# Failure Detectors Encapsulate Fairness*

Scott M. Pike, Srikanth Sastry, and Jennifer L. Welch

Dept. of Computer Science and Engineering
Texas A&M University
College Station TX-77843 USA
{pike, sastry, welch}@cse.tamu.edu

**Abstract.** Failure detectors are commonly viewed as abstractions for the synchronism present in distributed system models. However, investigations into the exact amount of synchronism encapsulated by a given failure detector have met with limited success. The reason for this is that traditionally, models of partial synchrony are specified with respect to real time, but failure detectors do not encapsulate real time. Instead, we argue that failure detectors encapsulate the *fairness* in computation and communication. Fairness is a measure of the number of steps executed by one process relative either to the number of steps taken by another process or relative to the duration for which a message is in transit. We argue that oracles are substitutable for the fairness properties (rather than real-time properties) of partially synchronous systems. We propose four fairness-based models of partial synchrony and demonstrate that they are, in fact, the 'weakest systems models' to implement the canonical failure detectors from the Chandra-Toueg hierarchy.

## 1 Introduction

The inability to distinguish a crashed process from a slow process makes it impossible to solve several classic problems in distributed computing in crash-prone asynchronous systems [9]. Efforts to circumvent this impossibility have spawned two complementary approaches. The first approach, called *partial synchrony* [8, 7], focuses on assuming explicit temporal guarantees on computation and communication to enable crash detection. The second approach focuses on augmenting asynchronous systems with oracles, called *failure detectors* [3], that provide potentially incorrect information about process crashes in the system.

It has long been held that failure detectors *encapsulate* partial synchrony. More precisely, a failure detector $\mathcal{D}$ *encapsulates* a partially-synchronous system model $M$ if and only if the following two conditions hold: (1) $\mathcal{D}$ can be implemented in $M$, and (2) every problem $P$ that is solvable in system model $M$ is also solvable in an asynchronous system augmented with $\mathcal{D}$. Alternatively (and more informally), the notion of *encapsulation* by a failure detector may be viewed synonymously with the notion of *mutual reducibility*; that is, a failure detector

* This work was supported in part by NSF grant 0964696 and Texas Higher Education Coordinating Board grant NHARP 000512-0130-2007

$\mathcal{D}$ encapsulates a system model $M$ if and only if $M$ can implement $\mathcal{D}$, and $\mathcal{D}$ can implement $M$. As such, if $\mathcal{D}$ encapsulates $M$, then $\mathcal{D}$ is substitutable for $M$ because any problem solvable in $M$ is also solvable in asynchrony augmented with $\mathcal{D}$.

**Partial Synchrony.** A system model is partially synchronous [8] if it provides temporal bounds on computational and/or communicational quantities such as message delays and process speeds. The knowledge of such bounds may be incomplete or unknown. Despite such uncertainty, partial synchrony is useful for solving problems in crash-prone distributed systems, and several such models have been proposed in the literature(*e.g.*, [8, 7, 12, 21, 22, 11, 20]). These models vary in the information they provide about these bounds, and consequently they have different crash detection capabilities. One way to formalize this notion of crash detection capability is with failure detectors.

**Failure Detectors.** Informally, a failure detector [3] can be viewed as a system service (or oracle) that can be queried for (potentially unreliable) information about process crashes. The unreliable outputs of such oracles can be false positives (suspecting live processes) or false negatives (not suspecting crashed processes). From an empirical standpoint, most fault-tolerant problems in distributed computing that are otherwise unsolvable in crash-prone asynchronous systems can be solved by either (1) assuming adequate degrees of partial synchrony [8], or (2) augmenting asynchronous systems with sufficiently powerful oracles [15]. This observation suggests that the axiomatic properties of oracles might *encapsulate* the temporal properties of (suitably defined) models of partial synchrony. Accordingly, this conjecture has led to the pursuit of 'weakest system models' to implement various classes of oracles.

Current work on the weakest system models for oracles (see Sect. 2) has met with limited success partly because the proposed system models assume real-time bounds on communication (and possibly computation too). Unfortunately, failure detectors do not preserve such real-time bounds. To find such weakest system models, we need to address a more fundamental question: what precisely about partial synchrony do failure detectors preserve?

**Results.** We answer the foregoing question by demonstrating that failure detectors (at least when restricted to the Chandra-Toueg hierarchy [3]) encapsulate *fairness*: a measure of the number of steps executed by a process relative to other events in the system. We argue that oracles are substitutable for the fairness properties (rather than real-time properties) of partially synchronous systems. We propose four fairness-based models of partial synchrony and demonstrate that they are, in fact, the 'weakest systems models' to implement the canonical failure detectors from the Chandra-Toueg hierarchy in the presence of arbitrary number of crash faults.

**Significance.** Our results further the shift in the direction of oracular research away from real-time notions of partial synchrony (which have traditionally been understood with respect to events that are essentially external to the system) and towards fairness-based partial synchrony (which can be understood solely with respect to other events that are internal to the system). In fact, our results

suggest that fairness is the currency for crash tolerance and research on weaker real-time bounds for crash tolerance should focus on enforcing appropriate fairness constraints on empirical systems relative to which known oracles can be implemented.

**Organization.** We present related work in Sec. 2. Sect. 3 provides specifications for the asynchronous system model, the four failure detectors, and the four fairness-based partially-synchronous systems that we consider. Sects. 4–6 present the four equivalences between the failure detectors and the fairness-based partially-synchronous systems. We conclude with a discussion in Sect. 7.

## 2 Related Work

**The Chandra-Toueg Hierarchy.** Chandra and Toueg [3] introduced the following four popular oracles: (1) the *perfect failure detector* $\mathcal{P}$, which never suspects any process before the process crashes, after some (unknown) time permanently suspects all the crashed processes, and never transitions from suspecting a process to not suspecting that process; (2) the *eventually perfect failure detector* $\diamond\mathcal{P}$, which after some (unknown) time stops suspecting correct processes and begins to permanently suspect all crashed processes; (3) the *strong failure detector* $\mathcal{S}$, which never suspects some correct process, after some (unknown) time permanently suspects all the crashed processes; (4) the *eventually strong failure detector* $\diamond\mathcal{S}$, which after some (unknown) time stops suspecting some correct process and begins to permanently suspects all the crashed processes.

**Chasing the Weakest Model.** Among the aforementioned four Chandra-Toueg oracles, a significant amount of work focuses on $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$. A line of work has focused on identifying the weakest system model assumptions that suffice for implementing these oracles. One approach is to weaken real-time constraints on synchrony, while another approach is to dispense with real-time altogether and instead constrain the relative ordering of certain events.

Under the first approach, the weakest real-time based message-passing model known to date that is sufficient to implement $\diamond\mathcal{P}$ with arbitrary number of crashes guarantees that relative process speeds are bounded (while absolute speeds may remain unbounded above and below) [22] and that there exists an upper bound on the average delay over subsets of messages that are separated by bounded bursts of messages that may experience unbounded (or infinite) delay [21]. Similarly, the weakest message-passing model known to date that is sufficient to implement $\diamond\mathcal{S}$ in the presence of up to $f$ process crashes guarantees that computation is synchronous and some correct process has $f$ timely outgoing links, although the set of timely links can vary over time [12].

Under the second approach, the weakest fairness-based message passing model known to date for implementing $\diamond\mathcal{P}$ in environments with at least two correct processes are the $\Theta$-model [11] and the ABC model [20]. The $\Theta$-model bounds the ratio of the end-to-end communication delay of messages that are simultaneously in transit, while the ABC model imposes a restriction on the ratio of the

number of messages that can be exchanged between pairs of processes in certain "relevant" segments of an asynchronous execution.

Similarly, for implementing $\diamond\mathcal{S}$, the weakest fairness-based message-passing model known to date was recently proposed by Raynal et al. [1] for systems consisting of $n$ processes with at most $f$ crash faults in which executions progress in "rounds" (the notion of a round is local to each process, not global), and processes send messages to all other processes in each round. A round terminates at a process when the process has received messages from $n - f$ processes for that round. The model guarantees that there exists some correct process $i$ such that eventually some subset of $f$ processes receive a message from $i$ in each of their rounds. Furthermore, this subset of $f$ processes can vary over time, but at all times such a subset exists.

An approach intermediate between the real-time-based and fairness-based approaches is presented by Biely et al. [2]. They prove equivalence (with respect to solvability of some problems) between some models and a set of oracles including $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$. Although the transformations presented in [2] do not preserve bounds on real-time message delay, the authors claim that these bounds are preserved in a "relativistic" sense, but they do not expound on the interpretation of the term "relativistic". Our work formalizes the "relativistic" message delay as a form of *communicational fairness*.

Rajsbaum et al. [18, 19] have tackled the problem of finding the weakest read-write shared memory model for implementing various kinds of oracles. They have shown that the power of so-called *limited scope* oracles can be expressed as restrictions on the number of read and write operations by each process in every round. These results are similar to ours in that they identify the power of oracles with some kind of "fairness". Our results differ from theirs in two ways. First, unlike [18, 19] we investigate the exact synchronism in perpetually accurate oracles $\mathcal{P}$ and $\mathcal{S}$ (we are the first to do so). Second, we consider message-passing systems instead of shared-memory systems. In fact, message-passing systems merit separate investigation because results regarding oracles in shared-memory models in general do not carry over to message-passing models. For instance, the weakest oracle for solving wait-free consensus in asynchronous shared-memory is not the same as that for asynchronous message-passing [14, 5].

## 3 Definitions

### 3.1 Asynchronous System Model

The asynchronous system [10] consists of a finite set of processes $\Pi$ which can communicate with each other by reliable communication channels. We consider the standard asynchronous system model [10], but with *correct-reliable* channels, and we assume that an arbitrary number of processes can crash. A concise description follows. The detailed description of the system model specifications is available in the full version of the paper at [16] and has been omitted here due to space limitations.

While reliable channels deliver every message sent without duplication or corruption, we consider a weaker form of channels called *correct-reliable* channels which are guaranteed to be reliable while both the sender and the receiver are live. Therefore, if the sender and the receiver are correct, then the channel connecting them is guaranteed to be reliable. Otherwise, the channel is guaranteed to behave like a reliable channel until either the sender or the receiver crashes. Afterwards, the channel is allowed to drop the messages in transit (but not corrupt any messages).

We posit the existence of a discrete global time base whose range of values is the natural numbers $\mathbb{N}$. Global time is used to mark or count the events that occur in the system, and it is not used to measure the real-time duration between two events. Therefore, the real-time duration between consecutive ticks of the global time may be arbitrary, but finite. In the remainder of this paper, 'time' will refer to global time unless explicitly stated otherwise.

Processes execute actions in *atomic steps*. In an atomic step, a process receives at most one message from each process, makes a state transition, and sends at most one message to each process. A *run* consists of an infinite sequence of steps taken by processes while executing an algorithm.

We consider only crash faults. That is, a process can fail only by *crashing*, which occurs when a process ceases execution without warning; a crashed process never resumes execution. Any process that is not crashed is considered to be *live*. In each run, processes are either *correct* or *faulty*. Correct processes execute actions according to their algorithm specification, and never fail, whereas *faulty* processes fail after finite time. In all runs, a fault process takes only finitely many steps whereas correct processes take infinitely many steps.

In order to demonstrate our results, we consider two variations of the asynchronous system. In the first variation, we assume that the asynchronous system is augmented with a failure detector. In such systems, the state transition function of each process also considers the output of its local failure detector module before determining the new state of the process and the set of messages to be sent. Sect. 3.2 describes the failure detectors considered in this paper.

In the second variation, we assume that there are certain constraints on the relative ordering of the atomic steps by different processes in the systems. Such constraints determine the *fairness* properties satisfied by runs of these systems. These constraints are described in Sect. 3.3.

### 3.2 Failure Detectors

The formal definitions of the Chandra-Toueg failure detectors are provided in [3]. Informally, failure detectors are characterized by the kind and degree of unreliability of their output which is a set of suspected processes. Here, we consider four classes of failure detectors from the original Chandra-Toueg hierarchy [3]: *Perfect failure detector* (denoted $\mathcal{P}$), *Strong failure detector* (denoted $\mathcal{S}$), *Eventually Perfect failure detector* (denoted $\diamond\mathcal{P}$), and *Eventually Strong failure detector* (denoted $\diamond\mathcal{S}$). All four aforementioned failure detectors guarantee that

*eventually every faulty process is permanently suspected by every correct process.* Additionally, the four failure detectors satisfy the following properties:

- $\mathcal{P}$ ensures that no process suspects any live process.
- $\mathcal{S}$ ensures that *some* correct process is never suspected.
- $\Diamond\mathcal{P}$ ensures that correct processes are eventually never suspected.
- $\Diamond\mathcal{S}$ ensures that *some* correct process eventually is never suspected.

### 3.3 Fairness Constraints

We claim that Chandra-Toueg failure detectors encapsulate *fairness* guarantees of the underlying system. Such fairness is of two kinds: *computational* and *communicational*. *Computational fairness* restricts the number of steps executed by processes relative to each other. *Communicational fairness* restricts the number of steps executed by the recipient of a message while that message is in transit.

**Computational Fairness.** A common specification for computational fairness is *bounded relative process speeds* [8] which states that the system has a bound $\Phi$ on relative process speeds if in all intervals where some process $i$ takes $\Phi+1$ steps, then all the processes not crashed in that interval are guaranteed to take at least 1 step. Note that this fairness property is symmetric; that is, if $i$'s process speed is bounded relative to $j$'s process speed, then *vice versa* is true as well. However, it is possible to define computational fairness properties that are asymmetric.

Consider our definition of proc-fairness. A process $i$ is said to be *$k$-proc-fair* (where $k$ is a non-negative integer) in an infinite suffix $\gamma$ of a run $\alpha$, if, for all processes $j \in \Pi$, in every segment of $\gamma$ in which $j$ takes $k+1$ steps, either (1) $i$ takes at least one step, or (2) $i$ is crashed. Note that $i$ being $k$-proc-fair with respect to $j$ does not imply $j$ being $k$-proc-fair with respect to $i$. As such, proc-fairness is an asymmetric fairness property.

We extend this notion of proc-fairness as follows:

- *$k$-proc-distinguished*: A process $i$ is said to be *$k$-proc-distinguished* in run $\alpha$ if $i$ is $k$-proc-fair in all suffixes of $\alpha$
- *Eventually $k$-proc-distinguished*: A process $i$ is said to be *eventually $k$-proc-distinguished* in $\alpha$ if there exists a prefix of $\alpha$ such that, in the infinite suffix of $\alpha$ that follows, $i$ is $k$-proc-fair.

Like proc-fairness, the property of being proc-distinguished in asymmetric as well. While other processes may be 'fair' with respect to a proc-distinguished process $i$, process $i$ need not be fair with respect to other processes; *i.e.*, a proc-distinguished process may take an unbounded number of steps in the duration between a non-proc-distinguished process' two consecutive steps. This is an important distinction between computational fairness and bounded relative process speeds defined in [8, 7]. Bounded relative process speeds may be viewed as a special case where every process is (eventually) $k$-proc-distinguished.

**Communicational Fairness.** Specifying temporal bounds on communication delay in terms of fairness is not straightforward. For a process $i$ to satisfy communicational fairness, it is necessary that $i$ not take 'too many steps' while a

message $m$ is en route to $i$. However, there is one exception: if the sender of $m$ crashes while $m$ is in transit to $i$, then $i$ can take an arbitrary number of steps before $m$ is delivered. In fact, $m$ may even be dropped.

We capture the above intuition through the following definition for a *com-fair* process. A process $i$ is said to be *d-com-fair* (where $d$ is a non-negative integer) in a suffix $\gamma$ of a run $\alpha$, if, for all processes $j \in \Pi$, for each message $m$ sent from $i$ to $j$ in $\gamma$, during the segment of $\gamma$ starting from the configuration in which $m$ is sent and ending with the configuration in which $m$ is received, either (1) $j$ takes no more than $d$ steps, or (2) $i$ is crashed.

We extend this notion of com-fairness as follows:

- *d-com-distinguished*: A process $i$ is said to be *d-com-distinguished* in run $\alpha$ if $i$ is $d$-com-fair in all suffixes of $\alpha$.
- *Eventually d-com-distinguished*: A process $i$ is said to be *eventually d-com-distinguished* in run $\alpha$ if there exists a prefix of $\alpha$ such that, in the infinite suffix of $\alpha$ that follows, $i$ is $d$-com-fair.

Recall that in traditional partially-synchronous models [8, 7] the bounds on message delay are measured as the number of steps taken by the *sender*. In contrast, our bounds on communicational fairness are measured as the number of steps taken by the receiver, for the following reason. Since these traditional models assume that relative process speeds are bounded, if some live process takes a bounded number of steps while a message is in transit, then all processes take a bounded number of steps while that message is in transit. Hence, asserting the existence of a bound on the number of steps by the sender is equivalent to asserting the existence of a bound on the number of steps by the recipient in the same time interval. In our case, since computational fairness is not a symmetric property, a bound on the number of steps by the sender need not translate to a bound on the number of steps by the receiver in the same time interval. Consequently, we denominate communicational fairness as the number of steps taken by the recipient.

Furthermore, we bound the number of steps taken by the recipient only while the sender is live for the following reason. While the sender is not crashed, it can successfully maintain an operational communication link with the recipient, and the link can ensure that messages are delivered before the recipient takes 'too many steps'. However, if the sender crashes, the link is no longer guaranteed to stay operational, and no guarantees can be provided on message delay and delivery.

### 3.4 Fairness-Based Partially-Synchronous System Models

In this subsection we present four fairness-based partially-synchronous systems models that represent the fairness encapsulated by the four Chandra-Toueg failure detectors specified in Sect. 3.2.

1. *All Fair* ($\mathcal{AF}$) is an asynchronous system model where: in every run, all processes are both $k$-proc-distinguished and $d$-com-distinguished, for known $k$ and $d$.

2. *Some Fair* ($\mathcal{SF}$) is an asynchronous system model where: in every run, some correct process $i$ is both $k$-proc-distinguished and $d$-com-distinguished, for known $k$ and $d$.
3. *Eventually All Fair* ($\diamond\mathcal{AF}$) is an asynchronous system model where: for each run, there exists a (potentially unknown) time after which the system behaves like $\mathcal{AF}$.
4. *Eventually Some Fair* ($\diamond\mathcal{SF}$) is an asynchronous system model where: for each run, there exists a (potentially unknown) time after which the system behaves like $\mathcal{SF}$.

## 4 Methodology

We claim that the Chandra-Toueg oracles encapsulate fairness (and not real-time) properties of the underlying system. We will show that the amount of fairness encapsulated by these oracles is specified by the aforedescribed fairness-based system models. In a precise sense, $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, and $\diamond\mathcal{SF}$ specify the exact amount of fairness encapsulated by $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, respectively. Alternatively, it can be said that in environments where an arbitrary number of processes may crash, $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, and $\diamond\mathcal{SF}$ are the 'weakest' system models to implement $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, respectively.

The methodology used to establish the above equivalence is as follows. First, we present a construction (described in Sect. 5) that uses a Chandra-Toueg oracle in an otherwise asynchronous system to schedule distributed applications such that each process executes its application steps 'fairly' with respect to other processes (and messages). The fairness properties guaranteed by the scheduler depend on the available failure detector. By employing $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, or $\diamond\mathcal{S}$, the scheduler provides fairness guarantees specified by $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, or $\diamond\mathcal{SF}$, respectively. This shows that the failure detectors encapsulate at least as much fairness as is specified in the corresponding fairness-based system models. Next, we present an algorithm (described in Sect. 6) which implements a Chandra-Toueg oracle on top of these fairness-based systems. When this algorithm is deployed in $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, or $\diamond\mathcal{SF}$, it implements $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, or $\diamond\mathcal{S}$, respectively. Thus, we show that these failure detectors encapsulate no more guarantees on fairness than what is provided by the corresponding fairness-based systems.

## 5 Extracting Fairness

In this section, we present a distributed scheduler that 'extracts' the fairness encapsulated by the Chandra-Toueg failure detectors. The algorithm presented is a universal construction for the Chandra-Toueg hierarchy in the sense that depending on the failure detector used by the algorithm, the appropriate fairness guarantees are provided by the distributed scheduler. For simplicity, we assume that the application at each process always has some enabled step that it can take. Therefore, the local scheduler module is always in one of two states:

*waiting* and *active*. When the scheduler module is *waiting*, the associated application module is not enabled to take steps. Upon becoming *active*, the scheduler module enables the associated application module to execute a single step and then the scheduler goes back to *waiting*. Additionally, the distributed scheduler 'intercepts' and forwards all the communication among the application modules.

The properties to be satisfied by the distributed scheduler are *local progress* and *fairness*. *Local progress* states that every correct process must be scheduled to execute its application steps infinitely often, regardless of process crashes in the system. *Fairness* properties are as follows:

If the distributed scheduler uses:

- $\mathcal{P}$, then the distributed scheduler implements the $\mathcal{AF}$ system model.
- $\mathcal{S}$, then the distributed scheduler implements the $\mathcal{SF}$ system model.
- $\diamond\mathcal{P}$, then the distributed scheduler implements the $\diamond\mathcal{AF}$ system model.
- $\diamond\mathcal{S}$, then the distributed scheduler implements the $\diamond\mathcal{SF}$ system model.

### 5.1 Interface Between Scheduler And Application

The scheduler provides three interfaces for an application module to interact with the local scheduler module and the application modules at other processes: $executeAPP()$, $receiveAPP()$, and $sendAPP()$. These interfaces are specified in Alg. 1.2 and described below.

The scheduler enables the application to take a step by invoking $executeAPP()$ and in response, the application takes a single atomic step. If multiple actions of the application are enabled to be executed, then the scheduler is assumed to make a non-deterministic choice among the enabled actions subject to the constraint of weak fairness (which states that a continuously enabled action is eventually executed).

The application receives messages sent by other processes by invoking $receiveAPP()$. The scheduler at each process $i$ takes all the messages destined for the application module at $i$ and stores them locally in a *receive buffer*. When the application invokes $receiveAPP()$, the scheduler returns the contents of the *local receive buffer* to the application.

The application sends messages by invoking the $sendAPP()$ interface. While taking a step, if the application at process $i$ invokes $sendAPP()$, the scheduler at $i$ stores all the messages that the application wants to send to all the processes in a *local send buffer*. The scheduler module at $i$ then sends the messages to destination process where they are stored in the *receive buffers* of the corresponding scheduler modules (in Actions 5 and 6 of Alg. 1.1 as described in Sect. 5.2). These messages are then received by the respective recipient processes when the latter invoke $receiveAPP()$.

### 5.2 Algorithm Description

The algorithm in Algs. 1.1 and 1.2 implements a distributed scheduler with dynamic heights (or priorities) and permits. Alg. 1.1 shows the actions of the

enum $\{waiting, active\}$ : $s_i.\mathsf{state} \leftarrow waiting$ *State variable is initially set to waiting*
**integer** $s_i.\mathsf{ht} \leftarrow 0$ *The height of process i*
$\forall j \in \Pi - \{i\}$ :
  **boolean** $s_i.\mathsf{permit}_j \leftarrow (i.id > j.id)$ *Process with higher id holds the shared permit*
  **boolean** $s_i.\mathsf{req}_j \leftarrow (i.id < j.id)$ *Process with lower id holds the shared request token*
  **integer** $s_i.\mathsf{ht}_j \leftarrow 0$ *Process i's view of the height of process j*
  **integer** $s_i.\mathsf{seq}_j \leftarrow 0$ *Generates a new sequence number to solicit messages from j*
  **integer** $s_i.\mathsf{maxAck}_j \leftarrow 0$ *The highest seq. no. among the messages received from j*
  set $s_i.\mathsf{send\_buffer}_j$ *The send buffer through which apps. at i send messages to j*
  set $s_i.\mathsf{receive\_buffer}_j$ *The receive buffer from which apps. at i receive msgs. from j*
**boolean** $s_i.\mathsf{permit}_j \leftarrow$ **true** *Process always holds its own the shared permit*

---

1 : $\{s_i.\mathsf{state} = waiting\} \longrightarrow$ *Action 1*
2 : $\forall j \in \Pi - \{i\}$ **where** $s_i.\mathsf{req}_j \wedge \neg s_i.\mathsf{permit}_j$ **do** *Request permit*
3 : **send** $\langle request, s_i.\mathsf{ht}\rangle$ **to** $s_j$; $s_i.\mathsf{req}_j \leftarrow false$

---

4 : $\{$**upon receive** $\langle \mathsf{request}, \mathsf{ht}\rangle$ **from** $s_j\} \longrightarrow$ *Action 2*
5 : $s_i.\mathsf{req}_j \leftarrow true$ *Send permit if $s_i$ is waiting*
6 : $s_i.\mathsf{ht}_j \leftarrow ht$ *and $s_j$ has higher priority*
7 : **if** $(s_i.\mathsf{permit}_j \wedge (s_i.\mathsf{state} = waiting) \wedge ((ht > s_i.\mathsf{ht}) \vee ((ht = s_i.\mathsf{ht}) \wedge (i < j)))$
8 : **send** $\langle permit, s_i.\mathsf{ht}\rangle$ **to** $s_j$; $s_i.\mathsf{permit}_j \leftarrow false$

---

9 : $\{$**upon receive** $\langle \mathsf{permit}, \mathsf{ht}\rangle$ **from** $s_j\} \longrightarrow$ *Action 3*
10 : $s_i.\mathsf{permit} \leftarrow true$ *Send permit if $s_i$ is waiting*
11 : $s_i.\mathsf{ht}_j \leftarrow ht$ *and $s_j$ has higher priority*
12 : **if** $(s_i.\mathsf{req}_j \wedge (s_i.\mathsf{state} = waiting) \wedge ((ht > s_i.\mathsf{ht}) \vee ((ht = s_i.\mathsf{ht}) \wedge (i < j)))$
13 : **send** $\langle permit, s_i.\mathsf{ht}\rangle$ **to** $s_j$; $s_i.\mathsf{permit}_j \leftarrow false$

---

14 : $\{(s_i.\mathsf{state} = waiting) \wedge (\forall j \notin \mathcal{D} :: s_i.\mathsf{permit}_j)\} \longrightarrow$ *Action 4 (Note: $\mathcal{D}$ is queried)*
15 : $s_i.\mathsf{state} \leftarrow active$ *Active upon holding permits from trusted processes*
16 : **foreach** $j$ in $\Pi - \{i\}$
17 : increment $s_i.\mathsf{seq}_j$ by 1 *Generate a new seq. no. to tag a request message*
18 : **send** $\langle reqMsg, s_i.\mathsf{seq}_j\rangle$ **to** $s_j$ *Send a request message to all processes*

---

19 : $\{$**upon receive** $\langle reqMsg, num\rangle$ **from** $s_j\} \longrightarrow$ *Action 5*
20 : $msgSet \leftarrow s_i.\mathsf{send\_buffer}_j$ *Received a mesg request.*
21 : $s_i.\mathsf{send\_buffer}_j \leftarrow \emptyset$
22 : **send** $\langle msgSet, num\rangle$ **to** $s_j$ *Send the contents of the local send buffer*

---

23 : $\{($**upon receive** $\langle msgSet', num\rangle$ **from** $s_j)\} \longrightarrow$ *Action 6*
24 : $s_i.\mathsf{receive\_buffer}_j \leftarrow s_i.\mathsf{receive\_buffer}_j \cup msgSet'$ *Add to local receive buffer*
25 : $s_i.\mathsf{maxAck}_j \leftarrow max(num, s_i.\mathsf{maxAck}_j)$ *Update max. ack receive so far.*

---

26 : $\{(s_i.\mathsf{state} = active) \wedge (\forall j \in \Pi - \{i\} :: ((s_i.\mathsf{maxAck}_j = s_i.\mathsf{seq}_j) \vee (j \in \mathcal{D})))\} \longrightarrow$
*Action 7 (Note that the failure detector $\mathcal{D}$ is queried)*
27 : $executeAPP()$ *Execute an app. step; $executeAPP()$ is specified in Alg. 1.2*
28 : $s_i.\mathsf{ht} \leftarrow \min(\forall j \in \Pi - \{i\} :: s_i.\mathsf{ht}_j, s_i.\mathsf{ht}) - 1$ *Reduce height below all neighbors*
29 : $\forall j \in \Pi - \{i\}$ **where** $(s_i.\mathsf{permit}_j)$ *do whose height is known.*
30 : **send** $\langle permit, s_i.\mathsf{ht}\rangle$ **to** $s_j$; $s_i.\mathsf{permit}_j \leftarrow false$ *Send all held permits*
31 : $s_i.\mathsf{state} \leftarrow waiting$ *Exit the active state after executing an app. step*

**Alg. 1.1.** Actions for scheduler at process $i$.

**procedure** $executeAPP()$
  execute an enabled application action in which
    application action invokes $receiveAPP()$ to receive messages
    application action invokes $sendAPP(m, j)$ to send message $m$ to process $j$

**procedure** $receiveAPP()$
  $returnValue \leftarrow \cup_{\forall j \in \Pi - \{i\}} \{(s_i.\mathsf{receive\_buffer}_j, j)\}$
  $\forall j \in \Pi - \{i\}$ **do** $s_i.\mathsf{receive\_buffer}_j \leftarrow \emptyset$
  **return** $returnValue$

**procedure** $sendAPP(m, j)$
  $s_i.\mathsf{send\_buffer}_j \leftarrow s_i.\mathsf{send\_buffer}_j \cup \{m\}$

**Alg. 1.2.** Interaction between the scheduler and the application.

scheduler and Alg. 1.2 shows the interface between the scheduler and the scheduled application. The idea of dynamic heights and permits (also called *forks*) is borrowed from the algorithms to solve the dining philosophers problem in [17]. All the processes are assigned a static id and all the ids are known to all the processes in the system.

In Alg. 1.1 each process $i$ has the following variables: $s_i.\mathsf{state}$ which determines if the process is *waiting* or *active*. The height of a process is stored in the variable $s_i.\mathsf{ht}$ which is initially 0. For each process $j$ in the system, $i$ maintains the variables: (a) $s_i.\mathsf{permit}_j$ to determine if the permit shared with $j$ is currently held by $i$, (b) $s_i.\mathsf{req}_j$ to determine if the request token to request a permit from $j$ is currently at $i$, and (c) $s_i.\mathsf{ht}_j$ which stores the last received value of $j$'s height (in permits and request messages).

All processes start in the *waiting* state with the permits at higher-id processes and request tokens at lower-id processes. For a *waiting* process to become *active*, it must collect all its shared permits. A *waiting* process requests missing permits in Action 1. Upon receiving such a request in Action 2, the process determines if the request should be honored based on the following condition: if the process is *waiting*, holds the shared permit, and the requesting process has greater height (or equal height and higher process-id), then the process relinquishes the permit. Otherwise the process simply holds the token and defers sending the permit if the permit is present.

Upon receiving a permit in Action 3, the process again determines if the permit should be kept (to be sent later) or sent based on the same condition mentioned previously.

When a *waiting* process (say) $i$ receives shared permits from all the processes not suspected by the failure detector $\mathcal{D}$, $i$ becomes *active* in Action 4. Upon becoming *active*, $i$ sends an application-message request (denoted $\langle reqMsg \rangle$) with a new sequence number ($s_i.\mathsf{seq}_j$) to each process $j$ in the system. Upon receiving such a message in Action 5, process $j$ sends the contents of its *local send buffer* appended with the sequence number in response. Process $i$ receives such a message sent by $j$ in Action 6; process $i$ adds the contents of the received message

to its *local receive buffer* and updates its local state to reflect the latest sequence number for which $i$ has received a response from $j$ (stored in $s_i.\mathsf{macAck}_j$). Upon receiving responses from all trusted processes for the $\langle reqMsg \rangle$ messages sent with the latest sequence number (that is, $s_i.\mathsf{seq}_j = s_i.\mathsf{maxAck}_j$ for all $j$ trusted by $i$), Action 7 is enabled at process $i$. In Action 7, process $i$ invokes $executeAPP()$ to execute an application step before exiting.

This mechanism of receiving application messages before invoking *execute* $APP()$ ensures that an active process $i$ 'waits on' all the messages sent by a correct and trusted process $j$, thus guaranteeing that a correct and trusted process $j$ is also a com-distinguished process.

When a process executes an application step, the application invokes *receive* $APP()$ described in Alg. 1.2 to receive all the messages in the *local receive buffer*, and the application action sends messages by invoking $sendAPP()$ described in Alg. 1.2 which simply adds the message to the *local send buffer*.

Eventually, the process exits its *active* state by reducing it height below all processes (whose shared permits it holds), sends all the permits away and transits to *waiting* in Action 7.

Relinquishing the shared permits before *waiting* ensures that a correct and trusted process receives permits from other processes every time the other processes take an application step. The reduction in height ensures that the process does not 'steal' the permits (by sending a request token with greater height) after relinquishing them. This allows a correct and trusted process to become a proc-distinguished process as well.

The proof of correctness is available in the full version of the paper at [16] and has been omitted here due to space limitations.

## 6 Extracting Chandra-Toueg Failure Detectors From Fairness-Based Systems

In this section we present an algorithm that implements the failure detectors $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$ in the system models $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, and $\diamond\mathcal{SF}$, respectively. This result combined with the result in Sect. 5 shows that $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, and $\diamond\mathcal{SF}$ have the minimal synchronism necessary to implement $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, respectively. The algorithm is as follows:

The failure detector module at each process $i$ maintains a timer $\mathsf{timerValue}_j$ for each process $j$ in the system which counts down from $k + d$ to 0, where the bounds on fairness in the system models of Sect. 3.4 are specified by the existence of $k$-proc-distinguished and $d$-com-distinguished processes. Every time process $i$ takes a step, it receives zero or more messages from all other processes, decrements the value of $\mathsf{timerValue}_j$ by 1, and sends a heartbeat to each process $j$ in the system. If $i$ receives a heartbeat from $j$, then $i$ trusts $j$ and resets the value of $\mathsf{timerValue}_j$ to $k + d$. If $\mathsf{timerValue}_j$ is decremented to 0, then $i$ suspects $j$. The pseudo-code for the algorithm is given in Alg. 1.3.

The proof of correctness is available in the full version of the paper at [16] and has been omitted here due to space limitations.

```
constant timeOut ← k + d
set suspectList ← ∅
∀j ∈ Π − {i} :
  integer timerValue_j ← timeOut
-------------------------------------------------------------------------------
1 :  {true} ⟶                                                        Action 1
2 :     receive ⟨msgSet⟩          Receives zero or more messages from each process
3 :     ∀j ∈ Π − {i} do
4 :       send ⟨HB⟩ to j                          Send a heartbeat to each process
5 :       if (⟨HB, j⟩ ∈ msgSet)
6 :         suspectList ← suspectList − {j}       Trust upon receiving a heartbeat
7 :         timerValue_j ← timeOut                                  Reset timer
8 :       if (timerValue_j = 0)
9 :         suspectList ← suspectList ∪ {j}                Suspect upon timer expiry
10 :      timerValue_j ← max(timerValue_j − 1, 0)   Decrement timer for each process
```

**Alg. 1.3.** Implementing Chandra-Toueg Oracles In System Models Where (Some) Processes are $k$-Proc-Distinguished and $d$-Com-Distinguished

## 7 Discussion

**Complete Synchrony and $\mathcal{P}$.** It was first noted in [4] that there exist time-free problems solvable in synchronous systems that are unsolvable with $\mathcal{P}$. This points to a 'gap in the synchronism' between $\mathcal{P}$ and the synchronous system. The following corollary of our results explains this gap.

$\mathcal{AF}$ — the weakest system model to implement $\mathcal{P}$ — is extremely similar to the synchronous system model with message delay being denominated in recipient's steps in the former and in real time in the latter. However, there is one significant difference. $\mathcal{AF}$ ensures full synchrony for all messages as long as the senders are live. When a sender crashes, $\mathcal{AF}$ 'loses synchronism' for all the sender's messages that are still in transit. On the other hand, synchronous systems ensure the synchronism for these messages as well. This difference in the behavior between $\mathcal{AF}$ and synchronous systems is the 'gap in synchronism' between the perfect failure detector $\mathcal{P}$ and synchronous systems. To our knowledge, we are the first to characterize this gap.

**On Solving Consensus.** It is well known that $\diamond\mathcal{S}$ is the weakest failure detector to solve consensus in asynchronous systems with a majority of correct processes [14], and we have shown that $\diamond\mathcal{SF}$ is the weakest fairness-based system model to implement $\diamond\mathcal{S}$. Does that mean $\diamond\mathcal{SF}$ is the weakest system model to solve consensus? The answer is *no*. While $\diamond\mathcal{S}$ is the weakest to solve consensus only in majority-correct environments, $\diamond\mathcal{SF}$ is the weakest to implement $\diamond\mathcal{S}$ in all environments. This obervationsuggests that there is a weaker system model which can implement $\diamond\mathcal{S}$ in majority-correct environments, but not in all environments.

**Open Questions.**

We have argued that several oracles encapsulate fairness in executions and provided evidence by demonstrating that the oracles in the Chandra-Toueg hierarchy encapsulate such fairness constraints. This opens a larger question: *do all oracles encapsulate fairness?* The answer is arguably *no*. Notable candidates for counterexamples include the failure detectors proposed in [13] whose output can be arbitrary and need not provide semantic information about process crashes alone. This presents another question: *what set of oracles do encapsulate fairness?* This question is open even when restricted to the extended Chandra-Toueg hierarchy (which include oracles like $\mathcal{T}$ [6], and other parametric oracles like the ones in [2, 19]). If it turns out that all oracles that output process ids do encapsulate fairness, then it provides us with a clean hierarchy of fairness-based system models that mirrors the extended Chandra-Toueg hierarchy. On the other hand, if we discover that there exist oracles within the extended Chandra-Toueg hierarchy that do not encapsulate fairness, then the implication is that these oracles encapsulate something other than fairness. Knowledge of this other encapsulated information could help in designing crash tolerant systems.

Another consequence of oracles encapsulating fairness is that fault environments might encapsulate fairness as well. Recall that the weakest oracles sufficient to solve problems in distributed systems vary depending on the number of processes that may crash. For instance, consider fault-tolerant consensus. Recall that $\diamond\mathcal{S}$ is the weakest to solve the problem only in majority-correct environments [14]. In environments where an arbitrary number of processes may crash, the weakest failure detector for the problem is a stronger oracle $(\diamond\mathcal{S}, \Sigma)$ [5]. Given that $\diamond\mathcal{S}$ encapsulates some fairness constraints, and $\Sigma$ can be implemented in an asynchronous system with majority correct, we conjecture that $\Sigma$ and majority-correct encapsulate equivalent fairness constraints in the system. Furthermore, this implies that fairness is also encapsulated by constraints on the number of processes that may crash in the system. Based on the above observations and arguments, consider the following question: Is fairness a more general primitive to understand crash fault tolerance in distributed systems? That is, can fairness unify the different weakest failure detector results for the same problem in different fault environments?

Much effort is spent pursuing the 'weakest' real-time-based models to implement certain oracles (like $\Omega$, $\diamond\mathcal{P}$, and such) for two reasons: (1) bounds in many empirical distributed systems are specified with respect to real time, and (2) these oracles are known to be the weakest to solve many problems in distributed computing. However, given the dependence of the weakest-oracle results on the fault environment, and the conjecture that fault environments themselves could encapsulate fairness, it is perhaps beneficial to investigate the 'weakest' real-time-based models to guarantee appropriate fairness constraints (rather than oracles) so that these constraints can then be encapsulated by various combinations of oracles and fault environments.

**Acknowledgment.** We would like to thank the anonymous reviewers and Michel Raynal for their valuable comments and suggestions to improve this paper.

## References


1. Anta, A.F., Raynal, M.: From an asynchronous intermittent rotating star to an eventual leader. IEEE Transactions on Parallel and Distributed Systems 21(9), 1290–1303 (2010), http://dx.doi.org/10.1109/TPDS.2009.163
2. Biely, M., Hutle, M., Penso, L.D., Widder, J.: Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In: Proceedings of the $9^{th}$ International Symposium on Stabilization, Safety, and Security of Distributed Systems. pp. 4–20 (2007), http://dx.doi.org/10.1007/978-3-540-76627-8_4
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM 43(2), 225–267 (1996), http://dx.doi.org/10.1145/226643.226647
4. Charron-Bost, B., Guerraoui, R., Schiper, A.: Synchronous system and perfect failure detector: solvability and efficiency issues. In: International Conference on Dependable Systems and Networks. pp. 523–532 (2000), http://doi.ieeecomputersociety.org/10.1109/ICDSN.2000.857585
5. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In: Proceedings of the 23rd ACM symposium on Principles of Distributed Computing (PODC). pp. 338–346 (2004), http://dx.doi.org/10.1145/1011767.1011818
6. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Journal of Parallel and Distributed Computing 65(4), 492–505 (2005), http://dx.doi.org/10.1016/j.jpdc.2004.11.008
7. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM 34(1), 77–97 (1987), http://doi.acm.org/10.1145/7531.7533
8. Dwork, C., Lynch, N.A., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM 35(2), 288–323 (Apr 1988), http://doi.acm.org/10.1145/42282.42283
9. Fich, F., Ruppert, E.: Hundreds of impossibility results for distributed computing. Distributed Computing 16(2-3), 121–163 (2003), http://dx.doi.org/10.1007/s00446-003-0091-y
10. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985), http://doi.acm.org/10.1145/3149.214121
11. Hermant, J.F., Widder, J.: Implementing reliable distributed real-time systems with the $\Theta$-model. In: Proceedings of the 9th International Conference on the Principles of Distributed Systems. pp. 334–350 (2005), http://dx.doi.org/10.1007/11795490_26
12. Hutle, M., Malkhi, D., Schmid, U., Zhou, L.: Chasing the weakest system model for implementing $\Omega$ and consensus. IEEE Transactions on Dependable and Secure Computing 6(4), 269–281 (2009), http://dx.doi.org/10.1109/TDSC.2008.24
13. Jayanti, P., Toueg, S.: Every problem has a weakest failure detector. In: Proceedings of the $27^{th}$ ACM symposium on Principles of distributed computing. pp. 75–84 (2008), http://doi.acm.org/10.1145/1400751.1400763
14. Lo, W.K., Hadzilacos, V.: Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In: Proceedings of the

8th International Workshop on Distributed Algorithms. pp. 280–295 (1994), http://dx.doi.org/10.1007/BFb0020440
15. Mostefaoui, A., Mourgaya, E., Raynal, M.: An introduction to oracles for asynchronous distributed systems. Future Gener. Comput. Syst. 18(6), 757–767 (2002), http://dx.doi.org/10.1016/S0167-739X(02)00048-1
16. Pike, S.M., Sastry, S., Welch, J.L.: Failure detectors encapsulate fairness. Tech. Rep. 2010-7-1, Department of Computer Science and Engineering, Texas A&M University (2010), http://www.cse.tamu.edu/academics/tr/2010-7-1
17. Pike, S.M., Song, Y., Sastry, S.: Wait-free dining under eventual weak exclusion. In: Proceedings of the 9th International Conference on Distributed Computing and Networking. pp. 135–146 (2008), http://dx.doi.org/10.1007/978-3-540-77444-0_11
18. Rajsbaum, S., Raynal, M., Travers, C.: Failure detectors as schedulers (an algorithmically-reasoned characterization). Tech. Rep. 1838, IRISA, Université de Rennes, France (2007), http://hal.inria.fr/inria-00139317/PDF/PI-1838.pdf
19. Rajsbaum, S., Raynal, M., Travers, C.: The iterated restricted immediate snapshot model. In: Proceeding of $14^{th}$ Annual International Conference on Computing and Combinatorics. pp. 487–497 (2008), http://dx.doi.org/10.1007/978-3-540-69733-6_48
20. Robinson, P., Schmid, U.: The Asynchronous Bounded-Cycle Model. In: Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems. pp. 246–262 (2008), http://dx.doi.org/10.1007/978-3-540-89335-6_20
21. Sastry, S., Pike, S.M.: Eventually perfect failure detection using ADD channels. In: Proceedings of the 5th international Symposium on Parallel and Distributed Processing and Applications. pp. 483–496 (2007), http://dx.doi.org/10.1007/978-3-540-74742-0_44
22. Sastry, S., Pike, S.M., Welch, J.L.: Crash fault detection in celerating environments. In: Procceding of the $23^{rd}$ IEEE International Parallel and Distributed Processing Symposium. pp. 1–12 (2009), http://dx.doi.org/10.1109/IPDPS.2009.5161050