Reconciling Fault-Tolerant Distributed Algorithms and Real-Time Computing
(Extended Abstract)

Heinrich Moser and Ulrich Schmid

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

Abstract. We present generic transformations, which allow to translate classic fault-tolerant distributed algorithms and their correctness proofs into a real-time distributed computing model (and vice versa). Owing to the non-zero-time, non-preemptible state transitions employed in our real-time model, scheduling and queuing effects (which are inherently abstracted away in classic zero step-time models, sometimes leading to overly optimistic time complexity results) can be accurately modeled. Our results thus make fault-tolerant distributed algorithms amenable to a sound real-time analysis, without sacrificing the wealth of algorithms and correctness proofs established in classic distributed computing research. By means of an example, we demonstrate that real-time algorithms generated by transforming classic algorithms can be competitive even w.r.t. optimal real-time algorithms, despite their comparatively simple real-time analysis.

1 Introduction

Executions of distributed algorithms are typically modeled as sequences of zero-time state transitions (steps) of a distributed state machine. The progress of time is solely reflected by the time intervals between steps. Owing to this assumption, it does not make a difference, for example, whether messages arrive at a processor simultaneously or nicely staggered in time: Conceptually, the messages are processed instantaneously in a step at the receiver when they arrive. The zero step-time abstraction is hence very convenient for analysis, and a wealth of distributed algorithms, correctness proofs, impossibility results and lower bounds have been developed for models that employ this assumption [3].

In real systems, however, computing steps are neither instantaneous nor arbitrarily preemptible: A computing step triggered by a message arriving in the middle of the execution of some other computing step is 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. Real-time systems research has established powerful techniques for analyzing those effects [11], such that worst-case response times and even end-to-end delays [12] can be computed.

Our *real-time model* for message-passing systems introduced in [8, 9] reconciles the distributed computing and the real-time systems perspective: By replacing zero-time steps by non-zero time steps, it allows to reason about queuing effects and puts scheduling in the proper perspective. In sharp contrast to the classic model, the end-to-end delay of a message is no longer a model parameter, but results from a real-time analysis based on job durations and communication delays.

In view of the wealth of distributed computing results, determining the properties which are preserved when moving from the classic zero step-time model to the real-time model is important: This transition should facilitate a real-time analysis *without* invalidating classic distributed computing analysis techniques and results. In [6, 5], we developed powerful general *transformations*: We showed that a system adhering to some particular instance of the real-time model can simulate a system that adheres to some instance of the classic model (and vice versa). All the transformations presented in [6] were based on the assumption of a *fault-free* system, however.

Contributions: In this paper, we generalize our transformations to the *fault-tolerant* setting: Processors are allowed to either *crash* or even behave *arbitrarily* (Byzantine) [2]. We define (mild) conditions on problems, algorithms and system parameters, which allow to re-use classic fault-tolerant distributed algorithms in the real-time model, and to employ classic correctness proof techniques for fault-tolerant distributed algorithms designed for the real-time model. As our transformations are generic, i.e., work for any algorithm adhering to our conditions, proving their correctness has already been a non-trivial exercise in the fault-free case [6], and became definitely worse in the presence of failures. We apply our transformation to the well-known problem of Byzantine agreement and analyze the timing properties of the resulting real-time algorithm.

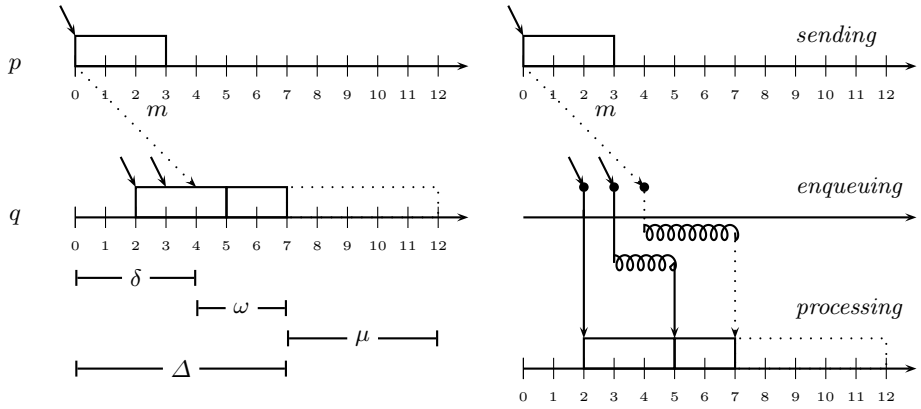
The full paper version of this extended abstract—including all proofs and more detailed explanations—can be found in an accompanying research report [10].

Related Work: We are not aware of much existing work that is similar in spirit to our approach. In order to avoid repeating the overview of related work already presented in [8] and [6], we relegated it to the research report as well.

2 System Models

A *distributed system* consists of a set of *processors* and some means for communication. In this paper, we will assume that a *processor* is a *state machine* running some kind of *algorithm* and that communication is performed via message-passing over *point-to-point links* between pairs of processors.

The algorithm specifies the state transitions the processor may carry out. In distributed algorithms research, the common assumption is that state transitions are performed in zero time. Thus, transmission delay bounds typically represent *end-to-end delay bounds*: All kinds of delays are abstracted away in one system parameter.



(a) Timing parameters for some msg. m (b) Enqueuing shown explicitly

Fig. 1. Real-time model

The transformations introduced in this paper will relate two different distributed computing models. In both models, processors are equipped with hardware clocks and connected via reliable links.

Classic model: In what we call the classic synchronous model, processors execute zero-time steps (called *actions*) and the only model parameters are lower and upper bounds on the end-to-end delays $[\underline{\delta}^-, \underline{\delta}^+]$.

Real-time model: In this model, the zero-time assumption is dropped, i.e., the end-to-end delay bounds are split into bounds on the transmission time of a message (which we will call *message delay*) $[\delta^-, \delta^+]$ and on the actual processing time $[\mu^-, \mu^+]$. In contrast to the *actions* of the classic model, we call the non-zero-time computing steps in the real-time model *jobs*.

Figure 1 shows the major timing-related parameters, namely, *message delay* (δ, measured from the beginning of the sending job), *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 (but 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 model. Rather, those bounds (if they exist) must be derived from the system parameters $[\delta^-, \delta^+]$, $[\mu^-, \mu^+]$ and the message pattern of the algorithm, by performing a real-time analysis. These system parameters may depend on the number of messages sent in the sending job: For example, $\delta^+_{(3)}$ is the upper bound on the message delay of messages sent by a job sending three messages in total.

Running an algorithm will result in an *execution* in a classic system and a *real-time run (rt-run)* in a real-time system. An execution is a sequence of zero-time actions (which encapsulate both message reception and processing), whereas an rt-run is a sequence of receive events, jobs, and drop events. On non-faulty

processors, every incoming message causes a receive event and either a job—if the message is accepted by the (non-idling, non-preemptive) scheduling/admission policy—or a drop event.

2.1 Failures and Admissibility

A failure model indicates whether a given execution or rt-run is *admissible* w.r.t. a given system running a given algorithm. In this work, we restrict our attention to the f-f'-ρ failure model, which is a hybrid failure model ([4, 13, 1]) that incorporates both crash and Byzantine faulty processors. Of the n processors in the system,

- at most $f \geq 0$ may *crash* and
- at most $f' \geq 0$ may be *arbitrarily faulty* ("Byzantine").

All other processors are called *correct*.

A given execution (resp. rt-run) conforms to the f-f'-ρ failure model, if all message delays are within $[\underline{\delta}^-, \underline{\delta}^+]$ (resp. $[\delta^-, \delta^+]$) and the following conditions hold:

- All *timers* set by a processor trigger an action (resp. a receive event) at their designated hardware clock time. For notational convenience, expiring timers are modeled as incoming *timer messages*.
- On all non-Byzantine processors, clocks drift by at most ρ.
- All correct processors make state transitions as specified by the algorithm. In the real-time model, they obey the scheduling/admission policy, and all of their jobs take between μ^- and μ^+ time units.
- A crashing processor behaves like a correct one until it crashes. In the classic model, all actions *after the crash* do not change the state and do not send any messages. In the real-time model, after a processor has crashed, all messages in its queue are dropped, and every new message arriving will be dropped immediately rather than being processed. *Unclean crashes* are allowed: the last action/job on a processor might execute only a *prefix* of its state transition sequence.

In the analysis and the transformation proofs, we will examine given executions and rt-runs. Therefore, we know which processors behaved correct, crashing or Byzantine faulty. Note, however, that this information is only available during analysis; the algorithms themselves, including the simulation algorithms presented in the following sections, do not know which of the other processors are faulty. The same holds for timing information: While, during analysis, we can say that an event occurred at some exact real time t, the only information available to the algorithm is the local hardware clock reading at the beginning of the action or the job.

2.2 State Transition Traces

The *global state* of a system is composed of the current real-time t and the local state of every processor s_p. Rt-runs do not allow a well-defined notion of global states, since they do not fix the exact time of state transitions in a job. Thus, we use the "microscopic view" of *state-transition traces* (st-traces) to assign real-times to all atomic state transitions.

Example 1. Let J be a job in a real-time run ru, which starts in state *oldstate*, sends some message, then switches to some state s and finally to state *newstate*. If tr is an st-trace of ru, it contains the following *state transition events* (st-events) ev' and ev'':

- $ev' = (transition : t', p, oldstate, s)$
- $ev'' = (transition : t'', p, s, newstate)$

$t' \leq t''$, and both must be between the start and the end time of J.

In addition, every input message m arriving at some time t^* is represented by an st-event $(input : t^*, m)$. *Input messages* are messages from outside the system that can be used, for example, to start an algorithm.

Clearly, there are *multiple* possible st-traces for a single rt-run. Executions in the classic model have corresponding st-traces as well, with the st-events having the same time as the corresponding action.

A *problem* $\mathcal{P}$ is defined as a set of (or a predicate on) st-traces. An execution or an rt-run *satisfies* a problem if $tr \in \mathcal{P}$ holds for all its st-traces. If all st-traces of all admissible rt-runs (or executions) of some algorithm in some system satisfy $\mathcal{P}$, we say that this algorithm *solves* $\mathcal{P}$ in the given system.

3 Transformation Real-Time to Classic Model

As the real-time model is a generalization of the classic model, the set of systems covered by the classic model is a strict subset of the systems covered by the real-time model. More precisely, every system in the classic model can be specified in terms of the real-time model with $[\delta^- = \underline{\delta}^-, \delta^+ = \underline{\delta}^+]$ and $[\mu^- = 0, \mu^+ = 0]$. Intuition tells us that impossibility results also hold for the general case, i.e., that an impossibility result for some classic system holds for all real-time systems with $[\delta^- \leq \underline{\delta}^-, \delta^+ \geq \underline{\delta}^+]$ and arbitrary $[\mu^-, \mu^+]$, because the additional delays do not provide the algorithm with any useful information.

As it turns out, this conjecture is true: There is a simulation that allows to use an algorithm designed for the real-time model in the classic model—and, thus, to transfer impossibility results from the classic to the real-time model—provided the following conditions hold:

Cond1 *Problems must be simulation-invariant.* [6] Informally speaking, the problem only cares about a subset of the processors' state variables and their hardware clock values.

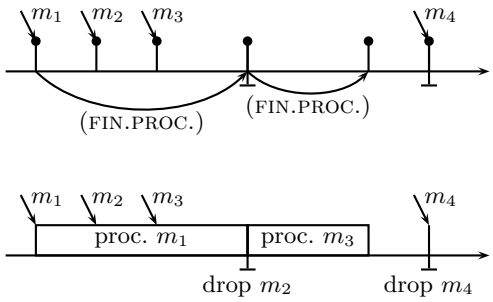


Fig. 2. Example: Execution ex of $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ and corresponding rt-run ru of $\mathcal{A}$

Cond2 *The delay bounds in the classic system must be at least as restrictive as those in the real-time system.* As long as $\delta^-_{(\ell)} \leq \underline{\delta}^-$ and $\delta^+_{(\ell)} \geq \underline{\delta}^+$ holds (for all ℓ), any message delay of the simulating execution ($\underline{\delta} \in [\underline{\delta}^-, \underline{\delta}^+]$) can be directly mapped to a message delay in the simulated rt-run ($\delta = \underline{\delta}$), such that $\delta \in [\delta^-_{(\ell)}, \delta^+_{(\ell)}]$ is satisfied. Thus, a simulated message corresponds directly to a simulation message with the same message delay. [10] shows how this requirement can be weakened to $\delta^-_{(1)} \leq \underline{\delta}^- \wedge \delta^+_{(1)} \geq \underline{\delta}^+$.

Cond3 *Hardware clock drift must be reasonably low.* Assume a system with very inaccurate hardware clocks, combined with very accurate processing delays: 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 ρ is very high and $\mu^+ - \mu^-$ is very low, the precision of this simple "clock" might be better than the one of the hardware clock. Thus, algorithms might in fact benefit from the processing delay, as opposed to the zero step-time situation. To avoid such effects, the hardware clock must be "accurate enough" to define (time-out) a time span which is guaranteed to lie within μ^- and μ^+.

The complete proof is contained in [10]. Note that it is technically very different from the proof in the fault-free setting [8, 6], since (Byzantine) failures may not only affect the simulated algorithm, but also the simulation algorithm.

Roughly, the proof proceeds as follows: Let $\mathcal{A}$ be a real-time algorithm that solves some problem $\mathcal{P}$ in some real-time system s under failure model f-f'-ρ.

Simulation: We run algorithm $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ in a classic system $\underline{s}$. $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ consists of algorithm $\mathcal{A}$ on top of an algorithm that simulates a real-time system with scheduling (according to some scheduling policy pol) and queueing effects. It simulates jobs with a duration between μ^- and μ^+ by setting a timer—denoted (FIN.PROC.), i.e., "finished processing" in Fig. 2—and enqueuing all messages that arrive before the timer has expired.

Transformation: As a next step, we transform every execution ex of $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ into a corresponding rt-run ru of $\mathcal{A}$ (see Figure 2): The jobs simulated by (FIN.PROC.) messages are mapped to "real" jobs in the rt-run. Note that, on Byzantine processors, every action in the execution can simply be mapped to a corresponding receive event and a zero-time job, since jobs on Byzantine nodes

do not need to obey any timing restrictions. It can be shown that ru is an admissible rt-run of $\mathcal{A}$ conforming to failure model f-f'-ρ.

State transition argument: Since (a) ru is an admissible rt-run of algorithm $\mathcal{A}$ in s, and (b) $\mathcal{A}$ is an algorithm solving $\mathcal{P}$ in s, it follows that ru *satisfies* $\mathcal{P}$. Choose any st-trace tr^{ru} of ru where all state transitions are performed at the beginning of the job. Since ru satisfies $\mathcal{P}$, $tr^{ru} \in \mathcal{P}$. The transformation ensures that exactly the same state transitions are performed in ex and ru (omitting the variables required by the simulation algorithm). Since (i) $\mathcal{P}$ is a simulation-invariant problem, (ii) $tr^{ru} \in \mathcal{P}$, and (iii) every st-trace tr^{ex} of ex performs the same state transitions on algorithm variables as some tr^{ru} of ru at the same time, it follows that $tr^{ex} \in \mathcal{P}$ and, thus, ex *satisfies* $\mathcal{P}$.

By applying this argument to every admissible execution ex of $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ in $\underline{s}$, we see that every such execution satisfies $\mathcal{P}$. Thus, $\underline{\mathcal{S}}_{\mathcal{A},pol,\mu}$ solves $\mathcal{P}$ in $\underline{s}$ under failure model f-f'-ρ.

4 Transformation Classic to Real-Time Model

When running a real-time model algorithm in a classic system, as shown in the previous section, 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.

Unfortunately, this is not the case for transformations in the other direction, i.e., running a classic model algorithm in a real-time system: 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 the beginning and the end of the job. Thus, there could be algorithms that solve some problem in the classic model, but fail to do so in the real-time model.

Fortunately, however, it is possible to show that if some algorithm solves some 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, where the end-to-end delay bounds Δ^- and Δ^+ of the real-time system equal the message delay bounds $\underline{\delta}^-$ and $\underline{\delta}^+$ of the simulated classic system. For the fault-free case, this has been shown in [6].

Definition 2. *[6] Let tr be an st-trace. A* μ^+-shuffle *of tr is constructed by moving transition st-events in tr at most $\mu^+_{(\ell)}$ time units into the future without violating causality. Every st-event may be shifted by a* different *value v, $0 \leq v \leq \mu^+_{(\ell)}$. In addition, input st-events may be moved arbitrarily far into the past.*

$\mathcal{P}^*_{\mu^+}$ is the set of all μ^+-shuffles of all st-traces of $\mathcal{P}$.

A typical example for μ^+-shuffles is the *Mutual Exclusion* problem: If $\mathcal{P}$ is *5-second gap mutual exclusion* (no processor may enter the critical section for

5 seconds after the last processor left it), then $\mathcal{P}^*_{\mu^+}$ with $\mu^+ = 2$ is *3-second gap mutual exclusion*. If $\mathcal{P}$ is *causal mutual exclusion* (there is a causal chain between consecutive *exit* and *enter* operations), then $\mathcal{P} = \mathcal{P}^*_{\mu^+}$. [6]

Formally, the following conditions must be satisfied for the transformation to work in the fault-tolerant case:

Cond1 *There is a feasible end-to-end delay assignment* $[\Delta^-, \Delta^+] = [\underline{\delta}^-, \underline{\delta}^+]$. Finding such an assignment might require a (non-trivial) real-time analysis to break the circular dependency: On the one hand, the (classic) algorithm might need to know $\underline{\delta}^-$ and $\underline{\delta}^+$. On the other hand, the (real-time) end-to-end delay bounds Δ^- and Δ^+ involve the queuing delay and are thus dependent on the message pattern of the algorithm and, hence, on $\underline{\delta}^-$ and $\underline{\delta}^+$.

Cond2 *The scheduling/admission policy (a) only drops irrelevant messages and (b) schedules input messages in FIFO order.* "Irrelevant messages" that do not cause an algorithmic state transition could be, for example, messages that obviously originate from a faulty sender or, in round-based algorithms, late messages from previous rounds.

Cond3 *The algorithm tolerates late timer messages*, and *the scheduling policy ensures that timer messages get processed soon after being received.* In the classic model, a timer message scheduled for hardware clock time T gets processed at time T. In the real-time model, on the other hand, the message *arrives* when the hardware clock reads T, but it might get queued if the processor is busy. Still, an algorithm designed for the classic model might depend on the message being processed exactly at hardware clock time T. Thus, either (a) the algorithm must be tolerant to timers being processed later than their designated arrival time or (b) the scheduling policy must ensure that timer messages do not experience queuing delays—which might not be possible, since we assume a non-idling and non-preemptive scheduler.

Combining both options yields the following condition: The algorithm tolerates timer messages being processed up to α real-time units after the hardware clock read T, and the scheduling policy ensures that no timer message experiences a queuing delay of more than α. Options (a) and (b) outlined above correspond to the extreme cases of $\alpha = \infty$ and $\alpha = 0$.

These requirements can be encoded in failure models: f-f'-ρ+latetimers$_\alpha$, a failure model on executions in the classic model, is weaker than f-f'-ρ (i.e., $ex \in f\text{-}f'\text{-}\rho \Rightarrow ex \in f\text{-}f'\text{-}\rho$+latetimers$_\alpha$), since timer messages may arrive late by at most α seconds in the former. On the other hand, f-f'-ρ+precisetimers$_\alpha$, a failure model on rt-runs in the real-time model that restricts timer message queuing by the scheduler to at most α seconds, is stronger than the unconstrained f-f'-ρ (i.e., $ru \in f\text{-}f'\text{-}\rho$+precisetimers$_\alpha \Rightarrow ru \in f\text{-}f'\text{-}\rho$).

Again, the complete proof is contained in [10], and it follows the same line of reasoning as the one in the previous section. This time, no simulation is required ($\mathcal{S}_{\underline{\mathcal{A}}} := \underline{\mathcal{A}}$), and the transformation is straight-forward by mapping each job to an action occurring at the begin time of the job.

5 Example: The Byzantine Generals

We consider the *Byzantine Generals* problem [2]: A commanding general must send an order to his $n-1$ lieutenant generals such that

IC1 All loyal lieutenants obey the same order.
IC2 If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.

In the context of computer science, *generals* are processors, *orders* are binary values and *loyal* means fault-free. It is well-known that f Byzantine faulty processors can be tolerated if $n > 3f$. The challenge lies in the fact that a faulty processor might send out asymmetric information: The commander might send value 0 to the first lieutenant, value 1 to the second lieutenant and no message to the remaining lieutenants. Thus, the lieutenants (some of which might be faulty as well) need to exchange information afterwards to ensure that IC1 is satisfied.

[2] presents an "oral messages" algorithm, which we will call $\underline{A}$: Initially (round 0), the value from the commanding general is broadcast. Afterwards, every round basically consists of broadcasting all information received in the previous round. After round f, the non-faulty processors have enough information to make a decision that satisfies IC1 and IC2.

What makes this algorithm interesting in the context of this paper is the fact that (a) it is a synchronous round-based algorithm and (b) the number of messages exchanged during each round increases exponentially: After receiving v from the commander in round 0, each lieutenant p sends "$p : v$" to all other lieutenants in round 1. In round 2, it relays the messages received in the previous round, e.g., processor q would send "$q : p : v$", meaning: "processor q says: (processor p said: (the commander said: v))", to all processors except p, q and the commander. More generally, in round $r \geq 2$, every processor multicasts $\#_S = (n-2)\cdots(n-r)$ messages, each sent to $n-r-1$ recipients, and receives $\#_R = (n-2)\cdots(n-r-1)$ messages.

Implementing synchronous rounds in the classic model is straightforward when the clock skew is bounded; for simplicity, we will hence assume that the hardware clocks are perfectly synchronized. At the beginning of a round (at some hardware clock time t), all processors perform some computation, send their messages and set a timer for time $t + \underline{\delta}^+$, after which all messages for the current round have been received and processed and the next round can start.

We model these rounds as follows: The round start is triggered by a timer message. The triggered action, labeled as C, (a) sets a timer for the next round start and (b) initiates the broadcasts (using a timer message that expires immediately). The broadcasts are modeled as $\#_S$ actions on each processor (labeled as S), connected by timer messages that expire immediately. Likewise, the $\#_R$ actions receiving messages are labeled R.

To make this algorithm work in the real-time model, one would need to determine the longest possible round duration W, i.e., the maximum time required for any one processor to execute all its C, S and R jobs, and replace the delay of the "start next round" timer from $\underline{\delta}^+$ to this value.

Let us take a step back and examine the problem from a strictly formal point of view: Given algorithm $\underline{\mathcal{A}}$, we will try to satisfy Cond1, Cond2 and Cond3, so that the transformation of Sect. 4 can be applied.

For this example, let us restrict our failure model to a set of f processors that produce only benign message patterns, i.e., a faulty processor may crash or modify the message contents arbitrarily, but it must not send additional messages or send the messages at a different time (than a fault-free or crashing processor would). We will denote this restricted failure model as f^* and claim (proof omitted) that the result established in Sect. 4 also holds for this model, i.e., that a classic algorithm conforming to model f^*+latetimers$_\alpha$ can be transformed to a real-time algorithm in model f^*+precisetimers$_\alpha$.

Let us postpone the problem of determining a feasible assignment (Cond1) until later. Cond2 can be satisfied easily by choosing a suitable scheduling/admission policy. Cond3 deals with timer messages, and this needs some care: Timer messages must arrive "on time" *or* the algorithm must be able to cope with late timer messages *or* a little bit of both (which is what factor α in Cond3 is about). In $\underline{\mathcal{A}}$, we have two different types of timer messages: (a) the timer messages initiating the send actions and (b) those starting a new round.

How can we ensure that $\underline{\mathcal{A}}$ still works under failure model f^*+latetimers$_\alpha$ (in the classic model)? If the timers for the S jobs each arrive α time units later, the last send action occurs $\#_S \cdot \alpha$ time units *after* the start of the round instead of *immediately* at the start of the round. Likewise, if the timer for the round start occurs α time units later, everything is shifted by α. To take this shift into account, we just have to set the round timer to $\underline{\delta}^+ + (\#_S + 1)\alpha$.

As soon as we have a feasible assignment, the transformation of Sect. 4 will guarantee that $\mathcal{S}_{\underline{\mathcal{A}}}$ solves $\mathcal{P} = \mathcal{P}^*_{\mu^+} = \text{IC1} + \text{IC2}$ under failure model f^*+precisetimers$_\alpha$. For the time being, we choose $\alpha = \mu^+_{(n-1)}$, so the round timer in $\mathcal{S}_{\underline{\mathcal{A}}}$ waits for $\Delta^+ + (\#_S + 1)\mu^+_{(n-1)}$ time units. This is a reasonable choice: Since the S jobs are chained by timer messages expiring immediately, these timer messages are delayed at least by the duration of the job setting the timer. We will later see that $\mu^+_{(n-1)}$ suffices.

Returning to Cond1, the problem of determining Δ^+ can be solved by a very conservative estimate: Choose $\Delta^+ = \delta^+_{(n-1)} + \mu^+_{(0)} + \#^f_S \cdot \mu^+_{(n-1)} + (\#^f_R - 1)\mu^+_{(0)}$, with $\#^f_S$ and $\#^f_R$ denoting the maximum number of send and receive jobs (= the number of such jobs in the last round f); this is the worst-case time required for one message transmission, one C, all S and all R except for one (= the one processing the message itself). Clearly, the end-to-end delay of one round r message—consisting of transmission plus queuing but not processing—cannot exceed this value if the algorithm executes in a lock-step fashion and no rounds overlap. This is ensured by the following lemma: *For all rounds r, the following holds: (a) The round timer messages on all processors start processing simultaneously. (b) As soon as the round timer messages starting round r arrive, all messages from round $r-1$ have been processed.* Since, for our choice of Δ^+, the round timer waits for $\delta^+_{(n-1)} + (\#^f_S + \#_S + 1)\mu^+_{(n-1)} + \#^f_R \cdot \mu^+_{(0)}$ time units, it

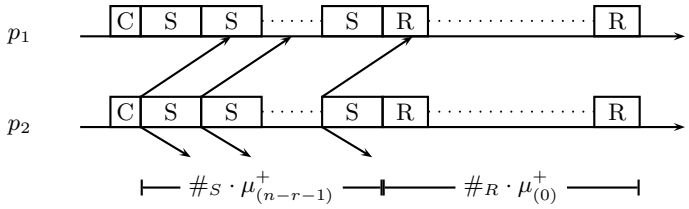


Fig. 3. Example rt-run of a Byzantine Generals round

is plain to see that this is more than enough time to send, transmit and process all pending round r messages by choosing a scheduling policy that favors C jobs before S jobs before R jobs. Formally, this can be shown by a simple induction on r. Considering this scheduling policy and this lemma, it becomes apparent that $\alpha = \mu^+_{(n-1)}$ was indeed sufficient (see above): A timer for an S job is only delayed until the current C or S job has finished.

Thus, we end up with an algorithm $\mathcal{S}_{\underline{\mathcal{A}}}$ satisfying IC1 and IC2, with synchronous round starts and a round duration of $\delta^+_{(n-1)} + (\#^f_S + \#_S + 1)\mu^+_{(n-1)} + \#^f_R \cdot \mu^+_{(0)}$.

Competitive Factor: Since the transformation is generic and does not exploit the round structure, the round duration is considerably larger than necessary: Our transformation requires *one* fixed "feasible assignment" for Δ^+; thus, we had to choose $\#^f_S$ and $\#^f_R$ instead of $\#_S$ and $\#_R$, which are much smaller for early rounds.

Since the rounds are disjoint—no messages cross the "round barrier"—and $\underline{\delta}^+/\Delta^+$ are only required for determining the round duration, the transformation results still hold if α and Δ^+ are fixed *per round*. This allows us to choose $\alpha = \mu^+_{(n-r-1)}$ and $\Delta^+ = \delta^+_{(n-r-1)} + \mu^+_{(0)} + \#_S \cdot \mu^+_{(n-r-1)} + (\#_R - 1)\mu^+_{(0)}$, resulting in a round duration of

$$W^{est} = \mu^+_{(0)} + (2\#_S + 1)\mu^+_{(n-r-1)} + \delta^+_{(n-r-1)} + (\#_R - 1)\mu^+_{(0)}.$$

Even though the round durations are quite large—they increase exponentially with the round number—it turns out that this bound obtained by our model transformation is only a constant factor away from the optimal solution, i.e., from the round duration W^{opt} determined by a precise real-time analysis [7]: $W^{est} \leq 4W^{opt}$ [10]. In conjunction with the fact that the transformed algorithm is much easier to get and to analyze, this reveals that our generic transformations are indeed a powerful tool for obtaining real-time algorithms.

6 Conclusions

We introduced a real-time model for message-passing distributed systems with processors that may crash or even behave Byzantine, and established simulations

that allow to run an algorithm designed for the classic zero-step-time model in some instance of the real-time model (and vice versa). Precise conditions that guarantee the correctness of these transformations are also given. The real-time model thus indeed reconciles fault-tolerant distributed and real-time computing, by facilitating real-time analysis without sacrificing classic distributed computing knowledge. In particular, our transformations allow to re-use existing classic fault-tolerant distributed algorithms and proof techniques in the real-time model, resulting in solutions which are competitive w.r.t. optimal real-time algorithms.

References

1. Biely, M., Schmid, U., Weiss, B.: Synchronous consensus under hybrid process and link failures. Theoretical Computer Science In Press, Corrected Proof, – (2010)
2. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems 4(3), 382–401 (July 1982)
3. Lynch, N.: Distributed Algorithms. Morgan Kaufman Publishers, Inc., San Francisco, USA (1996)
4. Meyer, F.J., Pradhan, D.K.: Consensus with dual failure modes. In: In Digest of Papers of the 17th International Symposium on Fault-Tolerant Computing. pp. 48–54. Pittsburgh (Jul 1987)
5. Moser, H.: A Model for Distributed Computing in Real-Time Systems. Ph.D. thesis, Technische Universität Wien, Fakultät für Informatik (May 2009)
6. Moser, H.: Towards a real-time distributed computing model. Theoretical Computer Science 410(6–7), 629–659 (Feb 2009)
7. Moser, H.: The byzantine generals' round duration. Research Report 9/2010, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria (2010)
8. Moser, H., Schmid, U.: Optimal clock synchronization revisited: Upper and lower bounds in real-time systems. In: Proceedings of the International Conference on Principles of Distributed Systems (OPODIS). pp. 95–109. LNCS 4305, Springer Verlag, Bordeaux & Saint-Emilion, France (Dec 2006)
9. Moser, H., Schmid, U.: Reconciling distributed computing models and real-time systems. In: Proceedings Work in Progress Session of the 27th IEEE Real-Time Systems Symposium (RTSS'06). pp. 73–76. Rio de Janeiro, Brazil (Dec 2006)
10. Moser, H., Schmid, U.: Reconciling fault-tolerant distributed algorithms and real-time computing. Research Report 11/2010, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (2010), http://www.vmars.tuwien.ac.at/documents/extern/2770/RR.pdf
11. Sha, L., Abdelzaher, T., Arzen, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Systems Journal 28(2/3), 101–155 (2004)
12. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocess. Microprogram. 40(2-3), 117–134 (1994)
13. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid process and link failures. Distributed Computing 20(2), 115–140 (Aug 2007), http://www.vmars.tuwien.ac.at/documents/extern/2282/journal.pdf