TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

DISSERTATION

A Model for Distributed Computing in Real-Time Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Anleitung von

Univ.Prof. Dr. Ulrich Schmid

Institut für Technische Informatik (E182)

eingereicht an der Technischen Universität Wien
Fakultät für Informatik
von

Heinrich Moser

Matr.Nr. 9925866
Pülslgasse 22/4
1230 Wien

Wien, am 11. Mai 2009

Kurzfassung

Diese Dissertation stellt ein neues, fehlertolerantes Modell für verteile Datenverarbeitung in Echtzeitsystemen vor, das sowohl die Perspektive der klassischen verteilten Datenverarbeitungsmodelle als auch die der Echtzeitsystemforschung berücksichtigt. Üblicherweise wird bei der Analyse von verteilten Algorithmen die vereinfachende Annahme getroffen, dass Rechenschritte in Nullzeit durchgeführt werden. Unser Modell basiert auf den bisher gängigen Modellierungstechniken verteilter System, lässt jedoch im Unterschied dazu die Nullzeitannahme fallen. Diese Vorgehensweise erlaubt größtmögliche Wiederverwendbarkeit existierender Ergebnisse, eröffnet jedoch auch die bisher verschlossene Möglichkeit, Scheduling-Analysen durchzuführen. Mit Hilfe der in dieser Arbeit vorgestellten Transformationsalgorithmen untersuchen wir die Beziehung zwischen dem klassischen synchronen Systemmodell und unserem Echtzeitmodell: Wir zeigen, wie Algorithmen von einem in das andere Modell übergeführt werden können und welche Eigenschaften echter Computersysteme durch die Nullzeitannahme bisher nur verfälscht wahrgenommen wurden.

Um diesen Unterschied anhand eines konkreten Beispiels zu demonstrieren, untersuchen wir das Problem der deterministischen Uhrensynchronisation in fehlerfreien Systemen mit perfekter Ganggenauigkeit und zeigen, dass – in unserem Echtzeitmodell – kein Algorithmus existieren kann, der optimale Synchronisationsgenauigkeit bei konstanter Laufzeit sicherstellt. Da jedoch ein solcher Algorithmus im klassischen Systemmodell bekannt ist, haben wir hier ein Beispiel, bei dem die klassische Analyse zu optimistische Ergebnisse liefert. Wir zeigen, dass das Erreichen optimaler Synchronisationsgenauigkeit einen Zeitaufwand von $\Omega(n)$ erfordert und präsentieren einen dazu passenden $\mathrm{O}(n)$-Algorithmus.

Allgemein gilt, dass bei diesem Synchronisationsproblem die Anzahl der Nachrichten, die von einem Algorithmus benötigt werden, in Abhängigkeit von der Synchronisationsgenauigkeit steht. Dieses Ergebnis führt uns einerseits zu der oben erwähnten Schranke von $\Omega(n)$ für optimale Genauigkeit, erlaubt jedoch auch Aussagen über den nicht-optimalen Fall: Es zeigt sich, dass nicht-optimale Synchronisationsgenauigkeit auch von einem Algorithmus mit konstanter Laufzeit erreicht werden kann, allerdings nur dann, wenn das darunterliegende Netzwerksystem Broadcasts in konstanter Zeit erlaubt.

Auch die Synchronisation von Uhren mit Gangabweichung wird in dieser Arbeit unter dem Echtzeitaspekt behandelt. Konkret untersuchen wir das Teilproblem, den aktuellen Wert einer auf einem anderen Computersystem befindlichen Uhr so genau wie möglich zu schätzen, präsentieren einen Algorithmus, der dieses Problem löst, und beweisen, dass keine bessere Schätzgenauigkeit erzielt werden kann. Abschließend zeigen wir, wie diese Schätzmethode mit einer optimalen Konvergenzfunktion in einem hochpräzisen, fehlertoleranten Uhrensynchronisationsalgorithmus kombiniert werden kann.

Abstract

This work introduces a fault-tolerant 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 schedulability analysis and retains compatibility with classic distributed computing analysis techniques and results. We provide general simulations and validity conditions for transforming algorithms from the classic synchronous model to our real-time model and vice versa, and investigate whether/which properties of real systems are inaccurately or even wrongly captured when resorting to zero step-time models.

We revisit the well-studied problem of deterministic drift- and failure-free internal clock synchronization for this purpose, and show that no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. Since such an algorithm is known for the classic model, this is an instance of a problem where the standard distributed computing analysis gives too optimistic results. We prove that optimal precision is only achievable with algorithms that take $\Omega(n)$ time in our model, and present a matching $\mathrm{O}(n)$ algorithm.

As a more general result, we provide a lower bound on the number of messages required to obtain a certain clock synchronization precision. In the case of optimal precision, this leads to the aforementioned bound of $\Omega(n)$. With respect to non-optimal precision equal to the message delay uncertainty, our result implies that constant time complexity is possible if, and only if, the system allows for constant-time broadcasts.

As 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. Moreover, we show how to combine our clock estimation method with an optimal convergence function, resulting in a high-precision fault-tolerant clock synchronization algorithm.

This work has been supported by the Austrian Science Foundation (FWF) under grants P17757 and P20529.

Contents

Part I. Computing Models

1. Introduction

The first part of this thesis, starting with Chapter 2, will present a novel framework for modeling executions of algorithms in distributed real-time systems. In the second part, starting with Chapter 7, the usefulness of this model is demonstrated by applying it to some well-known clock synchronization problems and comparing the results with those obtained by classic modeling techniques. Hence, some knowledge in the areas of distributed computing, real-time systems and clock synchronization is vital to the understanding of this thesis.

This introductory chapter shall familiarize the reader with the required basics of these lines of research, outlines the structure and the results of this thesis and compares them to similar approaches in dedicated *related work* sections.

1.1. Computing Models

A *distributed system* consists of a set of individual *processors* capable of exchanging information. The *distributed computing* line of research is concerned with the study of algorithms solving a given *problem* in such a distributed system, preferably in an efficient manner.

Distributed State Machines

Obviously, lots of abstraction and simplification are necessary to reduce an arrangement of CPUs, network controllers and transmission media to a mathematical model that allows to reason about generic results in distributed computing.

Processors (CPUs, or computers in general) are represented by *state machines*: Given a set of variable names (e.g. $\{participants, free\_places\}$), a *state* can be seen as values assigned to these variables; for example, $\{participants = \{\text{Peter}, \text{Martin}\}, free\_places = 3\}$ would be a valid state. The processor can perform *state transitions*, i.e., it changes from one state to another. In general, the processor cannot make arbitrary state transitions, but rather runs some kind of *algorithm*. The algorithm specifies the initial state (e.g. $\{participants = \{\}, free\_places = 5\}$) and the *state-transition function*, which can be seen as a set of rules describing which state transitions are allowed, formalized as a function mapping a state and, possibly, some kind of *input*, to a new state and, possibly, also to some kind of *output*. For example, a state transition function representing the code

```
**upon** receiving "subscribe $name$":
    **if** $free\_places > 0$ and $name \notin participants$:
        $free\_places \leftarrow free\_places - 1$
        $participants \leftarrow participants \cup \{name\}$
        send "subscription confirmed"
```

would map ($\{participants = \{\text{Daniel}\}, free\_places = 4\}$, “subscribe Josef”) to the new state $\{participants = \{\text{Daniel}, \text{Josef}\}, free\_places = 3\}$ and to the output “subscription confirmed”.

A *distributed system* consists of a set of such processors and some means of communication. Thus, the distributed system also specifies how processors receive their input and what happens to their output. Common examples include

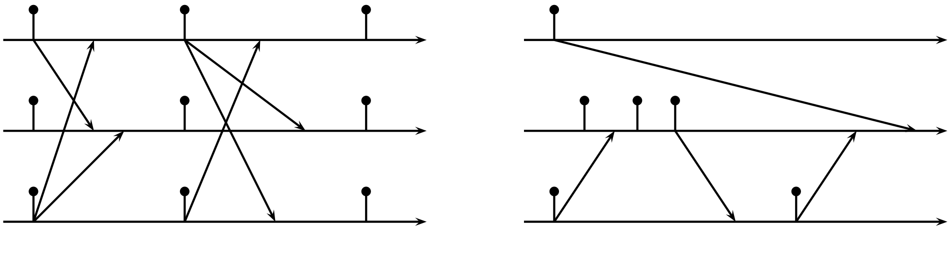
- *a message bus*: Once a processor outputs some message, all other processors receive this message as input some time later.
- *shared memory*: All processors can read and modify a common pool of registers; changes by one processor can be seen by all other processors immediately.
- *point-to-point communication*: Some processors are connected through communication links. The output of the transition function consists of messages sent over specific links, e.g., “send message A to processor p and message B to processor q”.

In this thesis, we will restrict our attention to message-based point-to-point communication in fully-connected networks. *Fully-connected* means that every processor can send a message to every other processor.

An *execution* or *run* of an algorithm in such a distributed system can be visualized as a *space-time diagram*, such as the ones found in Figures 1.1 and 1.2. A horizontal line represents a processor and arrows symbolize messages being sent and delivered. With respect to state transitions (also known as *computing steps*), “ticks” stand for atomic, immediate state transitions at this exact time, whereas boxes (e.g. in Fig. 1.2(c)) more generally specify that some code is being executed, with the exact times of state transitions unknown. Note that messages are always sent during a computing step.

When processors deviate from their specified behavior, they are considered *faulty*. The following list contains a few well-known examples of ways in which a processor can fail:

- *Clean crash*: A processor eventually stops working, i.e., after some point in time, no more computing steps occur on this processor.
- *Unclean crash*: A processor crashes in the middle of a computing step, i.e., at some point in time, the processor executes a *part* of a computing step and then stops working completely. This case is more difficult to tolerate, since a processor might only send a subset of its outgoing messages while crashing—contrary to the clean crash case, where a computing step is either completed in full or does not occur at all.
- *Byzantine* or *arbitrary* faults: A Byzantine faulty processor can behave arbitrarily; its computing steps do not have to conform to the algorithm’s state transition function. For example, it can send out messages with arbitrary, misleading information, manipulate messages while forwarding them or just stop working like in the crash failure case. This failure model applies to “broken” processors sending out nonsensical data as well as to malicious processors deliberately trying to prevent the distributed algorithm from reaching its goal.



(a) Lock-step Rounds

(b) Asynchronous

Figure 1.1.: Lock-step Synchronous and Asynchronous Model

A *fault-tolerant algorithm* is an algorithm which solves a given problem despite a certain number of processors being faulty.

Classic Distributed Computing Models

Since the algorithm specifies the state transitions the processor may carry out, the question remains as to *when* these transitions are performed. Together with timing bounds on message transmission, the answer to this question determines the *synchrony* of the computing model. To understand the wide variety of models available, let us look at the two extreme cases [Lyn96, AW04]:

- The *synchronous lock-step model* (Figure 1.1(a)) splits the execution into *rounds*: In every round, (1) one message can be sent from every processor to every neighbor, (2) all these messages arrive, and (3) each processor makes a state transition using the received messages as input. Note that, in the figure, the same computing step performs the state transition for round k as well as the message sending for round $(k + 1)$.
- At the other end of the spectrum, there is the *asynchronous model* (Figure 1.1(b)), which does not bound (neither relative nor absolute) processing speeds or message transmission times in any way. The only restriction is that every message eventually gets delivered and every processor takes an infinite number of computing steps.

Both models have their advantages and their drawbacks: The strong assumptions of the synchronous lock-step model make it easy to design algorithms: The developer can be sure that during some processor's round k computing step, all other processors have already performed their round $(k - 1)$ steps and all round $(k - 1)$ messages have already arrived. However, there are lots of real-world systems where these assumptions just do not hold. The assumptions of the asynchronous system, on the other hand, are so weak that they are easily satisfied by real-world computer systems. At the downside, many well-known problems are very hard or even impossible to solve in the asynchronous model [FLP85].

There is a wide range of *partially synchronous* models, which attempt to find a compromise between these two extreme cases: They bound, for example, the relative processing speeds of processors (that is, between two computing steps of one processor, another processor may not perform more than x computing steps) or the message transmission delay (that is, no message transmission may take less than y or more than z time units). A few examples from literature will be outlined in the *related work* section below.

However, these models still suffer from one common problem, which we will illustrate by comparing a real-world execution with its simplified representation in one of the “classic” distributed computing models. Figure 1.2(a) gives an (already considerably simplified) glimpse of what happens in a real-world system. Consider the second message arriving at processor p: This message triggers the scheduler, requiring some CPU overhead (= the gray box) for switching to the correct task responsible for processing that message. This task starts a job (= the white box), which performs some computations and eventually sends out a few messages via some communication medium. This sending causes some media access control overhead (= the MAC box), which might very well occur in parallel to the CPU, if MAC is performed by a separate network controller. The first message sent by that job (= m_3) eventually arrives at processor q, which is currently busy with processing m_1; thus, m_3 is enqueued (= the coiled arrow) until the scheduler of q decides that the CPU is ready to process it.

Figure 1.2(b) shows the same situation, using the *message-driven synchronous (non-lock-step) system model*: The execution of the distributed algorithm is represented by a sequence of atomic computing steps that are triggered by an incoming message and executed in zero time. All timing-related factors that occur in the real-world system, such as scheduling overhead, processing time, queuing time or network delays are encapsulated in the *end-to-end delay*, i.e., in the time between the zero-time action sending a message and the one receiving it. With respect to timing assumptions, the synchronous model assumes that these end-to-end delays are bounded: There is a constant lower bound ($\underline{\delta}^-$) as well as a constant upper bound ($\underline{\delta}^+$).

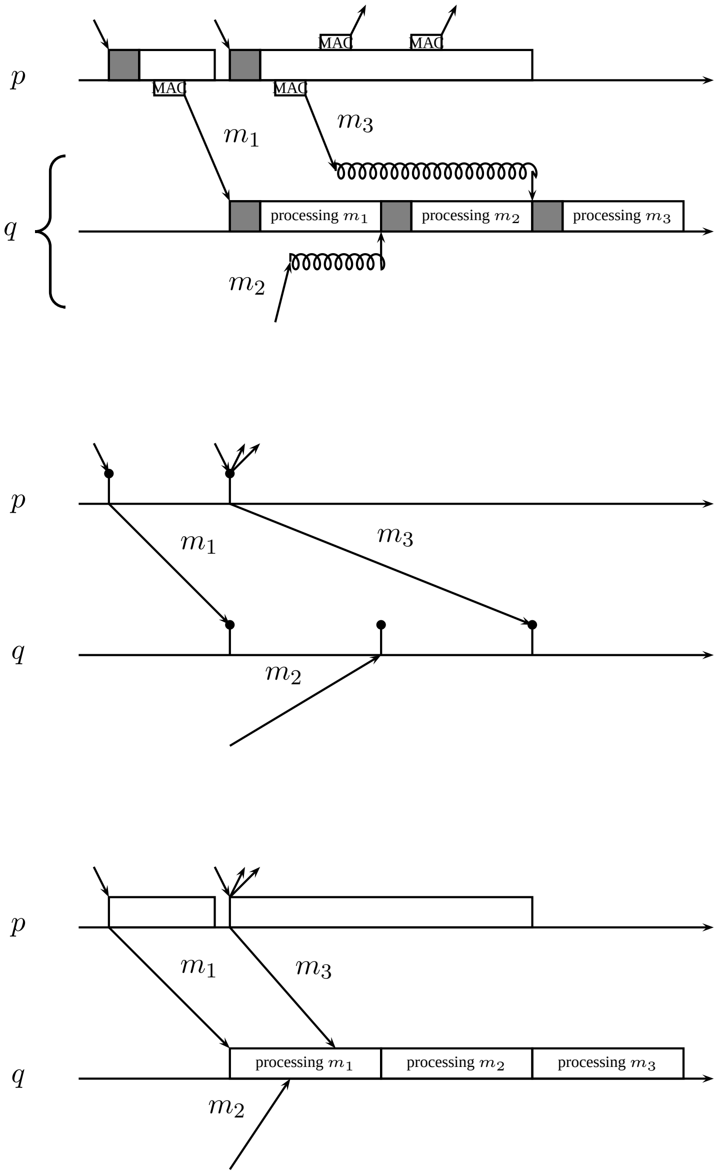
With this assumption, it does not make a difference, for example, whether messages arrive at a processor simultaneously or nicely staggered in time: the messages are processed instantaneously when they arrive. The zero step-time abstraction is, hence, very convenient for analysis, and a wealth of distributed algorithms, impossibility results and lower bounds have been developed for models that employ this assumption [Lyn96].

Scheduling Theory

In real systems, however, computing steps are neither instantaneous nor arbitrarily preemptable: A computing step triggered by a message arriving in the middle of the execution of some other computing step is usually delayed until the current computation is finished. This results in queuing phenomenons, which depend not only on the actual message arrival pattern but also on the queuing/scheduling discipline employed.

Traditionally, such problems are the central topic of a separate line of research, devoted to real-time systems [But97]. Let a *job* be defined as a small unit of work to be performed by a CPU, characterized by (at least)

- an *arrival time*, the time by which this job is ready for execution,
- a *computation time* (also known as *worst-case execution time*), the time span required to execute this job on the CPU, and
- a *deadline*, the latest time by which this job must have finished.



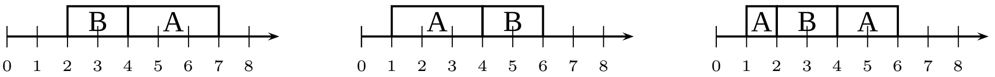
(a) “Real” event-driven execution

(b) Classic distributed computing abstraction (*executions*)

(c) Our real-time computing model (*rt-runs*)

Figure 1.2.: Modeling distributed system executions

The goal of *uniprocessor real-time scheduling* is to create a *schedule*, which assigns these jobs to a processor in a way such that all jobs meet their deadlines. For example, consider the job set $\{(A : 1, 3, 7); (B : 2, 2, 5)\}$, with the numbers denoting the arrival time, the computation time and the deadline, respectively. Figure 1.3 shows (a) a feasible schedule, in which both jobs meet their deadlines, and (b) an infeasible schedule, in which job B misses its deadline. If the jobs cannot be interrupted, it is plain to see that (a) is the only solution. Otherwise, if *preemption* is permitted, other assignments, such as (c), are possible. More advanced examples include *precedence relations* among jobs, e.g., some job A must be (completely) executed before job B may start, or *shared resources*, e.g., A and B may require exclusive access to the same resource, possibly causing schedules such as the one in Figure 1.3(c) to become infeasible.



(a) Feasible schedule (b) Infeasible schedule (c) Feasible preemptive schedule

Figure 1.3.: Real-time schedules

Usually, jobs are not declared individually but rather summarized into *tasks*: A task consists of an infinite sequence of jobs, all having the same execution time and relative deadline (= the time span between (absolute) arrival and (absolute) deadline). The arrival time of the jobs is specified by the *arrival pattern* of the task: The jobs of a *periodic* task arrive at regular, constant intervals; *sporadic* tasks release their jobs irregularly, but with some minimum time (*sporadicity interval*) in between job arrivals; and *aperiodic* tasks do not specify any arrival restriction.

Determining whether a feasible schedule exists or not can be surprisingly difficult: For example, the general *feasibility analysis* problem is NP-hard, even in the case of only periodic task sets (with arbitrary deadlines) [BHR90]. Thus, many analysis techniques are pessimistic (sufficient but not necessary) and/or only apply to special cases (such as relative deadlines being equal to the period) [SAA$^+$04]. *Worst-case response time analysis* is a generalization of feasibility analysis: Rather than finding out whether all jobs meet the a-priori given deadline, the worst-case difference between the arrival of some particular job and its completion is determined.

Multiprocessor real-time scheduling is a natural generalization: Instead of only one CPU, jobs can be scheduled to multiple processing units in parallel. Although multiprocessor scheduling is not yet as well understood as the uniprocessor case, a lot of results and heuristics do exist (again, cf. [SAA$^+$04] for an overview).

Unfortunately, real-time scheduling in loosely-coupled distributed systems, i.e., scheduling involving multiple processors connected through a network, has only been addressed for very restricted types of problems: Informally speaking, the concept of tasks with a-priori known arrival patterns has been extended to *information flows*, starting at some processor (again, with some known arrival pattern) and then traveling through the network in a linear fashion. For example, some external signal could arrive at processor p, causing a job to be executed, then travel to processor q over a communication link, again causing a job execution, and finally arriving at processor r, initiating the last job. *Holistic scheduling* [TC94] or the *trajectory approach* [MMG04] allow to determine the overall worst-case response time of such a flow.

Nevertheless, many problems in distributed computing do not exhibit such a linear flow of information, starting with some sensor and ending in some actuator or data aggregation node. In particular, fault-tolerance often requires a set of "equal" processors exchanging messages on a regular basis (*rounds*)—well-known examples are distributed agreement problems, such as Byzantine consensus or non-blocking atomic commitment, and synchronization problems, such as tick synchronization or clock synchronization. In these cases, the arrival patterns of messages might not be known in advance; often, there is even a circular dependency: The arrival pattern of some round k messages determines the queuing delays, which in turn influence the time at which round $(k + 1)$ messages are sent, and thus, the arrival pattern of these messages. Although the approaches mentioned above are a promising start for extending schedulability analysis to loosely-coupled distributed systems, so far, no suitable generic modeling framework for analyzing queuing effects of distributed algorithms exists.

Related Work

We are not aware of much existing work in the distributed computing area of research that also addresses real-time aspects. Somewhat an exception is the work by Neiger & Toueg [NT93], which identifies general problems and conditions that preserve the correctness of a solution based on perfectly synchronized clocks when logical clocks are used instead. The underlying model assumes non-zero step times, but considers them sufficiently small to completely ignore queuing effects. Moreover, in contrast to our work, they restrict their attention only to problems whose specification can be written in a way which does not refer to real time. Another example of a non-zero step time model is the remote memory reference (RMR) model for shared-memory systems [AY96, AKH03] by Anderson et. al. It assumes computing step times which depend on the number of conflicting shared memory accesses. The RMR model has been used for deriving several algorithms, e.g. for mutual exclusion, and related lower bounds. Since it is not applicable to message-passing systems, however, our results are not comparable.

Another branch of research where distributed computing and real-time systems issues are combined are modeling frameworks [AD94, LV95, LV96, MMT91, SGSAL98, KLSV03]. Such frameworks allow formal modeling and analysis of complex distributed real-time systems. A representative example are Timed I/O Automata (TIOA) [KLSV03], which can change state both via ordinary discrete transitions and via continuous trajectories. TIOAs facilitate hierarchical composition, abstraction, and proofs of safety and liveness properties. However, none of the above modeling frameworks supports non-zero step times and thus real-time schedulability analysis of distributed algorithms. By contrast, our work addresses exactly this issue.

Apart from those lines of research, we are not aware of too many distributed computing papers that incorporate real-time scheduling issues at all: In [HLL02], for example, Hermant and Le Lann demonstrate the power of such an integrated approach by introducing fast failure detectors, which facilitate very fast detection times and thus quickly terminating asynchronous consensus algorithms.

Turning our attention to distributed system models, the partially synchronous DLS model [DDS87, DLS88] extends the asynchronous model by adding an absolute upper bound Δ on message transmission delays and an upper bound Φ on the relative computing speed of any two processors. Formally, it is assumed that every processor takes at least one and at most Φ computing steps during any real-time interval of duration Φ; note that the unit of real-time is actually the computing step time of the fastest processor in the system here. A send step sends one message to one recipient, a receive step makes some messages that arrived so far available to the processor. A single computing step can be either a send step or a receive step, but not both. Hence, receiving a message and sending a response is not possible in zero time here. Additionally, sender queuing (but not receiver queuing) is modeled explicitly by allowing only one message per send step.

The semi-synchronous model [ADLS94, Mav92, PS92] uses a similar approach; however, there is no need for separate send and receive steps, and computing steps can send out multiple messages at once. [ADLS94] does not normalize the real time units to the speed of the fastest processor, i.e., the real time between two consecutive steps of one processor is within some fixed interval $[c_1, c_2]$. [Mav92] assumes that $c_2 = 1$, i.e., the slowest processor determines the time base.

Both the DLS model and the semi-synchronous model conveniently abstract away queuing effects at the receiving processor, since a single receive event (DLS model) or a single computing step (semi-synchronous model) can process all messages received so far. As Chapter 7 of this thesis will show, this issue can make an important difference. Moreover, both models lack a lower bound on the message transmission delay. In the context of clock synchronization, this is an important difference to our model: In the case of drifting clocks, not only the message delay uncertainty but also the absolute bounds on message transmissions affect the achievable precision (cf. Chapter 8).

The partially synchronous Theta model [LLS03, WLLS05, HW05a] as well as the Asynchronous Bounded-Cycle model (ABC model) [RS08] are mentioned here, since they provide an additional motivation for our line of research. Both models are based on the assumption that, due to the dynamic nature of distributed algorithms and the corresponding queuing effects, it is not feasible to assume fixed a-priori bounds on the end-to-end delay of messages. They retain the zero step-time assumption but bound *ratios*, either on the end-to-end delay of messages simultaneously in transit (Theta model) or on the number of forward and backward edges in certain message chain cycles (ABC model). Clearly, such assumptions only hold if there is a strong relationship between the queuing delays in different parts of the system, either at the same physical instant of time (in case of the Theta model) or within a “causally related” region of the space-time diagram (in case of the ABC model). Due to lack of appropriate analysis techniques, the claim that this relationship holds in real systems has only been verified experimentally [Alb05]. We believe that the real-time distributed computing model presented in this thesis is a mandatory prerequisite for any attempt to verify this assumption analytically.

There are also a few approaches in the real-time systems community that aim at an integrated schedulability analysis in distributed systems. One notable example combining local processor scheduling with network communication issues is *holistic scheduling*, introduced in [TC94]: Assuming synchronized clocks, end-to-end delay bounds of “information flows” across a shared network bus can be derived by means of recurrence relations based on the (a-priori known) bounds on the worst-case frequency of all tasks and messages. In [PGGGGH98], this work is extended by adding a best-case analysis, improving the accuracy of the estimated jitter. However, contrary to the modeling requirements of many distributed algorithms, they assume a strictly linear sequence of actions in the system, i.e., within one flow, every computing action on a processor sends a message to at most one other destination processor. The *trajectory approach* [MMG04] provides bounds on the end-to-end delays which are less pessimistic; however, they also model a flow as a fixed, linear path through the network.

The Gap

This brief overview demonstrates that the distributed computing view on system models on one hand and the scheduling results obtained by the real-time community on the other hand operate at entirely different levels of abstraction and solve different problems: Whereas the former is concerned with the correctness of algorithms, usually expressed by some predicate on the internal state of the processors involved, the latter is only interested in the ability of an algorithm to meet some a-priori defined deadlines.

However, as we show in the second part of this thesis by means of the clock synchronization problem, queuing issues *can be* relevant for the correctness of an algorithm and/or the tightness of a lower bound. Bridging this gap and analyzing these effects requires a model which,

- on the one hand, is “compatible” with the classic distributed computing models, such that the wealth of existing results can be reused, but,
- on the other hand, explicitly models queuing effects, thereby allowing us to incorporate real-time scheduling issues and to perform a worst-case response time analysis.

Consequently, the first part of this thesis 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.

Consider the example in Figure 1.2(c): Introducing the processing delay as an additional system parameter allows us to split the end-to-end delay into

- the message delay (= the arrow),
- the queuing delay (= the distance between the arrow head and the start of the corresponding box), and
- the processing delay (= the box).

This model hence enables us to demonstrate the impact of queuing effects on distributed algorithms (in particular, on clock synchronization), while keeping most of the mathematical simplicity of the classic distributed computing abstraction (Figure 1.2(b)).

Thus, in the resulting real-time computing model, a system can be specified by bounding the message and the processing delay; the queuing delay, however, is not a system parameter here but rather depends dynamically on the message pattern of the algorithm and the *scheduling policy*, i.e., on the order, in which queued messages are processed. In fact, this is a major advantage of our approach: In the classic computing model, comparing algorithms running in “the same system”, i.e., with the same bounds on the end-to-end delay, can be misleading, since it ignores the fact that the message pattern of the algorithm itself influences the queuing delays, and, thus, the end-to-end delays.

1.2. Clock Synchronization

Apart from making distributed algorithms amenable to real-time analysis, our model also allows us 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 the second part of this thesis, we revisit the well-studied problem of deterministic clock synchronization for this purpose.

The term *clock synchronization* spans a wide range of distributed computing challenges. All of these try to *synchronize* actions on processors in a distributed systems, since added synchrony allows a lot of distributed computing problems to be solved in a much simpler way. Clock synchronization does not necessarily require real clocks: For example, Lamport clocks [Lam78] or vector clocks [Fid88, Mat88] use integer values, messages tagged with these values and a simple maximum function to obtain an ordering of events in an execution, which is applicable even in completely asynchronous systems. If there is some degree of synchrony already present in the system, more sophisticated *tick synchronization* algorithms [ST87, Mav92] can be used, which cause the processors in a distributed system to increment their counter variables “as simultaneously as possible”.

In this thesis, however, we are mainly interested in “classic” clock synchronization, in systems where the processors are equipped with local, possibly *imperfect* hardware clocks. Imperfect clocks exhibit some kind of *drift*, meaning that they do not run at the same rate as real time but rather a bit slower or a bit faster. The goal is to adjust the local clock values of different processors to satisfy some clock synchronization condition. In particular, *external clock synchronization* is concerned with minimizing the difference between the adjusted local clocks and some external reference clock such as GPS time, at all points in time. In contrast, *internal clock synchronization* does not assume access to an external time source but rather aims at minimizing the difference between the adjusted local clocks of any pair of processors (for all possible pairs, at all points in real time).[1] These adjusted clocks must satisfy some *progress* condition, usually requiring the clocks to stay within a linear envelope of real time.[2]

[1] To ease presentation, we will use the well-established term *precision* for this difference, although we are aware that “imprecision” would be more appropriate, technically. Thus, we will try to avoid misleading phrases such as “high precision” (usually referring to a small difference) and “low precision” (usually referring to a large difference) in the formal parts of this work and use unambiguous terminology instead.

[2] Interestingly, only requiring the adjusted clocks to increase without bound is not sufficient, since it allows for impractical solutions such as logarithmically increasing clocks (Theorem 1 of [DHS86]).

Like many other works in this area of research, much of this thesis builds on the foundations laid by Lundelius and Lynch [LL84b]. They established a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ on the achievable clock synchronization precision in a drift- and fault-free scenario, with $\underline{\varepsilon}$ denoting the end-to-end delay uncertainty (i.e., the difference between maximum and minimum end-to-end delay) and n the number of processors. Their work, which assumes a fully-connected network in which every processor can directly communicate with every other processor, was later generalized to arbitrary networks [HMM85], with [BW01] providing closed-form upper and lower bounds for some specific topologies. In Chapter 7, we will examine the problem of drift- and fault-free internal clock synchronization in a fully-connected network in our real-time distributed computing model.

With respect to fault-tolerance, it should be noted that internal clock synchronization is unsolvable if at least one third of the processors is Byzantine faulty [DHS86]; recall that a Byzantine faulty processor can behave arbitrarily, in particular, it may lie about its current clock value. In general, optimal-precision clock synchronization with drifting clocks is an open problem even in classic distributed computing: Optimal results are only available in case of a-priori given message patterns (“passive” clock synchronization) [PSR94, OPS99]; unfortunately, optimal message patterns and hence optimal “active” clock synchronization algorithms cannot be inferred from this research.

Interestingly, existing drift- and fault-tolerant internal clock synchronization algorithms can be reviewed in terms of a generic structure [Sch86]: Periodically, the algorithm detects the need for resynchronization. Then, information is gathered about the clock values of the other processors, usually by exchanging messages. After this data has been collected, a new local clock value is calculated based on some *convergence function*. To our surprise, the second step—a problem known as *remote clock estimation*—had not been solved optimally yet. In Chapter 8, we present a tight bound on the achievable estimation error, again using our real-time computing model.

For calculating a new clock value based on the remote clock estimations, an optimal convergence function using *fault-tolerant averaging* has been presented by Fetzer and Christian [FC95a, FC95b]. In a fully-connected system with n processors, f of which can be faulty, the main idea of fault-tolerant averaging is, for each processor p, to take all n clock readings (as estimated by p), discard the f largest and the f smallest values, and then calculate the arithmetic mean of the remaining interval [WL88]. The *differential fault-tolerant midpoint function* of [FC95b] extends this approach by slightly modifying this interval and bounding the maximum correction value, resulting in an optimal convergence function. Chapter 9 of this thesis provides a sample implementation that combines their convergence function with our clock reading method in one algorithm.

Related Work

Lots of results already exist for external as well as internal clock synchronization in classic distributed computing models. A comprehensive summary would go far beyond the scope of this work; [SLWL90] provides an exhaustive overview of fundamental results in this area of research, whereas [AP98] classifies existing clock synchronization algorithms according to their internal structure. A special issue of the Real-Time Systems journal assembles results with a focus on global time in large scale distributed real-time systems [Sch97]. Many recent discoveries can be found in the area of clock synchronization in wireless sensor networks [SBK05].

The main results of the second part of this thesis are lower and upper bounds on clock synchronization (or subproblems thereof) while considering the need to schedule message processing steps. Actually, this is a known problem: In [WL88], for example, Welch and Lynch mention that implementing their clock synchronization algorithm in a real-world setting required staggering the broadcast of messages (which would otherwise be sent almost at the same time), to avoid the situation that too many messages arrive simultaneously at the same processor. Still, this topic has not received much attention in literature. In fact, we are aware of only two papers that consider deterministic clock synchronization in connection with real-time scheduling:

- Basu and Punnekkat [BP03] propose simple variants of Srikanth & Toueg's tick synchronization algorithm [ST87]. Their algorithms stop the local clock while the resynchronization is in progress, thereby avoiding the problems that usually occur when the clocks being synchronized are also used to schedule tasks in heavily loaded real-time systems.

- Mavronicolas [Mav92] provides a lower bound for the precision achievable in a "single-shot" version of tick synchronization in the semi-synchronous model. This model differs in various significant ways from the one used in this thesis: Computing step duration and clock drift are directly related, the lower bound on the message transmission delay is always zero, and, like in the partially synchronous model of [DLS88], a single computing step can process multiple incoming messages. As outlined in the previous section, this last point conveniently abstracts away queuing issues; however, as Chapter 7 of this thesis will show, this issue can make an important difference. Thus, our results are not directly comparable.

To the best of our knowledge, other papers dealing with clock synchronization in real-time systems do not incorporate queuing issues at all [MFNT00], assume a-priori given bounds on the receiver queue length [VRC97], circumvent this problem by delegating the task of timestamping or processing messages to special-purpose low-level hardware [KO87, SL96, SR87], or restrict the precision analysis to empirical evaluations [ZSSZ08].

The subproblem of remote clock estimation is handled/analyzed sub-optimally or abstracted away entirely in the wealth of existing research on clock synchronization: Most papers employ trivial clock estimation algorithms only, based on a one-way or round-trip time-transfer via messages [EK73], and provide a fairly coarse analysis that (at best) incorporates clock drift [Cri89] and clock granularity [SS97]. Alternatively, as in [FC95a, FC95b], 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.

Since we are aiming at deterministic algorithms here, we do not consider probabilistic clock synchronization [Cri89, Arv94], statistically optimal estimations [EK73, MST99] and similar topics. Likewise, gradient clock synchronization [FL04, LLW08], which analyzes the effect of the network graph diameter on the synchronization precision between neighboring nodes, is out of the scope of our work.

Clock Synchronization in the Real-Time Distributed Computing Model

As it turns out, clock synchronization is a particularly suitable choice for analysis under our real-time distributed computing model, since the achievable synchronization precision is known to depend on the end-to-end delay uncertainty. Since non-zero computing step times are likely to affect end-to-end delays, one may expect that some results obtained under the classic model do not hold under the real-time model—if there are such effects at all.

Our analysis confirms that this is indeed the case: We show that even in the drift-free case no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model. Since such an algorithm has been given for the classic model [LL84b], this is an instance of a problem where the standard distributed computing analysis gives too optimistic results. Actually, we show that optimal precision is only achievable with algorithms that take $\Omega(n)$ time, even if they are provided with a constant-time broadcast primitive.

Obviously, clock synchronization in the drift-free case is a one-shot problem: After the clocks have been synchronized to a certain precision, they stay synchronized forever. Thus, time complexity does not really matter. However, since clock drift is practically unavoidable, the situation changes when generalizing these algorithms to real systems. In the extreme case of clocks with very high drift rates, a sub-optimal algorithm with low time complexity might perform better than a drift-free-optimal algorithm with high time complexity, since the former algorithm can be executed more frequently.

Contrary to the drift-free case, we do not solve the problem of internal clock synchronization with drifting clocks in real-time systems conclusively in this work—recall that this is even an open problem in the (probably easier) case of classic zero step-time distributed computing models. However, as a first step towards this goal, we examine the (deceptively simple) subproblem of *remote clock estimation* in the real-time computing model. Our results, consisting of an algorithm and a matching lower bound, precisely quantify the effect of system parameters such as clock drift, message delay uncertainty and step duration on optimal remote clock estimation.

Based on the classic round-based resynchronization scheme, where processors with drifting clocks initiate a clock synchronization protocol every time their clocks reach a multiple of some predefined time span, denoted the *resynchronization period*, we combine our remote clock reading method with the optimal convergence function of [FC95b] and determine an upper bound on the precision achievable with this algorithm.

1.3. Roadmap

This section explains the structure of this thesis and gives a brief overview of each chapter.[3]

[3] Preliminary versions of this work have been published in [MS06a], [MS06b], [MS08] and [Mos09].

Computing Models

In Chapter 2, we formalize the *classic computing model* ($\underline{\mathcal{M}}$), based on the well-known synchronous non-lock-step computing model for message-passing systems (both point-to-point and broadcast-based).

In Chapter 3, we define our *real-time computing model* ($\mathcal{M}$), which differs from $\underline{\mathcal{M}}$ by just providing atomic computing steps of non-zero duration. Consequences of this change, such as the need for scheduling, queuing and admission control, are discussed.

In Chapter 4, we introduce the concept of *failure models*. This ensures a strict separation between (a) generic validity conditions that every instance of $\underline{\mathcal{M}}$ or $\mathcal{M}$ has to satisfy, and (b) specific failure models, which determine to what extent the system has to adhere to its specification.

In Chapter 5, we discuss the challenges of designing a formal notation for *distributed computing problems* and present, as a solution, a framework for explicitly modeling state transitions in the real-time computing model.

In Chapter 6, we analyze the relationship between the classic and the real-time computing model by providing *transformations* in both directions: We show that a system adhering to some particular instance of $\mathcal{M}$ can simulate a system that adheres to some particular instance of $\underline{\mathcal{M}}$ (and vice versa). Consequently, certain distributed algorithms designed for a classic computing model can be run under the real-time computing model, for example.

Clock Synchronization

In Chapter 7, we revisit *deterministic internal clock synchronization* in synchronous systems, in the absence of failures and clock drift. It is known that the local clocks of n fully-connected processors cannot be synchronized with precision less than $(1 - 1/n)\underline{\varepsilon}$ when using messages with end-to-end delay uncertainty $\underline{\varepsilon}$. A constant time algorithm achieving this bound in the classic computing model also exists [LL84b].

We show that this is not true in the real-time computing model: optimal precision is only achievable with algorithms that take $\Omega(n)$ time. On the other hand, achieving a sub-optimal precision of $\mathrm{O}(\underline{\varepsilon})$ is achievable in constant time, if, and only if, a constant-time broadcast primitive is available.

In Chapter 8, we provide an optimal solution for the problem of how to continuously estimate a source processor's clock in the case of drifting clocks. The *remote clock estimation* algorithm is complemented by a matching lower bound on the achievable maximum clock reading error.

In Chapter 9, we give examples of how to apply this remote clock reading technique in external as well as fault-tolerant internal clock synchronization.

In Chapter 10, we summarize our results and conclude with an outlook on further work and open issues.

Part I.

Computing Models

2. Classic Distributed Computing

In clock synchronization research [LL84a, BW01, PSR94, AHR93, LL84b], system models are considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted “Partially Synchronous Reliable/Unreliable Models” in [SLWL90], such models are nowadays called (non-lock-step) 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 zero step-time synchronous model commonly used in clock synchronization research, e.g., in [LL84b]. Here it will be referred to as the *classic computing model*.

Note that the model described in this chapter is *message-driven*, i.e., computing steps are always triggered by messages [HW05b, BW06]. We do, however, retain compatibility to time-driven models through the concept of *timer messages*, which always arrive when the hardware clock reaches a certain value.

2.1. Preliminaries

Let seq be a sequence whose elements are totally ordered by some relation $\prec^{seq}$. Within this work, the notion of *causal dependency* will be used for various elements (actions, jobs, receive events, drop events, aj-events, st-events) of such a sequence. Every such element x has an associated processor $proc(x)$. There can be two types of dependencies between these elements (cf. *happened before* relation, [Lam78]).

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

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

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

Definition 2.1. Some sequence *captures message causality* if the ordering of its elements ($\prec^{seq}$) is consistent with the message dependency relation, formally: $\forall x, x' \in seq : x \xrightarrow{M} x' \Rightarrow x \prec^{seq} x'$.

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

Observation 2.2. *If seq captures message causality, seq' is a reordering of seq and seq' is causally consistent with seq, then seq is also causally consistent with seq'.*

2.2. System Model

We consider a network of n *processors* Π, which communicate by passing unique[1] messages, using either a unicast, multicast or broadcast primitive. The system-wide set of messages in transit will be denoted $intransit\_msgs$. Each processor p is equipped with a CPU, some local memory, a hardware clock HC_p, and reliable, non-FIFO links to all other processors. The hardware clock $HC_p : \mathbb{R}^+ \to \mathbb{R}^+$ maps dense real-time[2] to dense clock-time; it can be read but not changed by its processor. HC_p is hence not part of the local state $state_p$, but considered separately.

The CPU is running an *algorithm* $\underline{\mathcal{A}}$, which is specified as (a) a mapping from processor indices to a set of initial states and (b) a transition function. Processor p's set of *initial states* is denoted $Init_p(\underline{\mathcal{A}})$. The *transition function* takes the processor index p, one incoming message (taken from the current $intransit\_msgs$), receiver processor p's current local state $oldstate$ and hardware clock reading HC_p as input, and yields a list of states and *messages to be sent*, e.g. $[oldstate, msg, int.st._1, int.st._2, newstate]$, as output. The intermediate states, $int.st._1$ and $int.st._2$ in our example, are usually neglected in the classic computing model, as the state transition from $oldstate$ to $newstate$ is instantaneous anyway. We explicitly model these states to retain compatibility with our real-time computing model, where they will become more important.

Formally, a notation such as $\underline{\mathcal{A}}(m, oldstate, T) = [oldstate, \dots, newstate]$ will be used to refer to the output of the transition function of algorithm $\underline{\mathcal{A}}$ when a message m arrives on a processor with state $oldstate$ at hardware clock time T. For ease of presentation, we will omit the processor index since, in our model, it is implicitly contained in the message (cf. Section 4.1).

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

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

[1] Note that uniqueness is only required for analysis, cf. Section 4.1.

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

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

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

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

Booting We assume that every processor p in the system is in some initial state $istate_p \in Init_p(\underline{\mathcal{A}})$ right from the system start, at real-time $t = 0$. Clearly, in our message-driven setting, at least one input message is required to trigger the first action in an execution. For simplicity, we assert that the *algorithm* may specify whether it requires only one such message or one message for each processor. We will assume that all of these *init messages* arrive within a sufficiently short time interval, so that the initialization uncertainty does not significantly affect the time complexity of our algorithms. On the other hand, we consider the initialization uncertainty to be large enough to prohibit system-wide initial synchronization.

2.3. Hardware Clocks

The hardware clock of any processor p starts with some arbitrary initial value $HC_p(0)$ and then increases strictly, continuously and without bound. Depending on the problem under consideration, some additional restriction on the hardware clock is usually specified. For example,

- in Section 6.3, we assume that the hardware clocks are able to measure some real-time duration within a given interval $[\mu^-, \mu^+]$, i.e., we assume that there is some known value $\tilde{\mu}$, such that waiting for $\tilde{\mu}$ clock time units results in a real-time duration no shorter than μ^- and no longer than μ^+;
- in Chapter 7, we study the problem of drift-free clock synchronization and, thus, assume that all clocks progress at the same rate as real time.

A common assumption, which we also use in Chapter 8, is that each clock HC_p has 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, $t > t' \geq 0$:

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

With respect to the notation used in this work, when talking about *time units*, we mean *real-time units*, unless otherwise noted.

2.4. Executions

An execution in the classic computing model is a sequence ex of actions and an associated set of n hardware clocks $HC^{ex} = \{HC_p^{ex}, HC_q^{ex}, \ldots\}$. 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 $HC(ac) = HC_p^{ex}(t)$ and the state transition sequence $trans(ac) = [oldstate, \ldots, newstate]$ (including messages). Let $states(ac)$ be defined as the list (= sequence) of all states and $sent(ac)$ as the list of all messages in $trans(ac)$. The abbreviations $oldstate(ac)$ and $newstate(ac)$ will be used for the first and the last entry in $states(ac)$.

As an execution is a *sequence* of actions, there is a well-defined total order $\prec^{ex}$ on actions. We will omit the superscripts of $\prec^{ex}$ and HC_p^{ex} if the associated execution is clear from context. A message dependency ($ac \xrightarrow{M} ac'$) between two actions ac and ac' exists if $msg(ac') \in sent(ac)$. $intransit\_msgs(ac)$ denotes the set of messages in transit *after* action ac has sent all its messages but before any following action $ac' \succ ac$ in ex has had the opportunity to send or process messages.

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

EX1 ex must be a sequence of actions with a well-defined total order $\prec^{ex}$. The sequence must capture message causality (cf. Definition 2.1) and $time(ac)$ must be non-decreasing.

EX2 Processor states can only change during an action, i.e., if there are two actions $ac \prec ac'$ on the same processor p and there is no action on p between ac and ac', $newstate(ac) = oldstate(ac')$.

EX3 The first action ac at every processor p must occur in an initial state (denoted $istate_p^{ex}$) of $\underline{\mathcal{A}}$, i.e., $istate_p^{ex} = oldstate(ac) \in Init_p(\underline{\mathcal{A}})$.

EX4 The hardware clock readings of actions must be consistent with the hardware clock associated with the execution, i.e., $HC(ac) = HC_{proc(ac)}^{ex}(time(ac))$. The hardware clock readings must increase strictly ($\forall t, t', p : t < t' \Rightarrow HC_p^{ex}(t) < HC_p^{ex}(t')$), continuously and without bound.

EX5 Messages must be unique, i.e., there is at most one action sending some message m and at most one action receiving it. Message sending and receiving must be in the correct causal order. Messages can only be sent by and processed by the processor specified in the message.

EX6 Every non-input message that is received must have been sent.

Note that these properties do not require, for example, that all ordinary messages obey the message delay bounds or that all state transitions are in accordance with the transition function of $\underline{\mathcal{A}}$. These conditions will be specified by the *failure model* (see Section 4.2).

2.5. Systems

A *classic system* $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ is a system adhering to the classic computing model defined in Section 2.2, parameterized by the system size n and the interval $[\underline{\delta}^-, \underline{\delta}^+]$ specifying the bounds on the message delay. The uncertainty $\underline{\varepsilon}$ is defined as $\underline{\delta}^+ - \underline{\delta}^-$.

In [MS06a] and [Mos09], the notion of *s-admissible executions* was used: An execution is $\underline{s}$-admissible w.r.t. some system $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$, if the execution comprises n processors and the message delay for each ordinary message stays within $[\underline{\delta}^-, \underline{\delta}^+]$. This definition was useful for modeling failure-free executions; however, in the context of this work, it has been replaced with the more powerful concept of “failure models”. Informally speaking, a failure model specifies additional properties that an execution must satisfy, for example, “no message takes more than $\underline{\delta}^+$ time units to be delivered”.

Claiming that an algorithm $\underline{\mathcal{A}}$ solves a certain problem $\mathcal{P}$ for a classic system $\underline{s}$ under a given failure model $\underline{\mathcal{C}}$ means that all possible executions of $\underline{\mathcal{A}}$ that satisfy $\underline{\mathcal{C}}$ must also satisfy the properties required by $\mathcal{P}$ (see Chapter 5). The task of finding such an algorithm can be seen as providing a winning strategy to a player in an execution-creation game against an adversary, where the player provides the sets of initial states and the state transition function and the adversary chooses one initial state and the hardware clocks for every processor and controls the message delays (within the bounds $[\underline{\delta}^-, \underline{\delta}^+]$ provided by the system) as well as other factors permitted by the failure model. Note carefully that it is the system/the adversary and not the algorithm that determines the actual message delays in the classic computing model.

3. Towards the Real-Time Computing Model

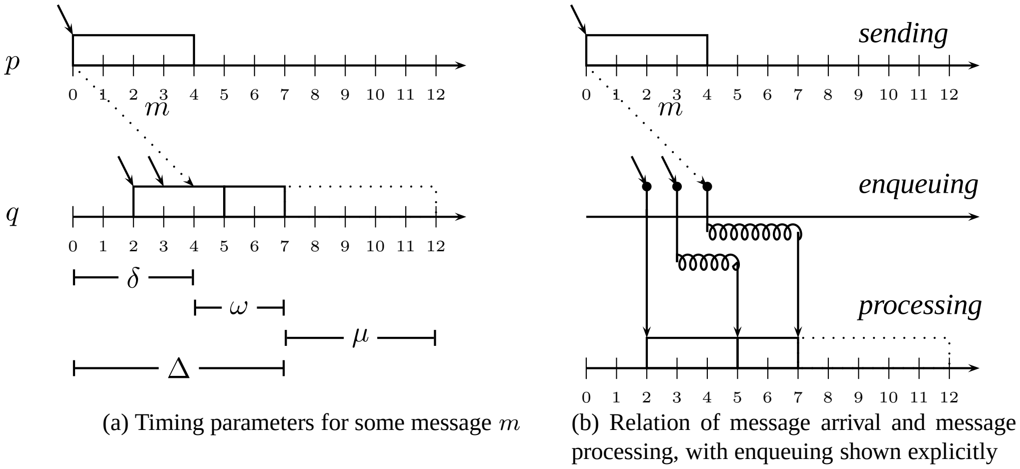
At a first glance, zero step-time computing models appear to be a good choice for modeling real-world systems, where message delays are often much higher than message processing times. There are applications like high speed networks, however, where this is not the case. Additionally, and more importantly,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

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

3.1. 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[1] 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

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



(a) Timing parameters for some message m

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

Figure 3.1.: Real-time computing model

real-time computing model from a zero-time action in the classic computing model, we will use the term *job* to refer to the former.

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

Queuing We must now distinguish two modes of a processor at any point in real-time t: *idle* and *busy* (i.e., currently executing a job). Since computing steps cannot be interrupted, a *queue* is needed to store ordinary, timer and input messages arriving while the processor is busy. We assume that messages are stored in the queue in the order in which they have arrived.

Scheduling 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.

We assume that the scheduling policy is *non-idling*; when the processor is idle, processing of an incoming message starts immediately. Similarly, when the processor finishes a job and the queue is non-empty, a message from the queue is taken and processing of the corresponding job starts without further delay.

Admission control In the classic zero step-time computing model, under certain failure models, a 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.

Each node is equipped with an *admission control* component, allowing the scheduler to drop certain messages instead of processing them. In contrast to the scheduling policy, the admission control function is usually specific to the algorithm. For example, in round-based algorithms, a policy such as "accept only the first round k message from every processor p" could make sense. This separation of concerns between the admission control component and the scheduling policy allows the scheduling policy to be optimized towards optimal performance for messages from correct processors, without having to worry about overloads created by faulty processors.

Scheduling/admission policy Formally, both the scheduling and the admission control policy are represented by a single function pol such that

$$pol(\text{queue state}, \text{algorithm state}, \text{HC reading}) = (\text{msg}, \text{queue state}^{\text{new}})$$

with queue state$^{\text{new}}$ $\subseteq$ queue state; msg $\notin$ queue state$^{\text{new}}$; and

- msg $\in$ queue state, representing the case where one message gets scheduled (and some messages might be dropped), or
- msg $= \perp$ and queue state$^{\text{new}}$ $= \emptyset$, representing the case where all messages in the queue (if any) are dropped.

This function is used whenever a scheduling decision is made, i.e., (a) at the end of a job and (b) whenever the queue is empty and a new message just arrived, and causes msg to be processed.

Since we assume *non-preemptive* scheduling, a message received while the processor is currently busy will be neither scheduled nor dropped until the current job has finished. "Delaying" the admission control decision in such a way has the advantage that no intermediate states can ever be used for admission control decisions.

Message delay The delay of a message is measured from the real-time of the *start of the job* sending the message to the arrival real-time at the destination processor (where the message will be enqueued or, if the processor is idle, immediately causes the corresponding job to start). Like in the classic computing model, message delays of ordinary messages must be within a system-wide lower bound δ^- and an upper bound δ^+. Like processing delays, the message delays and hence the bounds $[\delta^-, \delta^+]$ may again depend on the number of messages sent in the sending job.

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

Hardware clock We assume that the hardware clock can only be read at the beginning of a job.[2] 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.

State transitions Contrary to the classic computing model, the state transitions $oldstate \to \ldots \to newstate$ in a single computing step need not happen at the same time: Typically, they occur at different times during the job, allowing an intermediate state to be valid on a processor for some non-zero duration.

End-to-end delay Figure 3.1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. Part (a) shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dashed arrow. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not necessarily be known to the algorithm. Bounds on the queuing delay ω and the end-to-end delay Δ, however, are *not* parameters of the system model—in sharp contrast to the classic computing model (recall Chapter 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, $[\delta^-, \delta^+]$, $[\mu^-, \mu^+]$), the message pattern of the algorithm and the scheduling/admission policy, by performing a real-time schedulability analysis.

Part (b) of Figure 3.1 shows the detailed relation between message arrival (enqueuing) and actual message processing.

Note that messages dropped by the scheduler also have a well-defined end-to-end delay: $\Delta = \delta + \omega$, with ω denoting the queuing delay until the message is dropped (as opposed to the queuing delay until the message starts being processed). Thus, Δ for a dropped message is the time between the start of the job sending the message and the "drop event". Recall that ω need not be 0, since the decision to drop messages is only made whenever a scheduling decision is necessary.

3.2. Real-time Runs

This section formalizes the notion of a *real-time run* (*rt-run*), which corresponds to an execution in the classic computing model. A *rt-run* ru consists of a sequence of receive events, jobs and drop events, and of an associated set of n hardware clocks HC_p^{ru}.

A *receive event* $R = (\text{receive} : p, m, t)$ for a message m arriving at processor p at real-time t consists of the processor index $proc(R) = p$, the message $msg(R) = m$, and the arrival real-time $time(R) = t$. Recall that t is the enqueuing time in Figure 3.1(b).

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

A *job* J starting at real-time t on processor p is a 6-tuple, consisting of the processor index $proc(J) = p$, the message being processed $msg(J)$, the start time $begin(J) = t$, the job processing time $duration(J)$, the hardware clock reading $HC(J) = HC_p^{ru}(t)$, and the state transition sequence $trans(J) = [oldstate, \ldots, newstate]$. $states(J)$, $sent(J)$, $oldstate(J)$ and $newstate(J)$ are abbreviations for parts of $trans(J)$ and defined analogously to the classic computing model (see Section 2.4). Let $end(J)$ be defined as $begin(J) + duration(J)$.

A *drop event* $D = (\text{drop} : p, m, t)$ at real-time t on processor p consists of the processor index $proc(D) = p$, the message $msg(D)$, and the dropping real-time $time(D) = t$. These events represent messages getting dropped by the admission control component rather than being processed by a job.

Figure 3.1 provides an example of a rt-run, containing three receive events and three jobs on the second processor. For example, the dashed job on the second processor q consists of $(q, m, 7, 5, HC_q(7), [oldstate, \ldots, newstate])$, with m being the message received during the receive event $(\text{receive} : q, m, 4)$. Note that neither the actual state transition times nor the actual sending times of the sent messages are recorded in a job. Measuring all message delays from the beginning of a job and knowing that the state transitions and the message sends occur in the listed order at arbitrary times during the job is usually sufficient for algorithm and complexity analysis. The more detailed notion of *state transition traces* will be introduced later in Section 5.2.2.

Clearly, not all sequences of receive events, jobs and drop events are valid real-time system runs. Analogous to executions in the classic computing model, a rt-run of some algorithm $\mathcal{A}$ must satisfy the following properties:

RU1 ru must be a sequence of receive events, drop events and jobs with a well-defined total order $\prec^{ru}$. The sequence must capture message causality, and the begin times ($begin(J)$ for jobs, $time(R)$ and $time(D)$ for receive and drop events) must be non-decreasing.

RU2 Processor states can only change during a job.

RU3 The first job J at every processor p must occur in an initial state (denoted $istate_p^{ru}$) of $\mathcal{A}$, i.e., $istate_p^{ru} = oldstate(J) \in Init_p(\mathcal{A})$.

RU4 The hardware clock readings of jobs must be consistent with the hardware clocks associated with the rt-run. The hardware clock readings must increase strictly, continuously and without bound.

RU5 Messages must be unique, i.e., there is at most one job sending some message m, at most one receive event receiving it, and at most one job processing it or drop event dropping it. Message sending, receiving and processing/dropping must be in the correct causal order. Messages must only be sent by and received/processed/dropped by the processor specified in the message.

RU6 Every non-input message that is received must have been sent. Every message that is processed or dropped must have been received.

In addition, we require the following in the real-time model:

RU7 Jobs do not overlap: If $J \prec J'$ and $proc(J) = proc(J')$, then $end(J) \leq begin(J')$.

RU8 Drop events can only occur when a scheduling decision is made, i.e., immediately after a receive event when the processor is idle, or immediately after a job has finished processing.

A message dependency ($J \xrightarrow{M} R$) exists between a job J and a receive event R if $msg(R) \in sent(J)$. Clearly, RU5 and RU6 imply a local dependency between the receive event receiving a message and the job processing or the drop event dropping it. Thus, there is a (transitive) causal dependency between a job sending a message and the job processing that message.

A processor p is *busy* at time t if there is some job J such that $begin(J) \leq t < end(J)$; otherwise, it is *idle*.

3.3. Systems

A real-time system s is a system adhering to the real-time computing model, parameterized by the system size n and two intervals $[\delta^-, \delta^+]$ and $[\mu^-, \mu^+]$, specifying the bounds on the message delay and on the job duration.

Considering δ^-, δ^+, μ^- and μ^+ to be constants would give an unfair advantage to broadcast-based algorithms when comparing some algorithms' time complexity: Computation steps would take between μ^- and μ^+ time units, independently of the number of messages sent. This makes it impossible to derive a meaningful time complexity lower bound for systems in which a constant-time broadcast primitive is not available. Corollary 7.17 will show an example.

Therefore, the interval boundaries δ^-, δ^+, μ^- and μ^+ can be either constants or non-decreasing functions $\{0, \ldots, n-1\} \rightarrow \mathbb{R}^+$, 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 3.1. During some job, ordinary messages to exactly three processors are sent. The duration of this job lies within $[\mu^-_{(3)}, \mu^+_{(3)}]$. Each of these messages has a message delay between $\delta^-_{(3)}$ and $\delta^+_{(3)}$. The delays of the three messages need not be the same.

To be useful, these functions must satisfy some conditions:

- Intervals must be well-defined: $\forall \ell : \delta^-_{(\ell)} \leq \delta^+_{(\ell)} \wedge \mu^-_{(\ell)} \leq \mu^+_{(\ell)}$
- Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. Formally, $\forall i, j \geq 1 : f_{(i+j)} \leq f_{(i)} + f_{(j)}$ (for $f = \delta^-$, δ^+, μ^- and μ^+).

[3] 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 ordinary messages and destination processors in 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.

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.

Similar to executions in the classic computing model, the creation of an rt-run can be seen as a game of a player (the algorithm) against an adversary in the "arena" of a system s, a failure model $\mathcal{C}$ (see next chapter) and a scheduling/admission policy pol. For example, when using the failure model FAULT-FREE, the player provides sets of initial states and the state transition function, and the adversary can

- for every processor, choose an initial state from the set provided by the player, hardware clock parameters (such as initial value or drift, depending on the hardware clock model used) and the time at which the init message will arrive,
- for every ordinary 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 ordinary messages, choose a value within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$ for its processing time (and associated overhead, e.g. for scheduling).

4. Admissibility of Executions and Real-Time Runs

This chapter will formalize the notion of a *message*, introduce the concept of *failure models*, and provide definitions for the well-known FAULT-FREE, f-CRASH and f-BYZANTINE failure models.

4.1. Messages

Formally, a "real message" sent or processed during an execution or rt-run, e.g. *"ordinary message 14635 from p to q containing ‹Hello›"*, is a tuple consisting of

1. some identifier which makes the message unique w.r.t. the given execution or rt-run and
2. the "message template" specified in the transition function, e.g. *"ordinary message from p to q containing ‹Hello›"*, which, in turn, is a data structure consisting of

   a) the "message content" $content(m)$, e.g. *"Hello"*,

   b) the message type information $type(m) \in \{\text{ordinary message}, \text{timer message}, \text{input message}\}$ and

   c) the delivery information, which depends on the type of message:

      - For some ordinary message m_o, this is the sending processor $sender(m_o)$ and the receiving processor $destination(m_o)$.
      - For some timer message m_t, this is the processor $proc(m_t)$ and the designated arrival (hardware clock) time. Let $sHC(m_t)$ denote the hardware clock time for which the timer message m_t is set or $HC(ac)/HC(J)$ of the job setting the timer, whichever is higher. This is the hardware clock value by which the timer is supposed to arrive.
      - For some input message m_i, the delivery information contains the destination processor $destination(m_i)$.

To ease presentation, we will just use the term "message" for real messages, message templates and message contents, when it is clear from context which part of the message is meant. In addition, we will mix these types of messages without explicitly converting them. For example, $trans(J) = \mathcal{A}(msg(J), oldstate(J), HC(J))$ will be used to denote that job J's transition sequence conforms to algorithm $\mathcal{A}$'s transition function. This notation is informal since $trans(J)$ contains real messages and $msg(J)$ is a real message, whereas the transition function only specifies message templates.

Note that the uniqueness of a real message is only required for analysis. Uniqueness can be guaranteed, for example, by referring to the unique (job that sent the message, destination processor)-pair or by numbering messages sequentially.

4.2. Failure Models

“Conformance to a certain failure model” replaces the notion of “s-admissibility” used in previous works ([MS06a], [Mos09]). Formally, a failure model is a predicate $\mathcal{C}$ on a (system, algorithm, scheduling/admission policy, execution/rt-run) tuple. For example, in the classic computing model and the real-time computing model with bounded-drift clocks, $\mathcal{C}$ can be seen as a characteristic function $\mathcal{C}((n, [\underline{\delta}^-, \underline{\delta}^+]), \underline{\mathcal{A}}, ex)$ or $\mathcal{C}((n, [\delta^-, \delta^+], [\mu^-, \mu^+]), \mathcal{A}, pol, ru)$, respectively, indicating whether ex/ru is an *admissible* execution/rt-run w.r.t. the given failure model in the given system running the given algorithm or not.

To illustrate the concept of failure models, this section will define the well-known failure models FAULT-FREE, f-CRASH and f-BYZANTINE with bounded drift for the classic as well as for the real-time computing model.

4.2.1. Prerequisites

For each execution ex, we define $\mathcal{AC}(ex)$ as the set of actions in ex. Likewise, for each rt-run ru, let $\mathcal{R}(ru)$, $\mathcal{D}(ru)$ and $\mathcal{J}(ru)$ denote the sets of receive events, drop events and jobs in ru, respectively. Let $\mathcal{JD}(ru) = \mathcal{J}(ru) \cup \mathcal{D}(ru)$.

With respect to messages, let $\mathcal{M}_o(ex/ru)$ denote the set of ordinary messages and $\mathcal{M}_t(ex/ru)$ the set of timer messages.[1]

Abbreviations

The failure models in this section will be presented as first-order logic predicates in the form FAILURE-MODEL(ru) $:\Leftrightarrow \forall x : P(x) \wedge \forall y : Q(y) \wedge \ldots$. To ease presentation, we will

- write FAILURE-MODEL(ru) instead of FAILURE-MODEL$(s, \mathcal{A}, pol, ru)$ and write $P(x)$ instead of $P(ru, x)$, likewise for failure models and predicates based on classic computing model executions,
- write $\forall ac : \ldots$ and $\exists ac : \ldots$ instead of $\forall ac \in \mathcal{AC}(ex) : \ldots$ and $\exists ac \in \mathcal{AC}(ex) : \ldots$, with R, D, J, JD, m_o, m_t and p associated analogously with the sets $\mathcal{R}(ru)$, $\mathcal{D}(ru)$, $\mathcal{J}(ru)$, $\mathcal{JD}(ru)$, $\mathcal{M}_o(ex/ru)$, $\mathcal{M}_t(ex/ru)$ and Π, respectively,
- avoid parentheses when the desired operator precedence is clear (e.g. from context or from indentation).

[1] Formally,
$\mathcal{M}_o(ex) = \{m : type(m) = \text{ordinary msg.} \wedge (\exists ac \in \mathcal{AC}(ex) : msg(ac) = m \vee m \in sent(ac))\}$,
$\mathcal{M}_o(ru) = \{m : type(m) = \text{ordinary msg.} \wedge ((\exists R \in \mathcal{R}(ru) : msg(R) = m) \vee (\exists J \in \mathcal{J}(ru) : m \in sent(J)))\}$.
$\mathcal{M}_t(ex/ru)$ are defined analogously with $type(m) =$ timer message.

Scheduling and admission control

The predicates $obeys\_pol(R)$ and $obeys\_pol(J)$ will be used to refer to the scheduling and the admission control policy: $obeys\_pol(R)/(J)$ is satisfied, if

- at time $time(R)$, after R, if no job is currently running (in the case of $obeys\_pol(R)$) or
- at time $end(J)$, after J, if there are still messages that have been received but not processed or dropped (in the case of $obeys\_pol(J)$),

a scheduling decision is made, causing messages to be dropped and/or a job to be started (according to the chosen policy pol).

4.2.2. FAULT-FREE

This is how a fault-free failure model with bounded drift is usually defined:

FAULT-FREE$_\rho(ex)$ ***(classic computing model)*** $:\Leftrightarrow$

$\forall m_o : is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+)$	All ordinary msgs obey the message delay bounds.
$\wedge\ \forall m_t : arrives\_timely(m_t)$	All timers arrive in time.
$\wedge\ \forall ac : follows\_alg(ac)$	All actions execute the algorithm.
$\wedge\ \forall p : bounded\_drift(p, \rho)$	The drift of all hardware clocks is bounded by ρ.

FAULT-FREE$_\rho(ru)$ ***(real-time computing model)*** $:\Leftrightarrow$

$\forall m_o : is\_timely\_msg(m_o, \delta^-, \delta^+)$	All ordinary msgs obey the message delay bounds.
$\wedge\ \forall m_t : arrives\_timely(m_t)$	All timers arrive in time.
$\wedge\ \forall R : obeys\_pol(R)$ $\wedge\ \forall J : obeys\_pol(J)$	Scheduling/admission according to pol.
$\wedge\ \forall J : follows\_alg(J)$	All jobs execute the algorithm.
$\wedge\ \forall J : is\_timely\_job(J, \mu^-, \mu^+)$	All jobs obey the processing delay bounds.
$\wedge\ \forall p : bounded\_drift(p, \rho)$	The drift of all hardware clocks is bounded by ρ.

In addition to $obeys\_pol$, which is defined in the previous section, the following predicates are used in the definition of FAULT-FREE:

$$
\begin{aligned}
&is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+) :\Leftrightarrow \exists ac, ac' : m_o \in trans(ac) \wedge m_o = msg(ac') \\
&\qquad\qquad\qquad\qquad \wedge\, time(ac') - time(ac) \in [\underline{\delta}^-, \underline{\delta}^+] \\
&arrives\_timely(m_t) :\Leftrightarrow \exists ac, ac' : m_t \in trans(ac) \wedge m_t = msg(ac') \wedge HC(ac') = sHC(m_t) \\
&follows\_alg(ac) :\Leftrightarrow trans(ac) = \underline{A}(msg(ac), oldstate(ac), HC(ac))
\end{aligned}
$$

$$
\begin{aligned}
&is\_timely\_msg(m_o, \delta^-, \delta^+) :\Leftrightarrow \exists J, R : m_o \in trans(J) \wedge m_o = msg(R) \\
&\qquad\qquad\qquad\qquad \wedge\, time(R) - begin(J) \in [\delta^-, \delta^+] \\
&arrives\_timely(m_t) :\Leftrightarrow \exists J, R : m_t \in trans(J) \wedge m_t = msg(R) \\
&\qquad\qquad \wedge\, (time(R) = \max\{HC^{-1}_{proc(m_t)}(sHC(m_t)), end(J)\})
\end{aligned}
$$

$$
\begin{aligned}
&follows\_alg(J) :\Leftrightarrow trans(J) = \mathcal{A}(msg(J), oldstate(J), HC(J))\\
&is\_timely\_job(J, \mu^-, \mu^+) :\Leftrightarrow duration(J) \in [\mu^-, \mu^+]
\end{aligned}
$$

$$bounded\_drift(p, \rho) :\Leftrightarrow \forall t > t' \geq 0 : (t - t')(1 - \rho) \leq HC_p(t) - HC_p(t') \leq (t - t')(1 + \rho)$$

Note that $arrives\_timely$ in the real-time computing model ensures that timer messages arrive either at $sHC(m_t)$ or at the end of the job sending the message, whichever happens later. This allows for code lines such as “set timer m_t for current_hc”, which cause a timer message to arrive directly after the job containing this line has finished, possibly triggering another job right away (unless any other messages in the queue are preferred by the scheduling policy).

For ease of presentation, δ^-, δ^+, μ^- and μ^+ are treated as constants here; the generalization to functions (cf. Section 3.3) is straightforward: The generic version of is_timely_job would be defined as $duration(J) \in [\mu^-_{(\ell)}, \mu^+_{(\ell)}]$; likewise, is_timely_msg would become $\exists J, R : m_o \in trans(J) \wedge m_o = msg(R) \wedge time(R) - begin(J) \in [\delta^-_{(\ell)}, \delta^+_{(\ell)}]$. In both cases, ℓ refers to the number of ordinary messages in $sent(J)$.

4.2.3. f-CRASH

Other failure models can be derived from FAULT-FREE by adding exceptions to certain parts of the predicate. Consider, for example, a model in which up to f processors may crash. Intuitively, a crashed processor is a processor that eventually stops making state transitions and sending messages.

In the real-time computing model, we can model this elegantly by messages still arriving through receive events but jobs no longer being scheduled. In the classic computing model, however, message reception and processing is tightly coupled within a single action. Thus, in the classic computing model, we will replace all state transitions after a processor has crashed with the one-element “NOP transition sequence” $[s]$, with $s := oldstate(ac) = newstate(ac)$.

The notion of crashing defined below allows for *unclean crashes*, i.e., the last action/job on a processor might execute only part of its state transition sequence. Changes to FAULT-FREE are underlined.

f-CRASH$_\rho(ex)$ *(classic computing model)* $:\Leftrightarrow$

$$
\begin{aligned}
&\exists F : |F| = f \wedge F \subseteq \Pi\\
&\qquad \wedge\ \forall m_o : is\_timely\_msg(m_o, \underline{\delta^-}, \underline{\delta^+})\\
&\qquad \wedge\ \forall m_t : arrives\_timely(m_t)\\
&\qquad \wedge\ \forall ac : follows\_alg(ac) \vee \underline{[proc(ac) \in F \wedge ((is\_last(ac) \wedge follows\_alg\_partially(ac))}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \underline{\vee\ arrives\_after\_crash(ac))]}\\
&\qquad \wedge\ \forall p : bounded\_drift(p, \rho)
\end{aligned}
$$

f-CRASH$_\rho(ru)$ *(real-time computing model)* $:\Leftrightarrow$

$$\begin{array}{l}
\exists F : |F| = f \land F \subseteq \Pi \\
\quad \land\ \forall m_o : \mathit{is\_timely\_msg}(m_o, \delta^-, \delta^+) \\
\quad \land\ \forall m_t : \mathit{arrives\_timely}(m_t) \\
\quad \land\ \forall R : \mathit{obeys\_pol}(R) \underline{\lor [\mathit{proc}(R) \in F \land \mathit{arrives\_after\_crash}(R) \land \mathit{drops\_msg}(R)]} \\
\quad \land\ \forall J : \mathit{obeys\_pol}(J) \underline{\lor [\mathit{proc}(J) \in F \land \mathit{is\_last}(J) \land \mathit{drops\_all\_queued}(J)]} \\
\quad \land\ \forall J : \mathit{follows\_alg}(J) \underline{\lor [\mathit{proc}(J) \in F \land \mathit{is\_last}(J) \land \mathit{follows\_alg\_partially}(J)]} \\
\quad \land\ \forall J : \mathit{is\_timely\_job}(J, \mu^-, \mu^+) \\
\quad \land\ \forall p : \mathit{bounded\_drift}(p, \rho)
\end{array}$$

The following predicates are used in addition to those defined in the previous section, with “suffix” denoting a possibly empty sequence of states and messages:

$$\begin{array}{l}
\mathit{is\_last}(ac) :\Leftrightarrow \forall ac' : (ac \prec ac' \land \mathit{proc}(ac) = \mathit{proc}(ac')) \Rightarrow \mathit{trans}(ac') = [\mathit{oldstate}(ac')] \\
\mathit{follows\_alg\_partially}(ac) :\Leftrightarrow \exists \text{suffix} : \mathit{trans}(ac) + \text{suffix} = \underline{\mathcal{A}}(\mathit{msg}(ac), \mathit{oldstate}(ac), \mathit{HC}(ac)) \\
\mathit{arrives\_after\_crash}(ac) :\Leftrightarrow \exists ac^{last} : ac^{last} \prec ac \land \mathit{proc}(ac^{last}) = \mathit{proc}(ac) \land \mathit{is\_last}(ac^{last})
\end{array}$$

$$\begin{array}{l}
\mathit{arrives\_after\_crash}(R) :\Leftrightarrow \exists J^{last} : J^{last} \prec R \land \mathit{proc}(J^{last}) = \mathit{proc}(R) \land \mathit{is\_last}(J^{last}) \\
\mathit{drops\_msg}(R) :\Leftrightarrow \exists D : \mathit{time/proc/msg}(D) = \mathit{time/proc/msg}(R) \\
\mathit{is\_last}(J) :\Leftrightarrow \nexists J' : \mathit{proc}(J) = \mathit{proc}(J') \land J \prec J' \\
\mathit{drops\_all\_queued}(J^{last}) :\Leftrightarrow \\
\quad \forall R : [\mathit{proc}(R) = \mathit{proc}(J^{last}) \land (\nexists JD : JD \prec J^{last} \land \mathit{msg}(JD) = \mathit{msg}(R))] \\
\qquad \Rightarrow \exists D : \mathit{time}(D) = \mathit{end}(J^{last}) \land \mathit{msg}(D) = \mathit{msg}(R) \\
\mathit{follows\_alg\_partially}(J) \Leftrightarrow \exists \text{suffix} : \mathit{trans}(J) + \text{suffix} = \mathcal{A}(\mathit{msg}(J), \mathit{oldstate}(J), \mathit{HC}(J))
\end{array}$$

4.2.4. f-BYZANTINE

Another common extension of FAULT-FREE is a model with f Byzantine (i.e. arbitrary faulty) processors. Note that the fact that all jobs need a receive event does not reduce the power of a Byzantine node since it can send an arbitrary number of timer messages to itself.

f-BYZANTINE$_\rho(ex)$ *(classic computing model)* $:\Leftrightarrow$

$$\begin{array}{l}
\exists F : |F| = f \land F \subseteq \Pi \\
\quad \land\ \forall m_o : \mathit{is\_timely\_msg}(m_o, \underline{\delta}^-, \underline{\delta}^+) \\
\quad \land\ \forall m_t : \mathit{arrives\_timely}(m_t) \underline{\lor [\mathit{proc}(m_t) \in F]} \\
\quad \land\ \forall ac : \mathit{follows\_alg}(ac) \underline{\lor [\mathit{proc}(ac) \in F]} \\
\quad \land\ \forall p : \mathit{bounded\_drift}(p, \rho) \underline{\lor [p \in F]}
\end{array}$$

f-BYZANTINE$_\rho(ru)$ *(real-time computing model)* $:\Leftrightarrow$

$$\begin{array}{l}
\exists F : |F| = f \land F \subseteq \Pi \\
\quad \land\ \forall m_o : \mathit{is\_timely\_msg}(m_o, \delta^-, \delta^+) \\
\quad \land\ \forall m_t : \mathit{arrives\_timely}(m_t) \underline{\lor [\mathit{proc}(m_t) \in F]}
\end{array}$$

$$\begin{array}{l}
\wedge\ \forall R : obeys\_pol(R) \underline{\vee [proc(R) \in F]} \\
\wedge\ \forall J : obeys\_pol(J) \underline{\vee [proc(J) \in F]} \\
\wedge\ \forall J : follows\_alg(J) \underline{\vee [proc(J) \in F]} \\
\wedge\ \forall J : is\_timely\_job(J, \mu^{-}, \mu^{+}) \underline{\vee [proc(J) \in F]} \\
\wedge\ \forall p : bounded\_drift(p, \rho) \underline{\vee [p \in F]}
\end{array}$$

5. Problems, Algorithms and Proofs

This chapter defines what it means to prove that some algorithm solves some given problem. The aim of this chapter is to provide a formal framework for specifying a problem in the same way for the classic as well as for the real-time model. The following sections present two suitable approaches.

5.1. aj-problems

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

Definition 5.1 (aj-events[1])**.** The *aj-event* ev corresponding to action ac or to job J is a 4-tuple, consisting of the processor index $proc(ev) = proc(ac)/proc(J)$, the start real-time $begin(ev) = time(ac)/begin(J)$, the hardware clock value $HC(ev) = HC(ac)/HC(J)$ and the state transition sequence $trans(ev) = trans(ac)/trans(J)$.

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

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

Note, however, that aj-problems do not restrict algorithm messages. This makes them well-suited for system model transformation proofs, since changing the message that triggered some computing step (for example, by encapsulating it into a message of the simulation algorithm) does not violate an algorithm's capability to solve some particular aj-problem.

Example 5.2 (Terminating (Drift-Free) Clock Synchronization)**.** Let $is\_lastevent(ev, p)$ be $true$ if ev is the last aj-event on processor p. Formally: $is\_lastevent(ev, p) :\Leftrightarrow proc(ev) = p \wedge \nexists ev' : ((ev \prec ev') \wedge (proc(ev') = p))$.

- Precondition[2]: Apart from the init messages, there are no input messages.
- *Termination:* All processors eventually terminate.
  $\forall p : \exists ev : is\_lastevent(ev, p)$

[1] Note that this definition of *aj-events* has nothing to do with *receive events* or *drop events* in rt-runs.

[2] The fact that hardware clocks do not drift is not a precondition here. Thus, this problem can only be solved under drift-free failure models.

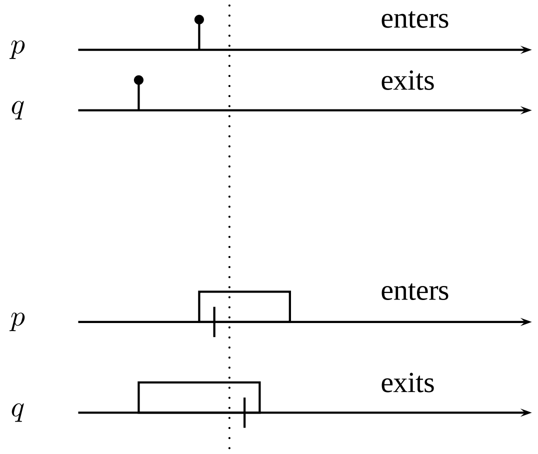


Figure 5.1.: Example of a mutual exclusion violation in the real-time computing model (top: aj-trace, bottom: rt-run).

- *Agreement:* After all processors have terminated, all processors have adjusted clocks (= hardware clock plus some local adjustment variable adj) within γ of each other.

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

This example reveals that *aj-problem* specifications have some drawbacks: predicates can only be defined for points in time where some event occurs. This is especially inconvenient for the definition of *drifting* clock synchronization (see Example 5.9 in Section 5.2.4). In addition, the usage of some distinguished state like $newstate$ is error-prone. Consider, for example, the following mutual exclusion condition: a processor p may only enter the critical section during event ev, if $\forall q : newstate(last(q, ev)).in\_cs = false$, with $last(q, ev)$ being the last event on processor q before ev. In the classic computing model, this condition ensures mutual exclusion. In the real-time computing model, however, the situation depicted in Figure 5.1 can occur. While the aj-trace gives the impression that mutual exclusion is maintained, the rt-run shows that this is not always the case. As the actual state transitions can occur at any time during a job (marked as ticks in the figure), it may happen that, at a certain time (marked as a dotted vertical line), p has entered the critical section although q has not left yet.

5.2. st-problems

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

5.2.1. Requirements

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

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

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

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

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

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

5.2.2. State Transitions

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

- $(process : t, p, m)$: At time $time(ev) = t$, processor $proc(ev) = p$ starts processing message $msg(ev) = m$.
- $(send : t, p, m)$: At time $time(ev) = t$, processor $proc(ev) = p$ sends message $msg(ev) = m$.
- $(transition : t, p, s, s')$: At time $time(ev) = t$, processor $proc(ev) = p$ changes its internal state from $oldstate(ev) = s$ to $newstate(ev) = s'$.[3]
- $(input : t, m)$: At time $time(ev) = t$, input message $msg(ev) = m$ arrives from an external source.

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

Definition 5.3. Each action ac at time t on processor p triggered by some message m is mapped to $(process : t, p, m)$, followed by $(send : t, p, m')$ or $(transition : t, p, s, s')$ for every message and every state transition in $trans(ac)$ (in the correct order). If m is an input message, there is a $(input : t, m)$ st-event immediately before the $process$ st-event. All of these st-events carry the same time t.

A message dependency ($ev \xrightarrow{M} ev'$) between two events ev and ev' exists if $type(ev) \in \{send, input\}$, $type(ev') = process$ and $msg(ev) = msg(ev')$. As the order of the original execution is preserved, this definition implies that message causality is captured in the newly created st-trace, since every execution captures message causality (condition EX1, cf. Section 2.4).

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

Definition 5.4. Each job J starting at time t with duration d on processor p triggered by some message m is mapped to $(process : t, p, m)$, followed by $(send : t', p, m')$ or $(transition : t', p, s, s')$ for every message and every state transition in $trans(J)$ (in the correct order). The state transition and send times (t') must be within $[t, t + d]$ and non-decreasing.

Receive events are only mapped to the st-trace if they are caused by input messages. In that case, the receive event is mapped to $(input : t, m)$. A drop event at time t on processor p triggered by message m is mapped to $(process : t, p, m)$.

In the st-trace, the st-events are ordered by their time while preserving the original order of the rt-run as much as possible. The times of $send$ st-events (within $[t, t + d]$) must be chosen such that message causality is captured.[4]

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

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

Any st-events occurring at the same time t can be reordered as long as the reordering is causally consistent with the original st-trace (recall Section 2.1). Every such reordering results in another valid st-trace. Thus, for every execution, there is one unique set of st-events, which can be ordered into many st-traces. In the real-time computing model, however, the set of st-events corresponding to some real-time run ru is usually not unique, even if all jobs occur at different times, as the state transitions and message sends within some job can occur at different times within the job processing interval.

Example 5.5. Assume $\underline{\delta}^- = 0$, i.e., messages can be sent in zero time. Let ex be an execution consisting of two actions ac $(p, m_{init}, t, HC_p(t), [s_{old}, s_1, m, s_{new}])$ and ac' $(q, m, t, HC_q(t), [s'_{old}, s'_{new}])$. Figure 5.2 shows the st-traces corresponding to ex.

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

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

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

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

Figure 5.2.: Example of three st-traces.

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

5.2.3. Global States

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

- *At the beginning*:
  Insert a set $\{(t, istate_1, \ldots, istate_n, \{\}) : 0 \leq t \leq t'\}$, 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 $\{(t, s_1, \ldots, s_n, pending\_msgs) : time(ev) \leq t \leq time(ev')\}$ containing the global state after ev but before ev'. The effects of st-events on the global state are as follows:

  - $(process : t, p, m)$ removes m from $pending\_msgs$,
  - $(send : t, p, m)$ or $(input : t, m)$ adds m to $pending\_msgs$, and
  - $(transition : t, p, s, s')$ changes processor p's state to s'.

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

The state sets are totally ordered by time.

Example 5.6. Figure 5.3 shows the first st-trace presented in Figure 5.2, annotated by the generated state sets.

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

Let $gstates(tr)$ denote the set of all global states appearing in the annotated st-trace tr. The annotated st-trace implies a total order $\prec^{tr}$ on the set of all st-events and all global states, i.e., on the set $tr \cup gstates(tr)$.

5.2.4. Problem Definitions

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

Example 5.7 (Mutual Exclusion)**.** We define the following predicates:

$$is\_enter(ev) :\Leftrightarrow type(ev) = transition \land oldstate(ev).in\_cs = false \\ \land newstate(ev).in\_cs = true,$$

$$is\_want\_to\_enter(ev) :\Leftrightarrow type(ev) = input \land content(msg(ev)) = \text{“want to enter”},$$

$$\left\{ \begin{pmatrix} 0 \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix}, \cdots, \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, (input : t, m_{init}), \left\{ \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{m_{init}\} \end{pmatrix} \right\}, (process : t, p, m_{init}),$$

$$\left\{ \begin{pmatrix} t \\ s_{old} \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, (transition : t, p, s_{old}, s_1), \left\{ \begin{pmatrix} t \\ s_1 \\ s'_{old} \\ \{\} \end{pmatrix} \right\}, (send : t, p, m), \left\{ \begin{pmatrix} t \\ s_1 \\ s'_{old} \\ \{m\} \end{pmatrix} \right\},$$

$$(transition : t, p, s_1, s_{new}), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{old} \\ \{m\} \end{pmatrix} \right\}, (process : t, q, m), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{old} \\ \{\} \end{pmatrix} \right\},$$

$$(transition : t, q, s'_{old}, s'_{new}), \left\{ \begin{pmatrix} t \\ s_{new} \\ s'_{new} \\ \{\} \end{pmatrix}, \cdots \right\}$$

Figure 5.3.: Example of an annotated st-trace, containing both st-events and global states

with $is\_exit(ev)$ and $is\_want\_to\_exit(ev)$ defined analogously. Then, mutual exclusion can be specified as follows:

- Precondition I: For every processor, the st-events of its input messages form a (finite or infinite) sequence starting with "init" and then alternating between "want to enter" and "want to exit" (starting with "want to enter").

- Precondition II: Eventually, every processor will be told to leave the critical section. Formally, for every processor p and every integer i: If tr contains at least i st-events satisfying $is\_want\_to\_enter$ on p and at least i st-events satisfying is_enter on p, then tr also contains at least i st-events satisfying $is\_want\_to\_exit$ on p.[5]

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

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

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

  $\forall ev \in tr : is\_want\_to\_enter(ev) \Rightarrow (\exists g \succ ev : s_{proc(ev)}(g).in\_cs = true)$

[5] Note that the condition on the $enter$ st-events is necessary: If this precondition just required an equal number of $want\_to\_enter$ and $want\_to\_exit$ st-events, an algorithm could wait for the $want\_to\_exit$ message *before* entering the critical section. This is not desired behavior, since it would allow the mutual exclusion algorithm to force the duration of a critical section to be arbitrarily small.

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

  $\forall ev \in tr : is\_want\_to\_exit(ev) \Rightarrow (\exists g \succ ev : s_{proc(ev)}(g).in\_cs = false)$

- *Safety:* Do not enter or exit the critical section without a reason.

  $$\forall p : \forall g \in gstates(tr) : count(is\_enter, p, g) \leq count(is\_want\_to\_enter, p, g) \\ \wedge\, count(is\_exit, p, g) \leq count(is\_want\_to\_exit, p, g)$$

  with $count(P, p, g)$ denoting the number of st-events satisfying P on p before g.

Example 5.8 (Terminating (Drift-Free) Clock Synchronization [LL84b])**.** Note that this problem can only be solved under drift-free failure models. We define $is\_finalstate(g) :\Leftrightarrow \forall g' \succ g : \forall p : s_p(g) = s_p(g')$. Let the *adjusted clock value* $AC_p(g)$ be defined as $HC_p^{tr}(time(g)) + s_p(g).adj$.

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

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

- *Termination:* All processors eventually terminate.

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

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

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

Example 5.9 (Drifting Clock Synchronization [AW04])**.** $AC_p(g)$ is defined as in the previous example.

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

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

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

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

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

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

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

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

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

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

5.2.5. Relationship to aj-problems

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

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

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

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

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

5.3. Proofs

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

The notion of *failure models* (cf. Section 4.2) can be used to prove that some algorithm *solves* some problem $\mathcal{P}$ in a certain system. In the classic computing model, we can define correctness and impossibility in the usual way:

Definition 5.10 (Correctness)**.** An algorithm $\underline{\mathcal{A}}$ solves some problem $\mathcal{P}$ in some system $\underline{s}$ under some failure model $\underline{\mathcal{C}}$ if, and only if, for every execution ex of $\underline{\mathcal{A}}$ satisfying $\underline{\mathcal{C}}(\underline{s}, \underline{\mathcal{A}}, ex)$, ex also satisfies $\mathcal{P}$.

Definition 5.11 (Impossibility)**.** A problem $\mathcal{P}$ is impossible to solve in some system $\underline{s}$ under some failure model $\underline{\mathcal{C}}$ if, and only if, for every algorithm $\underline{\mathcal{A}}$ there exists an execution ex of $\underline{\mathcal{A}}$ which satisfies $\underline{\mathcal{C}}(\underline{s}, \underline{\mathcal{A}}, ex)$ but violates $\mathcal{P}$.

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

Definition 5.12 (Correctness)**.** An algorithm $\mathcal{A}$ solves some problem $\mathcal{P}$ in some system s under some failure model $\mathcal{C}$ with scheduling/admission policy pol if, and only if, for every rt-run ru of $\mathcal{A}$ satisfying $\mathcal{C}(s, \mathcal{A}, pol, ru)$, ru also satisfies $\mathcal{P}$.

Definition 5.13 (Impossibility)**.** A problem $\mathcal{P}$ is impossible to solve in some system s under some failure model $\mathcal{C}$ with scheduling/admission policy pol if, and only if, for every algorithm $\mathcal{A}$, there exists an rt-run ru of $\mathcal{A}$ that satisfies $\mathcal{C}(s, \mathcal{A}, pol, ru)$ but violates $\mathcal{P}$.

5.4. Notation for Specifying Algorithms

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

5.5. Time Complexity

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

6. Transformations

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

One direction (Section 6.3), simulating a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ on top of a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$, where the message delays of the real-time system match those of the classic system, is quite straightforward: It suffices to implement an artificial processing delay, the queuing of messages arriving during such a simulated job, and the scheduling/admission policy. This simulation allows to run any real-time computing model algorithm $\mathcal{A}$ designed for a system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^- \leq \underline{\delta}^-$, $\delta^+ \geq \underline{\delta}^+$ on top of it, thereby resulting in a correct classic computing model algorithm.

For the other direction (Section 6.2), it is possible to simulate a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$, where the end-to-end delays of the classic system match those of the real-time system, i.e. $[\underline{\delta}^-, \underline{\delta}^+] = [\Delta^-, \Delta^+]$. Recall that the end-to-end delay bounds are equal to those of the message delay in the classic, but not in the real-time computing model, since the end-to-end delays additionally depend on queuing effects in the latter. Thus, this direction is more tricky: First, because of the uncertainty regarding when a job's state transition is actually performed, the transformed algorithm solves a slightly different problem than the original algorithm. Second, and more importantly, a *real-time schedulability analysis* must be conducted in order to break the circular dependency of algorithm $\underline{\mathcal{A}}$ and end-to-end delays $\Delta \in [\Delta^-, \Delta^+]$ (and vice versa): On one hand, the classic computing model algorithm $\underline{\mathcal{A}}$, run atop of the simulation, might need to know the *simulated* message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, which are just the end-to-end delay bounds $[\Delta^-, \Delta^+]$ of the underlying simulation. Those end-to-end delays, on the other hand, involve the queuing delay ω and are thus dependent on (the message pattern of) $\underline{\mathcal{A}}$ and hence on $[\underline{\delta}^-, \underline{\delta}^+]$. This circular dependency is "hidden" in the parameters of the classic computing model, but necessarily pops up when one tries to instantiate this model in a real system.

6.1. Problem Transformations

When running a real-time computing model algorithm in a classic system (Section 6.3), the st-traces of the simulated rt-run and the ones of the actual execution are very similar: Ignoring variables solely used by the simulation algorithm, it turns out that the same state transitions occur in the rt-run and in the corresponding execution. Consequently, this transformation inherently preserves most correctness and impossibility proofs.

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

Fortunately, however, it is possible to show that if some algorithm solves some st-problem $\mathcal{P}$ in some classic system, the same algorithm can be used to solve a variant of $\mathcal{P}$, denoted $\mathcal{P}^*_{\mu^+}$, in some corresponding real-time system. The following section will formalize the exact relationship between $\mathcal{P}$ and $\mathcal{P}^*_{\mu^+}$.

6.1.1. Shuffles

Definition 6.1. Let tr be a st-trace. A μ^+*-shuffle* of tr is constructed by:

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

   If μ^+ is a function $\{0, \ldots, n-1\} \to \mathbb{R}$ rather than a number (cf. Section 3.3), a $send$ or $transition$ st-event ev may be moved by at most $\mu^+_{(\ell)}$ time units, with ℓ representing the number of $send$ st-events sending non-timer messages between the last $process$ st-event $\prec ev$ and the first $process$ st-event $\succ ev$. Intuitively, this corresponds to the number of non-timer messages sent by the action or job in the original execution.

2. moving $input$ st-events in tr arbitrarily far into the past[1] without changing their order with respect to other $input$ st-events.

None of these moving operations may violate causal dependency, i.e., the st-trace must be causally consistent with tr to be a valid μ^+-shuffle of tr. Causal dependency could be violated by changing the order of st-events occurring on the same processor or by causing messages to be processed before they have been sent (cf. Section 2.1). Since $gstates(tr)$ is a function of HC^{tr} and tr, $gstates(tr)$ changes during shuffling. Note that HC^{tr} is not modified by shuffling operations.

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

Observation 6.2. *The order of processor-local state transitions does not change, as otherwise causal dependency would be violated.*

[1] For the purpose of the proof of Theorem 6.11, this condition can be weakened. Let ev' be the event starting the busy period (= period, where $process$ st-events are at most μ^+ time units apart). As our model assumes a non-idling scheduler, it suffices to allow the $input$ st-event ev to be moved back to any time in the interval $[time(ev'), time(ev)]$.

Observation 6.3. *Let tr and $tr' \in$ shuffles(tr, μ^+) be st-traces. Let g be a global state in $gstates(tr)$ and p be a processor. There is a global state g' in $gstates(tr')$ with $time(g) \leq time(g') \leq time(g) + \mu^+$ such that $s_p(g) = s_p(g')$. Informally, this means that if a processor is in a certain state in a st-trace, it will be in the same state in a shuffled st-trace, but this state might be delayed by up to μ^+ time units.*

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

Definition 6.4. Let $\mathcal{P}$ be a st-problem, represented as a set of st-traces. Then $\mathcal{P}^*_{\mu^+}$ is defined as $\bigcup_{tr \in \mathcal{P}}$ *shuffles*(tr, μ^+). Informally speaking, $\mathcal{P}^*_{\mu^+}$ is equivalent to $\mathcal{P}$ with the exception that the problem is still solved if an arbitrary number of message sends and state transitions may happen up to μ^+ time units later (without violating causality) and external inputs arrive earlier.

Note that, as $\mathcal{P}$ is a subset of $\mathcal{P}^*_{\mu^+}$, $\mathcal{P}^*_{\mu^+}$ is a weaker problem than $\mathcal{P}$, i.e., if some algorithm solves $\mathcal{P}$ (in some system under some failure model), it also solves $\mathcal{P}^*_{\mu^+}$ (in the same system under the same failure model). In fact, for some st-problems, it even holds that $\forall \mu^+ : \mathcal{P}^*_{\mu^+} = \mathcal{P}$. We will call such st-problems *shuffle-compatible problems*, which informally means that they are invariant against time shifts.

6.1.2. Simulation-Invariant Extensions

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

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

- Every state occurring in the st-trace, i.e., $oldstate$ and $newstate$ of every $transition$ st-event, may be extended by variables from $\mathcal{V}$ (and their valuations).
- An arbitrary number of $process$ and $send$ st-events may be inserted, modified or removed.
- $transition$ st-events may be inserted as long as they do not modify any variables other than those in $\mathcal{V}$.
- The result must be a valid st-trace, e.g., every message processed must have been sent by a $send$ st-event or must originate from an $input$ st-event, and every $newstate(ev)$ must correspond to $oldstate(ev')$ of the following st-event on the same processor. Formally, some ex (satisfying EX1–EX6) or ru (satisfying RU1–RU8) must exist which can be mapped to tr using Definition 5.3 or 5.4.

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

6.1.3. Examples

All examples in this section are simulation-compatible.

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

We claim that an algorithm solving $\mathcal{P}^{*}_{\mu^{+}}$ with $\mu^{+} = 3$ seconds also solves *0-second gap mutual exclusion* (defined analogously). Looking ahead to Theorem 6.11, this means that a *3-second gap mutual exclusion* algorithm designed for a classic system can be used to solve the *0-second gap mutual exclusion* problem in some real-time system with $\mu^{+} = 3$ and the other parameters determined by the feasible assignment (cf. Section 6.2).

Proof. We will show by contradiction that an algorithm solving $\mathcal{P}^{*}_{\mu^{+}}$ solves the 0-gap mutual exclusion problem. Assume that there exists a rt-run ru with st-trace tr' satisfying $\mathcal{P}^{*}_{\mu^{+}}$ where *mutual exclusion* or *0s-gap* is violated, i.e., there is some time t in which two processors p and q are inside the critical section. This can happen either by both being inside the critical section in the same global state, thus violating the classical *mutual exclusion* condition, or by a zero-time st-event sequence $exit_p, \ldots, enter_q$ (w.l.o.g.), thus violating *0s-gap*.

As ru satisfies $\mathcal{P}^{*}_{\mu^{+}}$, $tr' \in \mathcal{P}^{*}_{\mu^{+}}$. By the definition of $\mathcal{P}^{*}_{\mu^{+}}$, this means that tr' is a 3-second shuffle of some st-trace $tr \in \mathcal{P}$. Thus, in tr, q is in the critical section at some time within $[t - 3s, t]$ and p is in the critical section at some (maybe other) time within $[t - 3s, t]$ (recall Observation 6.3). If p and q are in the critical section at the same global state in $gstates(tr)$, *mutual exclusion* is violated. Otherwise, one of them exits and the other one enters, causing the *3s-gap* condition to be violated. Both cases contradict the assumption that $\mathcal{P}$ solves *3-second gap mutual exclusion*.

Liveness I/II and *safety* in $\mathcal{P}^{*}_{\mu^{+}}$ follow directly from the same property in $\mathcal{P}$, as $enter$ and $exit$ st-events as well as local states are only moved forward w.r.t. tr (again, cf. Observation 6.3), whereas $want\_to\_enter$ and $want\_to\_exit$ st-events are only moved backwards w.r.t. tr. □

Causal Mutual Exclusion Let $\mathcal{P}$ be the *causal mutual exclusion* problem, defined by the properties in Section 5.2.4 and the additional requirement that every state transition in which a processor enters a critical section must causally depend on the last exit, formally $\forall ev, ev' \in tr : (ev = last(is\_exit, ev') \wedge is\_enter(ev')) \Rightarrow (ev \rightarrow ev')$, with $last(P, ev)$ denoting the last st-event ev' satisfying P with $ev' \prec ev$ (or $\perp$, if no such st-event exists).

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

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

In $\mathcal{P}$, the new *exit-enter causality* condition and the *mutual exclusion* condition imply that there is a causal sequence $enter_p \to exit_p \to enter_q \to exit_q \to \cdots$ containing *all* enter and exit st-events. Since shuffles must be causally consistent with the original st-trace, $enter_p \prec exit_p \prec enter_q \prec exit_q \prec \cdots$ still holds for all st-traces in $\mathcal{P}^*_{\mu^+}$, guaranteeing (a) that *mutual exclusion* is not violated in $\mathcal{P}^*_{\mu^+}$ and (b) that $last(is\_exit, ev')$ returns the same exit st-event in tr and tr' for each enter st-event ev'. Since shuffles neither change the processor-local order of st-events nor modify the messages, all causal dependencies (cf. Section 2.1) still exist in $\mathcal{P}^*_{\mu^+}$. Thus, *exit-enter causality* also holds in $\mathcal{P}^*_{\mu^+}$. □

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

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

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

aj-problems Every aj-problem can be specified as a st-problem with restrictions solely on $process$ and $input$ st-events (cf. Section 5.2.5) and on the local order of $process$, $send$ and $transition$ st-events (specified as $trans(ac^{aj})$). As $process$ st-events are not changed by shuffles and the local order of the aforementioned st-events does not change, every aj-problem whose input message restrictions are not violated by shifting $input$ st-events backwards in time is a shuffle-compatible problem.

6.2. Reusing Classic Computing Model Algorithms

In this section, we will show how to simulate a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ if the end-to-end delays bounds Δ^- and Δ^+ of the real-time system equal the message delay bounds $\underline{\delta}^-$ and $\underline{\delta}^+$ of the simulated classic system. Thereby, we provide a transformation of a classic computing model algorithm solving some problem $\mathcal{P}$ into a real-time computing model algorithm solving $\mathcal{P}^*_{\mu^+}$ (cf. Section 6.1.1).

6.2.1. Feasible Assignment

The key to this transformation is a very simple simulation: Recall that an algorithm is specified as a mapping from processor indices to a set of initial states and a transition function, and that the transition function is defined identically for the classic and the real-time computing model. Let $\mathcal{S}_{\underline{\mathcal{A}}}$ be an algorithm for the real-time computing model, comprising exactly the same initial states and transition function as a given classic computing model algorithm $\underline{\mathcal{A}}$. From a more practical point of view, $\mathcal{S}_{\underline{\mathcal{A}}}$ can be expressed as given in Figure 6.1.

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

procedure $\mathcal{S}_{\underline{\mathcal{A}}}$−process_message(msg, current_hc)
  $\underline{\mathcal{A}}$−process_message(msg, current_hc)
```

Figure 6.1.: Simulation algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ (classic computing model atop of real-time computing model)

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

Clearly, the end-to-end delay bounds of $\mathcal{S}_{\underline{\mathcal{A}}}$ are the result of some real-time schedulability analysis f of running $\mathcal{S}_{\underline{\mathcal{A}}}$ with some scheduling/admission policy pol under some failure model $\mathcal{C}$:

$$[\Delta^-, \Delta^+] = f\big(n, [\delta^-, \delta^+], [\mu^-, \mu^+], \mathcal{S}_{\underline{\mathcal{A}}}, pol, \mathcal{C}\big). \tag{6.1}$$

In turn, these end-to-end delay bounds specify the system parameters of the simulated classic system:

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

Since $\mathcal{S}_{\underline{\mathcal{A}}}$ depends on $\underline{\mathcal{A}}$ and $\underline{\mathcal{A}}$ might need to know the systems bounds $[\underline{\delta}^-, \underline{\delta}^+]$, this leads to a circular dependency with respect to $[\Delta^-, \Delta^+]$ in eq. (6.1).

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

$$[\Delta^-, \Delta^+] = F\big(n, [\delta^-, \delta^+], [\mu^-, \mu^+], [\Delta^-, \Delta^+]\big). \tag{6.2}$$

We do not want to embark on the intricacies of advanced real-time schedulability analysis techniques here, see [SAA$^+$04] for an overview. For the purpose of this work, quite simple 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. This technique is used in Chapter 7, whereas Section 8.2.3, on the other hand, presents a more complex example for determining Δ^+.

Anyway, if eq. (6.2) provided by the real-time schedulability analysis can be solved for $[\Delta^-, \Delta^+]$, resulting in meaningful bounds $\Delta^- \leq \Delta^+$, they can be assigned to the algorithm parameters $[\underline{\delta}^-, \underline{\delta}^+]$. We will call such an assignment *feasible*. Any feasible assignment of $[\underline{\delta}^-, \underline{\delta}^+]$ results in a correct implementation of the real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$, since it ensures that both $\underline{\mathcal{A}}$ and the end-to-end delays are within their specifications. Such a feasible assignment may not exist for some (real-time system, algorithm, scheduling/admission policy, failure model) tuples.

6.2.2. Scheduling/Admission Policy

Contrary to the classic computing model, running an algorithm in a real-time system requires a *scheduling/admission policy* (cf. Section 3.1), which not only determines the processing order of incoming messages but also allows messages to be dropped. For running $\mathcal{S}_{\underline{\mathcal{A}}}$, this policy pol can be arbitrary, as long as the following two conditions are satisfied:

- Only "irrelevant" messages are dropped when running $\mathcal{S}_{\underline{\mathcal{A}}}$ with pol in system s. More specifically, only messages that would have caused a job J with a NOP state transition ($trans(J) = [oldstate(J)]$) are allowed to be dropped. For example, in round-based algorithms, this could be messages from previous rounds or round k messages from a processor from which such a message has already been received (indicating a link or processor fault).

  Formally, pol and $\underline{\mathcal{A}}$ must satisfy the following condition: If pol drops a message m at hardware clock time T on a processor with state s (i.e., if $\exists Q, Q', next : pol(Q, s, T) = (next, Q') \wedge m \in Q \wedge m \neq next \wedge m \notin Q'$), then $\underline{\mathcal{A}}(m, s, T) = [s]$.

- Input messages must be processed in FIFO order. Formally, if input messages m_1 and m_2 are in the queue and m_1 has been received before m_2, then m_2 must not be dropped or processed before m_1 has been dropped or processed.

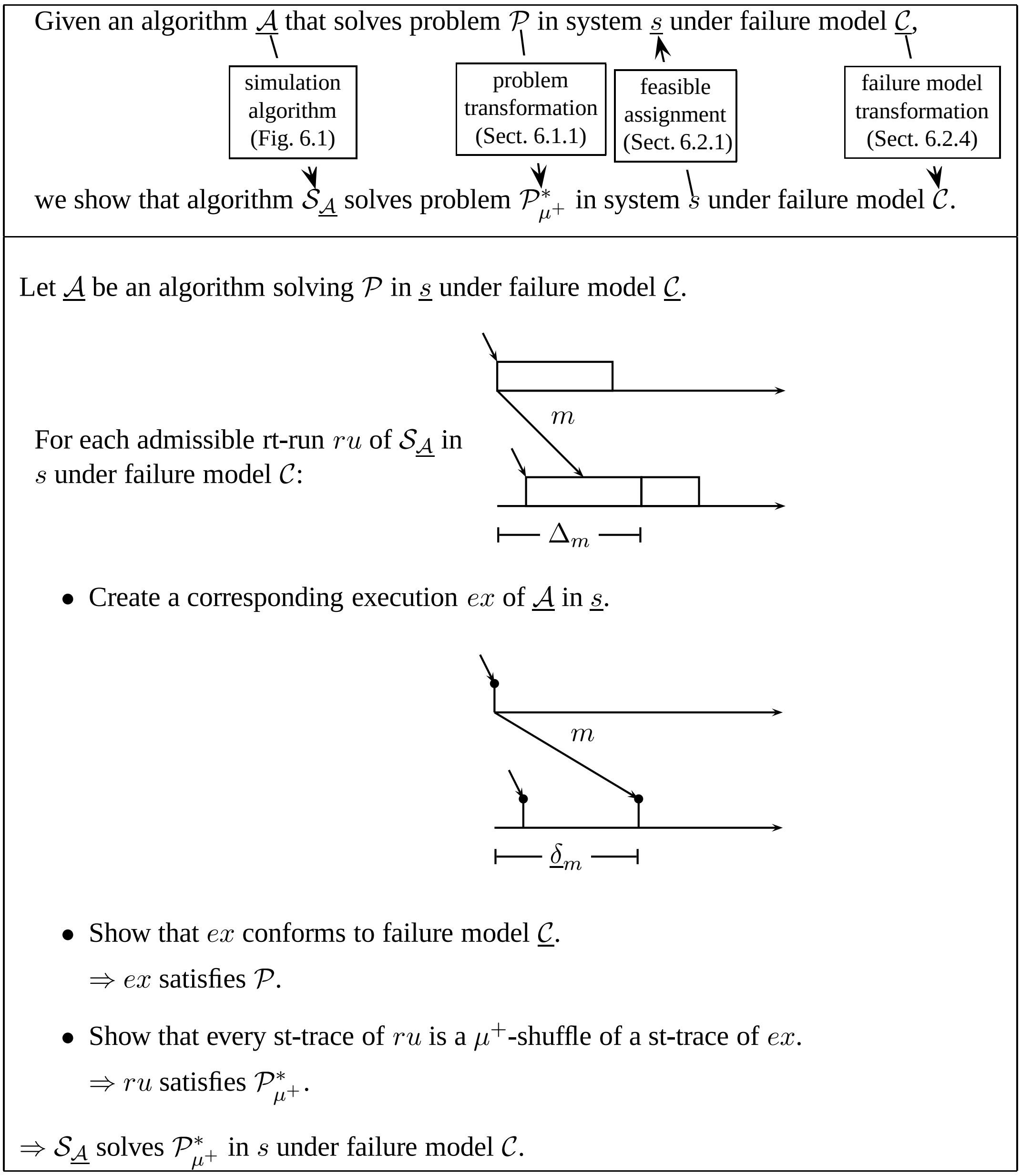
6.2.3. Transformation $T_{R \to C}$

As shown in the outline (Figure 6.2), the proof works by transforming every rt-run of $\mathcal{S}_{\underline{\mathcal{A}}}$ into a corresponding execution of $\underline{\mathcal{A}}$. By showing that (a) this execution is a valid execution of $\underline{\mathcal{A}}$ and (b) the execution and the rt-run have (roughly) the same state transitions, the fact that the rt-run satisfies $\mathcal{P}^*_{\mu^+}$ can be derived from the fact that the execution satisfies $\mathcal{P}$. This transformation, $ex = T_{R \to C}(ru)$, works by

- mapping each job J in ru to an action ac in ex:

$$proc(ac) \leftarrow proc(J) \qquad time(ac) \leftarrow begin(J) \qquad trans(ac) \leftarrow trans(J)$$

$$msg(ac) \leftarrow msg(J) \qquad HC(ac) \leftarrow HC(J)$$



Given an algorithm $\underline{\mathcal{A}}$ that solves problem $\mathcal{P}$ in system $\underline{s}$ under failure model $\underline{\mathcal{C}}$,

simulation algorithm (Fig. 6.1)
problem transformation (Sect. 6.1.1)
feasible assignment (Sect. 6.2.1)
failure model transformation (Sect. 6.2.4)

we show that algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ solves problem $\mathcal{P}^*_{\mu^+}$ in system s under failure model $\mathcal{C}$.

Let $\underline{\mathcal{A}}$ be an algorithm solving $\mathcal{P}$ in $\underline{s}$ under failure model $\underline{\mathcal{C}}$.

For each admissible rt-run ru of $\mathcal{S}_{\underline{\mathcal{A}}}$ in s under failure model $\mathcal{C}$:

- Create a corresponding execution ex of $\underline{\mathcal{A}}$ in $\underline{s}$.

- Show that ex conforms to failure model $\underline{\mathcal{C}}$.

  $\Rightarrow$ ex satisfies $\mathcal{P}$.

- Show that every st-trace of ru is a μ^+-shuffle of a st-trace of ex.

  $\Rightarrow$ ru satisfies $\mathcal{P}^*_{\mu^+}$.

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

Figure 6.2.: Transformation outline (Theorem 6.11)

- mapping each drop event D in ru to a NOP action ac in ex, with $state$ denoting the $newstate$ of the last job finishing on $p = proc(D)$ before D (or $istate_p^{ru}$, if there is no such job), and T being defined as $HC_p^{ru}(time(D))$:

$$proc(ac) \leftarrow proc(D) \qquad time(ac) \leftarrow time(D) \qquad trans(ac) \leftarrow [state]$$
$$msg(ac) \leftarrow msg(D) \qquad HC(ac) \leftarrow T$$

- setting $HC_p^{ex} = HC_p^{ru}$ for all p.

Receive events in ru are ignored.

Lemma 6.5. *If ru is a valid rt-run of $\mathcal{S}_{\underline{\mathcal{A}}}$, $ex = T_{R\to C}(ru)$ is a valid execution of $\underline{\mathcal{A}}$.*

Proof. EX1–6 (cf. Section 2.4) are satisfied in ex: EX1 follows from RU1 by ordering the actions like their corresponding jobs and drop events. EX2 follows from RU2 and the fact that the order of jobs in ru corresponds to the order of actions in ex, that the transition sequence is not changed and that the "correct" state is chosen for actions corresponding to drop events. EX3 is a direct consequence of RU3 and the fact that both ru and ex run the same algorithm (i.e. use the same initial state). Since ru and ex use the same hardware clocks, RU4 suffices to satisfy EX4. EX5 follows directly from RU5, and EX6 follows from RU6. Thus, ex is a valid execution of $\underline{\mathcal{A}}$. □

Lemma 6.6. *For every message m in ex, the message delay $\underline{\delta}_m$ is equal to the end-to-end delay $\Delta_{m'}$ of its corresponding message m' in ru.*

Proof. By construction of ex, the sending time of every message stays the same ($time(ac) = begin(J)$, with ac and J being the sending action/job; recall that message delays are measured from the start of the sending job rather than from the $send$ st-event). For dropped messages, the drop time in ru equals the receiving/processing time in ex ($time(ac) = time(D)$, with ac being the processing action and D being the drop event). For other messages, the processing time in ru equals the receiving/processing time in ex ($time(ac) = begin(J)$, with ac being the processing action and J being the processing job). □

6.2.4. Failure Model Compatibility

Since the failure model is dependent on the system model (classic or real-time), we need to establish a relationship between two failure models $\underline{\mathcal{C}}$ and $\mathcal{C}$, such that the following holds:

Definition 6.7. A failure model $\underline{\mathcal{C}}$ is called $T_{R\to C}$*-compatible* to $\mathcal{C}$, if the following holds for every rt-run ru of some algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ in some system s under failure model $\mathcal{C}$ with a scheduling/admission policy only dropping irrelevant messages (cf. Section 6.2.2): ex, the execution created by applying transformation $T_{R\to C}$ to ru, conforms to failure model $\underline{\mathcal{C}}$ in system $\underline{s}$, with $\underline{s}$ containing a feasible assignment w.r.t. s, $\underline{\mathcal{A}}$ and the chosen scheduling/admission policy.

This relationship needs to be shown for every (classic failure model, real-time failure model)-pair used in the transformation. As an example, we will prove compatibility for some variants of bounded-drift f-CRASH and f-BYZANTINE (and, thus, for FAULT-FREE = 0-CRASH = 0-BYZANTINE).

f-CRASH

First, we define two variants of f-CRASH. Differences to f-CRASH$_\rho(ex)$ and f-CRASH$_\rho(ru)$ (cf. Section 4.2.3) are underlined.

f-CRASH$_\rho$+latetimers$_\alpha(ex)$ *(classic computing model)* :⇔

$$\begin{aligned}
&\exists F : |F| = f \wedge F \subseteq \Pi \\
&\quad \wedge\ \forall m_o : is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+) \\
&\quad \wedge\ \forall m_t : arrives\_timely(m_t) \underline{\vee\ is\_late\_timer(m_t, \alpha)} \\
&\quad \wedge\ \forall ac : follows\_alg(ac)\ \vee\ [proc(ac) \in F \wedge ((is\_last(ac) \wedge follows\_alg\_partially(ac)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee\ arrives\_after\_crash(ac))] \\
&\quad \wedge\ \forall p : bounded\_drift(p, \rho)
\end{aligned}$$

f-CRASH$_\rho$+precisetimers$_\alpha(ru)$ *(real-time computing model)* :⇔

$$\begin{aligned}
&\exists F : |F| = f \wedge F \subseteq \Pi \\
&\quad \wedge\ \forall m_o : is\_timely\_msg(m_o, \delta^-, \delta^+) \\
&\quad \wedge\ \forall m_t : arrives\_timely(m_t) \\
&\quad \wedge\ \forall R : obeys\_pol(R)\ \vee\ [proc(R) \in F \wedge arrives\_after\_crash(R) \wedge drops\_msg(R)] \\
&\quad \wedge\ \forall J : obeys\_pol(J)\ \vee\ [proc(J) \in F \wedge is\_last(J) \wedge drops\_all\_queued(J)] \\
&\quad \wedge\ \forall J : follows\_alg(J)\ \vee\ [proc(J) \in F \wedge is\_last(J) \wedge follows\_alg\_partially(J)] \\
&\quad \wedge\ \forall J : is\_timely\_job(J, \mu^-, \mu^+) \\
&\quad \wedge\ \underline{\forall m_t : gets\_processed\_precisely(m_t, \alpha)} \\
&\quad \wedge\ \forall p : bounded\_drift(p, \rho)
\end{aligned}$$

$$\begin{aligned}
is\_late\_timer(m_t, \alpha) :\Leftrightarrow\ &\exists ac, ac' : m_t \in trans(ac) \wedge m_t = msg(ac') \\
&\wedge time(ac') \in HC^{-1}_{proc(m_t)}(sHC(m_t)) + [0, \alpha]
\end{aligned}$$

$$\begin{aligned}
&gets\_processed\_precisely(m_t, \alpha) :\Leftrightarrow \\
&\quad \exists JD : msg(JD) = m_t \wedge time(JD) \in HC^{-1}_{proc(m_t)}(sHC(m_t)) + [0, \alpha]
\end{aligned}$$

As a reminder: f-CRASH$_\rho(ex)$ and f-CRASH$_\rho(ru)$ require every timer m_t to arrive at the designated hardware clock time $sHC(m_t)$ (= the time for which m_t is set or the hardware clock time at the beginning of the job setting the timer) or at the hardware clock time at the end of the job setting m_t, whichever happens later (cf. Section 4.1). In the classic computing model, they must be processed immediately; in the real-time computing model, they are allowed to be queued.

f-CRASH$_\rho$+precisetimers$_\alpha(ru)$ *requires* that timers *start processing* at most α time units after their designated time. On the other hand, f-CRASH$_\rho$+latetimers$_\alpha(ex)$ *allows* timers to *arrive* up to α real-time units later than their designated time.

Note that for all ru and all $\alpha \geq 0$, f-CRASH$_\rho$+precisetimers$_\alpha(ru) \Rightarrow f$-CRASH$_\rho(ru)$, and that for all ex and all $\alpha \geq 0$, f-CRASH$_\rho(ex) \Rightarrow f$-CRASH$_\rho$+latetimers$_\alpha(ex)$.

The following lemma shows that a classic computing model algorithm designed for the f-CRASH failure model with bounded drift and tolerating late timer arrival by at most α can be used in a real-time system under the condition that queuing effects delay the processing of timers by at most α time units.

Lemma 6.8. *f-CRASH$_\rho$+latetimers$_\alpha(ex)$ is $T_{R\to C}$-compatible to f-CRASH$_\rho$+precisetimers$_\alpha(ru)$.*

Proof. Let F denote the same set of processors in both f-CRASH$_\rho$+latetimers$_\alpha(ex)$ and f-CRASH$_\rho$+precisetimers$_\alpha(ru)$. We can show that $ex = T_{R\to C}(ru)$ (according to Definition 6.7) satisfies f-CRASH$_\rho$+latetimers$_\alpha(ex)$, if ru satisfies f-CRASH$_\rho$+precisetimers$_\alpha(ru)$.

- $|F| = f \wedge F \subseteq \Pi$

  Follows from the existence of such a set F in f-CRASH$_\rho$+precisetimers$_\alpha(ru)$.

- $\forall m_o : is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+)$

  Follows from Lemma 6.6 and the fact that Definition 6.7 assumes a feasible assignment (i.e. $[\underline{\delta}^-, \underline{\delta}^+] = [\Delta^-, \Delta^+]$).

- $\forall m_t : arrives\_timely(m_t) \vee is\_late\_timer(m_t, \alpha)$

  Let t denote $HC^{-1}_{proc(m_t)}(sHC(m_t))$, i.e., the real time by which timer m_t should arrive. $gets\_processed\_precisely(m_t, \alpha)$ ensures that the job or drop event taking care of m_t starts at most α real time units after t. Due to the transformation rules of $T_{R\to C}$, this job or drop event is transformed into an action ac receiving and processing m_t and occurring at the same time as the job or drop event. Thus, $is\_late\_timer(m_t, \alpha)$ is satisfied.

- $\forall ac$: either
  (a) $follows\_alg(ac)$ or
  (b) $proc(ac) \in F \wedge is\_last(ac) \wedge follows\_alg\_partially(ac)$ or
  (c) $proc(ac) \in F \wedge arrives\_after\_crash(ac)$

  Let $F' \subseteq F$ be the set of processors actually crashing (or terminating) in ru, i.e., the set of processors p for which some job J_p^{last} with $is\_last(J_p^{last})$ exists.

  - *Non-faulty processors:* For $p \notin F'$, all jobs in ru on p adhere to the algorithm. The corresponding actions in ex occur at the same hardware clock time, process the same message and have the same state transition sequence. Thus, (a), $follows\_alg(ac)$, holds for them as well.

    W.r.t. drop events, we defined, for the purposes of this transformation, that only messages that would have caused a NOP state transition may be dropped by pol (cf. Section 6.2.2). Due to the $\forall R/J : obeys\_pol(R)/(J)$ conditions and RU8, drop events occurring on non-faulty processors must conform to pol. "Would have caused a NOP state transition" means that the algorithm returns a NOP state transition for the current (message, hardware clock, state) tuple. Thus, the action ac corresponding to this drop event satisfies (a), $follows\_alg(ac)$.

  - *Before the processor crashes:* For $p \in F'$, the same arguments hold for all jobs $J \prec J_p^{last}$ on p and all drop events before J_p^{last}. Thus, (a) also holds for their corresponding actions.

  - *During the crash:* For $J = J_p^{last}$, the definition of $follows\_alg\_partially(ac)/(J)$ directly translates to the corresponding action ac_p^{last}. Since there are no jobs $J \succ J_p^{last}$ on p, only actions based on drop events can occur in p after ac_p^{last}, causing ac_p^{last} to satisfy $is\_last(ac_p^{last})$. Thus, ac_p^{last} satisfies (b).
  - *After the processor crashes:* By definition of $is\_last(J)$, no jobs occur in ru after a processor has crashed. Drop events occurring after a processor has crashed need not (and usually will not) obey the scheduling policy: Messages received and queued before the last job are dropped directly after that job (see predicate $drops\_all\_queued(J)$), and messages received afterwards are dropped immediately (see predicate $arrives\_after\_crash(R)$). Since $ac_p^{last} \prec ac$ holds for all actions ac corresponding to such drop events (on some processor p), (c), $arrives\_after\_crash(ac)$, is satisfied.

- $\forall p : bounded\_drift(p, \rho)$

  Follows from the equivalent condition in f-CRASH$_\rho$+precisetimers$_\alpha(ru)$ and the fact that $T_{R\to C}$ ensures that $HC_p^{ex} = HC_p^{ru}$ for all p. □

Basically, the choice of α allows to put the burden either on the scheduler in the real-time system (low α, timers must be scheduled early) or on the algorithm of the classic system (high α, the algorithm must tolerate delayed timers). Note that $is\_late\_timer(m_t, 0) = arrives\_timely(m_t)$, and, thus, f-CRASH$_\rho$+latetimers$_0(ex) = f$-CRASH$_\rho(ex)$. Likewise, we can extend the domain of α with ∞ by defining $is\_late\_timer(m_t, \infty) :\Leftrightarrow true$ and $gets\_processed\_precisely(m_t, \infty) :\Leftrightarrow true$. Thus, f-CRASH$_\rho$+precisetimers$_\infty(ru) = f$-CRASH$_\rho(ru)$, and it is plain to see that Lemma 6.8 still holds for $\alpha = \infty$.

Observation 6.9. *f-CRASH$_\rho(ex)$ is $T_{R\to C}$-compatible to f-CRASH$_\rho$+precisetimers$_0(ru)$, and f-CRASH$_\rho$+latetimers$_\infty(ex)$ is $T_{R\to C}$-compatible to f-CRASH$_\rho(ru)$.*

f-BYZANTINE

We define f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$ and f-BYZANTINE$_\rho$+latetimers$_\alpha(ex)$ analogous to their crash failure counterparts:

f-BYZANTINE$_\rho$+latetimers$_\alpha(ex)$ ***(classic computing model)*** $:\Leftrightarrow$
$\exists F : |F| = f \wedge F \subseteq \Pi$
  $\wedge\ \forall m_o : is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+)$
  $\wedge\ \forall m_t : arrives\_timely(m_t)\ \underline{\vee\ is\_late\_timer(m_t, \alpha)} \vee [proc(m_t) \in F]$
  $\wedge\ \forall ac : follows\_alg(ac) \vee [proc(ac) \in F]$
  $\wedge\ \forall p : bounded\_drift(p, \rho) \vee [p \in F]$

f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$ ***(real-time computing model)*** $:\Leftrightarrow$
$\exists F : |F| = f \wedge F \subseteq \Pi$
  $\wedge\ \forall m_o : is\_timely\_msg(m_o, \delta^-, \delta^+)$
  $\wedge\ \forall m_t : arrives\_timely(m_t) \vee [proc(m_t) \in F]$

$\wedge\ \forall R : \mathit{obeys\_pol}(R) \vee [\mathit{proc}(R) \in F]$
$\wedge\ \forall J : \mathit{obeys\_pol}(J) \vee [\mathit{proc}(J) \in F]$
$\wedge\ \forall J : \mathit{follows\_alg}(J) \vee [\mathit{proc}(J) \in F]$
$\wedge\ \forall J : \mathit{is\_timely\_job}(J, \mu^-, \mu^+) \vee [\mathit{proc}(J) \in F]$
$\wedge\ \underline{\forall m_t : \mathit{gets\_processed\_precisely}(m_t, \alpha) \vee [\mathit{proc}(m_t) \in F]}$
$\wedge\ \forall p : \mathit{bounded\_drift}(p, \rho) \vee [p \in F]$

Lemma 6.10. *f-BYZANTINE$_\rho$+latetimers$_\alpha(ex)$ is $T_{R\to C}$-compatible to failure model f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$.*

Proof. Let F denote the same set of processors in both f-BYZANTINE$_\rho$+latetimers$_\alpha(ex)$ and f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$. It can be shown that ex, the $T_{R\to C}$ transformation of ru according to Definition 6.7, satisfies f-BYZANTINE$_\rho$+latetimers$_\alpha(ex)$, if ru satisfies f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$.

- $|F| = f \wedge F \subseteq \Pi$

  Follows from the existence of such a set F in f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$.

- $\forall m_o : \mathit{is\_timely\_msg}(m_o, \underline{\delta}^-, \underline{\delta}^+)$

  Follows from Lemma 6.6 and the fact that Definition 6.7 assumes a feasible assignment (i.e. $[\underline{\delta}^-, \underline{\delta}^+] = [\Delta^-, \Delta^+]$).

- $\forall m_t : \mathit{arrives\_timely}(m_t) \vee\ \mathit{is\_late\_timer}(m_t, \alpha) \vee [\mathit{proc}(m_t) \in F]$

  For $\mathit{proc}(m_t) \in F$, this condition is satisfied trivially. For timer messages on correct processors, $\mathit{is\_late\_timer}(m_t, \alpha)$ follows from $\mathit{gets\_processed\_precisely}(m_t, \alpha)$ by the same reasoning as in Lemma 6.8.

- $\forall ac : \mathit{follows\_alg}(ac) \vee [\mathit{proc}(ac) \in F]$

  Again, for $\mathit{proc}(ac) \in F$, this condition is satisfied trivially; for correct processors, the proof follows the same line of reasoning as the "non-faulty processors" part of Lemma 6.8.

- $\forall p : \mathit{bounded\_drift}(p, \rho) \vee [p \in F]$

  This follows from the equivalent condition in f-BYZANTINE$_\rho$+precisetimers$_\alpha(ru)$ and the fact that $T_{R\to C}$ ensures that $HC_p^{ex} = HC_p^{ru}$ for all p. □

6.2.5. Transformation Proof

Theorem 6.11. *Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system, pol be a scheduling/admission policy and $\mathcal{P}$ be a problem. If*

- *there exists an algorithm $\underline{\mathcal{A}}$ for solving $\mathcal{P}$ in some classic system $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ under some failure model $\underline{\mathcal{C}}$ **[A1]**[2],*

[2] To aid the reader in following the arguments of this proof, we will label assumptions, definitions and lemmas used solely in this proof in bold face, e.g. **[A1]**/**[D1]**/**[L1]**, and reference them in parenthesis, e.g. ([A1])/([D1])/([L1]).

- *$\underline{s}$ contains a feasible assignment w.r.t. $\mathcal{S}_{\underline{\mathcal{A}}}$, s and pol (cf. Section 6.2.1)* ***[A2]****,*
- *scheduling/admission policy pol only drops irrelevant messages* ***[A3]*** *and ensures that input messages are processed in FIFO order* ***[A4]*** *(cf. Section 6.2.2), and*
- *$\underline{\mathcal{C}}$ is $T_{R\to C}$*-compatible *to some real-time failure model $\mathcal{C}$ (cf. Section 6.2.4)* ***[A5]****,*

then $\mathcal{S}_{\underline{\mathcal{A}}}$ given in Figure 6.1 solves $\mathcal{P}^*_{\mu^+}$ in s under failure model $\mathcal{C}$ with scheduling/admission policy pol ***[GOAL]****.*

Proof. Let ru be an rt-run of $\mathcal{S}_{\underline{\mathcal{A}}}$ in s under failure model $\mathcal{C}$ with scheduling/admission policy pol **[D1]**. By Lemma 6.5, $ex = T_{R\to C}(ru)$ is a valid execution of $\underline{\mathcal{A}}$ **[L1]**. Since $\underline{\mathcal{C}}$ is $T_{R\to C}$-compatible to $\mathcal{C}$ ([A5]), $\underline{s}$ contains a feasible assignment ([A2]) and pol only drops irrelevant messages ([A3]), Definition 6.7 ensures that ex conforms to failure model $\underline{\mathcal{C}}$ in $\underline{s}$ **[L2]**.

As $\underline{\mathcal{A}}$ is an algorithm solving $\mathcal{P}$ in $\underline{s}$ under failure model $\underline{\mathcal{C}}$ ([A1]) and ex is a valid execution of $\underline{\mathcal{A}}$ ([L1]) conforming to failure model $\underline{\mathcal{C}}$ in $\underline{s}$ ([L2]), ex satisfies $\mathcal{P}$ (cf. Definition 5.10) **[L3]**.

To show that ru satisfies $\mathcal{P}^*_{\mu^+}$, we must show that every st-trace tr' of ru is a μ^+-shuffle of a st-trace tr of ex. Let tr' be a st-trace of ru **[D2]**. We can construct tr from tr' as follows:

- Move the time of every $send$ and $transition$ st-event back to the time of their corresponding $process$ st-event. The $send$ and $transition$ st-events belonging to the same job should directly follow their $process$ st-event and the order of these $process$, $send$ and $transitions$ st-events must not change (of course, the order w.r.t. st-events of other jobs will change). tr is still causally consistent with tr' (see Sections 2.1 and 5.2.2), as the processor-local order of st-events is not changed, $process$ st-events are not moved and $send$ st-events are only moved backwards in time.
- Move the time of every $input$ st-event forward so that it has the same time as its corresponding $process$ st-event processing the input message. The $input$ st-event must directly precede the $process$ st-event. Clearly, this does not violate causal consistency with tr' either.

Since pol ensures that input messages are processed in FIFO order ([A4]), the above operations are an inverse subset of the μ^+-shuffle operations (see Definition 6.1); thus, tr' is a μ^+-shuffle of tr **[L4]**. Still, we need to show that tr is a st-trace of ex (cf. Definition 5.3):

- *Every action in ex is correctly mapped to st-events in tr:* Every job J in ru is mapped to an action ac in ex and a sequence of one $process$, multiple $send/transition$ and at most one $input$ st-event in tr. Following Definitions 5.3 and 5.4, there are two differences in the mapping of some job J to st-events and the corresponding action ac to st-events:
  - The $process$, $state$ and $transition$ st-events all occur at the same time $time(ac)$ when mapping an action. The construction of tr ensures that this is the case.

  - If $msg(ac)$ is an input message, the corresponding $input$ st-event occurs at the same time as the $process$ st-event processing it. Since ru satisfies RU6, there is also such an $input$ st-event in tr', and, thus, in tr. The construction of tr ensures that this $input$ st-event has the correct position in tr.

  Every drop event D in ru is mapped to a NOP action ac, i.e., an action with $trans(ac) = [s]$, $s := oldstate(ac) = newstate(ac)$, in ex. Both D and ac get mapped to the same single $process$ st-event, without any following $send$ or $transition$ st-events. If the dropped message was an input message, the same reasoning as above applies w.r.t. the $input$ st-event.

- *Every st-event in tr belongs to an action in ex:* Every st-event in tr' (and, thus, every corresponding st-event in tr) is based on either a job, an input message receive event or a drop event in ru. By construction of ex, every job and every drop event is mapped to one action, requiring the same amount of $process$, $send$ and $transition$ st-events. Every input message receive event in ru results in an $input$ st-event. By Definition 5.3, this $input$ st-event belongs to the action processing it.

Thus, we can conclude that tr is a st-trace of ex **[L5]**. As $\underline{\mathcal{A}}$ solves $\mathcal{P}$ in $\underline{s}$ under failure model $\underline{\mathcal{C}}$ ([A1]) and ex is an execution of $\underline{\mathcal{A}}$ in $\underline{s}$ under $\underline{\mathcal{C}}$ ([L2]), this ([L5]) implies that $tr \in \mathcal{P}$ (cf. Definition 5.10) **[L6]**. Since tr' is a μ^+-shuffle of tr ([L4]) and $tr \in \mathcal{P}$ ([L6]), Definition 6.4 states that $tr' \in \mathcal{P}^*_{\mu^+}$ **[L7]**.

As this ([L7]) holds for every st-trace tr' of every rt-run ru of $\mathcal{S}_{\underline{\mathcal{A}}}$ in s under failure model $\mathcal{C}$ with scheduling/admission policy pol ([D1, D2]), Definition 5.12 states that $\mathcal{S}_{\underline{\mathcal{A}}}$ solves $\mathcal{P}^*_{\mu^+}$ in s under failure model $\mathcal{C}$ with scheduling/admission policy pol ([GOAL]). □

6.3. Reusing Real-Time Computing Model Algorithms

As the real-time computing model is a generalization of the classic computing model, the set of systems covered by the classic computing model is a (strict) subset of the systems covered by the real-time computing model. More precisely, every system in the classic computing model $(n, [\underline{\delta}^-, \underline{\delta}^+])$ can be specified in terms of the real-time computing model $(n, [\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+], [\mu^- = 0, \mu^+ = 0])$. Thus, every result (correctness or impossibility) for some classic system also holds in the corresponding real-time system with the same message delay bounds, $\mu^-_{(\ell)} = \mu^+_{(\ell)} = 0$ for all ℓ, and an admission control component that does not drop any messages. Intuition tells us that impossibility results also hold for the general case, i.e., that an impossibility result for some classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ holds for all real-time systems $(n, [\delta^- \leq \underline{\delta}^-, \delta^+ \geq \underline{\delta}^+], [\mu^-, \mu^+])$ for arbitrary μ^-, μ^+ as well, because the additional delay does not provide the algorithm with any useful information.

For simulation-compatible problems (recall Section 6.1.2) this conjecture is true, unless the system has a very inaccurate hardware clock and there is very little uncertainty in the processing delay, i.e., $\mu^+ - \mu^-$ is very low. In that case, timing information might be gained from the processing delay, for example, by increasing a local variable by $(\mu^- + \mu^+)/2$ during each computing step. If the precision of this logical clock exceeds the one of the hardware clock, algorithms might in fact benefit from the processing delay as opposed to the zero step-time situation. Thus, this section will concentrate on systems where this cannot happen. In particular, we will assume that the hardware clock is “accurate enough” to estimate a real-time value within $[\mu^-, \mu^+]$.

We will provide a formal general transformation of impossibility results from the classic to the real-time computing model by using yet another simulation, this time in the other direction. Although the simulation algorithm is slightly more complex than the one in the previous section, we do not require a schedulability analysis to obtain a feasible assignment here (since both $[\underline{\delta}^-, \underline{\delta}^+]$ and $[\delta^-, \delta^+]$ are system parameters), and the problem transformation $\mathcal{P} \to \mathcal{P}^{>}_{\mathcal{V}}$ is much less restrictive than $\mathcal{P} \to \mathcal{P}^{*}_{\mu^+}$.

```
var queue ← $empty$
var idle ← $true$
< local state (= global variables of $\mathcal{A}$)>

procedure $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$−process_message(msg, current_hc)

  if msg ≠ (FINISHED-PROCESSING)                 /* type (a), (b) and (e) */
    queue.add(msg)

  if idle or msg = (FINISHED-PROCESSING)         /* type (a), (c), (d) and (e) */

    var next; /* apply scheduling/admission policy */
    (next, queue) ← $pol$(queue, <local state>, current_hc)

    if next = $\perp$                             /* type (d) and (e) */
      idle ← $true$
    else                                          /* type (a) and (c) */
      idle ← $false$
      $\mathcal{A}$−process_message(next, current_hc)
      $\ell$←number of ordinary messages sent by $\mathcal{A}$
      set timer (FINISHED-PROCESSING) for current_hc + $\tilde{\mu}_{(\ell)}$
```

Figure 6.3.: Simulation algorithm $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ (real-time computing model atop of classic computing model)

6.3.1. Algorithm

Figure 6.3 provides an algorithm $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ designed for the classic computing model, which allows us to simulate a real-time system, and, thus, to use an algorithm $\mathcal{A}$ designed for the real-time computing model to solve problems in a classic system. The algorithm essentially simulates queuing, scheduling, and execution of real-time model computing steps (jobs) of (hardware clock time) duration $\tilde{\mu}$, and can hence be parameterized with some function $\tilde{\mu} : \{0, \dots, n-1\} \to \mathbb{R}^+$, some real-time computing model algorithm $\mathcal{A}$ and some scheduling/admission policy pol. We define that $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ has the same initial states as $\mathcal{A}$, with additionally $queue = empty$ and $idle = true$.

Figure 6.4 outlines the five main types of state transitions (a)–(e) in the simulation algo-

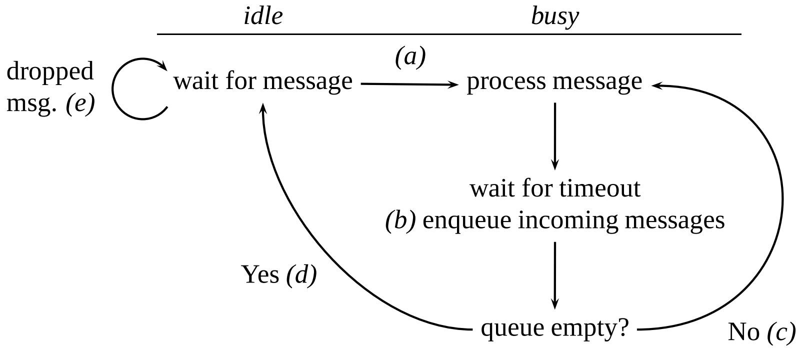


Figure 6.4.: State diagram (algorithm in Figure 6.3)

rithm: At every point in time, the simulated processor is either *idle* (local variable $idle = true$) or *busy* ($idle = false$). Initially, the processor is idle. As soon as the first algorithm message[3] arrives [type (a) action], the processor becomes busy and waits for $\tilde{\mu}_{(\ell)}$ hardware clock time units[4] (unless the message gets dropped by the scheduling/admission policy immediately [type (e) action], which means that the processor stays idle). All algorithm messages arriving while the processor is busy are enqueued [type (b) action]. After these $\tilde{\mu}_{(\ell)}$ hardware clock time units have passed (modeled as a (FINISHED-PROCESSING) timer message arriving), the queue is checked and a scheduling/admission decision is made (possibly dropping messages). If it is empty, the processor returns to its idle state [type (d) action]; otherwise, the next message is processed [type (c) action].

Note that this transformation requires the hardware clocks to be "sufficiently accurate", i.e., waiting for $\tilde{\mu}_{(\ell)}$ hardware clock units must always result in a (possibly varying) real-time delay between $\mu^-_{(\ell)}$ and $\mu^+_{(\ell)}$. Formally:

Definition 6.12. Let d^- and d^+ ($d^- \leq d^+$) be real-time durations, and let D be a hardware clock time duration. A hardware clock HC_p can "*estimate* $[d^-, d^+]$ *by* D", if it holds that $\forall T : HC_p^{-1}(T + D) - HC_p^{-1}(T) \in [d^-, d^+]$. In particular, we say that a hardware clock HC_p can "*estimate* $[\mu^-, \mu^+]$ *by* $\tilde{\mu}$", if, for every $\ell, 0 \leq \ell \leq n-1$, HC_p can estimate $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$ by $\tilde{\mu}_{(\ell)}$.

6.3.2. Failure Model Requirements

Due to its extreme simplicity, the simulation algorithm for $T_{R\to C}$ ($\mathcal{S}_{\underline{\mathcal{A}}}$, see Section 6.2) did not impose any restrictions on the failure model. This allowed us to pursue a "modular approach" and describe most of the proof as generally as possible (Sections 6.2.3 and 6.2.5), with only a small part dependent on the specific failure model (Section 6.2.4).

[3] In this section, we will split the set of messages into "algorithm messages" (sent because they are specified in algorithm $\mathcal{A}$) and (FINISHED-PROCESSING) messages (required internally by the simulation algorithm).

[4] ℓ being the number of ordinary messages sent during that computing step, cf. Section 3.3.

With respect to the transformation $T_{C\to R}$ presented below, $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ (see Figure 6.3) is non-trivial and simulates the CPU, the scheduler and the admission control component of a real-time system. Note, however, that some parts of this system are always assumed to be fault-free in a real-time system as defined in Chapter 3: For example, in a valid rt-run, messages are not allowed to "appear out of nowhere" (= condition RU6), independent of the failure model; in addition, state transitions are only allowed during jobs (= condition RU2). In $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$, however, an execution in which some message appears in $queue$ which has not been received, or an execution in which some algorithm $\mathcal{A}$ variable changes during a type (e) action (which is not mapped to a job, see below) would both be perfectly valid executions in the f-BYZANTINE failure model. Nevertheless, the corresponding simulated rt-run would not be a valid rt-run, since it violates RU2 or RU6.

Thus, for this transformation, we assume that $\underline{\mathcal{C}}$, the failure model in which $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ is executed, is at least as restrictive as FAULT-FREE$_\rho(ex)$, and that ρ is small enough (in other words: the processing delay uncertainty is large enough) such that some value within $[\mu^-, \mu^+]$ can be estimated. The following lemma generalizes the relationship between clock drift and the ability to estimate values:

Lemma 6.13. *Let d^- and d^+ ($d^- \leq d^+$) be real-time durations, and let D be a hardware clock-time duration. If*

$$\text{(a)}\qquad \rho \leq \frac{d^+ - d^-}{d^+ + d^-} \qquad\qquad \textit{and} \qquad\qquad \text{(b)}\qquad D = 2\frac{d^+ d^-}{d^+ + d^-}, \tag{6.3}$$

then all hardware clocks whose drift is bounded by ρ estimate $[d^-, d^+]$ by D (according to Definition 6.12).

Proof. Let HC_p be a hardware clock with bounded drift ρ. By definition of $bounded\_drift$ (cf. Sec. 4.2.2),

$$(1+\rho) \geq \frac{HC_p(t) - HC_p(t')}{t - t'} \geq (1-\rho) \qquad \forall t > t' \geq 0.$$

Since hardware clocks in executions and rt-runs are strictly increasing, continuous and unbounded, HC_p is an invertible function and this can be rewritten as

$$\frac{1}{1+\rho} \leq \frac{HC_p^{-1}(T) - HC_p^{-1}(T')}{T - T'} \leq \frac{1}{1-\rho} \qquad \forall T > T' \geq HC_p(0);$$

in particular,

$$\frac{D}{1+\rho} \leq HC_p^{-1}(T+D) - HC_p^{-1}(T) \leq \frac{D}{1-\rho} \qquad \forall T \geq HC_p(0).$$

Applying part (b) and then part (a) of eq. (6.3) results in

$$\frac{2\frac{d^+d^-}{d^++d^-}}{1+\frac{d^+-d^-}{d^++d^-}} \leq \frac{2\frac{d^+d^-}{d^++d^-}}{1+\rho} \leq HC_p^{-1}(T+D) - HC_p^{-1}(T) \leq \frac{2\frac{d^+d^-}{d^++d^-}}{1-\rho} \leq \frac{2\frac{d^+d^-}{d^++d^-}}{1-\frac{d^+-d^-}{d^++d^-}},$$

which can be simplified to

$$d^- \leq HC_p^{-1}(T+D) - HC_p^{-1}(T) \leq d^+. \qquad \square$$

Corollary 6.14. *If, for all ℓ, $\rho \leq \frac{\mu^+_{(\ell)} - \mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$ and $\tilde{\mu}_{(\ell)} := 2\frac{\mu^+_{(\ell)}\mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$, then all hardware clocks whose drift is bounded by ρ estimate $[\mu^-, \mu^+]$ by $\tilde{\mu}$.*

Note that FAULT-FREE$_\rho(ex)$, i.e., a failure model with “bounded-drift clocks”, has been chosen for ease of presentation. In fact, any other type of hardware clock guaranteeing that there exists some $\tilde{\mu}_{(\ell)}$ such that Definition 6.12 is satisfied is sufficient.

6.3.3. Algorithm Properties

All actions occurring within an execution ex of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ under failure model FAULT-FREE$_\rho(ex)$ fall into one of the five groups illustrated in Figure 6.4:

(a) algorithm message arriving which is immediately processed,

(b) algorithm message arriving which is enqueued,

(c) (FINISHED-PROCESSING) timer message arriving, causing some message from the queue to be processed,

(d) (FINISHED-PROCESSING) timer message arriving when no messages are in the queue (or all messages in the queue get dropped),

(e) algorithm message arriving which is immediately dropped.

The following can be asserted for every such execution:

Observation 6.15. *Every type (c) action has a corresponding type (b) action where the algorithm message being processed in the type (c) action (Line 19) is enqueued (Line 8). More generally, every message removed from $queue$ by pol in a type (c) or (d) action has been received before by a corresponding type (b) action.*

Observation 6.16. *Every type (a) and every type (c) action sending ℓ ordinary messages also sends one* (FINISHED-PROCESSING) *timer message which arrives $\tilde{\mu}_{(\ell)}$ hardware clock time units later (Line 21).*

Lemma 6.17. *Initially and directly after executing some action ac, processor $p = proc(ac)$ is in one of two well-defined states:*

- *State 1* ***(idle)****: $newstate(ac).idle = true$, $newstate(ac).queue = empty$, there is no* (FINISHED-PROCESSING) *timer message to p in $intransit\_msgs(ac)$*[5]*,*

- *State 2* ***(busy)****: $newstate(ac).idle = false$, there is exactly one* (FINISHED-PROCESSING) *timer message to p in $intransit\_msgs(ac)$.*

[5] Recall from Section 2.4 that $intransit\_msgs(ac)$ denotes the set of messages in transit after ac has completed.

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

- The message is a (FINISHED-PROCESSING) timer message. If the queue was empty or all messages got dropped (Line 15; recall that $next = \bot$ implies $queue = empty$ due to our non-idling assumption, cf. Section 3.1), the processor switches to state 1 [type (d) action]. Otherwise, a new (FINISHED-PROCESSING) timer message is generated. Thus, the processor stays in state 2 [type (c) action].

- The message is an algorithm message. The message is added to the queue and the processor stays in state 2 [type (b) action]. □

The following observation follows directly from this lemma and the design of the algorithm:

Observation 6.18. *Type (a) and (e) actions can only occur in idle state, type (b), (c) and (d) actions only in busy state. Type (a) and (d) actions change the state (from idle to busy and from busy to idle, respectively), all other actions keep the state (see Figure 6.4).*

Lemma 6.19. *After a type (a) or (c) action ac sending ℓ ordinary messages occurred at hardware clock time T on processor p in ex, the next type (a), (c), (d) or (e) action on p can occur no earlier than at hardware clock time $T + \tilde{\mu}_{(\ell)}$, when the (FINISHED-PROCESSING) message sent by ac has arrived.*

Proof. Since ac is a type (a) or (c) action, $newstate(ac).idle = false$, which, by Lemma 6.17, cannot change until no more (FINISHED-PROCESSING) messages are in transit. By Observation 6.16, this cannot happen earlier than at hardware clock time $T + \tilde{\mu}_{(\ell)}$. Lemma 6.17 also states that no second (FINISHED-PROCESSING) message can be in transit simultaneously.

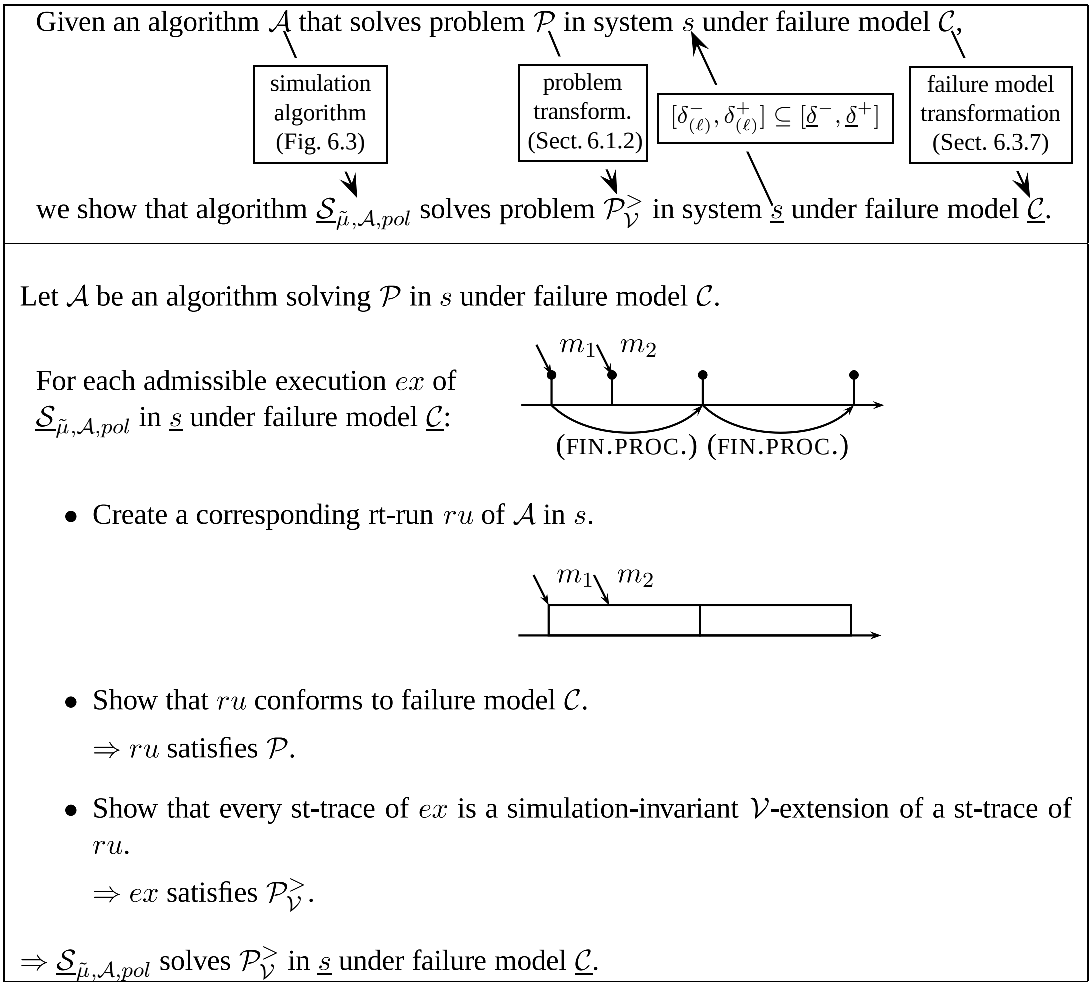
Thus, between T and $T + \tilde{\mu}_{(\ell)}$, $idle = false$ and only algorithm messages arrive at p, which means that only type (b) actions can occur. □

6.3.4. Transformation $T_{C\to R}$

As shown in the outline (Figure 6.5), the proof works by transforming every execution ex of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ into a corresponding rt-run of $\mathcal{A}$. By showing that (a) this rt-run is a valid rt-run of $\mathcal{A}$ and (b) the execution and the rt-run have (roughly) the same state transitions, the fact that the execution satisfies $\mathcal{P}_{\mathcal{V}}^{>}$ can be derived from the fact that the rt-run satisfies $\mathcal{P}$.

The transformation $ru = T_{C\to R}(ex)$ constructs a rt-run ru. We set $HC_p^{ru} = HC_p^{ex}$ for all p, such that both ex and ru have the same hardware clocks. Depending on the type of action, a corresponding receive event, job and/or drop event in ru is constructed for each action ac:

- Definitions:



Given an algorithm $\mathcal{A}$ that solves problem $\mathcal{P}$ in system s under failure model $\mathcal{C}$,

simulation algorithm (Fig. 6.3)

problem transform. (Sect. 6.1.2)

$[\delta^-_{(\ell)}, \delta^+_{(\ell)}] \subseteq [\underline{\delta}^-, \underline{\delta}^+]$

failure model transformation (Sect. 6.3.7)

we show that algorithm $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ solves problem $\mathcal{P}^{>}_{\mathcal{V}}$ in system $\underline{s}$ under failure model $\underline{\mathcal{C}}$.

Let $\mathcal{A}$ be an algorithm solving $\mathcal{P}$ in s under failure model $\mathcal{C}$.

For each admissible execution ex of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ in $\underline{s}$ under failure model $\underline{\mathcal{C}}$:

- Create a corresponding rt-run ru of $\mathcal{A}$ in s.

- Show that ru conforms to failure model $\mathcal{C}$.

  $\Rightarrow$ ru satisfies $\mathcal{P}$.

- Show that every st-trace of ex is a simulation-invariant $\mathcal{V}$-extension of a st-trace of ru.

  $\Rightarrow$ ex satisfies $\mathcal{P}^{>}_{\mathcal{V}}$.

$\Rightarrow$ $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ solves $\mathcal{P}^{>}_{\mathcal{V}}$ in $\underline{s}$ under failure model $\underline{\mathcal{C}}$.

Figure 6.5.: Transformation outline (Theorem 6.23)

- $trans^*(ac)$: Let $trans^*(ac)$ contain $trans(ac)$ (1) without the simulation algorithm variables $queue$ and $idle$, (2) without state transitions only involving simulation variables and (3) without any sending of (FINISHED-PROCESSING) messages.
- $\mu(ac)$: Let $\mu(ac)$ of a type (a) or (c) action ac be the real-time interval between $time(ac)$ and the arrival of the (FINISHED-PROCESSING) message sent by ac (cf. Observation 6.16). Note that $\mu(ac)$ denotes a real-time interval, whereas $\tilde{\mu}_{(\ell)}$ is defined in hardware clock time units.

- Type (a): This action is mapped to a receive event R and a subsequent job J in ru.

$$
\begin{array}{lll}
proc(R) \leftarrow proc(ac) & proc(J) \leftarrow proc(ac) & duration(J) \leftarrow \mu(ac) \\
msg(R) \leftarrow msg(ac) & msg(J) \leftarrow msg(ac) & HC(J) \leftarrow HC(ac) \\
time(R) \leftarrow time(ac) & begin(J) \leftarrow time(ac) & trans(J) \leftarrow trans^*(ac)
\end{array}
$$

- Type (b): This action is *usually*[6] mapped to a receive event R in ru:

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

- Type (c): This action is mapped to a job J in ru. Let m be the algorithm message of the corresponding type (b) action (cf. Observation 6.15), i.e., the message chosen as *next* in Line 13.

  $proc(J) \leftarrow proc(ac)$ $\quad begin(J) \leftarrow time(ac)$ $\quad HC(J) \leftarrow HC(ac)$

  $msg(J) \leftarrow m$ $\quad duration(J) \leftarrow \mu(ac)$ $\quad trans(J) \leftarrow trans^*(ac)$

  In addition, for every message m removed from $queue$ but not chosen as $next$ in Line 13 (if any), a drop event D is created right before J:

  $proc(D) \leftarrow proc(ac)$ $\quad msg(D) \leftarrow m$ $\quad time(D) \leftarrow time(ac)$

- Type (d): Similar to type (c) actions, a drop event D is created for every message m removed from $queue$ in Line 13 (if any):

  $proc(D) \leftarrow proc(ac)$ $\quad msg(D) \leftarrow m$ $\quad time(D) \leftarrow time(ac)$

- Type (e): This action is mapped to a receive event R and a subsequent drop event D in ru, both with the same parameters:

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

To illustrate this transformation, Figure 6.5 shows an example with actions of types (a), (b), (c) and (d) occurring in ex (in this order) and the resulting rt-run ru.

Lemma 6.20. *There is a one-to-one correspondence between* (FINISHED-PROCESSING) *messages in ex and jobs in ru: A job J exists in ru if, and only if, there is a corresponding* (FINISHED-PROCESSING) *message m in ex, with $begin(J) = time(ac)$ of the action ac sending m and $end(J) = time(ac')$ of the action ac' receiving m.*

Proof. (FINISHED-PROCESSING) $\rightarrow$ *job:* Note that (FINISHED-PROCESSING) messages in ex are only sent in type (a) and (c) actions. $T_{C\rightarrow R}$ ensures that for both kinds of actions a job exists in ru which ends exactly at the time at which the (FINISHED-PROCESSING) message arrives in ex (recall the definition of $\mu(ac)$).

job $\rightarrow$ (FINISHED-PROCESSING)*:* Follows from the fact that, due to the rules of $T_{C\rightarrow R}$, jobs only exist in ru if there is a corresponding type (a) or (c) action in ex. These actions send (FINISHED-PROCESSING) messages, and the definition of $\mu(ac)$ ensures that these messages do not arrive until the job has completed. □

[6] There is one special case in which $time(R)$ is set to a different value, see below for details.

6.3.5. Special Case: Timer Messages

As explained in Section 4.2.2, there is a subtle difference between the classic and the real-time computing model with respect to the $arrives\_timely(m_t)$ predicate of FAULT-FREE: In a rt-run, a timer message m_t sent during some job J arrives at the end of the job ($end(J)$) if the desired arrival hardware clock time ($sHC(m_t)$) occurs while J is still in progress. On the other hand, in an execution, the timer message always arrives at $sHC(m_t)$.

For $T_{C \to R}$ this means that the transformation rule for type (b) actions changes: If the type (b) action ac for timer message $m_t = msg(ac)$ occurs at some time $t = time(ac)$ while the (FINISHED-PROCESSING) message corresponding to the simulated job that sent m_t is still in transit (cf. Lemma 6.20 and note that this lemma is unaffected by this change of $T_{C \to R}$), then the corresponding receive event R does not occur at t but rather at $t' = time(ac')$, with ac' denoting the type (c) or (d) action where the (FINISHED-PROCESSING) message arrives. Still, $R \prec J$ and $R \prec D$ shall hold, for any job J or drop event D created by transforming ac' with $T_{C \to R}$.

This change ensures that the receive event in the simulated rt-run occurs at the correct time, i.e., no earlier than at the end of the job sending the timer message. One inconsistency still remains, though: The order of the messages in the queue might differ between the simulated queue in the execution (i.e., variable $queue$) and the queue in the rt-run constructed by $T_{C \to R}$: In the execution, m_t is added to $queue$ at time t, whereas in the rt-run, m_t is added to the real-time queue at time t'. This could make a difference, for example, when another message arrives between t and t'.

Since $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ "knows" about $\mathcal{A}$, it is obviously possible for the simulation algorithm to detect such cases and reorder $queue$ accordingly. We have decided not to include these details in Figure 6.3, since the added complexity might make it more difficult to understand the main structure of the simulation algorithm. For the remainder of Section 6.3, we will assume that such a reordering takes place.

6.3.6. Validity of the Constructed Rt-run

Lemma 6.21. *If ex is a valid execution of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ under failure model FAULT-FREE$_\rho(ex)$, $ru = T_{C \to R}(ex)$ is a valid rt-run of $\mathcal{A}$.*

Proof. Let $red(s)$ be defined as state s without the simulations variables $queue$ and $idle$. We will show that RU1–8 defined in Section 3.2 are satisfied:

RU1 Since the transformation rules of $T_{C \to R}$ only create items in ru whose begin times match those of their corresponding actions, RU1 (non-decreasing begin times) follows from EX1 by applying these transformation rules sequentially to all ac in ex. RU1 also requires message causality: Since (1) $trans(J)$ of every job J corresponds to a subset of $trans(ac)$ of some action ac occurring at the same time and (2) $msg(R)$ of every receive event R corresponds to some message $msg(ac)$ of some action ac occurring at the same time, it is not hard to see that a message m violating message causality (by being sent after being received) can only exist in ru if a corresponding message m' already existed in ex, which is prohibited by EX1.

RU2 Assume by contradiction that there are two subsequent jobs J and J' on the same processor p such that $newstate(J) \neq oldstate(J')$. According to the transformation rules of $T_{C \to R}$, J corresponds to some type (a) or (c) action ac and $red(newstate(ac)) = newstate(J)$. The same holds for J', which corresponds to some type (a) or (c) action ac' with $red(oldstate(ac')) = oldstate(J')$. Since $newstate(J) \neq oldstate(J')$, $red(newstate(ac)) \neq red(oldstate(ac'))$. As EX2 holds in ex, there must be some action ac'' in between ac and ac' such that $red(oldstate(ac'')) \neq red(newstate(ac''))$. This yields two cases, both of which lead to a contradiction: (1) ac'' is a type (a) or (c) action. In that case, there would be some corresponding job J'' with $J \prec J'' \prec J'$ in ru, contradicting the assumption that J and J' are subsequent jobs. (2) ac'' is a type (b), (d) or (e) action. Since, in our fault-free failure model, these kinds of actions only change $queue$ and $idle$, this contradicts $red(oldstate(ac'')) \neq red(newstate(ac''))$.

RU3 On every processor p, $oldstate(J)$ of the first job J on p in ru is equal to $red(oldstate(ac))$ of the first type (a) or (c) action ac on p in ex. Following the same reasoning as in the previous point, we can argue that $red(oldstate(ac)) = red(oldstate(ac'))$, with ac' being the first (any type) action on p in ex. Since, by definition of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$, the set of initial states of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ equals the one of $\mathcal{A}$ (extended with $queue = empty$ and $idle = true$), RU3 follows from EX3.

RU4 Follows easily from $HC_p^{ru} = HC_p^{ex}$, the transformation rules of $T_{C \to R}$ and the fact that EX4 holds in ex.

RU5 *At most one job sending m*: Follows from the fact that every action ac is mapped to at most one job J, $sent(J)$ is a subset of $sent(ac)$, and EX5 holds in ex.

At most one receive event receiving m: Follows from the fact that every action ac is mapped to at most one receive event R in ru receiving the same message and EX5 holds in ex.

At most one job processing m or drop event dropping m: Since EX5 holds in ex, every message received in $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ is unique. Thus, every message gets put into $queue$ at most once and, since pol is a valid scheduling/admission policy and $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ is executed fault-free, every message is removed from $queue$ at most once. Transformation $T_{C \to R}$ is designed such that a job or drop event with $msg(J/D) = m$ is created in ru if, and only if, m gets removed from $queue$ in the corresponding action.

Correct causal order: The correct order of message sends and receive events is already ensured by RU1. W.r.t. jobs and drop events, consider the five different types of actions. Type (a): J is created right after R. Type (b): No job or drop event is created. Type (c) and (d): By Observation 6.15, every message removed from $queue$ (= every message for which a job or drop event is created by $T_{C \to R}$) has been received before by a type (b) action. By $T_{C \to R}$, a receive event has been created for this message. Type (e): D is created right after R.

Correct processor specified in the message: Follows from the fact that EX5 holds in ex and that $T_{C \to R}$ does not change the processor at which messages are sent, received, processed or dropped.

RU6 $T_{C\to R}$ ensures that all message sends, except for (FINISHED-PROCESSING), are transferred into ru (recall the definition of $trans^*$). Likewise, all receptions of such algorithm messages (type (a), (b) and (e) actions) are transferred into corresponding receive events in ru. No new messages (i.e., messages not present in ex) are introduced into ru by $T_{C\to R}$. Thus, RU6 follows from EX6.

RU7 Consider two jobs $J \prec J'$ on the same processor $proc(J) = proc(J') = p$. $T_{C\to R}$ ensures that there is a corresponding type (a) or (c) action for every job in ru. Let ac and ac' be the actions corresponding to J and J' and note that $time(ac) = begin(J)$ and $time(ac') = begin(J')$. Lemma 6.19 implies that ac' cannot occur until the (FINISHED-PROCESSING) message sent by ac has arrived. Since $duration(J)$ is set to $\mu(ac)$ in $T_{C\to R}$, the definition of $\mu(ac)$ ensures that J' cannot start before J has finished.

RU8 Drop events occur in ru only when there is a corresponding type (c), (d) or (e) action in ex. Type (c) and (d) actions are triggered by a (FINISHED-PROCESSING) message arriving; thus, by Lemma 6.20, there is a job in ru finishing at that time. W.r.t. type (e) actions, Observation 6.18 shows that p is idle in ex when a type (e) action occurs, which, by Lemma 6.17, means that no (FINISHED-PROCESSING) message is in transit and, thus, by Lemma 6.20, there is no job active in ru. Therefore p is idle in ru and $T_{C\to R}$ ensures that a receive event occurs at the time of the type (e) action. □

6.3.7. Failure Model Compatibility

Lemma 6.22. *Let ex be an execution of some algorithm $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ in some system $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ under failure model FAULT-FREE$_\rho(ex)$. Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system, and let $\tilde{\mu}_{(\ell)}$ be defined as $2\frac{\mu^+_{(\ell)}\mu^-_{(\ell)}}{\mu^+_{(\ell)}+\mu^-_{(\ell)}}$. If, for all $\ell \in \{0, \dots, n-1\}$, $\delta^-_{(\ell)} \le \underline{\delta}^-$, $\delta^+_{(\ell)} \ge \underline{\delta}^+$ and $\rho \le \frac{\mu^+_{(\ell)}-\mu^-_{(\ell)}}{\mu^+_{(\ell)}+\mu^-_{(\ell)}}$, then ru, the rt-run of $\mathcal{A}$ created by applying transformation $T_{C\to R}$ to ex, conforms to failure model FAULT-FREE$_\rho(ru)$ in system s with scheduling/admission policy pol.*

Proof. Lemma 6.21 has shown that ru is a valid rt-run of $\mathcal{A}$. Thus, we need to show that ru satisfies FAULT-FREE$_\rho(ru)$:

- $\forall m_o : is\_timely\_msg(m_o, \delta^-, \delta^+)$

  All actions that receive algorithm messages (types (a), (b) and (e)) are mapped to receive events occurring at the same real-time as the action. All actions possibly sending algorithm messages (types (a) and (c)) are mapped to jobs starting at the same real time as the action. Since $\delta^-_{(\ell)} \le \underline{\delta}^-$ and $\delta^+_{(\ell)} \ge \underline{\delta}^+$ for all ℓ, the required delay condition for ru follows directly from the fact that FAULT-FREE$_\rho(ex)$, and, thus, $\forall m_o : is\_timely\_msg(m_o, \underline{\delta}^-, \underline{\delta}^+)$ holds in ex.

- $\forall m_t : arrives\_timely(m_t)$

  Algorithm timer messages in ex sent for some hardware clock value T on some processor p cause a type (a), (b) or (e) action ac at some time t with $HC(ac) = T$

when they are received. As all of these actions are mapped to receive events R with $msg(R) = msg(ac)$ and $time(R) = t$ (or $time(R) = end(J)$ of the job J sending the timer, see Section 6.3.5), and the hardware clocks are the same in ru and ex, timer messages arrive at the correct time in ru.

- $\forall R : obeys\_pol(R)$

  Observe that, due to the design of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ and $T_{C\to R}$, variable $queue$ in ex represents the queue state of ru. Every receive event in ru occurring while the processor is idle corresponds to either a type (a) or a type (e) action. In every such action, a scheduling decision according to pol is made (Line 13) and $T_{C\to R}$ ensures that either a drop event (type (e) action) or a job (type (a) action) according to the output of that scheduling decision is created.

- $\forall J : obeys\_pol(J)$

  The same reasoning as in the previous point applies: Every job in ru finishing corresponds to a type (c) or (d) action in ex in which the (FINISHED-PROCESSING) message representing that job arrives. Both of these actions cause a scheduling decision (Line 13) to be made on $queue$ (which corresponds to ru's queue state), and corresponding drop events and/or a corresponding job (only type (c) actions) are created by $T_{C\to R}$.

- $\forall J : follows\_alg(J)$

  Let ac be the type (a) or (c) action corresponding to J. ac executes all state transitions of $\mathcal{A}$ (Line 19) for either $msg(ac)$ (type (a) action) or some message from the queue (type (c) action) and the current hardware clock time, plus some additional operations which only affect variables $queue$ and $idle$ and (FINISHED-PROCESSING) messages. Thus, $T_{C\to R}$'s choice of $HC(J)$ and $msg(J)$ as well as the definition of $trans^*$ ensure that $trans(J)$ conforms to algorithm $\mathcal{A}$.

- $\forall J : is\_timely\_job(J, \mu^-, \mu^+)$

  By definition of $T_{C\to R}$, $duration(J) = \mu(ac)$, with $\mu(ac)$ denoting the transmission time of the (FINISHED-PROCESSING) message sent by the action ac corresponding to job J. Since $arrives\_timely(m_t)$ holds for (FINISHED-PROCESSING) messages m_t in ex, there are exactly $\tilde{\mu}_{(\ell)}$ hardware clock time units between the sending and the reception of the (FINISHED-PROCESSING) message sent by ac (see Line 21 of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$). By Corollary 6.14, this corresponds to some real-time interval $\mu(ac)$ within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$. Since ℓ equals the number of ordinary messages sent in J (see Line 20 of the algorithm and the transformation rules for type (a) and (c) actions in $T_{C\to R}$), $is\_timely\_job(J, \mu^-, \mu^+)$ holds.

- $\forall p : bounded\_drift(p, \rho)$

  Follows from the definition that $HC^{ru}_p = HC^{ex}_p$ and the fact that the corresponding $bounded\_drift$ condition holds in ex. □

6.3.8. Transformation Proof

Theorem 6.23. *Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system and $\mathcal{P}$ be a problem. If*

- *there exists an algorithm $\mathcal{A}$ which solves problem $\mathcal{P}$ in some real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with some scheduling/admission policy pol under failure model FAULT-FREE$_\rho(ru)$* ***[A1]**,*
- *$\forall \ell : \delta^-_{(\ell)} \leq \underline{\delta}^-$ and $\delta^+_{(\ell)} \geq \underline{\delta}^+$* ***[A2]**, and*
- *$\forall \ell : \rho \leq \frac{\mu^+_{(\ell)} - \mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$* ***[A3]**,*

then $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ given in Figure 6.3 with $\tilde{\mu}_{(\ell)} = 2\frac{\mu^+_{(\ell)}\mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$ ***[D1]** solves $\mathcal{P}^{\gtrsim}_{\mathcal{V}}$ in $\underline{s}$ under failure model FAULT-FREE$_\rho(ex)$, with $\mathcal{V} = \{queue, idle\}$* ***[GOAL]**.*

Proof. Let ex be such an execution of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ in $\underline{s}$ under failure model FAULT-FREE$_\rho(ex)$ **[D2]**. By Lemmas 6.21 and 6.22 (in conjunction with [A2], [A3] and [D1]), $ru = T_{C \to R}(ex)$ is a valid execution of $\mathcal{A}$ in s with scheduling/admission policy pol under failure model FAULT-FREE$_\rho(ru)$ **[L1]**.

As $\mathcal{A}$ is an algorithm solving $\mathcal{P}$ in s with policy pol under failure model FAULT-FREE$_\rho(ru)$ ([A1]) and ru is a valid rt-run of $\mathcal{A}$ in s with policy pol conforming to failure model FAULT-FREE$_\rho(ru)$ ([L1]), ru satisfies $\mathcal{P}$ (cf. Definition 5.12) **[L2]**.

To show that ex satisfies $\mathcal{P}^{\gtrsim}_{\mathcal{V}}$, we must show that every st-trace tr' of ex is a simulation-invariant $\mathcal{V}$-extension of a st-trace tr of ru (cf. Section 6.1.2). Let tr' be a st-trace of ex **[D3]**. We can construct tr from tr' by sequentially performing these operations:

1. Remove the variables $queue$ and $idle$ from all states.
2. Remove any $transition$ st-events that only manipulated $queue$ and/or $idle$. Note that, due to the previous step, these st-events satisfied $oldstate = newstate$.
3. Let ac be a type (b) action receiving some message m.
   a) If there is a corresponding type (c) action ac' in which m gets processed by calling $\mathcal{A}$-process_message(m, current_hc) (cf. Observation 6.15), let ev and ev' be the $process$ st-events corresponding to ac and ac'. Remove ev and modify ev' such that $msg(ev') = msg(ev)$, i.e., that ev' processes the message originally processed in ev, rather than a (FINISHED-PROCESSING) message.
   b) If there is a corresponding type (c) or (d) action ac' in which m gets dropped, i.e., m gets removed from $queue$ in Line 13 without being chosen as $next$, let ev and ev' be the $process$ st-events corresponding to ac and ac'. Move ev into the future such that $time(ev) = time(ev')$ and ev is right before ev'.
4. Remove all $process$ and $send$ st-events that are (still) processing or sending (FINISHED-PROCESSING) messages.

Since all of the operations used to construct tr can be reverted by using the rules outlined in Section 6.1.2, tr' is a simulation-invariant $\mathcal{V}$-extension of tr **[L3]**. We now need to show that tr is a st-trace of ru.

- *Every job in ru is correctly mapped to st-events in tr:* Every job J in ru is based on either a type (a) or a type (c) action ac in ex. Following Definitions 5.3 and 5.4, the st-events produced by mapping ac are the same as the st-events produced by mapping J, with the following differences:

  - The st-events mapped by ac contain the simulation variables. However, they have been removed by the transformation from tr' to tr.
  - If ac is a type (c) action, its $process$ st-event processes a (FINISHED-PROCESSING) message rather than the algorithm message received in the corresponding type (b) action. The creation of tr (step 3a) also ensures that the correct message is used in tr.
  - If ac is a type (a) action and $msg(ac)$ is an input message, there is an additional $input$ st-event before the $process$ st-event. By construction of ru, however, there is a receive event at the time of the type (a) action corresponding to the $input$ st-event in tr.

- *Every drop event in ru is correctly mapped to a $process$ st-event in tr:* Every drop event D in ru is based on a type (c), (d) or (e) action ac in ex.

  With respect to type (c) or (d) actions, $T_{C \rightarrow R}$ ensures that drop events are created only for messages that are removed from $queue$ without being chosen as $next$ during that action. The creation of tr (step 3b) ensures that a corresponding $process$ st-event (ev) is present in tr.

  With respect to type (e) actions, note that no messages are sent with this kind of action (i.e., there are no $send$ st-events) and that the creation of tr removes all $transition$ st-events corresponding to that action (steps 1 and 2), leaving only the $process$ st-event corresponding to the drop event in ru.

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

- *Every st-event in tr corresponds to a job, input message receive event or drop event in ru:* Every st-event in tr' is based on an action ac in ex—in the natural way, as specified in Definition 5.3. Since the transformation $tr' \rightarrow tr$ does not add any st-events, every st-event in tr is based on an action ac in ex as well. Consider the different types of actions:

  - Type (a): The st-events in tr' contain the $send$ and the $transition$ st-events of $\mathcal{A}$-process_message(msg, current_hc) and additional steps taken by the simulation algorithm. The transformation from tr' to tr ensures that these additional steps (and only these) are removed. Thus, the remaining st-events in tr correspond to the job J corresponding to ac. If the message received by ac was an input message, the $input$ st-event corresponds to the receive event created with the $T_{C \to R}$ rule for type (a) actions.

- Type (b): This type of action only performs state transitions w.r.t. simulation variables and does not send any messages (i.e., there are no corresponding $send$ st-events). In the transformation from tr' to tr, all $transition$ st-events of this action are removed during steps 1 and 2. The $process$ st-event is either removed (during step 3a, if the message gets processed later) or corresponds to the drop event dropping the message (see step 3b), which is inserted into ru by the $T_{C \to R}$ rule for type (c)/(d) actions. The only st-event left based on this type (b) action is one $input$ st-event, if the received message was an input message. This $input$ st-event corresponds to the receive event created by the $T_{C \to R}$ rule for type (b) actions.
- Type (c): As in type (a) actions, the transformation from tr' to tr ensures that only the $send$ and the $transition$ st-events of $\mathcal{A}$-process_message(msg, current_hc) are left, with msg being the message received in the corresponding type (b) action. The transformation from tr' to tr ensures that the $process$ st-event in tr contains msg as the received message. Thus, the $process$ st-event and the following $send$ and $transition$ st-events match exactly with the job created in the $T_{C \to R}$ rule for type (c) actions.
- Type (d): Only state transitions involving simulation variables are performed. All of these $transition$ st-events are lost during the creation of tr. As the $process$ st-event processes a (FINISHED-PROCESSING) message, it is removed as well.
- Type (e): This kind of action gets mapped to the following st-events: An $input$ st-event, if $msg(ac)$ was an input message, a $process$ st-event and one or more $transition$ st-events only modifying simulation variables. The latter are removed by the transformation from tr' to tr, the $input$ st-event corresponds to the receive event and the $process$ st-event to the drop event created by $T_{C \to R}$ for type (e) actions.

Thus, we can conclude that tr is a st-trace of ru **[L4]**. As $\mathcal{A}$ solves $\mathcal{P}$ in s with policy pol under failure model FAULT-FREE$_\rho(ru)$ ([A1]) and ru is a rt-run of $\mathcal{A}$ in s with policy pol under failure model FAULT-FREE$_\rho(ru)$ ([L1]), this ([L4]) implies that $tr \in \mathcal{P}$ (cf. Definition 5.12) **[L5]**. Since tr' is a simulation-invariant $\mathcal{V}$-extension of tr ([L3]) and $tr \in \mathcal{P}$ ([L5]), $tr' \in \mathcal{P}_{\mathcal{V}}^{>}$ (cf. Section 6.1.2) **[L6]**.

As this ([L6]) holds for every st-trace tr' of every execution ex of $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ in $\underline{s}$ under failure model FAULT-FREE$_\rho(ex)$ ([D2,D3]), Definition 5.10 states that $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ solves $\mathcal{P}_{\mathcal{V}}^{>}$ in $\underline{s}$ under failure model FAULT-FREE$_\rho(ex)$ ([GOAL]). □

6.3.9. Generalization

We finally note that the bound $\delta^-_{(\ell)} \leq \underline{\delta}^-$ and $\delta^+_{(\ell)} \geq \underline{\delta}^+$ for all ℓ in Theorem 6.23 is overly conservative. The following bound suffices.

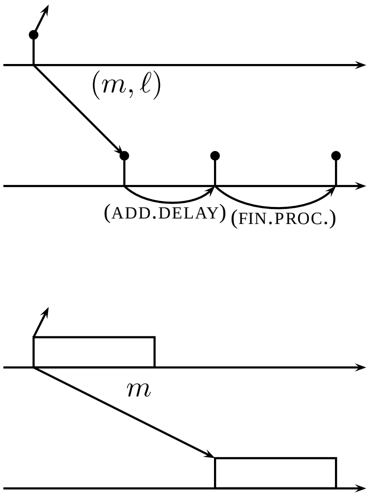


Figure 6.6.: Illustration of $\underline{\mathcal{S}'}_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$

Theorem 6.24. *Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system and $\mathcal{P}$ be a problem. If*

- *there exists an algorithm $\mathcal{A}$ which solves problem $\mathcal{P}$ in some real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with some scheduling/admission policy pol under failure model FAULT-FREE$_\rho(ru)$,*
- *$\delta^-_{(1)} \le \underline{\delta}^-$ and $\delta^+_{(1)} \ge \underline{\delta}^+$, and*
- *$\forall \ell : \rho \le \frac{\mu^+_{(\ell)} - \mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$ and $\rho \le \frac{(\delta^+_{(\ell)} - \delta^+_{(1)}) - (\delta^-_{(\ell)} - \delta^-_{(1)})}{(\delta^+_{(\ell)} - \delta^+_{(1)}) + (\delta^-_{(\ell)} - \delta^-_{(1)})}$,*

then $\underline{\mathcal{S}'}_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$ with $\tilde{\mu}_{(\ell)} = 2\frac{\mu^+_{(\ell)}\mu^-_{(\ell)}}{\mu^+_{(\ell)} + \mu^-_{(\ell)}}$ and $\tilde{\delta}_{(\ell)} = 2\frac{(\delta^+_{(\ell)} - \delta^+_{(1)})(\delta^-_{(\ell)} - \delta^-_{(1)})}{(\delta^+_{(\ell)} - \delta^+_{(1)}) + (\delta^-_{(\ell)} - \delta^-_{(1)})}$ solves $\mathcal{P}^>_{\mathcal{V}}$ in $\underline{s}$ under failure model FAULT-FREE$_\rho(ex)$.

Proof. Since algorithm $\underline{\mathcal{S}'}_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$ and its proof are very similar to $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$, only an informal description is given as follows: First, note that $\delta^+_{(1)} \ge \underline{\delta}^+ \Leftrightarrow \forall l : \delta^+_{(\ell)} \ge \underline{\delta}^+$, due to $\delta^+_{(\ell)}$ being non-decreasing with respect to ℓ (cf. Section 3.3). Thus, the extended simulation algorithm mainly allows $\delta^-_{(\ell)}$ to be greater than $\underline{\delta}^-$ for $\ell > 1$. However, since $\varepsilon_{(\ell)} \ge \varepsilon_{(1)}$ (again, see Section 3.3), we can ensure that the simulated message delays lie within $\delta^-_{(\ell)}$ and $\delta^+_{(\ell)}$, although the real message delay might be smaller than $\delta^-_{(\ell)}$, by introducing an artificial, additional message delay within $[\delta^-_{(\ell)} - \delta^-_{(1)}, \delta^+_{(\ell)} - \delta^+_{(1)}]$ upon receiving a message sent by a job sending ℓ ordinary messages in total. Note that Lemma 6.13, the restriction on ρ and the definition of $\tilde{\delta}$ ensure that such a delay can be estimated by $\tilde{\delta}$.

Of course, being able to add this delay implies that the algorithm message is wrapped into a simulation message that also includes the value ℓ. Figure 6.6 illustrates the principle of this algorithm and the transformation of an execution of $\underline{\mathcal{S}'}_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$ into an rt-run. □

Part II.

Clock Synchronization

7. Optimal Drift-Free Clock Synchronization

This chapter analyzes the *terminating clock synchronization* problem in the drift- and failure-free case. This problem, which was already presented in Section 5.2.4 as an example, is defined as follows:

Definition 7.1 (Terminating Clock Synchronization to within γ)**.** Let the *adjusted clock value* $AC_p(g)$ be defined as $HC_p^{tr}(time(g)) + s_p(g).adj$.

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

  $\forall ev \in tr : (type(ev) = input) \Rightarrow (content(msg(ev)) = \text{“init”})$

- *Termination:* All processors eventually terminate.

  $\exists g \in gstates(tr) : is\_finalstate(g)$[1]

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

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

In the classic computing model, a tight bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ has been proved in [LL84b] as the best achievable clock synchronization precision. In addition, an algorithm $\underline{\mathcal{A}}(n, \underline{\delta}^-, \underline{\delta}^+)$ has been given, which guarantees this optimal precision in every classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$ with $\underline{\varepsilon} = \underline{\delta}^+ - \underline{\delta}^-$. The algorithm works by sending one timestamped message from every processor to every other processor, and then computing the average of the estimated clock differences as a correction value. Every processor broadcasts its timestamped message as soon as its init message arrives.

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

Theorem 7.2. *In the real-time computing model, clock synchronization to within* $(1-\frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ *is possible.*

Proof. The existence of a real-time computing model algorithm achieving this precision will be shown by applying Theorem 6.11 to the classic computing model algorithm of [LL84b]

[1] $is\_finalstate(g) :\Leftrightarrow \forall g' \succ g : \forall p : s_p(g) = s_p(g')$

(henceforth called "algorithm $\underline{\mathcal{LL}}$"). As outlined in Section 6.1.3, terminating clock synchronization is a shuffle-compatible problem. For the real-time system, we choose some arbitrary non-idling scheduling/admission policy pol which does not drop any messages. Algorithm $\underline{\mathcal{LL}}$ has been shown to be correct with respect to failure model FAULT-FREE$_0(ex)$ in [LL84b]. Since the algorithm does not use any timer message, this implies correctness in the more relaxed model $\underline{\mathcal{C}} :=$ FAULT-FREE$_0$+latetimers$_\infty(ex)$ (= 0-CRASH$_0$+latetimers$_\infty(ex)$, as defined in Section 6.2.4). For the real-time computing model, we choose failure model $\mathcal{C} :=$ FAULT-FREE$_0(ru)$ (= 0-CRASH$_0(ru)$ = 0-CRASH$_0$ $+ \, precisetimers_\infty$). By Observation 6.9, these choices of $\underline{\mathcal{C}}$ and $\mathcal{C}$ are $T_{R \to C}$-compatible.

Thus, the only thing left to show is that there exists a feasible assignment such that $\underline{\delta}^- = \Delta^-$ and $\underline{\delta}^+ = \Delta^+$. We can determine general bounds Δ^- and Δ^+ with some simple observations: Since, in $\underline{\mathcal{LL}}$ (and, thus, in $\mathcal{S}_{\underline{\mathcal{LL}}}$), all messages are sent as broadcasts to $n-1$ recipients, $\Delta^- = \delta^-_{(n-1)}$. With respect to Δ^+, note that every processor receives exactly one message from every other processor. The worst-case scenari o for the end-to-end delay hence occurs if all $n-1$ messages plus the one init message arrive simultaneously: After delivery of these messages (taking $\delta^+_{(n-1)}$), the receiver's own broadcast send step (taking $\mu^+_{(n-1)}$) as well as $n-2$ receive steps ($\mu^+_{(0)}$) must complete before the last receive step can start. An upper bound on the end-to-end delay of running $\mathcal{S}_{\underline{\mathcal{LL}}}$ in the real-time computing model is hence $\Delta^+ = \delta^+_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)}$.

Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system in which we want to synchronize clocks. We know that $\underline{\mathcal{LL}}(n, \Delta^-, \Delta^+)$ will synchronize clocks to within $\gamma = (1 - \frac{1}{n})(\Delta^+ - \Delta^-)$ in the classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$. Theorem 6.11 shows that $\mathcal{S}_{\underline{\mathcal{LL}}}$ provides clock synchronization to within $(1-\frac{1}{n})(\Delta^+ - \Delta^-) = (1-\frac{1}{n})(\delta^+_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)} - \delta^-_{(n-1)}) = (1-\frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ in s. □

As far as the time complexity of the above algorithm is concerned (cf. Section 5.5), we observe that at most $\delta^+_{(n-1)}$ time units after the last init message arrived all processors have all $n-1$ messages in their queue (or already processed). Due to non-idling scheduling, this implies a maximum time complexity of $\max\big(\delta^+_{(n-1)}, \mu^+_{(n-1)}\big) + (n-1) \cdot \mu^+_{(0)}$, which occurs if all processors' init messages arrive at the same time. In sharp contrast to the classic computing model, where the time complexity of this algorithm is $\mathrm{O}(1)$, the worst-case time complexity in the real-time computing model is hence $\Theta(n)$.

Likewise, we can use the other transformation to prove that clock synchronization closer than $(1-\frac{1}{n})\varepsilon_{(1)}$ is impossible in real-time systems. Note that the impossibility in a drift- and fault-free environment (FAULT-FREE$_0(ru)$) trivially implies the impossibility in any system with drifting clocks and/or faulty processors.

Theorem 7.3. *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 real-time computing model algorithm $\mathcal{A}$ and some scheduling policy pol which can provide clock synchronization for some real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ to within $\gamma < (1-\frac{1}{n})\varepsilon_{(1)}$ under failure model FAULT-FREE$_0(ru)$. Applying Theorem 6.24 would imply that $\underline{\mathcal{S}'}_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$ provides clock synchronization to within $\gamma < (1 - \frac{1}{n})(\underline{\delta}^+ - \underline{\delta}^-)$ for some classic system $(n, [\underline{\delta}^- = \delta^-_{(1)}, \underline{\delta}^+ = \delta^+_{(1)}])$. This, however, contradicts the well-known lower bound result of [LL84b]. □

7.1. Algorithms

The comparison of Theorems 7.2 and 7.3 raises the obvious question of whether the lower bound of $(1 - \frac{1}{n})\varepsilon_{(1)}$ is tight in the real-time computing model. In this section, we will answer this in the affirmative: We show how the algorithm presented in [LL84b] can be modified to avoid queuing effects and thus provides optimal precision in a real-time system $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$. We will first present an algorithm achieving a precision of $(1 - \frac{1}{n})\varepsilon_{(n-1)}$ (which equals $(1 - \frac{1}{n})\varepsilon_{(1)}$ 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 every case.

7.1.1. Generalization of Existing Results

Two lemmas from [LL84b] can be generalized to our setting. Let Δ^- and Δ^+ be the lower and upper bound on the end-to-end delay.

Lemma 7.4. *If q receives a timestamped message from p with end-to-end delay*[2] *uncertainty $\varepsilon_\Delta := \Delta^+ - \Delta^-$, q can estimate p's hardware clock value within an error of at most $\frac{\varepsilon_\Delta}{2}$.*

Proof. (Similar to Lemma 5 of [LL84b].) We define $D := HC_p(t) - HC_q(t)$ to be the actual difference between the hardware clocks of p and q (a constant, as clocks do not drift) and E to be the estimated difference, as estimated by q. Thus, we have to show that q can calculate some E such that $|E - D| \le \frac{\varepsilon_\Delta}{2}$.

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

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

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

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

```
var estimates ← $empty$
var adj

procedure process_message(msg, current_hc)
  /* start alg. by sending (SEND) to proc. 0 */
  if msg = (SEND)
    send (TIME, current_hc) to all other processors
  elseif msg = (TIME, remote_hc)
    estimates.add(remote_hc − current_hc + $\frac{\dot\delta^- + \dot\delta^+}{2}$)
    if estimates.count = ID
      set timer (SEND) for current_hc + $\max(\dot\varepsilon - \dot\delta^- + \dot\mu^+, \dot\mu^+)$
    if estimates.count = n−1
      adj ← $(\sum estimates)/n$
```

Figure 7.1.: Clock-synchronization algorithm to within $\varepsilon_{(n-1)}$, code for processor ID

Lemma 7.5. *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. The proof can be obtained by a simple adaption of Theorem 7 of [LL84b] to a general err. □

7.1.2. Optimality for Broadcast Systems

Note. As all jobs in the algorithm of Figure 7.1 send either zero or $n-1$ messages, we will use the abbreviations $\dot\delta^-$, $\dot\delta^+$, $\dot\mu^-$, $\dot\mu^+$ and $\dot\varepsilon$ to refer to $\delta^-_{(n-1)}$, $\delta^+_{(n-1)}$, $\mu^-_{(n-1)}$, $\mu^+_{(n-1)}$ and $\varepsilon_{(n-1)}$, respectively.

In the introduction of Chapter 7, the principle of algorithm $\underline{\mathcal{LL}}$ [LL84b] has been described. It can easily be modified to avoid queuing effects by "serializing" the information exchange, rather than sending all messages simultaneously.

The modified algorithm, depicted in Figure 7.1, works as follows: The n fully-connected processors have IDs $0, \ldots, n-1$. The first processor (0) sends its clock value to all other processors. Processor i waits until it has received the message from processor $i-1$, waits for another $\max(\dot\varepsilon - \dot\delta^- + \dot\mu^+, \dot\mu^+)$ time units and then broadcasts its own hardware clock value. That way, every processor receives the hardware clock values of all other processors with uncertainty $\dot\varepsilon$, provided that no queuing occurs (which will be shown below). This information suffices to synchronize clocks to within $(1 - \frac{1}{n})\dot\varepsilon$. We assume here that only one init message is sent (only to processor 0), as additional init messages could cause unwanted queuing effects and would hence necessitate a second round of message exchanges.

Lemma 7.6. *No queuing occurs when running the algorithm in Figure 7.1.*

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

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

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

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

Using this lemma, it is not difficult to show the following Theorem 7.7:

Theorem 7.7 (Optimal broadcasting algorithm)**.** *The algorithm of Figure 7.1 achieves a precision of* $(1 - \frac{1}{n})\dot{\varepsilon}$*, which is tight if communication is performed by a constant-time broadcast primitive, i.e., if* $\varepsilon_{(n-1)} = \varepsilon_{(1)}$*. It performs exactly* n *broadcasts and has a time complexity that is at least* $\Omega(n)$*.*

Proof. On each processor, the *estimates* set contains the estimated differences between the local hardware clock and the hardware clocks of the other processors. As no queuing occurs by Lemma 7.6, the end-to-end delays are just the message delays. Line 9 in the algorithm of Figure 7.1 ensures that the estimate is calculated as specified in the proof of Lemma 7.4. Thus, the estimates have a maximum error of $\frac{\dot{\varepsilon}}{2}$. According to Lemma 7.5, these estimates allow the algorithm to calculate an adjustment value in Line 13 that guarantees clock-synchronization to within $(1 - \frac{1}{n})\dot{\varepsilon}$.

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

7.1.3. Optimality for Unicast Systems

Note. As all jobs in the algorithm of Figure 7.2 send either zero or one messages, we will use the abbreviations $\dot{\delta}^-$, $\dot{\delta}^+$, $\dot{\mu}^-$, $\dot{\mu}^+$ and $\dot{\varepsilon}$ to refer to $\delta^-_{(1)}$, $\delta^+_{(1)}$, $\mu^-_{(1)}$, $\mu^+_{(1)}$ and $\varepsilon_{(1)}$, respectively.

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

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

Figure 7.2.: Clock-synchronization algorithm to within $\varepsilon_{(1)}$, code for processor ID

The algorithm of the previous section provides clock synchronization to within $(1-\frac{1}{n})\varepsilon_{(n-1)}$. However, unless constant-time broadcast is available, $\varepsilon_{(1)}$ will usually be smaller than $\varepsilon_{(n-1)}$. The algorithm can be adapted to unicast sends as follows (see Figure 7.2):

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.

By the design of the algorithm, every processor i goes though five phases. The only exceptions are processor 0, which starts at phase 3, and processor $n-1$, which skips the second receive phase.

1. *First receive phase*: i receives TIME messages from all processors $\{0,\ldots,i-1\}$ in the order of increasing processor number.

2. *Wait phase*: After having received $i-1$'s TIME message, Line 13 causes i to wait for $W := \max(\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+, \dot{\mu}^+)$ time units.

3. *Send phase*: i sends TIME messages to all processors (each in its own job, all jobs $\dot{\mu}^+$ time units apart).

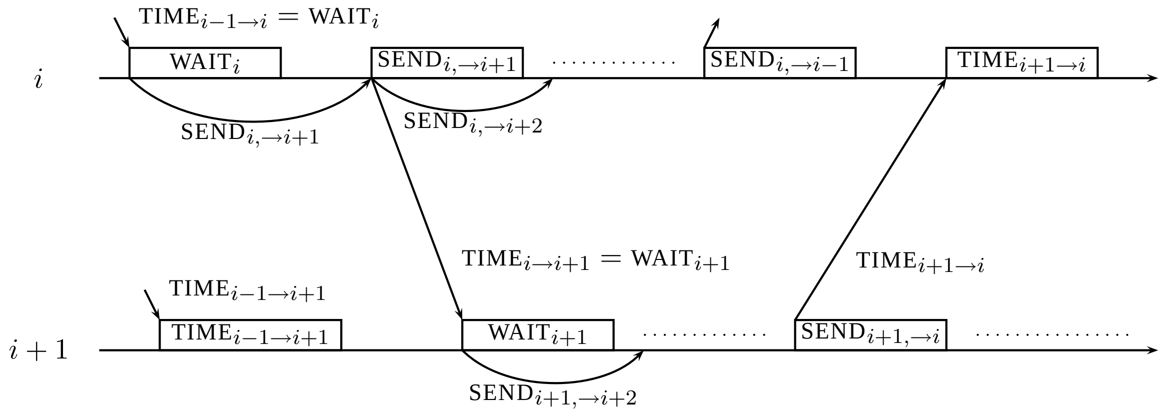
4. *Second receive phase*: i receives TIME messages from all processors $\{i+1,\ldots,n-1\}$ in the order of increasing processor number.

5. *Terminated phase*: No more messages are received; i has terminated.

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

with begin time $begin(\text{WAIT}_0) = begin(\text{SEND}_{0,\to 1}) - W$. (Recall that W, the wait time, is $\max(\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+, \dot{\mu}^+)$.)

See Figure 7.3 for an example. Note that every processor sends exactly one TIME message to every other processor.



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

Figure 7.3.: Processor i $(0 < i < n-2)$ switching from first receive phase to wait, from wait to send, and from send to second receive phase.

Lemma 7.8. *If $\dot{\mu}^+ > 0$, the following invariant holds for all rt-runs of the algorithm in Figure 7.2 under FAULT-FREE$_0(ru)$: All messages received on some processor i are received in the following order: $\langle \text{TIME}_{0\to i}, \ldots, \text{TIME}_{i-1\to i} = \text{WAIT}_i, \text{SEND}_{i,\to i\oplus 1}, \ldots, \text{SEND}_{i,\to i\oplus(n-1)}, \text{TIME}_{i+1\to i}, \ldots, \text{TIME}_{n-1\to i}\rangle$. All receive events on the same processor are at least $\dot{\mu}^+$ time units apart, which implies that no queuing occurs.*[3]

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

Proof. Let R_k be the k-th receive event in the rt-run. We will show by induction on k that all receive events occur at the right time and in the right order, and, thus, they are processed immediately without queuing delay.

Initially, R_1 contains the init message $\text{SEND}_{0,\to 1}$, which is the correct first message for processor 0. Assume that the condition holds for $R_1, \ldots, R_{k-1}$ and consider the following possibilities in which R_k (on processor i at time t) could violate the invariant by arriving too early:

$$\langle \text{TIME}_{0\to i}, \underbrace{\ldots, \text{TIME}_{i-1\to i} = \text{WAIT}_i}_{1}, \underbrace{\text{SEND}_{i,\to i\oplus 1}}_{2}, \underbrace{\ldots, \text{SEND}_{i,\to i\oplus(n-1)}}_{3}, \underbrace{\text{TIME}_{i+1\to i}}_{4}, \underbrace{\ldots, \text{TIME}_{n-1\to i}}_{1}\rangle$$

1. *First/second receive and wait phase*: Assume for $0 < j < i$ (first receive phase/wait phase) or $i{+}1 < j < n$ (second receive phase) that $\text{TIME}_{j\to i}$ arrives at $t < begin(\text{TIME}_{j-1\to i}) + \dot{\mu}^+$. $\text{TIME}_{j\to i}$ has been sent no later than $t - \dot{\delta}^-$ by j's $\text{SEND}_{j,\to i}$ job. As the invariant

[3] For $\dot{\mu}^+ = 0$, it is obvious that no queuing occurs.

holds for all previous receive events (and $\text{SEND}_{j,\to i}$ causally precedes $\text{TIME}_{j\to i}$), the begin times of the $((i \ominus 1) \ominus j)$ previous send phase steps of process j ($\text{SEND}_{j,\to j\oplus 1}, \dots,$ $\text{SEND}_{j,\to i\ominus 1}$) and $\text{SEND}_{j,\to i}$ are exactly $\dot{\mu}^+$ time units apart, and WAIT_j starts at least $\dot{\varepsilon} - \dot{\delta}^- + 2\dot{\mu}^+$ time units before the first send phase step. This means that

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

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

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

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

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

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

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

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

2. *Wait → send phase*: Assume the $\text{SEND}_{i,\to i\oplus 1}$ timer message arrives at $t \neq begin(\text{WAIT}_i) + W$. As the $\text{SEND}_{i,\to i\oplus 1}$ timer is set in WAIT_i to W, this is a contradiction.

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

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

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

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

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

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

We can apply Lemma 7.4 to the algorithm of Figure 7.2 as well, resulting in estimates with a maximum error of $\frac{\varepsilon_{(1)}}{2}$ rather than $\frac{\varepsilon_{(n-1)}}{2}$. Thus, by Lemma 7.5, clock synchronization to within $(1-\frac{1}{n})\varepsilon_{(1)}$ can be achieved. As all job durations and message delays are independent of n this time ($\delta^+_{(1)}$ rather than $\delta^+_{(n-1)}$, etc.), the time complexity of this algorithm is $\mathrm{O}(n)$.

7.2. Lower Bounds

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

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

A strong indication for this result follows already from the work of Biaz and Welch [BW01]. They have shown that no algorithm can achieve better precision than $\frac{1}{2}diam(G)$ for any communication network G, with $diam(G)$ being the diameter of the graph when the edges are weighted with the uncertainties: In the classic computing model, a fully-connected network with equal link uncertainty $\underline{\varepsilon}$ can achieve no better precision than $\frac{1}{2}\underline{\varepsilon}$, whereas removing one link yields a lower bound of $\underline{\varepsilon}$. Thus, after removing one link, the optimal precision of $(1-\frac{1}{n})\underline{\varepsilon}$ shown by [LL84b] can no longer be achieved.

Unfortunately, the proof from [BW01] cannot be used directly in our context to derive the message complexity bound mentioned above: While they show that $(1-\frac{1}{n})\underline{\varepsilon}$ cannot be achieved if the system forbids the algorithm to use one system-chosen link, we have to show that if the algorithm is presented with a fully-connected network and decides not to use one algorithm-chosen link (which can differ for each execution/rt-run) dynamically, this algorithm cannot achieve optimal precision. A shifting argument similar to the one used in their proof (Theorem 3 of [BW01]) can be used, however.

Additionally, we will show that in the real-time computing model, the message and time complexity of clock synchronization to within suboptimal precision also depends on the complexity of $\delta^+_{(\ell)}$ and $\mu^+_{(\ell)}$ with respect to ℓ.

7.2.1. Shifting

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

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

Note that this sequence might not be a valid execution, as messages could be received before they are sent.

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

- receive events, jobs and drop events 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, jobs and drop events still occur at the same hardware clock time as before, i.e., $HC^{ru'}_{p_i}(t) := HC^{ru}_{p_i}(t) + x_i$.

We assume that, just like in a valid rt-run, the receive events and jobs in a shifted rt-run are ordered by their occurrence time and begin time, respectively. Apart from that, the reordering must preserve the original ordering as much as possible; in particular, if two elements a' and b' occur/start at the same time t in ru', and $a \prec b$ holds for the corresponding elements in ru, $a' \prec b'$ must hold in ru'.

Observation 7.9. *ru', the sequence created by shifting some valid rt-run ru, satisfies properties RU1–RU8 except for the fact that message sends can occur later than the message arrival, thus violating message causality. It hence suffices to show that every message is sent before it is received to prove that ru' is a valid rt-run.*

Lemma 7.10. *If ru conforms to FAULT-FREE$_\rho$, ru' is a shifted rt-run of ru and all messages in ru' obey the message delay bounds $[\delta^-, \delta^+]$, then ru' conforms to FAULT-FREE$_\rho$.*

Proof. Follows from the fact that all conditions of FAULT-FREE$_\rho$ except for is_timely_msg are unaffected by the shifting operation. □

7.2.2. Environment

Let $c \in \mathbb{R}^+$ be a constant and $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system with $n > 2$. Assume that $\mathcal{A}$ is an algorithm running with some scheduling/admission policy pol and providing clock synchronization to within $c \cdot \varepsilon_{(1)}$ in s under failure model FAULT-FREE$_0$. Let ru be a rt-run of $\mathcal{A}$ with policy pol in s under FAULT-FREE$_0$ where the message delays of all messages are the arithmetic mean of the lower and upper bound. Thus, modifying the delay of any message by $\pm\varepsilon_{(1)}/2$ still results in a value within the system model bounds. The duration of all jobs sending ℓ messages is $\mu^+_{(\ell)}$. Since admission control is not needed in a fault-free environment, we assume that no messages are dropped by pol.

7.2.3. Message Graph Diameter

Definition 7.11. Let the *message graph* of a rt-run ru be defined as an undirected graph containing all processors as vertices and exactly those links as edges over which at least one message is sent in ru.

Lemma 7.12. *The message graph of ru has a diameter of $2c$ or less.*

Proof. Assume by contradiction that the message graph has a diameter $D > 2c$. Let p and q be two processors at distance D. Let Π_d be the set of processors at distance d from p. We can construct ru' by shifting all processors in Π_d by $d \cdot \varepsilon_{(1)}/2$, i.e., all receive events and jobs on some processor in Π_d happen $d \cdot \varepsilon_{(1)}/2$ time units earlier although with the same hardware clock readings (see Figure 7.4 for an example). As processors in Π_d only exchange messages with processors in Π_{d-1}, Π_d and Π_{d+1}, message delays are changed by $-\varepsilon_{(1)}/2$, 0 or $\varepsilon_{(1)}/2$. Thus, by Lemma 7.10, ru' is a valid rt-run conforming to failure model FAULT-FREE$_0$.

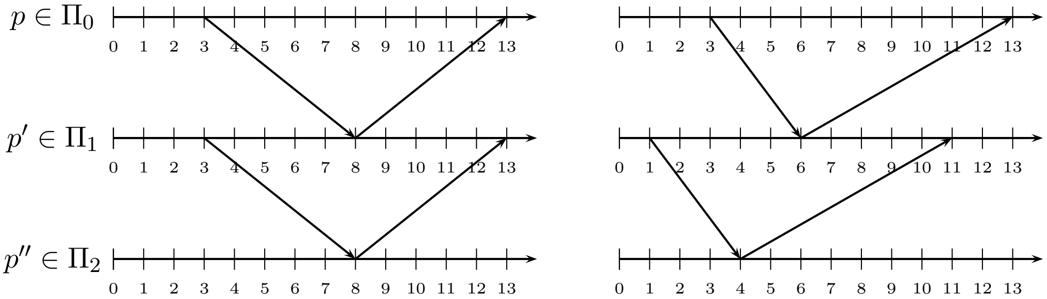
Let Δ and Δ' be the final (signed) differences between the adjusted clocks of p and q in ru and ru', respectively. As both rt-runs conform to FAULT-FREE$_0$ and $\mathcal{A}$ is assumed to be correct, $|\Delta| \leq c \cdot \varepsilon_{(1)}$ and $|\Delta'| \leq c \cdot \varepsilon_{(1)}$.

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

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

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

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



(a) Original

(b) Shifted

Figure 7.4.: Shifting by $d \cdot \varepsilon_{(1)}/2$ with $\varepsilon_{(1)} = 4$

7.2.4. Message Complexity

For clock synchronization to within some $\gamma < \varepsilon_{(1)}$ (i.e., $c < 1$), Lemma 7.12 implies that there exists a rt-run whose message graph has a diameter < 2, i.e., whose message graph is fully connected, and, therefore, has $\frac{n(n-1)}{2}$ edges. This leads to the following theorem:

Theorem 7.13. *Clock synchronization to within $\gamma < \varepsilon_{(1)}$ has a worst-case message complexity of $\Omega(n^2)$.*

Section 7.1 presented algorithms achieving optimal precision of $(1-\frac{1}{n})\varepsilon_{(1)}$ with $n(n-1) = \mathrm{O}(n^2)$ messages. Theorem 7.13 reveals that this bound is asymptotically tight. A weaker lower bound can be given for suboptimal clock synchronization by using the following simple graph-theoretical lemma:

Lemma 7.14. *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*, which states that an undirected graph with maximum degree d and diameter D has no more than $1 + d + d(d-1) + \cdots + d(d-1)^{D-1}$ nodes [GY04].

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

Combining Lemmas 7.12 and 7.14 shows that there is at least one processor in ru which exchanges (= sends or receives) at least $\lceil \sqrt[2c+1]{n} \rceil$ messages. More general:

Theorem 7.15. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$ in some real-time system s, there is at least one FAULT-FREE$_0$ rt-run in which at least one processor exchanges $\lceil \sqrt[2c+1]{n} \rceil$ messages.*

Corollary 7.16. *When synchronizing clocks to within $c \cdot \varepsilon_{(1)}$, there is no constant upper bound on the number of messages exchanged per processor.*

It is, however, possible to either bound the number of received messages *or* the number of sent messages per processor: Section 7.3.1 presents an algorithm synchronizing clocks to within $\varepsilon_{(1)}$ where every processor receives exactly one message. On the other hand, the algorithm in Section 7.3.2 also achieves this precision but bounds the number of sent messages per processor by 3.

7.2.5. Time Complexity

Theorem 7.15 immediately implies a lower bound on the worst-case time complexity of any algorithm that synchronizes clocks to within $c \cdot \varepsilon_{(1)}$: Some process p must exchange $m := \lceil \sqrt[2c+1]{n} \rceil$ messages, some k of which are received and the remaining ones are sent by p. Recalling $\delta^+_{(\ell)} \leq \ell \delta^+_{(1)}$ from Section 3.3[5], the algorithm's time complexity must be at least $\min_{k=0}^{m}(k \cdot \mu^+_{(0)} + \delta^+_{(m-k)})$.[6] Clearly, $k\mu^+_{(0)}$ is linear in k, so the interesting term is $\delta^+_{(m-k)}$, leading to the following corollary:

Corollary 7.17. *If multicasting a message in constant time is impossible, clock synchronization to within a constant factor of the message delay uncertainty cannot be done in constant time.*

In the case of optimal precision, n processors need to send and process at least $\frac{n(n-1)}{2}$ messages, so no algorithm can achieve a run time better than $\frac{n-1}{2}\mu^+_{(0)}$ or better than $\delta^+_{(\frac{n-1}{2})}$ (assuming optimal parallelism). This shows that the algorithm presented in Section 7.1.3 is not only tight regarding precision but also has asymptotically optimal time complexity $\mathrm{O}(n)$.

[5] Note that $\delta^+_{(\ell)} \leq \ell \delta^+_{(1)}$ follows directly from $\delta^+_{(i+j)} \leq \delta^+_{(i)} + \delta^+_{(j)}$.

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

```
1  var adj
2
3  procedure process_message(msg, current_hc)
4      /* start alg. by sending (INIT) to some proc. */
5      if msg = (INIT)
6          send current_hc to all other processors
7          adj ← 0
8      else
9          adj ← msg − current_hc + $\frac{\delta^-_{(n-1)}+\delta^+_{(n-1)}}{2}$
```

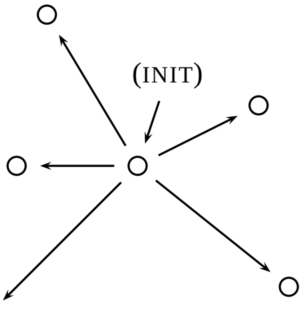


Figure 7.5.: Star Topology-based Clock Synchronization Algorithm

7.3. Achievable Precision for $o(n^2)$ Messages

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

7.3.1. Algorithm With Least Number of Messages

Figure 7.5 is actually a simpler version of the algorithm presented in Section 7.1: 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 7.4 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 from n (i.e., if constant-time broadcasting is possible), $\varepsilon_{(n-1)} = \varepsilon_{(1)}$ and the algorithm achieves this precision in constant time (w.r.t. n). Otherwise, the following modification puts the precision down to $\varepsilon_{(1)}$ in the general case as well:

- Do not send all messages during the same job but during subsequent jobs on the "master" processor.

- Replace $\delta^-_{(n-1)} + \delta^+_{(n-1)}$ in Line 9 with $\delta^-_{(1)} + \delta^+_{(1)}$.

The algorithm still exchanges only $n-1$ messages and has linear time complexity w.r.t. n. As Theorem 7.13 has shown, $\varepsilon_{(1)}$ is the best precision that can be achieved with less than a quadratic number of messages. As Corollary 7.17 has shown, this precision cannot be achieved in constant time in the general case.

7.3.2. Algorithm With Constant Bound on Number of Sent Messages per Processor

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

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

8. Optimal Remote Clock Estimation

As a first step towards optimal-precision clock synchronization in real-time systems with *drifting* clocks, we direct our attention to a deceptively simple subproblem of clock synchronization, namely, *remote clock estimation*. As outlined in the introduction, any existing clock synchronization algorithm can be reviewed in terms of a generic structure [Sch86], 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 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.

In this chapter, we provide an optimal solution for the problem of how to continuously estimate a source processor's clock; the algorithm is complemented by a matching lower bound on the achievable maximum clock reading error. Our results precisely quantify the effect of system parameters such as clock drift, message delay uncertainty and step duration on optimal clock estimation.

Since optimal remote clock estimation is trivially unsolvable in case of just a single crash failure, we assume a failure-free two-processor system with drifting clocks. Note carefully, however, that this does not restrict the applicability of our results to fault-free distributed clock synchronization algorithms: As outlined above, fault-tolerance in clock synchronization is usually maintained by choosing a fault-tolerant "convergence function", which calculates some correct new clock adjustment value despite some faulty remote clock readings. In fact, Chapter 9 will demonstrate how to incorporate our optimal clock estimation method in existing fault-tolerant clock synchronization algorithms.

8.1. Interval-Based Notation

A natural way to represent remote clock estimations would be 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 [LL84b, MS06a]. However, consider the two cases in Figure 8.1, 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.

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

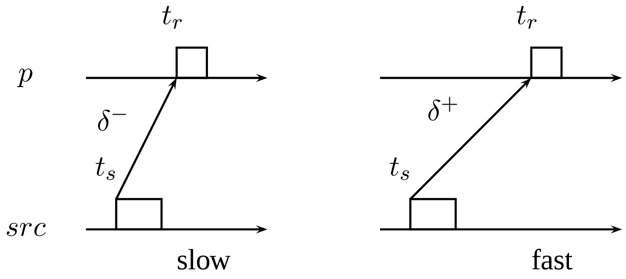


Figure 8.1.: p receiving a timestamped message from src.

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 8.1 (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:

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

8.2. Estimating a Remote Clock

The clock estimation algorithm presented in this section will repeatedly send messages from src to p as fast as possible.

8.2.1. System Model

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 Section 8.2.2, 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 also need to assume that

$$\mu^-_{(1)} \geq \mu^+_{(0)}. \tag{8.1}$$

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 eq. (8.1) 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).

Thus, in terms of the model introduced in the first part of this work, we assume a real-time system with $n = 2$ satisfying eq. (8.1) and a failure model FAULT-FREE$_{\rho_{src}, \rho_p}$, which is a natural extension of FAULT-FREE$_\rho(ru)$ to the case of two different hardware clock drift bounds.[1]

8.2.2. Algorithm

Consider the algorithm in Figure 8.2, 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 age, but 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 Section 8.2.1. 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

[1] Formally, the failure model specification of Section 4.2.2 is modified such that $\forall p : bounded\_drift(p, \rho)$ is replaced with $bounded\_drift(src, \rho_{src}) \wedge bounded\_drift(p, \rho_p)$.

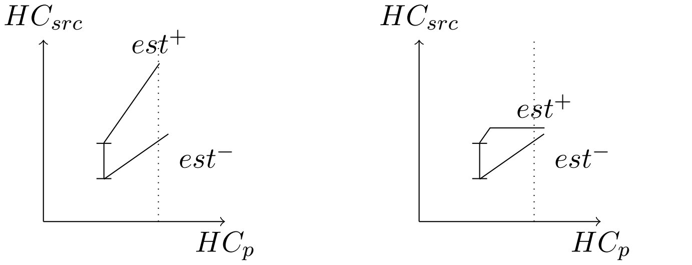
```
Processor $src$
procedure process_message(msg, current_hc )     /* start alg. by sending (SEND NOW) to src */
  send current_hc to $p$
  set (SEND NOW) timer for current_hc     /* will arrive at $end(current\_job)$ */

Processor $p$
var rcv_hc ← $-\infty$     /* local time of reception */
var send_hc ← $-\infty$     /* remote time of sending */

/* parameter current_hc of the following functions omitted for brevity */
function $age$ = current_hc − rcv_hc
public function $est^-$ = send_hc + $(1-\rho_{src})\left(\delta^- + age/(1+\rho_p)\right)$
public function $est^+$ = send_hc + $(1+\rho_{src})\left(\delta^+ + \min\{\mu^+_{(0)} + \mu^+_{(1)}, age/(1-\rho_p)\}\right)$

procedure process_message(msg, current_hc )
  var $HC_{src}$ ← msg
  if $HC_{src}$ > send_hc
    rcv_hc ← current_hc ; send_hc ← $HC_{src}$     /* one atomic step */
```

Figure 8.2.: Remote clock estimation algorithm



(a) Naive estimation

(b) Considering future messages

Figure 8.3.: p's estimate of src's hardware clock

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 quality of p's estimate of src's hardware clock depends on how well p can estimate Δ_m.

As can be inferred from Figure 8.3 and the definitions of est^- and est^+ in the algorithm, the maximum clock reading error is reached when the message is $(\mu^+_{(0)} + \mu^+_{(1)})(1-\rho_p)$ hardware clock time units old:

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

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

$$\nu = 2(\mu^+_{(0)} + \mu^+_{(1)})\rho_p \frac{\rho_p + \rho_{src}}{1 + \rho_p} \tag{8.2}$$

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

$$\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu \ . \tag{8.3}$$

8.2.3. Schedulability Analysis

Applying the system model restrictions from Section 8.2.1 to the algorithm allows us to make some general observations:

Observation 8.2. *Every timer set during some job starts processing at the end of that job. Formally, for all timer messages m_t: $(m_t \in trans(J)) \Rightarrow \exists J' : (begin(J') = end(J) \wedge msg(J') = m_t)$.*

Observation 8.3. *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., the 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 arbitrary scheduling policies, 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 11 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 have the admission control component filter these irrelevant messages, thus preventing them 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. Formally, this means that the scheduling/admission policy *pol* assumed in this proof can be arbitrary as long as no messages are dropped.

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).

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

- J'_i: The processing job of message i (on processor p).
- $\mathcal{F}_i := \{r : begin(J'_r) < begin(J'_i) \wedge r > i\}$: The set of "*fast*" messages $r > i$, that were processed (at 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.
- $f(i) := begin(J_i) + \delta^+ + \sum_{j \in \mathcal{F}_i \cup \{i\}} \mu^j_{(0)}$. This is an upper bound on the "*finishing*" real time by which all messages $\leq i$ have been processed. $\mu^j_{(0)}$ denotes the actual processing time $\in [\mu^-_{(0)}, \mu^+_{(0)}]$ of message j (= $duration(J'_j)$).

Observe that $f(i) \geq f(i-1)$, since $begin(J_i)$ 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 8.4. *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

$$\forall 1 \leq j \leq i-1 : end(J'_j) \leq f(i-1) \ . \tag{8.4}$$

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 eq. (8.4). Thus,

$$end(J'_i) > f(i) \ . \tag{8.5}$$

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 eq. (8.4), this leads to a contradiction with eq. (8.5). 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 $f(i) \geq f(i-1) + \mu^-_{(1)} + \mu^i_{(0)} - \mu^{i-1}_{(0)} + \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$.

Let $t = f(i) - \mu^i_{(0)} - \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$. 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 eq. (8.5):

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 $f(i) - t = \mu^i_{(0)} + \sum_{j \in \mathcal{F}^{new}} \mu^j_{(0)}$, 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 $f(i)$. □

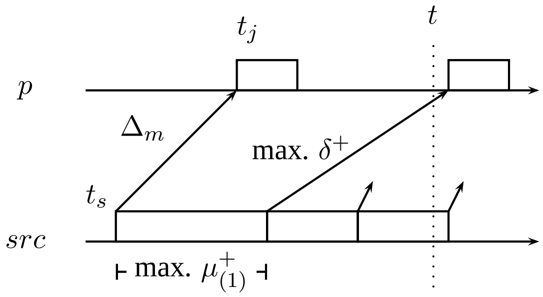


Figure 8.4.: Two consecutive messages from src to p

We call i a "relevant" message if $\mathcal{F}_i = \emptyset$. Thus, the following follows immediately from the previous lemma and the definition of $f(i)$:

Lemma 8.5. *The end-to-end delay Δ_m of every relevant message m is $\in [\delta^-, \delta^+]$.*

8.2.4. Proof of Correctness

Fix some rt-run ru and st-trace tr and let ev_{stable} be the $transition$ st-event[3] of the first relevant message from src to p. Such a message must exist: Due to our non-idling scheduler, the first message m arriving at p is also the first message being processed on p. Thus, $\mathcal{F}_m = \emptyset$ (otherwise, it could not be the first message being processed), which makes it a relevant message. 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 m starts and t_s being the starting time of the job sending m. Since $g \succ ev_{stable}$, such a message m must exist. Observe that Line 11 in the algorithm ensures that only relevant messages cause a state transition in p; thus, $s_p(g).send\_hc = HC_{src}(t_s)$ and $s_p(g).rcv\_hc = HC_p(t_j)$. Likewise, as defined in Line 5 of the algorithm, $age_p(g) = HC_p(time(g)) - HC_p(t_j)$.

Let $t = time(g)$ and $\Delta_m = t_j - t_s$ (cf. Figure 8.4). Note that Δ_m corresponds to δ_m, the message delay, plus any queuing delay m may experience. (For simplicity, Figure 8.4 shows a case without queuing.) Due to Lemma 8.5, we know that Δ_m is bounded by $[\delta^-, \delta^+]$. In addition, we define the following *drift factors*:

$$dr_p = \frac{HC_p(t) - HC_p(t_j)}{t - t_j} \tag{8.6a}$$

$$dr_{src} = \frac{HC_{src}(t) - HC_{src}(t_s)}{t - t_s} \tag{8.6b}$$

[3] For ease of presentation, we assume that lines 9–12 of the algorithm are executed in one single atomic step, i.e., we assume that there is only one $transition$ st-event for every job at 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 eq. (8.6a) 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,

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

Hence, we have:

Lemma 8.6. $HC_{src}(t) \geq est_p^-(g)$.

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

Lemma 8.7. $HC_{src}(t) \leq s_p(g).send\_hc + (age_p(g)/(1-\rho_p) + \delta^+)(1 + \rho_{src})$.

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

Lemma 8.8. $t - t_s \leq \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}$.

Proof. We will again use the numbering of messages as in Section 8.2.3. Recall Figure 8.4 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 message send jobs on src is at most $\mu^+_{(1)}$ (cf. Observation 8.3), $t'_s < t - \delta^+ - \mu^+_{(0)}$ 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 8.4 we know that $end(J'_{i+1}) \leq f(i+1) = t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu^j_{(0)}$.

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 $end(J'_{i+1}) \leq t'_s + \delta^+ + \sum_{j \in \mathcal{J}} \mu^j_{(0)}$ and that processing all of $\mathcal{J}$ takes at least $\sum_{j \in \mathcal{J}} \mu^j_{(0)}$ 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 some $t' < t$. This contradicts our assumption that $m = i$ is the last relevant message from src fully processed by p before g. □

Combining Lemma 8.8 with eq. (8.6b) results in

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

and hence

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

which, combined with Lemma 8.7 and the definition of est^+, yields the following result:

Lemma 8.9. $HC_{src}(t) \leq est^+_p(g)$.

Combining Lemmas 8.6 and 8.9 finally yields the following theorem, which proves that the algorithm in Figure 8.2 indeed solves the remote clock estimation problem according to Definition 8.1.

Theorem 8.10. *For all global states $g \succ ev_{stable}$, where ev_{stable} is the $transition$ st-event of the first message from src arriving at 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 $\nu = \mathrm{O}(\mu^+ \rho^2)$ given by eq. (8.2).

8.3. Lower Bound

In this section, we will show that the upper bound on Γ determined in Theorem 8.10 is tight, i.e., that the algorithm in Figure 8.2 is optimal with respect to the maximum clock reading error.

8.3.1. System Model

For the lower bound proof, we assume a two-processor system under failure model FAULT-FREE$_{\rho_{src},\rho_p}$ (analogous to the definition in Section 8.2.1). In addition, 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 Section 8.3.3, 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 without violating the message dependency between the $send$ and $process$ st-event of a message.

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 Figure 8.2, 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.

8.3.2. Proof

Assume by contradiction that there exists some deterministic algorithm $\mathcal{A}$ together with some scheduling/admission policy pol that allows processor p to continuously estimate processor src's hardware clock with a maximum clock reading error $\max\{est^+ - est^-\} < \Gamma$, with $\Gamma = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu$. 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 8.11. 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 $(ev_1, ev_2, \ldots, ev_\eta)$ and $(ev'_1, ev'_2, \ldots, ev'_{\eta'})$ 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, $HC^{tr}_p(time(ev_i)) = HC^{tr'}_p(time(ev'_i))$ 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 $tr[t_A, t_\Omega]$ will be 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 $\leq$; for example, $tr[0, t_\Omega)$ would contain only st-events ev with $0 \leq time(ev) < t_\Omega$.

Likewise, global states are sometimes used as boundaries: $tr[g_A, \ldots]$ and $tr[\ldots, g_\Omega]$ actually include 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 $p : tr[\ldots, g_\Omega] \approx tr'[\ldots, g'_\Omega]$.

Note: Since est^- and est^+ can be functions of the state of p and the current hardware clock value, it does not suffice to show that $s_p(g_1) = s_p(g_2)$ for some global states g_1 and g_2 of some indistinguishable st-traces tr_1 and tr_2. If we want to prove that est^- and est^+ are equal in g_1 and g_2, 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 $1 + \rho_{src}$.[4]
- 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. Section 8.3.1), 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 with the following differences (cf. Figure 8.5):

- src boots at time $t = \varepsilon = \delta^+ - \delta^-$ (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^-).

[4] Formally, $\frac{HC^{tr_1}_{src}(t) - HC^{tr_1}_{src}(t')}{t - t'} = 1 + \rho_{src}$, for all $t > t'$, i.e., the clock runs constantly at maximum speed.

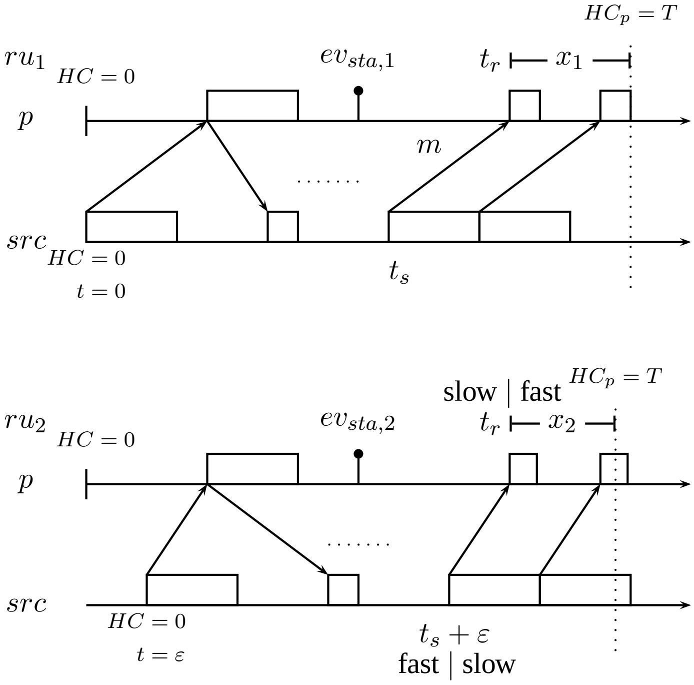


Figure 8.5.: ru_1 and ru_2 (timer messages not shown); $x_1 = \mu^+_{(0)} + \mu^+_{(1)}$; $x_2 = x_1 \frac{1-\rho_p}{1+\rho_p}$

- 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. Section 8.3.1). 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.[5]

Lemma 8.12. $p : tr_1[0, t_r] \approx tr_2[0, t_r]$ *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. □

[5] 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.

Since $time(ev_{sta,1}) \leq t_r$, this lemma also implies[6] the existence of a corresponding st-event $ev_{sta,2}$ in tr_2, in which p sets its $stable$ register.

Lemma 8.13. *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 8.13, p's hardware clock values at g_1 and g_2 are equal (denoted T and represented by the dotted line in Figure 8.5).

Lemma 8.14. $p : tr_1[0, g_1] \approx tr_2[0, g_2]$.

Proof. By Lemma 8.12, tr_1 and tr_2 are indistinguishable for src until t_s and $t_s + \varepsilon$, 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 $t_s + \varepsilon$. Both jobs send the same message m to p. Since our system model does not allow preemption, src's next job sending a message to p can start no earlier than at $t_s + \mu^+_{(1)}$ (tr_1) and at $t_s + \varepsilon + \mu^+_{(1)}$ (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. Figure 8.5).

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 times of jobs on p are shortened 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) which started before T' is still 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. Figure 8.5.) 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

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

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. □

Lemma 8.15. $HC^{tr_1}_{src}(time(g_1)) - HC^{tr_2}_{src}(time(g_2)) = \varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\mu^+_{(0)} + \mu^+_{(1)}) - \nu$.

Proof.

$$\begin{aligned} HC^{tr_1}_{src}(time(g_1)) &= HC^{tr_1}_{src}(t_r + \mu^+_{(0)} + \mu^+_{(1)}) = HC^{tr_1}_{src}(t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) \\ &= (t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)})(1+\rho_{src}) \\ &= t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)} + \rho_{src}(t_s + \delta^+ + \mu^+_{(0)} + \mu^+_{(1)}) \end{aligned}$$

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

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 eq. (8.2), denoting a small term in the order of $\mathrm{O}(\mu^+\rho^2)$. Thus,

$$\begin{aligned} &HC^{tr_2}_{src}(time(g_2)) = \\ &\quad t_s + \delta^- + \mu^+_{(0)} + \mu^+_{(1)} + \rho_{src}(t_s - \delta^- - \mu^+_{(0)} - \mu^+_{(1)}) - 2\rho_p(\mu^+_{(0)} + \mu^+_{(1)}) + \nu. \end{aligned}$$ □

We can now prove our lower bound theorem:

Theorem 8.16. *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. By the assumption that $\mathcal{A}$ is a correct algorithm that 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^+ on p, such that

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

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

Lemmas 8.13 and 8.14 have shown that $s_p(g_1) = s_p(g_2)$ and that $HC^{tr_1}_{src}(time(g_1)) = HC^{tr_2}_{src}(time(g_2))$. Since est^- and est^+ on p are functions of the local state and the hardware clock time, it holds that $est^-_p(g_1) = est^-_p(g_2)$ and $est^+_p(g_1) = est^+_p(g_2)$.

Lemma 8.15 reveals, however, that $HC^{tr_1}_{src}(time(g_1)) - HC^{tr_2}_{src}(time(g_2)) = \Gamma$, which provides the required contradiction. □

8.3.3. System Model Revisited

In Section 8.3.1, three assumptions were introduced, which simplify the lower bound proof. In this section, we will briefly discuss the consequences of dropping these assumptions.

1. In the rare case[7] that $\delta^- \geq \mu^+_{(0)}$ is not ensured, a potential message causality violation (i.e., a message's $process$ st-event occurring before its $send$ st-event) might force the adversary to execute a $send$ or (preceding) $transition$ st-event no later than δ^- time units after the beginning of the job. Thus, the precision lower bound for the general case is $\varepsilon + \rho_{src}(\delta^- + \delta^+) + 2(\rho_{src} + \rho_p)(\min\{\delta^-, \mu^+_{(0)}\} + \mu^+_{(1)}) - \nu'$, i.e., $\mu^+_{(0)}$ gets replaced by $\min\{\delta^-, \mu^+_{(0)}\}$. Analogously, ν' equals ν with $\mu^+_{(0)}$ replaced by this minimum expression.

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 Figure 8.5. Informally, this can be guaranteed due to the fact that even eventual properties are always satisfied within bounded time in a closed model like our real-time model (where all delays are bounded), see e.g. [RS08].

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 Section 8.3.2. Since only src receives messages here, only src can detect a difference between two rt-runs with different drift rates. Consider Figure 8.5 with the labels p and src reversed. In complete analogy to Lemma 8.14, 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.[8] Therefore we have an additional δ^- for the message transmission plus $\mu^+_{(0)}$ (or δ^-, 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 Section 8.3.2, finally leading to the same contradiction.

[7] Recall that δ^- and δ^+ are measured from the beginning of the job sending the message rather than from the $send$ st-event. Therefore, these values include local processing until the message is sent plus the message transmission delay.

[8] 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.

9. Examples of Fault-Tolerant Clock Synchronization

In this chapter, we will move from the two-processor clock estimation problem to its application in external and internal clock synchronization (with drifting clocks and failures, in contrast to Chapter 7).

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 Section 3.3, $\delta^-_{(\ell)}$, $\delta^+_{(\ell)}$ as well as $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ are assumed to be non-decreasing with respect to ℓ.

9.1. 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 Section 8.2, 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 Section 8.2 shows that, if src is correct, the worst case accuracy for any correct receiver p is $\alpha = \Gamma/2$ with

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

(cf. Theorem 8.10), where ℓ depends on the broadcasting method, $\dot{\mu}$ is the transmission period (see below), and $\dot{\nu} = \mathrm{O}(\dot{\mu}\rho^2)$ 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 $(n-1) \cdot \mu^+_{(1)}$.

9.2. Internal Clock Synchronization

As outlined in the introduction of Chapter 8, remote clock estimation is only a small, albeit important, part of the internal clock synchronization problem. In [FC95b], 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 [FC95b] 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 in the resynchronization times of different processors. Λ is the maximum clock reading error margin, i.e., $\Lambda = \Gamma/2$ in our setting.

In this section 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 [FC95b].

9.2.1. System Model

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.

Every $round\_len$ clock time units (the *resynchronization period*), 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, thereby satisfying two conditions:

(C1) 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.

(C2) Broadcasting 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.

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 Γ 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 (and instantly)

- storing the content of the message in some variable $send\_cv[q]$ and
- storing p's current hardware clock value in some variable $rcv\_hc[q]$[1],

then there would be no need to schedule a receive job on p's CPU at all.

Clearly, in general case, relying on instantaneous processing of incoming messages is an unreasonable assumption—actually, demonstrating this fact is one of the main points of this thesis. Nevertheless, 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, well aware that this limits the applicability of the algorithm in this section to systems where some hardware-based solution ensures that the message as well as its arrival time are recorded by a network controller in short, bounded time (which must be added to δ^+) [SKM$^+$00, SSHL97, HSS$^+$02].

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 8.10) by dropping all $\mu^+_{(0)}$ terms.

The clock synchronization 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; formally, this corresponds to failure model f-BYZANTINE$_\rho$ and a real-time system s with $n > 3f$ 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 (except for the one processing the (INIT)

[1] 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).

message) send $n-1$ messages, we will abbreviate $\delta^-_{(n-1)}$, $\delta^+_{(n-1)}$ and $\mu^+_{(n-1)}$ with δ^-, δ^+ and μ^+.

9.2.2. Booting

As initial synchronization is outside the scope of this thesis, we will assume that

- $HC_p(0) \in [0, \pi_I]$ for every processor p, and
- the (INIT) message for each processor arrives "shortly after" $t = 0$. In particular, it arrives at least $\mu^+_{(0)}$ time units before p's hardware clock reaches $round\_len - pre$.

(See below for the definitions of π_I, $round\_len$ and pre.)

9.2.3. Algorithm

Figure 9.1 shows the pseudo-code of the algorithm of [FC95b] in conjunction with our optimal clock estimation method. It includes a few optimizations specifically designed for large round durations:

- Processors do not broadcast continuously 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 (always equal to k or $k-1$) 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 primarily 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.

Due to the requirement of broadcasting its own clock value, many jobs are already active around local time $T = k \cdot round\_len$ at every processor. 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 22 of the algorithm). This, however, means that the state transition might not occur exactly at clock time $k \cdot round\_len$, but 0 to $2\mu^+$ real-time units later.[2]

[2] The case of $2\mu^+$ occurs when the logical clock reads $k \cdot round\_len - x$ at the start of some job J_1, 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.

Network controller of processor p

```
var send_cv [], rcv_hc []

upon receiving (CLOCK VALUE, $cv$) from some processor $q$        /* done in zero time */
  if ($cv >$ send_cv[$q$]) or (send_cv[$q$] is empty)
    rcv_hc[$q$] ← current_hc (); send_cv[$q$] ← $cv$
```

Processor p

```
const round_len, pre, post
var adj [] (adj [1] ← 0)
var k ← 1, bc_k ← 1      /* local clock round number, current broadcast round number */

function $cfn(my\_ac, estimates)$        /* convergence function as specified in [FC95b] */

/* Simplified notation − please read the note at the end of Section 9.2.3 */
function $age(q)$ = current_hc − rcv_hc[$q$]
function $est^{-}(q)$ = send_cv[$q$] + $(1-\rho)\left(\delta^{-} + age(q)/(1+\rho)\right)$
function $est^{+}(q)$ = send_cv[$q$] + $(1+\rho)\left(\delta^{+} + \min\{\mu^{+}, age(q)/(1-\rho)\}\right)$

function $AC^{i}(hc)$ = $hc$ + $adj[i]$          /* convert hardware clocks to adjusted clocks... */
function $HC^{i}(ac)$ = $ac - adj[i]$        /* ...and vice−versa */

procedure process_message(msg, current_hc, rcv_hc [], send_cv [])
  if msg = (INIT)
    set (SEND NOW) timer for round_len − pre  /* = $HC^{\mathrm{k}}$(k · round_len − pre) */

  if msg = (SEND NOW)
    send (CLOCK VALUE, $AC^{\mathrm{bc\_k}}$(current_hc)) to all

    if (bc_k = k) and ($AC^{\mathrm{k}}$(current_hc) $\geq$ k · round_len)   /* start new round? */
      k ← k+1; adj [k] ← $cfn(AC^{\mathrm{bc\_k}}$(current_hc), for all $q$ : $(est^{-}(q) + est^{+}(q))/2) -$ current_hc

    if $AC^{\mathrm{bc\_k}}$(current_hc) $<$ bc_k · round_len + post  /* continue or stop broadcasting */
      set (SEND NOW) timer for current_hc   /* timer will arrive at $end(current\_job)$ */
    else
      bc_k ← k                /* prepare for next round's broadcast */
      set (SEND NOW) timer for $HC^{\mathrm{k}}$(k · round_len − pre)
```

Figure 9.1.: Internal clock synchronization; [FC95b] combined with optimal clock reading

Like hardware clock values, we assume that the processor reads the values of $send\_cv$ and rcv_hc only at the very beginning of the job, i.e., $current\_hc$, $send\_cv$ and rcv_hc, when used in some job J, represent a snapshot of the state at time $begin(J)$. That way, we ensure that the transition sequence of a job is still independent of the actual times of the $transition$ st-event, and we avoid unrealistic effects such as rcv_hc (when read instantly) being larger than $current\_hc$ (when read at the beginning of the job). Consequently, $send\_cv$ and rcv_hc are not modeled as global variables but rather as parameters of $process\_message$.

Consequently, it should be noted that the names of the functions in Lines 8–10 ($age(q)$, $est^-(q)$, $est^+(q)$) were chosen just for notational convenience. In fact, age is a function $age(current\_hc, rcv\_hc[q])$; the same holds for est^- and est^+, which also include a third parameter $send\_cv[q]$. This is important because it means that the parameters for the convergence function in line 23 are based on the state of $current\_hc$, rcv_hc and $send\_cv$ at the begin time of the job (i.e., the values passed to $process\_message$) rather than at the time of the $transition$ st-event.

9.2.4. Analysis

The precision analysis in [FC95b] is based on a set of assumptions, which involve the following constants: Λ (maximum clock reading error margin[3]), 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).

t_p^k denotes the real-time by which processor p starts round k. Since the convergence function is called with data corresponding to the begin time of the job making the round switch (for the reasons outlined in the previous section), the begin time of the job must be chosen as t_p^k rather than the actual time of the $transition$ st-event representing line 23. Thus, from the point of view of the convergence function, the k-th clock is started at time t_p^k, although, from the clock synchronization perspective, this clock is not in use until up to μ^+ time units later. Therefore, we will have to compensate for this fact when determining our precision bound.

Figure 9.2 depicts an example rt-run during a round switch $i \to i+1$, with $T = i \cdot round\_len$. The current values of variables bc_k and k as well as the i-th logical clock are shown explicitly.

Theorem 9.1 (Theorem 1 of [FC95b]). *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., $\beta \leq r_{min}$.*

[3] Note that Λ is $\Gamma/2$: We defined the remote estimation interval as $[est^-, est^+]$, with Γ bounding $est^+ - est^-$. By contrast, [FC95b] defines a remote clock reading as a single value with a symmetric error of at most $\pm\Lambda$.

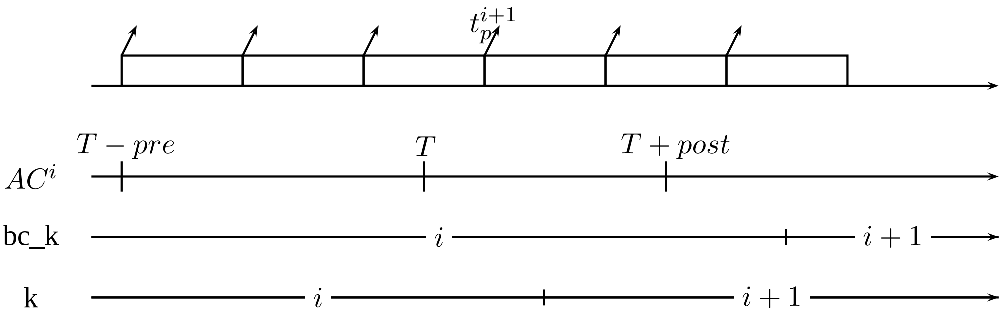


Figure 9.2.: Round switch from round i to $i+1$

(A5) $\pi_I \geq 2\Lambda + 2\rho r_{max} + 2\rho\beta$.

Then, the algorithm of [FC95b] guarantees that all logical (= adjusted) clocks of correct processors p *and* q *are synchronized to within a bound of* $\pi = \pi_I + 2\rho r_{max}$*, i.e.,*

$$|AC_p(g) - AC_q(g)| \leq \pi$$

for all global states g*, with* $AC_p(g) = HC_p(time(g)) + s_p(g).adj[s_p(g).k])$*. 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* $adj[k] - adj[k-1]$ *is* $\pm 2\rho r_{max}$ *on all processors.*

This is what would hold if the round switching state transition occurred at the very beginning of the job. Since this is not guaranteed in the real-time computing model,

- the actual clock synchronization precision, i.e., the value by which $|AC_p(g) - AC_q(g)|$ can be bounded, is $\pi_A = \pi + 2\rho\mu^+$ rather than π; a term $2\rho\mu^+$ is added to compensate for the fact that the "old" round $k-1$ clock can still be in use for another μ^+ time units after t_p^k;
- the precision values π_I and π are only relevant for analysis. For example, π_I holds at $t = \max_{p \in \Pi}\{t_p^k\}$ for the adjusted round k clocks $AC_p^k(t) = HC_p(t) + adj_p^k$. For purposes of analysis, these values are well-defined at time t since $HC_p(t)$ is part of the rt-run and adj_p^k, the value that p's variable adj[k] will have once the $transition$ st-event for line 23 has occurred, is a deterministic function of p's $send\_cv$, rcv_hc and $current\_hc$ variables at time $t_p^k \leq t$. Nevertheless, the algorithm might still not know the value of adj_p^k at time t (and, thus, still use the round $k-1$ clock).

Apart from that, the combined algorithm in Figure 9.1 starts new clocks in the same way as [FC95b]. Therefore, this theorem applies to our algorithm as well, with the aforementioned differences.

(A1) is satisfied by our booting assumption (Section 9.2.2), and (A5) can be guaranteed by choosing a suitably large π_I. 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 9.1, which is not stated explicitly but can be deduced from the proofs of [FC95b].

Lemma 9.2. *If* $round\_len$, r_{min}, r_{max} *and* β *are chosen such that*

$$round\_len \geq \pi + \mu^+(1+\rho) + r_{min}(1+\rho) \tag{9.1}$$

$$r_{max} \geq \frac{\pi + round\_len}{1-\rho} + \mu^+ + \beta \tag{9.2}$$

$$\beta \geq \frac{\pi + round\_len}{1-\rho} + \mu^+ - r_{min} \tag{9.3}$$

$$r_{min} \geq \beta, \tag{9.4}$$

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

Proof. (A4) holds trivially, since it equals (9.4). Assume by induction that (A2) and (A3) hold for rounds 1 to k. By (an inductive version of) Theorem 9.1, 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 our booting assumption.

Let q be the last processor switching to round k. Since this round switch was triggered by q's round $k-1$ clock reaching $(k-1)\cdot round\_len$, t_q^k occurs 0 to μ^+ real-time units later, and q's clock value was not changed by more than $2\rho r_{max}$, it follows that q's clock k value at time t_q^k is within $(k-1)\cdot round\_len + [-2\rho r_{max}, 2\rho r_{max}] + [0, \mu^+(1+\rho)]$. Recall that, at t_q^k, all round k clocks[4] are synchronized to within π_I with q and that $\pi = \pi_I + 2\rho r_{max}$. Thus, at t_q^k, it holds for the round k clock values cv_p of every correct clock p:

$$cv_p \in (k-1)\cdot round\_len + [-\pi, +\pi] + [0, \mu^+(1+\rho)] \tag{9.5}$$

Using (9.1), we can bound the number of hardware clock units left until $k \cdot round\_len$ is reached at p:

$$k \cdot round\_len - cv_p \in [r_{min}(1+\rho), \pi + round\_len] \tag{9.6}$$

Note that t_p^{k+1} might occur at most μ^+ 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 (9.2) result in

$$t_p^{k+1} - t_p^k \geq \frac{r_{min}(1+\rho)}{1+\rho} = r_{min} \qquad \text{and}$$

$$t_p^{k+1} - t_p^k \leq \beta + \frac{\pi + round\_len}{1-\rho} + \mu^+ \leq r_{max},$$

thus showing 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: (9.6) ensures that no processor can reach $k \cdot round\_len$ on its round k clock earlier than at real time $t_q^k + r_{min}$ nor later than at real time $t_q^k + \frac{\pi + round\_len}{1-\rho}$. Adding the μ^+ that may lie in between reaching $k \cdot round\_len$ and t_p^{k+1}, (9.3) shows that (A3) holds: $|t_p^{k+1} - t_q^{k+1}| \leq \beta$, for all p and q. □

[4] Although the algorithm might not know the value of its round k clock yet, we can still use it for analysis.

We can hence apply Theorem 9.1 to immediately get:

Theorem 9.3. *For a sufficiently large round length and sufficiently close initial synchronization, the algorithm of Figure 9.1 solves internal clock synchronization within* $\pi_A = 2\Gamma + 4\rho r_{max} + 2\rho\beta + 2\rho\mu^+$ *with* $\Gamma = \varepsilon + \rho(\delta^- + \delta^+) + 4\rho\mu^+ - \nu$ *and* $\nu = O(\mu^+\rho^2)$.

10. Conclusions

As argued in the introduction, all the standard distributed computing models in use nowadays either rely on the zero step-time assumption or use some other concept that abstracts away queuing effects at the receiver side. In the first part of this thesis, we defined and analyzed the real-time computing model, which differs from the classic non-lock-step synchronous model by just providing non-zero computing step times. Our aim is to establish the real-time computing model as a well-founded alternative to the classic models used in the analysis of distributed algorithms. Therefore, the definitions in this work strive to be as generic (and formal) as possible; the model is neither restricted to one particular type of failure nor to one particular class of problems. On the contrary: abstract *failure models* and *st-problems* allow for high flexibility in the definition of real-world failure scenarios and problem specifications.

This universality is beneficial for the transformations presented in Chapter 6, which are designed for arbitrary problems (and, in the case of simulation $\mathcal{S}_{\underline{A}}$, for arbitrary failure models). In addition, the reuse of classic algorithms in the real-time model illustrates the unique challenges arising when implementing a result of classic distributed computing research in a real system: End-to-end delay bounds must be determined via schedulability analysis (*feasible assignment*), state transitions occur at slightly different times (*μ^+-shuffle of the problem*), and measures might need to be taken to ensure the timely delivery of timer messages (*$T_{R\to C}$-compatibility between classic and real-time failure models*).

Although these transformations provide a very powerful and general tool for the study of problems in the real-time scenario, tight bounds usually require further analysis: We revisit a well-known synchronization problem—optimal deterministic clock synchronization in the drift- and failure-free case—in our real-time computing model. As it turned out, the classic analysis gives too optimistic results, supporting our claim that some properties of real systems are inaccurately or even wrongly captured when resorting to classic zero step-time models.

The naive approach of transforming the tight precision bound of $(1 - \frac{1}{n})\underline{\varepsilon}$ into the real-time model resulted in a lower bound of $(1 - \frac{1}{n})\varepsilon_{(1)}$ and an upper bound of $(1 - \frac{1}{n})(\varepsilon_{(n-1)} + \mu^+_{(n-1)} + (n-2) \cdot \mu^+_{(0)})$ (Theorems 7.2 and 7.3)—a gap, which was closed by devising a clock synchronization algorithm specifically designed for the needs of a system with non-zero step time duration.

In a sense, the results in the classic computing model were both too optimistic and too pessimistic at the same time: On the one hand, synchronization with optimal precision is possible in constant time in the classic computing model, whereas optimal synchronization in the real-time computing model has a time complexity of $\Theta(n)$. On the other hand, the best precision achievable in the classic model is only $(1 - \frac{1}{n})\underline{\varepsilon}$, whereas it turned out to be $(1 - \frac{1}{n})\varepsilon_{(1)}$ in the real-time model. This might not seem like a big difference; however, from the system model point of view, $\underline{\varepsilon} = \underline{\delta}^+ - \underline{\delta}^-$ is the uncertainty of the *end-to-end delay*, a quantity that encompasses real-world message transmission delay, queuing delays and processing delays. In contrast, $\varepsilon_{(1)} = \delta^+_{(1)} - \delta^-_{(1)}$ refers to the uncertainty of just the message transmission delay.

The explanation for this phenomenon is that the zero step-time assumption gives too much power both to the algorithm (by being able to process multiple incoming messages in a very short time) and to the adversary (by being able to assign end-to-end delays to messages that are not justified by the arrival patterns). As we have shown, analysis under the real-time computing model yields more realistic results.

A summary of all our findings related to drift-free clock synchronization in the real-time model presented in this thesis can be found in Table 10.1. Note that these results also cover the non-optimal case: 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.

Constraint	**Lower Bound**	**Matching Upper Bound**
-	Precision $\geq (1-\frac{1}{n})\varepsilon_{(1)}$ *Proof: Theorem 7.3*	Precision $\leq (1-\frac{1}{n})\varepsilon_{(1)}$ *Algorithm: Section 7.1.3*
-	Msg. complexity = $\Omega(n)$ *Proof: obvious*	Msg. complexity = $\mathrm{O}(n)$ *Algorithm: Section 7.3.1*
-	$\exists$ one processor exchanging $\Omega(\sqrt[2(\gamma/\varepsilon_{(1)})+1]{n})$ msgs. *Proof: Theorem 7.15*	
Achieve best precision (= $(1-\frac{1}{n})\varepsilon_{(1)}$)	Msg. complexity = $\Omega(n^2)$ Time complexity = $\Omega(n)$ *Proof: Section 7.2*	Msg. complexity = $\mathrm{O}(n^2)$ Time complexity = $\mathrm{O}(n)$ *Algorithms: Section 7.1*
Achieve best msg. complexity (= $\mathrm{O}(n)$)	Precision $\geq \varepsilon_{(1)}$ *Proof: Theorem 7.13*	Precision $\leq \varepsilon_{(1)}$ *Algorithm: Section 7.3.1*

Table 10.1.: Summary of Results on Drift-Free Clock Synchronization

As a first step towards drifting clock synchronization, 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. 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 our results on drift-free clock synchronization.

- 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.

We close this thesis by combining our optimal clock reading method with the optimal convergence function of [FC95b] into a fault-tolerant internal clock synchronization algorithm, guaranteeing a precision of $\varepsilon + \rho(\delta^- + \delta^+) + 4\rho\mu^+ - \nu$. The question of whether this combined algorithm is optimal is left open (see below).

The relationship between time complexity and precision in the drift-free case also sheds some light on a new aspect of clock synchronization algorithm design: Clearly, all our lower bound results also hold for drifting clocks. 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 precise but also more complex algorithm, depending on the system parameters.

Future Work/Open Issues

In this thesis, we created a sound foundation for a reconciliation of the distributed computing and the real-time systems perspective, which has been lacking up to now. However, our work has not only provided answers, but has also raised a lot of new and exciting open questions for future research:

The Real-Time Computing Model

- Removing the zero step-time assumption from the classic computing model was a particularly interesting extension, since, as a direct consequence, queuing delays—previously hidden in the end-to-end delay—became visible. Since all other assumptions of the classic model have been carried over to the real-time computing model, however, one wonders whether and which other aspects are worth being teased out and modeled explicitly. To name a few examples: in the real-time computing model, message sizes can be unbounded, jobs can perform computations of arbitrary complexity, preemption is impossible, and the scheduling policy must be non-idling. It would be interesting to know whether dropping any of these assumptions produces insights that justify the additional model complexity.

  We assume that message size is a particularly interesting target here. Starting multiple instances of an algorithm or echoing all data received so far are common design patterns in the area of fault-tolerant distributed computing. Consider, for example, lock-step round based algorithms: Comparing such algorithms in an extended real-time computing model with bounded message size could reveal that some algorithm requiring a large amount of rounds for completion might actually perform better than a competitor with less rounds, if the first algorithm requires less data to be exchanged per round.

- Our simulation $\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$, enabling real-time algorithms to be run in a classic system, requires failure model FAULT-FREE$_\rho(ex)$ with a "sufficiently small" clock drift ρ (cf. Section 6.3.2). This failure model has been chosen for ease of presentation. In fact, neither bounded drift nor fault-freeness of the processors are mandatory for this transformation. With respect to the hardware clocks, any other failure model guaranteeing that there exists some $\tilde{\mu}_{(\ell)}$ such that Definition 6.12 is satisfied would be sufficient. With respect to processor faults, a generalization to f-CRASH should also be fairly easy. We believe that an extension a Byzantine failure model is possible as well, but it will require a few changes to the $T_{C \to R}$ transformation rules.

- The classic and the real-time computing model described in this thesis assume that algorithms are deterministic, i.e., that the transition function $\mathcal{A}(message, oldstate, hardware\_clock\_time)$ of some algorithm $\mathcal{A}$ returns a single transition sequence. A generalization of the system models and the transformations of Chapter 6 to the non-deterministic case, where a *set* of possible state transition sequences is returned by the transition function, should be fairly straightforward.

- Since the real-time computing model allows us to apply real-time schedulability analysis techniques to distributed algorithms, we are looking out for problems and algorithms where trivial upper bounds on end-to-end delays do not suffice and such an analysis is required.

- As outlined in the introduction, one of our mid-term targets is to analytically verify the assumptions of the Theta model and the ABC model [RS08], which stipulate that—for certain algorithms—there is a certain correlation between queuing delays in different parts of a distributed system. Developing the real-time computing model was a mandatory first step for achieving this goal.

Drift-Free Clock Synchronization

- To show that constant-time synchronization to within $c \cdot \varepsilon_{(1)}$ (for any constant $c \in \mathbb{R}^+$) is impossible (unless a constant-time broadcast primitive exists), we proved that there exists at least one processor which exchanges at least $\lceil \sqrt[2c+1]{n} \rceil$ messages. However, all algorithms presented in this thesis have at least one processor exchanging (i.e., sending or receiving) n messages. So, although the lower bound served its purpose by helping us to derive a time complexity result, we do not think that this bound it tight, and we are curious about the (asymptotic) number of messages strictly required for drift-free clock synchronization.

- For the case where an asymptotically tight bound on the message complexity exists, namely, in the case of optimal precision, there is still a gap between the precise number of messages: The lower bound requires one message between every pair of processors, while our algorithms require two.

Drifting Clock Synchronization

- Clearly, the most obvious question with respect to the algorithm presented in Section 9.2 is: Is this algorithm optimal? After all, we used an optimal convergence function and an optimal remote clock reading method.

  Actually, there is a subtle difference between the remote clock estimation problem analyzed in Chapter 8 (*continuous* clock estimation) and the precise requirements of a round-based internal clock synchronization algorithm: Whereas the former problem aims at establishing a worst-case bound on the clock reading error *in the two-processor* *case, at all points in time*, the round-based clock synchronization algorithm requires a good estimate *in the multi-processor case, at the exact point of starting the new clock*. As the proof of our (continuous) remote clock estimation algorithm shows, the estimation error is not constant but varies; for example, right after receiving a message, the error is smaller than shortly afterwards. Thus, in a thorough analysis of the multiprocessor, single-shot case in the real-time computing model, it might turn out that having the messages of all processors arrive at almost the same time (but not too close, lest we get queuing delays) might be beneficial. We are not sure whether this makes a difference for round-based clock synchronization or not. Until this issue has been resolved, we cannot claim that our algorithm is optimal.

- In contrast to the more general results of Chapters 7 and 8, the clock synchronization algorithm of Section 9.2 assumes that processing incoming messages (i.e., recording the content of the message and the arrival time) can be done instantaneously (or within constantly bounded time, which can be added to $[\delta^-, \delta^+]$) upon arrival. Although hardware-based solutions to this problem are by now common practice in the analysis of clock synchronization in real-time systems (cf. the discussion and references in Section 9.2.1), it is not hard to think of many real-world systems where such specialized hardware is not available. Thus, clock synchronization without this assumption is a problem worth pursuing. In particular, we believe that the optimal trade-off between exchanging many messages (possibly causing queuing effects) and few messages (possibly causing clock readings to be out-of-date) will pose an interesting and insightful challenge.

- Although a worst-case precision bound has been established for the algorithm of Section 9.2, the average-case performance could still be improved by a few simple modifications:

  - The convergence function of [FC95b] 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 Section 8.1, this is not possible when a minimal symmetric error is needed. Therefore, replacing the convergence function of [FC95b] by an optimal one that supports asymmetric error margins—like the one in [SS03]—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 [Cri89].

Nomenclature

$\approx$	Indistinguishability, page 102
$\rightarrow^{seq}$	Causal dependency w.r.t. sequence seq, page 17
$\xrightarrow{L}^{seq}$	Local dependency w.r.t. sequence seq, page 17
$\xrightarrow{M}$	Message dependency, page 17
$\prec^{seq}$	Ordering relation of sequence seq
α	Offset for the arrival or processing of timer messages, page 55
$\underline{\mathcal{A}}$	Algorithm of the classic computing model or its transition function, page 18
$\mathcal{A}$	Algorithm of the real-time computing model or its transition function, page 27
ac	Computing step (action) in the classic computing model, page 20
$AC_p(g)$	Adjusted clock of processor p at global state g, page 79
$\underline{\mathcal{C}}$	Failure model in the classic computing model, page 32
$\mathcal{C}$	Failure model in the real-time computing model, page 32
Γ	Maximum remote clock reading error, page 94
γ	Clock synchronization precision in the drift-free case, page 79
$\underline{\delta}^-$, $\underline{\delta}^+$	Bounds on the message delay in the classic computing model, page 18
δ^-, δ^+	Bounds on the message delay in the real-time computing model, page 25
D	Drop event in the real-time computing model, page 27
Δ	End-to-end delay in the real-time computing model, page 26
$\underline{\varepsilon}$	Message delay uncertainty in the classic computing model, page 21
ε	Message delay uncertainty in the real-time computing model, page 29
est^-, est^+	Bounds on the estimated remote clock value, page 94
ev	State transition event, page 39

ex Execution in the classic computing model, page 20

f Number of faulty processors, page 31

g Global state, page 41

$gstates(tr)$ Set of all global states appearing in tr, page 42

HC_p Hardware clock of processor p, page 18

$Init_p(\underline{\mathcal{A}})$ Set of initial states of algorithm $\underline{\mathcal{A}}$, page 18

$istate_p$ Initial state of processor p, page 19

J Job in the real-time computing model, page 27

JD Job or drop event in the real-time computing model, page 32

ℓ Number of messages sent in a job, page 28

μ^-, μ^+ Bounds on the processing delay in the real-time computing model, page 23

m_o Ordinary message, page 31

m_t Timer message, page 31

n Number of processors, page 18

ω Queuing delay in the real-time computing model, page 26

Π Set of processors, page 18

$\mathcal{P}$ State-based problem, page 42

$\mathcal{P}^*_{\mu^+}$ μ^+-shuffle of problem $\mathcal{P}$, page 49

$\mathcal{P}^>_{\mathcal{V}}$ Simulation-invariant $\mathcal{V}$-extension of problem $\mathcal{P}$, page 50

pol Scheduling/admission control policy, page 25

R Receive event in the real-time computing model, page 26

ρ_p Drift rate of processor p, page 19

ru Real-time run in the real-time computing model, page 26

$\underline{s}$ System in the classic computing model, page 21

s System in the real-time computing model, page 28

$\mathcal{S}_{\underline{\mathcal{A}}}$ Simulation algorithm for reusing a classic algorithm $\underline{\mathcal{A}}$ in a real-time system, page 52

$\underline{\mathcal{S}}_{\tilde{\mu},\mathcal{A},pol}$ Simulation algorithm for reusing a real-time algorithm $\mathcal{A}$ in the classic model, page 62

$\underline{\mathcal{S}}'_{\tilde{\mu},\tilde{\delta},\mathcal{A},pol}$ Extended simulation algorithm for reusing a real-time algorithm $\mathcal{A}$ in the classic model, page 76

$sHC(m_t)$ Designated arrival (hardware clock) time for timer message m_t, page 31

s_p Local state of processor p, page 39

$T_{C \to R}$ Transformation for reusing a real-time algorithm in the classic model, page 66

$T_{R \to C}$ Transformation for reusing a classic algorithm in the real-time model, page 53

tr State transition trace, page 40

Bibliography

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

[ADLS94] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

[AHR93] Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization under different delay assumptions. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 109–120, New York, NY, USA, 1993. ACM Press.

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

[Alb05] Daniel Albeseder. Evaluation of message delay correlation in distributed systems. In *Proceedings of the Third Workshop on Intelligent Solutions for Embedded Systems*, Hamburg, Germany, May 2005.

[AP98] Emmanuelle Anceaume and Isabelle Puaut. Performance evaluation of clock synchronization algorithms. Technical Report RR-3526, INRIA, 1998.

[Arv94] K. Arvind. Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474–487, May 1994.

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

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

[BHR90] Sanjoy K. Baruah, R. R. Howell, and Louis E. Rosier. On preemptive scheduling of periodic, real-time tasks on one processor. In *MFCS '90: Proceedings on Mathematical foundations of computer science 1990*, pages 173–179, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

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

[But97] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

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

[BW06] Martin Biely and Josef Widder. Optimal message-driven implementation of Omega with mute processes. In *Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*, volume 4280 of *LNCS*, pages 110–121, Dallas, TX, USA, November 2006. Springer Verlag.

[Cri89] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DHS86] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[EK73] C. Ellingson and R. Kulpinski. Dissemination of system time. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 21(5):605–624, May 1973.

[FC95a] Christof Fetzer and Flaviu Cristian. Lower bounds for function based clock synchronization. In *Proceedings 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, CA, August 1995.

[FC95b] Christof Fetzer and Flaviu Cristian. An optimal internal clock synchronization algorithm. In *Proceedings of the Tenth Annual Conference on Computer Assurance (COMPASS'95)*, pages 187–196, Gaithersburg, MD, June 1995.

[Fid88] C. J. Fidge. Timestamps in message-passing systems that preserve partial ordering. In *Proc. 11th Australian Computing Conf.*, pages 56–66, 1988.

[FL04] Rui Fan and Nancy Lynch. Gradient clock synchronization. In *Proceedings of the Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 320–327, July 25-28, 2004.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[GY04] Jonathan L. Gross and Jay Yellen, editors. *Handbook of Graph Theory*, chapter 4. CRC Press, 2004.

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

[HMM85] J Y Halpern, N Megiddo, and A A Munshi. Optimal precision in the presence of uncertainty. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 346–355, New York, NY, USA, 1985. ACM Press.

[HSS$^+$02] Martin Horauer, Ulrich Schmid, Klaus Schossmaier, Roland Höller, and Nikolaus Kerö. PSynUTC — evaluation of a high precision time synchronization prototype system for Ethernet LANs. In *Proceedings of the 34th IEEE Precise Time and Time Interval Systems and Application Meeting (PTTI'02)*, pages 263–278, Reston, Virginia, December 2002.

[HW05a] Jean-François Hermant and Josef Widder. Implementing reliable distributed real-time systems with the Θ-model. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *LNCS*, pages 334–350, Pisa, Italy, December 2005. Springer Verlag.

[HW05b] Martin Hutle and Josef Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Proceedings of the Seventh International Symposium on Self Stabilizing Systems (SSS 2005)*, volume 3764 of *LNCS*, pages 153–170, Barcelona, Spain, October 2005. Springer Verlag.

[KLSV03] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. *Proceedings 24th IEEE International Real-Time Systems Symposium (RTSS'03)*, 00:166–177, 2003.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–939, 1987.

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

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

[LL84b] Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[LLS03] Gérard Le Lann and Ulrich Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003. (Replaced by Research Report 28/2005, Institut für Technische Informatik, TU Wien, 2005.).

[LLW08] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Clock synchronization with bounded global and local skew. In *FOCS '08: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 509–518, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., San Francisco, USA, 1996.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Oct. 1988.

[Mav92] M. Mavronicolas. An upper and a lower bound for tick synchronization. In *Proceedings Real-Time Systems Symposium*, pages 246–255, Dec 1992.

[MFNT00] M. Mock, R. Frings, E. Nett, and S. Trikaliotis. Continuous clock synchronization in wireless real-time applications. *Reliable Distributed Systems, 2000. SRDS-2000. Proceedings The 19th IEEE Symposium on*, pages 125–132, 2000.

[MMG04] Steven Martin, Pascale Minet, and Laurent George. The trajectory approach for the end-to-end response times with non-preemptive fp/edf. In Walter Dosch, Roger Y. Lee, and Chisu Wu, editors, *SERA*, volume 3647 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2004.

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

[Mos09] Heinrich Moser. Towards a real-time distributed computing model. *Theoretical Computer Science*, 410(6–7):629–659, Feb 2009.

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

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

[MS08] Heinrich Moser and Ulrich Schmid. Optimal deterministic remote clock estimation in real-time systems. In *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS)*, pages 363–387, Luxor, Egypt, December 2008.

[MST99] Sue Moon, Paul Skelley, and Don Towsley. Estimation and removal of clock skew from network delay measurements. In *IEEE INFOCOM 1999*, March 1999.

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

[OPS99] Rafail Ostrovsky and Boaz Patt-Shamir. Optimal and efficient clock synchronization under drifting clocks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 3–12, New York, NY, USA, 1999. ACM.

[PGGGGH98] J. C. Palencia Gutiérrez, J. J. Gutiérrez García, and M. González Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. *Proc. of the 10th EuroMicro Conference on Real-Time Systems*, pages 35–44, 1998.

[PS92] Stephen Ponzio and Ray Strong. Semisynchrony and real time. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, pages 120–135, Haifa, Israel, November 1992.

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

[RS08] Peter Robinson and Ulrich Schmid. The Asynchronous Bounded-Cycle Model. In *Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'08)*, volume 5340 of *Lecture Notes in Computer Science*, pages 246–262, Detroit, USA, November 2008. Springer Verlag. (Best Paper Award).

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

[SBK05] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281 – 323, 2005.

[Sch86] Fred B. Schneider. A paradigm for reliable clock synchronization. In *Proceedings Advanced Seminar of Local Area Networks*, pages 85–104, Bandol, France, April 1986.

[Sch97] Ulrich Schmid, editor. *Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems*, J. Real-Time Systems 12(1–3), 1997.

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

[SKM$^+$00] Ulrich Schmid, Johann Klasek, Thomas Mandl, Herbert Nachtnebel, Gerhard R. Cadek, and Nikolaus Kerö. A Network Time Interface M-Module for distributing GPS-time over LANs. *Real-Time Systems*, 18(1):24–57, January 2000.

[SL96] Klaus Schossmaier and Dietmar Loy. An ASIC supporting external clock synchronization for distributed real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 277–282, L'Aquila, Italy, June 1996.

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

[SR87] Kang G. Shin and P. Ramanathan. Clock synchronization of a large multiprocessor system in the presence of malicious faults. *IEEE Trans. Comput.*, 36(1):2–12, 1987.

[SS97] Ulrich Schmid and Klaus Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, March 1997.

[SS03] Ulrich Schmid and Klaus Schossmaier. Interval-based clock synchronization with optimal precision. *Information and Computation*, 186(1):36–77, October 2003.

[SSHL97] Klaus Schossmaier, Ulrich Schmid, Martin Horauer, and Dietmar Loy. Specification and implementation of the Universal Time Coordinated Synchronization Unit (UTCSU). *Real-Time Systems*, 12(3):295–327, May 1997.

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

[TC94] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.

[VRC97] Paulo Veríssimo, Luís Rodrigues, and Antonio Casimiro. CesiumSpray: a precise and accurate global clock service for large-scale systems. *Real-Time Systems*, 12(3):243–294, 1997.

[WL88] Jennifer Lundelius Welch and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

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

[ZSSZ08] Minghu Zhang, Senzu Shen, Jian Shi, and Ting Zhang. Simple clock synchronization for distributed real-time systems. *Industrial Technology, 2008. ICIT 2008. IEEE International Conference on*, pages 1–5, April 2008.

Curriculum Vitae

Name:	Heinrich Moser
Date of Birth:	1980-03-04
Address:	Pülslgasse 22/4, 1230 Wien, Austria
Web:	`http://ti.tuwien.ac.at/ecs/people/moser`
Mail:	`moser@ecs.tuwien.ac.at`

Education History

04/2005 – 05/2009	PhD study, TU Wien, Institute of Computer Engineering (ECS group) Advisor: Ulrich Schmid
03/2003 – 03/2005	Study of Computer Science, TU Wien Master curriculum: Software Engineering & Internet Computing Graduated with distinction as Dipl.-Ing. Master thesis nominated for Distinguished Young Alumnus Award
10/1999 – 03/2003	Study of Computer Science, TU Wien Bachelor curriculum: Software & Information Engineering Graduated with distinction as Bakk.techn.
09/1997 – 06/1998	Secondary School, Austria. Graduated with distinction
09/1996 – 06/1997	High School, VA, USA Graduated with “Diploma” and grade average of “A”
09/1990 – 06/1996	Secondary School, Austria

Work Experience

since 04/2005	Research assistant, TU Wien, Institute of Computer Engineering
09/2004 – 01/2005	Teaching assistant, TU Wien, Institute of Computer Engineering
03/2002 – 01/2003	Teaching assistant, TU Wien, Institute of Information Systems
since 08/2005	Self-employed software developer
09/1996 – 06/1997	Teaching assistant, Huguenot Academy High School, USA
1995 – 2005	Software developer, moWARE Software, Mödling