduler or in the network controller, which restricts the set of messages reaching the message queue.

5.2 External Clock Synchronization

In large-scale distributed systems such as the Internet, hierarchical synchronization algorithms like NTP have proven to be very useful. With respect to smaller networks, our results indicate that it pays off to minimize the dominant factor ε , which is severely increased by multi-hop communication. Thus, direct communication between the source and the “clients” will usually lead to tighter synchronization.

For this section, let n specify the number of processors in the system, ρ_{src} the drift rate of the source processor and ρ_* the drift rate of all other processors. The goal is for each processor $p \neq src$ to estimate src ’s clock as close as possible. The maximum estimation error is called *accuracy* α here. Note that external clock synchronization obviously implies internal clock synchronization with precision $\pi = 2\alpha$.

Consider a variant of the algorithm presented in Sect. 3, where src sends its hardware clock value not only to p but to all of the other $n - 1$ processors, and the receiver uses the midpoint of $[est^-, est^+]$ as its estimation of src ’s clock. Admission control is performed by only accepting messages from src . An obvious generalization of the analysis in Sect. 3 shows that, if src is correct, the worst case accuracy for any correct receiver p is $\alpha = \Gamma/2$ with

$$\Gamma = \varepsilon_{(\ell)} + \rho_{src}(\delta_{(\ell)}^- + \delta_{(\ell)}^+) + 2(\rho_{src} + \rho_*)(\mu_{(0)}^+ + \dot{\mu}) - \nu \quad ,$$

(cf. Theorem 1), where ℓ depends on the broadcasting method, $\dot{\mu}$ is the transmission period (see below), and $\nu = O(\dot{\mu}\rho^2)$ refers again to a usually negligible term. The precision achieved by any two correct receivers p, q is hence $\pi = \Gamma$.

In the real-time computing model, the required broadcasting can actually be implemented in two ways:

- (a) src uses a single job with broadcasting to distribute its clock value. In this case, the duration of each of its jobs is $\in [\mu_{(n-1)}^-, \mu_{(n-1)}^+]$ and the message delay of each message is $\in [\delta_{(n-1)}^-, \delta_{(n-1)}^+]$. Thus, $\ell = n - 1$ and $\dot{\mu} = \mu_{(n-1)}^+$.
- (b) src sends unicast messages to every client, in a sequence of $n - 1$ separate jobs that send only one message, i.e., $\ell = 1$. This reduces the message delay uncertainty from $\varepsilon_{(n-1)}$ to $\varepsilon_{(1)}$, but increases the period $\dot{\mu}$ in which every processor p receives src ’s message from $\mu_{(n-1)}^+$ to $(n - 1) \cdot \mu_{(1)}^+$.

5.3 Fault-Tolerant Internal Clock Synchronization

As outlined in the introduction, remote clock estimation is only a small, albeit important, part of the internal clock synchronization problem. In [17], Fetzer and Christian presented an optimal round- and convergence-function-based solution

to this problem. They assume the existence of a generic remote clock reading method, which returns the clock value of a remote clock within some symmetric error. Thus, extending their work is a perfect choice for demonstrating the applicability of our optimal clock estimation result in the context of internal clock synchronization.

The algorithm of [17] works as follows: Periodically, at the same logical time at every processor, the current clock values of all other clocks are estimated. These estimates are passed on to a fault-tolerant *convergence function*, which provides a new local clock value that is immediately applied for adjusting the clock. Provided that all clocks are sufficiently synchronized initially and the resynchronization period is chosen sufficiently large, the algorithm maintains a precision of $4\Lambda + 4\rho r_{max} + 2\rho\beta$, where r_{max} denotes the resulting maximum real-time round duration and β the maximum difference of the resynchronization times of different processors. Λ is the maximum clock reading error margin, i.e., $\Lambda = \Gamma/2$ in our setting.

In Appendix B, we present a detailed analysis of how to combine our clock estimation method with their convergence function, resulting in an internal clock synchronization algorithm that tolerates up to f arbitrary faulty processors, for $n > 3f$. The analysis includes a pseudo-code implementation and a correctness proof, which just establishes conditions that guarantee the preconditions of the proofs in [17]. The result is summarized in the following theorem:

Theorem 3. *For a sufficiently large resynchronization period and sufficiently close initial synchronization, fault-tolerant internal clock synchronization can be maintained within $\pi = 2\Gamma + 4\rho r_{max} + 2\rho\beta$ with $\Gamma = \varepsilon_{(n-1)} + \rho(\delta_{(n-1)}^- + \delta_{(n-1)}^+) + 4\rho\mu_{(n-1)}^+ - O(\mu^+\rho^2)$.*

6 Conclusions

We presented an algorithm solving the problem of continuous remote clock estimation in the real-time computing model, which guarantees a maximum clock reading error of $\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu_{(0)}^+ + \mu_{(1)}^+) - \nu$. Using an elaborate shifting and scaling argument, we also established a matching lower bound for the maximum clock reading error. This result leads to some interesting conclusions, which could aid real-time system designers in fine-tuning their systems:

- ε , the message delay uncertainty, dominates everything else, since it is the only parameter that is not scaled down by some clock drift $\rho \ll 1$. This matches previous results on drift-free clock synchronization [5].
- Both sender and receiver clock drift influence the attainable precision. However, the drift of the source clock has a bigger impact, since it affects not only the term involving the processing times $\mu_{(0)}^+ + \mu_{(1)}^+$, but also the (potentially larger) term involving message delays.

- Since the presented algorithm does not send messages from p to src , we can conclude that round-trips, which are well-known to improve remote clock estimation in the average case, do not improve the attainable worst case error.
- The lower bound implies that optimal precision induces a very high network load: $\mu_{(1)}^+$, the interval, in which messages from src to p are sent, is directly proportional to the maximum clock reading error.

Furthermore, we have shown how our optimal clock reading method can be used to solve the problem of external clock synchronization, and how it can be plugged into an existing internal clock synchronization algorithm.

Appendix

A State Transition Traces

Four distinct types of global state changes can occur in a st-trace. Formally, each of these can be represented by a *state transition event* (short: *st-event*) ev with $type(ev) \in \{start_processing, send, transition, input\}$.

- (*start_processing* : 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 $newstate(ev) = s'$ ⁷.
- (*input* : t, m):
At time $time(ev) = t$, input message $msg(ev) = m$ arrives from an external source.

Every rt-run can be mapped to multiple *state transition traces* (short: *st-traces*). Every such st-trace tr is a sequence of st-events with associated hardware clocks HC_p^{tr} (the superscript is omitted if clear from context). An st-trace is created by following a simple transformation rule:

Definition 3. *Each job J starting at time t with duration d on processor p triggered by some message m is mapped to (*start_processing* : 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.*

*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 time of a send st-event ($t' \in [t, t + d]$) sending some message m must be chosen such that message causality is not violated, i.e., $t' \leq t''$, with t'' being the *start_processing* st-event of the job processing m .*

Recall, however, that the message delay δ is measured from the start of the job, not from the time of the send st-event in the st-trace.

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

⁷ We use $oldstate(ev)$ and $newstate(ev)$ to refer to the states of a *transition* st-event; note that they do not necessarily match the first and last state of the transition sequence of a job, as $oldstate$ and $newstate$ of an st-event might, as well, be intermediate states in a job.

Formally, the global state g is defined as a tuple $(t, s_1, \dots, s_n, pending_msgs)$ containing the time $time(g) = t$, the state of all processors $s_1(g) \dots s_n(g)$ and the set of unprocessed messages $pending_msgs(g)$, which comprise messages in transit and messages that have been received but not processed yet. To ensure a well-defined global state even in between st-events, we annotate an st-trace by adding (at most countably many) sets of (either one or continuum many) global states:

- *At the beginning:*
Insert a set $\{(t, istate_1, \dots, istate_n, \{\}) : 0 \leq t \leq t'\}$, 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 $\{(t, s_1, \dots, s_n, pending_msgs) : time(ev) \leq t \leq time(ev')\}$ containing the global state after ev but before ev' . The effects of st-events on the global state are as follows:
 - $(start_processing : t, p, m)$ 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 $\{(t, s_1, \dots, s_n, \{\}) : time(ev) \leq t\}$ containing the global state after ev , i.e., the final state.

The state sets are totally ordered. Note carefully, however, that if, at some time t , k st-events occur in tr , there are $k + 1$ global states g with $time(g) = t$ in the annotated st-trace.

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 $tr \cup gstates(tr)$. We will omit the superscript if it is clear from context.

A distributed computing *problem* is specified as a predicate on some st-trace tr and the associated hardware clocks HC_p^{tr} . An algorithm solves a given problem if all st-traces of all rt-runs of this algorithm satisfy this predicate.

To summarize, the creation of an admissible rt-run and a matching st-trace can be seen as a game of a player (the algorithm) against an adversary in the “arena” of a system $s = (n, [\delta_{(\ell)}^-, \delta_{(\ell)}^+], [\mu_{(\ell)}^-, \mu_{(\ell)}^+])$. 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 and the hardware clock parameters (such as initial value or drift, depending on the hardware clock model used),
- for every message sent in a job, together with $\ell - 1$ other messages, choose a value within $[\delta_{(\ell)}^-, \delta_{(\ell)}^+]$ 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 sending ℓ messages, choose
 - a value within $[\mu_{(\ell)}^-, \mu_{(\ell)}^+]$ for its processing time (and associated overhead, e.g. for scheduling) and
 - the exact time of every state transition and message sending (within the processing time and within causality restrictions).
- define the scheduling policy (but see below).

To make our results as strong as possible, we will allow the adversary to control the scheduling policy in correctness proofs, but will assume an algorithm-controlled scheduler in lower bound proofs.

B Internal Clock Synchronization

Since it is assumed that the local hardware clocks cannot be modified, the (*logical*) clock of round k is represented as the sum of the current hardware clock reading and a local variable, the *adjustment value* $adj[k]$. Unless we explicitly mention “hardware clock values”, we will refer to this (adjusted) logical clock when talking about “clock values” in the remainder of this section.

The main idea of the algorithm of [17] is as follows: Every *round_len* hardware clock time units (the *resynchronization period*), which determines the end of round k , every processor p estimates the round k clock values of all remote clocks. An (optimal) fault-tolerant convergence function is applied to these clock estimates, which computes the initial value for p ’s local round $k + 1$ clock. To use our remote clock estimation algorithm with such a round-based algorithm, every processor p must broadcast messages containing its current round k clock value at the end of round k . This broadcasting

- must start early enough to ensure that every other processor q receives at least one round k message from p before applying its convergence function [*Condition C1*] and
- must not terminate too early to ensure that a “recent” round k message from p exists at q when the clock reading method is used by the convergence function. [*Condition C2*]

Since all processors need to broadcast simultaneously at the end of the round, scheduling delays are created which—in contrast to the “simple” case of external clock synchronization—influence the end-to-end delays of other messages. This would require replacing the message delay bounds $[\delta^-, \delta^+]$ in our bound on the maximum clock reading error T with the probably larger end-to-end delay bounds $[\Delta^-, \Delta^+]$. Thus, given some scheduling policy *pol*, a detailed real-time schedulability analysis would be needed for determining bounds on Δ^- and Δ^+ .

An alternative approach, which entirely avoids this problem, goes by assuming a more powerful hardware: If, upon receiving a message from q containing some clock value, the network controller of the destination processor p were capable of automatically

- storing the content of the message in some variable $send_cv[q]$ and

- storing p 's current hardware clock value in some variable $rcv_hc[q]$ ⁸,

then there would be no need to schedule a receive job on p 's CPU at all. Since the purpose of this section is to demonstrate the applicability of our clock reading method, rather than to analyze the effect of scheduling policies, we choose this approach for ease of presentation. Note that this assumption is beneficial for the analysis in another respect as well: No admission control is needed, since a faulty processor cannot create jobs on the CPU of another processor just by sending messages. This also “improves” the maximum clock reading error Γ (cf. Theorem 1) by dropping all $\mu_{(0)}^+$ terms.

The algorithm itself is fault-tolerant in the sense that at most f faulty processors may behave arbitrarily, as long as n , the total number of processors, is greater than $3f$. Since there is no special source processor, we assume that the same drift bound ρ holds for all processors. We also assume broadcast-based communication in this section, which means that the algorithm will guarantee a maximum clock reading error bound Γ of $\varepsilon + \rho(\delta^- + \delta^+) + 4\rho\mu^+ - \nu$, with $\nu = 4\mu^+ \frac{\rho^2}{1+\rho}$. As all jobs in this algorithm send $n-1$ messages, we will abbreviate $\delta_{(n-1)}^-$, $\delta_{(n-1)}^+$ and $\mu_{(n-1)}^+$ with δ^- , δ^+ and μ^+ .

B.1 The Algorithm

Fig. 7 shows the pseudo-code of the algorithm of [17] in conjunction with our optimal clock estimation method. It includes a few optimizations specifically designed for large round durations:

- Processors do not broadcast constantly but rather start and stop the broadcasts within the hardware clock time interval $[T - pre, T + post]$, with $T = k \cdot round_len$ denoting the logical round switching time. Clearly, pre and $post$ must be chosen to satisfy conditions *C1* and *C2* outlined above. Note that this means that p must continue to broadcast its round k clock value even after it has already switched to round $k+1$ (at or shortly after hardware clock time T). Thus, in addition to k , a second variable bc_k is used to record the round number of the clock to be broadcast currently.
- Since messages do not contain round numbers, broadcasting rounds must not overlap, i.e., pre , $post$ and the round length must be chosen such that no round $k+1$ message arrives at a processor p that has not yet finished broadcasting round k messages. This also requires a sufficiently large round duration.
- As a positive consequence of these round length assumptions, only $adj[k]$ and $adj[k-1]$ need to be kept in memory, rather than the whole array of past adjustment values.

⁸ Since rcv_hc is only used to measure the “age” of a message, it is not necessary to use an adjusted clock value here (in contrast to $send_cv$, which contains a logical clock value).

```

Processor p
1  const round_len, pre, post
2  var rcv_hc [], send_cv [], adj []
3  var k, bc_k          /* local clock round number, current broadcast round number */
4
5  function current_hc(), initial_synchronization (), my_processor_id() /* built-in */
6  function cfn(my_processor_id, estimates) /* convergence function as specified in [17] */
7
8  function age(q) = current_hc() - rcv_hc[q]
9  function  $est^-(q) = send\_cv[q] + (1 - \rho) (\delta^- + age(q)/(1 + \rho))$ 
10 function  $est^+(q) = send\_cv[q] + (1 + \rho) (\delta^+ + \min\{\mu^+, age(q)/(1 - \rho)\})$ 
11
12 upon booting:
13   k ← 1; bc_k ← 1; adj[k] ← initial_synchronization ()
14   set “send now” timer for round_len - pre
15
16 upon receiving a clock value cv from processor q: /* done atomically by network controller */
17   if cv > send_cv[q]:
18     rcv_hc[q] ← current_hc(); send_cv[q] ← cv
19
20 upon receiving “send now”:
21   send (current_hc() + adj[bc_k]) to all
22
23   if (bc_k = k) and (current_hc + adj[k] ≥ k · round_len): /* start new round? */
24     k ← k + 1; adj[k] ← cfn(my_processor_id(),  $\forall q : (est^-(q) + est^+(q))/2$ )
25
26   if current_hc() + adj[bc_k] < k · round_len + post: /* continue or stop broadcasting */
27     set “send now” timer for current_hc() /* timer will arrive at end(current_job) */
28   else:
29     bc_k ← k /* prepare for next round’s broadcast */
30     set “send now” timer for (k · round_len - pre)

```

Fig. 7. Internal clock synchronization; [17] combined with optimal clock reading

Due to broadcasting, many jobs are already active around local time $T = k \cdot \text{round_len}$. Therefore, we do not designate a separate job for calling the convergence function, but rather squeeze this into one of the broadcast jobs at the right time (line 23 of the algorithm). This, however, means that the state transition might not occur exactly at clock time $k \cdot \text{round_len}$, but 0 to $2\mu^+$ real-time units later.⁹

B.2 Analysis

The precision analysis in [17] is based on a set of assumptions, which involve the following constants: Λ (maximum clock reading error margin¹⁰), r_{min} , r_{max} (lower and upper bound on the real-time round duration), and β (maximum real-time delay between the starting of a round at different processors).

Let t_p^k denote the real-time by which processor p starts round k , i.e., by which it performs the (atomic) state transition of line 24.

Theorem 4 (Theorem 1 of [17]). *Assume that the following conditions are satisfied for all correct processors and all rounds:*

- (A1) *Initially, all clocks are synchronized to within some bound π_I .*
- (A2) *The (real-time) length of a round is bounded by r_{min} and r_{max} , i.e., $r_{min} \leq t_p^k - t_p^{k-1} \leq r_{max}$, for all p and k .*
- (A3) *All processors start their rounds within β real-time units, i.e., $|t_p^k - t_q^k| \leq \beta$, for all p, q, k .*
- (A4) *Rounds do not overlap, i.e., $\beta \leq r_{min}$.*
- (A5) $\pi_I \geq 2\Lambda + 2\rho r_{max} + 2\rho\beta$.

Then, the algorithm of [17] guarantees that all logical clocks of correct processors p and q are synchronized to within a bound of $\pi = \pi_I + 2\rho r_{max}$, i.e.,

$$|(current_hc_p() + adj_p[k_p]) - (current_hc_q() + adj_q[k_q])| \leq \pi$$

for all global states g , with X_p denoting the value of the variable or function X on processor p in global state g . At the beginning of a round, i.e., at the first g such that all correct processors have switched to round k , the smaller precision π_I holds. Moreover, the maximum local clock correction $adj_p[k] - adj_p[k-1]$ is $\pm 2\rho r_{max}$.

Since the combined algorithm in Fig. 7 starts new clocks in the same way as [17], this theorem applies to our algorithm as well. As initial synchronization

⁹ The case of $2\mu^+$ occurs when $HC_q(\text{start}(J_1)) = k \cdot \text{round_len} - x$, for some very small x , and the round change state transition occurs in the following job J_2 at the very end of the job. Recall that $current_hc()$ always refers to the hardware clock time of the *beginning* of the job.

¹⁰ Note that Λ is $\Gamma/2$: We defined the remote estimation interval as $[est^-, est^+]$, with Γ bounding $est^+ - est^-$. By contrast, [17] defines a remote clock reading as a single value with a symmetric error of at most $\pm\Lambda$.

is outside the scope of this paper, we will just assume that conditions (A1) and (A5) hold. For simplicity, we assume that all processors start at real time 0 with their adjusted clocks ($current_hc() + adj[1]$) synchronized to within π_I (see line 13 of the algorithm).

The following lemma will show that there is a choice of r_{min} , r_{max} and β for which (A2)-(A4) are satisfied. The proof is slightly informal because it uses an “inductive” version of Theorem 4, which is not stated explicitly but can be deduced from the proofs of [17].

Lemma 11. *If $round_len$, r_{min} , r_{max} and β are chosen such that*

$$round_len \geq \pi + 2\mu^+(1 + \rho) + r_{min}(1 + \rho) \quad (7)$$

$$r_{max} \geq \frac{\pi + round_len}{1 - \rho} + 2\mu^+ + \beta \quad (8)$$

$$\beta \geq \frac{\pi + round_len}{1 - \rho} + 2\mu^+ - r_{min} \quad (9)$$

$$r_{min} \geq \beta \quad (10)$$

then conditions (A2)-(A4) are satisfied.

Proof. (A4) holds trivially, since it equals (10). Assume by induction that (A2) and (A3) hold for rounds 1 to k . By (an inductive version of) Theorem 4, this implies that clocks are synchronized to within π_I directly after the last processor switched to round k , and that their adjustment value changed by at most $2\rho r_{max}$ as compared to round $k - 1$. For the induction start, i.e., round 1, this is guaranteed by (A1).

Let q be the last processor switching to round k . Since this round switch was triggered by q 's round $k - 1$ clock reaching $(k - 1) \cdot round_len$, q 's state transition occurred 0 to $2\mu^+$ real-time units later, and q 's clock value was not changed by more than $2\rho r_{max}$, it follows that q 's new clock value at time t_q^k is within $(k - 1) \cdot round_len + [-2\rho r_{max}, 2\rho r_{max}] + [0, 2\mu^+(1 + \rho)]$. Recall that, at t_q^k , all other clocks are synchronized to within π_I with q . Thus, at t_q^k , it holds for the clock values cv_p of every correct clock p :

$$cv_p \in (k - 1) \cdot round_len + [-\pi, +\pi] + [0, 2\mu^+(1 + \rho)] \quad (11)$$

Using (7), we can bound the number of hardware clock units left until $k \cdot round_len$ is reached at p :

$$k \cdot round_len - cv_p \in [r_{min}(1 + \rho), \pi + round_len] \quad (12)$$

Note that the state transition to round $k + 1$ (at real time t_p^{k+1}) might occur at most $2\mu^+$ real-time units after reaching $k \cdot round_len$. Since (A3) holds for k and, thus, $t_p^k \in [t_q^k - \beta, t_q^k]$, these bounds together with (8) show condition (A2): $r_{min} \leq t_p^{k+1} - t_p^k \leq r_{max}$.

To show (A3), we follow a similar line of reasoning: (12) ensures that no processor can reach $k \cdot round_len$ earlier than $t_q^k + r_{min}$ in real-time, and that no

processor reaches $k \cdot \text{round_len}$ later than $t_q^k + \frac{\pi + \text{round_len}}{1 - \rho}$ in real-time. Adding the $2\mu^+$ that may lie in between reaching $k \cdot \text{round_len}$ and switching the round, (9) shows that (A3) holds: $|t_p^{k+1} - t_q^{k+1}| \leq \beta$, for all p and q . \square

We can hence apply Theorem 4 to immediately get:

Corollary 1 (Theorem 3). *For a sufficiently large round length and sufficiently close initial synchronization, the algorithm of Fig. 7 solves internal clock synchronization within $\pi = 2\Gamma + 4\rho r_{max} + 2\rho\beta$ with $\Gamma = \varepsilon + \rho(\delta^- + \delta^+) + 4\rho\mu^+ - \nu$.*

B.3 Improvements

Although a good worst-case bound has been shown for the algorithm of Fig. 7, the average-case performance can still be improved by a few simple modifications.

- The convergence function of [17] expects remote clock estimations with a symmetric error, i.e., for every remote clock p , the remote clock reading method returns some value $est(p)$, such that p 's clock lies within $[est(p) - \Lambda, est(p) + \Lambda]$. To achieve the optimal error margin of $\Lambda = \Gamma/2$, we return $est(p) = (est^+(p) + est^-(p))/2$, thereby ensuring that the clock of p is guaranteed to be within $[est(p) - \Gamma/2, est(p) + \Gamma/2]$. This method has an inconvenient side-effect, however: In a benign execution where clocks do not drift and all messages take $\delta^- + \varepsilon/2$ time units, one would expect $est(p)$ to match the real value of p 's clock. However, as shown in Sect. 3.1, this is not possible when a minimal symmetric error is needed. Therefore, adapting the convergence function of [17] to support asymmetric error margins might yield a better result in average executions.
- The algorithm does not yet exploit all information that is available; in particular, “round-trip information” is ignored. Assume that p sends a fast message m to q , and, shortly after m has been received, q sends a fast message m' back to p . If q also includes information about m in m' , p can deduce that m' must have been fast, thereby significantly improving its estimate of q 's clock value. Note that this idea is exploited in probabilistic clock synchronization [7].

References

1. Ramanathan, P., Shin, K.G., Butler, R.W.: Fault-tolerant clock synchronization in distributed systems. *IEEE Computer* **23**(10) (October 1990) 33–42
2. 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).
3. Schmid, U., ed.: Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems. *J. Real-Time Systems* 12(1–3) (1997)
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. In: Proceedings of the International Conference on Principles of Distributed Systems (OPODIS). LNCS 4305, Bordeaux & Saint-Emilion, France, Springer Verlag (Dec 2006) 95–109
6. Moser, H., Schmid, U.: Reconciling distributed computing models and real-time systems. In: Proceedings Work in Progress Session of the 27th IEEE Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil (Dec 2006) 73–76
7. Cristian, F.: Probabilistic clock synchronization. *Distributed Computing* **3**(3) (1989) 146–158
8. Arvind, K.: Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* **5**(5) (May 1994) 474–487
9. Ellingson, C., Kulpinski, R.: Dissemination of system time. *Communications, IEEE Transactions on [legacy, pre - 1988]* **21**(5) (May 1973) 605–624
10. Moon, S., Skelley, P., Towsley, D.: Estimation and removal of clock skew from network delay measurements. In: IEEE INFOCOM 1999. (March 1999)
11. Schneider, F.B.: A paradigm for reliable clock synchronization. In: Proceedings Advanced Seminar of Local Area Networks, Bandol, France (April 1986) 85–104
12. Fetzer, C., Cristian, F.: Lower bounds for function based clock synchronization. In: Proceedings 14th ACM Symposium on Principles of Distributed Computing, Ottawa, CA (August 1995)
13. Mavronicolas, M.: An upper and a lower bound for tick synchronization. In: Proceedings Real-Time Systems Symposium. (Dec 1992) 246–255
14. 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
15. Ostrovsky, R., Patt-Shamir, B.: Optimal and efficient clock synchronization under drifting clocks. In: PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1999) 3–12
16. Schmid, U., Schossmaier, K.: Interval-based clock synchronization. *Real-Time Systems* **12**(2) (March 1997) 173–228
17. Fetzer, C., Cristian, F.: An optimal internal clock synchronization algorithm. In: Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS'95). (1995) 187–196
18. Moser, H.: Towards a real-time distributed computing model. Research Report 17/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria (2007) To appear in: Theoretical Computer Science, Special Issue.
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7) (1978) 558–565
20. Moser, H., Schmid, U.: Optimal deterministic remote clock estimation in real-time systems. Research Report 43/2008, Vienna University of Technology, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria (2008)
21. Robinson, P., Schmid, U.: The Asynchronous Bounded-Cycle Model. In: Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'08), Detroit, USA (November 2007)