Network Synchronization in the Crash-Recovery Model

Felix C. Freiling[1], Sven Henkel[2], and Josef Widder[3]

[1] Department of Computer Science, University of Mannheim, freiling@uni-mannheim.de
[2] Student, RWTH Aachen University, sven.henkel@rwth-aachen.de
[3] Embedded Computing Systems Group, Technical University of Vienna, widder@ecs.tuwien.ac.at

University of Mannheim
Department for Mathematics and Computer Science
Technical Report TR 2006-09

Technical University of Vienna
Embedded Computing Systems Group
Research Report 49/2006

May 15, 2006

Abstract. This work investigates the amount of information about failures required to simulate a synchronous distributed system by an asynchronous distributed system prone to crash-recovery failures. A *failure detection sequencer* Σ_{CR} for the crash-recovery failure model is defined, which outputs information about crashes and recoveries *and* about the state of the crashed or recovered processes. Using the simulation technique of a *synchronizer*, it is shown that in general it is impossible to implement a synchronizer in an asynchronous distributed system with an arbitrary number of concurrent crash-recovery faults. It is shown that a synchronizer is implementable given Σ_{CR} and an asynchronous distributed system with at least one correct process. Furthermore, it is proven that Σ_{CR} can be emulated in a synchronous distributed system and hence can be regarded as the weakest failure detection device suitable to implement a synchronizer in the crash-recovery failure model.

1 Introduction

In a *synchronous* distributed message-passing system, processes are tightly coupled: The computation proceeds in *rounds* meaning that all processes execute their local algorithm at the same speed. The local executions in such a system are triggered by a global pulse and thus performed concurrently on all processes. All messages sent in such a system are guaranteed to be delivered within the same round, i.e., before the next pulse happens. In contrast to synchronous systems, an *asynchronous* system does not provide any guarantees about processing speed differences or message delivery delays. Consequently, algorithms designed for synchronous systems do not necessarily work in an asynchronous system whereas asynchronous algorithms trivially work in synchronous systems.

It is well known that, in general, algorithms solving a specific problem in an asynchronous distributed system have a higher algorithmic complexity than algorithms solving the same problem in a synchronous one. Intuitively, this is because the asynchronous algorithm needs to put some effort into the synchronization of the processes. In order to reduce the algorithmic complexity of an asynchronous algorithm, it was proposed by Awerbuch [Awe85] to extract this synchronization task out of the algorithm and put it into a new module, called a *synchronizer*. Using a synchronizer module, it is possible to use synchronous algorithms in asynchronous systems (see Figure 1).

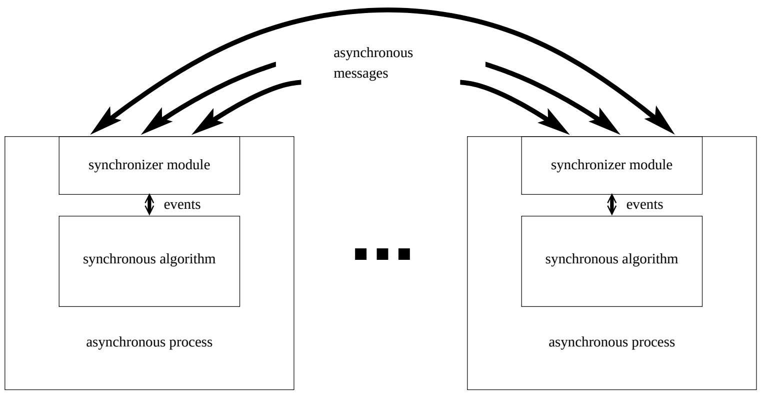

Fig. 1. Synchronizer concept

Related work. In a seminal paper, Awerbuch [Awe85] showed that a synchronizer is implementable in a fault-free environment. This implies that synchronous and asynchronous systems are equivalent regarding the solvability of distributed computing problems in this case. Faults, even simple ones like crash faults, make some difference, as is manifested by the famous impossibility result of Fischer, Lynch, and Paterson [FLP85] on fault tolerant consensus.

How much difference faults and their detectability make was explored by Chandra and Toueg [CT96] who proved that consensus is in fact solvable in asynchronous systems, provided that information about faults is eventually present.

Still, it is not perfectly understood how information on faults and the task of network synchronization are correlated. On the one hand, the results by Chandra and Toueg [CT96] suggest that this correlation is strong. On the other hand, the results by Charron-Bost, Guerraoui, and Schiper [CBGS00] showed that synchronous systems and the perfect failure detector are not equivalent.

Gärtner and Pleisch [GP02] later showed that in the crash-stop failure model a *failure detection sequencer* is sufficient and necessary to implement a synchronizer. Their failure detection sequencer Σ is mainly based on the perfect failure detector $\mathcal{P}$ — presented by Chandra and Toueg [CT96]. In contrast to $\mathcal{P}$, Σ not only outputs information about the crashes in the system, but also about the state of the crashed processes. It was shown that Σ is implementable in synchronous systems.

Setting. In this paper we consider the task of network synchronization under a more realistic fault model by investigating whether such an equivalence can be found in the crash-recovery failure model. Thus, we assume that processes that crash may recover later and resume participating in the distributed computation. Our setting is that processes recover from scratch, i.e., we assume the *absence of stable storage.* Under these assumptions we explore what kinds of properties a failure detection sequencer is required to have in order to be equivalent to the synchronous (lockstep) crash-recovery system.

Contributions. In this paper we provide definitions of the necessary and sufficient abstractions to implement a synchronizer in the crash-recovery model of failures.

- We define an appropriate failure detection sequencer for the crash-recovery model and show that it is sufficient to implement a synchronizer in this model as long as one process remains up all the time. Intuitively, such a sequencer accurately indicates crashes and recoveries of processes together with all messages sent by the crashing process since its most recent recovery. Hence, such a sequencer is a strict generalization of the sequencer of Gärtner and Pleisch [GP02].
- We show that the assumption of one "always-up" process is necessary, i.e., it is impossible to implement a synchronizer in the crash-recovery model if it is possible that all processes are down simultaneously — even with failure detectors or failure detector sequencers.
- Our results are based on an event-based definition of what a synchronous system is, which we call *lockstep synchrony.* We show that given such a lockstep synchronous system we can implement the failure detection sequencer for the crash-recovery model. Hence, our sequencer abstraction can be regarded as necessary to allow network synchronization in the crash-recovery model.

Roadmap. The paper is structured as follows: After a presentation of our system model in section 2, we define lockstep synchrony and synchronizers in section 3. Section 4 introduces the failure detection sequencer for the crash-recovery failure model, followed by the named impossibility result in section 5. In section 6 we present our synchronizer algorithm for systems with one correct process and we show that the used sequencer is the weakest failure detection device allowing such an implementation. Section 7 concludes this work.

Due to space limitations, only proof sketches are given. Detailed proofs for the main theorems can be found in the appendix.

2 Model

2.1 Asynchronous distributed system

An *asynchronous distributed system* consists of a set of *processes* $\Pi = \{p_1, \ldots, p_n\}$. Each process p_i has a local state s_i, which is determined by the values of its local variables. The *local algorithm* A_i of process p_i describes *state transitions* of s_i, denoted as *events*. We distinguish

- *internal* events $s_i \rightarrow s'_i$, which just affect the local state of p_i,
- *send* events $s_i \rightarrow (s'_i, m)$, describing the sending of a message m, and
- *receive* events $(s_i, m) \rightarrow s'_i$, representing the reception of a message m.

The global state $S = (s_1, \ldots, s_n, \mathcal{M})$ is composed of the local states of the system's processes and the set of messages in transit. The *distributed algorithm* $A = (A_1, \ldots, A_n)$ is the collection of the local algorithms. Hence the transitions of the local algorithms yield the transitions of the distributed algorithm: Internal events modify the state of the corresponding process, send and receive events additionally modify the set of messages in transit.

An execution of A is a maximal sequence $(S_1, S_2, \ldots)$ of global states, such that A provides a transition $S_i \rightarrow S_{i+1}$ for all i. We assume weak fairness for all executions, hence any event which is applicable in an infinite number of concurrent states is eventually executed. Consequently, every sent message is eventually delivered, since the receive event is applicable as soon as a message is sent.

Note that we assume a "sane" communication system, i.e. every message is sent to a designated recipient, only the recipient may receive the message, and the recipient is notified of the identity of the sender. No message is received twice and messages may only be lost under certain circumstances, as described in the following section.

We define the *causal order* as the smallest relation $\prec$ on the events of an execution, which satisfies:

- If e and f are different events on the same process and e happens before f, then $e \prec f$.
- If e is a *send* event and f is the corresponding *receive* event, then $e \prec f$.
- $\prec$ is transitive, i.e. if $e \prec f$ and $f \prec g$, then $e \prec g$.

2.2 Failures

In order to model the failures in a distributed system, we introduce the *failure pattern* $F : \mathcal{T} \rightarrow 2^{\Pi}$. It maps from an element of the time domain $\mathcal{T}$ to a subset of processes. $F(t)$ denotes the set of processes which are *down* at time t — i.e. not functional and not executing any algorithmic steps. The time t is only used for modelling purposes, the processes do not have access to the current time t or $F(t)$. For simplicity we assume that the time domain equals the set of natural numbers, i.e. $\mathcal{T} = \mathbb{N}$.

A process $p_i \in \Pi$ is called *up* at time t, iff $p_i \notin F(t)$. We say that p_i *crashes* at time t, iff p_i is up at time $t-1$ and down at time t. Moreover, p_i *recovers* at time t, iff p_i is down at time $t-1$ and up at time t. If a process crashes, it loses its local state and stops to execute any steps of its algorithm. Messages which are in transit to p_i while p_i is down may be lost. If a process recovers, it continues to execute its local algorithm from some initial state. We demand that a recovering process "knows" that it is recovering. Note that we do not require stable storage in the sequel of this work.

Following the naming conventions introduced by Aguilera *et al.* [ACT00], a process p_i is denoted as

- *always up*, iff $\forall t : p \notin F(t)$,
- *eventually up*, iff $\exists t : (p \in F(t)) \wedge (\forall t' > t : p \notin F(t'))$,
- *eventually down*, iff $\exists t : \forall t' > t : p \in F(t')$, and
- *unstable*, iff $\forall t : ((p \in F(t) \Rightarrow \exists t' > t : p \notin F(t')) \wedge (p \notin F(t) \Rightarrow \exists t' > t : p \in F(t')))$.

A process p_i is called *finally up* at time t, iff $\forall t' \geq t : p_i \notin F(t')$. Process p_i is denoted as *finally down* at time t, iff $\forall t' \geq t : p_i \in F(t')$. Always up processes are finally up at time 0.

3 Synchronization

To describe the interface provided by the synchronizer module, we introduce a new system model: the *lockstep synchronous* distributed system. It provides the same functionality at its interface as a truly synchronous distributed system. It is based on rounds and guarantees the delivery of messages in the same round as they were sent. It is well known that a lockstep synchronous distributed system is implementable in an asynchronous distributed system in fault-free settings [Awe85]. An algorithm implementing such a lockstep synchronous distributed system is called synchronizer.

The main difference between a synchronous and a lockstep synchronous distributed system is the concurrency of the round pulses: While the pulse in a synchronous distributed system happens at the same real time on all processes, in a lockstep synchronous distributed system we only demand that the pulses happen at the same "causal time" (i.e. the causal order of the pulse events is the same as if they were issued at the same real time). This change is necessary because true simultaneousness is unfeasible in an asynchronous distributed system.

Furthermore, we introduce explicit pulse requests in a lockstep synchronous distributed system. While one may argue that each process of a synchronous distributed system is just fast enough to complete the calculations of each round before the next pulse happens, such an assumption is too optimistic in a lockstep synchronous distributed system due to the arbitrary processing speeds of the underlying asynchronous distributed system. Thus we introduce an explicit pulse request, which has to be issued by a process once it finished the current round.

The lockstep synchronous distributed system reflects the behavior of a synchronous distributed system in the presence of failures: If a process crashes, the messages sent by the crashing process in the round of the crash are still guaranteed to be delivered. When a process recovers, it will eventually *resynchronize* by being granted some pulse numer and continue to take part in the distributed computation. Note that the pulse number granted during resynchronization has to be bigger than any other ever granted round number in the distributed system to allow the resynchronizing process to start in a "fresh" round.

An example execution for a lockstep synchronous distributed system is depicted in figure 2. The processes request the next pulse as soon as they finished the algorithmic steps for the current round. The next pulse is granted when all messages of the current round are delivered. Note that in a realistic implementation of a lockstep synchronous distributed system no process can be granted the next pulse before every other process initiated a request for that pulse. Thus for each pulse we can find a point in time, which separates all pulse requests from all pulse grants, as illustrated by the solid vertical lines in figure 2. Also note that the pulse grants provide an easy mean to find a consistent cut [Mat89] of the distributed computation.

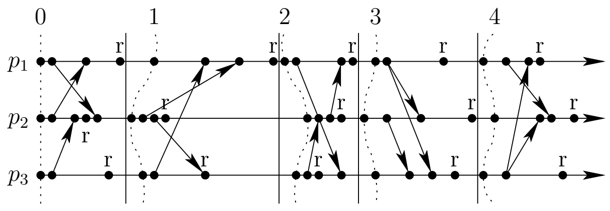

Fig. 2. Example execution in a lockstep synchronous distributed system: Pulse requests are abbreviated by "r", pulse grants for equal pulse numbers are connected by dotted lines.

Definition 1 (Lockstep Synchrony). *Let S be a distributed system with processes $p_1, \ldots, p_n$ and crash-recovery faults. The system provides the following interface to the processes:*

- *$pulsereq_i(r)$: Process p_i request pulse r from the system.*
- *$pulsegrant_i(r)$: The system grants pulse r to process p_i.*
- *$sync\_send_i(m)$: Process p_i sends message m.*
- *$sync\_receive_i(m)$: Process p_i receives message m.*

Let r_i be the biggest r' for which $pulsegrant_i(r')$ occurred. If no $pulsegrant_i(r')$ ever occurred, r_i is defined as 0. Let F be a failure pattern and let $T(e)$ denote the time at the occurrence of the event e.

A process p_i is called participating in round r, *if*

$$\forall t \in \{T(pulsegrant_i(r)), \ldots, T(pulsegrant_i(r+1))\} : p_i \notin F(t) \quad .$$

p_i is called participating *if it participates in round r_i.*

The following assumptions are made for the algorithm running on the processes:

- *No process p_i performs a sync_send$_i(m)$ (for any m) between the occurrences of pulsereq$_i(r)$ and pulsegrant$_i(r)$ (for any r).*
- *Every participating process p_i eventually executes pulsereq$_i(r_i + 1)$.*
- *Every process p_i executes pulsereq$_i(r)$ at most once for each r.*
- *Initially every process p_i waits for a pulsegrant$_i(0)$ before executing any request.*
- *Every recovering process p_i waits for a pulsegrant$_i(r)$ before executing any request.*

Such a system S is called lockstep synchronous distributed system*, if it satisfies the following conditions:*

- Integrity*: Every message received by process p_i from process p_j between pulsegrant$_i(r-1)$ and pulsegrant$_i(r)$ was sent by p_j between pulsegrant$_j(r-1)$ and pulsegrant$_j(r)$.*
- No Duplication*: No message is received more than once.*
- Validity*: If a process p_i gets a pulsegrant$_i(r)$, it has received all messages sent by each process p_j in each round r', $r' < r$, in which p_i participated.*
- Progress*: If a process p_i invokes pulsereq$_i(r)$ at some time t and p_i is finally up at time t, it will eventually experience pulsegrant$_i(r)$.*
- Startup*: Initially every correct process p_i will experience pulsegrant$_i(0)$.*
- Resynchronization Liveness*: If a process p_i recovers at some time t and p_i is finally up at time t, it will eventually receive a pulsegrant$_i(r)$.*
- Resynchronization Safety*: If a process p_i recovers and receives a pulsegrant$_i(r)$ for some r, r is at least larger by 2 than any number of a round in which some process participated.*

A *synchronizer* is a distributed algorithm, whose local modules provide the interface of a lockstep synchronous distributed system to the encapsulated algorithms.

4 Failure Detection Devices

A *failure detection device* is a distributed oracle which outputs information about the failures in an asynchronous distributed system based on the failure pattern $F(t)$. We focus on *interrupt-style* failure detection devices which notify their local algorithms of changes in the failure information by triggering events.

Roughly speaking, a failure detection device outputs, at some time t and some process p_i, information about $F(t)$ *and* the computation history up to t. The definition of a failure detection devices includes the “classical” failure detectors of Chandra and Toueg [CT96] as well as the failure detection sequencers of Gärtner and Pleisch [GP02]. Failure detectors provide information about the mere occurrence of failures in the system. Failure detection sequencers additionally output state information about the crashed processes.

Given two failure detection devices $\mathcal{D}$ and $\mathcal{D}'$, we say that $\mathcal{D}$ *emulates* $\mathcal{D}'$ (denoted $\mathcal{D} \succeq \mathcal{D}'$), iff there exists an asynchronous algorithm $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ which uses $\mathcal{D}$ as an algorithmic module and whose output is indistinguishable from the output of $\mathcal{D}'$. If $\mathcal{D} \succeq \mathcal{D}'$ and $\mathcal{D}' \not\succeq \mathcal{D}$, we say that $\mathcal{D}$ is *strictly stronger* than $\mathcal{D}'$.

A failure detection device $\mathcal{D}$ is called the *weakest failure detection device* for some algorithmic problem P, if $\mathcal{D}$ is necessary and sufficient to solve P in a fault-prone environment. The existence of an algorithm $A_{\mathcal{D}}$, which uses $\mathcal{D}$ and solves P, proves that $\mathcal{D}$ suffices to solve P. To prove the necessity of $\mathcal{D}$, it has to be shown that any failure detection device $\mathcal{D}'$ which allows to solve P emulates $\mathcal{D}$.

Since this work extends the result of Gärtner and Pleisch [GP02] to the crash-recovery model, the concept of a failure detection sequencer needs to be adapted to the crash-recovery failure model. Thus at first we introduce a new failure detector for the crash-recovery model — the perfect failure detector $\mathcal{P}_{CR}$ — which will be the basis for the sequencer defined later.

Definition 2 (Perfect Failure Detector (Crash-Recovery)). *The* perfect failure detector for the crash-recovery failure model $\mathcal{P}_{CR}$ *is an interrupt-style failure detector, which issues two events at its interface:*

- *suspect*$_i(p_j)$*: The failure detector of process p_i suspects p_j to be crashed.*
- *welcome*$_i(p_j)$*: The failure detector of process p_i claims that p_j recovered. We say that p_i welcomes p_j.*

$\mathcal{P}_{CR}$ *satisfies the following properties:*

- Integrity*: Every process is suspected at most once for every crash.*
- Crash Completeness*: If a process crashes and is finally down, it will eventually be suspected and no longer welcomed by every finally up process.*
- Recovery Completeness*: If a process recovers and is finally up, it will eventually be welcomed and no longer suspected by every finally up process.*
- Unstable Completeness*: Every unstable process will be suspected and welcomed an infinite number of times by every finally up process.*
- Suspect Validity*: If a process is suspected, it either was never suspected before, or it was welcomed after the last time it was suspected.*
- Welcome Validity*: Each process is welcomed at most once after each time being suspected.*

Note that $\mathcal{P}_{CR}$ may miss crashes and recoveries. The properties only guarantee the detection of the *last* crash or recovery of each process.

In the following we extend $\mathcal{P}_{CR}$ by adding state information to the suspect and welcome events. Gärtner and Pleisch [GP02] originally used an arbitrary number of predicates on the state of the crashed process as state information. We use a slightly weaker variant here and set the state information to the set of messages which were recently sent by the suspected (or welcomed) process to the suspecting (or welcoming) process. This weaker failure detection sequencer suffices in our context.

Definition 3 (Failure Detection Sequencer (Crash-Recovery)). *The* failure detection sequencer for the crash-recovery failure model Σ_{CR} *is an interrupt-style failure detection sequencer, which issues two events at its interface:*

- $suspect_i(p_j, s)$ *indicates a crash of* p_j *in the state* s*, and*
- $welcome_i(p_j, s)$ *indicates a recovery of* p_j *after a crash in state* s*,*

both issued by the failure detection sequencer module of process p_i*.*

The state-information about a process p_j *delivered by the crash-recovery sequencer module of process* p_i *is the set of messages that were sent by* p_j *to* p_i *after time* t'*, where* t' *is the time of the last recovery of* p_i*. Formally:*

$$\begin{aligned} last\_recovery(p_i, t) &= max\{t' | p_i \text{ recovered at time } t', t' \leq t\} \\ State_i(p_j, t) &= \{m | \text{message } m \text{ was sent by } p_j \text{ to } p_i \\ &\qquad\qquad \text{after } last\_recovery(p_i, t) \text{ and until time } t\} \end{aligned}$$

Σ_{CR} *satisfies the following properties:*

- Integrity*: Every process is suspected at most once for every crash.*
- Accuracy*: If a process is suspected to be crashed in state* s *or welcomed after a crash in state* s*, it did crash in state* s*.*
- Crash Completeness*: If a process crashes in some state* s *and is finally down, it will eventually be suspected to be crashed in* s *and no longer welcomed by every finally up process.*
- Recovery Completeness*: If a process recovers after a crash in state* s *and is finally up, it will eventually be welcomed after a crash in state* s *and no longer suspected by every finally up process.*
- Unstable Completeness*: Every unstable process will be suspected and welcomed an infinite number of times by every finally up process.*
- Suspect Validity*: If a process is suspected, it either was never suspected before, or it was welcomed after the last time it was suspected.*
- Welcome Validity*: Each process is welcomed at most once after each time being suspected.*

We will later show that Σ_{CR} can be implemented in lockstep synchrony. This implies that it is implemented under the same synchrony assumptions as the original sequencer of Gärtner and Pleisch [GP02].

5 Impossibility Result

We will now present a result which shows that it is impossible to implement a synchronizer in an asynchronous distributed system prone to crash-recovery failures, when we allow an arbitrary number of process crashes. Hence, in the remainder of this work, we postulate that at least one process in the distributed system is correct.

Theorem 1. *It is impossible to implement a lockstep synchronous distributed system in the crash-recovery failure model even with a crash-recovery sequencer if all processes are allowed to be down at the same time.*

Proof sketch: Assume by ways of contradition that a synchronizer algorithm exists. We construct two indistinguishable runs of that algorithm, where one of them violates the resynchronization safety requirement (see Figures 3 and 4): In run R_1 two processes p_1 and p_2 are initially down. Process p_2 eventually recovers and is granted some round number $r > 0$. In run R_2, p_2 is initially down, and p_1 is granted rounds 0 to $r+1$ and crashes. Afterwards, p_2 recovers. Due to the indistinguishability of runs R_1 and R_2 from the point of view of p_2, p_2 is granted round r, violating the resynchronization safety requirement, a contradiction.

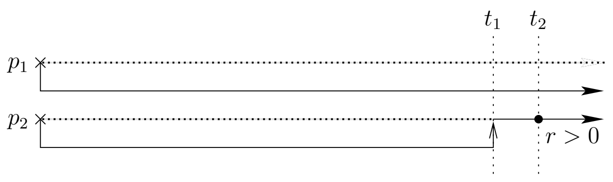

Fig. 3. Run R_1 in the proof of theorem 1. The *pulsegrant* events are annotated with their respective round numbers.

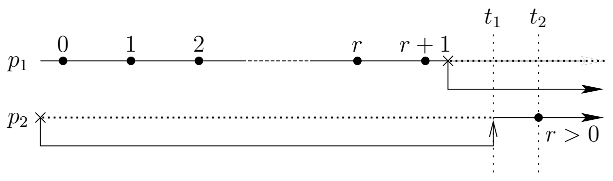

Fig. 4. Run R_2 in the proof of theorem 1. The *pulsegrant* events are annotated with their respective round numbers. The *pulsereq* events executed by p_1 are omitted.

6 Algorithms

In this section we first implement a synchronizer for the crash-recovery failure model using the crash-recovery failure detection sequencer Σ_{CR}. We subsequently show that Σ_{CR} is also necessary to implement a synchronizer in that model.

6.1 Synchronizer Algorithm

As shown in Section 5, it is justifiable to assume at least one correct process. Using such a process, algorithm 1 implements a synchronizer for the crash-recovery failure model. The algorithmic ideas are based on the synchronizer α proposed by Awerbuch [Awe85]. It can be regarded as an extension of a simplified version of the crash-stop synchronizer of Gärtner and Pleisch [GP02].

Messages sent by the synchronous algorithm are asynchronously transmitted to the recipient (lines 18–21). If a process finishes the computation of the current round, it signals this fact to the synchronizer by executing the *pulsereq* event and the synchronizer broadcasts a "done" message to all other processes in the system, along with the number of messages sent in the current round to the respective process (lines 22–27). Once a synchronizer received a done message for the current round from every process in the system, and it is certain that no more messages are in transit, it grants the next round to the encapsulated algorithm (lines 48–59). The issue of messages being delivered too early (pointed out by Lakshmanan and Thulasiraman [LT88]) is dealt with by buffering (lines 42–46) and delayed delivery (lines 60–66).

If a process crashes, it is eventually suspected by the sequencer module of every finally up process. In this case, the suspecting process no longer waits for the "done" message of the suspected process. Furthermore it extracts all messages out of the state information provided by the sequencer and initiates the delivery of not yet delivered messages.

The algorithm uses a variant of crash-stop fault tolerant uniform consensus as a building block — denoted as *max-consensus*. An algorithm solves the max-consensus problem if it satisfies the following properties:

- *Termination*: Every correct process eventually decides.
- *Uniform Agreement*: No two processes decide differently.
- *Validity*: Every decided value is greater than or equal to the maximum of the values proposed by correct processes.
- *Integrity*: No process decides twice.

Unlike uniform consensus, max-consensus does not decide on any value proposed by a correct process, but on a value which is at least as large as the maximum of all values proposed by correct processes. Max-consensus is easily implementable as a slight modification of the FloodSet algorithm proposed by Lynch [Lyn96]. The use of max-consensus together with the assumption of one correct process are crucial in achieving the resynchronization properties of the algorithm. Both ensure that round numbers monotonically increase.

If a crashed process recovers, it broadcasts an "I want to join" message and starts an instance of a max-consensus algorithm (lines 84–89). Every currently up process proposes its current round number incremented by 2 (lines 29–31). The processes decide on a common round number, which will be used as the first round number of the resynchronizing process (line 93).

Note that a crash-stop failure resistant version of max-consensus suffices in our case, although the synchronizer is prone to crash-recovery faults. Processes which crash during the execution of max-consensus do not take part in the rest of the max-consensus execution (regardless of a later recovery), since their proposal becomes irrelevant due to the crash. Thus we may "emulate" a crash-stop environment by assuming that all crashes are permanent. Furthermore, we can easily emulate a perfect crash-stop failure detector $\mathcal{P}$ by mapping the suspect events of Σ_{CR} to suspect events of $\mathcal{P}$ once for every crashing process.

Theorem 2. *Algorithm 1 provides the interface of a lockstep synchronous distributed system (as defined in definition 1) to the underlying processes, if they satisfy the assumptions in definition 1 and if they are prone to crash-recovery failures and at least one process is correct.*

6.2 Σ_{CR} is Necessary

Theorem 2 shows that Σ_{CR} is sufficient to implement a synchronizer in an asynchronous crash-recovery distributed system. We now show that Σ_{CR} is not only sufficient, but also necessary. We do this by implementing Σ_{CR} directly in a lockstep synchronous distributed system.

Algorithm 2 implements Σ_{CR} by the usage of a monitor process. Every monitor sends an "I'm in round r" message in every round. If some process misses to send this message due to a crash, it is suspected by all other processes. Every recovering monitor sends an "I recovered" message to every other monitor in the system as soon as it receives its first *pulsegrant* event after the recovery (line 11). Every monitor receiving this messages signals the recovery to its underlying process by issuing a *welcome* event (lines 30–34).

Theorem 3. *Algorithm 2 implements the sequencer Σ_{CR} in a lockstep synchronous distributed system.*

Proof sketch: We prove the properties of Σ_{CR} one by one: The crash completeness property is ensured by the progress property of the lockstep synchronous distributed system. The recovery and unstable completeness properties are guaranteed by the validity property of the lockstep synchronous distributed system. The accuracy property is also proven with the validity property.

Together with theorem 2 follows that Σ_{CR} is the weakest failure detection device suitable to implement such a synchronizer.

7 Discussion

In this work we investigated the problem of network synchronization in asynchronous distributed systems prone to crash-recovery faults. We reduced this problem to the problem of implementing a *synchronizer*, which provides a universal synchronization abstraction (roughly) by transforming an asynchronous distributed system into a synchronous one. We proved that this problem in unsolvable if all processes in the system are allowed to be down concurrently. Informally, the information about the synchrony of the whole system gets lost in such a scenario.

Algorithm 1 Crash-recovery synchronizer with sequencer Σ_{CR} (part 1)

```
Variables:
$current\_request \in \mathbb{N}$
$current\_round \in \mathbb{N}$
$receive\_buffer \subseteq \Pi \times \mathbb{N} \times \mathcal{M}$
$participating \subseteq \{1, \ldots, n\}$
$joining \in (\mathbb{N} \cup \{\bot, ?\})^n$
$send\_count \in \mathbb{N}^n$
$receive\_count \in \mathbb{N}^n$
$wait\_count \in (\mathbb{N} \cup \{\bot\})^n$

Process $p_i$:
upon $\langle init_i \rangle$ do
    $current\_request := 0$
    $current\_round := 0$
    $wait\_count := (0, \ldots, 0)$
    $receive\_buffer := \emptyset$
    $participating := \{1, \ldots, n\}$
    $joining := (\bot, \ldots, \bot)$
    trigger $\langle pulsegrant_i(0) \rangle$
end upon

upon $\langle sync\_send_i(p_j, m) \rangle$ do
    trigger $\langle async\_send_i(p_j, (current\_round, m)) \rangle$
    $send\_count[j] := send\_count[j] + 1$
end upon

upon $\langle pulsereq_i(r) \rangle$ do
    $current\_request := r$
    for all $j \in \{1, \ldots, n\}$ do
        trigger $\langle async\_send_i(p_j, (current\_round, (\text{“done”}, send\_count[j]))) \rangle$
    end for
end upon

upon $\langle async\_receive_i(p_j, (r, m)) \rangle$ do
    if $m = \text{“I want to join”}$ then
        $joining[j] := ?$
        trigger $\langle max\_consensus\_propose(j, current\_round + 2) \rangle$
    else
        if $r = current\_round$ then
            if $m = (\text{“done”}, cnt)$ then
                $wait\_count[j] := cnt$
            else
                if $m$ is no duplicate message then
                    trigger $\langle sync\_receive_i(p_j, m) \rangle$
                    $receive\_count[j] := receive\_count[j] + 1$
                end if
            end if
        else
            /* message is too early, store it */
            $receive\_buffer := receive\_buffer \cup \{(p_j, r, m)\}$
        end if
    end if
end upon
```

Algorithm 1 Crash-recovery synchronizer with sequencer Σ_{CR} (part 2)

```
48: upon $\langle(\forall j \in participating : wait\_count[j] = receive\_count[j]) \wedge (\forall j : joining[j] \neq ?\rangle$ do
49:     $wait\_count := (\perp, \ldots, \perp)$
50:     $send\_count := (0, \ldots, 0)$
51:     $receive\_count := (0, \ldots, 0)$
52:     $current\_round := current\_request$
53:     for all $j \in \{1, \ldots, n\}$ do
54:         if $joining[j] = current\_round$ then
55:             $participating := participating \cup \{j\}$
56:             $joining[j] := \perp$
57:         end if
58:     end for
59:     trigger $\langle pulsegrant_i(current\_request)\rangle$
60:     for all $(p, r, m) \in receive\_buffer$ do
61:         /* receive messages that are "on time" now */
62:         if $r = current\_round$ then
63:             trigger $\langle async\_receive\_fifo_i(p, (r, m))\rangle$
64:             $receive\_buffer := receive\_buffer \setminus \{(p, r, m)\}$
65:         end if
66:     end for
67: end upon

68: upon $\langle \Sigma$ suspects $p_j$ in state $s\rangle$ do
69:     /* Emulate reception of sent messages */
70:     for all $(r, m) \in s$ do
71:         trigger $\langle async\_receive\_fifo_i(p_j, (r, m))\rangle$
72:     end for
73:     $participating := participating \setminus \{j\}$
74:     $joining[j] := \perp$
75: end upon

76: upon $\langle recovery_i\rangle$ do
77:     /* Perform initialization */
78:     $current\_request := 0$
79:     $current\_round := 0$
80:     $wait\_count := (0, \ldots, 0)$
81:     $receive\_buffer := \emptyset$
82:     $participating := \{1, \ldots, n\}$
83:     $joining := (\perp, \ldots, \perp)$
84:     /* Inform other processes about resynchronization */
85:     for all $j \in \{1, \ldots, n\} \setminus \{i\}$ do
86:         trigger $\langle async\_send\_fifo_i(p_j, (0,$ "I want to join"$))\rangle$
87:     end for
88:     /* Propose 0 to instance $i$ of max-consensus */
89:     trigger $\langle max\_consensus\_propose(i, 0)\rangle$
90: end upon

91: upon $\langle max\_consensus\_decide_i(j, r)\rangle$ do
92:     if $i = j$ then
93:         trigger $\langle pulsegrant_i(r)\rangle$
94:     else
95:         $joining[j] := r$
96:     end if
97: end upon
```

Algorithm 2 Emulating crash-recovery sequencer Σ_{CR} in a lockstep synchronous distributed system.

```
Variables:
$current\_round \in \mathbb{N}^n$
$state \in \mathbb{M}(\mathcal{M})^n$
$suspected \in \{0,1\}^n$
Process monitor of process $p_i$:
upon $\langle pulse\_grant_i(r)\rangle$ do
    for all $j \in \{1,\ldots,n\} \setminus \{i\}$ do
        trigger $\langle sync\_send_i(j,$"I'm in round r"$)\rangle$
    end for
    if recovery then
        for all $j \in \{1,\ldots,n\} \setminus \{i\}$ do
            trigger $\langle sync\_send_i(j,$"I recovered"$)\rangle$
        end for
        $state := (\emptyset,\ldots,\emptyset)$
        $current\_round := (r-1,\ldots,r-1)$
        $suspected := (0,\ldots,0)$
    else
        for all $j \in \{1,\ldots,n\} \setminus \{i\}$ do
            if $r \geq current\_round[j] + 2$ then
                if $suspected[j] = 0$ then
                    trigger $\langle \text{suspect}(p_j, state[j])\rangle$
                    $suspected[j] := 1$
                end if
            end if
        end for
        trigger $\langle pulse\_grant_i(r)\rangle$ at underlying process $p_i$
    end if
end upon

upon $\langle sync\_receive_i(j,m)\rangle$ do
    if $m =$"I'm in round $r$" then
        $current\_round[j] := r$
    else if $m =$"I recovered" then
        if $suspected[j] = 0$ then
            trigger $\langle \text{suspect}(p_j, state[j])\rangle$
        end if
        trigger $\langle \text{welcome}(p_j, state[j])\rangle$
        $suspected[j] := 0$
    else
        $state[j] := state[j] \cup \{m\}$
        trigger $\langle sync\_receive_i(j,m)\rangle$ at underlying process $p_i$
    end if
end upon

upon $\langle init\rangle$ do
    $state := (\emptyset,\ldots,\emptyset)$
    $current\_round := (-1,\ldots,-1)$
    $suspected := (0,\ldots,0)$
end upon
```

Consequently, recovering processes are unable to continue their computation in a state which satisfies the synchrony properties of the distributed system.

Given one correct process, we showed that the failure detection sequencer Σ_{CR} is the weakest failure detection device suitable to implement a synchronizer. It provides information about the crashes and recoveries in the system and about the state of the outgoing communication channels of crashed processes. This channel state information is essential for the implementation of a synchronizer. Since Σ_{CR} suffices to implement a synchronizer, it encapsulates all information that an asynchronous distributed system prone to crash-recovery failures lacks compared to a synchronous one.

Future work. We did not elaborate on the efficiency of our synchronizer algorithm in this work. Future work might concentrate on the improvement its efficiency by means already proposed for the fault-free synchronizer α by Awerbuch [Awe85] or Peleg and Ullman [PU87]. But these improvements rely on more efficient communication trees, which would have to be reconstructed upon every failure and thus might not lead to actual improvements in our failure model.

An open question is the relation of the perfect failure detector for the crash-recovery failure model, $\mathcal{P}_{CR}$, to other failure detectors for this failure model: $\mathcal{P}_{CR}$ can easily emulate the ACT failure detector by Aguilera, Chen, and Toueg [ACT00]. Hence $\mathcal{P}_{CR}$ is at least as strong as ACT. We conjecture that it is even strictly stronger than the ACT failure detector, because ACT does not provide means to guarantee the integrity property in an emulation of $\mathcal{P}_{CR}$.

References

[ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[CBGS00] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 523–532, New York, USA, 2000. IEEE Computer Society.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[GP02] Felix C. Gärtner and Stefan Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 280–294, London, UK, 2002. Springer-Verlag.

[Lam95] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August/September 1995.

[LT88] Kadathur B. Lakshmanan and Krishnaiyan Thulasiraman. On the use of synchronizers for asynchronous communication networks. In *Proceedings of the 2nd International Workshop on Distributed Algorithms*, pages 257–277, London, UK, 1988. Springer-Verlag.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CS, 1996.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, 1989. Elsevier Science Publishers.

[PU87] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computi ng*, pages 77–85, New York, NY, USA, 1987. ACM Press.

A Proofs

Proofs are written in a structured style similar to proof trees of interactive theorem proving environments. This approach is advocated by Lamport who promises that this style "makes it much harder to prove things that are not true" [Lam95]. The proof is a sequence of numbered proof steps at different levels. Every proof step has a proof which may be refined at lower levels by additional proof steps. Proofs may also be read in a structured way, for example, by reading only the top level proof steps and going into sublevels only when necessary.

Theorem 1. *It is impossible to implement a lockstep synchronous distributed system in the crash-recovery failure model even with a crash-recovery sequencer if all processes are allowed to be down at the same time.*

Proof sketch: We prove this theorem by showing that it is impossible for any algorithm trying to implement a lockstep synchronous distributed system to satisfy the resynchronization safety requirement. We show this by constructing two indistinguishable runs, where one of them violates the resynchronization safety requirement.

Proof:

1. Assume: There exists an algorithm A that implements a lockstep synchronous distributed system in an asynchronous distributed system given an arbitrary number of crash-recovery failures and a crash-recovery sequencer.

   Prove: False.

   1.1. Consider the run R_1 depicted in figure 3: p_1 initially crashes and p_2 initially crashes, recovers at time t_1 and is finally up. When the algorithm A is executed with this failure pattern, there exists a time $t_2 > t_1$ when p_2 experiences a *pulsegrant*$_2(r)$ with $r > 0$.

PROOF: We assume that A is a correct implementation of a lockstep synchronous distributed system. Hence the resynchronization safety and liveness properties are satisfied by A. Therefore, there must exist some time t_2 after the recovery of p_2 when p_2 is granted a pulse number r with $r > 0$. $\square$

1.2. Consider the run R_2 of A, depicted in figure 4: p_2 initially crashes, revovers at time t_1 and is finally up, p_1 is granted the round numbers 0 to $r + 1$ and crashes before the recovery of p_2. This run is indistinguishable from R_1 from the point of view of p_2.

PROOF: Let A_1 and A_2 be the local parts of the algorithm A on p_1 and p_2. In both runs R_1 and R_2, A_1 and A_2 are not able to exchange any messages: In run R_1 A_1 is never executed, because p_1 is initially down and never recovers. In run R_2 p_2 is down when p_1 is up and thus we may assume that any message sent by A_1 is lost. Hence the message-trace observed by p_2 in run R_1 is indistinguishable from the message-trace observed by p_2 in run R_2. Furthermore, the output of the sequencer module of A_2 is equal in R_1 and R_2: In both runs p_1 is eventually suspected by the sequencer module of A_2 with an empty set of messages as state information. Thus R_1 and R_2 are indistinguishable from the viewpoint of p_2. $\square$

1.3. In run R_2 the algorithm A must issue $\mathit{pulsegrant}_2(r)$ with the same r as in run R_1.

PROOF: Step 1.1 shows that p_2 must be granted a round number r with $r > 0$ at some time $t_2 > t_1$. Furthermore, the runs R_1 and R_2 are indistinguishable for p_2, as shown in step 1.2. Because A is a deterministic algorithm, it has to issue $\mathit{pulsegrant}_2(r)$ at time t_2 in run R_2, with the same r as in run R_1. $\square$

1.4. Q.E.D.

PROOF: Step 1.3 contradicts the assumption that A is correct: It violates the resynchronization safety requirement. $\square$

2. Q.E.D.

PROOF: Follows indirectly from step 1.

Theorem 2. *Algorithm 1 provides the interface of a lockstep synchronous distributed system (as defined in definition 1) to the underlying processes, if they satisfy the assumptions in definition 1 and if they are prone to crash-recovery failures and at least one process is correct.*

PROOF SKETCH: We are going to prove the properties of a lockstep synchronous distributed system one by one: The *no duplication* property is achieved by the explicit no duplication check in the algorithm. *Progress* is satisfied because of the liveness assumptions of the underlying processes, the communication system, and the sequencer. *Validity* and *integrity* are guaranteed by the preconditions of the *sync_receive* and *pulsegrant* events. The *resynchronization* properties are guaranteed by the used max-consensus subprotocol. The *startup* property is ensured by granting the initial pulse in the initialization of the algorithm.

PROOF:

1. ASSUME: At least one process is correct.

   PROVE: The algorithm satisfies all properties of a lockstep synchronous distributed system.

1.1. The algorithm satisfies no duplication, i.e. no message is received more than once.

PROOF: The delivery of each synchronous message m is triggered in line 38 of the algorithm. Thus m must have passed the check for duplicate messages in line 37. Hence no message can be received twice. □

1.2. The algorithm satisfies integrity, i.e. every message received by process p_i from process p_j between $\mathit{pulsegrant}_i(r-1)$ and $\mathit{pulsegrant}_i(r)$ was sent by p_j between $\mathit{pulsegrant}_j(r-1)$ and $\mathit{pulsegrant}_j(r)$.

1.2.1. If p_i delivers a message m sent by p_j and the last round number granted to p_i is r', the last round number granted to p_j was also r' when it sent the message.

PROOF: The delivery of the message m is initiated in line 38 of the algorithm. Thus the test in line 33 was passed and $r = \mathit{current\_round}$ holds. $\mathit{current\_round}$ contains the latest round number which was granted to p_i, hence $\mathit{current\_round} = r' = r$. Furthermore r is the latest round number granted to p_j when it sent m. □

1.2.2. Q.E.D.

Follows from step 1.2.1 and the fact that the processes are granted consecutive round numbers. □

1.3. The algorithm satisfies progress, i.e. if a finally up process p_i invokes $\mathit{pulsereq}_i(r)$, it will eventually experience $\mathit{pulsegrant}_i(r)$.

1.3.1. ASSUME: Process p_i is finally up in round r and invokes $\mathit{pulsereq}_i(r+1)$ at time t_1.

PROVE: $\exists t_2 > t_1$: process p_i experiences $\mathit{pulsegrant}_i(r+1)$ at time t_2.

1.3.1.1. Every process p_j which participates in round r eventually executes $\mathit{pulsereq}_j(r+1)$.

PROOF: Follows from the assumption in definition 1. □

1.3.1.2. Every synchronizer module at each process p_j participating in round r eventually asynchronously sends a "done" message for round r to each process.

PROOF: Follows from step 1.3.1.1 and lines 22–26 of the algorithm. □

1.3.1.3. Process p_i will eventually receive "done" messages from all processes participating in round r.

PROOF: Process p_i will never crash, because we assume that it is finally up. Thus the liveness property of the underlying communication system and step 1.3.1.2 guarantee that the "done" messages will eventually be received. □

1.3.1.4. Process p_i eventually suspects all processes, which do not participate in round r.

PROOF: Process p_i will never crash, because we assume that it is finally up. Thus the completeness properties of the sequencer guarantee that every process which is not participating will eventually be suspected. □

1.3.1.5. Eventually p_i will decide some value for all instances of max-consensus started in the current round.

PROOF: p_i is finally up, i.e. it looks like a correct process to the crash-stop max-consensus algorithm. Hence the termination property of max-consensus guarantees that p_i will eventually decide some value for all instances of max-consensus. □

1.3.1.6. $\exists t_2 > t_1$: at time t_2 the synchronizer module of p_i triggers a $\mathit{pulsegrant}_i(r+1)$ at the underlying process.

PROOF: From step 1.3.1.4 and line 73 follows that eventually holds:

$$j \in \mathit{participating} \Rightarrow p_j \ \mathit{participates\ in\ round}\ r$$

Moreover step 1.3.1.3 and line 35 guarantee that eventually holds:

$$j \in \mathit{participating} \Rightarrow \mathit{received}[j] = 1$$

From step 1.3.1.5 and line 95 follows that eventually holds:

$$\forall j \in \{1, \ldots, n\} : \mathit{joining}[j] \neq ?$$

Thus the condition in line 48 will eventually be true and $\mathit{pulsegrant}_i(r+1)$ will be executed. □

1.3.1.7. Q.E.D.

Follows directly from step 1.3.1.6. □

1.3.2. Q.E.D.

Follows directly from the step 1.3.1. □

1.4. The algorithm satisfies validity, i.e. if a process p_i gets a $\mathit{pulsegrant}_i(r)$, it has received all messages sent by each process p_j in each round r', with $r' < r$, in which p_i participated.

1.4.1. If a process p_i participates in round $r-1$ and gets a $\mathit{pulsegrant}_i(r)$, it has received all messages sent by all processes in round $r-1$.

1.4.1.1. p_i has received all messages sent by all processes p_j in round $r-1$, with $j \in \mathit{participating}$.

PROOF: The condition in line 48 guarantees that a "done" was received for every p_j with $j \in \mathit{participating}$. Furthermore, the condition guarantees that for each process p_j the number of messages received from p_j by p_i matches the number of messages sent by p_j to p_i. Due to the no duplication property shown in step 1.1, all messages must have been received. □

1.4.1.2. p_i has received all messages sent by all processes p_j in round $r-1$, with $j \notin \mathit{participating}$.

PROOF: When j is removed from *participating* (line 73), information about all messages sent by p_j in round $r-1$ is provided by the sequencer and their reception is initiated (line 71). □

1.4.1.3. Q.E.D.

Follows from steps 1.4.1.1 and 1.4.1.2. □

1.4.2. Q.E.D.

Follows from step 1.4.1 and the fact that each process requests and is granted growing pulse numbers. □

1.5. The algorithm satisfies resynchronization safety, i.e. if a process p_i recovers and receives a $\mathit{pulsegrant}_i(r)$, r is at least larger by 2 than any number of a round in which some process participated.

1.5.1. The maximum value proposed to instance i of max-consensus is larger by 2 than any number of a round in which some process participated.

PROOF: Each proposed value is the current round number of the proposing process incremented by 2. Thus the maximum value is larger by 2 than any ever reached round. □

1.5.2. If instance i of max-consensus decides some value, it is at least larger by 2 than any number of a round in which some process participated.

PROOF: Step 1.5.1 shows that all proposed values are bigger by 2 than any ever reached round. The validity property of max-consensus guarantees, that the decided value is at least equal to the maximum of all proposed values. □

1.5.3. Q.E.D.

Step 1.5.2 shows that the value decided by p_i is at least larger by 2 than any round number of a round in which some process participated. This value is granted as the next round number in line 93. □

1.6. The algorithm satisfies resynchronization liveness, i.e. if a process p_i recovers and is finally up, it will eventually receive a $\mathit{pulsegrant}_i(r)$.

1.6.1. ASSUME: p_i recovers at time t_1 and is finally up.

PROVE: $\exists t_2 > t_1$: p_i invokes $\mathit{pulsegrant}_i(r)$ at time t_2.

1.6.1.1. $\exists t_3 > t_1 : p_i$ proposes a value for max-consensus at time t_3.

PROOF: After recovery p_i will eventually execute line 89 of the algorithm. Hence t_3 exists. □

1.6.1.2. $\exists t_4 \geq t_3$: All finally up processes proposed a value for max-consensus at time t_4.

PROOF: Step 1.6.1.1 shows that p_i eventually proposes a value. All other finally up processes eventually receive the "I want to join" message from p_i. Hence all finally up processes eventually propose some value for max-consensus in line 31. Thus t_4 exists. □

1.6.1.3. $\exists t_2 > t_4 : p_i$ decides a value for max-consensus at time t_2.

PROOF: Step 1.6.1.2 shows that all finally up process eventually propose a value for max-consensus. Thus the termination property of max-consensus ensures that eventually all finally up processes decide some value. As p_i is finally up, there exists a time $t_2 > t_4$ when p_i decides a value. □

1.6.1.4. Q.E.D.

Step 1.6.1.3 shows that p_i eventually decides some value for max-consensus. This value is granted to p_i as the next round number in line 93. □

1.6.2. Q.E.D.

Follows from step 1.6.1. □

1.7. The algorithm satisfies startup, i.e. initially every correct process p_i will experience $\mathit{pulsegrant}_i(0)$.

PROOF:Follows directly from line 16 of the algorithm. □

1.8. Q.E.D.

PROOF: The algorithm satisfies all seven properties of a lockstep synchronous distributed system, as shown in steps 1.1–1.7. □

2. Q.E.D.

PROOF: Follows from step 1. □

Theorem 3. *Algorithm 2 implements the sequencer Σ_{CR} in a lockstep synchronous distributed system.*

PROOF SKETCH: We will prove the properties of Σ_{CR} one by one: The crash completeness property is ensured by the progress property of the lockstep synchronous distributed system. The recovery and unstable completeness properties are guaranteed by the validity property of the lockstep synchronous distributed system. The accuracy property is also proven with the validity property.

PROOF:

1. The algorithm satisfies all properties of the crash-recovery sequencer Σ_{CR}.

1.1. The algorithm satisfies accuracy, i.e. if a process is suspected to be crashed in state s or welcomed after a crash in state s, it did crash in state s.

PROOF: If a process is suspected or welcomed it must have crashed, because it either missed to send an "I'm in round r" message for some round or it sent an "I recovered" message after recovery. Furthermore the provided state information matches the set of messages that were received by the suspecting process after its last recovery. The validity property of the lockstep synchronous distributed system ensures that this set contains all messages sent by the suspected process. □

1.2. The algorithm satisfies crash completeness, i.e. if a process crashes in some state s and is finally down, it will eventually be suspected to be crashed in s and no longer welcomed by every finally up process.

1.2.1. ASSUME: Process p_i crashes and is finally down.

PROVE: p_i is eventually suspected by all finally up processes.

PROOF: The progress property of the lockstep synchronous distributed system ensures that each finally up process will eventually receive a *pulsegrant*(r) with $r \geq r' + 2$, where r' is the last round number that was acknowledged by p_i by sending an "I'm in round r'" message. Thus each finally up process will eventually suspect p_i. □

1.2.2. ASSUME: Process p_i crashes and is finally down.

PROVE: p_i is no longer welcomed by any process.

PROOF: If a process is welcomed, it must have sent an "I recovered" message (line 30–34). As p_i is finally down, it will never recover and therefore never send an "I recovered" message. Thus p_i will no longer be welcomed. □

1.2.3. Q.E.D.

Steps 1.2.1 and 1.2.2 show that a finally down process is eventually suspected by every finally up process and no longer welcomed by any process. Moreover the provided state information contains all messages that were sent by the crashed process to the suspecting process, because accuracy holds (as shown in step 1.1). □

1.3. The algorithm satisfies recovery completeness, i.e. if a process recovers after a crash in state s and is finally up, it will eventually be welcomed after a crash in state s and no longer suspected by every finally up process.

1.3.1. Assume: Process p_i recovers and is finally up.
Prove: Every finally up process eventually welcomes p_i.

Proof: When p_i recovers, it sends an "I recovered" message to each other process. Every finally up process will receive this message (due to the validity property of the lockstep synchronous distributed system) and welcome the recovered process. □

1.3.2. Assume: Process p_i recovers and is finally up.
Prove: p_i is no longer suspected by any process.

Proof: As p_i is finally up, it sends an "I'm in round r" message in every round. Thus no process will ever suspect p_i. □

1.3.3. Q.E.D.

Steps 1.3.1 and 1.3.2 show that a finally up process is eventually welcomed by every finally up process and no longer suspected. Moreover the provided state information contains all messages that were sent by the crashed process to the suspecting process, because accuracy holds (as shown in step 1.1). □

1.4. The algorithm satisfies unstable completeness, i.e. every unstable process will be suspected and welcomed an infinite number of times by every finally up process.

Proof: An unstable process recovers an infinite number of times. Thus it will be suspected by each process an infinite number of times, because it sends an infinite number of "I recovered" messages. □

1.5. The algorithm satisfies integrity, i.e. every process is suspected at most once for every crash.

Proof: The algorithm keeps track of the current suspicion status of each process in the variable *suspected*. Due to the check in line 18, each process is suspected at most once after a crash. Before being suspected the next time, each process must be welcomed first and thus will not be suspected until its next crash. □

1.6. The algorithm satisfies suspect validity, i.e. if a process is suspected, it either was never suspected before, or it was welcomed after the last time it was suspected.

Proof: The algorithm keeps track of the current suspicion status of each process in the variable *suspected*. Due to the check in line 18, no process can be suspected twice without being welcomed in between (lines 34–35). □

1.7. The algorithm satisfies welcome validity, i.e. each process is welcomed at most once after each time being suspected.

Proof: A process is only welcomed when it sends an "I recovered" message (lines 30–35). This message is only sent once per recovery. Thus the no duplication property of the lockstep synchronous distributed system ensures that each process is welcomed only once per recovery. Furthermore the algorithm ensures in lines 31–32 that a process is suspected for a crash before being welcomed after that crash. □

1.8. Q.E.D.

Follows from steps 1.1–1.7 of the algorithm. □

2. Q.E.D.

Follows from step 1. □