



**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-01661127v2

**HAL Id: hal-01661127**

**<https://hal.science/hal-01661127v2>**

Submitted on 14 May 2018 (v2), 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.

# 1 The Weakest Failure Detector to Solve the 2 Mutual Exclusion Problem in an Unknown 3 Dynamic Environment

4 Etienne Mauffret

5 LISTIC/Université Savoie Mont Blanc

6 Denis Jeanneau, Luciana Arantes and Pierre Sens

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

## 8 — Abstract —

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

18 **2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms  
19 → Distributed algorithms

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

22 **Digital Object Identifier** 10.4230/LIPIcs...

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

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

## 27 **1 Introduction**

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

38 The mutual exclusion problem, introduced by Dijkstra in [9], is a fundamental problem  
39 in distributed computing requiring that their processes get exclusive access to one or more  
40 shared resources by executing a segment of code called critical section (CS). It specifies that,



© Author: Please provide a copyright holder;

licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

41 at any time, each process is either in the try, critical, exit or remainder section. Processes  
 42 cycle through these sections in order. Two processes cannot be in the critical section at the  
 43 same time (*safety property*), and if a process is in the try section, then at some time later  
 44 some process is in the critical section (*liveness property*).

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

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

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

66 Failure detectors were introduced in [5] as a way to circumvent the impossibility to  
 67 solve consensus in crash-prone asynchronous systems ([10]). In [8], the  $\mathcal{T}$  failure detector  
 68 was shown to be the weakest failure detector to solve fault-tolerant mutual exclusion in  
 69 message passing systems with a majority of correct processes. Then, in [3], the  $(\mathcal{T}+\Sigma^l)$   
 70 failure detector was shown to be the weakest failure detector to solve the same problem with  
 71 no assumption on the number of process failures. Both of these results are restricted to  
 72 known, static systems without recovery. Our paper expands on these results by providing  
 73 a definition of  $(\mathcal{T}+\Sigma^l)$  adapted to unknown systems where crashed processes can recover,  
 74 and proving that it is the weakest failure detector to solve fault-tolerant mutual exclusion  
 75 in unknown dynamic systems with partial memory loss.

76 The contributions of our paper are threefold:

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

83 The rest of the paper is organized as follows: Section 2 presents our distributed system  
 84 model. Section 3 presented an adapted definition of the  $(\mathcal{T}+\Sigma^l)$  failure detector. Section 4  
 85 provides an algorithm solving mutual exclusion using  $(\mathcal{T}+\Sigma^l)$ . Finally, Section 5 proves  
 86 that  $(\mathcal{T}+\Sigma^l)$  is necessary to solve mutual exclusion in an unknown dynamic distributed  
 87 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.

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

138 Our algorithms assume that the underlying SEND/RECEIVE implementation handles mes-  
 139 sage forwarding, and therefore behaves the same way that it would in a complete commu-  
 140 nication graph with reliable channels.

## 141 2.4 Knowledge Model

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

## 152 2.5 Problem Definition

153 We consider the Recoverable Mutual Exclusion (RME) problem, which we define in our  
 154 model as follows. At any point in time, a process is either in the remainder, try, critical or  
 155 exit section. We consider that every user is well-formed, that is that a user will go through  
 156 the remainder, try, critical and exit sections in the correct order. In case of a crash and  
 157 recovery, a well-formed user will restart in the critical section if it was in the critical section  
 158 when it crashed, and will restart in the remainder section otherwise (this is the *critical*  
 159 *section re-entry* property of [11]).

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

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

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

165 Additionally, we consider the following fairness property:

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

## 169 3 Failure Detector

170 Failure detectors were introduced by Chandra and Toueg in [5] as a way to circumvent  
 171 the impossibility to solve consensus in crash-prone asynchronous systems [10]. They are  
 172 distributed oracles which provide unreliable information on process crashes. The information

173 is unreliable in the sense that correct processes might be falsely suspected of having crashed,  
 174 and faulty processes might still be trusted after they crashed. Different classes of failure  
 175 detectors provide different properties on the reliability of the information provided to the  
 176 processes.

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

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

### 185 3.1 Failure Detectors for Mutual Exclusion

186 In [8], Delporte-Gallet et al. introduce the trusting failure detector  $\mathcal{T}$  and prove that it is the  
 187 weakest failure detector to solve fault-tolerant mutual exclusion in a system with a majority  
 188 of correct processes.  $\mathcal{T}$  provides each process with a list of trusted processes. It ensures that  
 189 faulty processes are eventually not trusted by any correct process (strong completeness),  
 190 that eventually all correct processes trust each other (eventually strong accuracy), and that  
 191 at all times, if  $\mathcal{T}$  stops trusting a process, then the process is crashed.

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

### 199 3.2 The $(\mathcal{T}+\Sigma^l)$ Failure Detector

200 The existing definition of  $(\mathcal{T}+\Sigma^l)$  is meant for static, known networks, and therefore we  
 201 need to provide a new definitions suited to unknown dynamic networks.

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

206 Additionally, the traditional properties of  $(\mathcal{T}+\Sigma^l)$  are expressed in terms of *correct* and  
 207 *faulty* processes. The version of  $(\mathcal{T}+\Sigma^l)$  used here was rewritten using the concepts of  
 208 *eventually up* and *eventually down* processes instead.

209 The  $(\mathcal{T}+\Sigma^l)$  failure detector provides each process  $p$  with a set of trusted process identit-  
 210 ies, denoted  $tq_p$ , and a flag denoted  $rdy_p$ .  $rdy_p$  is initially set to  $\perp$ , and then is changed to  $\top$   
 211 once the failure detector has gathered enough information to verify the live pairs intersection  
 212 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  
 213 say that process  $p$  trusts process  $q$  at time  $t$  if  $q \in tq_p^t$ , that  $p$  suspects  $q$  at time  $t$  if  $q \notin tq_p^t$ ,  
 214 and that process  $p$  is ready at time  $t$  if  $rdy_p^t = \top$ . The following properties must be verified.

- 215 ■ **Eventually strong accuracy:** Every *eventually up* process  $p$  is eventually trusted  
 216 forever by every process that is not *eventually down*.

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

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

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

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

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

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

#### 239 **4 Sufficiency of $(\mathcal{T} + \Sigma^l)$ to solve Fault-Tolerant Mutual Exclusion**

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

##### 242 **4.1 Algorithm Description**

243 In Algorithm 1, each process  $p$  which is in the try section issues a request of the form  
 244  $(round_p, p)$ , where  $round_p$  is the current round number of  $p$ . Requests are totally ordered  
 245 by their priority, which is defined as follows:  $priority(round_p, p) > priority(round_q, q) \Leftrightarrow$   
 246  $round_p < round_q$  **or**  $[round_p = round_q$  **and**  $p < q]$ .

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

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

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

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

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

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

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

---

**Algorithm 1** Solving RME with  $(\mathcal{T}+\Sigma^l)$ : 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:    $crit_p \leftarrow true; req_p \leftarrow false$ 
10: procedure EXIT SECTION
11:   wait for  $recovering_p = false$ 
12:    $crit_p \leftarrow false$ 
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  $crit_p = false$  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,  $crit_p, q$ )
28:   wait for  $update_p = \emptyset$ 
29:    $recovering_p \leftarrow false$ 
30:   CHECK REQUESTS()
31: when  $q$  added to  $tq_p$ 
32:   if  $req_p = true$  then SEND(REQUEST,  $round_p, q$ )
33: when  $q$  removed from  $tq_p$ 
34:    $grants_p \leftarrow grants_p \setminus \{q\}$ 
35:    $requests_p \leftarrow requests_p \setminus \{(*, q)\}$ 
36:    $update_p \leftarrow update_p \setminus \{q\}$ 
37:   if  $gid_p = q$  then
38:      $(gid_p, grnd_p) \leftarrow (-1, -1)$ 
39:   CHECK REQUESTS()
40: upon reception of REQUEST ( $round$ ) from
    $src$  do
41:    $requests_p \leftarrow requests_p \cup \{(round, src)\}$ 
42:    $last\_round_p[src] \leftarrow round$ 
43:   CHECK REQUESTS()
44: upon reception of GRANT () from  $src$  do
45:   if  $gid_p \neq -1$  and  $gid_p \neq p$  then
46:     SEND(REJECT,  $src$ )
47:   else if  $recovering_p = false$  then
48:      $grants_p \leftarrow grants_p \cup \{src\}$ 
49: upon reception of DONE () from  $src$  do
50:    $requests_p \leftarrow requests_p \setminus \{(*, src)\}$ 
51:    $(gid_p, grnd_p) \leftarrow (-1, -1)$ 
52:   CHECK REQUESTS()
53: upon reception of REJECT () from  $src$  do
54:    $(gid_p, grnd_p) \leftarrow (-1, -1)$ 
55:   CHECK REQUESTS()
56: upon reception of COMEBACK ( $crit\_src$ )
   from  $src$  do
57:    $requests_p \leftarrow requests_p \setminus \{(*, src)\}$ 
58:   if  $crit\_src = false$  and  $gid_p = src$ 
   then
59:      $(gid_p, grnd_p) \leftarrow (-1, -1)$ 
60:     CHECK REQUESTS()
61:   SEND(UPDATE,  $gid_p = src, last\_round_p[src], src \in$ 
    $grants_p, round_p, req_p, src$ )
62: upon reception of UPDATE ( $grant\_p, last\_rnd,$ 
    $grant\_src, round, req$ ) from  $src$  do
63:    $last\_round_p[src] \leftarrow round$ 
64:    $round_p \leftarrow \text{MAX}(round_p, last\_rnd)$ 
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:    $update_p \leftarrow update_p \setminus \{src\}$ 

```

---



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

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

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

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

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

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

274 The following types of messages are used by Algorithm 1:

275 REQUEST: asks for permission to enter CS. The message contains the round number of  
 276 the sender.

277 GRANT: grants permission to a process to enter CS.

278 DONE: notifies other processes that the sender just exited CS.

279 REJECT: warns that the sender already gave its permission to a process different from  
 280 the sender, thus preventing deadlocks.

281 COMEBACK: notifies other processes that the sender just recovered from a crash.

282 UPDATE: gives information to a recently recovered process about requests, previously  
 283 given permissions and its current round number.

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

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

296 When  $p$  receives a REQUEST message from process  $q$  (lines 40 – 43), it updates its know-  
 297 ledge about  $q$ 's round number and adds the new request to its  $requests_p$  set. It then calls  
 298 CHECK REQUESTS to decide if it should send a grant to the new requester.

299 When  $p$  receives a GRANT message from process  $q$ , if  $p$  already granted its permission  
 300 to some other process then it informs  $q$  by responding with a REJECT message to prevent  
 301 deadlocks (line 46). Otherwise, if  $p$  is not in the recovery phase, then it accepts  $q$ 's permission  
 302 by adding it to its  $grants_p$  set.

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

306 If  $p$  receives a DONE or REJECT message from process  $gid_p$ , it cancels the permission  
 307 granted to  $gid_p$  (lines 51 and 54) and calls CHECK REQUESTS. In the case of a DONE  
 308 message, the request from  $gid_p$  is also deleted from  $requests_p$  (line 50), since  $gid_p$  is not

309 requesting CS anymore. However, in the case of a REJECT, the request from  $gid_p$  is still  
 310 valid and must be kept, even if it is not the highest priority request.

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

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

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

## 330 4.2 Proof of correctness

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

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

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

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

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

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

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

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

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

350 **By crashing.** If  $p$  crashed between  $t_G$  and  $t'$ , then its  $gid_p$  got reset to  $-1$ . This also  
 351 means that  $p$  entered the recovery phase (lines 24 – 29) at some time  $t'' \in [t_G, t']$ . Since  
 352  $q_1 \in tq_p^{t_G}$ , then according to the trusting accuracy property of  $(\mathcal{T} + \Sigma^l)$ , either  $q_1$  crashed  
 353 before  $t''$  and will never recover (which is a contradiction), or  $q_1 \in tq_p^{t''}$ .  $p$  will therefore

## XX:10 Failure Detector for Mutual Exclusion in Dynamic Networks

354 send a COMEBACK message to  $q_1$  on line 27, and  $q_1$  will respond with a UPDATE message  
355 with the  $grant\_src$  parameter set to  $true$ , which will cause  $p$  to set its  $gid_p$  back to  $q_1$ .  
356 Since  $p$  cannot have sent a GRANT message while in the recovery phase (because of the test  
357 on line 17), then  $p$  cannot send the GRANT to  $q_2$  at time  $t$  which is a contradiction.

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

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

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

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

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

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

384 We can now prove the Claim 1 by contradiction.

385 **Proof.** Let  $p_1, p_2$  be two alive, distinct processes. Let us suppose that  $p_1$  enters CS at time  
386  $t_1$ , and  $p_2$  enters CS at time  $t_2$ . Let us suppose that neither process leaves CS until after the  
387 other process has entered it. According to the live pairs intersection property of  $(\mathcal{T} + \Sigma^l)$ ,  
388 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  
389 that  $q \in grants_{p_1}$  at time  $t_1$  and  $q \in grants_{p_2}$  at time  $t_2$ . There are two cases:

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

397 **Second case:**  $q = p_1$  or  $q = p_2$ . Without loss of generality, let us assume that  $q = p_1$ .  
398 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  
399 for  $q$  to send a GRANT message while in CS (because of the test on line 17), it follows that

400  $q$  sent the GRANT before entering CS. There could be a run in which  $p_2$  received the GRANT  
 401 immediately after it was sent, therefore adding  $q$  to  $grants_{p_2}$  before  $q$  entered CS, which is  
 402 in contradiction with Lemma 2. ◀

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

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

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

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

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

415 One of the following cases applies:

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

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

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

423 4) At time  $t$ ,  $gid_p = -1$ . When  $p$  calls CHECK REQUESTS, it will pass the test one line  
 424 17 since  $requests_p$  contains at least  $p$ 's request, and  $crit_p$  and  $recovering_p$  cannot be *true*  
 425 while in CS.  $p$  will then set  $gid_p$  to something different from  $-1$  on line 18.

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

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

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

440 3) During  $[t_R, t']$ ,  $p$  crashes before receiving enough GRANT messages to enter CS. When  
 441  $p$  recovers, its  $grants_p$  set is reinitialized and does not contain  $q$ . Since  $q$  was previously in  
 442  $tq_p$  and  $q$  is not eventually down, it follows from the trusting accuracy property of  $(\mathcal{T} + \Sigma^l)$   
 443 that  $q$  is still in  $tq_p$  after  $p$  recovers.  $p$  will therefore send a COMEBACK message to  $q$  on  
 444 line 27 with the  $crit\_src$  parameter set to *false*. When  $q$  receives the COMEBACK message,  
 445 it will set  $gid_q$  to  $-1$  on line 59. Note that because of the first in, first out property,  $q$  will

## XX:12 Failure Detector for Mutual Exclusion in Dynamic Networks

446 necessarily receive  $p$ 's request before the COMEBACK message. Additionally,  $p$  will receive  
447  $q$ 's GRANT message before  $q$ 's UPDATE message, and will ignore the grant because of the test  
448 on line 47. ◀

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

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

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

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

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

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

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

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

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

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

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

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

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

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

500 Since  $p$  is eventually up, according to the quorum readiness property of  $(\mathcal{T} + \Sigma^l)$ , the  
 501 eventually  $rdy_p = \top$ .

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

503 We can now prove Claim 2.

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

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

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

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

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

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

525 ◀

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

527 ▶ **Theorem 6 (Correctness).** *The Algorithm 1 solves the RME using  $(\mathcal{T} + \Sigma^l)$  in any unknown*  
 528 *dynamic environment.*

529 ▶ **Corollary 7 (Sufficiency).** *The  $(\mathcal{T} + \Sigma^l)$  failure detector is sufficient to solve the RME in*  
 530 *any unknown dynamic environment with partial memory loss.*

531 **5 Necessity of  $(\mathcal{T}+\Sigma^l)$  to solve Fault-Tolerant Mutual Exclusion**

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

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

540 The proof will make use of two algorithms, both of which share the following local  
 541 variables:

542  $\text{trust}_p \leftarrow \{p\}$  is the set of all processes that process  $p$  has heard of, that  $p$  does not  
 543 suspect. This variable is in stable storage.

544  $\text{start}_p \leftarrow \text{false}$  is a flag used to delay the start of the RME algorithm.

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

---

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


---

```

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

```

---

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

551 ► **Lemma 8.** *Provided that each eventually up process  $p$  eventually sets  $\text{start}_p$  to true,*  
 552 *Algorithm  $\mathcal{B}_{\text{RME}}$  solves the RME problem in  $\mathcal{M}_{\text{RME}}$ .*

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

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

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

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

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

---

**Algorithm 3** Reduction Algorithm  $T_{\mathcal{A}_{\text{RME}} \rightarrow (\mathcal{T} + \Sigma^t)}$ : code for process  $p$ 


---

```

1: procedure TASK 1
2:    $\mathcal{A}_{\text{RME}}.\text{TRY}(p)$ 
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:        $\underline{waitlist}_p \setminus \underline{donelist}_p = \emptyset$ 
10:       $\underline{donelist}_p \leftarrow \emptyset$ 
11:       $req_p \leftarrow true$ 
12:       $\mathcal{B}_{\text{RME}}.\text{TRY}$ 
13:       $\mathcal{B}_{\text{RME}}.\text{EXIT}$ 
14:       $req_p \leftarrow false$ 
15:      if  $\underline{trust}_p \cap \underline{crash}_p = \emptyset$  then
16:         $tq_p \leftarrow \underline{trust}_p$ 
17:         $rdy_p \leftarrow \top$ 
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:      $\mathcal{A}_{\text{RME}}.\text{TRY}(q)$ 
25:      $\mathcal{A}_{\text{RME}}.\text{EXIT}(q)$ 
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 ( $req, trust\_src$ )
34:     from  $src$  do
35:        $\underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$ 
36:       if  $req = true$  then  $\underline{waitlist}_p \leftarrow$ 
37:          $\underline{waitlist}_p \cup \{src\}$ 
38:       else
39:          $\underline{waitlist}_p \leftarrow \underline{waitlist}_p \setminus \{src\}$ 
40:          $\underline{donelist}_p \leftarrow \underline{donelist}_p \cup \{src\}$ 
41:   upon reception of QUORUM ( $trust\_src, crash\_src$ )
42:     from  $src$  do
43:        $\underline{trust}_p \leftarrow \underline{trust}_p \cup trust\_src$ 
44:        $\underline{crash}_p \leftarrow \underline{crash}_p \cup crash\_src$ 
45:       if  $rdy_p = \perp$  then
46:          $tq_p \leftarrow \underline{trust}_p \setminus \underline{crash}_p$ 

```

---



## XX:16 Failure Detector for Mutual Exclusion in Dynamic Networks

567  $tq_p \leftarrow \emptyset$ : the output of the  $(\mathcal{T} + \Sigma^l)$  failure detector, which verifies the trusting and  
568 quorum properties.

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

570  $waitlist_p \leftarrow \emptyset$ : the set of processes to which  $p$  must yield priority for CS. This is used  
571 to ensure starvation freedom. Note that this variable is in stable storage.

572  $donelist_p \leftarrow \emptyset$ : the set of processes to which  $p$  already yielded priority for CS. This  
573 prevents  $p$  from always being passed over for CS access.

574 Algorithm 3 initially starts two tasks in parallel: TASK 1 and TASK 2. Later on when  
575 process  $p$  receives knowledge of other processes, it starts a new task for each process  $q$   
576 (denoted TASK 3 +  $q$ ).

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

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

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

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

596 When a process  $p$  receives a QUORUM message, it updates its local  $trust_p$  and  $crash_p$   
597 information and, if  $rdy_p$  is currently  $\perp$  (and therefore  $p$  is not currently trying to verify the  
598 live pairs intersection property), then  $p$  updates its  $tq_p$ .

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

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

602 ► **Lemma 10** (Crashed completeness). *A process can only be added to  $crash_p$  if it crashed*  
603 *forever.*

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

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

607 **Proof.** Let  $p$  be an eventually down process, and  $q$  be a process that is not eventually down.  
608 Note that by construction, a process can never be added to  $tq_q$  without being added to  
609  $trust_q$  first. There are two cases:

610  $p$  was never added to  $trust_q$ . Then the property is immediately verified.

611  $p$  was added to  $trust_q$ . Let  $r$  be some eventually up process. Eventually,  $q$  will send an  
612 ALIVE message to  $r$  with its  $trust_q$ . As a result,  $r$  will eventually add  $p$  to its  $trust_r$ .  $r$  will

613 then start TASK 3 +  $p$ . After  $p$  crashes forever, eventually  $r$  will reach line 26 and add  $p$  to  
 614  $\underline{crash}_r$ .

615 Let  $t_1$  be a time after which all eventually down processes have crashed. Let  $t_2 \geq t_1$   
 616 be a time after which there are no more messages sent by eventually down processes in the  
 617 system. After  $t_2$ , neither  $q$  nor  $r$  will ever add an eventually down process into their  $\underline{trust}$  set  
 618 again. According to Lemma 9,  $r$  will then eventually remove all eventually down processes  
 619 from  $\underline{trust}_r$  on line 21. Since according to Lemma 10 only eventually down processes can  
 620 be in  $\underline{crash}_r$ , after this time  $r$  will always pass the test on line 15 and therefore  $r$  will send  
 621 a QUORUM message to  $q$  infinitely often.

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

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

629 **Proof.** Let  $p$  be an eventually up process, and  $q$  a process that is not eventually down.  
 630 Eventually,  $q \in \underline{known}_p$ . According to the liveness property of RME,  $p$  will eventually enter  
 631 its own CS and send an ALIVE message to  $q$  on line 6. When  $q$  receives the message, it will  
 632 add  $p$  to its  $\underline{trust}_q$  set on line 34. It follows from Lemma 10 that  $p$  will never be in  $\underline{crash}_q$ .  
 633 According to the proof for Claim 4,  $q$  will update its  $tq_p$  infinitely often with  $\underline{trust}_q$ , either  
 634 on line 16 or on line 43. As a result,  $p \in tq_q$  forever. ◀

635 ▶ **Claim 6 (Trusting accuracy).** By construction, the only way that a process can be removed  
 636 from  $tq_p$  is by being added to  $\underline{crash}_p$ . The proof then follows directly from Lemma 10.

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

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

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

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

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

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

654 Note that  $tq_p$  is set from  $\underline{trust}_p$  on line 16 after  $p$  recently went through the global try,  
 655 critical, and exit sections with  $\mathcal{B}_{\text{RME}}$  on lines 12 – 13. By construction, every process from  
 656 which  $p$  received information (even indirectly) in  $\mathcal{B}_{\text{RME}}$  since last entering the try section is

657 in  $\text{trust}_p$  at that time. Observe also that the only way to remove a process identity from  
 658  $\text{trust}_p$  is on line 21, which cannot be reached between lines 12 and 16.

659 Let  $p_1$  and  $p_2$  be two processes, and let  $t$  be some time at which both are alive. Then  
 660 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  
 661 follows from Lemma 11 that  $\text{trust}_{p_1}$  at time  $t_1$  and  $\text{trust}_{p_2}$  at time  $t_2$  intersect. ◀

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

663 ▶ **Theorem 12 (Correctness).** *The Algorithm 3 implements  $(\mathcal{T}+\Sigma^l)$  in  $\mathcal{M}_{\text{RME}}$ .*

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

## 666 Conclusion

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

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

---

## 679 References

- 680 1 Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for dis-  
 681 tributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, February 1991.
- 682 2 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in  
 683 the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.
- 684 3 Vibhor Bhatt, Nicholas Christman, and Prasad Jayanti. Extracting quorum failure de-  
 685 tectors. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed  
 686 Computing, PODC 2009*, pages 73–82, 2009.
- 687 4 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-  
 688 varying graphs and dynamic networks. *IJPEDES*, 27(5):387–408, 2012.
- 689 5 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed  
 690 systems. *Journal of the ACM*, 43(2):225–267, 1996.
- 691 6 Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A fault tolerant algorithm for distributed  
 692 mutual exclusion. In *Ninth Symposium on Reliable Distributed Systems, SRDS 1990*, pages  
 693 146–154, 1990.
- 694 7 Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Tight failure detection  
 695 bounds on atomic object implementations. *JACM*, 57(4), 2010.
- 696 8 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mu-  
 697 tual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and  
 698 Distributed Computing*, 65(4):492–505, apr 2005.
- 699 9 E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications  
 700 of the ACM*, 8(9):569, 1965.

- 701 **10** Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed  
702 consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- 703 **11** Wojciech M. Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic  
704 time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*,  
705 *PODC 2017*, pages 211–220, 2017.
- 706 **12** Wojciech M. Golab and Aditya Ramaraju. Recoverable mutual exclusion: [extended ab-  
707 stract]. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Comput-*  
708 *ing, PODC 2016*, pages 65–74, 2016.
- 709 **13** Prasad Jayanti and Anup Joshi. Recoverable FCFS mutual exclusion with wait-free re-  
710 recovery. In *31st International Symposium on Distributed Computing, DISC 2017*, pages  
711 30:1–30:15, 2017.
- 712 **14** S. Nishio, E. Manning, and K. Li. A resilient mutual exclusion algorithm for computer  
713 networks. *IEEE Transactions on Parallel and Distributed Systems*, 1:344–356, 07 1990.
- 714 **15** Thibault Rieutord, Luciana Arantes, and Pierre Sens. Détecteur de défaillances minimal  
715 pour le consensus adapté aux réseaux inconnus. In *Algotel*, 2015.
- 716 **16** Julien Sopena, Luciana Bezerra Arantes, and Pierre Sens. Performance evaluation of a fair  
717 fault-tolerant mutual exclusion algorithm. In *25th IEEE Symposium on Reliable Distributed*  
718 *Systems (SRDS 2006)*.