The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment.
Etienne Mauffret, Denis Jeanneau, Luciana Arantes, Pierre Sens

To cite this version:
Etienne Mauffret, Denis Jeanneau, Luciana Arantes, Pierre Sens. The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment.. [Technical Report] LISTSIC; Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606. 2017. <hal-01661127v2>

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment

Etienne Mauffret
LISTIC/Université Savoie Mont Blanc

Denis Jeanneau, Luciana Arantes and Pierre Sens
Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6

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 \((T+\Sigma^l)\) [3], which is the weakest failure detector to solve mutual exclusion in known static distributed systems. In this paper, we prove that \((T+\Sigma^l)\) is also the weakest failure detector to solve mutual exclusion in unknown dynamic systems with partial memory losses. We consider that crashed processes may recover.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms → Distributed algorithms

Keywords and phrases Distributed algorithms, dynamic networks, mutual exclusion, failure detectors

Digital Object Identifier 10.4230/LIPIcs...

Funding 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.

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 opportunistic 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 these systems are unknown, which means that processes do not initially know which other processes belong to the network, and only discover it 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,
Failure Detector for Mutual Exclusion in Dynamic Networks

at any time, each process is either in the try, critical, exit or remainder section. 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 papers 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 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

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 \( T \) 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 \( (T + \Sigma^l) \)
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 expands on these results by providing
a definition of \( (T + \Sigma^l) \) adapted to unknown systems where crashed processes can recover,
and proving that it is the weakest failure detector to solve fault-tolerant mutual exclusion
in unknown dynamic systems with partial memory loss.

The contributions of our paper are threefold:

- Adapting the definition of \( (T + \Sigma^l) \) for unknown systems where crashed processes can
  recover;
- A RME algorithm that runs on top of the proposed model using the \( (T + \Sigma^l) \) failure
detector and which tolerates crashes and recovery of processes, thus proving that \( (T + \Sigma^l) \)
is sufficient to solve RME in our model;
- A reduction algorithm proving the necessity of \( (T + \Sigma^l) \) to solve RME in our model.

The rest of the paper is organized as follows: Section 2 presents our distributed system
model. Section 3 presented an adapted definition of the \( (T + \Sigma^l) \) failure detector. Section 4
provides an algorithm solving mutual exclusion using \( (T + \Sigma^l) \). Finally, Section 5 proves
that \( (T + \Sigma^l) \) is necessary to solve mutual exclusion in an unknown dynamic distributed
system.
2 Model

This section presents the distributed system model used throughout the rest of the paper.

2.1 System Model

The system is composed of a finite set of processes, denoted $\Pi$. 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. Because 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 it knows of.

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.

- **Unicity 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. This does not require 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 $C_5$ 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 $\text{known}_p$ (in stable storage) that initially contains only $p$. Eventually, $\text{known}_p$ contains the set of all processes that are not eventually down. For the sake of simplicity, our algorithms do not attempt to define the $\text{known}_p$ 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 is either in the remainder, try, critical or 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 (this is 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.

### 3 Failure Detector

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 $D_1$ is said to be weaker than $D_2$ if there exists a distributed algorithm that can implement $D_1$ using the information on failures provided by $D_2$. Intuitively, this means that the computing power provided to the system by $D_2$ is stronger than the computing power provided by $D_1$. 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. introduce the trusting failure detector $T$ and prove that it is the weakest failure detector to solve fault-tolerant mutual exclusion in a system with a majority of correct processes. $T$ provides each process with a list of trusted processes. It ensures that faulty processes are eventually not trusted by any correct process (strong completeness), that eventually all correct processes trust each other (eventually strong accuracy), and that at all times, if $T$ stops trusting a process, then the process is crashed.

Bhatt et al. introduce in [3] the $\Sigma^1$ quorum failure detector. $\Sigma^1$ is a variant of the $\Sigma$ quorum failure detector [7] adapted for the mutual exclusion problem. It provides each process with a quorum of process identities that are eventually ensured to be correct, and also ensures that if two processes are alive at some point in time, then all of their quorums up to this point intersect. The paper shows that $T$ and $\Sigma^1$ used together, denoted $(T+\Sigma^1)$, constitute the weakest failure detector to solve mutual exclusion with any number of process failures in static, known systems.

### 3.2 The $(T+\Sigma^1)$ Failure Detector

The existing definition of $(T+\Sigma^1)$ is meant for static, known networks, and therefore we need to provide a new definitions suited to 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 $\bot$ concept introduced in [15].

Additionally, the traditional properties of $(T+\Sigma^1)$ are expressed in terms of correct and faulty processes. The version of $(T+\Sigma^1)$ used here was rewritten using the concepts of eventually up and eventually down processes instead.

The $(T+\Sigma^1)$ 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 $\bot$, and then is changed to $\top$ once the failure detector has gathered enough information to verify the live pairs intersection property. We denote $tq_{lp}$ the value of $tq_p$ at time $t$, and $rdy_{lp}$ the value of $rdy_p$ at time $t$. We say that process $p$ trusts process $q$ at time $t$ if $q \in tq_{lp}$, that $p$ suspects $q$ at time $t$ if $q \notin tq_{lp}$, and that process $p$ is ready at time $t$ if $rdy_{lp} = \top$. 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.
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 $(\mathcal{T}+\Sigma^t)$. Similarly, the strong completeness and live pairs intersection properties are the original properties of $\Sigma^t$, 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 $(\mathcal{T}+\Sigma^t)$.

Note that the strong completeness is both a trusting property and a quorum property, since both $\mathcal{T}$ and $\Sigma^t$ 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 $\mathcal{T}$ ans $\Sigma^t$ 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.

In a static, known system with reliable channels and prone to crash failures without recovery, this new definition of $(\mathcal{T}+\Sigma^t)$ is equivalent to the traditional definition $(\mathcal{T}+\Sigma^t)$.

4 Sufficiency of $(\mathcal{T}+\Sigma^t)$ 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 $(\mathcal{T}+\Sigma^t)$ failure detector.

4.1 Algorithm Description

In Algorithm 1, each process $p$ which is in the try section issues a request of the form $(\text{round}_p, p)$, where $\text{round}_p$ is the current round number of $p$. Requests are totally ordered by their priority, which is defined as follows: $\text{priority(\text{round}_p, p)} > \text{priority(\text{round}_q, q)} \Leftrightarrow \text{round}_p < \text{round}_q \text{ or } [\text{round}_p = \text{round}_q \text{ and } p < q]$.

The highest function (called on line 18) takes a list of requests and returns the couple $(\text{round}, id)$ of the request with the highest priority among the trusted processes according to $tq_p$.

Each process $p$ has access to the output of its respective local failure detector, $tq_p$ and $rdy_p$. It also keeps the following local variables, initialized with the indicated value:

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

- $\text{round}_p \leftarrow 0$: 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.

- $\text{last\_round}_p \leftarrow 0$: a table associating each known process identity with its last known round number. This is used to restore the round number of other processes after they crash and recover.

- $\text{req}_p \leftarrow \text{false}$: a flag indicating that $p$ is currently in the try section.

- $\text{requests}_p \leftarrow \emptyset$: the set of requests received by $p$. Each request is a couple $(\text{round}, pid)$. 

Algorithm 1 Solving RME with \( (T+\Sigma') \): code for process \( p \)

1: **procedure** TRY SECTION
2: 
3: 31:
4: \[ \textbf{if } 32: \text{ \textbf{end procedure}} \]
5: \[ \text{req}_{p} \leftarrow \text{true} \]
6: \[ \text{round}_{p} \leftarrow \text{round}_{p} + 1; \text{grants}_{p} \leftarrow \{p\} \]
7: \[ \text{end procedure} \]
8: \[ \text{for } q \in \text{req}_{p} \text{ do send}(	ext{REQUEST, round}_{p}, q) \]
9: \[ \text{requests}_{p} \leftarrow \text{requests}_{p} \cup \{(\text{round}_{p}, p)\} \]
10: \[ \text{CHECK REQUESTS()} \]
11: \[ \text{end procedure} \]
12: \[ \text{wait for } gid_{p} = p \text{ and } rd_{p} = \top \text{ and } \text{crit}_{p} \leq \text{grants}_{p}. \]
13: \[ \text{crit}_{p} \leftarrow \text{true}; \text{req}_{p} \leftarrow \text{false} \]
14: \[ \text{procedure EXIT SECTION} \]
15: \[ \text{wait for } gid_{p} = p \text{ and } \text{crit}_{p} \leftarrow \text{false} \]
16: \[ \text{end procedure} \]
17: \[ \text{for } q \in \text{grants}_{p} \setminus \{p\} \text{ do send}(	ext{DONE, q}) \]
18: \[ \text{grants}_{p} \leftarrow \{p\}; \text{requests}_{p} \leftarrow \text{requests}_{p} \setminus \{(\ast, p)\} \]
19: \[ \text{CHECK REQUESTS()} \]
20: \[ \text{end procedure} \]
21: \[ \text{procedure RECONNECTION} \]
22: \[ \text{recovering}_{p} \leftarrow \text{true} \]
23: \[ \text{update}_{p} \leftarrow \text{req}_{p} \]
24: \[ \text{for } q \in \text{update}_{p} \text{ do send}(	ext{COMEBACK, crit}_{p}, q) \]
25: \[ \text{wait for } \text{recover}_{p} = \emptyset \]
26: \[ \text{recovering}_{p} \leftarrow \text{false} \]
27: \[ \text{end procedure} \]
28: \[ \text{when } q \text{ added to } \text{req}_{p} \]
29: \[ \text{when } q \text{ removed from } \text{req}_{p} \]
30: \[ \text{CHECK REQUESTS()} \]
31: \[ \text{CHECK REQUESTS()} \]
32: \[ \text{CHECK REQUESTS()} \]
33: \[ \text{CHECK REQUESTS()} \]
34: \[ \text{CHECK REQUESTS()} \]
35: \[ \text{CHECK REQUESTS()} \]
36: \[ \text{CHECK REQUESTS()} \]
37: \[ \text{CHECK REQUESTS()} \]
38: \[ \text{CHECK REQUESTS()} \]
39: \[ \text{CHECK REQUESTS()} \]
40: \[ \text{upon reception of REQUEST (round) from src do} \]
41: \[ \text{requests}_{p} \leftarrow \text{requests}_{p} \cup \{(\text{round}, src)\} \]
42: \[ \text{last_round}_{p}[src] \leftarrow \text{round} \]
43: \[ \text{CHECK REQUESTS()} \]
44: \[ \text{upon reception of GRANT () from src do} \]
45: \[ \text{if } gid_{p} \neq -1 \text{ and } gid_{p} \neq p \text{ then} \]
46: \[ \text{SEND(REJECT, src)} \]
47: \[ \text{else if } \text{recovering}_{p} = \text{false} \text{ then} \]
48: \[ \text{grants}_{p} \leftarrow \text{grants}_{p} \cup \{\text{src}\} \]
49: \[ \text{upon reception of DONE () from src do} \]
50: \[ \text{requests}_{p} \leftarrow \text{requests}_{p} \setminus \{(\ast, src)\} \]
51: \[ \text{if } \text{crit}_{src} = \text{false and } gid_{p} = src \text{ then} \]
52: \[ \text{CHECK REQUESTS()} \]
53: \[ \text{upon reception of REJECT () from src do} \]
54: \[ \text{CHECK REQUESTS()} \]
55: \[ \text{upon reception of COMEBACK (crit}_{src} \text{ from src do} \]
56: \[ \text{CHECK REQUESTS()} \]
57: \[ \text{SEND(UPDATE, gid}_{p} = src, last_round}_{p}[src], src \in \text{grants}_{p}, \text{round}_{p}, \text{req}_{p}, src) \]
58: \[ \text{upon reception of UPDATE (grant}_{p}, last}_{rd}, \text{grant}_{src}, \text{round}, \text{req}) \text{ from src do} \]
59: \[ \text{CHECK REQUESTS()} \]
60: \[ \text{if } \text{grant}_{src} = \text{true then} \text{ p previously} \]
61: \[ \text{true then} \text{ src previously} \]
62: \[ \text{CHECK REQUESTS()} \]
63: \[ \text{CHECK REQUESTS()} \]
64: \[ \text{CHECK REQUESTS()} \]
65: \[ \text{CHECK REQUESTS()} \]
66: \[ \text{CHECK REQUESTS()} \]
67: \[ \text{CHECK REQUESTS()} \]
68: \[ \text{CHECK REQUESTS()} \]
69: \[ \text{CHECK REQUESTS()} \]
70: \[ \text{CHECK REQUESTS()} \]
71: \[ \text{CHECK REQUESTS()} \]
Failure Detector for Mutual Exclusion in Dynamic Networks

\[ gid_p \leftarrow -1: \text{the identity of the last process to which } p \text{ granted its permission, or } -1 \text{ if } p \]
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.

\[ grnd_p \leftarrow -1: \text{the current round number of the process to which } p \text{ granted its permission,} \]
or \(-1\) if \( p \) did not grant it.

\[ grants_p \leftarrow \{p\}: \text{the set of processes from which } p \text{ received a GRANT message.} \]
\[ recovering_p \leftarrow \text{false}: \text{a flag indicating that } p \text{ is currently attempting to rebuild its} \]
volatile memory after a crash. Calls to TRY SECTION and EXIT SECTION will be delayed
while \( recovering_p = \text{false.} \)

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

All of these local variables, except for \( crit_p \), are stored in volatile memory. This means
that after a crash and recovery, they are reinitialized to the above default value.

The following types of messages are used by Algorithm 1:

- REQUEST: asks for permission to enter CS. The message contains the round number of
the sender.
- GRANT: grants permission to a process to enter CS.
- DONE: notifies other processes that the sender just exited CS.
- REJECT: warns that the sender already gave its permission to a process different from
the sender, thus preventing deadlocks.
- COMEBACK: notifies other processes that the sender just recovered from a crash.
- UPDATE: gives information to a recently recovered process about requests, previously
given permissions and its current round number.

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).

When a process \( p \) wants to access the critical section, it executes the TRY SECTION: \( p \)
increments its \( round_p \) and resets its \( grants_p \) set (line 4), then broadcasts a REQUEST to
every process in \( tq_p \) (line 5). If a new process is discovered while \( p \) is still in the try section,
the request will also be sent to this new 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.

When \( p \) receives a REQUEST message from process \( q \) (lines 40 – 43), it 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 \( p \) receives 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, then 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
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 flag to true, which will temporarily prevent the algorithm from going into the try or exit sections (lines 2 and 11) and 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 \( \text{tq}_{p} \). Other processes will send UPDATE messages in response, which enables \( p \) to restore its \( \text{last}_\text{round}_{p} \), \( \text{round}_{p} \), \( \text{gid}_{p} \), \( \text{grnd}_{p} \) and \( \text{requests}_{p} \) 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 to choose a process to grant its permission to (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 (\( \text{crit}_{q} = \text{false} \)), 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 \( \text{requests}_{p} \), \( \text{grants}_{p} \) and \( \text{update}_{p} \) 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 (Unicity of the permission). Let \( p,q_{1},q_{2} \) be three distinct alive processes. If \( p \in \text{grants}_{q_{1}} \) 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 \( \text{gid}_{p} \). Note that the definition of the \( \text{HIGHEST} \) function also implies that \( q \in \text{tq}_{p} \) 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 \( t_{G} \), \( \text{gid}_{p} = q_{1} \)).

Let us assume that there is a time \( t > t_{G} \) such that \( p \in \text{grants}_{q_{1}} \). 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 \( \text{gid}_{p} \) to \( -1 \) or to \( p \) at some time \( t' \in [t_{G},t] \) (otherwise \( p \) cannot pass the test on line 17). This affection can only be done in one of the following ways:

**Line 38:** then \( q_{1} \notin \text{tq}_{p}' \). Since \( q_{1} \in \text{tq}_{p}^{q_{1}} \), according to the trusting accuracy property of \( (T+\Sigma)^{d} \), \( q_{1} \) has crashed at some time before \( t' \) and will never recover. It is therefore impossible that \( p \in \text{grants}_{q_{1}} \) at time \( t \).

**By crashing.** If \( p \) crashed between \( t_{G} \) and \( t' \), then its \( \text{gid}_{p} \) 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 \( q_{1} \in \text{tq}_{p}' \), then according to the trusting accuracy property of \( (T+\Sigma)^{d} \), either \( q_{1} \) crashed before \( t'' \) and will never recover (which is a contradiction), or \( q_{1} \notin \text{tq}_{p}'' \). \( p \) will therefore
send a COMEBACK message to $q_1$ on line 27, and $q_1$ will respond with an 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 $(T+\Sigma^l)$,
there is a process $q$ such that $q \in t_{lp_1} \cap t_{lp_2}$. 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
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 \( \text{grants}_p \), and \( q \) will eventually either crash forever or set \( \text{gid}_q \) 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 \((\mathcal{T} + \Sigma')\), \( p \) will eventually be removed from \( \text{ts}_q \) and \( q \) will set \( \text{gid}_q \) to \(-1\) on line 38.

2) \( q \) is eventually down, and \( p \) is not. Then according to the strong completeness property of \((\mathcal{T} + \Sigma')\), \( q \) will eventually be removed from \( \text{ts}_p \) and \( p \) will remove \( q \) from \( \text{grants}_p \) on line 34.

3) At time \( t \), \( \text{gid}_p \neq -1 \) and \( \text{gid}_p \neq p \). Then when \( p \) receives \( q \)'s \text{GRANT} message, it will never add \( q \) to \( \text{grants}_p \), and \( q \) will send a \text{REJECT} message instead (line 46). When \( q \) receives the \text{REJECT} message, it will set \( \text{gid}_q \) to \(-1\) (line 54).

4) At time \( t \), \( \text{gid}_p = -1 \). When \( p \) calls \text{CHECK REQUESTS}, it will pass the test one line 17 since \( \text{request}_p \) contains at least \( p \)'s request, and \( \text{crit}_p \) and \( \text{recovering}_p \) cannot be true while in CS. \( p \) will then set \( \text{gid}_p \) 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 \( \text{gid}_p = p \) 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 \( \text{ts}_p \), and \( \text{rdy}_p \) 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 \( \text{grants}_p \) on line 14 and send a DONE message to \( q \) on line 13. When \( q \) receives the DONE message, it will set \( \text{gid}_q \) 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 \( \text{rdy}_p \) stays equal to \( \bot \)). Then at time \( t' \) when \( p \) receives \( p_h \)'s request, it will call \text{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 \( \text{gid}_p \) to \( p_h \). It will then remove \( q \) from \( \text{grants}_p \) on line 21 and send a \text{REJECT} message to \( q \) on line 22. When \( q \) receives the \text{REJECT} message, it will set \( \text{gid}_q \) to \(-1\) on line 54.

3) During \([t_R, t']\), \( p \) crashes before receiving enough \text{GRANT} messages to enter CS. When \( p \) recovers, its \( \text{grants}_p \) set is reinitialized and does not contain \( q \). Since \( q \) was previously in \( \text{ts}_p \) and \( q \) is not eventually down, it follows from the trusting accuracy property of \((\mathcal{T} + \Sigma')\) that \( q \) is still in \( \text{ts}_p \) after \( p \) recovers. \( p \) will therefore send a \text{COMEBACK} message to \( q \) on line 27 with the \text{crit\_src} parameter set to \text{false}. When \( q \) receives the \text{COMEBACK} message, it will set \( \text{gid}_q \) 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 $\text{round}_p$ 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 $(T+\Sigma^l)$, $p$ 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 $(T+\Sigma^l)$, there is a time $t_1 \geq t_0$ such that $\forall t > t_1$, $tq_0^p$ 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 $\text{last\_round}_q[p]$ will be up to date with $q$'s latest $\text{round}_p$ value from before the crash. $q$ will then respond with an UPDATE message, and $p$ will update its $\text{round}_p$ value on line 63 before leaving the recovery phase. As a result, crashes after $t_3$ do not reduce or reset $p$'s $\text{round}_p$ 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 $\text{round}_p$ 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 $\text{round}_p$ 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 $\text{round}_p$ 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 $t' > t$ after which $p$ has the highest priority of all requesting processes in the system.
If \( \text{gid}_p = q_2 \) with \( q_2 \) distinct from \( q \) after \( t' \), then according to Lemma 3, eventually \( p \) will set its \( \text{gid}_p \) to \(-1\) and then call \text{CHECK REQUESTS}. \( p \) will then set itself as \( \text{gid}_p \) on line 18 and will never change \( \text{gid}_p \) again.

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

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 \( \text{round}_p \) 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 6** (Correctness). The Algorithm 1 solves the RME using \((T + \Sigma^l)\) in any unknown dynamic environment.

**Corollary 7** (Sufficiency). The \((T + \Sigma^l)\) failure detector is sufficient to solve the RME in any unknown dynamic environment with partial memory loss.
In this section we prove that the \((T + \Sigma^t)\) 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 \(\mathcal{M}_{RME}\) with partial memory loss, in which RME can be solved with some algorithm \(A_{RME}\). We will then show that the properties of \((T + \Sigma^t)\) can be implemented in \(\mathcal{M}_{RME}\).

The following proof is inspired by the proofs for the necessity of \(T\) and \(\Sigma\) 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 will make use of two algorithms, both of which share the following local variables:

- \(\text{trust}_p \leftarrow \{p\}\) is the set of all processes that process \(p\) has heard of, that \(p\) does not suspect. This variable is in stable storage.
- \(\text{start}_p \leftarrow \text{false}\) is a flag used to delay the start of the RME algorithm.

First we introduce the algorithm \(B_{RME}\). \(B_{RME}\) has exactly the same code as \(A_{RME}\), except that every call to the \text{send} primitive is replaced by a call to \(B_{RME} \_\text{send}\), as defined in Algorithm 2.

Algorithm 2 Modified \text{send} primitive for \(B_{RME}\)

1: \(\text{procedure } B_{RME} \_\text{send}(\text{msg, dest})\)
2: \(\text{wait for start}_p = \text{true}\)
3: \(\text{send}(\text{msg, trust}_p, \text{dest})\)
4: \(\text{upon reception of (msg, trust} _\text{src}) \text{ from src do}\)
5: \(\text{wait for start}_p = \text{true}\)
6: \(\text{trust}_p \leftarrow \text{trust}_p \cup \text{trust} _\text{src}\)
7: \(B_{RME} \_\text{deliver}(\text{msg})\)

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

Lemma 8. Provided that each eventually up process \(p\) eventually sets \(\text{start}_p\) to \text{true}, Algorithm \(B_{RME}\) solves the RME problem in \(\mathcal{M}_{RME}\).

Proof. The only difference between \(A_{RME}\) and \(B_{RME}\) that could prevent \(B_{RME}\) from solving RME is the wait on lines 2 and 5. A process that never sets \(\text{start}_p\) to \text{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 \(\text{start}_p\) to \text{true}, then for the purpose of \(B_{RME}\), that process behaves exactly as an always down process would in a run of \(A_{RME}\). ▶

We can now introduce Algorithm 3, which makes use of \(A_{RME}\) and \(B_{RME}\) to implement the properties of \((T+\Sigma^t)\).

In addition to \(\text{trust}_p\) and \(\text{start}_p\), Algorithm 3 makes use of the following local variables:

- \(\text{known}_p \leftarrow \{p\}\): as detailed in Section 2, \(\text{known}_p\) represents the knowledge that \(p\) has of other processes in the system. The algorithm does not show how \(\text{known}_p\) is kept up to date, but simply expects that \(\text{known}_p\) will eventually contain the process identities of (at least) all eventually up processes.
- \(\text{crash}_p \leftarrow \emptyset\): the set of all processes that \(p\) is certain have crashed forever. Note that this variable is in stable storage.
Algorithm 3 Reduction Algorithm $T_{\text{ARME} \rightarrow (T + E')}$: code for process $p$

1: procedure task 1 
2: $\text{ARME}.\text{TRY}(p)$ 
3: $\text{start}_p \leftarrow \text{true}$ 
4: loop forever: 
5: for $q \in \text{known}_p$ do 
6: send(ALIVE, req$_p$, trust$_p$, q) 
7: endfor 
8: procedure task 2 
9: loop forever: 
10: wait for $\text{waitlist}_p \setminus \text{donelist}_p = \emptyset$ 
11: $\text{donelist}_p \leftarrow \emptyset$ 
12: $\text{req}_p \leftarrow \text{true}$ 
13: $\text{BRME}.\text{TRY}$ 
14: $\text{BRME}.\text{EXIT}$ 
15: if $\text{trust}_p \setminus \text{crash}_p = \emptyset$ then 
16: $\text{tq}_p \leftarrow \text{trust}_p$ 
17: $\text{rdy}_p \leftarrow \top$ 
18: for $q \in \text{known}_p$ do 
19: send(QUORUM, trust$_p$, crash$_p$, q) 
20: endfor 
21: else 
22: $\text{trust}_p \leftarrow \text{trust}_p \setminus \text{crash}_p$ 
23: endif 
24: $\text{procedure task 3} + q$ 
25: known$_p \leftarrow \text{known}_p \cup \{q\}$ 
26: $\text{ARME}.\text{TRY}(q)$ 
27: $\text{ARME}.\text{EXIT}(q)$ 
28: crash$_p \leftarrow \text{crash}_p \cup \{q\}$ 
29: procedure reconnection 
30: $\text{tq}_p \leftarrow \text{trust}_p \setminus \text{crash}_p$ 
31: for $q \in \text{trust}_p$ do 
32: Start TASK 3 + q 
33: when $q \neq p$ is added to $\text{trust}_p$ 
34: Start TASK 3 + q 
35: upon reception of ALIVE (req, trust$_\text{src}$) from src do 
36: if req $= \text{true}$ then $\text{waitlist}_p \leftarrow \text{waitlist}_p \cup \{src\}$ 
37: $\text{trust}_p \leftarrow \text{trust}_p \cup \text{trust}_\text{src}$ 
38: $\text{donelist}_p \leftarrow \text{donelist}_p \cup \{src\}$ 
39: upon reception of QUORUM (trust$_\text{src}$, crash$_\text{src}$) from src do 
40: $\text{trust}_p \leftarrow \text{trust}_p \cup \text{trust}_\text{src}$ 
41: $\text{crash}_p \leftarrow \text{crash}_p \cup \text{crash}_\text{src}$ 
42: if rdy$_p = \bot$ then 
43: $\text{tq}_p \leftarrow \text{trust}_p \setminus \text{crash}_p$
Algorithm 3 initially starts two tasks in parallel: **task 1** and **task 2**. Later on when process $p$ receives knowledge of other processes, it starts a new task for each process $q$ (denoted **task 3 + q**).

Each process $p$ has its own CS, which is handled by algorithm $A_{\text{RME}}$ and accessed with $A_{\text{RME}.\text{try}}(p)$. Additionally, there is a global CS which is handled by algorithm $B_{\text{RME}}$ and accessed with $B_{\text{RME}.\text{try}}$.

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 $p$ crashed forever). **task 1** also lets $p$ send 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 $\text{trust}_p$, $\text{waitlist}_p$, and $\text{donelist}_p$ up to date.

In **task 2**, $p$ tries infinitely often to access the global CS. The wait on line 9 helps ensure that the global CS ensures the starvation freedom property. 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 $(T + \Sigma')$ 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 $\text{trust}_p$ set instead.

**task 3 + q** is started by $p$ when $q$ is added to $\text{trust}_p$, and is used to detect $q$’s permanent crash.

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

**Lemma 9** (Starvation freedom). Every eventually up processes passes the lines 12 – 13 infinitely often.

The proof for Lemma 9 can be found in the appendix.

**Lemma 10** (Crashed completeness). A process can only be added to $\text{crash}_p$ if it crashed forever.

The proof for Lemma 10 can be found in the appendix.

**Claim 4** (Strong completeness). Algorithm 3 ensures the strong completeness property of $(T + \Sigma')$ 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 $\text{tq}_q$ without being added to $\text{trust}_q$ first. There are two cases:

- $p$ was never added to $\text{trust}_q$. Then the property is immediately verified.
- $p$ was added to $\text{trust}_q$. Let $r$ be some eventually up process. Eventually, $q$ will send an **alive** message to $r$ with its $\text{trust}_q$. As a result, $r$ will eventually add $p$ to its $\text{trust}_r$. $r$ will
then start task 3 + p. After p crashes forever, eventually r will reach line 26 and add p to \( \text{crash}_r \).

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 trust set again. According to Lemma 9, r will then eventually remove all eventually down processes from \( \text{trust}_q \) on line 21. Since according to Lemma 10 only eventually down processes can be in \( \text{crash}_r \), after this time r will always pass the test on line 15 and therefore r will send a \( \text{QUORUM} \) 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 \( t_{q_p} \). 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 \( r_{dy_p} \) to \( \perp \). Then, the next time that q receives a \( \text{QUORUM} \) message from r, it will add p to \( \text{crash}_q \) and remove it from \( t_{q_p} \) on line 43.

\[ \text{Claim 5 (Eventually strong accuracy).} \quad \text{Algorithm 3 ensures the eventually strong accuracy property of } (T + \Sigma^1) \text{ in } M_{RME}. \]

**Proof.** Let \( p \) be an eventually up process, and q a process that is not eventually down. Eventually, \( q \in \text{known}_p \). 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 \( \text{trust}_q \) set on line 34. It follows from Lemma 10 that \( p \) will never be in \( \text{crash}_q \).

According to the proof for Claim 4, q will update its \( t_{q_p} \) infinitely often with \( \text{trust}_q \), either on line 16 or on line 43. As a result, \( p \in t_{q_p} \) forever.

\[ \text{Claim 6 (Trusting accuracy).} \quad \text{By construction, the only way that a process can be removed from } t_{q_p} \text{ is by being added to } \text{crash}_p. \quad \text{The proof then follows directly from Lemma 10.} \]

\[ \text{Claim 7 (Quorum readiness).} \quad \text{Algorithm 3 ensures the quorum readiness property of } (T + \Sigma^1) \text{ in } M_{RME}. \]

**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 \( r_{dy_p} \) to \( \top \) forever.

\[ \text{Lemma 11 (Message reception intersection).} \quad \text{Let } p_1 \text{ and } p_2 \text{ be two processes that enter the CS of } B_{RME} \text{ at time } t_1 \text{ (resp. } t_2 \text{). Let } Q_1 \text{ (resp. } Q_2 \text{) be the set of all processes from which } p_1 \text{ (resp. } p_2 \text{) received information from (directly or through forwarding) since the last time it entered the try section before } t_1 \text{ (resp. } t_2 \text{). Then either one of the process crashed permanently before the other entered CS, or } Q_1 \cap Q_2 \neq \emptyset. \]

The proof for Lemma 11 can be found in the appendix.

\[ \text{Claim 8 (Live pairs intersection).} \quad \text{Algorithm 3 ensures the live pairs intersection property of } (T + \Sigma^1) \text{ in } M_{RME}. \]

**Proof.** The live pairs intersection property only applies when \( r_{dy_p} \) is set to \( \top \), and the only way to set \( r_{dy_p} \) to \( \top \) is on line 17. Since lines 28 and 43 can only be reached when \( r_{dy_p} \) is set to \( \perp \), it follows that at any time \( r_{dy_p} \) is equal to \( \top \), the current value of \( t_{q_p} \) was set on line 16.

Note that \( t_{q_p} \) is set from \( \text{trust}_p \) on line 16 after \( p \) recently went through the global try, critical, and exit sections with \( B_{RME} \) on lines 12 – 13. By construction, every process from which \( p \) received information (even indirectly) in \( B_{RME} \) since last entering the try section is
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 11 that \( \text{trust}_{p_1} \) at time \( t_1 \) and \( \text{trust}_{p_2} \) at time \( t_2 \) intersect. ◀

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

**Theorem 12 (Correctness).** The Algorithm 3 implements \( (T + \Sigma^l) \) in \( M_{RME} \).

**Corollary 13 (Necessity).** The \( (T + \Sigma^l) \) failure detector is necessary to solve the RME in any unknown dynamic environment with partial memory loss.

### Conclusion

In this paper, we introduced a definition of the \( (T + \Sigma^l) \) failure detector adapted to unknown dynamic systems with partial memory loss and where faulty processes may recover. We proved that \( (T + \Sigma^l) \) 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.

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.

### References


