The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment

Etienne Mauffret, Élise Jeanneau, Luciana Arantes, Pierre Sens

The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment

Etienne Mauffret[1], Denis Jeanneau[2], Luciana Arantes[2], and Pierre Sens[2]

[1]*LISTIC/Université Savoie Mont Blanc, France*
[2]*Sorbonne Université, CNRS, INRIA, LIP6, France*

Abstract

Mutual exclusion is one of the fundamental problems in distributed computing but existing mutual exclusion algorithms are unadapted to the dynamics and lack of membership knowledge of current distributed systems (e.g., mobile ad-hoc networks, peer-to-peer systems, etc.). Additionally, in order to circumvent the impossibility of solving mutual exclusion in asynchronous message passing systems where processes can crash, some solutions include the use of $(\mathcal{T}+\Sigma^l)$ [3], which is the weakest failure detector to solve mutual exclusion in known static distributed systems. In this paper, we define a new failure detector $\mathcal{T}\Sigma^{lr}$ which is equivalent to $(\mathcal{T}+\Sigma^l)$ in known static systems, and prove that $\mathcal{T}\Sigma^{lr}$ is the weakest failure detector to solve mutual exclusion in unknown dynamic systems with partial memory losses. We consider that crashed processes may recover.

1 Introduction

Distributed algorithms are traditionally conceived for message-passing distributed environments which are static and whose membership is known. However, new environments such as mobile ad-hoc wireless network (MANET) or wireless sensor network (WSN), peer-to-peer networks, and opportunist grids or clouds provide access to services or information regardless of node location, mobility pattern, or global view of the system. These new systems are dynamic, which means that the communication graph evolves over time, processes might join or leave the system, or crash and recover during the run. Additionally, they are unknown, i.e., processes do not initially know which other processes belong to the network, and only discover them during the run. Therefore, distributed algorithms that run on top of these new systems can not use prior distributed models for static known systems.

The mutual exclusion problem, introduced by Dijkstra in [9], is a fundamental problem in distributed computing requiring that their processes get exclusive access to one or more shared resources by executing a segment of code called critical section (CS). It specifies that, at any time, each process is either in the try, critical, exit or remainder sections. Processes cycle through these sections in order. Two processes cannot be in the critical section at the same time (*safety property*), and if a process is in the try section, then at some time later some process is in the critical section (*liveness property*).

Several mutual exclusion algorithms which tolerate process crash failures in the context of known static distributed systems have been proposed in the literature [14] [16] [1]. However, these works do not consider dynamic networks, or that a crashed process can recover. Furthermore, mutual exclusion algorithms that tolerate crash-recovery processes were mostly defined in the shared memory model, such as [12], [11], and [13], where shared variables are stored in non-volatile memory. One crash-recovery mutual exclusion algorithm for message-passing systems of which we are aware was proposed in [6] but its recovery solution works provided that failures do not occur in adjacent connected processes. Hence, the conception of mutual exclusion in unknown dynamic distributed systems where crashed processes can recover presents great challenges.

A definition of recoverable mutual exclusion (RME) for systems with crash-recovery was presented in [12] and further studied in [11] and [13]. A main change with regard to previous definitions of fault-tolerant mutual exclusion is the *critical section re-entry* property, which specifies that if a process p crashes while in the critical section and later recovers, then no other process may enter the critical section until p re-enters it after its recovery. Intuitively, this means that the lock on the critical section is not released in the case of a temporary crash.

In this paper we consider RME on top of a message passing model, where each process has access to a volatile memory of unbounded size, which is lost after a crash and recovery, and a non-volatile memory (stable storage) of bounded size. We denote this model the *partial memory loss* model.

Failure detectors were introduced in [5] as a way to circumvent the impossibility to solve consensus in crash-prone asynchronous systems ([10]). In [8], the $\mathcal{T}$ failure detector was shown to be the weakest failure detector to solve fault-tolerant mutual exclusion in message passing systems with a majority of correct processes. Then, in [3], the $(\mathcal{T}+\Sigma^l)$ failure detector was shown to be the weakest failure detector to solve the same problem with no assumption on the number of process failures.

Both of these results are restricted to known, static systems without recovery. Our paper aims to extend these results to unknown systems where crashed processes can recover, with partial memory loss.

$(\mathcal{T}+\Sigma^l)$ is the sum of two failure detectors. It provides two outputs, one of which verifies the properties of $\mathcal{T}$, and another one which verifies the properties of Σ^l. We call $\mathcal{T}\Sigma^l$ the detector which provides a single output verifying the properties of both $\mathcal{T}$ and Σ^l.

The contributions of our paper are as follows:

- A proof that $(\mathcal{T}+\Sigma^l)$ is equivalent to $\mathcal{T}\Sigma^l$;
- The definition of the $\mathcal{T}\Sigma^{lr}$ failure detector, which is equivalent to $\mathcal{T}\Sigma^l$ in known, static systems without recovery, and is the weakest failure detector to solve RME in our model;
- A RME algorithm that runs on top of the proposed model using the $\mathcal{T}\Sigma^{lr}$ failure detector and which tolerates crashes and recovery of processes, thus proving that $\mathcal{T}\Sigma^{lr}$ is sufficient to solve RME in our model;
- A reduction algorithm proving the necessity of $\mathcal{T}\Sigma^{lr}$ to solve RME in our model.

The rest of the paper is organized as follows: Section 2 presents our distributed system model. Section 3 provides failure detector definitions and proves the equivalence between $(\mathcal{T}+\Sigma^l)$ and $\mathcal{T}\Sigma^l$. Section 4 gives an algorithm solving mutual exclusion using $\mathcal{T}\Sigma^l$. Finally, Section 5 proves that $\mathcal{T}\Sigma^l$ is necessary to solve mutual exclusion in an unknown dynamic distributed system.

2 Model and Problem Definition

This section presents the distributed system model used throughout the rest of the paper and the definition of the Recoverable Mutual Exclusion (RME) problem.

2.1 System Model

The system is composed of a finite set of processes, denoted Π. Each process is uniquely identified. Additionally, *processes are asynchronous* (there is no bound on the relative speed of processes). They communicate by sending each other messages with a point-to-point SEND/RECEIVE primitive.

Communications are asynchronous (there is no bound on message transfer delay).

2.2 Failure Model

A process can *crash* (stop executing) during the run, and may *recover* from the crash, or not.

Each process has access to both a volatile memory and a stable storage of bounded size. After a crash and recovery, the variables in volatile memory are reset to their initial default values. As each process has access to stable storage, we say that this model deals with partial memory loss. In the rest of the paper, the names of variables in stable storage is underlined.

A process is said to be *alive* at time t if it never stopped executing before t or if it recovered since the last time it stopped executing. A process which is not alive at time t is said to be *crashed* at time t.

In the traditional crash failure model, processes are grouped into *faulty* processes, which eventually crash, and *correct* processes, which never crash. However, in a

crash-recovery model, in any run, we consider three types of processes [2]:

1) *Eventually up* processes, which stop crashing after some time and remain alive forever. This type also includes processes that never crash (*always up*).

2) *Eventually down* processes, which eventually crash and never recover. This type also includes processes that crashed immediately at the start of the run and never recovered (*always down*).

3) *Unstable* processes, which crash and recover infinitely often. We assume that, infinitely often, each unstable process manages to stay alive long enough to at least send a message to each other process of which it is aware.

2.3 Connectivity Model

The system is *dynamic* in the sense that the edges in the communication graph can appear and disappear during the run. In other words, at any given time instant, each edge in the graph might or might not be available. Without any further assumption, a system in which no edge is ever available would fit this model. Since nothing can be computed in such a system, additional assumptions are needed. Therefore, we assume that the following properties are verified:

- **Dynamic connectivity**: Every message sent by a process that is not *eventually down* to a process that is not *eventually down* is received at least once.
- **Uniqueness of reception**: Every message sent is received at most once.
- **First in, first out**: If process p sends a message m_1 to q and then sends m_2 to q, if q receives m_2 then it received m_1 first.

These properties imply not only that channels are reliable, but also that each pair of processes that are not *eventually down* is connected infinitely often by a path over time. This means that when a process p sends a message to process q, then there is a path from p to q such that at some point in the future, every edge on this path will be available in the correct order, and sufficiently long for the message to cross the edge. Note that it is not necessary that all the edges on the path to be available at the same time, and the path that a pair of processes uses to communicate is not required to be the same every time. This connectivity assumption is referred to as a Time-Varying Graph of class $\mathcal{C}_5$ in [4].

Our algorithms assume that the underlying SEND/RECEIVE implementation handles message forwarding and, therefore, behaves the same way that it would in a complete communication graph with reliable channels.

2.4 Knowledge Model

The system is *unknown*, i.e., processes initially have no information on system membership or the number of processes of the system, and are only aware of their own identity. The identities of other processes can only be learned through exchanging messages. More practically, each process p has access to a local variable $\underline{known}_p$ (in stable storage) that initially contains only p. Eventually, $\underline{known}_p$ contains the set of all processes that are not eventually down. For the sake of simplicity, our algorithms do not attempt to define the $\underline{known}_p$ variable and simply assume that an underlying discovery algorithm eventually fills it with the necessary process identities. This is not a strong assumption, since the *dynamic connectivity* property ensures that all processes will be able to communicate (and therefore learn of each other's existence) infinitely often.

2.5 Problem Definition

We consider the Recoverable Mutual Exclusion (RME) problem, which we define in our model as follows. At any point in time, a process either (1) does not try to access the critical section and is in the remainder section, (2) attempts to access the critical section through the try section, (3) is in critical section (CS) or (4) has recently left the critical section and is in the exit section. We consider that every user is well-formed, that is that a user will go through the remainder, try, critical and exit sections in the correct order. In case of a crash and recovery, a well-formed user will restart in the critical section if it was in the critical section when it crashed, and will restart in the remainder section, otherwise (the *critical section re-entry* property of [11]).

A fault-tolerant mutual exclusion algorithm must provide a TRY SECTION and an EXIT SECTION procedures

such that the following properties are satisfied:

Safety: Two distinct alive processes p and q can not be in CS at the same time.

Liveness: If an eventually up process p stopped crashing and is in the try section, then at some time later some process that is not eventually down is in CS.

Additionally, we consider the following fairness property:

Starvation Freedom: If no process stays in its critical section forever, then every eventually up process that stopped crashing and reaches its try section will eventually enter its CS.

Note that stable storage is necessary to solve the problem. Indeed, if a process p is in the critical section when all processes simultaneously crash, without stable storage there is no way for p to re-enter the critical section after recovery since no process remembers that p was in the critical section in the first place.

3 Failure Detectors

Failure detectors were introduced by Chandra and Toueg in [5] as a way to circumvent the impossibility to solve consensus in crash-prone asynchronous systems [10]. They are distributed oracles which provide unreliable information on process crashes. The information is unreliable in the sense that correct processes might be falsely suspected of having crashed, and faulty processes might still be trusted after they crashed. Different classes of failure detectors provide different properties on the reliability of the information provided to the processes.

Failure detectors are used as an abstraction of the system model assumptions.

A failure detector $\mathcal{D}_1$ is said to be *weaker* than $\mathcal{D}_2$ if there exists a distributed algorithm that can implement $\mathcal{D}_1$ using the information on failures provided by $\mathcal{D}_2$. Intuitively, this means that the computing power provided to the system by $\mathcal{D}_2$ is stronger than the computing power provided by $\mathcal{D}_1$. A failure detector that is sufficient to solve a given problem while being weaker than every other failure detector that can solve it, is said to be the weakest failure detector to solve that problem. It follows that the weakest failure detector to solve a problem can be implemented in any system in which the problem can be solved.

3.1 Failure Detectors for Mutual Exclusion

In [8], Delporte-Gallet et al. introduced the trusting failure detector $\mathcal{T}$ and proved that it is the weakest failure detector to solve fault-tolerant mutual exclusion in a system with a majority of correct processes. $\mathcal{T}$ provides each process p with a list of trusted processes, denoted tr_p. We denote tr_p^t the value of tr_p at time t. We say that process p trusts process q at time t if $q \in tr_p^t$ and that p suspects q at time t if $q \notin tr_p^t$. The following properties must be verified.

- **Eventually strong accuracy**: Every correct process p is eventually trusted forever by every correct process q, that is $\exists t : \forall t' > t, p \in qr_q^{t'}$
- **Strong completeness**: Every faulty process p is eventually suspected forever by every correct process q, that is $\exists t : \forall t' > t, p \notin qr_q^{t'}$
- **Trusting accuracy**: For any process p, if there exist times t and $t' > t$ such that $q \in tq_p^t$ and $q \notin tq_p^{t'}$, then q is faulty.

Bhatt et al. introduce in [3] the Σ^l quorum failure detector. Σ^l is a variant of the Σ quorum failure detector [7] adapted for the mutual exclusion problem. It provides each process p with a quorum of process identities, denoted qr_p. Similarly to tr_p, we denote qr_p^t the value of qr_p at time t. Σ^l verifies the following properties.

- **Strong completeness**: Every faulty process p is eventually suspected forever by every correct process q, that is $\exists t : \forall t' > t, p \notin qr_q^{t'}$
- **Live pairs intersection**: If two processes p and q are both alive at time t, then for any couple of time instants $t_1 \leq t$ and $t_2 \leq t$, $qr_p^{t_1} \cap qr_q^{t_2} \neq \emptyset$.

Bhatt et al. show in [3] that $\mathcal{T}$ and Σ^l used together, denoted $(\mathcal{T}+\Sigma^l)$, constitute the weakest failure detector to solve mutual exclusion with any number of process failures in static, known systems.

$(\mathcal{T}+\Sigma^l)$ provides two outputs, tr_p and qr_p, with tr_p verifying the properties of $\mathcal{T}$ and qr_p verifying the properties of Σ^l. We call $\mathcal{T}\Sigma^l$ the failure detector that provides a single output tq_p verifying the properties of both $\mathcal{T}$ and Σ^l.

Theorem 1. *$(\mathcal{T}+\Sigma^l)$ is equivalent to $\mathcal{T}\Sigma^l$.*

Proof. $(\mathcal{T}+\Sigma^l)$ can be implemented using the output of $\mathcal{T}\Sigma^l$: it suffices to always return the value of tq_p as both tr_p and qr_p. Therefore, $(\mathcal{T}+\Sigma^l)$ is weaker than $\mathcal{T}\Sigma^l$.

$(\mathcal{T}+\Sigma^l)$ is sufficient to solve FTME, as shown in [3]. In Section 5 of this paper, we will prove that it is possible to use RME to implement the $\mathcal{T}\Sigma^{lr}$ failure detector (defined in Section 3.2). The same reasoning and algorithms that we use in Section 5 can be used to show that it is possible to use FTME to implement $\mathcal{T}\Sigma^l$. It follows that $(\mathcal{T}+\Sigma^l)$ is sufficient to implement $\mathcal{T}\Sigma^l$, and as a result, $\mathcal{T}\Sigma^l$ is weaker than $(\mathcal{T}+\Sigma^l)$. □

3.2 The $\mathcal{T}\Sigma^{lr}$ Failure Detector

The existing definition of $\mathcal{T}\Sigma^l$ is for static, known networks, and therefore we need to provide new definitions, suitable for unknown dynamic networks.

In an unknown system, the lack of initial information renders difficult the implementation of some failure detector properties which must apply from the start of the run, in particular the intersection property. To circumvent this problem, we make use of the $\perp$ concept introduced in [15].

Additionally, the traditional properties of $\mathcal{T}\Sigma^l$ are expressed in terms of *correct* and *faulty* processes. $\mathcal{T}\Sigma^{lr}$ was rewritten using the concepts of *eventually up* and *eventually down* processes instead.

The $\mathcal{T}\Sigma^{lr}$ failure detector provides each process p with a set of trusted process identities, denoted tq_p, and a flag denoted rdy_p. rdy_p is initially set to $\perp$ and changes to $\top$ once the failure detector has gathered enough information to verify the live pairs intersection property. We denote tq_p^t the value of tq_p at time t, and rdy_p^t the value of rdy_p at time t. We say that process p trusts process q at time t if $q \in tq_p^t$, that p suspects q at time t if $q \notin tq_p^t$, and that process p is ready at time t if $rdy_p^t = \top$. The following properties must be verified.

- **Eventually strong accuracy**: Every *eventually up* process p is eventually trusted forever by every process that is not *eventually down*.
- **Strong completeness**: Every *eventually down* process p is eventually suspected forever by every process that is not *eventually down*.
- **Trusting accuracy**: For any process p, if there exist times t and $t' > t$ such that $q \in tq_p^t$ and $q \notin tq_p^{t'}$, then q is *eventually down* and will never be alive after t'.
- **Quorum readiness**: Every *eventually up* process is eventually ready forever.
- **Live pairs intersection**: If two processes p and q are both alive at time t, then for any couple of time instants $t_1 \leq t$ and $t_2 \leq t$, $(rdy_p^{t_1} = \top \wedge rdy_q^{t_2} = \top) \implies tq_p^{t_1} \cap tq_q^{t_2} \neq \emptyset$.

The eventually strong accuracy, strong completeness and trusting accuracy properties are the original properties of $\mathcal{T}$, adapted for a crash-recovery model. We call these properties the trusting properties of $\mathcal{T}\Sigma^{lr}$.

Similarly, the strong completeness and live pairs intersection properties are the original properties of Σ^l, adapted for our model. The new quorum readiness property, along with the rdy_p output variable, was added to deal with the lack of initial information in an unknown system. We call these properties the quorum properties of $\mathcal{T}\Sigma^{lr}$.

Note that the strong completeness is both a trusting property and a quorum property, since both $\mathcal{T}$ and Σ^l make use of this same property.

Both trusting and quorum properties apply to the same set tq_p, which is different from preexisting definitions in which $\mathcal{T}$ and Σ^l are two separate oracles with separate outputs. In Section 5, we will prove that this combined version of the detector is necessary to solve RME.

Note that in a static, known system with reliable channels and prone to crash failures without recovery, $\mathcal{T}\Sigma^{lr}$ is equivalent to $\mathcal{T}\Sigma^l$, and therefore $(\mathcal{T}+\Sigma^l)$.

4 Sufficiency of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion

In this section we introduce Algorithm 1 and prove that it solves the RME in any unknown dynamic environment enriched with the $\mathcal{T}\Sigma^{lr}$ failure detector.

Algorithm 1 Solving RME with $\mathcal{T}\Sigma^{lr}$: code for process p

1: **procedure** TRY SECTION
2: **wait for** $recovering_p = false$
3: $req_p \leftarrow true$
4: $round_p \leftarrow round_p + 1$; $grants_p \leftarrow \{p\}$
5: **for** $\forall q \in tq_p$ **do** SEND(REQUEST, $round_p, q$)
6: $requests_p \leftarrow requests_p \cup \{(round_p, p)\}$
7: CHECK REQUESTS()
8: **wait for** $gid_p = p$ **and** $rdy_p = \top$ **and** $tq_p \subseteq grants_p$
9: $\underline{crit}_p \leftarrow true$; $req_p \leftarrow false$
10: **procedure** EXIT SECTION
11: **wait for** $recovering_p = false$
12: $\underline{crit}_p \leftarrow false$
13: **for** $\forall q \in grants_p \setminus \{p\}$ **do** SEND(DONE, q)
14: $grants_p \leftarrow \{p\}$; $requests_p \leftarrow requests_p \setminus \{(*, p)\}$
15: CHECK REQUESTS()
16: **procedure** CHECK REQUESTS
17: **if** ($gid_p = -1$ **or** $gid_p = p$) **and** $requests_p \neq \emptyset$ **and** $\underline{crit}_p = false$ **and** $recovering_p = false$ **then**
18: $(grnd_p, gid_p) \leftarrow$ HIGHEST($requests_p$)
19: **if** $gid_p \neq p$ **then** SEND(GRANT, gid_p)
20: **for** $\forall q \in grants_p \setminus \{p\}$ **do**
21: $grants_p \leftarrow grants_p \setminus \{q\}$
22: SEND(REJECT, q)
23: **procedure** RECONNECTION
24: $recovering_p \leftarrow true$
25: $update_p \leftarrow tq_p$
26: **for** $\forall q \in update_p$ **do**
27: SEND(COMEBACK, $\underline{crit}_p, q$)
28: **wait for** $update_p = \emptyset$
29: $recovering_p \leftarrow false$
30: CHECK REQUESTS()
31: **when** q added to tq_p
32: **if** $req_p = true$ **then** SEND(REQUEST, $round_p, q$)
33: **when** q removed from tq_p
34: $grants_p \leftarrow grants_p \setminus \{q\}$
35: $requests_p \leftarrow requests_p \setminus \{(*, q)\}$
36: $update_p \leftarrow update_p \setminus \{q\}$
37: **if** $gid_p = q$ **then**
38: $(gid_p, grnd_p) \leftarrow (-1, -1)$
39: CHECK REQUESTS()
40: **upon reception of** REQUEST ($round$) **from** src **do**
41: $requests_p \leftarrow requests_p \cup \{(round, src)\}$
42: $last\_round_p[src] \leftarrow round$
43: CHECK REQUESTS()
44: **upon reception of** GRANT () **from** src **do**
45: **if** $gid_p \neq -1$ **and** $gid_p \neq p$ **then**
46: SEND(REJECT, src)
47: **else if** $recovering_p = false$ **then**
48: $grants_p \leftarrow grants_p \cup \{src\}$
49: **upon reception of** DONE () **from** src **do**
50: $requests_p \leftarrow requests_p \setminus \{(*, src)\}$
51: $(gid_p, grnd_p) \leftarrow (-1, -1)$
52: CHECK REQUESTS()
53: **upon reception of** REJECT () **from** src **do**
54: $(gid_p, grnd_p) \leftarrow (-1, -1)$
55: CHECK REQUESTS()
56: **upon reception of** COMEBACK ($crit\_src$) **from** src **do**
57: $requests_p \leftarrow requests_p \setminus \{(*, src)\}$
58: **if** $crit\_src = false$ **and** $gid_p = src$ **then**
59: $(gid_p, grnd_p) \leftarrow (-1, -1)$
60: CHECK REQUESTS()
61: SEND(UPDATE, $gid_p = src, last\_round_p[src], src \in grants_p, round_p, req_p, src$)
62: **upon reception of** UPDATE ($grant\_p, last\_rnd, grant\_src, round, req$) **from** src **do**
63: $last\_round_p[src] \leftarrow round$
64: $round_p \leftarrow$MAX($round_p, last\_rnd$)
65: **if** $grant\_src = true$ **then** ▷ p previously granted src
66: $(gid_p, grnd_p) \leftarrow (src, round)$
67: **if** $grant\_p = true$ **then** ▷ src previously granted p
68: $grants_p \leftarrow grants_p \cup \{src\}$
69: **if** $req = true$ **then** ▷ src is requesting
70: $requests_p \leftarrow requests_p \cup \{(round, src)\}$
71: $update_p \leftarrow update_p \setminus \{src\}$

4.1 Algorithm Description

Let's consider p the sender (source) of a message. The following types of messages are used by Algorithm 1:

REQUEST: p has asked for permission to enter CS. The message contains the round number of the sender.

GRANT: p has granted permission to a requesting process to enter CS.

DONE: notifies other processes that p has just exited its CS.

REJECT: warns that p has already given its permission to another process different from itself, thus preventing deadlocks.

COMEBACK: notifies processes that p has just recovered from a crash.

UPDATE: p gives information to a recently recovered process q about p's requesting state, previously given permissions that p granted to q and vice-versa, and q's last round number of which p is aware.

In Algorithm 1, each request to enter the CS, issued by p, is tagged by a sequence round number.

Besides having access to the output, tq_p and rdy_p, of its local failure detector, process p also keeps the following local variables, initialized with the indicated value:

$\underline{crit}_p \leftarrow false$: a flag indicating that p is currently in CS. It is the only variable kept in stable storage. Thus, $\underline{crit}_p$ is not reinitialized after a crash and recovery.

$round_p \leftarrow 0$: the local round number of p, which is used to number its requests. It is also used to define the current priority of p to access the critical section.

$last\_round_p \leftarrow \emptyset$: a table associating each known process identity with its last known round number. It is used to restore the round number of other processes after they crash and recover.

$req_p \leftarrow false$: a flag indicating that p is currently in the try section.

$requests_p \leftarrow \emptyset$: the set of requests received by p which are pending. Each request is a couple $(round, pid)$.

$gid_p \leftarrow -1$: the identity of the last process to which p granted its permission, or -1 if p did not grant it. It indicates that p sent a GRANT message to gid_p, and that this permission was not canceled by the reception of a DONE or REJECT message yet.

$grnd_p \leftarrow -1$: the current round number of the process to which p granted its permission, or -1 if p did not grant it.

$grants_p \leftarrow \{p\}$: the set of processes from which p received a GRANT message.

$recovering_p \leftarrow false$: a flag indicating that p is currently attempting to rebuild its volatile memory after a crash. Calls to TRY SECTION and EXIT SECTION will be delayed while $recovering_p = false$.

$update_p \leftarrow \emptyset$: the set of processes from which p waits for an UPDATE message. This variable is only used during the recovery phase, i.e., while $recovering_p = true$.

All of these local variables, except for $\underline{crit}_p$, are stored in volatile memory. This means that after a crash and recovery, they are reinitialized to the above default value.

Requests are totally ordered by their priority, which is defined as follows: $priority(round_p, p) > priority(round_q, q) \Leftrightarrow round_p < round_q$ **or** $[round_p = round_q$ **and** $p < q]$. The HIGHEST function takes a list of requests and returns the couple $(round, id)$ of the request with the highest priority among the trusted processes according to tq_p.

The CHECK REQUESTS procedure is extensively used in Algorithm 1. Provided that process p did not already grant its permission to another process and is not in CS, CHECK REQUESTS compares the requests that p received so far by calling the HIGHEST function (line 18), and sends a GRANT message to the process with the highest priority (line 19). In case p received grants from other processes before granting its own permission, it will send REJECT messages to the processes in $grants_p$ in order to prevent a deadlock (lines 20 – 22).

Whenever process p wants to access the critical section, it executes the TRY SECTION: p increments its $round_p$, resets its $grants_p$ set (line 4), and then broadcasts a REQUEST to every process in tq_p (line 5). If p gets knowledge of a new process while it is still in the try section, the request will also be sent to this process (line 32). Process p adds its own request to its $requests_p$ before calling CHECK REQUESTS (lines 6 – 7), and finally waits for permissions from every process in tq_p (and its own permission, line 8) before entering CS.

Upon reception of a REQUEST message from process q (lines 40 – 43), process p updates its knowledge about q's round number and adds the new request to its $requests_p$ set. It then calls CHECK REQUESTS to decide if it should send a grant to the new requester.

When receiving a GRANT message from process q, if p already granted its permission to some other process

then it informs q by responding with a REJECT message to prevent deadlocks (line 46). Otherwise, if p is not in the recovery phase, it accepts q's permission by adding it to its $grants_p$ set.

Upon finishing the critical section and calling EXIT SECTION, p sends to all trusted processes a DONE message (line 13). Then, p resets its $grants_p$ set and cancels its request (line 50) before calling CHECK REQUESTS to grant its permission to the next process.

If p receives a DONE or REJECT message from process gid_p, it cancels the permission granted to gid_p (lines 51 and 54) and calls CHECK REQUESTS. In the case of a DONE message, the request from gid_p is also deleted from $requests_p$ (line 50), since gid_p is not requesting CS anymore. However, in the case of a REJECT, the request from gid_p is still valid and must be kept, even if it is not the highest priority request.

If p crashes and recovers, the RECONNECTION procedure will be called first. This procedure initiates the recovery phase (lines 24 – 29) by switching the $recovering_p$ flag to $true$, which will temporarily prevent the algorithm from going into the try or exit sections (lines 2 and 11) and from sending or accepting a grant (lines 17 and 47). During the recovery phase, p attempts to recover the information it lost during the crash by sending a COMEBACK message to every process in tq_p. Other processes will send UPDATE messages in response, which enables p to restore its $last\_round_p$, $round_p$, gid_p, $grnd_p$, and $requests_p$ variables (lines 63 – 71). The recovery phase ends when every process to which p sent a COMEBACK has either responded with an UPDATE message (line 71), or crashed (line 36). After recovering, p calls CHECK REQUESTS in order to choose a process to which it will grant its permission (line 30).

If p receives a COMEBACK message from a process q, it cancels any request previously received from q, since a process in recovery phase can only be in the remainder or critical section (by definition of a well-formed process). If q is in its remainder section ($\underline{crit}_p = false$), then p cancels any permission it might have granted to q previously (lines 58 – 60). Finally, p sends an UPDATE message to q.

Whenever p is informed by the failure detector that a process q is eventually down (lines 33 – 39), p deletes q from its $requests_p$, $grants_p$ and $update_p$ sets. If q was the process to which p granted permission, then p cancels the permission (line 38) and calls CHECK REQUESTS to grant its permission to another process, if appropriate.

4.2 Proof of correctness

We will prove, through the following claims, that any run of Algorithm 1 solves the RME problem.

Claim 1 (Safety)**.** *Two distinct alive processes p and q can not be in CS at the same time.*

In order to prove Claim 1 we need to pose the following lemmata.

Lemma 1 (Uniqueness of the permission)**.** *Let p, q_1, q_2 be three distinct alive processes. If $p \in grants_{q_1}$ at a time t then p cannot send a* GRANT *message to q_2 at time t.*

Proof. The only way that p can send a GRANT message to a process q is on line 19, after it selected q as its gid_p. Note that the definition of the HIGHEST function also implies that $q \in tq_p$ at the time when the GRANT message is sent.

Suppose that p has sent a GRANT message at time t_G to another process q_1 (and therefore at time t_G, $gid_p = q_1$).

Let us assume that there is a time $t > t_G$ such that $p \in grants_{q_1}$. Let us then suppose that p sends a GRANT message to another process q_2 at time t.

In order to send a GRANT message to q_2, p has to set gid_p to -1 or to p at some time $t' \in [t_G, t]$ (otherwise p cannot pass the test on line 17). This affectation can only be done in one of the following lines:

Line 38: then $q_1 \notin tq_p^{t'}$. Since $q_1 \in tq_p^{t_G}$, according to the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$, q_1 has crashed at some time before t' and will never recover. It is therefore impossible that $p \in grants_{q_1}$ at time t.

By resetting gid_p to -1 after a crash. If p crashed between t_G and t', then its gid_p got reset to -1. This also means that p entered the recovery phase (lines 24 – 29) at some time $t'' \in [t_G, t']$. Since $q_1 \in tq_p^{t_G}$, then according to the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$, either q_1 crashed before t'' and will never recover (which is a contradiction), or $q_1 \in tq_p^{t''}$. p will therefore send a COMEBACK message to q_1 on line 27, and q_1 will respond with a UPDATE message with the $grant\_src$ parameter set to $true$, which will cause p to set its gid_p back to q_1. Since p cannot have sent a GRANT message while in the recovery phase (because of the test on line 17), then p cannot send the GRANT to q_2 at time t which is a contradiction.

Line 59: then p received a COMEBACK message from q_1 at some time $t'' \in [t_G, t']$. This means that q_1 crashed and went into the recovery phase. p will respond with an update message to q_1. Since q_1 cannot leave the recovery phase until it receives p's update and because of the first in, first out property, then p's GRANT message to q_1 was received either (1) before q_1 crashed, in which case the GRANT was forgotten, or (2) during the recovery phase, in which case q_1 will ignore the GRANT because of the test on line 47. In both cases, $p \notin grants_{q_1}$ after t'', which is a contradiction.

Line 51 or 54: then p received a DONE or REJECT message from q_1 at time t'. There are two cases. If q_1 sent the DONE or REJECT message after receiving the GRANT, then q_1 removed p from $grants_{q_1}$ on line 14 (resp. line 21) and did not add it back in afterwards, which is a contradiction. Otherwise, q_1 sent the DONE or REJECT message before receiving p's GRANT. Since q_1 only sends DONE or REJECT messages to processes from which it previously received a GRANT, then p sent another GRANT message to q_1 before t_G. This means that p sent two consecutive GRANT messages to q_1 without receiving a DONE or REJECT message in between. The only way this could happen is if p set its gid_p to -1 or p between sending the two GRANT messages without receiving a DONE or REJECT, which is a contradiction since this proof eliminated every other way of doing that.

Hence, we can not have $p \in grants_{q_1}$ and p sending a GRANT message to q_2 at the same time, which conclude the proof of Lemma 1 . □

Lemma 2 (Self permission)**.** *Let p, q be two distinct alive processes. If $p \in grants_q$ then p can not enter CS.*

Proof. If $p \in grants_q$, then p sent a GRANT message to q and therefore set its gid_p to q. The reasoning of the proof for Lemma 1 can be used to show that p cannot change the value of its gid_p until q has removed p from its $grants_q$.

Since p is required to have its gid_p set to p in order to enter CS (line 8), then it is impossible for p to enter CS until after q removed p from $grants_q$. □

We can now prove the Claim 1 by contradiction.

Proof. Let p_1, p_2 be two alive, distinct processes. Let us suppose that p_1 enters CS at time t_1, and p_2 enters CS at time t_2. Let us suppose that neither process leaves CS until after the other process has entered it. According to the live pairs intersection property of $\mathcal{T}\Sigma^{lr}$, there is a process q such that $q \in tq_{p_1}^{t_1} \cap tq_{p_2}^{t_2}$. It follows from the wait condition on line 8 that $q \in grants_{p_1}$ at time t_1 and $q \in grants_{p_2}$ at time t_2. There are two cases:

First case: p_1, p_2 and q are all distinct. Therefore, q sent a GRANT message to p_1 before t_1 and a GRANT message to p_2 before t_2. Additionally, neither process removed q from their $grants$ set before entering CS. Without loss of generality, let us assume that q sent the GRANT message to p_1 first. There could be a run in which p_1 received the message immediately, and therefore added q to $grants_{p_1}$ before q sent the second GRANT to p_2. In this run, q sends a GRANT message to p_2 while $q \in grants_{p_1}$ at the same time, which is in contradiction with Lemma 1.

Second case: $q = p_1$ or $q = p_2$. Without loss of generality, let us assume that $q = p_1$. Since $q \in grants_{p_2}$ at time t_2, q sent a GRANT message to p_2 before t_2. Since it is impossible for q to send a GRANT message while in CS (because of the test on line 17), it follows that q sent the GRANT before entering CS. There could be a run in which p_2 received the GRANT immediately after it was sent, therefore adding q to $grants_{p_2}$ before q entered CS, which is in contradiction with Lemma 2. □

Claim 2 (Starvation freedom)**.** *If no process stays in its critical section forever, then every eventually up process that stopped crashing and reaches its try section will eventually enter its CS.*

To prove the Claim 2, we pose the following lemmata:

Lemma 3 (Deadlock-free)**.** *Assuming that no process stays in CS forever, if a process p, which does not have the highest priority among the requesting processes, receives at least one* GRANT *from another process q, p will eventually either crash forever or remove q from $grants_p$, and q will eventually either crash forever or set gid_q to -1.*

Proof. Let p be a process in its try section at time t. There exists a distinct process p_h which is also in its try section at time t and has the highest priority among requesting processes.

Let q be a process distinct from p that sends a GRANT message that p receives at time t. It follows that p sent a REQUEST message to q at some time $t_R < t$.

One of the following cases applies:

1) p is eventually down, and q is not. Then according to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, p will eventually be removed from tq_q and q will set gid_q to -1 on line 38.

2) q is eventually down, and p is not. Then according to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, q will eventually be removed from tq_p and p will remove q from $grants_p$ on line 34.

3) At time t, $gid_p \neq -1$ and $gid_p \neq p$. Then when p receives q's GRANT message, it will never add q to $grants_p$ and will send q a REJECT message instead (line 46). When q receives the REJECT message, it will set gid_q to -1 (line 54).

4) At time t, $gid_p = -1$. When p calls CHECK REQUESTS, it will pass the test one line 17 since $requests_p$ contains at least p's request, and $\underline{crit}_p$ and $recovering_p$ cannot be *true* while in CS. p will then set gid_p to something different from -1 on line 18.

It follows from the cases above that the only way Lemma 3 could be false is if neither p nor q are eventually down, and $gid_p = p$ at time t. Since p is not eventually down, then p will eventually receive p_h's request at some time $t' > t$. Then one of the following cases applies:

1) During $[t_R, t']$, p does not crash, receives GRANT messages from every process in tq_p, and rdy_p is set to $\top$. Then p will end the wait on line 8 and enter CS. When p leaves CS, it will remove q from $grants_p$ on line 14 and send a DONE message to q on line 13. When q receives the DONE message, it will set gid_q to -1 on line 51.

2) During $[t_R, t']$, p does not crash and does not receive enough GRANT messages to enter CS (or rdy_p stays equal to $\perp$). Then at time t' when p receives p_h's request, it will call CHECK REQUESTS on line 43. p will pass the test on line 17 and, since p_h is the requesting process with the highest priority, p will set gid_p to p_h. It will then remove q from $grants_p$ on line 21 and send a REJECT message to q on line 22. When q receives the REJECT message, it will set gid_q to -1 on line 54.

3) During $[t_R, t']$, p crashes before receiving enough GRANT messages to enter CS. When p recovers, its $grants_p$ set is reinitialized and does not contain q. Since q was previously in tq_p and q is not eventually down, it follows from the trusting accuracy property of $\mathcal{T}\Sigma^{lr}$ that q is still in tq_p after p recovers. p will therefore send a COMEBACK message to q on line 27 with the *crit_src* parameter set to $false$. When q receives the COMEBACK message, it will set gid_q to -1 on line 59. Note that because of the first in, first out property, q will necessarily receive p's request before the COMEBACK message. Additionally, p will receive q's GRANT message before q's UPDATE message, and will ignore the grant because of the test on line 47. □

Lemma 4 (Decreasing priority)**.** *Assuming that no process stays in the CS forever, if an unstable process p is in the try section infinitely often, then the value of $round_p$ increases infinitely often (and therefore, p's priority decreases infinitely often).*

Proof. Let p be an unstable process that is in the try section infinitely often. By definition, p also crashes infinitely often. Let q be any eventually up process. According to the eventually strong accuracy property of $\mathcal{T}\Sigma^{lr}$, p will eventually trust q forever.

Let t_0 be a time after which every eventually down process crashed permanently, every eventually up process stopped crashing, and p started trusting q. According to the strong completeness property of $\mathcal{T}\Sigma^{lr}$, there is a time $t_1 \geq t_0$ such that $\forall t > t_1$, tq_p^t does not contain any eventually down process. Let $t_2 > t_1$ be the first time after t_1 that p crashes, and let $t_3 > t_2$ be the first time after t_2 that p enters the try section.

Every request sent by p after t_3 is sent only to processes that are not eventually down, including q. According to the dynamic connectivity property, q will receive every request sent by p after t_3. Every time that p crashes after t_3, p will send a COMEBACK message to q. Because of the first in, first out property, q will receive p's last request before receiving the COMEBACK message, and therefore when q receives the COMEBACK its $last\_round_q[p]$ will be up to date with q's latest $round_p$ value from before the crash. q will then respond with an UPDATE message, and p will update its $round_p$ value on line 63 before leaving the recovery phase. As a result, crashes after t_3 do not reduce or reset p's $round_p$ value.

At any time $t > t_3$, there are three possibilities:

1) p is in the exit or remainder section at time t. By assumption, p will eventually enter the try section, and therefore increase its $round_p$ value on line 4.

2) p is in the CS at time t. Since by assumption no process stays in the section forever, p will eventually leave CS and the case above applies.

3) p is in the try section at time t. Eventually, p will either enter CS (and the case above applies), or p will crash before entering the CS and therefore it will be in the remainder section after recovery (and the first case applies).

In all cases, there is a time $t' > t$ such that $round_p$ increases at time t'. □

Lemma 5 (Highest priority starvation freedom)**.** *Let t be a time after all eventually up processes stopped crashing. Assuming that no process stays in CS forever, if an eventually up process p is in the try section and has the highest priority among requesting eventually up processes at time t, then eventually p enters CS.*

Proof. Let p be an eventually up process that is in the try section with the highest priority among requesting eventually up processes at time t. By contradiction, let us assume that p never enters CS after t. It follows that p will never leave the try section, since it will neither crash nor enter CS. Therefore, p will never re-enter the try section and increase its $round_p$ value on line 4. It follows that p's priority will never change after t.

Let q_1 be any unstable process. According to Lemma 4, q_1 will either eventually stop entering the try section (in which case its priority becomes irrelevant), or q_1's priority will be reduced infinitely often, in which case p's priority will eventually be higher than q_1's. As a result, there is a time $t' \geq t$ after which p has the highest priority of all requesting processes in the system.

If $gid_p = q_2$ with q_2 distinct from q after t', then according to Lemma 3, eventually p will set its gid_p to -1 and then call CHECK REQUESTS. p will then set itself as gid_p on line 18 and will never change gid_p again.

According to the dynamic connectivity property, eventually every process in tq_p will have received p's request. Let q_3 be any process that received p's request. If $gid_{q_3} \neq -1$ and $gid_{q_3} \neq q_3$, then after t', according to Lemma 3, q_3 will eventually set gid_{q_3} to -1. When gid_{q_3} is equal to -1 or q_3 after t', then q_3 will set it to p on line 18 and send a GRANT message to p on line 19. As a result, p will receive a GRANT message from every process in tq_p.

Since p is eventually up, according to the quorum readiness property of $\mathcal{T}\Sigma^{lr}$, the eventually $rdy_p = \top$.

Finally, p will pass the wait condition on line 8 and enter CS, which is a contradiction. □

We can now prove Claim 2.

Proof. Let p be an eventually up process that stopped crashing and is in its try section at time t. By contradiction, let us assume that p never enters CS after t. It follows that p will never leave the try section, since it will neither crash nor enter CS. Therefore, p will never re-enter the try section and increase its $round_p$ value on line 4. It follows that p's priority will never change after t, and that every requesting unstable process will eventually have a lower priority than p.

Let Q be the set of all requesting eventually up processes with higher priority than p. Let q be the process in Q with the highest priority. It follows from Lemma 5 that eventually, q will enter CS. After q leaves CS, it will either (1) stop requesting forever (and therefore leave Q) or (2) enter the try section again and therefore decrease its priority. By induction, q will eventually not have the highest priority amongst requesting processes anymore, and another process in Q will take its place. As a result, eventually Q will become empty since every process in it will either stop requesting or increase its priority infinitely often.

Finally, p will become the requesting eventually up process with the highest priority, and according to Lemma 5, will enter CS, which is a contradiction. □

Claim 3 (Liveness)**.** *If an eventually up process p stopped crashing and is in the try section, then at some time later some process that is not eventually down is in CS.*

Proof. Let p be an eventually up process that stopped crashing and is in the try section. There are two possibilities:

- Some process eventually stays in CS forever. In this case, liveness is ensured.
- Otherwise, according to Claim 2, p will eventually enter CS, thus ensuring liveness.

□

From Claim 1 and Claim 3 we can deduce the following theorem:

Theorem 2 (Correctness)**.** *The Algorithm 1 solves the RME using $\mathcal{T}\Sigma^{lr}$ in any unknown dynamic environment.*

Corollary 1 (Sufficiency)**.** *The $\mathcal{T}\Sigma^{lr}$ failure detector is sufficient to solve the RME in any unknown dynamic environment with partial memory loss.*

5 Necessity of $\mathcal{T}\Sigma^{lr}$ to solve Fault-Tolerant Mutual Exclusion

In this section we prove that the $\mathcal{T}\Sigma^{lr}$ failure detector is necessary to solve the RME problem in any unknown dynamic system with partial memory loss. For this purpose, we assume that there is an unknown dynamic system model $\mathcal{M}_{\text{RME}}$ with partial memory loss, in which RME can be solved with some algorithm $\mathcal{A}_{\text{RME}}$. We will then show that the properties of $\mathcal{T}\Sigma^{lr}$ can be implemented in $\mathcal{M}_{\text{RME}}$.

Although the purpose of this section is to show that $\mathcal{T}\Sigma^{lr}$ can be implemented with RME in $\mathcal{M}_{\text{RME}}$, the same arguments and algorithms used here can also be used to show that $\mathcal{T}\Sigma^{l}$ can be implemented with fault-tolerant mutual exclusion in a static, known system without recovery. As a result, this section is also a part of the proof for Theorem 1.

The following proof is inspired by the proofs for the necessity of $\mathcal{T}$ and Σ^l in [8] and [3], respectively. The main additional challenge is to merge the two proofs, since both trusting and quorum properties must apply for a same set tq_p.

The proof uses two algorithms, both of which share the following local variables:

$\underline{trust}_p \leftarrow \{p\}$ is the set of all processes that process p has heard of and that it does not suspect. This variable is in stable storage.

$start_p \leftarrow false$ is a flag used to delay the start of the RME algorithm.

Firstly, we introduce the algorithm $\mathcal{B}_{\text{RME}}$. $\mathcal{B}_{\text{RME}}$ has exactly the same code as $\mathcal{A}_{\text{RME}}$, except that every call to the SEND primitive is replaced by a call to $\mathcal{B}_{\text{RME}}$ _SEND, as defined in Algorithm 2.

Algorithm 2 serves two purposes: (1) by using $\underline{trust}_p$, it enables p to keep track of which processes it heard of while trying to access CS; (2) by using $start_p$, it enables p to delay the start of the RME algorithm.

Lemma 6. *Provided that each eventually up process p eventually sets $start_p$ to true, Algorithm $\mathcal{B}_{\text{RME}}$ solves the RME problem in $\mathcal{M}_{\text{RME}}$.*

Algorithm 2 Modified SEND primitive for $\mathcal{B}_{\text{RME}}$

1: **procedure** $\mathcal{B}_{\text{RME}}$ _SEND$(msg, dest)$
2: **wait for** $start_p = true$
3: SEND$(msg, \underline{trust}_p, dest)$
4: **upon reception of** $(msg, trust\_src)$ **from** src **do**
5: **wait for** $start_p = true$
6: $\underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$
7: $\mathcal{B}_{\text{RME}}$ _DELIVER(msg)

Proof. The only difference between $\mathcal{A}_{\text{RME}}$ and $\mathcal{B}_{\text{RME}}$ that could prevent $\mathcal{B}_{\text{RME}}$ from solving RME is the wait on lines 2 and 5. A process that never sets $start_p$ to $true$ cannot participate in the algorithm. By assumption, this is only a problem for processes that are not eventually up. If a process never sets $start_p$ to $true$, then for the purpose of $\mathcal{B}_{\text{RME}}$, that process behaves exactly as an always down process would behave in a run of $\mathcal{A}_{\text{RME}}$. □

We can now introduce Algorithm 3, which makes use of $\mathcal{A}_{\text{RME}}$ and $\mathcal{B}_{\text{RME}}$ to implement the properties of $\mathcal{T}\Sigma^{lr}$.

In addition to $\underline{trust}_p$ and $start_p$, Algorithm 3 uses following local variables:

$\underline{known}_p \leftarrow \{p\}$: as discussed in Section 2, $\underline{known}_p$ represents the knowledge that p has of other processes in the system. The algorithm does not show how $\underline{known}_p$ is kept up to date, but simply expects that $\underline{known}_p$ will eventually contain the process identities of (at least) all eventually up processes.

$\underline{crash}_p \leftarrow \emptyset$: the set of all processes that p is certain have crashed forever. Note that this variable is in stable storage.

$tq_p \leftarrow \emptyset$: the output of the $\mathcal{T}\Sigma^{lr}$ failure detector, which verifies the trusting and quorum properties.

$rdy_p \leftarrow \perp$: the other output variable of $\mathcal{T}\Sigma^{lr}$, which verifies the quorum properties.

$\underline{waitlist}_p \leftarrow \emptyset$: the set of processes to which p must grant permission for CS. This is used to ensure starvation freedom. Note that this variable is in stable storage.

$donelist_p \leftarrow \emptyset$: the set of processes to which p already granted permission for CS. It prevents p from always being passed over for CS access.

Algorithm 3 initially starts two tasks in parallel: TASK 1 and TASK 2. Later on, whenever process p gets knowledge of a process q, it starts a new task for q (denoted TASK $3 + q$).

Algorithm 3 Reduction Algorithm $T_{\mathcal{A}_{\text{RME}} \to \mathcal{TS}^{tr}}$: code for process p

1: **procedure** TASK 1
2: $\quad \mathcal{A}_{\text{RME}}.\text{TRY}(p)$
3: $\quad start_p \leftarrow true$
4: $\quad$ **loop forever:**
5: $\quad\quad$ **for** $q \in \underline{known}_p$ **do**
6: $\quad\quad\quad$ SEND(ALIVE, $req_p, \underline{trust}_p, q$)
7: **procedure** TASK 2
8: $\quad$ **loop forever:**
9: $\quad\quad$ **wait for** $\underline{waitlist}_p \setminus donelist_p = \emptyset$
10: $\quad\quad donelist_p \leftarrow \emptyset$
11: $\quad\quad req_p \leftarrow true$
12: $\quad\quad \mathcal{B}_{\text{RME}}.\text{TRY}$
13: $\quad\quad \mathcal{B}_{\text{RME}}.\text{EXIT}$
14: $\quad\quad req_p \leftarrow false$
15: $\quad\quad$ **if** $\underline{trust}_p \cap \underline{crash}_p = \emptyset$ **then**
16: $\quad\quad\quad tq_p \leftarrow \underline{trust}_p$
17: $\quad\quad\quad rdy_p \leftarrow \top$
18: $\quad\quad\quad$ **for** $q \in \underline{known}_p$ **do**
19: $\quad\quad\quad\quad$ SEND(QUORUM, $\underline{trust}_p, \underline{crash}_p, q$)
20: $\quad\quad$ **else**
21: $\quad\quad\quad \underline{trust}_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$
22: **procedure** TASK 3 + q
23: $\quad \underline{known}_p \leftarrow \underline{known}_p \cup \{q\}$
24: $\quad \mathcal{A}_{\text{RME}}.\text{TRY}(q)$
25: $\quad \mathcal{A}_{\text{RME}}.\text{EXIT}(q)$
26: $\quad \underline{crash}_p \leftarrow \underline{crash}_p \cup \{q\}$
27: **procedure** RECONNECTION
28: $\quad tq_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$
29: $\quad$ **for** $q \in \underline{trust}_p$ **do**
30: $\quad\quad$ **Start** TASK 3 + q
31: **when** $q \neq p$ is added to $\underline{trust}_p$
32: $\quad$ **Start** TASK 3 + q
33: **upon reception of** ALIVE $(req, trust\_src)$ **from** src **do**
34: $\quad \underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$
35: $\quad$ **if** $req = true$ **then** $\underline{waitlist}_p \leftarrow \underline{waitlist}_p \cup \{src\}$
36: $\quad$ **else**
37: $\quad\quad \underline{waitlist}_p \leftarrow \underline{waitlist}_p \setminus \{src\}$
38: $\quad\quad donelist_p \leftarrow donelist_p \cup \{src\}$
39: **upon reception of** QUORUM $(trust\_src, crash\_src)$ **from** src **do**
40: $\quad \underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$
41: $\quad \underline{crash}_p \leftarrow \underline{crash}_p \cup crash\_src$
42: $\quad$ **if** $rdy_p = \bot$ **then**
43: $\quad\quad tq_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$

Each process p has its own CS, which is handled by algorithm $\mathcal{A}_{\text{RME}}$ and accessed with $\mathcal{A}_{\text{RME}}.\text{TRY}(p)$. Additionally, there is a global CS which is handled by algorithm $\mathcal{B}_{\text{RME}}$ and accessed with $\mathcal{B}_{\text{RME}}.\text{TRY}$.

In TASK 1, p enters its own CS and then never leaves it. Since in this case a well-formed process restarts in the CS after a recovery, this means that a recovering process will restart TASK 1 directly after line 2 if it previously managed to enter its own CS. This enables other processes to detect p's failure if it crashes permanently (if another process manages to access p's CS in TASK $3 + p$, it means that p crashed forever). In TASK 1, p also sends information to the rest of the system about its own identity and whether or not p is trying to access the global CS. These ALIVE messages are used by other processes to keep $\underline{trust}_p$, $\underline{waitlist}_p$, and $donelist_p$ up to date.

In TASK 2, p tries infinitely often to access the global CS. The wait on line 9 helps to ensure that the global CS starvation freedom property is satisfied. After entering and leaving the global CS, if p entered it using only messages from processes that are not crashed (test on line 15), then p updates its $\mathcal{T}\Sigma^{lr}$ output variables and informs other processes with QUORUM messages. However, if p used information from crashed processes to enter CS, it removes them from its $\underline{trust}_p$ set.

TASK $3 + q$ is started by p when q is added to $\underline{trust}_p$, and is used to detect q's permanent crash.

When a process p receives a QUORUM message, it updates its local $\underline{trust}_p$ and $\underline{crash}_p$ information and, if rdy_p is currently $\perp$ (and therefore p is not currently trying to verify the live pairs intersection property), then p updates its tq_p.

Lemma 7 (Starvation freedom)**.** *Every eventually up process passes the lines 12 – 13 infinitely often.*

Proof. By contradiction, let us assume that there is an eventually up process p which does not go through CS infinitely often. There are two ways this could happen: p is either stuck in the wait on line 9 forever, or p is stuck in try section on line 12 forever.

First let us assume that p is stuck in try section forever. Since the liveness property of RME is verified, and since no process can stay in CS forever (since the CS has no code), it follows that there is a process q that enters CS infinitely often.

Eventually, $p \in \underline{known}_q$ and $q \in \underline{known}_p$. Since p set req_p to $true$ on line 11, then eventually q will receive an ALIVE message from p with req set to $true$, and q will add p to $\underline{waitlist}_q$. Because of the first in, first out property, q will eventually stop receiving any ALIVE message from p that has the req value set to $false$. Since q passes the line 10 infinitely often, eventually $p \notin donelist$. Since $p \in \underline{waitlist}_q \setminus donelist_q$, then eventually q will wait forever on line 9, which is a contradiction.

Now let us assume that p is stuck on line 9 forever. Let W be the set of processes that stay in $\underline{waitlist}_p \setminus donelist_p$ for infinitely long. Note that a process q that is not stuck forever in the try section on line 12 would have their req set to $false$ and therefore would send an ALIVE message to p with req set to $false$, and would be removed from $\underline{waitlist}_p \setminus donelist_p$ as a result. It follows that every $q \in W$ is stuck forever on line 12. If q is eventually down, it eventually crashes forever and therefore cannot be in W. If q is eventually up, according to the previous paragraph it eventually enters CS and therefore cannot be in W. If q is unstable, it eventually crashes and resets its req_q to $false$, and therefore cannot be in W. As a result, W is empty and p eventually ends the wait on line 9.

□

Lemma 8 (Crashed completeness)**.** *A process can only be added to $\underline{crash}_p$ if it crashed forever.*

Proof. A process can only be added to $\underline{crash}_p$ on lines 26 and 41. In order for p to add a process to $\underline{crash}_p$ on line 41, some other process q must have added it to $\underline{crash}_q$ on line 26 first.

In order for p to add a process q to $\underline{crash}_p$ on line 26, p must first have started TASK $3 + q$. This can only happen if p added q to $\underline{trust}_p$. A process can be added to $\underline{trust}_p$ on lines 34 and 40, or by receiving information from q as part of algorithm $\mathcal{B}_{\text{RME}}$. If q sent a QUORUM message, then it must have passed the CS on lines 12 – 13 and therefore sent or received information as part of algorithm $\mathcal{B}_{\text{RME}}$, which means that $start_q$ was set to $true$. Whether q set $start_q$ to $true$ on line 3 or sent an ALIVE message on line 6, it had to enter its own CS on line 2 first.

Since q entered its own CS before p started TASK $3 + q$ and will never leave it, the only way that p can reach line 26 and add q to $\underline{crash}_p$ is if q crashed forever. □

Claim 4 (Strong completeness)**.** *Algorithm 3 ensures the strong completeness property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually down process, and q be a process that is not eventually down. Note that by construction, a process can never be added to tq_q without being added to $\underline{trust}_q$ first. There are two cases:

p was never added to $\underline{trust}_q$. Then the property is immediately verified.

p was added to $\underline{trust}_q$. Let r be some eventually up process. Eventually, q will send an ALIVE message to r which contains $\underline{trust}_q$. Therefore, r will eventually add p to its $\underline{trust}_r$ and will then start TASK $3 + p$. After p crashes forever, eventually r will reach line 26 and add p to $\underline{crash}_r$.

Let t_1 be a time after which all eventually down processes have crashed. Let $t_2 \geq t_1$ be a time after which there are no more messages sent by eventually down processes in the system. After t_2, neither q nor r will ever add an eventually down process into their $\underline{trust}$ set again. According to Lemma 7, r will then eventually remove all eventually down processes from $\underline{trust}_r$ on line 21. Since according to Lemma 8 only eventually down processes can be in $\underline{crash}_r$, after this time r will always pass the test on line 15 and therefore r will send a QUORUM message to q infinitely often.

If q goes through the loop in TASK 1 infinitely often, it will act like r and eventually never have p in its tq_q. If q is unstable and does not go through the loop in TASK 1 infinitely often, then after it stops going through the loop it will crash and reset its rdy_q to $\perp$. Then, the next time that q receives a QUORUM message from r, it will add p to $\underline{crash}_q$ and remove it from tq_q on line 43 □

Claim 5 (Eventually strong accuracy)**.** *Algorithm 3 ensures the eventually strong accuracy property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually up process, and q a process that is not eventually down. Eventually, $q \in \underline{known}_p$. According to the liveness property of RME, p will eventually enter its own CS and send an ALIVE message to q on line 6. When q receives the message, it will add p to its $\underline{trust}_q$ set on line 34. It follows from Lemma 8 that p will never be in $\underline{crash}_q$. According to the proof for Claim 4, q will update its tq_p infinitely often with $\underline{trust}_q$, either on line 16 or on line 43. As a result, $p \in tq_q$ forever. □

Claim 6 (Trusting accuracy)**.** *By construction, the only way that a process can be removed from tq_p is by being added to $\underline{crash}_p$. The proof then follows directly from Lemma 8.*

Claim 7 (Quorum readiness)**.** *Algorithm 3 ensures the quorum readiness property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. Let p be an eventually up process. According to the proof for Claim 4, p will pass the test on line 15 infinitely often. After p stops crashing, the next time it reaches line 17, it will set rdy_p to $\top$ forever. □

Lemma 9 (Message reception intersection)**.** *Let p_1 and p_2 be two processes that enter the CS of $\mathcal{B}_{\text{RME}}$ at time t_1 (resp. t_2). Let Q_1 (resp. Q_2) be the set of all processes from which p_1 (resp. p_2) received information from (directly or through forwarding) since the last time it entered the try section before t_1 (resp. t_2). Then either one of the process crashed permanently before the other entered CS, or $Q_1 \cap Q_2 \neq \emptyset$.*

Proof. By contradiction, let us assume that $Q_1 \cap Q_2 = \emptyset$.

First let us assume that in $\mathcal{B}_{\text{RME}}$, a process r might send a message to a process s to authorize s to enter CS before s has entered the try section. In this case, it is possible that every process in the system would send such a message to s before s enters the try section. Let us now consider a run in which a process s' different from s later enters the try section. If $\mathcal{B}_{\text{RME}}$ allows some process to authorize s', then all other processes might do the same thing. As a result, if s is not permanently crashed, s and s' might enter CS at the same time, thus violating the safety property. If $\mathcal{B}_{\text{RME}}$ does not allow any process to authorize s', then s might never enter the try section, thus violating the liveness property. It follows that in $\mathcal{B}_{\text{RME}}$, only messages received after entering the try section can authorize a process to enter CS.

Let us now consider a run in which every message between Q_1 and Q_2 is delayed until after both p_1 and p_2 have left CS. This means that the system is partitioned, and therefore algorithm $\mathcal{B}_{\text{RME}}$ cannot possibly prevent a run in which both p_1 and p_2 enter CS at the same time, thus violating the safety property of RME. □

Claim 8 (Live pairs intersection)**.** *Algorithm 3 ensures the live pairs intersection property of $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Proof. The live pairs intersection property only applies when rdy_p is set to $\top$, and the only way to set rdy_p to $\top$ is on line 17. Since lines 28 and 43 can only be reached when rdy_p is set to $\bot$, it follows that at any time rdy_p is equal to $\top$, the current value of tq_p was set on line 16.

Note that tq_p is set from $\underline{trust}_p$ on line 16 after p recently went through the global try, critical, and exit sections with $\mathcal{B}_{\text{RME}}$ on lines 12 – 13. By construction, every process from which p received information (even indirectly) in $\mathcal{B}_{\text{RME}}$ since last entering the try section is in $\underline{trust}_p$ at that time. Observe also that the only way to remove a process identity from $\underline{trust}_p$ is on line 21, which cannot be reached between lines 12 and 16.

Let p_1 and p_2 be two processes, and let t be some time at which both are alive. Then for any time $t_1 \leq t$ when p_2 reached line 16 and any time $t_2 \leq t$ when p_2 reached line 16, it follows from Lemma 9 that $\underline{trust}_{p_1}$ at time t_1 and $\underline{trust}_{p_2}$ at time t_2 intersect. □

From Claims 4 to 8, we can deduce the following theorem:

Theorem 3 (Correctness)**.** *The Algorithm 3 implements $\mathcal{T}\Sigma^{lr}$ in $\mathcal{M}_{\text{RME}}$.*

Corollary 2 (Necessity)**.** *The $\mathcal{T}\Sigma^{lr}$ failure detector is necessary to solve the RME in any unknown dynamic environment with partial memory loss.*

Conclusion

In this paper, we redefined the $(\mathcal{T}+\Sigma^l)$ failure detector into the $\mathcal{T}\Sigma^{lr}$ failure detector adapted to unknown dynamic systems with partial memory loss and where faulty processes may recover. We proved that $\mathcal{T}\Sigma^{lr}$ is both necessary and sufficient to solve the RME problem in such systems, and it is therefore the weakest failure detector to solve RME in unknown dynamic systems with partial memory loss.

Additionally, we showed that the properties of $(\mathcal{T}+\Sigma^l)$ can apply to two separate output variables or to a single one, without changing the strength of the failure detector.

We focused on a specific definition of the mutual exclusion problem for crash-recovery, more specifically the variant where processes stay in CS after a temporary crash. It would be interesting to study other definitions, considering, for instance, that temporary crashes make a process to restart from the remainder section, even if it was in the critical section previously. On the other hand, the definition that we adopted in this paper provides stronger properties, and notably ensures that once a process, which is not eventually down, enters the critical section, it does not have to leave it until it decides to.

Acknowledgements

E. Mauffret was supported by the RainbowFS project, funded by the ANR project ANR-16-CE25-0013 within the program (DS0703) 2016.

D. Jeanneau was supported by the Labex SMART, supported by French state funds managed by the ANR within the Investissements d'Avenir programme under reference ANR-11-LABX-65.

References

[1] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, February 1991.

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

[3] Vibhor Bhatt, Nicholas Christman, and Prasad Jayanti. Extracting quorum failure detectors. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009*, pages 73–82, 2009.

[4] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *IJPEDS*, 27(5):387–408, 2012.

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

[6] Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Ninth Symposium on Reliable Distributed Systems, SRDS 1990*, pages 146–154, 1990.

[7] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection bounds on atomic object implementations. *JACM*, 57(4), 2010.

[8] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, apr 2005.

[9] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

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

[11] Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017*, pages 211–220, 2017.

[12] Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended abstract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016*, pages 65–74, 2016.

[13] Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *31st International Symposium on Distributed Computing, DISC 2017*, pages 30:1–30:15, 2017.

[14] S. Nishio, E. Manning, and K. Li. A resilient mutual exclusion algorithm for computer networks. *IEEE Transactions on Parallel and Distributed Systems*, 1:344–356, 07 1990.

[15] Thibault Rieutord, Luciana Arantes, and Pierre Sens. Détecteur de défaillances minimal pour le consensus adapté aux réseaux inconnus. In *Algotel*, 2015.

[16] Julien Sopena, Luciana Bezerra Arantes, and Pierre Sens. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*.