



HAL
open science

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

Etienne Mauffret, Élise Jeanneau, Luciana Arantes, Pierre Sens

► **To cite this version:**

Etienne Mauffret, Élise Jeanneau, Luciana Arantes, Pierre Sens. The Weakest Failure Detector to Solve the Mutual Exclusion Problem in an Unknown Dynamic Environment. [Technical Report] LISTIC; Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606. 2017. hal-01661127v1

HAL Id: hal-01661127

<https://hal.science/hal-01661127v1>

Submitted on 11 Dec 2017 (v1), last revised 31 Oct 2018 (v3)

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 Fault Tolerant Mutual Exclusion Problem in an Unknown Dynamic Environment

Etienne Mauffret*, Denis Jeanneau†, Luciana Arantes†, Pierre Sens†

*LISTIC - Polytech Annecy-Chambéry

†Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6

Abstract—Many traditional distributed algorithms are unadapted to the new dynamic architectures that have emerged, such as mobile ad-hoc networks. Mutual exclusion is one of the fundamental problems in distributed computing, and cannot be solved in an asynchronous message passing systems where processes can crash. Existing solutions include the use of $(\mathcal{T}+\Sigma^l)$, which is the weakest failure detector to solve mutual exclusion in known, static distributed systems. In this paper, we introduce the $(\mathcal{T}+\Sigma^l)^d$ failure detector and prove that it is the weakest failure detector to solve mutual exclusion in unknown dynamic systems with partial memory losses.

I. 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 wireless network (MANET), sensor wireless network (WSN), peer-to-peer networks, and opportunist grids or clouds provide access to services or information regardless of node location, mobility patten 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 the membership of the system, and can 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 systems.

The mutual exclusion problem, introduced by Dijkstra in [1], 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 section. Processes cycle through these sections in order. Two processes cannot be in the critical section at the

same time, and if a process is in the try section, then at some time later some process is in the critical section.

Several mutual exclusion algorithms in the context of static and known distributed systems that tolerate crash failures have been proposed in the literature [2] [3] [4]. However, none of them consider dynamic systems, neither 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 [5] [6], where shared variables are stored in non-volatile memory. A crash-recovery mutual exclusion for message-passing systems of which we are aware was proposed in [7] but its recovery solution works provided that failures do not occur in adjacent processes. Hence, the conception of mutual exclusion in unknown dynamic systems and where crashed processes can recover presents great challenges.

A definition of recoverable mutual exclusion (RME) for systems with crash-recovery was presented in [5] and further studied in [6]. 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 event of a temporary crash.

In this paper we consider FTME 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 denoted this model the *partial memory loss* model.

Failure detectors were introduced in [8] as a way to circumvent the impossibility to solve consensus in crash-prone asynchronous systems ([9]). In [10], 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 [11], 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 were confined to known, static systems without recovery. Our paper expands on these results by presenting a new failure detector $(\mathcal{T}+\Sigma^l)^d$, 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:

- Definition of properties necessary to be included in the $(\mathcal{T}+\Sigma^l)$ failure detector in order to cope with the dynamics of the system, the unknown membership and crash-recovery with partial memory losses;
- the weakest failure detector $(\mathcal{T}+\Sigma^l)^d$ to solve FTME in unknown dynamic systems with partial memory loss;
- a FTME algorithm that runs on top of the proposed model using the $(\mathcal{T}+\Sigma^l)^d$ failure detector.

The rest of the paper is organized as follows: Section II presents our distributed system model. Section III defines the $(\mathcal{T}+\Sigma^l)^d$ failure detector. Section IV provides an algorithm solving mutual exclusion using $(\mathcal{T}+\Sigma^l)^d$. Finally, Section V proves that $(\mathcal{T}+\Sigma^l)^d$ is necessary to solve mutual exclusion in an unknown dynamic distributed system.

II. MODEL

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

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). The system is *dynamic* in the sense that the edges in the communication graph can appear and disappear during the run. Communication channels are assumed to be reliable during the periods of time where the corresponding edge is present.

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:

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.

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 $known_p$ (in stable storage) that initially contains only p . Eventually, $known_p$ contains the set of all processes that are not eventually down. For the sake of simplicity, our algorithms do not attempt to define the $known_p$ variable and simply assume that an underlying discovery algorithm eventually fills it with the necessary process identities.¹

Problem Definition: We consider the Fault-Tolerant Mutual Exclusion (FTME) 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 [5]).

A fault-tolerant mutual exclusion algorithm must provide a TRY SECTION and an EXIT SECTION procedures such that the following properties are satisfied:

¹Typically, this discovery could be done by the algorithm implementing the failure detector.

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.

III. FAILURE DETECTOR

Failure detectors were introduced by Chandra and Toueg in [8] as a way to circumvent the impossibility to solve consensus in crash-prone asynchronous systems [9]. 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.

A. Failure Detectors for Mutual Exclusion

In [10], Delporte-Gallet et al. introduce the trusting failure detector \mathcal{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. \mathcal{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 \mathcal{T} stops trusting a process, then the process is crashed.

Bhatt et al. introduce in [11] the Σ^l quorum failure detector. Σ^l is a variant of the Σ quorum failure detector

[12] 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 \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.

B. Adapting Failure Detectors for Unknown Dynamic Systems

Most distributed computing papers on failure detectors make an assumption on the connectivity of the network (often, the communication graph is expected to be connected or complete). In a dynamic system, connectivity assumptions become much weaker and, more importantly, problem-dependant. Identifying which level of connectivity is necessary and/or sufficient to solve a given problem is one of the main challenges of dynamic systems.

Therefore, a failure detector cannot be said to be sufficient to solve a problem in a dynamic system, unless the detector itself includes a connectivity property which is sufficient to solve the problem. For this reason, we augment the original $(\mathcal{T}+\Sigma^l)$ failure detector with the dynamic connectivity property.

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 [13] and adapt it into the rdy_p variable of $(\mathcal{T}+\Sigma^l)^d$.

Finally, in order to ensure the mutual exclusion in a crash-recovery model with partial memory losses, it is necessary to guarantee that (1) there is no duplication of messages, and (2) messages that a process has no memory of sending cannot be received later.

C. The $(\mathcal{T}+\Sigma^l)^d$ Failure Detector

The $(\mathcal{T}+\Sigma^l)^d$ 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 then is changed 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 $t_2 \leq t$, $(rdy_p^{t_1} = \top \wedge rdy_q^{t_2} = \top) \implies tq_p^{t_1} \cap tq_q^{t_2} \neq \emptyset$.
- **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.

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 $(\mathcal{T} + \Sigma^l)^d$.

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 $(\mathcal{T} + \Sigma^l)^d$.

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.

The dynamic connectivity, unicity of reception and first in, first out properties are new properties added to the failure detector in order to deal with dynamics and memory loss.

It might be argued that the dynamic connectivity and unicity of reception properties are unnecessary in a model with reliable channels. However, since the system is dynamic, even if every channel is reliable, a given edge in the communication graph may not be present at any given time, and the path used for two processes to communicate may change during the run. The dynamic connectivity property is therefore not trivial. Additionally, in a network with such a dynamic

graph, an algorithm might send each message multiple times in order to ensure message reception, which is why the unicity of reception property is not trivial either.

In a static, known system with reliable channels and prone to crash failures without recovery, $(\mathcal{T} + \Sigma^l)^d$ is trivially equivalent to $(\mathcal{T} + \Sigma^l)$.

IV. SUFFICIENCY OF $(\mathcal{T} + \Sigma^l)^d$ TO SOLVE FAULT-TOLERANT MUTUAL EXCLUSION

In this section we introduce Algorithm 1 and prove that it solves the FTME in any unknown dynamic environment enriched with the $(\mathcal{T} + \Sigma^l)^d$ failure detector.

A. Algorithm Description

In Algorithm 1, each process p which is in the try section issues a request of the form $(round_p, p)$, where $round_p$ is the current round number of p . 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** $[round_p = round_q \text{ and } p < q]$.

The HIGHEST function (called on line 18) 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 .

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:

$crit_p \leftarrow false$: a flag indicating that p is currently in CS. This is the only variable kept in stable storage. Thus, $crit_p$ 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. This 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 . 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 did not receive 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.

Algorithm 1 Solving FTME with $(\mathcal{T}+\Sigma^l)^d$: 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:     $(grnd_p, gid_p) \leftarrow \text{HIGHEST}(requests_p)$ 
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 ( $\underline{crit}_p, src$ ) from  $src$  do
57:   $requests_p \leftarrow requests_p \setminus \{(*, src)\}$ 
58:  if  $\underline{crit}_p = false$  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:   $round_p \leftarrow \text{MAX}(round_p, last\_rnd)$ 
65:  if  $grant\_src = true$  then  $\triangleright p$  previously granted  $src$ 
66:     $(gid_p, grnd_p) \leftarrow (src, round)$ 
67:  if  $grant\_p = true$  then  $\triangleright src$  previously granted  $p$ 
68:     $grants_p \leftarrow grants_p \cup \{src\}$ 
69:  if  $req = true$  then  $\triangleright src$  is requesting
70:     $requests_p \leftarrow requests_p \cup \{(round, src)\}$ 
71:     $update_p \leftarrow update_p \setminus \{src\}$ 

```

$recovering_p \leftarrow false$: 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 until after the recovery is done, 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.

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 a permission has already been given to another process, 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 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 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 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. If q is in its remainder section ($crit_p = false$), 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.

B. Proof of correctness

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

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

In order to prove the 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 $p \in grants_{q_1}$ 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 ways:

Line 38: then $q_1 \notin tq_p^{t'}$. Since $q_1 \in tq_p^{t_G}$, according to the trusting accuracy property of $(\mathcal{T} + \Sigma^l)^d$, 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 crashing. 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 $(\mathcal{T} + \Sigma^l)^d$, 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 of $(\mathcal{T} + \Sigma^l)^d$, 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 concludes the proof of Lemma 1. \square

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 $grants_q$. \square

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 $(\mathcal{T} + \Sigma^l)^d$, 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. \square

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 $(\mathcal{T} + \Sigma^l)^d$, 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 $(\mathcal{T} + \Sigma^l)^d$, 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 on line 17 since $requests_p$ contains at least p 's request, and $crit_p$ 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 $(\mathcal{T} + \Sigma^l)^d$ 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 of $(\mathcal{T} + \Sigma^l)^d$, 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. \square

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 $(\mathcal{T} + \Sigma^l)^d$, 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 $(\mathcal{T} + \Sigma^l)^d$, 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 of $(\mathcal{T} + \Sigma^l)^d$, 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 of $(\mathcal{T} + \Sigma^l)^d$, 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' . \square

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 of $(\mathcal{T} + \Sigma^l)^d$, 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 $(\mathcal{T} + \Sigma^l)^d$, the eventually $rdy_p = \top$.

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

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 . Similarly to the proof of Lemma 5, we can show 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. \square

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. \square

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

Theorem 1 (Correctness). *The Algorithm 1 solves the FTME using $(\mathcal{T} + \Sigma^l)^d$ in any unknown dynamic environment.*

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

V. NECESSITY OF $(\mathcal{T} + \Sigma^l)^d$ TO SOLVE FAULT-TOLERANT MUTUAL EXCLUSION

In this section we prove that the $(\mathcal{T} + \Sigma^l)^d$ failure detector is necessary to solve the FTME problem in

any unknown dynamic system with partial memory loss. For this purpose, we assume that there is an unknown dynamic system model $\mathcal{M}_{\text{FTME}}$ with partial memory loss, in which FTME can be solved with some algorithm $\mathcal{A}_{\text{FTME}}$. We will then show that the properties of $(\mathcal{T} + \Sigma^l)^d$ can be implemented in $\mathcal{M}_{\text{FTME}}$.

A. Dynamic connectivity, unicity of reception and first in, first out properties

We consider a deterministic algorithm $\mathcal{A}_{\text{FTME}}$ that solves FTME.

Claim 4 (Dynamic connectivity). *The dynamic connectivity property of $(\mathcal{T} + \Sigma^l)^d$ is verified in $\mathcal{M}_{\text{FTME}}$.*

Proof. By contradiction, we suppose that the dynamic connectivity property is not verified in $\mathcal{M}_{\text{FTME}}$. Let there be a system consisting of two eventually up processes p and q . Let there be a run of $\mathcal{A}_{\text{FTME}}$ in which both p and q are in the try section at some point in time, and they never receive each other's messages. If $\mathcal{A}_{\text{FTME}}$ makes p wait for a message from q before entering CS, since p and q are in a symmetrical situation, then both processes will wait forever, which violates the liveness property. If $\mathcal{A}_{\text{FTME}}$ makes p enter CS without waiting for a message from q , then there is a run in which both processes enter CS at the same time, thus violating the safety property. In both cases there is a contradiction, and therefore eventually up processes must be ensured to eventually receive messages in $\mathcal{M}_{\text{FTME}}$.

Let there be a system composed of an eventually up process p and another process q which is not eventually down. Both processes are in the try section at some point in time. Note that there is no way for either process to ever know for sure whether q is unstable or eventually up, and therefore q will try to enter CS and p must wait for a message from q to ensure safety (as in the case above). If q is unstable, it might never receive any message from p and therefore never knows of p 's existence. In this case q will never send any message to p , and p will wait forever, thus violating the liveness property. It follows that unstable processes must also be ensured to eventually receive messages in $\mathcal{M}_{\text{FTME}}$. \square

Claim 5 (Unicity of reception). *The unicity of reception property of $(\mathcal{T} + \Sigma^l)^d$ is verified in $\mathcal{M}_{\text{FTME}}$.*

Proof. By contradiction, we suppose that any message can be received several times by processes. Let p be a process and R a run of an $\mathcal{A}_{\text{FTME}}$ such that in R , p manages to access CS. Let us denote this sequence by S_1 . After S_1 , p crashes and another process p' manages to reach CS. Before p' leaves the CS, let us suppose that

p recovers and receives all the messages that p already received before notification of other processes. p can not dissociate this sequence from the sequence S_1 and, as $\mathcal{A}_{\text{FTME}}$ is deterministic, p will enter CS without receiving any new messages and before p' leaves CS. In this case, the safety property is violated, which is a contradiction. \square

Claim 6 (First in, first out). *The first in, first out property of $(\mathcal{T} + \Sigma^l)^d$ is verified in $\mathcal{M}_{\text{FTME}}$.*

Algorithm 2 Implementation of a FIFO property

```

1: procedure FIFO_SEND( $msg, dest$ )
2:   wait for  $dest \notin \underline{sent}_p$ 
3:    $\underline{sent}_p \leftarrow \underline{sent}_p \cup \{dest\}$ 
4:   SEND(MSG,  $msg, dest$ )
5: upon reception of MSG ( $msg$ ) from  $src$  do
6:   FIFO_DELIVER( $msg$ )
7:   SEND(ACK,  $src$ )
8: upon reception of ACK () from  $src$  do
9:    $\underline{sent}_p \leftarrow \underline{sent}_p \setminus \{src\}$ 

```

Proof. Provided that a SEND primitive that verifies the dynamic connectivity and unicity of reception properties is available, Algorithm 2 implements FIFO_SEND primitive that ensures the first in, first out property.

\underline{sent}_p is a local variable in stable storage that is initialized to \emptyset . It enables process p to remember to which processes it previously sent a message, and prevents it from sending another message to the same process until an ACK message has been received. \square

B. Trusting and quorum properties

The following proof is inspired from the proofs for the necessity of \mathcal{T} and Σ^l in [10] and [11], respectively. The main additional challenge is to merge the two proofs, since both trusting and quorum properties must apply on a same set tq_p .

The proof will make use of 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, that p does not suspect. This variable is in stable storage.

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

First we introduce the algorithm $\mathcal{B}_{\text{FTME}}$. $\mathcal{B}_{\text{FTME}}$ has exactly the same code as $\mathcal{A}_{\text{FTME}}$, except that every call to the SEND primitive is replaced by a call to $\mathcal{B}_{\text{FTME_SEND}}$, as defined in Algorithm 3.

The point of Algorithm 3 is twofold: (1) it enables p to keep track of which processes it heard of while trying

Algorithm 3 Modified SEND primitive for $\mathcal{B}_{\text{FTME}}$

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

to access CS, with $trust_p$; (2) it enables p to delay the start of the FTME algorithm, with $start_p$.

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

Proof. The only difference between $\mathcal{A}_{\text{FTME}}$ and $\mathcal{B}_{\text{FTME}}$ that could prevent $\mathcal{B}_{\text{FTME}}$ from solving FTME 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 $\mathcal{B}_{\text{FTME}}$, that process behaves exactly as an always down process would in a run of $\mathcal{A}_{\text{FTME}}$. Since always down processes exist in $\mathcal{M}_{\text{FTME}}$, and do not prevent $\mathcal{A}_{\text{FTME}}$ from solving FTME, then this is not a problem. \square

We can now introduce Algorithm 4, which makes use of $\mathcal{A}_{\text{FTME}}$ and $\mathcal{B}_{\text{FTME}}$ to implement the trusting and quorum properties of $(\mathcal{T} + \Sigma^l)^d$.

In addition to $trust_p$ and $start_p$, Algorithm 4 makes use of the following local variables:

$known_p \leftarrow \{p\}$: as detailed in Section II, $known_p$ represents the knowledge that p has of system membership. The algorithm will not detail how $known_p$ is kept up to date, and it is simply expected to eventually contain the process identities of (at least) all eventually up processes.

$crash_p \leftarrow \emptyset$: 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 $(\mathcal{T} + \Sigma^l)^d$ failure detector, which must verify the trusting and quorum properties.

$rdy_p \leftarrow \perp$: the other output variable of $(\mathcal{T} + \Sigma^l)^d$, which must verify the quorum properties.

$waitlist_p \leftarrow \emptyset$: the set of processes to which p must cede priority 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 ceded priority for CS. This prevents p from always being passed over for CS access.

Algorithm 4 initially starts two tasks in parallel: TASK 1 and TASK 2. Later on when process p receives knowledge of other processes, it will start one more task for each process q (denoted TASK 3 + q).

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

In TASK 1, p enters its own CS and then never leaves it. Since 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 someone else 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 $trust_p$, $waitlist_p$ and $donelist_p$ up to date.

In TASK 2, p tries infinitely often to access the global CS. The wait on line 9 helps ensure that the global CS ensures not only liveness but also starvation freedom. After entering and leaving the global CS, if p entered said CS using only messages from processes that are not crashed (test on line 15), then p updates its $(\mathcal{T} + \Sigma^l)^d$ output variables and informs other processes with QUORUM messages. However if p used information from crashed process to enter CS, it removes them from its $trust_p$ set instead.

TASK 3 + q is started by p when q is added to $trust_p$, and is used to detect q 's permanent crash.

When a process p receives a QUORUM message, it updates its local $trust_p$ and $crash_p$ 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 processes passes the CS on 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 FTME 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.

Algorithm 4 Reduction Algorithm $T_{\mathcal{A}_{\text{FTME}} \rightarrow (\mathcal{T} + \Sigma^t)^a}$: code for process p

```

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

```

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 $p \notin \underline{donelist}$. Since $p \in \underline{waitlist}_q \setminus \underline{donelist}_q$, 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 $\underline{waitlist}_p \setminus \underline{donelist}_p$ 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 $\underline{waitlist}_p \setminus \underline{donelist}_p$ 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 $\mathcal{B}_{\text{FTME}}$. 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 $\mathcal{B}_{\text{FTME}}$, 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. \square

Claim 7 (Strong completeness). *Algorithm 4 ensures the strong completeness property of $(\mathcal{T} + \Sigma^t)^d$ in $\mathcal{M}_{\text{FTME}}$.*

\square *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 $trust_q$ first. There are two cases.

First case: p was never added to $trust_q$. Then the property is immediately verified.

Second case: p was added to $trust_q$. Let r be some eventually up process. Eventually, q will send an ALIVE to r with its $trust_q$. As a result, r will eventually add p to its $trust_r$. r will then start TASK 3 + p . After p crashes forever, eventually r will reach line 26 and add p to $crash_r$.

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 7, r will then eventually remove all eventually down processes from $trust_r$ on line 21. Since according to Lemma 8 only eventually down processes can be in $crash_r$, 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 $crash_q$ and remove it from tq_q on line 43 \square

Claim 8 (Eventually strong accuracy). *Algorithm 4 ensures the eventually strong accuracy property of $(\mathcal{T} + \Sigma^l)^d$ in \mathcal{M}_{FTME} .*

Proof. Let p be an eventually up process, and q a process that is not eventually down. Eventually, $q \in known_p$. According to the liveness property of FTME, 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 $trust_q$ set on line 34. It follows from Lemma 8 that p will never be in $crash_q$. According to the proof for Claim 7, q will update its tq_p infinitely often with $trust_q$, either on line 16 or on line 43. As a result, $p \in tq_q$ forever. \square

Claim 9 (Trusting accuracy). *By construction, the only way that a process can be removed from tq_p is by being added to $crash_p$. The proof then follows directly from Lemma 8.*

Claim 10 (Quorum readiness). *Algorithm 4 ensures the quorum readiness property of $(\mathcal{T} + \Sigma^l)^d$ in \mathcal{M}_{FTME} .*

Proof. Let p be an eventually up process. According to the proof for Claim 7, 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. \square

Lemma 9 (Message reception intersection). *Let p_1 and p_2 be two processes that enter the CS of \mathcal{B}_{FTME} 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}_{FTME} , 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}_{FTME} 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}_{FTME} 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}_{FTME} , 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}_{FTME} 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 FTME. \square

Claim 11 (Live pairs intersection). *Algorithm 4 ensures the live pairs intersection property of $(\mathcal{T} + \Sigma^l)^d$ in \mathcal{M}_{FTME} .*

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 \perp , 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 $trust_p$ on line 16 after p recently went through the global try, critical and exit sections with \mathcal{B}_{FTME} on lines 12 – 13. By construction, every process that p received information from (even indirectly) in \mathcal{B}_{FTME} since last entering the try section

is in trust_p at that time. Also note that the only way to remove a process identity from 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 trust_{p_1} at time t_1 and trust_{p_2} at time t_2 intersect. \square

From Claims 4 to 11, we can deduce the following theorem:

Theorem 2 (Correctness). *The Algorithm 4 implements $(\mathcal{T}+\Sigma^l)^d$ in $\mathcal{M}_{\text{FTME}}$.*

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

CONCLUSION

In this paper, we introduced the $(\mathcal{T}+\Sigma^l)^d$ failure detector as an adaptation of $(\mathcal{T}+\Sigma^l)$ for unknown dynamic systems with partial memory loss. We proved that $(\mathcal{T}+\Sigma^l)^d$ is both necessary and sufficient to solve the FTME problem in such a system, and it is therefore the weakest failure detector to solve FTME 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 a different definition of the problem. For example, a weaker mutual exclusion could specify that temporary crashes cause a process to restart from the remainder section, even if it was in the critical section previously. However, the variant of the problem studied in this paper provides stronger properties, and notably ensures that once a process that is not eventually down enters the critical section, it does not have to leave it until it decides to.

REFERENCES

- [1] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.
- [2] S. Nishio, E. Manning, and K. Li, "A resilient mutual exclusion algorithm for computer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 344–356, 07 1990.
- [3] J. Sopena, L. B. Arantes, and P. Sens, "Performance evaluation of a fair fault-tolerant mutual exclusion algorithm," in *25th IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, 2-4 October 2006, Leeds, UK, 2006, pp. 225–234.
- [4] D. Agrawal and A. El Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 1–20, Feb. 1991.

- [5] W. M. Golab and D. Hendler, "Recoverable mutual exclusion in sub-logarithmic time," in *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, 2017, pp. 211–220.
- [6] P. Jayanti and A. Joshi, "Recoverable FCFS mutual exclusion with wait-free recovery," in *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, 2017, pp. 30:1–30:15.
- [7] Y. Chang, M. Singhal, and M. T. Liu, "A fault tolerant algorithm for distributed mutual exclusion," in *Ninth Symposium on Reliable Distributed Systems, SRDS 1990, Huntsville, Alabama, USA, October 9-11, 1990, Proceedings*, 1990, pp. 146–154.
- [8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
- [9] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.
- [10] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov, "Mutual exclusion in asynchronous systems with failure detectors," *Journal of Parallel and Distributed Computing*, vol. 65, no. 4, pp. 492–505, apr 2005.
- [11] V. Bhatt, N. Christman, and P. Jayanti, "Extracting quorum failure detectors," in *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing, PODC 2009, Calgary, Alberta, Canada, August 10-12, 2009*, 2009, pp. 73–82.
- [12] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui, "Tight failure detection bounds on atomic object implementations," *JACM*, vol. 57, no. 4, 2010.
- [13] T. Rieutord, L. Arantes, and P. Sens, "DéTECTEUR DE DÉFAILLANCES MINIMAL POUR LE CONSENSUS ADAPTÉ AUX RÉSEAUX INCONNUS," in *Algotel*, 2015.