Mutual Exclusion in Asynchronous Systems with Failure Detectors *†

Carole Delporte-Gallet Hugues Fauconnier
Laboratoire d'Informatique Algorithmique: Fondements et Applications
University of Paris VII

Rachid Guerraoui Petr Kouznetsov
Distributed Programming Laboratory,
Swiss Federal Institute of Technology in Lausanne

Abstract

This paper defines the *fault-tolerant mutual exclusion* problem in a message-passing asynchronous system and determines the weakest failure detector to solve the problem. 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.

1 Introduction

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 [5, 11, 12, 17] involves managing access to a single, indivisible resource that can only support one user at a time (*mutual exclusion* property). The user 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 does not crash) wants to enter its critical section, then there eventually will be *some* correct process in its CS (*progress* property), even if some process crashed while in the critical section.

*This work is partially supported by the Swiss National Science Foundation (project number 510-207).
†Technical report ID:200227

Evidently, the problem cannot be solved deterministically in a crash-prone asynchronous system without any information about failures[1]: there is no way to determine that a process in its CS is crashed or just slow. 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 [6].

To circumvent the impossibility of consensus, Chandra and Toueg [3] 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 [3], it is shown that a rather weak failure detector $\diamond\mathcal{W}$ is sufficient to solve consensus in an asynchronous system with a majority of correct processes, and that $\diamond\mathcal{W}$ can be implemented using partial synchrony assumptions. In [2], it is shown that $\diamond\mathcal{W}$ is also necessary to solve consensus. In short, $\diamond\mathcal{W}$ is the *weakest* failure detector to solve consensus.

A natural question follows: what is the *weakest* failure detector to solve the fault-tolerant mutual exclusion problem? Traditionally, mutual exclusion algorithms either consider a failure-free model [13, 19], or suppose that (1) every crash is eventually detected by every correct process and (2) no correct process is suspected [1, 16]: the conjunction of (1) and (2) is equivalent to the assumption of the *perfect* failure detector $\mathcal{P}$ [3]. 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 [3]).

Roughly speaking, failure detector $\mathcal{T}$ eventually and permanently (1) *trusts* (considers to be *up*) every correct process and (2) does not trust any crashed process. If $\mathcal{T}$ stops trusting a process, then the process is crashed. $\mathcal{T}$ 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 [11, 12]: a process that wishes to enter its CS first gets trusted by *some* correct process, then draws a ticket and is served in the order of its ticket number. $\mathcal{T}$ 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 required by a process, is the permanent stay of some correct process in its CS (*starvation-freedom* property).

Besides, we show that if at least one process can crash, then no algorithm can

[1] We do not consider here probabilistic mutual exclusion algorithms [4, 7].

solve the problem using a strictly weaker failure detector. Intuitively, this stems from the fact that, if a process in its CS does not deliberately resign, another process can enter its CS only if it is sure that the first process is crashed. We use this very fact to extract the information provided by $\mathcal{T}$ from any algorithm that solves fault-tolerant mutual exclusion.

In other words, $\mathcal{T}$ 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 [3]) which is strictly weaker than $\mathcal{P}$ and which is sufficient to solve the problem even with an arbitrary number of failures.

Finally, we turn our attention to group mutual exclusion [8, 9, 10], 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.

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 necessary and sufficient to solve the problem, respectively. 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 a crash-prone asynchronous message passing system model augmented with the failure detector abstraction [2].

System. The system consists of a set of n processes $\Pi = \{1, .., n\}$ $(n > 1)$. 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.[2] We take the range $\mathbb{N}$ of the clock's ticks to be the set of natural numbers.

[2]More precisely, the information about global time can come *only* from failure detectors.

Failures and failure patterns. Processes are subject to *crash* failures. A *failure pattern* F is a function from the global time range $\mathbb{N}$ 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 $correct(F) = \Pi - \cup_{t \in \mathbb{N}} F(t)$, the set of *correct* processes. A process $p \notin F(t)$ is said to be *up* at time t. A process $p \in F(t)$ is said to be *crashed* at time t. 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. $\mathcal{E}_f$ consists of all failure patterns in which up to f processes can crash. We consider $\mathcal{E}_f$ with $0 < f < n$, in which at least one process might crash and at least one process is correct.

Failure detectors. A *failure detector history H with range $\mathcal{R}$* is a function from $\Pi \times \mathbb{N}$ 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. Typically, this information includes the set of processes that i currently suspects to have crashed.[3] Among the failure detectors defined in [3], we consider the following ones, each one defined by a *completeness* and an *accuracy* property:

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.

Algorithms, configurations, schedules, and runs. 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 of the processes. $A(i)$ denotes the automaton running on process i. Computation proceeds in steps of the given algorithm A. In each step of A, process i performs atomically the

[3] In [2], failure detectors can output values from an arbitrary range. In determining the weakest failure detector for our fault-tolerant mutual exclusion problem, we indeed consider the original failure detector model of [2], i.e., we do not make any assumption a priori on the range of a failure detector.

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 $A(i)$, based on its state at the beginning of the step, the message received in the receive phase, and the value that i sees in the failure detector query phase. Note that the received message is chosen *non-deterministically* 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 the current configuration if and only if $m = \lambda$ or m is in the current message buffer.

A *schedule* S of algorithm A is a (finite or infinite) sequence of steps of A. $S_\top$ denotes the empty schedule. We say that a schedule S *is applicable to a configuration* C if and only if (a) $S = S_\top$, 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 $T \subseteq \mathbb{N}$ 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 $t \leq |S|$, if $S[t]$ is of the form (i, m, d, A) then: (1) i has not crashed by time $T[t]$, i.e., $i \notin F(T[t])$ and (2) d is the value of the failure detector module of i at time $T[t]$, i.e., $d = H_\mathcal{D}(i, T[t])$.

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 $T \subseteq \mathbb{N}$ is an *infinite* list of increasing time values indicating when each step S occurred. In addition to satisfying the 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).[4]

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 $\mathcal{D}$.

[4]In fact, in this paper we can succeed with a weaker guarantee, such as "every correct process eventually receives every message sent to it by any *correct* process".

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, $R_{A' \rightarrow A}$, 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* $\mathcal{E}$.

Weakest failure detector. If for failure detectors $\mathcal{D}$ and $\mathcal{D}'$ there is an algorithm $R_{\mathcal{D}' \rightarrow \mathcal{D}}$ that transforms $\mathcal{D}'$ into $\mathcal{D}$ in environment $\mathcal{E}$ ($R_{\mathcal{D}' \rightarrow \mathcal{D}}$ emulates histories of $\mathcal{D}$ using histories of $\mathcal{D}'$), we say that $\mathcal{D}$ *is reducible to* $\mathcal{D}'$ *in* $\mathcal{E}$ ($R_{\mathcal{D}' \rightarrow \mathcal{D}}$ is called a *reduction* algorithm), or $\mathcal{D}$ *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 $R_{\mathcal{D}' \rightarrow \mathcal{D}}$ 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 reducible to $\mathcal{D}'$ in $\mathcal{E}$.

3 The fault-tolerant mutual exclusion problem

We define here the *fault-tolerant mutual exclusion* problem (from now on - FTME) using the terminology and notations given in [14]. We associate to every process $i \in \Pi$ a *user* u_i that can require exclusive access to the critical section. The users can be thought of as application programs.

As in [14], every process $i \in \Pi$ and every user u_i are modelled as state machines. A process $i \in \Pi$ and the corresponding user u_i interact using try_i, $crit_i$, $exit_i$ and rem_i actions. The input actions of process i (and outputs of u_i) are the try_i action, indicating the wish of u_i to enter its CS, and the $exit_i$ action, indicating the wish of u_i to leave its critical section. The output actions of i (and inputs of u_i) are the $crit_i$ action, granting the access to its critical section, and the rem_i action, which tells u_i that it can continue its work out of its critical section.

A sequence of try_i, $crit_i$, $exit_i$ and rem_i actions for the composition (u_i, i) is called a *well-formed execution* if it is a prefix of the cyclically ordered sequence $\{try_i, crit_i, exit_i, rem_i\}$. A user u_i is called a *well-formed user* if it does not violate the cyclic order of actions $try_i, crit_i, exit_i, rem_i, ...$

Given an execution of (u_i, i), we say that process i is

- in its *remainder section* (a) initially or (b) in between any rem_i action and the following try_i action;
- in its *trying section* (or i is a *volunteer*) in between any try_i action and the following $crit_i$ action;
- in its *critical section (CS)* in between any $crit_i$ action and the following (a) $exit_i$ action or (b) crash of i;

- in its *exit section* (or i is a *resigner*) in between any $exit_i$ action and the following rem_i action.

A mutual exclusion algorithm defines trying and exit protocols for every process i. We say that the algorithm solves the FTME problem if, under the assumption that every user is well-formed, any run of the algorithm satisfies the following properties:

Well-formedness: For any $i \in \Pi$, the execution describing the interaction between u_i and i is well-formed.

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

Progress:

(1) If a correct process volunteers, then at some time later some correct process is in its CS.

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

We will show in Section 6 that any algorithm that solves the FTME problem with $\mathcal{T}$ in an environment with a majority of correct processes 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 volunteers enters its CS at some time later.

4 The trusting failure detector

This section introduces a new failure detector that we call the *trusting* failure detector and we denote by $\mathcal{T}$. Failure detector $\mathcal{T}$ is such that $\mathcal{R}_{\mathcal{T}} = 2^{\Pi}$ and $H_{\mathcal{T}}(i,t)$ represents the set of processes that process i *trusts* (i.e., that i considers to be up) at time 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:

Trusting completeness

Eventually, no crashed process is trusted by any correct process. That is:

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

Trusting accuracy

(1) Eventually, every correct process is permanently trusted by every correct process. That is:

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

(2) Every process j that stops being trusted by a process i is crashed. That is:

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

Example. Figure 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 $\{1\}$: $H(1,t_1) = \{1\}$, i.e., process 1 trusts only itself. At time $t_2 > t_1$, processes 2 and 3 also get trusted by process 1: $H(1,t_2) = \{1,2,3\}$. Process 3 crashes and at some time later is not trusted anymore by process 1 : $\forall t \geq t_3, H(1,t) = \{1,2\}$. Note that the id of crashed process 4 is never output by the module of $\mathcal{T}$ at process 1, that is, process 1 never trusts process 4.

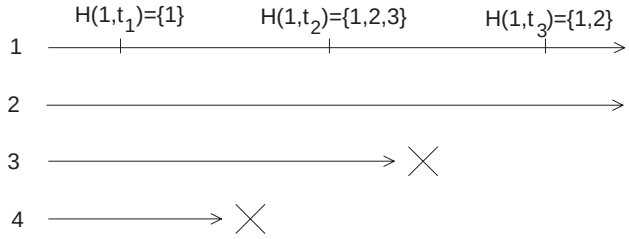

Figure 1: Failure detection scenario for $\mathcal{T}$.

Now we identify the position of $\mathcal{T}$ in the hierarchy of failure detectors introduced in [3].

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

Proof:
(a) We first show that $\mathcal{T}$ is reducible to $\mathcal{P}$. For this we define a reduction algorithm $R_{\mathcal{P}\rightarrow\mathcal{T}}$ by outputting $H_{\mathcal{T}}(i,t) = \Pi \backslash H_{\mathcal{P}}(i,t)$, $\forall t \in \mathbb{N}, \forall i \notin F(t)$.

Trusting completeness follows directly from the strong completeness property of $\mathcal{P}$. Indeed, if every crashed process is eventually suspected by every correct process, then eventually no crashed process is trusted.

Trusting accuracy follows from (1) the fact that initially every process is trusted and (2) the strong accuracy property of $\mathcal{P}$. Indeed, since $\mathcal{P}$ does not make mistakes, correct processes are always trusted. If a process stops being trusted, then it is crashed.
(b) Now we show that $\mathcal{P}$ is not reducible to $\mathcal{T}$. Intuitively, the proof follows from the fact that $\mathcal{T}$ is allowed not to give any information about a crashed process (see process 4 in the scenario of Figure 1).

Assume that there exists a reduction algorithm $R_{\mathcal{T}\rightarrow\mathcal{P}}$ that, for any failure pattern $F \in \mathcal{E}_f$, constructs $H_{\mathcal{P}}$ from $H_{\mathcal{T}}$, such that $H_{\mathcal{P}} \in \mathcal{P}(F)$.

Consider failure pattern $F_1 \in \mathcal{E}_f$, such that $F_1(0) = \{j\}$, $correct(F_1) = \Pi \backslash \{j\}$ (the only crashed process j is initially crashed) and take a history $H^1_{\mathcal{T}} \in$

$\mathcal{T}(F_1)$, such that $H^1_{\mathcal{T}}(i,t) = \Pi \backslash \{j\}$, $\forall i \neq j, \forall t \in \mathbb{N}$ (remember that we consider an environment where at least one process can crash). Consider run $R_1 = \langle F_1, H^1_{\mathcal{T}}, I, S_1, T\rangle$ of $R_{\mathcal{T}\rightarrow\mathcal{P}}$ that outputs a history $H^1_{\mathcal{P}} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists t_0 \in \mathbb{N}, i \in \Pi \backslash j : H^1_{\mathcal{P}}(i,t_0) = \{j\}$.

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 $\forall t \in \mathbb{N}$:

$$H^2_{\mathcal{T}}(i,t) = \begin{cases} H^1_{\mathcal{T}}(i,t), & t \leq t_0 \\ \{j\} \cup H^1_{\mathcal{T}}(i,t), & t > t_0 \end{cases}$$

Clearly, $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$. Consider run $R_2 = \langle F_2, H^2_{\mathcal{T}}, I, S_2, T\rangle$ of $R_{\mathcal{T}\rightarrow\mathcal{P}}$ that outputs a history $H^2_{\mathcal{P}} \in \mathcal{P}(F_2)$. Moreover, take $S_1[t] = S_2[t], \forall t \leq t_0$. Thus, j takes no steps in S_2 for all $t \leq t_0$. Since partial runs of R_1 and R_2 for $t \leq t_0$ are identical, the resulting history $H^2_{\mathcal{P}}$ is such that $H^2_{\mathcal{P}}(i,t_0) = \{j\}$, for some $i \in \Pi$. In other words, a correct process is suspected, and the strong accuracy of $\mathcal{P}$ is violated.

Thus, we come to a contradiction and $\mathcal{T} \prec_{\mathcal{E}_f} \mathcal{P}$. □

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

Proof:
(a) We first define a reduction algorithm outputting $H_{\diamond\mathcal{P}}(i,t) = \Pi \backslash H_{\mathcal{T}}(i,t)$, $\forall t \in \mathbb{N}, \forall i \notin F(t)$. Assume that the strong completeness property of $\diamond\mathcal{P}$ is violated. More precisely,

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

Hence, a crashed process is permanently trusted by some correct process - contradicting the trusting completeness property of $\mathcal{T}$.

Assume that the eventual strong accuracy property of $\diamond\mathcal{P}$ is violated. More precisely,

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

Hence, a correct process never becomes permanently trusted by some correct process - contradicting the trusting accuracy property of $\mathcal{T}$. Thus, $\diamond\mathcal{P} \preceq \mathcal{T}$.
(b) Now we show that $\mathcal{T}$ is not reducible to $\diamond\mathcal{P}$. Indeed, assume that there exists a reduction algorithm $R_{\diamond\mathcal{P}\rightarrow\mathcal{T}}$ that, for any failure pattern $F \in \mathcal{E}_f$, constructs $H_{\mathcal{T}}$ from $H_{\diamond\mathcal{P}}$, such that $H_{\mathcal{T}} \in \mathcal{T}(F)$.

Consider a failure-free pattern $F_1 \in \mathcal{E}_f$ and take $H^1_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_1)$, such that $\forall i, \forall t \in \mathbb{N} : H^1_{\diamond\mathcal{P}}(i,t) = \varnothing$. Consider a run $R_1 = \langle F_1, H^1_{\diamond\mathcal{P}}, I, S_1, T\rangle$ of $R_{\diamond\mathcal{P}\rightarrow\mathcal{T}}$ that outputs a history $H^1_{\mathcal{T}} \in \mathcal{T}(F_1)$. By the trusting accuracy (1) property of $\mathcal{T}$, there exists a time $t_0 \in \mathbb{N}$, such that $\forall t \geq t_0$ and $\forall i \in \Pi$: $H^1_{\mathcal{T}}(i,t) = \Pi$.

Now consider a failure pattern $F_2 \in \mathcal{E}_f$ in which j crashes at time $t_0 + 1$. More precisely, $\forall t \in \mathbb{N}$:

$$F_2(t) = \begin{cases} \varnothing, & t \leq t_0 \\ \{j\}, & t > t_0 \end{cases}$$

Take $H^2_{\diamond\mathcal{P}} \in \diamond\mathcal{P}(F_2)$, such that for all $t \in \mathbb{N}$ and $i \in \Pi$:

$$H^2_{\diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\diamond\mathcal{P}}(i,t), & t \leq t_0 \\ H^1_{\diamond\mathcal{P}}(i,t) \cup \{j\}, & t > t_0 \end{cases}$$

Now consider a run $R_2 = \langle F_2, H^2_{\diamond\mathcal{P}}, I, S_2, T\rangle$ of $R_{\diamond\mathcal{P}\rightarrow\mathcal{T}}$ that outputs a history $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$. Assume that $S_1[t] = S_2[t]$, $\forall t \leq t_0$. Clearly, for all $i \in \Pi$, $H^2_{\mathcal{T}}(i,t_0) = \Pi$. By the trusting completeness property of $\mathcal{T}$, there exists a time $t_1 > t_0$, such that $\forall i \neq j$: $H^2_{\mathcal{T}}(i,t_1) = \Pi\backslash\{j\}$.

Construct a history $H^3_{\diamond\mathcal{P}}$, such that for all $t \in \mathbb{N}$ and $i \in \Pi$:

$$H^3_{\diamond\mathcal{P}}(i,t) = \begin{cases} H^1_{\diamond\mathcal{P}}(i,t), & t \leq t_0 \\ H^2_{\diamond\mathcal{P}}(i,t), & t_0 < t \leq t_1 \\ \varnothing, & t > t_1 \end{cases}$$

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

Finally consider a run $R_3 = \langle F_1, H^3_{\diamond\mathcal{P}}, I, S_3, T\rangle$ that outputs a history $H^3_{\mathcal{T}} \in \mathcal{T}(F_1)$. Assume that $S_3[t] = S_2[t]$, $\forall t \leq t_1$. Since partial runs of R_2 and R_3 for $t \leq t_0$ are identical, there exists $i \neq j$, such that:

$$H^3_{\mathcal{T}}(i,t_0) = \Pi,$$
$$H^3_{\mathcal{T}}(i,t_1) = \Pi\backslash\{j\}.$$

In other words, j stops being trusted by i at time t_1. By the trusting accuracy (2) property of $\mathcal{T}$, j is crashed in F_1, which contradicts the assumption that F_1 is failure-free.

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

5 The necessary condition

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 using a failure detector $\mathcal{D}$. A reduction algorithm $R_{\mathcal{D}\rightarrow\mathcal{T}}$ is presented in Figure 2. At any time $t \in \mathbb{N}$ and for any process $i \in \Pi$, $R_{\mathcal{D}\rightarrow\mathcal{T}}$ outputs the set of processes trusted by i, $output_i(t)$.

The processes run n instances of algorithm A: $f_1, .., f_n$. The interaction between process i and instance f_j is defined through the actions try_{ij}, $crit_{ij}$, $exit_{ij}$ and rem_{ij} (each process i acts as a user of a critical section provided by f_j). We assume that if each f_i is used correctly (its users are well-formed), then f_i guarantees the properties of FTME.

The idea of the algorithm is the following. Initially, every process i is the only volunteer in its own instance f_i of A. Eventually, i crashes or enters its CS and sends message $[me, i, i]$ to all. Every correct process that has received $[me, i, i]$ starts trusting i and volunteers in instance f_i. Processes can leave their CSs only because of a crash. Thus, if a correct process $j \neq i$ enters its CS in f_i, then i has crashed. Process j sends message $[me, i, j]$ to all and every

correct process eventually stops trusting i. As a result, eventually, every correct process is permanently trusted by every correct process and no crashed process is trusted by any correct process. Moreover, the only reason to stop trusting a process i, is a crash of i.

```
1:  output_i := ∅                                          {* Initialization *}
2:  crashed_i := ∅

3:  try_ii

4:  upon crit_ij do
5:     send [me, j, i] to all

6:  upon receive [me, j, k] do
7:     if j = k and j ∉ crashed_i then
8:        output_i := output_i ∪ {j}
9:        if j ≠ i then
10:          try_ij
11:    else
12:       crashed_i := crashed_i ∪ {j}
13:       output_i := output_i \ {j}
```

Figure 2: Reduction algorithm $R_{\mathcal{D}\to\mathcal{T}}$ - process i.

Lemma 3 *The output of the reduction algorithm of Figure 2 satisfies the properties of the trusting failure detector $\mathcal{T}$.*

Proof: According to the algorithm of Figure 2, no process i volunteers twice in the same instance f_j or resigns. Thus, each i is well-formed with respect to each f_j.

Assume that the trusting completeness property of $\mathcal{T}$ is violated. More precisely,

$$\exists F,\ \exists i \in \mathit{correct}(F),\ \exists j \notin \mathit{correct}(F) : \forall t, \exists t' > t, j \in \mathit{output}_i(t').$$

By the algorithm of Figure 2, i volunteers in f_j at some time t_0. Due to the *progress* property of FTME, at some time $t_1 > t_0$, some correct process m enters its CS in f_j and sends $[me, j, m]$ to all. At some time $t_2 > t_1$, process i receives $[me, j, m]$, includes process j into $\mathit{crashed}_i$ and removes j from output_i. Since j stays forever in $\mathit{crashed}_i$, $\forall t > t_2 : j \notin \mathit{output}_i(t)$ - a contradiction. Thus, trusting completeness property of $\mathcal{T}$ is satisfied.

Assume that the first part of trusting accuracy is violated. More precisely,

$$\exists F, \exists i \in \mathrm{correct}(F), \exists j \in \mathrm{correct}(F), \forall t, \exists t' > t : j \notin \mathit{output}_i(t').$$

Two cases are possible:

(1) i never trusts j $\Rightarrow$ i never receives $[me, j, j]$ from j $\Rightarrow$ the only correct volunteer j never enters its CS in f_j - *progress* property is violated in f_j.

(2) i stops trusting j at time t_0 $\Rightarrow$ some process k entered its CS in f_j at time $t_1 < t_0$ $\Rightarrow$ either j is in its CS in f_j at t_1 - f_j violates *mutual exclusion* property, or j left its CS in f_j at $t < t_1$, not crashed and not resigned - *well-formedness* property is violated in f_j.

Assume that the second part of trusting accuracy is violated. More precisely,

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

By the algorithm of Figure 2, i stops trusting j only if some process $k \neq j$ enters its CS in f_j at some time t_0 and only if at some time $t_1 < t_0$ j entered its CS in f_j. But according to *mutual exclusion*, j had to leave its CS at time $t_2 < t_0$. Since j never resigns, it could leave its CS in f_j only if it crashes (due to the *well-formedness* property) - a contradiction.

Thus, the reduction algorithm of Figure 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

We give in Figure 3 an algorithm solving FTME using $\mathcal{T}$ in any environment $\mathcal{E}_f$ with a majority of correct processes ($f < \lceil \frac{n}{2} \rceil$). The algorithm uses the fact that $\diamond\mathcal{P} \preceq_{\mathcal{E}_f} \mathcal{T}$. More precisely, with $\mathcal{T}$ and the assumption of a majority of correct processes, we can implement a *total order broadcast*[5] algorithm [3].

Our algorithm of Figure 3 assumes:

- an algorithm implementing total order broadcast is provided through two primitives available at every process: TO-Broadcast() and TO-Deliver() (the latter returns the next delivered message in the total order, if there is no such message, it waits until the next message is delivered);
- that each process i has access to the output of its trusting failure detector module $\mathcal{T}_i$;
- environment $\mathcal{E}_f$ such that $f < \lceil \frac{n}{2} \rceil$;
- that each user u_i, $i \in \Pi$, is well-formed.

[5]Total order broadcast satisfies the following properties: *(1) validity* (i.e., if a correct process TO-Broadcasts a message m, it eventually TO-Delivers m); *(2) agreement* (i.e., if a process TO-Delivers a message m, every correct process eventually TO-Delivers m); *(3) integrity* (i.e., every message is TO-Delivered at most once, and only if the message was previously TO-Broadcast); *(4) total-order* (i.e., if a process i TO-Delivers m before m', then no process j can TO-Deliver m' without having TO-Delivered m).

```
1:  ready_i := false                                          {* Initialization *}
2:  r_i := 0
3:  trusted_i := ∅

4:  send [me, i] to all
5:  wait until received (n − f) [ack]
6:  ready_i := true
7:  while true do
8:    [j, k] := TO-Deliver()                                  {* wait for the next candidate *}
9:    if i = j then
10:     crit_i
11:   else
12:     wait until((j ∈ trusted_i and j ∉ T_i) or
          received [exit, j, k] or
          received [crash, j])
13:     if not received [exit, j, k] then
14:       send [crash, j] to all
15:     else
16:       send [exit, j, k] to all

17: upon receive [me, m] do
18:   wait until m ∈ T_i
19:   trusted_i := trusted_i ∪ m
20:   send [ack] to m

21: upon try_i do
22:   wait until ready_i
23:   r_i := r_i + 1
24:   TO-Broadcast([i, r_i])

25: upon exit_i do
26:   send [exit, i, r_i] to all
27:   rem_i
```

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

In our algorithm of Figure 3, each process i has the following variables:

1. a boolean $ready_i$, initially *false*, indicating whether i is ready to execute the trying protocol;
2. a list $trusted_i$, initially empty, of processes currently trusted by i;
3. an integer r_i, initially 0, indicating the number of times i has volunteered.

The idea of our algorithm is inspired by the well-known Bakery algorithm of Lamport [11, 12]: the processes that wish to enter their CSs first draw tickets and then are served in the order of their tickets numbers. First every candidate asks for a permission to proceed from some *correct* process and does not take steps (wait clause in line 5 in Figure 3) 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 that guarantees that the requests are eventually delivered in the same order by every correct process (line 8 in Figure 3). No candidate i can be served unless every candidate in the line before i has been served and has released the resource, or crashed (line 12 in Figure 3).

Now we prove the correctness of the algorithm of Figure 3. Let R be an arbitrary run of the algorithm for some failure pattern $F \in \mathcal{E}_f$ ($f < \lceil \frac{n}{2} \rceil$). First we prove Lemmata 5-7 for R.

Lemma 5 *For any $i \in \Pi$, the execution describing the interaction between u_i and i is well-formed.*

Proof: All users are well-formed. Thus, to complete the proof it is enough to show that i is not *the first* to violate the cyclic order of actions try_i, $crit_i$, $exit_i$, rem_i. More precisely, $\forall i \in \Pi$:

- the preceding action of each $crit_i$ is try_i, and
- the preceding action of each rem_i is $exit_i$.

The two properties follow from the code presented in Figure 3. □

Lemma 6 *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, $r_i = k$ and $r_j = l$. According to the algorithm, j can be in the CS at t_0 only if the corresponding TO-Deliver($[j, l]$) occurred at some time $t_1 < t_0$. By the ordering property of TO-Broadcast, one of the processes i and j delivered both messages $[i, k]$ and $[j, l]$. Without loss of generality, assume that TO-Deliver($[i, k]$) precedes TO-Deliver($[j, l]$) at process j (otherwise we can swap the indices i and j). That means that at some time $t_2 < t_1$, j passed the wait clause in line 12 while processing $[i, k]$. Thus, one of the following events occurred before t_2 at j:

(1) $i \in trusted_j$ **and** $i \notin \mathcal{T}_j$: by the trusting accuracy (2) property of $\mathcal{T}$, i is crashed at t_2. But it is in the CS at $t_0 > t_2$: a contradiction.

(2) j received $[exit, i, k]$: by the algorithm of Figure 3, i resigned from the CS with $r_i = k$ at t_2. But i is in the CS at $t_0 > t_2$: a contradiction.

(3) j received $[crash, i]$: by the algorithm of Figure 3, at some process m, at some time $t_3 < t_2$ the following is true: $i \in trusted_m$ and $i \notin \mathcal{T}_m$ (m stops trusting i). By the trusting accuracy (2) property of $\mathcal{T}$, i is crashed at t_2. But it is in the CS at $t_0 > t_2$: a contradiction.

Hence, mutual exclusion is guaranteed. □

Lemma 7 *If a correct process volunteers, then at some time later some correct process is in its CS. If a correct process resigns, then at some time later it enters its remainder section.*

Proof: Assume that a correct process i volunteers at time t_0 with $r_i = k$, and no correct process is ever in its CS after t_0. According to the algorithm, after t_0, process i never reaches line 10 of the algorithm. In other words, i is blocked at some wait clause. The first wait clause (line 5 in Figure 3) is not able to block the process, due to the trusting accuracy (1) property of $\mathcal{T}$ and the fact that $(n - f)$ processes are correct. Thus, eventually, $ready_i = true$, and wait clause in line 22 in Figure 3 cannot block the process neither. Thus, i issues TO-Broadcast($[i, k]$). The second wait clause (more precisely, the statement in line 8 in Figure 3) is not blocking neither, because of the guarantee that any broadcast message is eventually delivered by every correct process. Thus, i is blocked in the third clause (line 12 in Figure 3) while processing some $[j, l]$ $(i \neq j)$. Formally, none of the following is ever satisfied at i:

(1) $j \in trusted_i$ **and** $j \notin \mathcal{T}_i$;

(2) i received $[exit, j, l]$;

(3) i received $[crash, j]$.

We show first that if a correct process i is blocked while processing some entry $[j, l]$, then all correct processes are blocked while processing $[j, l]$. Indeed, assume that there exists a correct process m that is not blocked while processing $[j, l]$. That is, one of the following events occurred at m:

1. $m = j$ and j enters its CS. According to the assumption of the proof, no correct process is in its CS after t_0, thus, j resigned before t_0 and sent $[exit, j, l]$ to all. Correct process i eventually receives the message and releases : a contradiction.

2. $(j \in trusted_m$ and $j \notin \mathcal{T}_m) \Rightarrow j$ is crashed (by the trusting accuracy (2) property of $\mathcal{T}$) $\Rightarrow m$ sends $[crash, j]$ to all. Correct process i eventually receives the message and releases : a contradiction.

3. m received $[exit, j, l]$ and forward it to all. Correct process i eventually receives $[exit, j, l]$ and releases : a contradiction.

4. m received $[crash, j]$ and forward it to all. Correct process i eventually receives $[crash, j]$ and releases : a contradiction.

Thus, all correct processes are blocked while processing $[j, l]$. Process j is obviously crashed (if j is correct, then it is also blocked while processing $[j, l]$: a contradiction with the algorithm). By the algorithm, before invoking TO-Broadcast($[j, l]$), j received $(n - f)$ messages of type $[ack]$ (line 5 in Figure 3). Since at least $(n - f)$ processes are correct and $f < \lceil \frac{n}{2} \rceil$, there is at least one correct process m, such that $j \in trusted_m$ at some time t. By the trusting completeness property of $\mathcal{T}$, the condition ($j \in trusted_m$ and $j \notin \mathcal{T}_m$) is eventually satisfied. Hence, m can not be blocked while processing $[j, l]$: a contradiction.

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

Thus, progress is guaranteed. □

The following theorem comes directly after Lemmata 5, 6 and 7:

Theorem 8 *The algorithm of Figure 3 solves FTME using $\mathcal{T}$, in any environment $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$.*

Finally, combining this with the result of Section 5 we can state the following theorem:

Theorem 9 *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 Figure 3 solves a harder problem that, in addition to well-formedness, mutual exclusion and progress, satisfies also the starvation-freedom property of Section 3.

Indeed, assume that i is a correct volunteer 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 processed: if any process releases its CS, no process can be blocked in a wait clause (see line 12 in Figure 3). Finally, i eventually reaches its own entry $[i, k]$ in the total order and i enters its CS.

From Theorem 9 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 not only solves FTME, but also guarantees the starvation freedom property.

7 On the number of correct processes

Proposition 10 *No algorithm solves FTME using $\mathcal{T}$ in any environment $\mathcal{E}_f$ where $f \geq \lceil \frac{n}{2} \rceil$.*

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 from Y takes any step in R_1 and no process from X ever trusts any process from Y. Assume that correct process $i \in X$ is the only volunteer. By the *progress* property of FTME, i invokes $crit_i$ at some time t_1.

(2) R_2: no process from X takes any step in R_2, no process from Y takes any step before $t_1 + 1$ and no process from Y ever trusts any process from X. Assume that correct process $j \in Y$ is the only volunteer. By the *progress* property of FTME, j invokes $crit_j$ at some time t_2. Clearly, $t_1 < t_2$.

Assume that no user ever resigns in R_1 and R_2. We construct a run R that consists of the steps from $R_1 \cup R_2$ made before t_2. Now assume that every

process is correct in R, that the processes from X and Y start to trust each other *later* than t_2 (this is a valid history of $\mathcal{T}$) and that all messages sent between X and Y are delayed until $t_2 + 1$. Evidently, R is a valid run of A. But, since u_i and u_j never resign, they can not leave their CSs (due to the *well-formedness* property of FTME), at time $t_2 + 1$ both i and j are in their CSs - a contradiction with the *mutual exclusion* property of FTME. □

What happens in an environment where up to $n - 1$ processes can crash? Consider the extreme case of an environment $\mathcal{E}_f$, where $f = n - 1$. Is $\mathcal{P}$ the weakest failure detector to solve the problem in $\mathcal{E}_{n-1}$? A close look at the correctness proof for the algorithm of Figure 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 $\mathcal{S}$ [3] 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 $\mathcal{S}$ and the wait clause at line 5 of the algorithm in Figure 3 can be substituted for:

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 trusting accuracy (1) 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 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 11 *$\mathcal{T} + \mathcal{S} \prec_{\mathcal{E}_f} \mathcal{P}$, in any environment $\mathcal{E}_f$ with $0 < f < n$.*

Proof:

(a) $\mathcal{S} \prec_{\mathcal{E}_f} \mathcal{P}$ [3] and $\mathcal{T} \prec_{\mathcal{E}_f} \mathcal{P}$ (Proposition 2). That is, both $\mathcal{T}$ and $\mathcal{S}$ are reducible to $\mathcal{P}$. Thus, $\mathcal{T} + \mathcal{S} \preceq_{\mathcal{E}_f} \mathcal{P}$.

(b) Now we show that $\mathcal{P}$ is not reducible to $\mathcal{T} + \mathcal{S}$. Indeed, assume there exists an algorithm $R_{\mathcal{T}+\mathcal{S}\rightarrow\mathcal{P}}$ 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 $H_\mathcal{P} \in \mathcal{P}(F)$.

Consider failure pattern $F_1 \in \mathcal{E}_f$, such that $F_1(0) = \{j\}$, $correct(F_1) = \Pi\backslash\{j\}$ and take histories $H^1_\mathcal{T} \in \mathcal{T}(F_1)$ and $H^1_\mathcal{S} \in \mathcal{S}(F_1)$. Assume that the corresponding run $R_1 = \langle F_1, (H^1_\mathcal{T}, H^1_\mathcal{S}), I, S_1, T\rangle$ of $R_{\mathcal{T}+\mathcal{S}\rightarrow\mathcal{P}}$ outputs a history $H^1_\mathcal{P} \in \mathcal{P}(F_1)$. By the strong completeness property of $\mathcal{P}$: $\exists t_0 \in \mathbb{N}, i \in \Pi\backslash\{j\}$: $H^1_\mathcal{P}(i, t_0) = \{j\}$.

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 $\forall t \in \mathbb{N}$:

$$H^2_{\mathcal{T}}(i,t) = \begin{cases} H^1_{\mathcal{T}}(i,t), & t \leq t_0 \\ H^1_{\mathcal{T}}(i,t) \cup \{j\}, & t > t_0 \end{cases}$$

$$H^2_{\mathcal{S}}(i,t) = \begin{cases} H^1_{\mathcal{S}}(i,t), & t \leq t_0 \\ H^1_{\mathcal{S}}(i,t) \backslash \{j\}, & t > t_0 \end{cases}$$

Clearly, $H^2_{\mathcal{T}} \in \mathcal{T}(F_2)$ and $H^2_{\mathcal{S}} \in \mathcal{S}(F_2)$. Consider a run $R_2 = \langle F_2, H^2_{\mathcal{T}} \oplus H^2_{\mathcal{S}}, I, S_2, T \rangle$ of $R_{\mathcal{T}+\mathcal{S}\rightarrow\mathcal{P}}$ that outputs a history $H^2_{\mathcal{P}} \in \mathcal{P}(F_2)$, where $S_1[t] = S_2[t], \forall t \leq t_0$. Thus, j takes no steps in S_2 for all $t \leq t_0$. Since partial runs of R_1 and R_2 for $t \leq t_0$ are identical, the resulting history $H^2_{\mathcal{P}}$ is such that $H^2_{\mathcal{P}}(i, t_0) = \{j\}$, for some $i \in \Pi$. In other words, a correct process is suspected, 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 processes can crash. Determining the weakest failure detector to solve FTME in such environment is an open issue.

8 Group mutual exclusion

Group mutual exclusion [8, 9, 10] is a natural generalization of the classical mutual exclusion problem [5, 12], where a process wishing to enter its CS volunteers for a "session". Processes that request different sessions cannot be in their CSs at a time, but processes that request the same session can. 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.

Thus, in addition to the well-formedness and progress properties of FTME, fault-tolerant group mutual exclusion (FTGME) satisfy also the following properties (we follow the terminology used in Section 3):

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

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

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. In contrast to [8, 9], we do not make the assumption that a process can stay in its CS for a finite time only. Evidently, 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.

Clearly, FTGME is at least as hard as FTME: we can easily implement FTME from FTGME just associating every process with unique session. On

the other hand, we show here that $\mathcal{T}$ solves FTGME in a system with a majority of correct processes. Thus, FTME and FTGME are equivalent in our model.

In Figure 4, we present an algorithm that solves FTGME using $\mathcal{T}$. The $try_i(s)$ action indicates the wish of user u_i to enter its CS with session s. The $crit_i(s)$ action grants user u_i access to its CS with session s. Each process i uses the following variables:

1. a boolean $ready_i$, initially *false*, indicating whether i is ready to execute the trying protocol;
2. a list $trusted_i$, initially empty, of processes currently trusted by i;
3. an integer r_i, initially 0, indicating the number of requests for the CS that u_i has made;
4. an integer sn_i, indicating the id of the currently processed session;
5. a FIFO list $incs_i$ of processes requesting sn_i, $first(incs_i)$ dequeues the first element of a non-empty list $incs_i$.

The algorithm is similar to one 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() (the latter returns the next broadcast message in the total order). 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 different session is satisfied until the processes requested earlier the session s have resigned or crashed.

Now we state the correctness of the algorithm through Lemmata 12-15.

Lemma 12 *For any $i \in \Pi$, the execution describing the interaction between u_i and i is well-formed.*

Proof: Follows from the algorithm and the fact that all u_i ($i \in \Pi$) are well-formed. □

Lemma 13 *If two processes are in their critical sections at the same time, then they request for the same session.*

Proof: By contradiction, assume that processes i and j requesting sessions, respectively, s_i and s_j ($s_i \neq s_j$) are in their CSs at time t_0. Let, at time t_0, $r_i = k$ and $r_j = l$. According to the algorithm, j can be in the CS at t_0 only if the corresponding TO-Deliver($[j, l, s_j]$) occurred at some time $t_1 < t_0$ and, at t_1, $sn_j = s_j$. By the ordering property of TO-Broadcast, one of the processes i and j delivered both messages $[i, k, s_i]$ and $[j, l, s_j]$. Without loss of generality, assume that TO-Deliver($[i, k, s_i]$) precedes TO-Deliver($[j, l, s_j]$) at process j (otherwise we can swap the indices i and j). Hence, at some time $t_2 < t_1$, the *wait* clause in line 20 for (i, k) was overcome at process j. Thus, one of the following events occurred before t_2 at process j:

```
1: ready_i := false                                          {* Initialization *}
2: trusted_i := ∅
3: r_i := 0

4: send [me, i] to all
5: wait until receive (n − f) [ack]
6: ready_i := true
7: [j, k, s] := TO-Deliver()                                 {* wait for the next candidate *}
8: sn_i := s
9: while true do
10:   incs_i := ∅
11:   repeat
12:     if i = j then
13:       crit_i(s)
14:     incs_i := incs_i.(j, k)
15:     [j, k, s] := TO-Deliver()                            {* wait for the next candidate *}
16:   until (sn_i ≠ s)
17:   sn_i := s
18:   while incs_i ≠ ∅ do
19:     (l, m) := first(incs_i)
20:     wait until ((l ∈ trusted_i and l ∉ T_i) or
        received [exit, l, m] or
        received [crash, l])
21:     if not received [exit, l, m] then
22:       send [crash, l] to all
23:     else
24:       send [exit, l, m] to all

25: upon receive [me, m] do
26:   wait until m ∈ T_i
27:   trusted_i := trusted_i ∪ m
28:   send [ack] to m

29: upon try_i(u) do
30:   wait until ready_i
31:   r_i := r_i + 1
32:   TO-Broadcast([i, r_i, u])

33: upon exit_i do
34:   send [exit, i, r_i] to all
35:   rem_i
```

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

(1) $i \in trusted_j$ and $i \notin \mathcal{T}_j$: by the trusting accuracy (2) property of $\mathcal{T}$, i is crashed at t_2. But i is in the CS at $t_0 > t_2$: a contradiction.

(2) j received $[exit, i, k]$: by the algorithm of Figure 4, i resigned from the CS with $r_i = k$ at t_2. But i is in the CS at $t_0 > t_2$: a contradiction.

(3) j received $[crash, i]$: by the algorithm of Figure 4, at some process m, at some time $t_3 < t_2$ the following is true: $i \in trusted_m$ and $i \notin \mathcal{T}_m$ (m stops trusting i). By the trusting accuracy (2) property of $\mathcal{T}$, i is crashed at t_2. But it is in the CS at $t_0 > t_2$: a contradiction.

Hence, mutual exclusion is guaranteed. □

Lemma 14 *If a correct process volunteers, then at some time later some correct process is in its CS. If a correct process resigns, then at some time later it enters its remainder section.*

Proof: Assume that a correct process i volunteers at time t_0 with $r_i = k$, and no correct process ever is in its CS after t_0. Hence, according to the algorithm, i never reaches line 13 of the algorithm. Applying the same arguments as in the proof of Lemma 7, we come to a conclusion that i is blocked in the third clause (line 20 in Figure 3) while processing some (l, m) $(i \neq j)$. Formally, none of the following is ever satisfied at i:

(1) $l \in trusted_i$ **and** $l \notin \mathcal{T}_i$;

(2) received $[exit, l, m]$;

(3) received $[crash, l]$.

Analogously to the proof of Lemma 7, we can easily show that all correct processes are blocked while processing (l, m). Two cases are possible:

1. Process l is correct, then it is also blocked while processing (l, m). According to the algorithm, $l \in incs_l$ and, since l is correct and no $[exit, l, m]$ is received by l, l is in its CS : a contradiction.

2. Process l crashes at some moment, then eventually some correct process stops trusting it and sends $[crash, l]$ to all. The message is eventually received by i : a contradiction.

The second part of the lemma follows directly from the algorithm. Thus, progress in the sense of FTME is guaranteed. □

Lemma 15 *If a correct process i volunteers and no other process volunteers for a different session, then i eventually enters its CS.*

Proof: Assume that, at time t_0, a process i requests a session s with $r_i = k$ and no other process requests a different session. Thus, all processes requesting different sessions have resigned before t_0. Thus, for every request $[l, m, r]$ $(r \neq$

s), the entity (l, m) in the corresponding wait clause (line 20 in Figure 4) cannot block process i. Thus, eventually, i reaches the point when $[i, k, s]$ is processed (line 12) and enters its CS (line 13). □

Finally, we can state the following theorem:

Theorem 16 *For any environment $\mathcal{E}_f$ with $f < \lceil \frac{n}{2} \rceil$, $\mathcal{T}$ is the weakest failure detector to solve GFTME in $\mathcal{E}_f$.*

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

Analogously, in case when up to n processes can crash, we can solve FTGME with $\mathcal{T} + \mathcal{S}$, simply by substituting line 5 of the algorithm in Figure 4 can be substituted 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 (Figure 3) with the well-known algorithms of [15] and [18].[6]

The performance of mutual exclusion algorithms can be measured through the following metrics [19]: (a) the *bootstrapping delay*, which is the time required for a new user 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 user leaves the CS and before the next user enters the CS, and (d) the *response time*, which is the time interval a request 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 volunteers 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 average message propagation delay, and e_c the average CS execution time. The bootstrapping delay of our algorithm (Figure 3) is about $2t_c$: before volunteering for its CS, every process should receive the acknowledgement from a majority of the processes. The algorithm has a relatively high message complexity: each request for a CS requires $O(n^2)$ messages per CS invocation. The synchronization delay is low - t_c: that is, it requires only one communication step to inform the next waiting process that it can enter the

[6] The algorithms of [15] and [18] were designed for the failure-free asynchronous model but could be ported into the crash-prone model assuming $\mathcal{P}$. It would be interesting to determine the costs of implementing the algorithms in the crash-prone model. More details on the comparative analysis of the algorithms of [15] and [18] are available in [19].

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 Figure 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 Figure 3 could be optimized, e.g., by breaking the encapsulation of the total order broadcast box.

Metrics	**Maekawa** [15]	**RA** [18]	**$\mathcal{T}$-based**
Bootstrapping delay	0	0	$2t_c$
Number of messages	Low	Moderate	High
Sync. delay	$2t_c$ (deadlock-prone) t_c (deadlock-free)	t_c	t_c
Response time			
low load	$2t_c$	$2t_c$	$2t_c$
high load	$n(2t_c + e_c)$	$n(t_c + e_c)$	$n(t_c + e_c)$

Figure 5: Comparative performance analysis of mutual exclusion algorithms.

10 Concluding remark

Is it 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".

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, $\mathcal{T}$'s implementation has certain advantages comparing to $\mathcal{P}$'s (given synchrony assumptions). For example, in a possible implementation of $\mathcal{T}$, every process i can 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. In contrast, $\mathcal{P}$ does not allow this kind of "fine-tuning" of t_{ij}: there exists a maximal possible timeout Δ, such that i starts suspecting j as soon as t_{ij} exceeds Δ. In order to minimize the probability of mistakes, Δ is 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 properties of the failure detector. Thus, we can *implement* $\mathcal{T}$ in a more effective manner, 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 user needs to enter its CS for the first time, and higher message complexity inherited from using total order broadcast.

References

[1] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1 – 20, February 1991.

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, March 1996.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[4] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, April 2001.

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

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(3):374–382, April 1985.

[7] E. Gafni and M. Mitzenmacher. Analysis of timing-based mutual exclusion with random times. *SIAM Journal on Computing*, 31(3):816–837, 2001.

[8] V. Hadzilacos. A note on group mutual exclusion. In *20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 2001.

[9] Y.-J. Joung. Asynchronous group mutual exclusion. In *17th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 51–60, June 1998.

[10] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685, July 2001.

[11] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[12] L. Lamport. The mutual exclusion problem. Parts I&II. *Journal of the ACM*, 33(2):313–348, April 1986.

[13] S. Lodha and A. D. Kshemkalyan. A fair distributed mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):537–549, June 2000.

[14] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[15] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[16] D. Manivannan and 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*, pages 525–530, October 1994.

[17] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, Massachusetts, 1986.

[18] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[19] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, May 1993.