Reconciling Distributed Computing Models and Real-Time Systems

Heinrich Moser and Ulrich Schmid

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

November 13, 2006

Abstract: *This paper[1] presents a simple real-time distributed computing model for message-passing systems, which reconciles the distributed computing and the real-time systems perspective: By just replacing instantaneous computing steps with computing steps of non-zero duration, we obtain a model that both facilitates real-time scheduling analysis and retains compatibility with classic distributed computing analysis techniques and results. So far, we have developed general simulations and validity conditions for transforming algorithms from the classic synchronous computing model (without clock drift) to our real-time model and vice versa, and have started investigating whether/which properties of real systems are inaccurately or even wrongly captured when resorting to zero step-time models. One example is the $\Omega(1)$ time complexity lower bound for optimal deterministic internal clock synchronization, which turned out to be $\Omega(n)$ in the real-time model.*

1 Motivation

Executions of distributed algorithms are typically modeled as sequences of atomic computing steps that are executed in zero time. With this assumption, it does not make a difference, for example, whether messages arrive at a processor simultaneously or nicely staggered in time: The messages are processed instantaneously when they arrive. The zero step-time abstraction is hence very convenient for analysis, and a wealth of distributed algorithms, impossibility results and lower bounds have been developed for models that employ this assumption [2].

In real systems, however, computing steps are neither instantaneous nor arbitrarily preemptable: A computing step triggered by a message arriving in the middle of the execution of some other computing step is usually delayed until the current computation is finished. This results in queuing phenomenons, which depend not only on the actual message arrival pattern but also on the queuing/scheduling discipline employed. The real-time systems community has established powerful techniques for analyzing such effects [5], such that the resulting worst-case response times and end-to-end delays can be computed.

This paper briefly surveys a real-time distributed computing model for message-passing systems introduced in [3], which reconciles the distributed computing and the real-time systems perspective: By just replacing the zero step-time assumption with non-zero step times, we obtain a real-time distributed computing model that admits real-time analysis without invalidating standard distributed computing analysis techniques and results.

Apart from making distributed algorithms amenable to real-time analysis, our model also allows to address the interesting question whether/which properties of real systems are inaccurately or even wrongly captured when resorting to classic zero step-time models. In [4], a basic version of our real-time model has already been employed: We revisited the well-studied problem of deterministic internal clock synchronization [6, 1] for this purpose and showed that, contrary to the classic computing model, no clock synchronization algorithm with constant running time can achieve optimal precision in our real-time model.

Our ongoing work is primarily devoted to distributed algorithms that involve some non-trivial real-time scheduling analysis under our model.

2 Classic Computing Model

In distributed computing research, system models are usually considered where the uncertainty comes from varying message delays, failures, and drifting clocks. Denoted "Partially Synchronous Reliable/Unreliable Models" in [6], such models are nowadays called (non-lockstep) synchronous models in literature. In order to solely investigate the effects of non-zero step-times, our real-time computing model will be based on the simple failure-free synchronous model introduced in [1]. Here it will be referred to as the *classic computing model*.

2.1 Classic System Model

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

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

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

Actions can actually be triggered by ordinary messages and timer messages: Ordinary messages are transmitted over the links. The *message delay* $\underline{\delta}$ is the difference between the real time of the action sending the message and the real time of the action receiving the message. There is

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

a lower bound $\underline{\delta}^-$ and an upper bound $\underline{\delta}^+$ on the message delay of every ordinary message.

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

2.2 Systems and Admissible Executions

An execution in the classic computing model is a sequence of actions. An action ac occurring at real-time t at processor p is a 5-tuple, consisting of the processor index $proc(ac) = p$, the received message $msg(ac)$, the occurrence real-time $time(ac) = t$, the hardware clock value $HC(ac) = HC_p(t)$ and the state transition sequence $trans(ac) = [oldstate, \ldots, newstate]$ (including messages to be sent). A valid execution must satisfy obvious properties such as conformance with the transition function, timer messages arriving on time, and reliable message transmission.

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

Let $\underline{s} = (n, [\underline{\delta}^-, \underline{\delta}^+])$ be a classic system. An execution is *$\underline{s}$-admissible*, if the execution comprises n processors and the message delay for each ordinary message stays within $[\underline{\delta}^-, \underline{\delta}^+]$. The actions in the execution must be ordered by their occurrence time; in the case of concurrent events, the ordering must capture causality.

3 Real-Time Computing Model

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

3.1 Real-Time System Model

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

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

- We must now distinguish two modes of a processor at any point in real-time t: *idle* and *busy*. Since computing steps cannot be interrupted, a *queue* is needed to store messages arriving while the processor is busy.
- When and in which order messages collected in the queue are processed is specified by some *scheduling policy*, which is, in general, independent of the algorithm. Formally, a scheduling policy is specified as an arbitrary mapping from the current queue state (= a sequence of messages), the hardware clock reading, and the current local processor state onto a single message from that message sequence. The scheduling policy is used to select a new message from the queue whenever processing of a job has been completed. To ensure liveness, we assume that the scheduling policy is *non-idling*.
- The delay of a message is measured from the real time of the *start of the job* sending the message to the arrival real time at the destination processor (where the message will be enqueued or, if the processor is idle, immediately causes the corresponding job to start). Analogous to the classic computing model, message delays of ordinary messages must be within a system-wide lower bound δ^- and an upper bound δ^+. Like the processing delay, the message delay and hence the bounds $[\delta^-, \delta^+]$ may depend on the number of messages sent in the sending job.
- We assume that the hardware clock can only be read at the beginning of a job. This restriction in conjunction with our definition of message delays allows us to define transition functions in exactly the same way as in the classic computing model. After all, the transition function just defines the "logical" semantics of a transition, but not its timing.
- Contrary to the classic computing model, the state transitions $oldstate \to \cdots \to newstate$ in a single computing step need not happen at the same time: Typically, they occur at different times during the job, allowing an intermediate state to be valid on a processor for some non-zero duration.

Figure 1 depicts an example of a single job at the sender processor p, which sends one message m to receiver q currently busy with processing another message. It shows the major timing-related parameters in the real-time computing model, namely, *message delay* (δ), *queuing delay* (ω), *end-to-end delay* ($\Delta = \delta + \omega$), and *processing delay* (μ) for the message m represented by the dotted arrows. The bounds on the message delay δ and the processing delay μ are part of the system model, although they need not be known to the algorithm. Bounds on the queuing delay ω and the end-to-end delay Δ, however, are *not* parameters of the system model—in sharp contrast to the classic computing model (see Section 2), where the end-to-end delay always equals the message delay. Rather, those bounds (if they exist) must be derived from the system parameters $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ and the message pattern of the algorithm, by performing a real-time scheduling analysis.

3.2 Real-time Runs

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

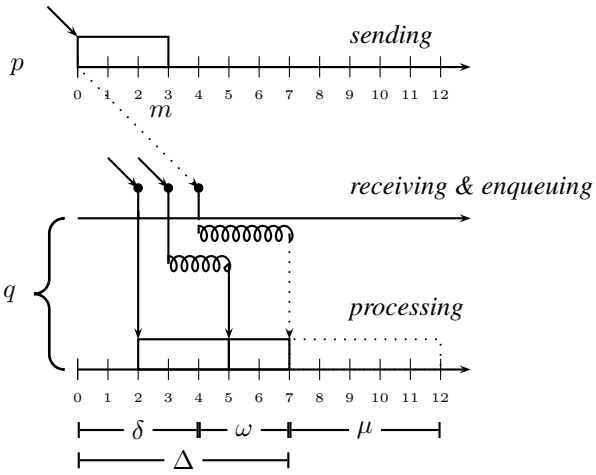

Figure 1: *Timing parameters for some message* m

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

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

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

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

3.3 Systems and Admissible Real-Time Runs

A real-time system s is defined by an integer n and two intervals $[\delta^-, \delta^+]$ and $[\mu^-, \mu^+]$. Considering δ^-, δ^+, μ^- and μ^+ to be constants would give an unfair advantage to broadcast-based algorithms when comparing algorithms' time complexity: Computation steps would take between μ^- and μ^+ time units, independently of the number of messages sent. This makes it impossible to derive a meaningful time complexity lower bound for systems in which a constant-time broadcast primitive is not available.

Therefore, the interval boundaries δ^-, δ^+, μ^- and μ^+ can be either constants or non-decreasing functions $\{0, \ldots, n-1\} \to \mathbb{R}^+$, representing a mapping from the number of destination processors to which ordinary messages are sent during that computing step to the actual message or processing delay bound.

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

Sending ℓ messages at once must not be more costly than sending those messages in multiple steps. In addition, we assume that the message delay uncertainty $\varepsilon_{(\ell)} := \delta^+_{(\ell)} - \delta^-_{(\ell)}$ is also non-decreasing in ℓ.

Definition 1. Let $s = (n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ be a real-time system. A rt-run is *s-admissible*, if the rt-run contains exactly n processors and satisfies the following timing properties: If ℓ is the number of processors to which messages are sent during some job J,

- The message delay (measured from $begin(J)$ to the corresponding receive event) of every message in $trans(J)$ must be within $[\delta^-_{(\ell)}, \delta^+_{(\ell)}]$.
- The job duration $d(J)$ must be within $[\mu^-_{(\ell)}, \mu^+_{(\ell)}]$.
- The ordering of receive events and jobs captures causality (cf. the well-known *happens-before* relation). For convenience, we also require that the rt-run, i.e., the sequence of receive events and jobs, is ordered by the occurrence time of receive events and the begin times of jobs.

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

- for every processor, choose an initial state from the set provided by the player and the hardware clock parameters (such as initial value or drift, depending on the hardware clock model used),
- for every message, choose a value within $[\delta^-, \delta^+]$ representing the sum of
  - the time between the start of the job which sends the message and the actual sending time of the message, and
  - the actual transmission delay of the message (until the receive event occurs),
- for every job, choose a value within $[\mu^-, \mu^+]$ for its processing time and associated overhead (e.g. scheduling),
- define the scheduling policy (but see next subsection).

3.4 Correctness and Impossibility

In the model presented so far, the scheduling policies are *adversary-controlled*, meaning that, in the game between player and adversary, first the player chooses the algorithm and afterwards the adversary can choose the scheduling policy which is most unsuitable for the algorithm. Thus, *correctness* proofs are *strong* (as the algorithm can defend itself against the most vicious scheduling policy), but *impossibility* proofs are *weak* (because the adversary has the scheduling policy on its side).

However, sometimes algorithms are designed for particular, a-priori-known scheduling policies, or the algorithm designer has the freedom to choose the scheduling policy which is most convenient for the algorithm. Thus, it is useful to define as *weak correctness* the correctness of an ⟨algorithm, scheduling policy⟩ pair and, analogously, as *strong impossibility* the absence of any such pair. In [3], we only consider strong correctness and strong impossibility.

4 Results Based on the Real-Time Computing Model

Transformations 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.

In [3], we presented two transformations for the case of drift-free clocks: One direction, simulating a real-time system $(n, [\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+], [\mu^- = \underline{\mu}, \mu^+ = \underline{\mu}])$ on top of a classic system $(n, [\underline{\delta}^-, \underline{\delta}^+])$, is quite straightforward: It suffices to implement an artificial processing delay $\underline{\mu}$, the queuing of messages arriving during such a simulated job, and the scheduling policy. This simulation allows to run any real-time computing model algorithm $\mathcal{A}$ designed for a system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$ with $\delta^- \leq \underline{\delta}^-$, $\delta^+ \geq \underline{\delta}^+$ and $\mu^- \leq \underline{\mu} \leq \mu^+$ on top of it, thereby resulting in a correct classic computing model algorithm.

The other direction, simulating a classic system $(n, [\underline{\delta}^- = \Delta^-, \underline{\delta}^+ = \Delta^+])$ on top of a real-time system $(n, [\delta^-, \delta^+], [\mu^-, \mu^+])$, is more tricky: First, the class of classic computing model algorithms $\underline{\mathcal{A}}$ that can be transformed into a real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ must be somewhat restricted. Second, and more importantly, a *real-time scheduling analysis* must be conducted in order to break the circular dependency of algorithm $\underline{\mathcal{A}}$ and end-to-end delays $\Delta \in [\Delta^-, \Delta^+]$ (and vice versa): On one hand, the classic computing model algorithm $\underline{\mathcal{A}}$, run atop of the simulation, might need to know the *simulated* message delay bounds $[\underline{\delta}^-, \underline{\delta}^+]$, which are just the end-to-end delay bounds $[\Delta^-, \Delta^+]$ of the underlying simulation. Those end-to-end delays, on the other hand, involve the queuing delay ω and are thus dependent on (the message pattern of) $\underline{\mathcal{A}}$ and hence on $[\underline{\delta}^-, \underline{\delta}^+]$. This circular dependency is "hidden" in the parameters of the classic computing model, but necessarily pops up when one tries to instantiate this model in a real system.

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

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

We do not want to embark on the intricacies of advanced real-time scheduling analysis techniques here, see [5] for an overview. For the purpose of simple problems like terminating clock synchronization (see below), quite trivial considerations are sufficient: A trivial end-to-end delay lower bound Δ^- is $\delta^-_{(1)}$. An upper bound Δ^+ can be obtained easily if, for example, there is an upper bound on the number of messages a processor receives in total.

Anyway, if eq. (1) provided by the real-time scheduling analysis can be solved for $[\Delta^-, \Delta^+]$, resulting in meaningful bounds $\Delta^- \leq \Delta^+$, they can be assigned to the algorithm parameters $[\underline{\delta}^-, \underline{\delta}^+]$. We will call such an assignment *feasible*. Any feasible assignment of $[\underline{\delta}^-, \underline{\delta}^+]$ results in a correct implementation of the real-time computing model algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$, since it ensures that both $\underline{\mathcal{A}}$ and the end-to-end delays are within their specifications. Our transformation in fact guarantees that any timer-free algorithm designed for some classic system $\underline{s}$ solving some suitable problem $\mathcal{P}$ can also solve that problem in the corresponding real-time system s if a feasible assignment for $[\underline{\delta}^-, \underline{\delta}^+]$ exists.

Clock Synchronization In [4], we revisited the well-known problem of optimal deterministic clock synchronization in the drift- and failure-free case [1]. We showed that the best precision achievable in the real-time computing model is $(1 - \frac{1}{n})\varepsilon_{(1)}$, which matches the well-known result in the classic computing model. It turned out, however, that the worst-case time complexity of every such optimal clock synchronization algorithm is at least $\Omega(n)$ in the real-time computing model. Since a $\mathrm{O}(1)$ (i.e., constant-time) algorithm achieving optimal precision is known for the classic computing model [1], this is an instance of a problem where the classic analysis gives too optimistic results. Note that we also established algorithms and lower bounds for sub-optimal clock synchronization in the real-time computing model. For example, we showed that clock synchronization to within a constant factor of the message delay uncertainty can be achieved in constant time only if a constant-time broadcast primitive is available.

5 Conclusion and Further Work

We presented a real-time computing model, which just adds non-zero computing step times to the classic computing model. Since it explicitly incorporates queuing effects, our model makes distributed algorithms amenable to real-time scheduling analysis, without, however, invalidating classic algorithms, analysis techniques, and impossibility/lower bound results. General transformations based on simulations between both models were established for this purpose, and a number of results regarding deterministic drift- and failure-free clock synchronization have been obtained.

Our ongoing research is devoted to applying our real-time computing model to clock synchronization in the case of drifting clocks and to establishing useful extensions of our model, such as allowing idling scheduling policies, preemption, and multiple processes per processor. Apart from this, we are investigating distributed algorithms for other problems in fault-tolerant distributed computing, which involve a non-trivial real-time scheduling analysis.

References

[1] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–240, 1984.

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

[3] H. Moser and U. Schmid. Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. Research Report 71/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2006. `http://www.vmars.tuwien.ac.at/php/pserver/extern/docdetail.php?DID=1973&viewmode=paper&year=2006`.

[4] —. 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. Springer Verlag, Bordeaux & Saint-Emilion, France, Dec 2006. (to appear, see [3] for an extended version).

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

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