Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 65 (2005) 492–505
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Mutual exclusion in asynchronous systems with failure detectors☆

Carole Delporte-Gallet[a], Hugues Fauconnier[a], Rachid Guerraoui[b], Petr Kouznetsov[b,*]

[a]*Laboratoire d'Informatique Algorithmique, Fondements et Applications, University of Paris VII, France*
[b]*School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland*

Received 30 August 2002; received in revised form 29 November 2004; accepted 29 November 2004

Abstract

This paper considers the *fault-tolerant mutual exclusion* problem in a message-passing asynchronous system and determines the weakest failure detector to solve the problem, given a majority of correct processes. This failure detector, which we call the *trusting* failure detector, and which we denote by $\mathcal{T}$, is strictly weaker than the perfect failure detector $\mathcal{P}$ but strictly stronger than the eventually perfect failure detector $\diamond\mathcal{P}$. The paper shows that a majority of correct processes is necessary to solve the problem with $\mathcal{T}$. Moreover, $\mathcal{T}$ is also the weakest failure detector to solve the fault-tolerant *group* mutual exclusion problem, given a majority of correct processes.

Keywords: Mutual exclusion; Fault-tolerance; Asynchronous system; Failure detectors; Group mutual exclusion

1. Introduction

1.1. Background

This paper addresses the fault-tolerant mutual exclusion problem in a distributed message-passing system where channels are reliable and processes can fail by crashing. The mutual exclusion problem [6,13,14,19] involves managing access to a single, indivisible resource that can be accessed by at most one process at a time (*mutual exclusion* property). The process accessing the resource is said to be in its *critical section* (CS). In the *fault-tolerant mutual exclusion* problem, we require that if a correct process (i.e., a process that takes an infinite number of steps of an algorithm assigned to it) wants to enter its CS, then there eventually will be *some* correct process in its CS (*progress* property), even if some process crashes (stops taking steps) while in its CS.

Evidently, the problem cannot be solved deterministically in a crash-prone asynchronous system without any information about failures: there is no way to determine that a process in its CS is crashed or just slow. (We do not consider here probabilistic mutual exclusion algorithms [5,8]. We also do not restrict ourselves to particular scenarios in which, for instance, no process can crash outside its remainder section). Clearly, no deterministic algorithm can guarantee fault-tolerant progress and mutual exclusion simultaneously. In this sense, the problem is related to the famous impossibility result that consensus cannot be solved deterministically in an asynchronous system that is subject to even a single crash failure [7].

1.2. Failure detectors

To circumvent the impossibility of consensus, Chandra and Toueg [4] introduced the notion of *failure detector*. Informally, a failure detector is a distributed oracle that gives (possibly incorrect) hints about which processes have crashed so far. Each process has access to a local *failure detector module* that monitors other processes in the system. In [4], it is shown that a rather weak failure detector $\diamond\mathcal{W}$ is sufficient to solve consensus in an asynchronous system

☆ This work is partially supported by the Swiss National Science Foundation (project number 510-207).
* Corresponding author. Fax: +41 21 693 7570.
E-mail addresses: carole.delporte@liafa.jussieu.fr (C. Delporte-Gallet), hugues.fauconnier@liafa.jussieu.fr (H. Fauconnier), rachid.guerraoui@epfl.ch (R. Guerraoui), petr.kouznetsov@epfl.ch (P. Kouznetsov).

0743-7315/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2004.11.008

with a majority of correct processes, and that $\diamond\mathcal{W}$ can be implemented using partial synchrony assumptions. In [3], it is shown in a precise sense that $\diamond\mathcal{W}$ is also necessary to solve consensus, given a majority of correct processes. In short, $\diamond\mathcal{W}$ is the *weakest* failure detector to solve consensus.

1.3. Trusting failure detector $\mathcal{T}$

A natural question follows: what is the *weakest* failure detector to solve the fault-tolerant mutual exclusion problem? Traditionally, mutual exclusion algorithms either assume that no process crashes outside its remainder section [6,13,14,19,15,21], or suppose that (1) every crash is eventually detected by every correct process and (2) no correct process is suspected [1,18]: the conjunction of (1) and (2) is equivalent to the assumption of the *perfect* failure detector $\mathcal{P}$ [4]. In other words, perfect information about failures is *sufficient* to solve the fault-tolerant mutual exclusion problem. But is $\mathcal{P}$ *necessary*? We show that the answer is "no": we can solve the problem using the *trusting* failure detector $\mathcal{T}$, a new failure detector we introduce here, which is strictly weaker than $\mathcal{P}$ (but strictly stronger than $\diamond\mathcal{P}$, the *eventually perfect* failure detector of [4]).

Roughly speaking, failure detector $\mathcal{T}$ satisfies the following properties: (1) there is a time after which $\mathcal{T}$ *trusts* every correct process, (2) there is a time after which $\mathcal{T}$ does not trust any crashed process, and (3) at all times, if $\mathcal{T}$ stops trusting a process, then the process is crashed. Failure detector $\mathcal{T}$ might however trust temporarily a crashed process as well as not trust temporarily a correct process. Intuitively, $\mathcal{T}$ can thus make mistakes and algorithms using $\mathcal{T}$ are, from a practical point of view, more resilient than those using $\mathcal{P}$.

The algorithm we present here to show that $\mathcal{T}$ is sufficient to solve fault-tolerant mutual exclusion assumes a majority of correct processes and is inspired by the well-known Bakery algorithm of Lamport [13,14]: a process that wishes to enter its CS first passes a guard (gets trusted by *some* correct process), then draws a ticket and is served in the order of its ticket number. Failure detector $\mathcal{T}$ guarantees that a crash of the process will be eventually detected by every correct process in the system. We show that, in addition to mutual exclusion and progress, our algorithm guarantees also a fairness property, ensuring that the only excuse for not granting the access to a CS requested by a correct process is the permanent stay of some correct process in its CS (*starvation-freedom* property).

We also show that $\mathcal{T}$ is *weaker* than any failure detector $\mathcal{D}$ sufficient to solve the problem ($\mathcal{T}$ provides at least as much information about failures as $\mathcal{D}$). Intuitively, this stems from the fact that, if a process i in its CS does not execute the exit protocol, another process can enter its CS only if it is sure that i is crashed. We present an algorithm that extracts the information provided by $\mathcal{T}$ from any algorithm that solves fault-tolerant mutual exclusion.

1.4. Contributions

- We show that $\mathcal{T}$ is indeed the weakest failure detector to solve the problem in a system with a majority of correct processes. We show also that the majority is actually necessary for any fault-tolerant mutual exclusion algorithm using $\mathcal{T}$.
- Then we address the question: what if we do not make the assumption of a majority of correct processes? Is $\mathcal{P}$ necessary? We show that it is still not: we present a failure detector $\mathcal{T}+\mathcal{S}$ (where $\mathcal{S}$ is the *strong* failure detector of [4]) which is strictly weaker than $\mathcal{P}$ and which is sufficient (but possibly not necessary) to solve the problem even with an arbitrary number of failures.
- Finally, we turn our attention to group mutual exclusion [9,11,12,22], a recent generalization of mutual exclusion and we show that $\mathcal{T}$ is the weakest to solve fault-tolerant group mutual exclusion (with a majority of correct processes). In other words, we show that the problem is equivalent to fault-tolerant mutual exclusion in an asynchronous system augmented with failure detectors and the assumption of a majority of correct processes. Analogously, failure detector $\mathcal{T}+\mathcal{S}$ is sufficient to solve fault-tolerant group mutual exclusion in an asynchronous system with an arbitrary number of failures.

1.5. Roadmap

The rest of the paper is organized as follows. Section 2 overviews the system model. Section 3 defines the fault-tolerant mutual exclusion problem. Section 4 introduces the trusting failure detector $\mathcal{T}$. Sections 5 and 6 show that $\mathcal{T}$ is, respectively, necessary and sufficient to solve the problem. Section 7 discusses the bounds on the number of correct processes necessary to solve the problem with $\mathcal{T}$ and introduces a failure detector $\mathcal{T}+\mathcal{S}$ which is sufficient to solve the problem without a majority of correct processes. Section 8 generalizes our result to the group mutual exclusion problem. Section 9 discusses the performance cost of the resilience provided by $\mathcal{T}$ and Section 10 concludes the paper with some practical remarks.

2. The model

We consider in this paper the traditional crash-prone asynchronous message passing system model augmented with the failure detector abstraction [4,3].

2.1. System

The system consists of a set of n processes $\Pi = \{1, \ldots, n\}$ ($n > 1$). Every pair of processes is connected by a reliable channel. Processes communicate by message passing. To simplify the presentation of our model, we assume the existence of a discrete global clock. This is a fictional device:

the processes have no direct access to it. (More precisely, the information about global time can come *only* from failure detectors.) We take the range $\mathbb{T}$ of the clock's ticks to be the set of natural numbers and 0 ($\mathbb{T} = \{0\} \cup \mathbb{N}$).

2.2. Failures and failure patterns

Processes are subject to *crash* failures. A *failure pattern* F is a function from the global time range $\mathbb{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed by time t. Once a process crashes, it does not recover, i.e., $\forall t < t' : F(t) \subseteq F(t')$. We define $correct(F) = \Pi - \cup_{t \in \mathbb{T}} F(t)$, the set of *correct* processes. A process $i \notin F(t)$ is said to be *alive* at time t. A process $i \in F(t)$ is said to be *crashed* at time t. Processes in $\Pi - correct(F)$ are called *faulty* in F. We do not consider Byzantine failures: a process either correctly executes the algorithm assigned to it, or crashes and stops forever executing any action. An environment $\mathcal{E}$ is a set of possible failure patterns. In this paper, we consider environments of the type $\mathcal{E}_f$ that consists of all failure patterns in which up to f processes can crash. We assume that $0 < f < n$: at least one process might crash and at least one process is correct.

2.3. Failure detectors

A *failure detector history H with range* $\mathcal{R}$ is a function from $\Pi \times \mathbb{T}$ to $\mathcal{R}$. $H(i, t)$ is the value of the failure detector module of process i at time t. A *failure detector* $\mathcal{D}$ is a function that maps each failure pattern to a *set* of failure detector histories (usually defined by a set of requirements that these histories should satisfy). $\mathcal{D}(F)$ denotes the set of possible failure detector histories with range $\mathcal{R}_{\mathcal{D}}$ permitted by $\mathcal{D}$ for the failure pattern F. Processes use a failure detector $\mathcal{D}$ in the sense that every process i has a failure detector module $\mathcal{D}_i$ that provides i with information about the failures in the system. We do not make any assumption a priori on the range of a failure detector.

Among the failure detectors defined in [4], we consider *perfect*, *eventually perfect* and *strong* failure detectors, each one outputs at every process a set of processes that the process currently *suspects* to have crashed. These failure detectors are defined by *completeness* and *accuracy* properties:

Perfect ($\mathcal{P}$): Strong completeness (i.e., every crashed process is eventually suspected by every correct process) and strong accuracy (i.e., no process is suspected before it crashes).

Eventually perfect ($\diamond\mathcal{P}$): Strong completeness and eventual strong accuracy (i.e., there is a time after which no correct process is ever suspected).

Strong ($\mathcal{S}$): Strong completeness and weak accuracy (i.e., some correct process is never suspected).

For any failure pattern F, $\mathcal{P}(F)$, $\diamond\mathcal{P}(F)$ and $\mathcal{S}(F)$ denote the sets of *all* histories satisfying the corresponding properties.

2.4. Algorithms, configurations, schedules, and runs

Following [3,4], we model the asynchronous communication channels as a message buffer which contains messages not yet received by their destinations. An *algorithm* A is a collection of n (possibly infinite state) deterministic automata, one for each process. $A(i)$ denotes the automaton on which process i is running algorithm A. Computation proceeds in *steps* of A. In each step of A, process i performs atomically the following three actions: (1) i receives a single message addressed to i from the message buffer, or a null message, denoted λ; (2) i queries and receives a value from its failure detector module; (3) i changes its state and sends a message to a single process according to the automaton $A(i)$. Note that the received message is chosen *nondeterministically* from the messages in the message buffer destined to i, or the null message λ.

A *configuration* defines the current state of each process in the system and the set of messages currently in the message buffer. Initially, the message buffer is empty. A step (i, m, d, A) of an algorithm A is uniquely determined by the identity of the process i that takes the step, the message m received by i during the step (m might be the null message λ), and the failure detector value d seen by i during the step. We say that a step $e = (i, m, d, A)$ *is applicable* to a configuration C if and only if $m = \lambda$ or m is in the message buffer of C. For a step e applicable to C, $e(C)$ denotes the unique configuration that results from applying e to C.

A *schedule* S of algorithm A is a (finite or infinite) sequence of steps of A. $S_\perp$ denotes the empty schedule. We say that a schedule S *is applicable to a configuration* C if and only if (a) $S = S_\perp$, or (b) $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc. For a finite schedule S applicable to C, $S(C)$ denotes the unique configuration that results from applying S to C.

A *partial run of algorithm A in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, S is a *finite* schedule of A, and $T \subseteq \mathbb{T}$ is a *finite* list of increasing time values (indicating when each step S occurred) such that $|S| = |T|$, S is applicable to I, and for all $k \leqslant |S|$, if $S[k] = (i, m, d, A)$ then: (1) i has not crashed by time $T[k]$, i.e., $i \notin F(T[k])$ and (2) d is the value of the failure detector module of i at time $T[k]$, i.e., $d = H_{\mathcal{D}}(i, T[k])$.

A *run of algorithm A in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where $F \in \mathcal{E}$ is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a failure detector history, I is an initial configuration of A, S is an *infinite* schedule of A, and $T \subseteq \mathbb{T}$ is an *infinite* list of increasing time values indicating when each step S occurred. In addition to satisfying properties (1) and (2) of a partial run, a run R should guarantee that (3) every correct process in F takes an infinite number of steps in S and eventually receives every message sent to it (this conveys the reliability of the communication channels). (In fact, the sufficient part of this paper holds

even with a weaker guarantee such as "every correct process eventually receives every message sent to it by any *correct* process".)

2.5. Problems and solvability

A *problem* is a set of runs (usually defined by a set of properties that these runs should satisfy). An algorithm A *solves a problem* M *in an environment* $\mathcal{E}$ *using a failure detector* $\mathcal{D}$ if all the runs of A in $\mathcal{E}$ using $\mathcal{D}$ are in M (i.e., they satisfy the properties of M). We say that a failure detector $\mathcal{D}$ *solves problem* M in $\mathcal{E}$ if there is an algorithm A which solves M in $\mathcal{E}$ using $\mathcal{D}$.

Let M and M' be any two problems and $\mathcal{E}$ be any environment. If for any algorithm A' that solves M' in $\mathcal{E}$, there is a transformation algorithm of A' into an algorithm A, $R_{A'\to A}$ such that A solves M in $\mathcal{E}$, we say that M' *is harder than* M *in* $\mathcal{E}$. If M' is harder than M in $\mathcal{E}$ and M is harder than M' in $\mathcal{E}$, we say that M and M' are *equivalent in* $\mathcal{E}$.

2.6. Weakest failure detector

If, for failure detectors $\mathcal{D}$ and $\mathcal{D}'$, there is an algorithm $R_{\mathcal{D}'\to\mathcal{D}}$ that transforms $\mathcal{D}'$ into $\mathcal{D}$ in environment $\mathcal{E}$ ($R_{\mathcal{D}'\to\mathcal{D}}$, called a *reduction* algorithm, emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$), we say that $\mathcal{D}$ *is weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$, and we write $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$. If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ but $\mathcal{D}' \not\preceq_{\mathcal{E}} \mathcal{D}$, we say that $\mathcal{D}$ *is strictly weaker than* $\mathcal{D}'$ *in* $\mathcal{E}$, and we write $\mathcal{D} \prec_{\mathcal{E}} \mathcal{D}'$. Note that $R_{\mathcal{D}'\to\mathcal{D}}$ does not need to emulate *all* histories of $\mathcal{D}$; it is required that all the failure detector histories it emulates be histories of $\mathcal{D}$.

We say that a failure detector $\mathcal{D}$ is *the weakest failure detector to solve a problem* M *in an environment* $\mathcal{E}$ if the following conditions are satisfied: (sufficiency) $\mathcal{D}$ solves M in $\mathcal{E}$ and (necessity) if a failure detector $\mathcal{D}'$ solves M in $\mathcal{E}$, then $\mathcal{D}$ is weaker than $\mathcal{D}'$ in $\mathcal{E}$.

3. The fault-tolerant mutual exclusion problem

In defining the *fault-tolerant mutual exclusion* problem (from now on—FTME) we use the terms of [16, Chapter 10]. The FTME problem involves the allocation of a single, indivisible, resource among n processes. An alive (not crashed) process with access to the resource is said to be in its *critical section* (*CS*). When a process is not involved in any way with the resource, it is said to be in its *remainder section*. To gain access to its critical section, a process executes a *trying protocol*, and after the process is done with the resource, it executes an *exit protocol*. This procedure can be repeated, so each process i cyclically moves from its remainder section (rem_i) to its *trying section* (try_i), then to its critical section (crit_i), then to its *exit section* (exit_i), and then back again to rem_i. We assume that every process i is *well-formed*, i.e., i does not violate the cyclic order of execution: $\mathsf{rem}_i, \mathsf{try}_i, \mathsf{crit}_i, \mathsf{exit}_i, \ldots$.[1]

A mutual exclusion algorithm defines trying protocol try_i and exit protocol $exit_i$ for every process i. (We do not restrict the process behavior in the critical and remainder sections.) We say that the algorithm solves the FTME problem if, under the assumption that every process is well-formed, any run of the algorithm satisfies the following properties:

Mutual exclusion: No two different processes are in their CSs at the same time.

Progress: (1) If a correct process is in its trying section, then at some time later some correct process is in its CS.

(2) If a correct process is in its exit section, then at some time later it enters its remainder section.

We will show in Sections 5 and 6 that, in an environment with a majority of correct processes, any algorithm that solves the FTME problem can be transformed into an algorithm satisfying not only the properties above but also the following fairness property:

Starvation freedom: If no process stays forever in its CS, then every correct process that reaches its trying section eventually enters its CS.

Note that mutual exclusion is a *safety* property while progress and starvation freedom are *liveness* properties.

4. The trusting failure detector

This section introduces a new failure detector that we call the *trusting* failure detector and we denote by $\mathcal{T}$. The range of $\mathcal{T}$ is $\mathcal{R}_{\mathcal{T}} = 2^{\Pi}$. Let $H_{\mathcal{T}}$ be any history of $\mathcal{T}$. $H_{\mathcal{T}}(i,t)$ represents the set of processes that process i *suspects* (i.e., considers to have crashed) at time t. We say that process i *trusts process* j *at time* t if $j \notin H_{\mathcal{T}}(i,t)$.

For every failure pattern F, $\mathcal{T}(F)$ is defined by the set of *all* histories $H_{\mathcal{T}}$ that satisfy the following properties:

Strong completeness: Eventually, every crashed process is permanently suspected by every correct process. That is

$$\forall i \notin correct(F), \exists t : \forall t' > t, \forall j \in correct(F), \; i \in H_{\mathcal{T}}(j,t').$$

Eventual strong accuracy: Eventually, no correct process is suspected by any correct process. That is

$$\forall i \in correct(F), \exists t : \forall t' > t, \forall j \in correct(F), \; i \notin H_{\mathcal{T}}(j,t').$$

[1] An alternative stronger definition of the problem can allow a process to be initially in its CS. Clearly, the perfect failure detector $\mathcal{P}$ is necessary for this problem. We instead follow the original definition of [16] where the competition between processes for the critical section is "fair", since none of them can usurp the CS from the very beginning.

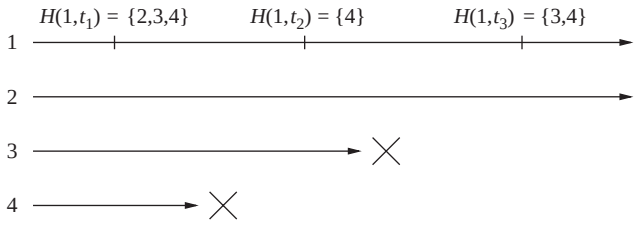

Fig. 1. Failure detection scenario for $\mathcal{T}$.

Trusting accuracy: Every process j that is suspected by a process i after being trusted by i is crashed. That is:

$$\forall i, j, t < t' : j \notin H_{\mathcal{T}}(i,t) \wedge j \in H_{\mathcal{T}}(i,t') \Rightarrow j \in F(t').$$

Fig. 1 depicts a possible scenario of failure detection with $\mathcal{T}$. We consider the system $\Pi = \{1, 2, 3, 4\}$. Initially, the failure detector module at process 1 outputs $\{2, 3, 4\}$: $H(1, t_1) = \{2, 3, 4\}$, i.e., process 1 trusts only itself. At time $t_2 > t_1$, processes 2 and 3 also get trusted by process 1: $H(1, t_2) = \{4\}$. Process 3 crashes and at some time later is not trusted anymore by process 1: $\forall t \geqslant t_3$, $H(1, t) = \{3, 4\}$. Note that process 1 never trusts process 4.

Now we identify the position of $\mathcal{T}$ in the hierarchy of failure detector s introduced in [4]. We show that, in any environment $\mathcal{E}_f$ with $0 < f < n$, $\diamond\mathcal{P}$ is strictly weaker than $\mathcal{T}$, and $\mathcal{T}$ is strictly weaker than $\mathcal{P}$. The "weaker" parts of the proofs follow directly form the definition of $\mathcal{T}$. The "strictly" parts of the proofs are done by contradiction: we assume that a reduction algorithm $R_{\mathcal{T} \to \mathcal{P}}$ (respectively, $R_{\diamond\mathcal{P} \to \mathcal{T}}$) exists and expose a run of the algorithm that violates some properties of $\mathcal{P}$ (respectively, $\mathcal{T}$).

Proposition 1. *$\mathcal{T} \prec_{\mathcal{E}_f} \mathcal{P}$, in any environment $\mathcal{E}_f$ with $0 < f < n$.*

Proof. (a) Clearly, $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{P}$ in any environment $\mathcal{E}$: $\mathcal{P}$ satisfies all properties of $\mathcal{T}$. Indeed, strong completeness is given for free, eventual strong accuracy is implied by strong accuracy of $\mathcal{P}$. Trusting accuracy follows from the fact that $\mathcal{P}$ guarantees that any suspected process is crashed.

(b) Now we show that $\mathcal{P}$ is not weaker than $\mathcal{T}$. Intuitively, it follows from the fact that $\mathcal{T}$ is allowed to make mistakes about processes (see the scenario of Fig. 1).

By contradiction, assume that there exists a reduction algorithm $R_{\mathcal{T} \to \mathcal{P}}$ that, for any failure pattern $F \in \mathcal{E}_f$ and any history $H_{\mathcal{T}} \in \mathcal{T}(F)$, constructs a history $H_{\mathcal{P}}$ such that $H_{\mathcal{P}} \in \mathcal{P}(F)$.

Consider failure pattern $F_1 \in \mathcal{E}_f$ such that $F_1(0) = \{j\}$, $correct(F_1) = \Pi - \{j\}$ (the only faulty process j is initially crashed) and take a history $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$ such that $H^1_{\mathcal{T}}(i, t) = \{j\}$, $\forall i \neq j, \forall t \in \mathbb{T}$ (remember that we consider an environment where at least one process can crash and at least one process is correct). Consider run $R_1 = \langle F_1, H^1_{\mathcal{T}}, I, S_1, T\rangle$ of $R_{\mathcal{T} \to \mathcal{P}}$ that outputs a history $H^1_{\mathcal{P}} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists k_0 \in \mathbb{N}, \exists l \in \Pi - \{j\}$: $H^1_{\mathcal{P}}(l, T[k_0]) = \{j\}$.

Consider failure pattern $F_2 \in \mathcal{E}_f$ such that $correct(F_2) = \Pi$ (F_2 is failure-free) and define a history $H^2_{\mathcal{T}}$ such that $\forall i \in \Pi$ and $\forall t \in \mathbb{T}$:

$$H^2_{\mathcal{T}}(i,t) = \begin{cases} \{j\}, & t \leqslant T[k_0], \\ \emptyset, & t > T[k_0]. \end{cases}$$

Note that $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$, and $\forall t \leqslant T[k_0]$, $\forall i \in \Pi - \{j\}$: $H^1_{\mathcal{T}}(i,t) = H^2_{\mathcal{T}}(i,t)$. Consider run $R_2 = \langle F_2, H^2_{\mathcal{T}}, I, S_2, T\rangle$ of $R_{\mathcal{T} \to \mathcal{P}}$ such that $S_1[k] = S_2[k]$, $\forall k \leqslant k_0$ (processes take the same steps in R_1 and R_2 up to time $T[k_0]$). Let R_2 outputs a history $H^2_{\mathcal{P}} \in \mathcal{P}(F_2)$. Since partial runs of R_1 and R_2 for $t \leqslant T[k_0]$ are identical, the resulting history $H^2_{\mathcal{P}}$ is such that $H^2_{\mathcal{P}}(l, T[k_0]) = \{j\}$, for some $l \in \Pi - \{j\}$. But process j is alive at $T[k_0]$ in F_2, i.e., the strong accuracy of $\mathcal{P}$ is violated—a contradiction.

Thus, $\mathcal{T} \prec_{\mathcal{E}_f} \mathcal{P}$. □

Proposition 2. *$\diamond\mathcal{P} \prec_{\mathcal{E}_f} \mathcal{T}$, in any environment $\mathcal{E}_f$ with $0 < f < n$.*

Proof. Clearly, $\diamond\mathcal{P} \preceq_{\mathcal{E}} \mathcal{T}$ in any environment $\mathcal{E}$: by definition, every $\mathcal{T}$ satisfies strong completeness and eventual strong accuracy.

Now we show that $\mathcal{T}$ is not weaker than $\diamond\mathcal{P}$. Intuitively, it follows from the fact that $\mathcal{T}$ is allowed to make only a *bounded* number of mistakes, while the number of mistakes $\diamond\mathcal{P}$ can make is unbounded.

By contradiction, assume that there exists a reduction algorithm $R_{\diamond\mathcal{P} \to \mathcal{T}}$ that, for any failure pattern $F \in \mathcal{E}_f$ and any history $H_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F)$, constructs a history $H_{\mathcal{T}}$ such that $H_{\mathcal{T}} \in \mathcal{T}(F)$.

Consider a failure-free pattern $F_1 \in \mathcal{E}_f$ ($correct(F_1) = \Pi$) and take $H^1_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_1)$ such that $\forall i, \forall t \in \mathbb{T}$: $H^1_{\diamond\mathcal{P}}(i,t) = \emptyset$. Consider a run $R_1 = \langle F_1, H^1_{\diamond\mathcal{P}}, I, S_1, T\rangle$ of $R_{\diamond\mathcal{P} \to \mathcal{T}}$ that outputs a history $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$. By the eventual strong accuracy property of $\mathcal{T}$, $\exists k_0 \in \mathbb{N}$, such that $\forall k \geqslant k_0$ and $\forall i \in \Pi$: $H^1_{\mathcal{T}}(i, T[k]) = \emptyset$.

Now consider a failure pattern $F_2 \in \mathcal{E}_f$ such that $correct(F_2) = \Pi - \{j\}$ in which j crashes at time $T[k_0]+1$. Take a history $H^2_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_2)$ such that for all $t \in \mathbb{T}$ and $i \in \Pi$:

$$H^2_{\diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\diamond\mathcal{P}}(i,t), & t \leqslant T[k_0], \\ \{j\}, & t > T[k_0]. \end{cases}$$

Now consider a run $R_2 = \langle F_2, H^2_{\diamond\mathcal{P}}, I, S_2, T\rangle$ of $R_{\diamond\mathcal{P} \to \mathcal{T}}$ that outputs a history $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$. Assume that $S_1[k] = S_2[k]$, $\forall k \leqslant k_0$. Clearly, for all $i \in \Pi$, $H^2_{\mathcal{T}}(i, T[k_0]) = \emptyset$. By the strong completeness property of $\mathcal{T}$, there exists a time $k_1 > k_0$ such that $\forall i \neq j$: $H^2_{\mathcal{T}}(i, T[k_1]) = \{j\}$.

Now we construct a history $H^3_{\diamond\mathcal{P}}$ such that for all $t \in \mathbb{T}$ and $i \in \Pi$:

$$H^3_{\diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\diamond\mathcal{P}}(i,t), & t \leqslant T[k_0], \\ H^2_{\diamond\mathcal{P}}(i,t), & T[k_0] < t \leqslant T[k_1], \\ \emptyset, & t > T[k_1]. \end{cases}$$

Clearly, $H^3_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_1)$.

Finally, consider a run $R_3 = \langle F_1, H^3_{\diamond\mathcal{P}}, I, S_3, T\rangle$ of $R_{\diamond\mathcal{P}} \to_{\mathcal{T}}$ that outputs a history $H^3_{\mathcal{T}} \in \mathcal{T}(F_1)$. Assume that $S_3[k] = S_2[k]$, $\forall k \leqslant k_1$. Since partial runs of R_2 and R_3 for $t \leqslant T[k_1]$ are identical, there exists $i \neq j$ such that:

$$H^3_{\mathcal{T}}(i, T[k_0]) = \emptyset,$$
$$H^3_{\mathcal{T}}(i, T[k_1]) = \{j\}.$$

In other words, j is suspected by i at time $T[k_1]$ after not being suspected by i at time $T[k_0] < T[k_1]$. By the trusting accuracy property of $\mathcal{T}$, j is crashed in F_1, which contradicts the assumption that F_1 is failure-free.

Thus, $\diamond\mathcal{P} \prec_{\mathcal{E}_f} \mathcal{T}$. □

```
1: output_i := Π                                          { Initialization }
2: crashed_i := ∅
3: start tasks 0, ..., n+1

4: task 0:
5:   try_ii                                              { i requests CS_i }
6:   send [me, i, i] to all                              { i enters CS_i }

                                  {An indication that k entered CS_k is received }
7: task k (k = 1, ..., n):
8:   upon receive [me, k, k] do
9:     if k ∉ crashed_i then
10:      output_i := output_i − {k}                       { i stops suspecting k }
11:    if k ≠ i then
12:      try_ik                                          { i requests CS_k }
13:      send [me, i, k] to all                          { i enters CS_k }

                                  {An indication that j entered CS_k is received }
14: task n + 1:
15:   upon receive [me, j, k] with j ≠ k do
16:     crashed_i := crashed_i ∪ {k}
17:     output_i := output_i ∪ {k}                        { i starts suspecting k }
```

Fig. 2. Reduction algorithm $R_{\mathcal{D}} \to_{\mathcal{T}}$ –process i.

5. The necessary condition for solving FTME

This section shows that the trusting failure detector $\mathcal{T}$ is necessary to solve FTME in *any* environment $\mathcal{E}$. In other words, we show that if a failure detector $\mathcal{D}$ solves FTME in $\mathcal{E}$, then $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{D}$.

Assume that an algorithm A solves FTME in an environment $\mathcal{E}$ using a failure detector $\mathcal{D}$. A reduction algorithm $R_{\mathcal{D}} \to_{\mathcal{T}}$ that transforms $\mathcal{D}$ into $\mathcal{T}$ is presented in Fig. 2. At any time $t \in \mathbb{T}$ and for any process $i \in \Pi$, $R_{\mathcal{D}} \to_{\mathcal{T}}$ outputs the set of processes suspected by i, $output_i(t)$.

In the algorithm of Fig. 2, processes can access n different critical sections: $CS_1, \ldots, CS_n$ by using n parallel instances of algorithm A. Let try_{ij}, crit_{ij}, exit_{ij} and rem_{ij} denote, respectively, trying, critical, exit and remainder sections of process i with respect to CS_j. Each process i *controls* critical section CS_i, i.e., in any run in which i is correct, i eventually gets access to CS_i. As a parallel task, i may request and gain access to any CS_j. (For brevity, we say that i *requests* CS_j and that i *enters* CS_j.) Process i requests CS_j $(j = 1, \ldots, n)$ by executing trying protocol try_{ij}. By definition, if CS_j is used correctly (the processes are well-formed with respect to CS_j), then A guarantees the properties of FTME.

The idea of the algorithm is the following. Initially, $\forall i \in \Pi$: $output_i = \Pi$ (every process is suspected). Process i first runs the trying protocol try_{ii} in order to enter CS_i. Since i is the only process in the trying section for CS_i, i eventually either crashes or enters CS_i and then sends the message $[me, i, i]$ to all. Every correct process that received $[me, i, i]$ stops suspecting i and executes try_{ij} in order to enter CS_i.

In our algorithm, a process can leave its CS only because of a crash. Thus, the only reason for which a correct process i can enter CS_j $(i \neq j)$ is the crash of j. In this case, process i sends the message $[me, i, j]$ to all processes. Every process that receives the message $[m, i, j]$ $(i \neq j)$ starts suspecting j.

As a result, eventually, no correct process is suspected by any correct process and every crashed process is permanently suspected by every correct process. Moreover, the only reason to start suspecting a process i after trusting it, is the crash of i. That is, the output of $\mathcal{T}$ is emulated.

To ensure progress of the failure detector output, the reduction algorithm of Fig. 2 maintains, at every process $i \in \Pi$, $n + 2$ parallel tasks:

- *task* 0 in which i runs the trying protocol try_{ii};
- *task* k $(k = 1, \ldots, n)$ in which i detects that k has entered CS_k, stops suspecting k and runs the trying protocol try_{ik} (lines 9–10 are executed atomically);
- *task* $n + 1$ in which i detects failures of other processes and starts suspecting them.

Lemma 3. *The algorithm of Fig. 2 emulates the trusting failure detector* $\mathcal{T}$.

Proof. According to the algorithm of Fig. 2, no process i requests twice the same instance CS_j or exits. Thus, each i is well-formed with respect to each CS_j. Note that, once entered CS_j, i can leave CS_j only if i crashes.

By contradiction, assume that the strong completeness property of $\mathcal{T}$ is violated. More precisely,

$$\exists F,\ \exists i \in correct(F),\ \exists j \notin correct(F) : \forall t, \exists t' > t,\ j \notin output_i(t').$$

Initially, $output_i = \Pi$ (every process is suspected). By the algorithm, initially, $j \in output_i$, and the correct process i removes j from $output_i$ (line 10 of Fig. 2) at most once and only if (a) the message $[me, j, j]$ is received (line 8), i.e., j is in CS_j (line 6) and, (b) $j \notin crashed_i$.

As a result, i runs try_{ij} in order to enter CS_j (line 12). By the *progress* property of FTME, at some time later, some correct process m is in CS_j. By the algorithm, m sends $[me, m, j]$ to all. Eventually, process i receives $[me, m, j]$ (j is faulty, thus, $m \neq j$). Since lines 9–10 are executed atomically, i cannot execute line 16 (while processing $[me, m, j]$) *before* executing line 10 (while processing $[me, j, j]$). As a result of processing $[me, m, j]$, i adds j to $output_i$ (line 17) and j stays in $output_i$ forever—a contradiction with.

Thus, strong completeness of $\mathcal{T}$ is satisfied.

By contradiction, assume that trusting accuracy is violated. More precisely,

$$\exists F,\ \exists i,\ \exists t' > t,\\ \exists j \notin F(t') : (j \notin output_i(t) \wedge j \in output_i(t')).$$

By the algorithm, i suspects j at time t' only if some process $k \neq j$ enters CS_j at some time $t_0 < t'$ and only if at some time $t_1 < t_0$ j itself entered CS_j. By the *mutual exclusion* property of FTME, j had to leave CS_j *before* t_0. Since j never executes the exit protocol, j could leave CS_j only because of its crash, that is, $j \in F(t')$—a contradiction.

By contradiction, assume now that eventual strong accuracy is violated. More precisely,

$$\exists F, \exists i \in correct(F),\\ \exists j \in correct(F), \forall t, \exists t' > t : j \in output_i(t').$$

Note that the assumption implies that $\forall t \in \mathbb{T}$, $j \in output_i(t)$, otherwise, trusting accuracy is violated.

Thus, i never stops suspecting j: by the algorithm, i never reaches line 10 while processing the reception of $[me, j, j]$. That is, either (1) i receives $[me, k, j]$ with $k \neq j$ and put j into $crashed_i$ (lines 15–17), or (2) i never receives $[me, j, j]$.

Assume that (1) is true. By the algorithm, $[me, k, j]$ with $k \neq j$ can be only received if k entered CS_j at some time t_0 and if at some time $t_1 < t_0$ j entered CS_j. Since j never executes the exit protocol, j could leave CS_j only if it is faulty—a contradiction.

Assume that (2) is true. Since both i and j are correct, j never sends $[me, j, j]$ (line 6). Thus, no process ever receives $[me, j, j]$. By the algorithm, a process k executes the trying protocol try_{kj} only if k received $[me, j, j]$. Thus, j is the only correct process that ever requests access to CS_j. By the *progress* property of FTME, j eventually enters CS_j and sends $[me, j, j]$ to all—a contradiction. Thus, the reduction algorithm of Fig. 2 guarantees the properties of $\mathcal{T}$. □

As a corollary, we obtain the following result.

Theorem 4. *For any environment $\mathcal{E}$, if a failure detector $\mathcal{D}$ solves FTME in $\mathcal{E}$, then $\mathcal{T} \preceq_{\mathcal{E}} \mathcal{D}$.*

6. The sufficient condition for solving FTME

We give in Fig. 3 an algorithm that solves FTME using $\mathcal{T}$ assuming an environment $\mathcal{E}_f$ with a majority of correct processes ($f < \lceil \frac{n}{2} \rceil$). The algorithm uses the fact that $\diamond\mathcal{P} \preceq_{\mathcal{E}_f} \mathcal{T}$ and, as a result, we can implement *total order broadcast* using $\mathcal{T}$ in $\mathcal{E}_f$ [4].

Total order broadcast is defined through the primitives to-broadcast() and to-deliver() and satisfies the following properties:

validity: if a correct process i to-broadcasts a message m, then i eventually to-delivers m;

agreement: if a process to-delivers a message m, every correct process eventually to-delivers m;

integrity: every message is to-delivered at most once, and only if the message was previously to-broadcast;

```
1:  ready_i := false                                   { Initialization }
2:  r_i := 0
3:  trusted_i := ∅
4:  start tasks 0, ..., n

    Trying protocol try_i:
5:  if not ready_i then
6:    send [me, i] to all                    { Send a trust request to all }
7:    wait until received ⌊n/2⌋ + 1 [ack]'s
8:    ready_i := true
9:  r_i := r_i + 1
10: to-broadcast([i, r_i])
11: repeat
12:   wait until the next request [j, k] is to-delivered
13:   if i ≠ j then
14:     wait until received [exit, j, k] or received [crash, j]
15: until i = j
16: { i enters CS }

    Exit protocol exit_i:
17: send [exit, i, r_i] to all

                                   { A crash of process l is detected }
18: task 0:
19:   upon (l ∈ trusted_i and l ∈ T_i) do
20:     trusted_i := trusted_i − {l}
21:     send [crash, l] to all                  { l stops being trusted }

                                { A trust request is received from m ∈ Π }
22: task m (m = 1, ..., n):
23:   upon receive [me, m] do
24:     wait until m ∉ T_i                      { Wait until m is trusted }
25:     trusted_i := trusted_i ∪ {m}
26:     send [ack] to m
```

Fig. 3. FTME algorithm using $\mathcal{T}$: process i.

total-order: if a process i to-delivers a message m before having delivered a message m', then no process j can to-deliver m' without having to-delivered m first. [2]

Note that the total-order property implies that if a process i to-delivered a message m and a process j to-delivered a message m', then either m is to-delivered by j before m' or m' is to-delivered by i before m.

The algorithm of Fig. 3 assumes that:

- an algorithm implementing total order broadcast is provided;
- every process i has access to the output of its trusting failure detector module $\mathcal{T}_i$;
- every process i is well-formed.

In our algorithm of Fig. 3, each process i maintains the following local variables:

(1) a boolean $ready_i$, initially *false*, indicating whether i is ready to execute the trying protocol;
(2) a set $trusted_i \subseteq \Pi$, initially empty, of processes currently trusted by i;
(3) an integer r_i, initially 0, indicating the number of times i has run the trying protocol;
(4) integers j and k indicating the last processed request of the type $[j, k]$ where j is the process that issued the request and k is j's request number.

Our algorithm also assumes that every process i stores the identifiers of all received messages in a buffer, so that, for a given message m, the predicate "received m" (lines 7 and 14 of Fig. 3) is true if and only if m has been previously received by i.

The idea of our algorithm is inspired by the well-known Bakery algorithm of Lamport [13,14]: the processes that wish to enter their CSs (the candidates) first draw tickets and then are served in the order of their tickets numbers. Before drawing a ticket, every candidate asks for a permission to proceed from some *correct* process and waits (line 7) until the permission is received (it eventually happens due to the assumption of a majority of correct processes in the system). Then the candidate is put into the waiting line implemented by the total order broadcast mechanism. Total order broadcast guarantees that the requests are eventually delivered in the same order (line 12), i.e., no candidate i can be served unless every candidate in the waiting line before i has been served and has released the resource, or crashed (line 14). If a process crashes in its CS, then at least one correct process will eventually detect the crash and informs the others (lines 19–21 in Fig. 3).

[2] This definition of the total-order property is slightly stronger than the one proposed in [10]: we require that all correct processes deliver the same sequence of messages, and all faulty processes deliver *prefixes* of this sequence. This distinction however does not matter for our results, since the algorithm given in [4] implements the strongest version of total order broadcast.

To ensure the *progress* property of FTME, in addition to the trying and exit protocols (respectively, lines 5–16 and line 17 of Fig. 3), the algorithm maintains, at every process $i \in \Pi$, $n+1$ parallel tasks:

- *task* 0 in which i detects failures of other processes;
- *task* m ($m = 1, \ldots, n$) in which i takes care of the trust request of process m.

Now we prove the correctness of the algorithm through Lemmas 5 and 6.

Lemma 5. *No two different processes are in their CSs at the same time.*

Proof. By contradiction, assume that i and j ($i \neq j$) are in their CSs at time t_0. Let, at time t_0, $r_i = k_i$ and $r_j = k_j$.

In the trying protocol (lines 5–16), every process to-broadcasts its request for a CS and no process enters its CS before having first to-delivered its request. Thus i must have to-delivered $[i, k_i]$ and j must have to-delivered $[j, k_j]$ before t_0. By the ordering property of to-broadcast, either both i and j to-delivered $[i, k_i]$ before having to-delivered $[j, k_j]$, or the contrary. Assume, without loss of generality, that to-deliver($[i, k_i]$) precedes to-deliver($[j, k_j]$) at j. That is, at some time $t_1 < t_0$, j passed the "wait" clause in line 14 while processing $[i, k_i]$. Thus, one of the following events occurred *before* t_1 at j:

(1) j received $[exit, i, k_i]$: by the algorithm, i left the CS with $r_i = k_i$ before time t_1. But i is in the CS with $r_i = k_i$ at $t_0 > t_1$—a contradiction.
(2) j received $[crash, i]$: by the algorithm, at some process m, at some time $t_2 < t_1$ the following is true: $i \in trusted_m$ and $i \in \mathcal{T}_m$. But i can be in $trusted_m$ only if previously $i \notin \mathcal{T}_m$ (lines 24–25). That is, m stopped trusting i at time t_2. By the trusting accuracy property of $\mathcal{T}$, i is crashed at t_2. But i is in the CS at $t_0 > t_2$—a contradiction.

Hence, mutual exclusion is guaranteed. □

Lemma 6. *If a correct process is in its trying section, then at some time later some correct process is in its CS. If a correct process is in its exit section, then at some time later it enters its remainder section.*

Proof. Assume that a correct process $\bar{i}$ in its trying section at some time t_c with $r_{\bar{i}} = \bar{r}$, and no correct process is ever in its CS after t_c. By the algorithm, $\bar{i}$ never reaches line 16. Thus, $\bar{i}$ is blocked in a "wait" clause or at the non-terminating repeat-until loop. The first "wait" clause (line 7 of Fig. 3) is not able to block the process, due to eventual strong accuracy of $\mathcal{T}$ and the fact that at least $\lfloor n/2 \rfloor + 1$ processes are correct. Thus, $\bar{i}$ eventually issues to-broadcast($[\bar{i}, \bar{r}]$). The second "wait" clause (more precisely, the statement in line 12 of Fig. 3) is not blocking neither, because of validity of total order broadcast: eventually, i to-delivers at least one

message–$[\bar{i}, \bar{r}]$. Further, if the "wait" clause in line 14 is not blocking, then validity of total order broadcast implies that $[\bar{i}, \bar{r}]$ is eventually to-delivered by $\bar{i}$, thus $\bar{i}$ exits the repeat-until loop and enters its CS.

Thus, $\bar{i}$ is blocked in the third "wait" clause (line 14 of Fig. 3) while processing some $[\bar{j}, \bar{k}]$ ($\bar{i} \neq \bar{j}$). Thus, $\bar{i}$ never receives $[exit, \bar{j}, \bar{k}]$ or $[crash, \bar{j}]$.

By integrity of total order broadcast, $\bar{j}$ has previously to-broadcast $[j, k]$ (line 10 of Fig. 3).

Let j be any process that reaches line 10.

We observe first that (Claim 1) j has been previously put in $trusted_m$ by some *correct* process m. Indeed, j received $\lfloor n/2 \rfloor + 1$ $[ack]$'s from processes that trusted j. Since at least $\lfloor n/2 \rfloor + 1$ processes are correct, j receives at least one $[ack]$ from a correct process m that previously put j in $trusted_m$ at some time t_0.

Then we notice that (Claim 2) if j is faulty, then, every correct process eventually receives $[crash, j]$. Indeed, if j is faulty, then, by trusting completeness of $\mathcal{T}$ and Claim 1, some correct process m eventually and permanently suspects j: $\exists t_1 > t_0 : \forall t > t_1 : j \in \mathcal{T}_m$. That is, eventually, the condition of line 19 is satisfied at m for j ($j \in trusted_m$ and $j \in \mathcal{T}_m$). Thus, m sends $[crash, j]$ to all processes and every correct process eventually receives it.

Hence, process $\bar{j}$ should necessarily be correct. Indeed, if $\bar{j}$ is faulty, then, by Claim 2, correct process $\bar{i}$ eventually receives $[crash, \bar{j}]$ and releases from waiting in line 14.

Further, we observe that trusting accuracy of $\mathcal{T}$ implies that (Claim 3) if a message $[crash, j]$ is received, then j is crashed.

Finally, we show that (Claim 4) if a correct process m passed an entry $[j, k]$ in the total order (is not blocked in line 14 while processing $[j, k]$), then no correct process can be blocked while processing $[j, k]$. Indeed, the following cases are possible:

(a) $j = m$: j enters its CS (line 16). By the assumption of the proof, no correct process is in its CS after t_c, thus, j left its CS before t_c and j sent $[exit, j, k]$ to all (line 17). Thus, every correct process eventually receives the message and releases.
(b) $j \neq m$, and j is faulty. By Claim 2, every correct process eventually receives $[crash, j]$ and releases.
(c) $j \neq m$, and j is correct. By Claim 3, m could only receive $[exit, j, k]$. Every correct process eventually receives $[exit, j, k]$ and releases.

Recall that $\bar{i}$ is blocked in line 14 while processing request $[\bar{j}, \bar{k}]$ ($\bar{i} \neq \bar{j}$). By Claim 2, $\bar{j}$ is correct, and, by Claim 4, $\bar{j}$ should have passed all entries in the total order that $\bar{i}$ has passed before reaching $[\bar{j}, \bar{k}]$. By the algorithm $\bar{j}$ enters its CS (line 16). By the assumption of the proof, no correct process is in its CS after t_c, thus, $\bar{j}$ left its CS before t_c and sent $[exit, \bar{j}, \bar{k}]$ to all. $\bar{i}$ eventually receives the message and releases—a contradiction.

The second part of the lemma follows directly from the algorithm: every correct process i that runs exit_i enters rem_i after a finite number of steps. That is, every correct process in its exit section eventually enters its remainder section.

Thus, progress is guaranteed. □

The following theorem follows directly from Lemmas 5 and 6:

Theorem 7. *The algorithm of Fig. 3 solves FTME using $\mathcal{T}$, in any environment $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$.*

Finally, combining Theorems 4 and 7, we can state the following result:

Theorem 8. *For any environment $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$, $\mathcal{T}$ is the weakest failure detector to solve FTME in $\mathcal{E}_f$.*

Remark. In fact, the algorithm of Fig. 3 solves a harder problem: in addition to mutual exclusion and progress, it satisfies also the starvation-freedom property.

Indeed, assume that a correct process i is in its trying section with $r_i = k$. Eventually, due to the properties of the total order broadcast, all entities $[j, l]$ preceding $[i, k]$ in the total order are eventually processed: if any process releases its CS, no process can be blocked in a "wait" clause (see line 14 in Fig. 3). Finally, i eventually reaches its own entry $[i, k]$ in the total order and i enters its CS.

From Theorem 8 it follows that any algorithm solving FTME (in $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$) can be transformed into an algorithm that solves FTME with the starvation freedom property.

7. On the number of correct processes

Proposition 9. *No algorithm solves FTME using $\mathcal{T}$ in any environment $\mathcal{E}_f$ where $f \geqslant \lceil \frac{n}{2} \rceil$.*

Proof. Assume that an algorithm A solves FTME using $\mathcal{T}$ in an environment where a majority of correct processes is not guaranteed. Let X and Y be any two disjoint sets of processes such that $\Pi = X \cup Y$ and $|X| = \lceil \frac{n}{2} \rceil$. Consider two possible runs of A:

(1) R_1: no process in Y takes any step in R_1 (e.g., processes in Y are initially crashed in R_1), and processes in X always suspect every process in Y. Assume that a correct process $i \in X$ is the only process in its trying section. By the progress property of FTME, i enters its CS at some time t_1.
(2) R_2: no process from X takes any step in R_2 (e.g., processes in X are initially crashed in R_2), no process in Y takes any step before $t_1 + 1$, and processes in Y always suspect every process in X. Assume that a correct process $j \in Y$ is the only process in its trying section. By

the *progress* property of FTME, j enters its CS at some time t_2. Clearly, $t_1 < t_2$.

Assume that no process ever runs an exit protocol in R_1 and R_2. We construct a run R that is identical to R_1 at any time in $[0, t_1]$ and identical to R_2 at any time in $[t_1 + 1, t_2]$. Now assume that every process is correct in R, the processes in X and Y start to trust each other *after* t_2 (this is a valid history of $\mathcal{T}$), and all messages sent between X and Y are delayed until $t_2 + 1$. Evidently, R is a valid run of A. But, since i and j never enter their exit section s, at time t_2 both i and j are in their CSs—a contradiction. □

Now we consider the extreme case of an environment $\mathcal{E}_f$, where $f = n - 1$ and question ourselves whether $\mathcal{P}$ is the weakest failure detector to solve the problem in $\mathcal{E}_{n-1}$. A close look at the correctness proof for the algorithm of Fig. 3 reveals that we use the assumption of a correct majority only to implement the total order broadcast primitive and to guarantee that for each correct process i, there is a correct process m that trusts i. If a strong failure detector $\mathcal{S}$ [4] is available, we can overcome both issues even if $n - 1$ processes can crash. Indeed, total order broadcast is implementable in $\mathcal{E}_{n-1}$ using $\mathcal{S}$ [4] and the "wait" clause in line 7 can be substituted by:

wait until receive[ack] from all $j \notin \mathcal{S}_i$.

By the strong completeness property of $\mathcal{S}$, eventually all processes not in $\mathcal{S}_i$ are correct. On the other hand, by the eventual strong accuracy of $\mathcal{T}$, every correct process is eventually trusted by all correct processes. Hence, this "wait" clause is not blocking.

By the weak accuracy property of $\mathcal{S}$, one correct process is never suspected. That is, some correct process m is never in $\mathcal{S}_i$, $\forall i \in \Pi$. If i crashes while i is in its CS, m can detect the crash and inform the other processes. Thus, we can implement FTME in $\mathcal{E}_{n-1}$ using failure detector $\mathcal{T} + \mathcal{S}$. For every failure pattern $F \in \mathcal{E}_f$ ($f < n$), $\mathcal{T} + \mathcal{S}$ outputs a pair of histories $(H_\mathcal{T}, H_\mathcal{S})$ ($\mathcal{R}_{\mathcal{T}+\mathcal{S}} = 2^\Pi \times 2^\Pi$), such that $H_\mathcal{T} \in \mathcal{T}(F)$ and $H_\mathcal{S} \in \mathcal{S}(F)$.

Proposition 10. *$\mathcal{T} + \mathcal{S} \prec_{\mathcal{E}_f} \mathcal{P}$, in any environment $\mathcal{E}_f$ with $0 < f < n$.*

Proof. (a) $\mathcal{S} \prec_{\mathcal{E}_f} \mathcal{P}$ [4] and $\mathcal{T} \prec_{\mathcal{E}_f} \mathcal{P}$ (Proposition 2). That is, both $\mathcal{T}$ and $\mathcal{S}$ are weaker than $\mathcal{P}$. Thus, $\mathcal{T} + \mathcal{S} \preceq_{\mathcal{E}_f} \mathcal{P}$.

(b) Now we show that $\mathcal{P}$ is not weaker than $\mathcal{T}+\mathcal{S}$. Indeed, assume there exists an algorithm $R_{\mathcal{T}+\mathcal{S}\to\mathcal{P}}$ that, for any failure pattern $F \in \mathcal{E}_f$, constructs $H_\mathcal{P}$ from $H_\mathcal{T} \in \mathcal{T}(F)$ and $H_\mathcal{S} \in \mathcal{S}(F)$, such that $H_\mathcal{P} \in \mathcal{P}(F)$.

Let $j, l \in \Pi$ and $j \neq l$. Consider failure pattern $F_1 \in \mathcal{E}_f$ such that $F_1(0) = \{j\}$, $correct(F_1) = \Pi - \{j\}$, and take histories $H^1_\mathcal{T} \in \mathcal{T}(F_1)$ and $H^1_\mathcal{S} \in \mathcal{S}(F_1)$ such that $\forall i \in \Pi, \forall t \in \mathbb{T}$: $H^1_\mathcal{T}(i, t) = \{j\}$ (j is always suspected) and $H^1_\mathcal{S}(i, t) = \Pi - \{l\}$ (l is never suspected). Assume that the corresponding run $R_1 = \langle F_1, (H^1_\mathcal{T}, H^1_\mathcal{S}), I, S_1, T\rangle$ of $R_{\mathcal{T}+\mathcal{S}\to\mathcal{P}}$ outputs a history $H^1_\mathcal{P} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists k_0 \in \mathbb{N}$: $H^1_\mathcal{P}(l, T[k_0]) = \{j\}$.

Consider failure pattern $F_2 \in \mathcal{E}_f$ such that $correct(F_2) = \Pi$ and define histories $H^2_\mathcal{T}$ and $H^2_\mathcal{S}$ such that $\forall i \in \Pi$ and $\forall t \in \mathbb{T}$:

$$H^2_\mathcal{T}(i, t) = \begin{cases} \{j\}, & t \leqslant T[k_0], \\ \emptyset, & t > T[k_0], \end{cases}$$

$$H^2_\mathcal{S}(i, t) = \begin{cases} \Pi - \{l\}, & t \leqslant T[k_0], \\ \emptyset, & t > T[k_0]. \end{cases}$$

Clearly, $H^2_\mathcal{T} \in \mathcal{T}(F_2)$ and $H^2_\mathcal{S} \in \mathcal{S}(F_2)$.

Consider a run $R_2 = \langle F_2, (H^2_\mathcal{T}, H^2_\mathcal{S}), I, S_2, T\rangle$ of $R_{\mathcal{T}+\mathcal{S}\to\mathcal{P}}$ that outputs a history $H^2_\mathcal{P} \in \mathcal{P}(F_2)$, where $S_1[k] = S_2[k], \forall k \leqslant k_0$. Thus, j takes no steps in S_2 for all $t \leqslant T[k_0]$. Since partial runs of R_1 and R_2 for $t \leqslant T[k_0]$ are identical, the resulting history $H^2_\mathcal{P}$ is such that $H^2_\mathcal{P}(l, T[k_0]) = \{j\}$. In other words, j is suspected before it crashes, and the *strong accuracy* of $\mathcal{P}$ is violated.

By (a) and (b), we have $\mathcal{T} + \mathcal{S} \prec_{\mathcal{E}_f} \mathcal{P}$. □

Hence, there is a failure detector $\mathcal{T} + \mathcal{S}$ which is strictly weaker then $\mathcal{P}$ and is sufficient to solve FTME in an environment where up to $n - 1$ processes can crash.

8. Group mutual exclusion

Group mutual exclusion [9,11,12] is a natural generalization of the classical mutual exclusion problem [6,14], where a process requests a "session" before entering its critical section. Processes are allowed to be in their critical sections simultaneously provided that they have requested the same session. Sessions represent resources each of which can be accessed simultaneously by an arbitrary number of processes, but no two of which can be accessed simultaneously.

Formally, the trying protocol of process i has an integer parameter s. We say that *i requests session s* if i is running the trying protocol $\mathsf{try}_i(s)$ or it is in its CS immediately after running $\mathsf{try}_i(s)$. As with FTME, we assume that every process i is well-formed.

Thus, in addition to the progress properties of FTME, fault-tolerant group mutual exclusion (FTGME) satisfies the group mutual exclusion and concurrent entering properties (we follow the terminology used in Section 3):

Progress: (1) If a correct process is in its trying section, then at some time later some correct process is in its CS.

(2) If a correct process is in its exit section, then at some time later it enters its remainder section.

Mutual exclusion: If two processes are in their critical sections at the same time, then they request the same session.

Concurrent entering: If a correct process i requests a session and no other process requests a different session, then i eventually enters its CS.

```
 1: ready_i := false                                   { Initialization }
 2: r_i := 0
 3: trusted_i := ∅
 4: inCS_i := ∅
 5: ls_i := −1
 6: start tasks 0, ..., n

    Trying protocol try_i(session_i):
 7: if not ready_i then
 8:    send [me, i] to all                            { Send a trust request }
 9:    wait until received ⌊n/2⌋ + 1 [ack]'s
10:    ready_i := true
11: r_i := r_i + 1
12: to-broadcast([i, r_i, session_i])
13: repeat
14:    wait until the next request [j, k, s] is to-delivered
15:    if inCS_i ≠ ∅ and s ≠ ls_i then
16:       wait until inCS_i = ∅ or received [crash, j]
17:    inCS_i := inCS_i ∪ {(j, k)}
18:    ls_i := s
19: until j = i
20: { i enters its CS }

    Exit protocol exit_i:
21: send [exit, i, r_i] to all

22: task 0:
23:    upon (j ∈ trusted_i and j ∈ T_i) do
24:       trusted_i := trusted_i − {j}    { A crash of process j is detected }
25:       send [crash, j] to all                      { j stops being trusted }
26:    upon ((j, k) ∈ inCS_i and
              (received [exit, j, k] or received [crash, j])) do
27:       inCS_i := inCS_i − {(j, k)}                 {[j, k, ls_i] releases the CS }

28: task m (m = 1, ..., n):
29:    upon receive [me, m] do
30:       wait until m ∉ T_i        { A trust request is received from m }
31:       trusted_i := trusted_i ∪ {m}                { m is trusted by i }
32:       send [ack] to m
```

Fig. 4. FTGME algorithm using $\mathcal{T}$: process i.

The last property means that, for a given session, a process that has *already* entered its CS cannot prevent another process requesting the same session from entering its CS. The property excludes trivial solutions of group mutual exclusion using any simple mutual exclusion algorithm. In contrast to [9,11,22], we do not make the assumption that a process can stay in its CS for a finite time only. This is the reason why we put "eventually" instead of "a bounded number of its own steps" as in [9,11,22] in the concurrent entering property. Clearly, if another process is concurrently trying to enter a different session, it can enter its CS first. In this case, the trying process *can* prevent another process from entering its CS.

FTGME is at least as hard as FTME: we can easily implement FTME from FTGME just associating every process with a unique session number. On the other hand, we show here that $\mathcal{T}$ solves FTGME in a system with a majority of correct processes. Thus, in the sense of failure detection, FTME and FTGME are equivalent.

In Fig. 4, we present an algorithm that solves FTGME using $\mathcal{T}$. For each process i, the algorithm of Fig. 4 defines trying protocol $\mathsf{try}_i(session_i)$ that handles the request of i for session $session_i$, and exit protocol exit_i. In the algorithm, each process i maintains the following local variables:

(1) a boolean $ready_i$, initially *false*, indicating whether i is ready to execute the trying protocol;
(2) an integer r_i, initially 0, indicating the number of requests for the CS that i has made;
(3) a set $trusted_i$, initially empty, of processes currently trusted by i;
(4) an integer ls_i, initially -1 (we assume that requested session numbers are non-negative), indicating the number of currently satisfied session;
(5) a set $inCS_i$, initially empty, of requests with session number ls_i that i suspects to be currently satisfied;
(6) integers j, k and s indicate the last processed request of the type $[j, k, s]$ where j is the process that issued the request, k is j's request number and s is the session that j requests.

The algorithm is similar to that of Section 6. Before requesting a session every process waits until it gets trusted by a correct process. The requests are broadcast using total order broadcast primitive to-broadcast(), and delivered through to-deliver(). If several consecutive requests for the same session s are placed in the total order, then the requests are satisfied simultaneously. No request for a new session $s' \neq s$ is satisfied until all processes requested earlier session s leave their CSs.

Now we state the correctness of the algorithm through Lemmas 11–13.

Lemma 11. *If two processes are in their critical sections at the same time, then they request the same session.*

Proof. Assume that processes i and j requesting sessions, respectively, s_i and s_j are in their CSs at some time t_0. Let, at time t_0, $r_i = k_i$ and $r_j = k_j$.

In the trying protocol (lines 7–20), every process to-broadcasts its request for a CS and no process enters its CS before having first to-delivered its own request. Thus i must have to-delivered $[i, k_i, s_i]$ and j must have to-delivered $[j, h_j, s_j]$ before t_0. By the ordering property of to-broadcast, either both i and j to-delivered $[i, k_i, s_i]$ before having to-delivered $[j, k_j, s_j]$, or the contrary. Assume, without loss of generality, that to-deliver$([i, k_i, s_i]$ precedes to-deliver$([j, k_j, s_j])$ at j.

By the algorithm, j can be in the CS with $session_j = s_j$ and $r_j = k_j$ at t_0 only if every entry $[j', k', s']$ with $s' \neq s_j$ in the total order preceding $[j, k_j, s_j]$ has passed through the "if" clause defined in lines 15–16 *before* time t_0. As a result, before time t_0, j has put (i, k_i) into $inCS_j$ and set ls_j to s_i (lines 17 and 18).

Since i is still in its CS with $r_i = k_i$ at time t_0, j could not have received $[exit, i, k_i]$ before t_0. Now assume that j received $[crash, i]$ before t_0: by the algorithm of Fig. 4, at some process m, at some time $t_1 < t_0$ the following is true:

$i \in trusted_m$ and $i \notin \mathcal{T}_m$ (m stops trusting i). By trusting accuracy of $\mathcal{T}$, i is crashed at t_1. But i is in the CS at $t_0 > t_1$—a contradiction.

Thus, j has not received $[exit, i, k_i]$ or $[crash, i]$ before t_0, i.e., the condition in line 26 is not satisfied at j before t_0. As a result, at the moment when j to-delivered $[j, k_j, s_j]$ (line 14), $(i, k_i) \in inCS_j$ and $ls_j = s_i$. Assume that j reaches line 15 while processing $[j, k_j, s_j]$ at some time $t_1 < t_0$ (j is in its CS at t_0). Furthermore, $inCS_j$ is non-empty at any $t \in [t_1, t_0]$ (it includes at least one entry (i, k_i)), j never receives $[crash, j]$ (by trusting accuracy of $\mathcal{T}$), and $ls_j = s_i$ at t_1. Thus, j can pass lines 15–16 and enter its CS before t_0 only if $s_j = s_i$. Hence, group mutual exclusion is guaranteed. □

Lemma 12. *If a correct process is in its trying section, then at some time later some correct process is in its CS. If a correct process is in its exit section, then at some time later it enters its remainder section.*

Proof. The proof is similar to the one of Lemma 6. Assume that a correct process $\bar{i}$ is in its trying section at time t_0, and no correct process ever enters its CS after t_0. Applying the arguments of Lemma 6, we observe that $\bar{i}$ is blocked in line 16 of Fig. 4 because some entry $(\bar{j}, \bar{k})$ never leaves $inCS_{\bar{i}}$ (line 27). Claims 1–4 of Lemma 6 are proved similarly. By Claims 1 and 2 of Lemma 6, $\bar{j}$ must be correct. By Claims 3 and 4 of Lemma 6 $\bar{j}$ should have passed all entries in the total order that precede $[\bar{j}, \bar{k}, \bar{s}]$ and entered its CS. Since no process is in its CS after t_0, $\bar{j}$ executed the exit protocol before t_0 and sent $[exit, \bar{j}, \bar{k}]$ to all. Thus $\bar{i}$ eventually receives $[exit, \bar{j}, \bar{k}]$ and releases—a contradiction. □

Lemma 13. *If a correct process i requests a session and no other process requests a different session, then i eventually enters its CS.*

Proof. Assume that, at time t_0, a process i requests a session s_i with $r_i = k_i$ and no other process requests a different session. Thus, all processes requesting different sessions have left their CSs or crashed before t_0. As a result, after some time, either $inCS_i = \emptyset$ or $ls_i = s_i$. By the algorithm, eventually, i starts processing its own request $[i, k_i, s_i]$ with $ls_i = s_i$ (lines 14–15) and enters its CS (line 20). □

Finally, we can state the following theorem:

Theorem 14. *For any environment $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$, $\mathcal{T}$ is the weakest failure detector to solve FTGME in $\mathcal{E}_f$.*

Remark. Similar to the FTME algorithm of Fig. 3, our FTGME algorithm solves (in $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$) a harder problem that, in addition to mutual exclusion, progress and concurrent entering, satisfies also the starvation freedom property.

Analogously, in case when up to $n-1$ processes can crash, we can solve FTGME with $\mathcal{T} + \mathcal{S}$, simply by substituting line 9 of the algorithm in Fig. 4 with:

wait until receive $[ack]$ from all $j \notin \mathcal{S}_i$.

9. Cost of resilience

In this section we compare the performance of our algorithm (Fig. 3) with the well-known algorithms of [17,20]. (The algorithms of [17,20] were designed for the failure-free asynchronous model but could be ported into the crash-prone model assuming $\mathcal{P}$. More details on the comparative analysis of the algorithms of [17,20] are available in [21].)

The performance of mutual exclusion algorithms can be measured through the following metrics [21]: (a) the *bootstrapping delay*, which is the time required for a new process before entering the CS for the first time; (b) the *number of messages* necessary per CS invocation, (c) the *synchronization delay*, which is the time required after a process leaves the CS and before the next process enters the CS, and (d) the *response time*, which is the time interval a requester waits to enter the CS after its request message have been sent out. We also consider two special loading conditions: *low load* and *high load*. In low load conditions, there is seldom more than one request to enter the CS at a time in the system. In high load conditions, any process that leaves the CS immediately executes the trying protocol again. In discussing performance, we concentrate here on the runs where no process crashes (the most frequent runs in practice), which are usually called *nice* runs.

We denote by t_c the maximum message propagation delay, and e_c the maximum CS execution time. The bootstrapping delay of our algorithm (Fig. 3) is bounded by $2t_c$: before processing any request for CS, every process should receive the acknowledgment from a majority of the processes. The algorithm has a relatively high message complexity: each request for CS requires $O(n^2)$ messages per CS invocation. The synchronization delay is bounded by t_c: that is, it requires only one communication step to inform the next waiting process that it can enter the CS. The response time in low load conditions is defined by the time to deliver a total order broadcast message–$2t_c$. At high loads, on the average, all other processes execute their CSs between two successive executions of the CS: the response time converges to $n(t_c + e_c)$.

The results of our comparative analysis are presented in Fig. 5. The performance degradation due to the use of $\mathcal{T}$ reflects the longer bootstrapping delay which is inherent to the use of $\mathcal{T}$ and higher message complexity inherited from using total order broadcast. It would be interesting to figure out to which extent our algorithm of Fig. 3 could be optimized, e.g., by breaking the encapsulation of the total order broadcast box.

Metrics	Maekawa [20]	RA [21]	$\mathcal{T}$-based
Bootstrapping delay	0	0	$2t_c$
Number of messages	Low	Moderate	High
Sync. delay	$2t_c$ (deadlock-prone)	t_c	t_c
	t_c (deadlock-free)		
Response time			
low load	$2t_c$	$2t_c$	$2t_c$
high load	$n(2t_c + e_c)$	$n(t_c + e_c)$	$n(t_c + e_c)$

Fig. 5. Comparative performance analysis of mutual exclusion algorithms.

10. Concluding remark

Is it more beneficial in practice to use a mutual exclusion algorithm based on $\mathcal{T}$, instead of a traditional algorithm assuming $\mathcal{P}$? The answer is "yes, to some extent". Indeed, if we translate the very fact of not trusting a correct process into a *mistake*, then $\mathcal{T}$ clearly tolerates mistakes whereas $\mathcal{P}$ does not. More precisely, $\mathcal{T}$ is allowed to make up to n^2 mistakes (up to n mistakes for each module $\mathcal{T}_i$, $i \in \Pi$). As a result, given synchrony assumptions, it is somewhat easier to implement $\mathcal{T}$ than $\mathcal{P}$.

For example, in a possible implementation of $\mathcal{T}$, every process i can, starting from 0, gradually increase the timeout t_{ij} corresponding to a heart-beat message sent to a process j until a response from j is received. Thus, every such t_{ij} can be flexibly adapted to the current network conditions. (Clearly, as soon as $\mathcal{T}$ starts trusting a site, it is not allowed to make mistakes about the site's operational state.)

In contrast, $\mathcal{P}$ does not allow this kind of "fine-tuning" of the timeouts: they are supposed to be known in advance. In order to minimize the probability of mistakes, the timeouts are normally chosen sufficiently large, and the choice is based on some a priori assumptions about current network conditions. This might exclude some remote sites from the group and violate the accuracy properties of the failure detector.

Thus, we can implement $\mathcal{T}$ in a more effective manner than $\mathcal{P}$, and an algorithm that solves FTME using $\mathcal{T}$ exhibits a smaller probability to violate the requirements of the problem, than one using $\mathcal{P}$, i.e., the use of $\mathcal{T}$ provides more resilience. As we have shown in Section 9, the performance cost of this resilience reflects the *bootstrapping delay*, i.e., the time a new process needs to enter its CS for the first time, and higher message complexity inherited from using total order broadcast.

Acknowledgments

We would like to thank the referees for detailed and helpful comments on the earlier version of the paper.

References

[1] D. Agrawal, A.E. Abbadi, An efficient and fault-tolerant solution for distributed mutual exclusion, ACM Trans. Comput. Systems 9 (1) (1991) 1–20.
[3] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, J. ACM 43 (4) (1996) 685–722.
[4] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, J. ACM 43 (2) (1996) 225–267.
[5] G. Chockler, D. Malkhi, M.K. Reiter, Backoff protocols for distributed mutual exclusion and ordering, in: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01), 2001, pp. 11–20.
[6] E.W. Dijkstra, Solution of a problem in concurrent programming control, Comm. ACM 8 (9) (1965) 569.
[7] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (3) (1985) 374–382.
[8] E. Gafni, M. Mitzenmacher, Analysis of timing-based mutual exclusion with random times, SIAM J. Comput. 31 (3) (2001) 816–837.
[9] V. Hadzilacos, A note on group mutual exclusion, in: Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01), 2001, pp. 100–106.
[10] V. Hadzilacos, S. Toueg, A modular approach to fault-tolerant broadcast and related problems, Technical Report, Cornell University, Computer Science, May 1994.
[11] Y.-J. Joung, Asynchronous group mutual exclusion, in: Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98), 1998, pp. 51–60.
[12] P. Keane, M. Moir, A simple local-spin group mutual exclusion algorithm, IEEE Trans. Parallel Distrib. Systems 12 (7) (2001) 673–685.
[13] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. ACM 17 (8) (1974) 453–455.
[14] L. Lamport, The mutual exclusion problem, Parts I&II, J. ACM 33 (2) (1986) 313–348.
[15] S. Lodha, A.D. Kshemkalyani, A fair distributed mutual exclusion algorithm, IEEE Trans. Parallel Distrib. Systems 11 (6) (2000) 537–549.
[16] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, LosAltos, CA, 1996.
[17] M. Maekawa, A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems, ACM Trans. Comput. Systems 3 (2) (1985) 145–159.
[18] D. Manivannan, M. Singhal, An efficient fault-tolerant mutual exclusion algorithm for distributed systems, in: Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems, 1994, pp. 525–530.
[19] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, Cambridge, MA, 1986.
[21] M. Singhal, A taxonomy of distributed mutual exclusion, J. Parallel Distrib. Comput. 18 (1) (1993) 94–101.
[22] K. Vidyasankar, A simple group mutual l-exclusion algorithm, Inform. Process. Lett. 85 (2003) 79–85.

Carole Delporte-Gallet Carole Delporte-Gallet is "Maître de Conférences" (Associate Professor) at University of Paris 7-Denis Diderot, France. She holds a M.S. degree in Mathematics and a Ph.D. in Computer Science from the University of Paris 7-Denis Diderot. Her research interests include distributed and fault-tolerant computing. Lately, her main interest concerns failure detectors from a theorical and practical point of view.

Hugues Fauconnier After a master degree in mathematics Hugues Fauconnier received his Ph.D. degree in computer science from the University of Paris 7-Denis Diderot, France, in 1982. He is Maître de Conférences (Associate Professor) of computer science at the University of Paris 7-Denis Diderot. His main research interests are distributed algorithms and fault tolerant computing.

Rachid Guerraoui Rachid Guerraoui is professor in computer science at the school of computer and communication sciences at the Ecole Polytechnique Fédérale de Lausanne (EPFL). Prior to that, Rachid has worked respectively at the Centre de Recherche de l'Ecole des Mines de Paris, the Commissariat à l'Energie Atomique (Paris) and HP Labs Palo Alto.

Petr Kouznetsov Petr Kouznetsov obtained a master degree in mathematics in Saint-Petersburg Institute of Fine Mechanics and Optics (Technical University). Currently, Petr is a Ph.D. student and research assistant in the Distributed Programming Laboratory of the Ecole Polytechnique Fédérale de Lausanne (EPFL).