4-17-2008

The Weakest Failure Detector to Solve Mutual Exclusion

Vibhor Bhatt
Dartmouth College

Nicholas Christman
Dartmouth College

Prasad Jayanti
Dartmouth College

The Weakest Failure Detector to Solve Mutual Exclusion

Vibhor Bhatt Nicholas Christman Prasad Jayanti

Dartmouth College, Hanover, NH
Dartmouth Computer Science Technical Report
TR2008-618

April 17, 2008

Abstract

Mutual exclusion is not solvable in an asynchronous message-passing system where processes are subject to crash failures. Delporte-Gallet et. al. determined the weakest failure detector to solve this problem when a majority of processes are correct. Here we identify the weakest failure detector to solve mutual exclusion in any environment, i.e., regardless of the number of faulty processes.

We also show a relation between mutual exclusion and consensus, arguably the two most fundamental problems in distributed computing. Specifically, we show that a failure detector that solves mutual exclusion is sufficient to solve non-uniform consensus but not necessarily uniform consensus.

1 Introduction

This paper addresses the mutual exclusion problem in an asynchronous distributed message-passing system where communication channels are reliable and processes can fail by crashing. The mutual exclusion problem involves managing access to a single, indivisible resource that can only support one user at a time. We say a user is in its *critical section* when it has access to the resource. The mutual exclusion problem includes a *progress condition*: if a correct process wants to enter its critical section, then eventually some correct process will gain access to its critical section (as long as no correct process remains in its critical section forever). That is, a crashed process cannot block correct processes from their critical sections.

Chandy and Misra [3] give a solution for the drinking philosophers problem—a generalization of the mutual exclusion problem—in a system without failures. Mutual exclusion cannot be solved deterministically in an asynchronous system without any information about failures because there is no way to determine whether a process in its critical section is crashed or just slow. This issue is related to the famous impossibility result from [7] that consensus is cannot be solved deterministically in a system subject to even a single crash failure.

Chandra and Toueg [2] introduced *failure detectors* to circumvent the impossibility of consensus. Informally, a failure detector is a distributed oracle that provides (possibly incorrect) information about which processes have failed. Each process has access to a *local failure detector module*

that monitors the other processes in the system. [2] shows that a relatively weak failure detector $\Diamond W$ is sufficient to solve consensus when a majority of processes are correct, and [1] shows that $\Diamond W$ is also necessary to solve consensus. Taking these two results together, $\Diamond W$ is the *weakest failure detector* to solve consensus when a majority of processes are correct.

Here we address the question: what is the weakest failure detector to solve mutual exclusion? In [5], Delporte-Gallet et. al. show that the *trusting failure detector* $\mathcal{T}$ is the weakest failure detector to solve mutual exclusion when a majority of processes are correct. They go on to show that ($\mathcal{T}$, S) is sufficient to solve mutual exclusion when there is no restriction on the number of failures, where S is the strong failure detector from the Chandra-Toueg hierarchy. But is ($\mathcal{T}$, S) necessary to solve mutual exclusion? The answer is no. In this paper, we show that ($\mathcal{T}$, Σ^s) is both sufficient and necessary to solve the problem, where Σ^s is a new failure detector that we call the *safe intersection quorum failure detector*. Furthermore, ($\mathcal{T}$, Σ^s) is weaker than ($\mathcal{T}$, S).

The safe intersection quorum failure detector upholds two properties: (1) eventually all local modules contain only correct processes and (2) if a process's local module outputs a quorum that does not intersect with a second process's quorum output at a later time, then the first process crashes before that later time. Intuitively, Σ^s provides overlapping quorums for all live processes. Σ^s is an adaptation of the quorum failure detector Σ originally introduced in [4] to solve uniform consensus when there is no restriction on the number of failures. It is also similar to the non-uniform quorum failure detector Σ^v introduced in [6] to solve nonuniform consensus when there is no restriction on the number of failures. However, Σ^s is strictly weaker than Σ and strictly stronger than Σ^v.

In the sufficiency algorithm in [5], Delporte-Gallet et. al. employ a "bakery style" algorithm: a process that wishes to go into the critical section draws a ticket and is granted access to the critical section in the order of its ticket number. A process will not be served until some other process trusts that process; that way, if a process crashes in its critical section, its failure will eventually be detected. Delporte-Gallet et. al. use an atomic broadcast primitive to totally order tickets. However, atomic broadcast cannot be implemented with only $\mathcal{T}$ in a system with no restrictions on the number of failures. In our algorithm, we adapt the bakery style algorithm to work an environment where we cannot totally order broadcasts. Each process has a token, representing its permission for another process to enter the critical section and a priority, which can be thought of as its ticket. To enter the critical section, a process must collect tokens from all the processes in its local Σ^s module. Processes give their tokens to the process that they trust with the highest ticket number. Since the quorums of live processes always intersect, no two live processes enter their critical sections at the same time.

In addition, we show that if any number of processes may crash, no algorithm can solve mutual exclusion with a strictly weaker failure detector. Given an algorithm that solves mutual exclusion, Delporte-Gallet et. al. showed that it is possible to extract the information provided by $\mathcal{T}$. We show that it is also possible to extract Σ^s. Intuitively, the set of processes that give any process permission to enter its critical section must intersect with the set that gives another process permission to enter the critical section. Otherwise, it is possible that the two sets give permission simultaneously, resulting in a violation of mutual exclusion. Therefore, ($\mathcal{T}$, Σ^s) is the weakest failure detector to solve mutual exclusion.

Finally, we conclude with a look at the relative strengths of the weakest failure detectors for mutual exclusion and consensus. We show that the weakest failure detector to solve mutual exclusion is

strictly stronger than the weakest failure detector to solve nonuniform consensus but incomparable to the weakest failure detector to solve uniform consensus. In appendix, we formalize a new idea of comparing problems in fault-prone systems independent of failure detectors, and we show that it is different from the traditional way of comparing problems based on their weakest failure detectors.

This paper is organized as follows. Section 2 describes the model of the system. Section 3 introduces failure detectors. Section 4 formally defines the mutual exclusion problem. Section 5 and 6 show that $(\mathcal{T}, \Sigma^s)$ is sufficient and necessary to solve the problem, respectively. Section 7 concludes with a comparison of mutual exclusion and consensus. Appendix A formalizes two different modes of problem comparison in crash-prone distributed systems. Appendix B shows that mutual exclusion is equivalent (in a sense that is defined Appendix A) to mutual exclusion with starvation freedom, an additional fairness property. In [10], Pike et. al. showed that that $\Diamond P$ is necessary to solve eventual mutual exclusion; Appendix C provides a more general proof of that result. Finally, Appendix D provides the proofs for two propositions that compare failure detectors.

2 Model

In this paper, we consider asynchronous message-passing distributed systems with no timing assumptions (the same model used in [1]). In particular, we make no assumptions on the time it takes to deliver a message or on relative process speeds. The system consists of a set of n processes $\Pi = \{1, 2, ..., n\}$ $(n > 1)$ that are completely connected with bidirectional, point-to-point, reliable links. We assume the existence of a discrete global clock—this is a fictional device to simplify the presentation; processes do not have access to it. We take the range of the clock's ticks to be the set of natural numbers $\mathbb{N}$.

2.1 Failures, failure patterns, and environments

Processes fail by crashing, i.e., by halting prematurely. A *failure pattern* is a function $F: \mathbb{N} \to 2^{\Pi}$ where $F(t)$ is the set of processes that have crashed through time t. A crashed process never recovers, i.e., $\forall t: F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathbb{N}} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. We say that a process i is *correct* if $i \in \mathit{correct}(F)$, and i *crashes* (or *is faulty*) if $i \in \mathit{crashed}(F)$.

An *environment* $\mathcal{E}$ is a set of failure patterns. Intuitively, an environment describes the number and timing of failures that can occur in the system. Thus, a result that applies to all environments is one that holds regardless of the number and timing of failures.

2.2 Failure detectors

Each process has access to a local failure detector module that provides (possibly incorrect) information about the failure pattern that occurs in an execution. A *failure detector history H with range R* is a function from $\Pi \times \mathbb{N}$ to R. $H(i, t)$ is the output value of the failure detector module of process i at time t. A *failure detector D* is a function that maps any failure pattern to a non-empty set of failure detector histories with range R_D (where R_D denotes the range of failure detector outputs of

D). $D(F)$ denotes the set of possible failure detector histories permitted by D for failure pattern F.

2.3 Algorithms, configurations, schedules, and runs

An algorithm A is a collection of n (possibly infinite-state) deterministic automata, one for each process in the system. We model the asynchronous communication channels as a message buffer containing all messages yet to be received at their destinations. Computation proceeds in atomic *steps* of A. In a step, a process may: receive a message from a process, query its failure detector, change states, and send messages to other processes. Formally, a step is a tuple $e = (i, m, d, A)$, where process i takes step e, m is the message (possibly the null message, denoted λ) received during e, d is the failure detector value seen by i in e, and A is the algorithm. Note that the message received in a step is chosen non-deterministically from the messages in the message buffer addressed to i, or the null message λ.

A *configuration* defines the current state of the each process in the system and the set of messages currently in the message buffer. A step (i, m, d, A) is *applicable* to a configuration C if and only if $m = \lambda$ or m is in the message buffer.

A *schedule* of an algorithm A is a finite or infinite sequence of steps of A. A schedule S is *applicable* to a configuration C if S is the empty schedule, or $S[1]$ is applicable to C, $S[2]$ is applicable to $S[1](C)$, etc., where $S[i]$ denotes the i^{th} step in S. A *run of the algorithm A using failure detector D in environment $\mathcal{E}$* is a tuple $R = (F, H, I, S, T)$ where F is a failure pattern in $\mathcal{E}$, H is a failure detector history in $D(F)$, I is an initial configuration of A, S is a schedule of A, and T is a list of increasing time values indicating when each step in S occurs such that (1) S is applicable to I, (2) for all $j \leq |S|$, if $S[j] = (i, m, d, A)$, then $i \notin F(T[j])$ and $d = H(i, T[j])$, (3) for all $j < k \leq |S|$: $T[j] \leq T[k]$, (4) every correct process takes an infinite number of steps in S, and (5) each message sent to a correct process is eventually received.

2.4 Solving problems with failure detectors

A *problem P* is defined by a set of properties that runs must satisfy. An algorithm A *uses failure detector D to solve a problem P in environment $\mathcal{E}$* if and only if all the runs of A using D in $\mathcal{E}$ satisfy the properties of P. We say that a failure detector D *can be used to solve problem P in environment $\mathcal{E}$* if there exists an algorithm A that uses D to solve P in $\mathcal{E}$.

2.5 The weakest failure detector

We must first explain how to compare two failure detectors D and D' in some environment. A transformation algorithm from D to D', denoted $T_{D \to D'}$, maintains a variable $output_i$ at each process i that emulates D'. Let O_R be the history of output values in R, i.e., $O_R(i, t)$ is the value of $output_i$ at time t. We say that $T_{D \to D'}$ *transforms D into D' in $\mathcal{E}$* if and only if for every $F \in \mathcal{E}$ and every run $R = (F, H, I, S, T)$ of $T_{D \to D'}$ using D, we have $O_R \in D'(F)$. If such an algorithm exists, we say that D' *is weaker than D* in $\mathcal{E}$, denoted $D \preceq_{\mathcal{E}} D'$. If $D \preceq_{\mathcal{E}} D'$ and $D' \not\preceq_{\mathcal{E}} D$, we say D *is strictly weaker than D'* in $\mathcal{E}$, denoted $D \prec_{\mathcal{E}} D'$.

A failure detector D^* is the *weakest failure detector to solve problem P in environment* $\mathcal{E}$ if and only if the following hold:

- **Sufficiency:** D^* can be used to solve P in $\mathcal{E}$.
- **Necessity:** For any failure detector D, if D can be used to solve P in $\mathcal{E}$, then $D^* \preceq_{\mathcal{E}} D$.

3 Failure detector ($\mathcal{T}$, Σ^s)

In this section, we recall the definition of the trusting failure detector $\mathcal{T}$ introduced in [5] and introduce our *safe intersection quorum failure detector*, denoted Σ^s. We also review some other failure detectors that give insight into the relative strength of $\mathcal{T}$ and of Σ^s.

3.1 Perfect, eventually perfect, and trusting failure detectors

3.1.1 Perfect and eventually perfect failure detectors

First, let us briefly recall the definition for the perfect failure detector $\mathcal{P}$ and the eventually perfect failure detector $\Diamond\mathcal{P}$ from [2]. $\mathcal{P}$ outputs a set of processes it *suspects* to have crashed such that the following properties hold:

Strong completeness: Eventually every faulty process is suspected by every correct process.

Strong accuracy: No process is suspected before it crashes.

$\Diamond\mathcal{P}$ is identical except the strong accuracy property is replaced with the following eventual strong accuracy property:

Eventual strong accuracy: Eventually no correct process is suspected.

3.1.2 Trusting failure detector

The *trusting failure detector* $\mathcal{T}$ outputs a set of *trusted* processes such that the following properties hold:

Trusting completeness: Eventually no faulty process is trusted by any correct process.

Trusting accuracy:

1. Every correct process is eventually trusted by every correct process.
2. Every process j that stops being trusted by a process i is faulty.

[5] showed that $\mathcal{T}$ is the weakest failure detector to solve mutual exclusion when a majority of processes are correct. In addition, [5] showed that in any environment with failures $\Diamond\mathcal{P} \prec \mathcal{T} \prec \mathcal{P}$.

3.2 Quorum failure detectors

Here we define the safe intersection quorum failure detector. Its motivation came from the quorum failure detector Σ and the non-uniform quorum failure detector Σ^v from [4] and [6] respectively.

3.2.1 Quorum failure detector and non-uniform quorum failure detector

The *quorum failure detector* Σ outputs a set of processes such that the following properties hold:

Completeness: Eventually, the quorums of correct processes contain only correct processes.

Intersection: Any two quorums intersect (at any times and at any processes).

The *non-uniform quorum failure detector* Σ^v is the non-uniform counterpart of Σ. It is identical to Σ except that instead of satisfying the intersection property, its outputs must satisfy:

Non-uniform intersection: Any two quorums output by correct processes intersect.

3.2.2 Safe intersection quorum failure detector

We now define the *safe-intersection quorum failure detector* denoted Σ^s. Σ^s outputs a set of processes, i.e., $R_{\Sigma^s} = 2^\Pi$. The idea is that if any two distinct processes i, j output non-intersecting quorums at times $t < t'$ respectively, then process i must crash before time t'. For a failure pattern F, $H \in \Sigma^s(F)$ if and only if H satisfies:

Completeness: Eventually, the quorums of correct processes contain only correct processes. Formally,

$$\exists t \in \mathbb{N}, \forall i \in \text{correct}(F), \forall t' > t\colon H(i, t') \subseteq \text{correct}(F)$$

Safe intersection: If process i outputs quorum Q_i at time t and process j outputs a quorum Q_j at time $t' > t$ such that Q_i and Q_j do not intersect, then i crashes before time t'. Formally,

$$\exists i, j \in \Pi, \exists t < t' \in \mathbb{N}\colon H(i, t) \cap H(j, t') = \phi \Rightarrow i \in F(t')$$

We prove the following propositions in Appendix D. The first compares the three quorum failure detectors. The second shows that $(\mathcal{T}, S)$ is strictly stronger than $(\mathcal{T}, \Sigma^s)$. This means that $(\mathcal{T}, S)$, which Delporte-Gallet et. al. [5] showed to be sufficient to solve mutual exclusion, is not the weakest failure detector to solve mutual exclusion.

Proposition 3.1 $\Sigma^v \prec \Sigma^s \prec \Sigma$.

Proposition 3.2 $(\mathcal{T}, \Sigma^s) \prec (\mathcal{T}, S)$

4 Mutual exclusion

Here we define the mutual exclusion problem in an asynchronous environment subject to crash failures. It is a natural problem involving the allocation of a single resource over a set of processes.

We have a critical section that we must allow all processes to access, but we can only have at most one process in the critical section at any given time. In formally defining the problem, we use the terminology and notations originally given by [8] and more recently used by [5].

Each process $i \in \Pi$ acts as an agent for exactly one user u_i. The process and user interact with four actions: try_i, $crit_i$, $exit_i$, and rem_i. The user u_i outputs try_i when it would like to enter its critical section and $exit_i$ when it would like to leave its critical section. These are the inputs to process i. The outputs of i—inputs to u_i—are $crit_i$, granting access to the critical section, and rem_i, indicating that u_i has left the critical section.

We define a sequence of try_i, $crit_i$, $exit_i$, rem_i actions to be *well-formed* for the pair (i, u_i) if it is a prefix of the cyclically ordered sequence $\langle try_i, crit_i, exit_i, rem_i \rangle$. We call u_i a *well-formed user* if u_i does not violate the cyclic order of the actions $try_i, crit_i, exit_i, rem_i$. Given an execution of (i, u_i) that observes the cyclic order, we say that u_i (equivalently i) is

- in its *remainder section* initially and in between any rem_i action and the following try_i
- in its *trying section* between any try_i action and the following $crit_i$
- in its *critical section* between any $crit_i$ action and either (a) the following $exit_i$ or (b) the time that i crashes
- in its *exit section* between any $exit_i$ action and the following rem_i

An algorithm for the mutual exclusion problem defines the trying and exiting protocols for every process $i \in \Pi$. We say that an algorithm A solves the mutual exclusion problem if, under the assumption that every user is a well-formed user, any run of A satisfies the following properties:

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

Mutual exclusion: No two distinct processes are in their critical sections at the same time.

Progress:

1. If a correct process tries to enter its critical section and no correct process remains in its critical section forever, then eventually some correct process enters its critical section.
2. If a correct process tries to exit its critical section, then it eventually enters its remainder section

Starvation freedom is an additional fairness property. In our paper, we will meet this condition in the algorithm we use to show sufficiency. Therefore, we will say that we solve starvation-free mutual exclusion.

Starvation freedom: If no correct process stays forever in its critical section, then every correct process that tries to enter its critical section eventually succeeds.

In appendix, we show that starvation-free mutual exclusion is equivalent (in a sense that we define formally) to mutual exclusion without the starvation freedom property. That is, given only an algorithm that solves mutual exclusion, we can solve starvation-free mutual exclusion.

5 Sufficiency

In this section, we prove that $(\mathcal{T}, \Sigma^s)$ is sufficient to solve starvation-free mutual exclusion in any environment by giving an algorithm that solves the problem. In addition, the algorithm we give is quiescent, i.e., if processes make a finite number of attempts to enter the critical section, then only finitely many messages are exchanged. In Section B.1, we give an overview of the intuition behind the algorithm. In Section B.2, we give the algorithm and prove that it is correct.

5.1 High-level description

Delporte et. al. solve mutual exclusion using $\mathcal{T}$ in an environment with a majority of correct processes. In [5], they use a "bakery style" algorithm in which processes that wish to enter their critical section take a ticket, wait until their number is called, and then enter the critical section. Their algorithm requires an atomic broadcast primitive, but atomic broadcast requires the failure detector (Ω, Σ) [1] in an environment with no guarantee on the number of correct processes. We later show in Section 7 that $(\mathcal{T}, \Sigma^s) \not\preceq (\Omega, \Sigma)$, so we cannot use atomic broadcast.

We present an algorithm which implements a bakery style approach to an environment where any number of processes may fail and thus we cannot totally order broadcasts. Each process maintains a local priority queue containing processes that are trying to enter their critical sections. In addition, each process has a *token*, which represents its permission for another process to enter its critical section. A process sends its token to the first process in its local queue that it trusts. Then the process waits for its token to be returned, reclaiming the token if the token recipient crashes. In order to enter the critical section, a process must collect tokens from all processes in its local Σ^s module. Therefore, the safe-intersection property of Σ^s guarantees that no other process will be able to collect tokens from all processes in its local Σ^s module, so no two processes enter their critical sections at the same time.

For the local queues, we define a process i's *priority* to be the pair (i, c_i), where c_i is the number of times i has already entered its critical section. We order priorities based on the following rule: for processes i, j, $(i, c_i) > (j, c_j)$ if and only if $c_i < c_j \vee (c_i = c_j \wedge i < j)$. We say that a process j is *first in* process i's queue if j has the greatest priority of all the processes that i trusts at a given time. When i wants to enter the critical section, i simply broadcasts (i.e., sends to all) its priority, and all processes that receive the message put i's priority in their local queues.

A process i sends its token to the process j first in its queue and then waits to receive its token back from j. j will enter its critical section if j collects tokens from all processes in its quorum Σ_j^s and j is first in its own queue. If j gains its critical section, j returns all the tokens with a *done* broadcast. In this case, i will remove j from its queue. However, if some other process k is inserted into the front of j's queue before j finishes collecting tokens, j will return all the tokens with a *deferred* message—this prevents j from holding onto some tokens that k needs to enter its critical section. Then i will eventually receive a broadcast from some process with higher priority than j

[1] Ω has the property that eventually all correct processes trust the same correct process. [6] showed (Ω, Σ) to be the weakest failure detector to solve consensus in any environment. [2] showed that consensus and atomic broadcast are equivalent, so (Ω, Σ) is the weakest failure detector to solve atomic broadcast in any environment. Note that it is possible to implement Σ from the null failure detector in an environment where a majority of processes are correct.

and send that process its token (notice that i keeps j in its queue). Finally, if j fails, by the trusting completeness property of $\mathcal{T}$, i will detect the failure and reclaim its token.

Each process i must collect tokens from a quorum Σ_i^s before entering its critical section. This means that, as long as i is correct, no other process j will be able to collect tokens from all the processes in its quorum Σ_j^s by the safe intersection property of Σ^s. Therefore, j will not enter its critical section. Finally, suppose a correct process i wants to enter its critical section. There are only finitely many priorities greater than i's, so eventually i will be first in every process's queue and subsequently gain access to its critical section. This ensures starvation freedom.

5.2 Formal algorithm and proof of correctness

We give the formal algorithm in Figure 1. Each process maintains three local variables:

- c_i: the number of times i has entered its critical section
- Q_i: a priority queue that contains processes in their trying sections. Priorities are determined by the following rule: $(i, c_i) > (j, c_j) \Leftrightarrow (c_i < c_j \vee (c_i = c_j \wedge i < j))$. Note that the operation *first*(Q_i) represents the *first in* property defined above, i.e., *first*$(Q_i) = (j, c_j)$ if (j, c_j) is the greatest priority in Q_i such that $j \in \mathcal{T}_i$.
- *tokens*$_i$: the set of tokens currently held by i
- *flag*$_i$: a flag that is true only when i is collecting tokens; it prevents i from giving up tokens while i is in the critical section

It remains to show that the algorithm from Figure 1 solves the starvation-free mutual exclusion problem, as defined in Section 4, in any environment. Through the following lemmata, we will show that, given that every user is well-formed, any run of the algorithm from Figure 1 satisfies the four requisite properties: well-formedness, mutual exclusion, progress, and starvation freedom.

Lemma 5.1 (Well-formedness) *For all $i \in \Pi$, the interaction between i and u_i is well-formed.*

Proof: From assumption, we know all users are well-formed. To complete the proof, we need only show that process i does not violate the cyclic order of actions $try_i, crit_i, exit_i, rem_i$. From the algorithm, it is clear that every $crit_i$ action is proceeded by a try_i action from u_i since i must broadcast its priority before it can enter its critical section. Any time u_i executes a $exit_i$ action, it is followed by a rem_i response from i in the next line of the algorithm. Thus, i does not violate the cyclic order. ∎

Lemma 5.2 (Mutual Exclusion) *No two distinct processes are in their critical sections at the same time.*

Proof: By way of contradiction, suppose distinct processes i and j are in their critical sections at the same time t. Let *tokens*$_i$ be the set of tokens that process i had upon entering the critical

Code executed at process i:

```
1  initialize
2      c_i ← 0
3      Q_i ← empty-queue
4      tokens_i ← φ
5      flag_i ← false
6      T_i ← φ, Σ_i^S ← Π

7  repeat forever
8      if Q_i is nonempty ∧ first(Q_i) = (j, c_j) for j ≠ i then
9          send(i, token) to j
10         wait for received(j, c_j, −) or j ∈ faulty_i
11     else if Q_i is nonempty ∧ first(Q_i) = (i, c_i) then
12         flag_i ← true
13         wait for Σ_i^S ⊆ tokens_i or first(Q_i) ≠ (i, c_i)
14         if Σ_i^S ⊆ tokens_i then
15             crit_i
16             wait for exit_i
17             c_i ← c_i + 1
18             send(i, c_i, done) to all
19             tokens_i ← φ
20             rem_i
21         flag_i ← false

22 repeat forever
23     if first(Q_i) ≠ (i, c_i)∧ ≠ flag then
24         send(i, c_i, deferred) to all j ∈ tokens_i
25         tokens_i ← φ

26 upon try_i do
27     send(i, c_i) to all

28 upon receive(j, c_j) do
29     insert (j, c_j) into Q_i

30 upon receive(j, token) do
31     tokens_i ← tokens_i ∪ {j}

32 upon receive(j, c_j, done) do
33     remove (j, c_j) from Q_i

34 function first(Q_i)
35     return (j, c_j) where (j, c_j) is the highest priority in Q_i such that j ∈ T_i
```

Figure 1: Quiescent algorithm to solve mutual exclusion using $(\mathcal{T}, \Sigma^s)$ in any environment

section, let $\Sigma_i^s \subseteq tokens_i$ be the value of i's safe intersection quorum failure detector upon entering its critical section, and let t_i be the first time that i saw the quorum Σ_i^s. Define $tokens_j$, Σ_j^s, and t_j analogously, and assume without loss of generality that $t_i < t_j$. Now there are two cases:

- **Case 1:** $\Sigma_i^s \cap \Sigma_j^s = \phi$. Then, by the safe intersection property of Σ^s, process i must have crashed before time t_j. Since t_j was the first time that j saw the quorum Σ_j^s, t_j is the earliest time that j could have entered its critical section with the set Σ_j^s of tokens. Since i has already crashed, j is the only process in its critical section. Thus, we have a contradiction.
- **Case 2:** $\Sigma_i^s \cap \Sigma_j^s \neq \phi$. Let k be a process in the intersection. Then k must have given its token to both i and j. Without loss of generality, assume that t_1 is the last time before t that k gives its token to i and $t_2 > t_1$ is the last time k gives its token to j before time t. Assume i has priority (i, c_i) and j has priority (j, c_j) in Q_k. Line 10 of the algorithm requires that, before sending its token to j, k waits until it receives a done message from i for (i, c_i), k stops trusting i, or k receives a deferred message from i for (i, c_i). Since k sends its token to j at time t_2, one of those three events must have occurred before time t_2. If k receives a done message from i for (i, c_i), then i has left its critical section before time t_2. Since i does not receive a token from k again until after time t, i cannot be the critical section with tokens from the same quorum Σ_i^s at time t—a contradiction. If k stops trusting i, from the trusting accuracy property of $\mathcal{T}$, we know that i has crashed before time t_2. Thus, we have a contradiction. Finally, suppose k receives a deferred message for (i, c_i) before time t_2. Then i returns the token without entering its critical section—again, since i does not receive a token from k again until after time t, we have a contradiction.

Therefore, mutual exclusion is guaranteed. ■

Lemma 5.3 (Progress) *There are two parts to the progress condition:*

1. *If a correct process i tries to enter its critical section and no correct process remains in its critical section forever, then eventually some correct process j enters its critical section*
2. *If a correct process i tries to exit its critical section, eventually it enters its remainder section.*

Proof: (1) follows from the starvation freedom condition we show below in Lemma 5.4. (2) is clear—when any process i receives an $exit_i$ action from its user u_i, it responds two lines later in line 20 with a rem_i action. ■

Finally, we claim that our algorithm satisfies starvation freedom. For the following lemma, we assume that no correct process remains in its critical section forever.

Lemma 5.4 (Starvation freedom) *If a correct process i tries to enter its critical section, then eventually it will succeed.*

Proof: Once a process j gains its critical section with priority (j, c_j), j never gains its critical section with another priority greater than or equal to (j, c_j). This is clear because j increments c_j

after exiting the critical section and c_j never decreases. We say (j, c_j) is a *used* priority. We will say that a process i has m^{th} *highest priority* if there are only $m-1$ unused priorities greater than (i, c_i) in the system. When a process i tries to enter its critical section, it broadcasts its priority (i, c_i). By the definition of priority comparison, there are only finitely many priorities greater than (i, c_i). Thus, for some $m \in \mathbb{N}$, i has m^{th} highest priority.

Using induction, we'll prove the following claim.

Claim 5.1 *For any $m \in \mathbb{N}$, if a correct process i has m^{th} highest priority and i enters its trying section, then i will eventually gain its critical section.*

Proof: In the base case, $m = 1$. Then i has the highest priority. When i enters its trying section, i broadcasts its priority (i, c_i). Eventually each correct process j will receive i's broadcast and trust i, say at time t. Then i will be first in Q_j until i gains its critical section. By way of contradiction, suppose i does not have some correct process j's token at time t and never receives a token from j at a later time. But the next time j begins the loop at line 7, j will clearly send its token to i. This means that j must wait forever at one of three lines: 10, 13, or 16. Since i is first in Q_j, j will not wait past time t at line 13. Finally, since no process remains in its critical section forever, j will not wait forever at line 16.

Then j must forever at line 10. Since i never receives a token from j, j must have sent its token to some other process $k \neq i$ who never returns j's token. Suppose k crashes. By the trusting completeness property of $\mathcal{T}$, j eventually detects k's failure, completes line 10, and then begins the loop at line 7—a contradiction. Now suppose k is correct, and k receives j's token at time $t' > t$. Since i is first in Q_k, if k ever sets $flag_k$ to false, the loop at line 22 will return j's token. Then k must wait forever at either line 13 or line 16 because these are the only wait statements at a point where $flag_k$ is true. But we have a contradiction—i is first in Q_k and k does not remain in its critical section forever, so k does not wait forever at line 13 or line 16. Thus, k returns its token to j. Then j does not wait forever at line 10, line 13, or line 16, so j eventually sends i its token. Since i is correct, eventually i receives tokens from all correct processes, and Σ_i^s contains only correct processes. Then, i enters its critical section.

Assume inductively that the claim holds for all natural numbers less than m. Now suppose i has m^{th} highest priority. Assume all faulty processes have crashed, all correct processes trust all other correct processes, and all quorums contain only correct processes. When i enters its trying section, i broadcasts its priority (i, c_i). Eventually every correct process receives i's broadcast and puts i in its queue at some time t. Let j be the process with highest priority (j, c_j) that tries to enter the critical section at any time after t. Suppose $j \neq i$. Then, by the induction hypothesis, j eventually enters the critical section since j has at least $(m-1)^{th}$ highest priority. But now (j, c_j) has been used, so i has the $(m-1)^{th}$ highest priority. Thus, applying the induction hypothesis a second time, i enters its critical section. Now suppose $j = i$. Then we can argue analogously to the base case that i eventually enters its critical section since i effectively has the highest priority. ∎

Therefore, we have that for any $m \in \mathbb{N}$, if a correct process i has m^{th} highest priority and i enters its trying section, i will eventually gain its critical section. Since for any priority (i, c_i), there finitely many greater priorities, we have starvation freedom. ∎

The following theorem comes directly from the lemmata proved in this section.

Theorem 1 *The algorithm in Figure 1 solves starvation-free mutual exclusion using ($\mathcal{T}$, Σ^s) in any environment.*

Finally, it is worth noting that we have given a quiescent algorithm.

Claim 5.2 (Quiescence) *The algorithm in Figure 1 is quiescent—if processes make finitely many attempts to enter their critical sections, then a finite number of messages are exchanged.*

Proof: Lets consider the maximum number of messages that may be exchanged for a process i to enter the critical section once. As we saw above in the proof for starvation freedom, when process i enters its trying section, i has m^{th} highest priority for some natural number m. i sends n messages to announce that i wants to enter the critical section. In the worst case, all n processes send i their tokens, but, i receives a message from some process j with higher priority that wants to enter the critical section before i has received enough tokens to enter its critical section. Then i sends n deferred messages to return all the tokens. This can happen at most $m - 1$ times. On the m^{th} occurrence, i will receive all the tokens it needs and enter its critical section, finally returning all the tokens with a done broadcast (another $2n$ messages). Therefore, at most $n + 2nm$ messages are exchanged. Since m and n are natural numbers, a finite number of messages are exchanged for any process to enter the critical section once. Therefore, if processes make finitely many attempts to enter their critical sections, then a finite number of messages are exchanged. ■

6 Necessity

In this section, we show that ($\mathcal{T}$, Σ^s) is necessary to solve mutual exclusion. Delporte et. al. have shown in [5] that $\mathcal{T}$ is necessary to solve mutual exclusion. In Section 6.1, we revisit their proof informally. In Section 6.2, we show that Σ^s is necessary to solve mutual exclusion.

6.1 $\mathcal{T}$ is necessary to solve mutual exclusion

As shown in [5], given an algorithm A that solves mutual exclusion, we can extract the trusting failure detector $\mathcal{T}$. This means that there is a transformation algorithm which uses only A to produce the output of $\mathcal{T}$. We informally present the ideas of this transformation below.

Each process i simulates n different runs $R_1, R_2, \ldots, R_n$ of A. Process i uses R_j to monitor process j. Initially only process j tries to enter its critical section in R_j. Since j is the only process that is trying to enter the critical section in R_j, eventually j succeeds. Once in its critical section, j broadcasts that it has reached its critical section and stays there forever. i does not trust j until i receives a message from j announcing that j has gained its critical section. Therefore, if j crashes before sending this message, then i never trusts j. Suppose i receives the message from j and begins trusting j. Then i tries to enter its critical section in R_j. By the mutual exclusion property, if i ever gains its critical section, then j has crashed, so i stops trusting j. If i never gains the critical section, then j is correct and i trusts j forever.

Clearly the algorithm described above meets the completeness and accuracy properties of $\mathcal{T}$. If j crashes before entering its critical section in R_j, then process i never trusts j. If j crashes after entering its critical in R_j, then i eventually gains its critical section in R_j by the progress condition of mutual exclusion, and i stops trusting j. Therefore we have completeness: eventually no faulty process is trusted by any correct process. If j is correct, j eventually enters the critical section of R_j and remains there forever. Therefore i never gains the critical section, so i trusts j. We have the first part of accuracy: every correct process is eventually trusted by every correct process. Finally, i stops trusting j only upon entering its critical section in R_j after j has entered its critical section. Since j tries to remain in its critical section forever, by mutual exclusion, we can be sure that j has crashed. This is the second part of accuracy: every process that stops being trusted by another process is faulty.

6.2 Σ^s is necessary to solve mutual exclusion

We now show that Σ^s is necessary to solve mutual exclusion. Suppose a failure detector D can be used to solve mutual exclusion. Then we'll show that there exists a transformation $T_{D\rightarrow\Sigma^s}$ that can be used to implement Σ^s. In Section 6.2.1, we provide a high-level description of our transformation along with some background needed to understand the transformation. In Section 6.2.2, we give the transformation and prove its correctness.

6.2.1 Background and high-level description

The technique we use in our transformation was introduced by Chandra et al. in [1] to show that any failure detector which solves non-uniform consensus can be transformed into a Ω. In this technique, processes maintain a local DAG (directed acyclic graph), which is a sampling of the failure detector values seen by processes in the system. The directions of the edges impose a partial order on the failure detector values seen by all processes. A DAG-based transformation has two components: 1) a *communication component*, in which processes exchange their local DAG to construct an increasing approximation of the infinite DAG (which contains every failure detector value seen by every process in a complete run) and 2) a *computation component*, in which processes use their local DAGs to periodically compute and output new failure detector values.

Process i performs the following sequence of steps to maintain its local DAG G_i in the communication component:

(1) receives a message, either a DAG G_j sent to i by process j, or the empty message.

(2) updates G_i by setting it equal to $G_i \cup G_j$ (if i received a DAG in the previous step).

(3) queries its local failure detector module D_i for the k^{th} time, gets a value d, adds the vertex $v = [i, d, k]$ to G_i, and creates an edge from every other vertex in G_i to the new vertex v.

(4) sends the updated DAG G_i to all processes.

Henceforth, we use $G_i(t)$ to denote the local graph G_i at process i at time t (created by the communication component of the transformation, as described above). For a vertex $v = [i, d, k]$,

let $\tau(v)$ be the time when process i took its k^{th} step. Note that this is the first time when vertex v appeared in local DAG of any process. We'll recall a few observations about the communication component of building DAGs (shown in [1]).

Proposition 6.1 *If $v_1 \rightarrow v_2$ is an edge in G_i for some process i, then $\tau(v_1) < \tau(v_2)$.*

The next proposition follows from the fact that correct processes take infinitely many steps in an infinite run.

Proposition 6.2 *If i and j are correct processes and v is a vertex in $G_i(t)$ for some time t, then there exists a time $t' \geq t$ such that $G_i(t')$ contains an edge from v to a vertex (j, d, k) for some values of d and k.*

Notice that a path of G_i can be used to simulate runs of A for failure pattern F and failure detector history H. Consider a path $P = ([p_1, d_1, k_1], [p_2, d_2, k_2], \ldots, [p_l, d_l, k_l])$. From Proposition 6.1, we know the following steps occurred sequentially: p_1 queried its failure detector and got the value d_1, p_2 queried its failure detector and got the value d_2, ..., p_l queried its failure detector and got the value d_l. Then it is possible to construct a schedule $S = ([p_1, d_1, m_1], [p_2, d_2, m_2], \ldots, [p_l, d_l, m_l])$ corresponding to a possible run of A in failure pattern F, where each m_i is either the empty message or a message present in the message buffer at $\tau([p_i, d_i, m_i])$. We say that S is *compatible* with P.

In the computation component of our reduction, each process i computes a output value $output_i$ such that the history of all such output values $output_i$ is a valid failure detector history of Σ^s. Below we define the notion of a *critically satisfied* path, which is essential for the reduction. Informally a path is critically satisfied if it is compatible with a schedule S such that, if process i tries to enter its critical section in S, then i eventually succeeds in entering its critical section in S. The formal definition of a critically satisfied path is given below.

Definition 6.1 *A path $P = ([p_1, d_1, k_1], [p_2, d_2, k_2], \ldots, [p_l, d_l, k_l])$ of G_i for a process i is called* critically satisfied *if there exists a a schedule $S = (p_1, d_1, m_1), (p_2, d_2, m_2), \ldots, (p_l, d_l, m_l)$ with the following properties :*

(1) process i is in the trying section at the beginning of S and all the other processes are in their remainder sections throughout S.

(2) For $1 \leq i \leq l$, message m_i is the oldest message sent to process p_i before the time $\tau([p_i, d_i, k_i])$.

(3) process i enters the critical section in l^{th} step of S, i.e. $i = p_l$ and i enters the critical section in the step (p_l, d_l, m_l) of S.

Now we are prepared to describe our transformation informally. Given a failure detector D and an algorithm A that solves mutual exclusion using D, we build a DAG based on the values of D in the manner described above (the communication component). Simultaneously, each process

i simulates a run R_i of the algorithm A based on the DAG G_i. In R_i, process i (and only process i) tries to enter its critical section. Therefore, as long as i is correct, i eventually succeeds by the progress condition of mutual exclusion. Upon gaining its critical section, i exits its critical section immediately and begins trying to enter its critical section again. In this manner, i continues cycling through its critical section forever in R_i. Hence in the computation component of the reduction, process i waits for a new vertex $v = [i, d, k]$ to be added by the communication component. Then, process i looks for a critically satisfied path starting from v. Then i updates $output_i$ equal to the set of processes that participated in that critically satisfied path. In the following section, we will show that completeness condition of Σ^s is met since we continually refresh the output with a set of processes still participating in the algorithm, and we will show that the mutual exclusion condition of A guarantees safe intersection.

6.2.2 $T_{D \to \Sigma^s}$ and proof of correctness

We give the transformation $T_{D \to \Sigma^s}$ formally in Figure 2. Throughout this section, let A be an algorithm that solves mutual exclusion. It remains to show that values of $output_i$ form a valid failure detector history for Σ^s, i.e., that they obey the completeness and safe intersection properties of Σ^s.

We can define the *limit DAG* for a correct process i, denoted G_i^∞, to be the limit of $G_i(t)$ as t approached infinity. In the following lemma, we show that G_i^∞ contains infinitely many critically satisfied paths, which helps us show completeness.

Lemma 6.1 *If i is a correct process and v is a vertex in $G_i(t)$ for any time t, then there exists a time $t' > t$ such that $G_p(t')$ contains a critically satisfied path starting from vertex v.*

Proof: Let $v = [i, d, k]$. By Proposition 6.2, for any correct process j, there exists a time t' such that $G_i(t')$ contains an edge from v to $(j, -, -)$. We can apply Proposition 6.1 repeatedly to obtain an infinite path P in G_i^∞ such that correct processes appear in round robin fashion as we traverse the path P. We can construct an infinite schedule S compatible to P such that (1) process i and only process i is in its trying section and (2) in each step (j, d, m), the message received is the oldest message in the message buffer addressed to j or λ if no such message exists. Note that S corresponds to a complete run of A, which solves mutual exclusion.

By the progress property of mutual exclusion, S must have a finite prefix S_0 in which i enters its critical section. This corresponds to a finite prefix P_0 of P. Since i is in its trying section at the beginning of S_0 but gains its critical section in the last step of S_0, P_0 is a critically satisfied path. ∎

Lemma 6.2 (Completeness) *If i is correct, then eventually* output_i *contains only correct processes.*

Proof: Let t be a time after which all faulty processes have crashed. Let $v = [i, d, k]$ be the earliest vertex in G_i such that $\tau(v) > t$ and i looks for a critically satisfied path starting from v (in line 17). Since i is correct, i eventually finds critically satisfied path P starting at vertex v by Lemma 6.1 above. By Proposition 6.1, all processes in P took steps after time t and thus are correct. In addition, since all faulty processes have crashed, clearly no faulty process can participate in any future critically satisfied path, so $output_i$ contains only correct processes at all times after t. ∎

Code executed at process i

1 **initialize**
2 $G_i \leftarrow \phi$
3 $count_i \leftarrow 0$
4 $output_i \leftarrow \Pi$

Simulation of A
5 **repeat forever**
6 try_{ii} (try to enter the critical section in simulation of A)
7 **wait for** $crit_{ii}$
8 $exit_{ii}$

Communication component
8 **repeat forever**
9 receive m from message buffer
10 **if** $m = (j, G_j)$ for some $j \in \Pi$ **then**
11 $G_i \leftarrow G_i \cup G_j$
12 $d \leftarrow D_i$
13 add $[i, d, count]$ to G_i and add edges from all the vertices in G_i to $[i, d, count]$
14 send(i, G_i) to all processes
15 $count_i \leftarrow count_i + 1$

Computation component
16 **repeat forever**
17 **wait for** G_i contains a critically satisfied path $P = ([p_1, d_1, k_1], [p_2, d_2, k_2], \ldots, [p_l, d_l, k_l])$, where $p_1 = i, d_1 = d, k_1 = count$
18 $D_i \leftarrow \{p_1, p_2, \ldots p_l\}$

Figure 2: Transformation algorithm $T_{D \rightarrow \Sigma^s}$

Lemma 6.3 (Safe-intersection) *For processes i, j, let $P^i = ([p_1^i, d_1^i, k_1^i], [p_2^i, d_2^i, k_2^i], \ldots, [p_l^i, d_l^i, k_l^i])$ and $P^j = ([p_1^j, d_1^j k_1^j], [p_2^j, d_2^j, k_2^j], \ldots, [p_m^j, d_m^j, k_m^j])$ be two critically satisfied paths in $G_i(t)$ and $G_j(t')$ respectively for some $t, t' \in \Pi$ such that $\{p_1^i, p_2^i, \ldots p_l^i\} \cap \{p_1^j, p_2^j, \ldots p_l^j\} = \phi$ and $\tau([p_l^i, d_l^i, k_l^i]) < \tau([p_m^j, d_m^j, k_l^m])$. Then process i crashes before time $\tau([p_m^j, d_m^j, k_m^j])$.*

Proof: Since P^i is a critically satisfied path, there exists a schedule S^i such that, (i) i tries to enter critical section in S^i and eventually succeeds in the last step of S^i, (ii) in the t^{th} step of S for $1 \leq t \leq l$, process p_t^i takes a step and gets d_i^t from its failure detector module and receives the oldest message m_t^i addressed to it. There exists an analogous schedule S^j for process j.

Let $output_i = \{p_1^i, p_2^i, \ldots, p_l^i\}$ and $output_j = \{p_1^j, p_2^j, \ldots, p_l^j\}$ be the set of processes taking steps in S^i and S^j respectively. We know that $output_i \cap output_j = \phi$. By way of contradiction, suppose i does not crash by $\tau([p_m^j, d_m^j, k_m^j])$. Then we'll create a run R of A which violates the mutual exclusion property.

In run R both schedules execute concurrently as follows:

(1) Both i and j immediately enter their trying sections in R; all the other processes are in their remainder section throughout R.

(2) All messages between the processes in $output_i$ and the processes $output_j$ are delayed until time $\tau([\,p_m^j, d_m^j, k_m^j\,])$.

(3) At time $\tau([\,p_t^i, d_t^i, k_t^i\,])$, the step $[\,p_t^i, d_t^i, m_t^i\,] \in S^i$ is executed for all $\forall t \in [1, l]$, and, at time $\tau([\,p_t^j, d_t^j, k_t^j\,])$, the step $[\,p_t^j, d_t^j, m_t^j\,] \in S^j$ is executed $\forall t \in [1, m]$.

(4) When i enters its critical section at time $\tau([\,p_l^i, d_l^i, k_l^i\,])$, i remains in its critical section until time $\tau([\,p_m^j, d_m^j, k_m^j\,])$.

By construction, at time $\tau([\,p_m^j, d_m^j, k_m^j\,])$, all processes in $output_i$ and $output_j$ are in the same state as in S^i and S^j respectively because they cannot distinguish R from the run in which only S^i or S^j occurs. Hence i and j enter their critical sections in R at times $\tau([\,p_l^i, d_l^i, k_l^i\,]) < \tau([\,p_m^j, d_m^j, k_l^m\,])$ respectively. Since i does not crash before $\tau([\,p_m^j, d_m^j, k_l^m\,])$, i and j are both in their critical sections at the same time, violating the mutual exclusion property. Therefore i crashes before $\tau([\,p_m^j, d_m^j, k_l^m\,])$. ∎

The previous lemmata give us the following theorem:

Theorem 2 *In any environment $\mathcal{E}$, if a failure detector D and an algorithm A can be used to solve mutual exclusion in $\mathcal{E}$, then $D \preceq_{\mathcal{E}} \Sigma^s$.*

Proof: In Figure 2, we give an transformation that uses D and A to implement a variable $output_i$ at each process i. We need only to show that the values of $output_i$ obey the two properties of Σ^s.

(i) **For each process i, $output_i$ obeys the safe-intersection property**:
Before i finds a critically satisfied path for the first time, $output_i = \Pi$, which trivially satisfies the safe intersection property. Process i only updates $output_i$ to the set of processes that took steps in its most recent critically satisfied path. Therefore, the safe intersection property is guaranteed by Lemma 6.3.

(ii) **For each correct process i, eventually $output_i$ contains only correct process** :
This is immediate from Lemma 6.2. ∎

Theorem 1 and Theorem 2 above give us the main result of our paper.

Theorem 3 *The failure detector $(\mathcal{T}, \Sigma^s)$ is the weakest failure detector to solve mutual exclusion in any environment.*

7 Mutual Exclusion and Consensus

In this section we establish a relation between mutual exclusion and consensus. We compare these problems through their weakest failure detectors. This gives us insight on the difficulty of solving one problem relative to the other.

The following lemma compares $(\mathcal{T}, \Sigma^s)$ with the weakest failure detector to solve non-uniform consensus, (Ω, Σ^v).

Lemma 7.1 $(\Omega, \Sigma^v) \prec (\mathcal{T}, \Sigma^s)$

Proof: We know $\Omega \equiv \Diamond W$ from [1], and $\Diamond W \prec \Diamond P$ from [2]. We know that $\Diamond P \prec \mathcal{T}$ from [5]. Thus, $\Omega \prec \mathcal{T}$, and $\Sigma^v \preceq \Sigma^s$ from Proposition 3.1, giving us $(\Omega, \Sigma^v) \preceq (\mathcal{T}, \Sigma^s)$. Finally, we'll show that $\Sigma^s \not\preceq (\Omega, \Sigma^v)$ to complete the proof.

By way of contradiction, suppose $\Sigma^s \preceq (\Omega, \Sigma^v)$ and let $T_{(\Omega, \Sigma^v) \rightarrow \Sigma^s}$ be a transformation algorithm that implements Σ^s using (Ω, Σ^v). We create three runs, R_1, R_2, and R_3, with $\Pi = \{a, b\}$. Throughout all three runs, the local failure detector modules give the following values: $\Omega_a = \Sigma^v_a = \{a\}, \Omega_b = \Sigma^v_b = \{b\}$. In R_1, a is correct and b crashes immediately. By the completeness property of Σ^s, $T_{(\Omega, \Sigma^v) \rightarrow \Sigma^s}$ must exclude b from its output at process a at some time t_1. In R_2, b is correct and a crashes immediately. By the completeness property of Σ^v, $T_{(\Omega, \Sigma^v) \rightarrow \Sigma^s}$ must exclude a from its output at process b at some time t_2.

In R_3, both a and b crash at time $t_3 > max(t_1, t_2)$. All messages sent between a and b are delayed until after both time t_3. Since a cannot distinguish R_3 from R_1, $T_{(\Omega, \Sigma^v) \rightarrow \Sigma^s}$ excludes b from its output at process a at t_1. Since b cannot distinguish R_3 from R_1, $T_{(\Omega, \Sigma^v) \rightarrow \Sigma^s}$ excludes a from its output at process b at t_2. Without loss of generality, assume $t_1 \leq t_2$. Then R_3 violates the safe intersection property since a and b output non-intersecting quorums at times t_1 and t_2 respectively but a does not crash until after time t_2—a contradiction. ∎

Corollary 4 *The weakest failure detector to solve consensus is strictly stronger than the weakest failure detector to solve non-uniform consensus.*

Lemma 7.2 $(\mathcal{T}, \Sigma^s)$ *is incomparable to* (Ω, Σ)

Proof: We begin by showing that $\mathcal{T} \not\preceq (\Omega, \Sigma)$. By way of contradiction, suppose $\mathcal{T} \preceq (\Omega, \Sigma)$ and let $T_{(\Omega, \Sigma) \rightarrow \mathcal{T}}$ be the transformation algorithm that implements $\mathcal{T}$ using (Ω, Σ). We create three runs, R_1, R_2, and R_3, with $\Pi = \{a, b\}$. Throughout all three runs the local failure detector modules give the following values: $\Omega_a = \Omega_b = \{a\}$, $\Sigma_a = \Sigma_b = \{a\}$. In R_1, both a and b are correct. By the trusting accuracy of $\mathcal{T}$, eventually $T_{(\Omega, \Sigma) \rightarrow \mathcal{T}}$ outputs $\{a, b\}$ at process a at some time t_1. R_2 is identical to R_1 until time t_1, at which point b crashes. By trusting accuracy of $\mathcal{T}$, there is a time $t_2 > t_1$ such that $T_{(\Omega, \Sigma) \rightarrow \mathcal{T}}$ outputs $\{a\}$ at process a. In R_3, both a and b are correct. R_3 is identical to R_1 until time t_1; from t_1 until t_2, a does not receive any messages from b. Unable to distinguish R_3 from R_2, $T_{(\Omega, \Sigma) \rightarrow \mathcal{T}}$ will output $\{a\}$ at process a at time t_2. This violates the trusting accuracy of $\mathcal{T}$—a contradiction.

Now we show that $\Sigma \not\preceq (\mathcal{T}, \Sigma^s)$. By way of contradiction, suppose $\Sigma \preceq (\mathcal{T}, \Sigma^s)$ and let $T_{(\mathcal{T}, \Sigma^s) \rightarrow \Sigma}$ be the transformation algorithm that implements Σ using $(\mathcal{T}, \Sigma^s)$. We construct two runs, R_1 and R_2, with $\Pi = \{a, b\}$. In R_1, a is correct, and b does not take any steps; in R_1, a's local failure detector modules give the following values throughout the run: $\mathcal{T}_a = \Sigma^s_a = \{a\}$. By the completeness property of Σ, $T_{(\mathcal{T}, \Sigma^s) \rightarrow \Sigma}$ must eventually exclude b from its output at process a at some time t. Now construct R_2 such that b is correct, but all messages sent by b are delayed until time t. a's local failure detector modules give the same values as in R_1; b's give the following: $\mathcal{T}_b =$

$\{b\}$, $\Sigma_b^s = \{a, b\}$. Then $T_{(\mathcal{T}, \Sigma^s) \to \Sigma}$ excludes b from its output at process a at time t since a cannot distinguish R_2 from R_1. However, in R_2, a crashes at time $t + 1$. By the completeness property of Σ, b must exclude a from its output at some later time. But this gives us a contradiction—b violates the intersection property of Σ. ■

From the previous two lemmas we state the following theorem.

Theorem 5 *For any environment $\mathcal{E}$, a failure detector D which solves mutual exclusion in $\mathcal{E}$ is sufficient to solve non-uniform consensus but not necessarily uniform consensus.*

This means that mutual exclusion is a strictly harder problem than non-uniform consensus but incomparable to uniform consensus.

References

[1] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Journal of the ACM, 43(4):685-722, July 1996.

[2] Tushar Deepak Chandra, and Sam Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM, 43(2):225-267, March 1996.

[3] K. Mandi Chandy and Jayadev Misra. The Drinking Philosophers Problem. ACM Transactions on Programming Languages and Systems 6(4): 632-646, October 1984.

[4] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui. Shared memory vs. message passing. Technical Report IC/2003/77, EPFL, 2003.

[5] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual Exclusion in Asynchronous Systems with Failure Detectors. Journal of Parallel and Distributed Computing (JPDC), 65(4), April 2005.

[6] Jonathan Eisler, Vassos Hadzilacos, Sam Toueg. The weakest failure detector to solve nonuniform consensus. Distributed Computing 19(4): 335-359, 2007.

[7] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM 8(9): 569, April 1985.

[8] Nancy Lynch. Distributed Algorithms. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[9] Scott Pike, Yantao Song, and Kaustav Ghoshal. Wait-Free Dining Under Eventual Weak Exclusion. Technical report: TAMU-CS-TR-2006-5-1, 2006.

[10] Yantao Song, Scott Pike, and Srikanth Sastry. The Weakest Failure Detector for Wait-Free, Eventually-Fair Mutual Exclusion. Technical report: TAMU-CS-TR-2007-2-2, 2007.

A Problem comparison

In order to prove that we have found the weakest failure detector D^* to solve a problem P, we must show that, given a failure detector D and an algorithm A that solves P using D, we can extract the failure detector D^*. Sometimes, we use both the algorithm A and the underlying failure detector D to extract D^*—for example, we use both to extract Σ^s in Section 6 from an algorithm that solves mutual exclusion using a given failure detector. Other times, we can extract D^* using only the algorithm A—for example, we extract $\Diamond P$ from only an algorithm that solves eventual mutual exclusion in Section C. Then, in some sense, we can say that the problems of eventual mutual exclusion and "implementing $\Diamond P$" are equivalent. An interesting question arises: are the problems A and "implementing D^*" always equivalent?

In this section, we introduce two notions of problem comparison, one based on failure detectors and algorithms and the other based only on algorithms. Then, we will show the relation between the two types of problem equivalence in Theorem 6. This relation suggests that the answer to the above question is no: implementing a problem's weakest failure detector can be harder than solving the problem itself.

Definition A.1 *For problems Q and Q', we say that Q* requires weaker failure detection *than Q' if, given a failure detector D and an algorithm A which solves Q' using D, we can solve Q. We denote Q* requires weaker failure detection *than Q' with $Q \preceq_{FD} Q'$. If $Q \preceq_{FD} Q'$ and $Q' \not\preceq_{FD} Q$, we say that Q* requires strictly weaker failure detection *than Q', denoted $Q \prec_{FD} Q'$.*

Definition A.2 *For problems Q and Q', we say that Q* is weaker *than Q' if, given an algorithm A that solves Q', we can solve Q. We denote Q* is weaker *than Q' with $Q \preceq_P Q'$. If $Q \preceq_P Q'$ and $Q' \not\preceq_P Q$, we say that Q* is strictly weaker *than Q', denoted $Q \prec_P Q'$.*

From the definitions, it is clear that if Q is weaker than Q', then Q requires weaker failure detection than Q'. However, we will show that the opposite is not true, i.e., if Q requires weaker failure detection than Q', it does not imply that Q is weaker than Q'. We prove this claim by giving two problems Q, Q' such that $Q \preceq_{FD} Q'$ but $Q \not\preceq_P Q'$. The problems are "implementing Ω" and uniform consensus respectively. We now define these two problems:

Implementing Ω: When queried, each process must output a process id such that eventually all correct processes output the same correct process's id permanently.

Uniform consensus: Each process proposes a value v. Eventually, each correct process must decide on a value such that the following properties hold:

- *Termination:* Eventually all processes decide on some value.
- *Agreement:* No two processes decide differently.
- *Validity:* If a process decides some value v, then some process proposed v.

Lemma A.1 *Implementing Ω requires weaker failure detection than uniform consensus.*

Proof: [1] showed that Ω is necessary to solve uniform consensus. Therefore, if there exists a failure detector D and an algorithm A that solves uniform consensus, then there exists a transformation $T_{D\rightarrow\Omega}$ that transforms D into Ω. Then, to solve the problem of implementing Ω, when queried, each process i should output the value of their local module of $T_{D\rightarrow\Omega}$, $output_i$. Since $T_{D\rightarrow\Omega}$ is a valid transformation that implements the failure detector Ω, eventually all correct processes will output the same correct process's id permanently. ■

We will show that, given only an algorithm A that solves uniform consensus, it is impossible to solve implementing Ω. Given this algorithm A that implements uniform consensus, all live processes propose a value and eventually A returns a decision value such that the properties of uniform consensus are satisfied.

Lemma A.2 *Given only an algorithm A that solves uniform consensus, implementing Ω is not solvable.*

Proof: By way of contradiction, suppose we have an algorithm A' that solves implementing Ω using only A. Then, in any run R_1 of A', there is some time t_1 after which all correct processes output the same correct process's id, say i, when queried. Now consider another run R_2 in which i fails at time t_1. Then at some time $t_2 > t_1$, all correct processes must output some other correct process's id, say $j \neq i$ when queried. Finally, consider a third run R_3 which is identical to R_1 until time t_1, at which point all messages sent by process i are delayed until time t_2 and i's proposal is not selected as a decision value in any run of A. Since the other processes cannot distinguish R_2 from R_3 without failure detection, all correct processes output j at time t_2 even though i is correct. Therefore, given any run R of A' where all the correct processes output process i's id permanently starting at time t, we can construct a run R' that is identical to R until time t in which the correct processes change their output at some time $t' > t$. We will use this technique to show that there is a run of A' in which all the correct processes never output the same correct process's id permanently.

Construct a run R^* in which all processes are correct. Suppose all the correct processes output the same correct process's id, say i at time t in R^*. Then, all messages from i are delayed until some time $t' > t$ and i's proposal is not selected in any run of A until after time t' such that at time t' all the correct processes output some other correct process's id, say $j \neq i$. If R^* exists, then clearly A' does not solve implementing Ω since the correct processes never output the same process's id permanently—anytime they output the same process's id, they are forced to change their output at some later time.

Thus, R^* must not exist. Then, there exists some time t at which all the correct processes output process i's id permanently. But as we showed above, correct processes cannot afford to wait forever for messages from i or i's proposals to be selected by A since they cannot distinguish R^* from another run R' in which i fails at time t. Thus, if all messages from i are delayed long enough, all correct processes must eventually output another correct process's id, $j \neq i$. Thus, we have a contradiction, there is no process i such that all correct processes output i's id permanently. Therefore, R^* exists, implying that A' does not solve implementing Ω. ■

Now we can state the main result of this section:

Theorem 6 *If Q requires weaker failure detection than Q', it does not imply that Q is weaker than Q'*

Proof: We prove this by example. From the two previous lemmata, we have that implementing Ω requires weaker failure detection than uniform consensus, but implementing Ω is not weaker than uniform consensus. ■

What does this theorem mean? Suppose we have a problem Q with a weakest failure detector D^*. Since D^* is the weakest failure detector for Q, clearly $Q \preceq_P$ implementing D^* because D^* is sufficient to solve Q. The above theorem shows that the converse is not always true, i.e., $\forall Q$, D^* implementing $D^* \not\preceq_P Q$. One such example is uniform consensus. In addition, suppose that another problem Q' is solvable given D^*, i.e., $Q' \preceq_{FD} Q$. The above theorem also says that Q' is not necessarily solvable given only an algorithm that solves Q; we might need the underlying failure detector as well. Intuitively, this means that the DAG building technique introduced in [1] is necessary to extract the weakest failure detectors from some problems like uniform consensus since no algorithm exists that can implement Ω given only an algorithm to solve uniform consensus.

B Starvation-free mutual exclusion $\preceq_P$ mutual exclusion

In this section, we show that, given an algorithm A that solves mutual exclusion, we can solve starvation-free mutual exclusion. Note that mutual exclusion only guarantees progress: if a correct process is trying to enter its critical section, eventually some correct process gains its critical section. When we showed $(\mathcal{T}, \Sigma^s)$ is necessary to solve mutual exclusion, we used both a failure detector D and an algorithm A that solves mutual exclusion using D. Since we used D to build a *DAG* in our transformation $T_{D \to \Sigma^s}$ in Figure 2 in Section 6, this problem is non-trivial. That is, we cannot simply extract $(\mathcal{T}, \Sigma^s)$ from A and use it to implement the algorithm for solving starvation-free mutual exclusion we gave in Figure 1 in Section 5. Instead, we will use A to implement a different algorithm A' that is similar to that given in Figure 1.

B.1 High-level description

Our algorithm consists of two phases. We call the first phase the *initialization phase* and the second phase the *main phase*. The purpose of the initialization phase is to ensure two conditions for the main phase: (1) the set of processes that reach the main phase is a superset of $correct(F)$ and (2) if a process i fails during the main phase, then all correct processes eventually detect i's failure. These conditions provide the failure detection necessary for a simple main phase based on a priority queue.

In the initialization phase, each pair of processes i, j shares a mutual exclusion primitive $mutex_{ij}$ (implemented by A). i (and only i) tries to enter the critical section of $mutex_{ij}$. Since i is the only correct process trying to enter the critical section of $mutex_{ij}$, eventually i gains $crit_{ij}$. At this point, i broadcasts (i.e., sends to all) an init-done message, remains in the critical section of $mutex_{ij}$ forever, and proceeds to the main phase. Clearly, if i is correct, i will reach the main phase. Upon receiving an init-done message from i, j tries to enter the critical section of $mutex_{ij}$. If i is

faulty, then eventually j will gain $crit_{ij}$. Otherwise j will never gain $crit_{ij}$. Thus, if i fails during the main phase, j is certain to detect i's failure as long as j is correct.

In the main phase, trying processes contend for a mutual exclusion primitive $mutex_{main}$. When i wants to enter its starvation-free critical section, i broadcasts its priority (i, c_i). Processes store priorities in a local priority queue, as in the sufficiency algorithm in Figure 1. If and only if i is at head of its own local queue, i tries to enter its critical section in $mutex_{main}$. If a process i gains $crit_{main}$, i enters its starvation-free critical section. Then i broadcasts a done message for $crit_{main}$ once it has exited its starvation-free critical section and processes remove (i, c_i) from their local queues. In addition, processes remove (i, c_i) from their queues upon learning that i has failed based on the monitoring performed during the initialization phase.

Since a process i only enters the starvation-free critical section if and only if it has gained $crit_{main}$, clearly the property of mutual exclusion is preserved. Starvation freedom follows in analogous fashion to the proof of starvation freedom for the algorithm in Figure 1. We argue that, if i is correct and in its trying section, there are only finitely many priorities greater than i's priority, so eventually i will gain its critical section.

B.2 Formal algorithm and proof of correctness

We give the formal algorithm in Figure 3. Each process maintains four local variables:

- $faulty_i$: the set of all processes that i has found to be faulty based the initialization phase monitoring
- c_i: the number of times i has entered its critical section
- Q_i: a priority queue that contains processes in their trying sections. Priorities are determined by the following rule: $(i, c_i) > (j, c_j) \Leftrightarrow c_i < (c_j \vee c_i = c_j \wedge i < j)$

It remains to prove that A' given in Figure 3 solves starvation-free mutual exclusion problem, as defined in Section 4, in any environment. Through the following Lemmata, we will show that, given that every user is well-formed, any run of the algorithm using failure detector $(\mathcal{T}, \Sigma^s)$ satisfies the four requisite properties: *well-formedness, mutual exclusion, progress,* and *starvation freedom.*

Lemma B.1 (Well-formedness) *For all $i \in \Pi$, the interaction between i and u_i is well-formed.*

Proof: From assumption, we know all users are well-formed. To complete the proof, we need only show that process i does not violate the cyclic order of actions $try_i, crit_i, exit_i, rem_i$. From the algorithm, it is clear that every $crit_i$ action is proceeded by a try_i action from u_i since i must broadcast its priority before i enters its critical section. Any time u_i executes a $exit_i$ action, it is followed by a rem_i response from i in the next line of the algorithm. Thus, i does not violate the cyclic order. ■

Lemma B.2 (Mutual Exclusion) *No two distinct processes are in their critical sections at the same time.*

Code executed at process i:

Initialization phase

```
1  initialize
2      faulty_i ← φ
3      c_i ← 0
4      Q_i ← empty-queue

5      try to enter the critical section of mutex_ij for all j ∈ Π
6      wait for crit_ij for all j ∈ Π
7      broadcast(i, init-done)
8      proceed to main phase

9  upon receive(j, init-done) do
10         try to enter the critical section of mutex_ji

11 upon crit_ji do
12     faulty_i ← faulty_i ∪ {j}
```

Main phase

```
13 repeat forever
14     if Q_i is nonempty ∧ head(Q_i) = (i, c_i) then
15         try to enter the critical section of mutex_main
16         wait for crit_main
17         crit_i
18         wait for exit_i
19         c_i ← c_i + 1
20         broadcast(i, c_i, done)
21         exit the critical section of mutex_main
22         wait for rem_main
23         rem_i

24 upon try_i do
25     broadcast(i, c_i, try)

26 upon receive(j, c_j, try) do
27     insert (j, c_j) into Q_i

28 upon j added to faulty_i ∨ receive(j, c_j, done) do
29     remove (j, c_j) from Q_i
```

Figure 3: Algorithm A' to solve starvation-free mutual exclusion in any environment using only mutual exclusion primitives

Proof: Suppose distinct processes i and j are in their critical sections at the same time. Then, both i and j are also in the critical section of $mutex_{main}$. Then we have a contradiction—A does not meet the condition of mutual exclusion. ■

Lemma B.3 (Progress) *There are two parts to the progress condition:*

1. *If a correct process i tries to enter its critical section and no correct process remains in its critical section forever, then eventually some correct process j enters its critical section*
2. *If a correct process i tries to exit its critical section, eventually it enters its remainder section.*

Proof: (1) follows from the starvation freedom condition we show below in B.4. (2) is clear—when any process i receives an $exit_i$ action from its user u_i, we respond with a rem_i action four lines later. The only wait statement in those four lines is at line 22, where i waits to enter its remainder section after exiting $crit_{main}$. But i does not wait forever at line 22 by the progress condition of A. ■

For the following lemma, assume no correct process stays in its critical section forever.

Lemma B.4 (Starvation freedom) *If a correct process i tries to enter its critical section, then eventually it will succeed.*

Proof: First, notice that all correct processes eventually reach the main phase. Suppose i is correct. Since i is the only process trying to enter the critical section of $mutex_{ij}$ (for each j), i eventually succeeds. Thus i enters the main phase. Recall the definition of m^{th} highest priority from the proof of Lemma 5.4. We'll prove the following claim using induction.

Claim B.1 *For any $m \in \mathbb{N}$, if a correct process i has m^{th} highest priority and i enters its trying section, then i will eventually gain its critical section.*

Proof: In the base case, $m = 1$. Then i has the highest priority. When i enters its trying section, i broadcasts its priority (i, c_i). Eventually every correct process j receives i's broadcast at some time t and i is at the head of Q_j. Suppose i has not yet gained the critical section at time t. Let S be the set of correct processes trying to enter their critical sections in $mutex_{main}$ at time t and suppose, in the worst case, each process $j \in S$ gains $crit_{main}$ while i is still waiting at line 16. Then j enters its critical section. Eventually j exits its critical section since no correct process remains in its critical section forever. Then j exits $crit_{main}$ and eventually gets rem_{main} in response by the progress condition of A. Next, j repeats the loop starting at line 13. But since i is at the head of Q_j, j does not try to enter its critical section in $mutex_{main}$ again until j receives a done message for (i, c_i). Therefore, there is some time $t' > t$ at which i is the only correct process trying to enter its critical section in $mutex_{main}$. Furthermore, since i is at the head of each correct process's queue, no other correct process will try to enter its critical section in $mutex_{main}$. Therefore, by the progress condition of A, i will eventually gain $crit_{main}$ and enter its starvation-free critical section.

Assume inductively that the claim holds for all natural numbers less than m. Now suppose i has m^{th} highest priority. When i enters its trying section, i broadcasts its priority (i, c_i) and tries to

enter the critical section of $mutex_{main}$. Eventually every correct process receives i's broadcast and puts i in its queue at some time t. Let j be the process with the highest priority (j, c_j) that enters its trying section after time t. Suppose $j \neq i$. Then j has at least $(m-1)^{th}$ highest priority, so j eventually enters its critical section by the induction hypothesis. But then (j, c_j) has been used, so i has at least $(m-1)^{th}$ highest priority. Applying the induction hypothesis a second time, we have that i eventually enters its critical section. Now suppose $i = j$. We can argue analogously to the base case above that i eventually enters its critical section since i effectively has the highest priority. ■

Therefore, we have that for any natural number m, if a correct process i has m^{th} highest priority and i enters its trying section, i will eventually gain its critical section. Since for any priority (i, c_i), there finitely many greater priorities, we have starvation freedom. ■

The following theorem comes directly from the lemmata proved in this section.

Theorem 7 *Starvation-free mutual exclusion $\preceq_P$ mutual exclusion.*

Proof: We have shown that, given an algorithm A that solves mutual exclusion, we can implement an algorithm A' shown above in Figure 3 that solves starvation-free mutual exclusion.

C $\Diamond P$ is necessary to solve eventual mutual exclusion

The eventual mutual exclusion problem is identical to mutual exclusion problem except that the condition of mutual exclusion is eventual: there is a time after which no two distinct processes are in their critical sections at the same time. In this section, we show that $\Diamond P$ is necessary to solve eventual mutual exclusion.

In [10], Pike et. al. showed this result but used a stronger version of the eventual mutual exclusion problem. They assumed an additional fairness condition called *eventual bounded waiting*: for every run, there exists a time t and a bound b such that, if a correct process i enters its trying section after time t, no process j enters its critical section more than b times before i enters its critical section. Clearly eventual bounded waiting is stronger than starvation freedom. Starvation freedom guarantees only that every correct process that tries to enter its critical section eventually succeeds; eventual bounded waiting guarantees that every correct process that tries to enter its critical section will succeed before any other process enters its critical section $b+1$ times.

In the proof that $\Diamond P$ is necessary for mutual exclusion, [10] uses the eventual bounded waiting property to implement an "elastic clock" which serves as the basis of a timeout system to monitor failures. Here we show that $\Diamond P$ is necessary to solve eventual mutual exclusion without the eventual bounded waiting condition and without the starvation-free fairness property—a more general result.

Suppose there exists an algorithm A that solves eventual mutual exclusion. We'll implement $\Diamond P$ using A. Each process i uses n eventual mutual exclusion primitives, $mutex_{1i}$, $mutex_{2i}$, ..., $mutex_{ni}$ (implemented by A), to monitor the other processes. Only i and j have access to the primitive $mutex_{ji}$. At the beginning of the algorithm, i suspects every other process j. Process j tries to enter

Code executed at process i:

Initialization

1 **initialize**
2 $output_i \leftarrow \Pi$
3 $M[1...n] = \{mutex_{1i}, mutex_{2i}, ..., mutex_{ni}\}$
4 try to enter critical section of $mutex_{ij}$ for all $j \in \Pi$

Tell others i is alive

5 **upon** $crit_{ij}$ **do**
6 send(i, *alive*) to j
7 **upon** receive(j, *suspect*) **do**
8 send(i, *alive*) to j

Monitor others

9 **upon** receive(j, *alive*) **do**
10 $output_i \leftarrow output_i - \{j\}$
11 try to enter critical section of $mutex_{ji}$
12 **upon** $crit_{ji}$ **do**
13 send(i, *suspect*) to j
14 exit $mutex_{ji}$
15 $output_i \leftarrow output_i \cup \{j\}$

Figure 4: Algorithm $T_{D \rightarrow \Diamond P}$ that implements $\Diamond P$ given an algorithm that solves eventual mutual exclusion

the critical section of $mutex_{ji}$. Since j is the only process trying to enter $mutex_{ji}$, if j is correct, eventually j will gain its critical section $crit_{ji}$. When j gains $crit_{ji}$, j sends an *alive* message to i. j will remain in the critical section forever. Whenever process i receives an alive message from j, i stops suspecting j. However, i tries to enter the critical section of $mutex_{ji}$. If i succeeds in entering the critical section, then i suspects j, sends j a *suspect* message, and exits the critical section. Each time j finds out that i suspects j (by receiving a suspect message from i), j sends another alive message to i.

Recall that $\Diamond P$ satisfies strong completeness and eventual strong accuracy. Suppose j is faulty. Then eventually j crashes. Therefore j sent finitely many alive messages. After i receives the last alive message from j, i will try to enter the critical section of $crit_{ji}$. Since i is the only correct process trying to enter the critical section of $crit_{ji}$, i eventually succeeds and suspects j permanently. Thus, we have strong completeness. Now suppose j is correct. Then j enters the critical section of $crit_{ji}$ and stays there permanently. By the property of eventual mutual exclusion, eventually i is no longer able to gain access to the critical section of $mutex_{ji}$. Thus, i sends j finitely many suspect messages. After receiving the last suspect message, j replies with an alive message. If i is correct, i eventually receives that alive message and never suspects j again. Thus, we have eventual strong accuracy.

We give the formal algorithm $T_{D \rightarrow \Diamond P}$ that transforms D into $\Diamond P$ in Figure 4. Each process i's local module of $\Diamond P$ is represented by the variable $output_i$.

Lemma C.1 (Strong Completeness) *For each process i, eventually* output$_i$ *permanently suspects all faulty processes.*

Proof: Suppose j crashes at time t_1. First suppose that i never receives an alive message from j. From the algorithm, it is clear that i only removes j from $output_i$ after receiving an alive message from j. Then i never removes j from $output_i$, so i permanently suspects j from the beginning. Now suppose i receives alive messages from j. Since j crashes at time t, i receives finitely many alive messages from j. Suppose i receives the last alive message from j at time t_2. Then i removes j from $output_i$ and tries to enter the critical section of $mutex_{ji}$. After time t_1, i is the only correct process trying to enter the critical section of $mutex_{ji}$, so i will eventually succeed at some time $t_3 > t_2$. Then i will add j to $output_i$ at time t_3. Since i never receives another alive message from j, i permanently suspects j after time t_3. ■

Lemma C.2 (Eventual Strong Accuracy) *For each process i, if i is correct, eventually every other correct process j permanently does not suspect i.*

Proof: If i is correct, then i eventually gains the critical section of $mutex_{ij}$ at some time t_1 since i is the only process trying to enter the critical section of $mutex_{ij}$. i sends an alive message to j, and j eventually receives this alive message at time t_2 since j is correct. Then j removes i from $output_j$. i stays in the critical section of $mutex_{ij}$ forever. If j adds i to $output_j$, j must have gained the critical section of $mutex_{ij}$, so j sends i a suspect message. By the eventual mutual exclusion property of $mutex_{ij}$, j only gains the critical section of $mutex_{ij}$ finitely many times. Then j only removes i from $output_j$ finitely many times, and j only sends finitely many suspect messages to i. Suppose i receives the last suspect message at time t_3. Then i responds with an alive message. Since j is correct, j receives the alive message at time $t_4 > t_3$. Then j removes i from $output_j$ at time t_4 and never adds i to $output_j$ again. ■

We can state the theorem directly from the previous two lemmata.

Theorem 8 *$\Diamond P$ is necessary to solve eventual mutual exclusion.*

Proof: Given a failure detector D and an algorithm A that solves eventual mutual exclusion using D, we have a transformation $T_{D \to \Diamond P}$. Thus $\Diamond P \preceq D$, implying that $\Diamond P$ is necessary to solve eventual mutual exclusion. ■

In [9], Pike et. al. showed that $\Diamond P$ is sufficient to solve eventual mutual exclusion. Combining that result with Theorem 8 above, we have that $\Diamond P$ is the weakest failure detector to solve eventual mutual exclusion. To complete our treatment of eventual mutual exclusion, we show that eventual mutual exclusion is equivalent $\preceq_P$ mutual exclusion. The transformation algorithm $T_{D \to \Diamond P}$ given in Figure 4 in Section C relies only on the algorithm A used to solve eventual mutual exclusion (and not on the failure detector D the algorithm uses). Therefore, given an algorithm A that solves eventual mutual exclusion, we can implement $\Diamond P$ using $T_{D \to \Diamond P}$. In [9], Pike et. al. give an algorithm to solve starvation-free eventual mutual exclusion using $\Diamond P$. Combining these two facts, we have that starvation-free eventual mutual exclusion $\preceq_P$ eventual mutual exclusion.

D Failure detector comparisons

Here we provide proofs for the assertions we made about the relative strengths of failure detectors in Section 3.

Claim D.1 *In an environment where any number of processes may fail,* $\Sigma^{\upsilon} \prec \Sigma^{s} \prec \Sigma$.

Proof: From the definitions, it is clear that $\Sigma^{\upsilon} \preceq \Sigma^{s} \preceq \Sigma$. We complete the proof by showing that $\Sigma^{s} \not\preceq \Sigma^{\upsilon}$ and that $\Sigma \not\preceq \Sigma^{s}$. For both arguments, we consider a system of two processes, i.e., $\Pi = \{a, b\}$.

By way of contradiction, assume that $\Sigma^{s} \preceq \Sigma^{\upsilon}$ and let $T_{\Sigma^{\upsilon} \rightarrow \Sigma^{s}}$ be the transformation algorithm that implements Σ^{s} using Σ^{υ}. We create three runs, R_1, R_2, and R_3, of $T_{\Sigma^{\upsilon} \rightarrow \Sigma^{s}}$ to reach a contradiction. Throughout all three runs, the local failure detector modules give the following values: $\Sigma^{\upsilon}_{a} = \{a\}$, $\Sigma^{\upsilon}_{b} = \{b\}$. In R_1, a is correct and b crashes immediately. By the completeness property of Σ^{s}, $T_{\Sigma^{\upsilon} \rightarrow \Sigma^{s}}$ must exclude b from its output at process a at some time t_1. In R_2, b is correct and a crashes immediately. Then, by the completeness property of Σ^{s}, $T_{\Sigma^{\upsilon} \rightarrow \Sigma^{s}}$ must exclude a from its output at process b at some time t_2. Finally, in R_3, all messages between a and b are delayed until time $t_3 = max(t_1, t_2)$ and b crashes after time t_3. Since a cannot distinguish R_3 from R_1, a excludes b from its output in R_3 at time t_1. Since b cannot distinguish R_3 from R_2, b excludes a from its output in R_3 at time t_2. Thus, a and b output non-intersecting quorums. Since neither a nor b has crashed at t_3, R_3 violates the safe intersection property of Σ^{s}. Thus, we have a contradiction—$\Sigma^{s} \not\preceq \Sigma^{\upsilon}$.

Now, by way of contradiction, assume that $\Sigma \preceq \Sigma^{s}$ and let $T_{\Sigma^{s} \rightarrow \Sigma}$ be the transformation algorithm that implements Σ using Σ^{s}. We create two runs, R_1 and R_2, of $T_{\Sigma^{\upsilon} \rightarrow \Sigma^{s}}$ to reach a contradiction. In R_1, the local failure detector modules give the following values throughout the run: $\Sigma^{s}_{a} = \{a\}$, $\Sigma^{s}_{b} = \{a, b\}$. In R_1, a is correct and b crashes immediately. By the completeness property of Σ, $T_{\Sigma^{s} \rightarrow \Sigma}$ must exclude b from its output at process a at some time t. In R_2, b is correct, a crashes at time $t + 1$, the local failure detector modules give the same values until time $t + 1$, and all messages sent between a and b are delayed until time $t + 1$. Since a cannot distinguish R_2 from R_1, a excludes b from its output at time t. At time $t + 1$, a crashes, and $\Sigma^{s}_{b} = \{b\}$ for the rest of R_2. By the completeness property of Σ, eventually $T_{\Sigma^{s} \rightarrow \Sigma}$ must exclude a from its output at process b. But then a and b output non-intersecting quorums, so R_2 violates the intersection property of Σ. Thus, we have a contradiction—$\Sigma \not\preceq \Sigma^{s}$. ∎

Claim D.2 *In an environment where any number of processes may fail,* $(\mathcal{T}, \Sigma^{s}) \prec (\mathcal{T}, S)$.

Proof: First, notice that $(\mathcal{T}, \Sigma^{s}) \preceq (\mathcal{T}, S)$. Clearly $\mathcal{T} \preceq (\mathcal{T}, S)$. The weak accuracy property of S states that some correct process is never suspected. Then, we can implement Σ^{s} from S at process i as follows: i queries S_i and outputs $\Pi - S_i$. Since some correct process is never suspected, that correct process is contained in all quorums, so the safe intersection property of Σ^{s} is satisfied. By the strong completeness property of S, eventually every crashed process is permanently suspected. Thus, all quorums contain only correct processes, so the completeness property of Σ^{s} is also satisfied. Therefore, $\Sigma^{s} \preceq (\mathcal{T}, S)$.

We complete the proof by showing that $S \not\preceq (\mathcal{T}, \Sigma^s)$, implying that $(\mathcal{T}, S) \not\preceq (\mathcal{T}, \Sigma^s)$. By way of contradiction, suppose $S \preceq (\mathcal{T}, \Sigma^s)$ and let $T_{(\mathcal{T}, \Sigma^s) \to S}$ be the transformation algorithm that implements S using $(\mathcal{T}, \Sigma^s)$. We create two runs, R_1 and R_2, of $T_{(\mathcal{T}, \Sigma^s) \to S}$ to reach a contradiction. In R_1, the local failure detector modules give the following values throughout the run: $\mathcal{T}_a = \{a\}$, $\mathcal{T}_b = \{b\}$, $\Sigma^s_a = \{a\}$, $\Sigma^s_b = \{a, b\}$. In R_1, a is correct, and b crashes immediately. By the completeness property of S, $T_{(\mathcal{T}, \Sigma^s) \to S}$ eventually outputs $\{b\}$ at process a at some time t. In R_2, b is correct, a crashes at time $t + 1$, the local failure detector modules give the same values until time $t + 1$, and all messages sent between a and b are delayed until time $t + 1$. Since a cannot distinguish R_2 from R_1, a outputs $\{b\}$ at time t. At time $t + 1$, a crashes, and Σ^s_b outputs $\{b\}$ for the rest of R_2. By the completeness property of S, eventually $T_{(\mathcal{T}, \Sigma^s) \to S}$ must output $\{a\}$ at process b. But R_2 violates the weak accuracy property of S since b is the only correct process and a suspected b at time t. Thus, we have a contradiction—$S \not\preceq (\mathcal{T}, \Sigma^s)$. ■