Failure Detectors Encapsulate Fairness[⋆]

Scott M. Pike[1], Srikanth Sastry[2], and Jennifer L. Welch[3]

[1] Dept. of Computer Science and Engineering
Texas A&M University
College Station TX-77843 USA
pike@cse.tamu.edu

[2] Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge MA-02139 USA
sastry@csail.mit.edu

[3] Dept. of Computer Science and Engineering
Texas A&M University
College Station TX-77843 USA
welch@cse.tamu.edu

Abstract. Failure detectors have long been 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 failure detectors 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 system models' to implement the canonical failure detectors from the Chandra-Toueg hierarchy. We also propose a set of fairness-based models which encapsulate the $\mathcal{G}_c$ parametric failure detectors which eventually and permanently suspect crashed processes, and eventually and permanently trust some fixed set of c correct processes.

Keywords: Failure Detectors, Partial Synchrony, Fairness, Crash Faults, Fault Tolerance, Schedulers

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 [22, 21]. Efforts to circumvent this impossibility have spawned two complementary approaches. The first approach, called *partial synchrony* [18, 17], 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* [11], 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) every problem solvable in an asynchronous system augmented with $\mathcal{D}$ is also solvable in system model M, and (2) every problem solvable in system model M is also solvable in an asynchronous system augmented with $\mathcal{D}$. As such, if $\mathcal{D}$ encapsulates M, then $\mathcal{D}$ is substitutable for M because any problem solvable in M is

[⋆] A preliminary version of this paper was presented at the 14th International Conference On Principles Of Distributed Systems (OPODIS) in 2010. This work is supported in part by NSF grants CCF-0964696 and CCF-0937274, and Texas Higher Education Coordinating Board grant NHARP 000512-0130-2007. This works is also partially supported by Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

also solvable in asynchrony augmented with $\mathcal{D}$. Alternatively (and informally), the notion of *encapsulation* by a failure detector may be viewed synonymously with the notion of *mutual reducibility*; that is, a failure detector $\mathcal{D}$ encapsulates a system model M if and only if (1) there exists an algorithm that implements $\mathcal{D}$ in system model M, and (2) there exists an asynchronous algorithm that queries $\mathcal{D}$ and implements a 'virtual' system that satisfies the properties of M.

Partial Synchrony. A system model is partially synchronous [18] if it provides potentially incomplete, or unknown, temporal bounds on computational and/or communicational quantities such as message delays and process speeds. 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.*, [18, 17, 27, 40, 41, 26, 39]). 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 [11] 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 augmenting the asynchronous system with either (1) adequate degrees of partial synchrony [18] or (2) sufficiently powerful oracles [31]. 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 [11]) 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 system models' to implement the canonical failure detectors from the Chandra-Toueg hierarchy in the presence of arbitrary number of crash faults. We extend our results to the failure detectors introduced in [6] and propose a family of fairness-based partially synchronous models that are encapsulated by these failure detectors.

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 that 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.

The constructions presented in the paper may be of independent interest. The scheduler presented in Sect. 5 is 'universal' (relative to the canonical failure detectors from [11] and [6]) in the sense that a single algorithm for the scheduler uses the available failure detector as a 'plugin' and automatically enforces the maximal fairness encapsulated by that failure detector. Similarly, the failure-detector algorithm in Sect. 6 is agnostic to the guarantees on fairness provided by the underlying system model; for fairer system models, the algorithm automatically implements stronger failure detectors.

Organization. We present related work in Sect. 2. Sect. 3 provides specifications for the asynchronous system model, the four failure detectors from [11], 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 use the constructions from Sects. 4–6 in Sect. 7 and present a new family of fairness-based system models that are equivalent to the failure detectors introduced in [6]. Finally, we conclude with a discussion in Sect. 8.

2 Related Work

The Chandra-Toueg Hierarchy. In [11], Chandra and Toueg introduced eight failure detector oracles, which form the Chandra-Toueg hierarchy. It was also shown in [11] that the Chandra-Toueg hierarchy can be collapsed to the following four 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 trusts a previously suspected process; (2) the *eventually perfect failure detector* ($\diamond\mathcal{P}$), which eventually and permanently stops suspecting correct processes and permanently suspects all crashed processes; (3) the *strong failure detector* ($\mathcal{S}$), which never suspects some correct process, and eventually and permanently suspects all the crashed processes; (4) the *eventually strong failure detector* ($\diamond\mathcal{S}$), which eventually and permanently stops suspecting some correct process and permanently suspects all the crashed processes.

Chasing the Weakest Message-Passing Model. Among the aforementioned four oracles, a significant amount of research focuses on $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$[4]. Among the many results on $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$, a line of work has focused on identifying the weakest system model assumptions that suffice for implementing these oracles in message-passing systems. 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, consider system models to implement $\diamond\mathcal{S}$. It is shown in [11] that a system with unknown and eventual bounds on relative process speeds and message delay is sufficient to implement $\diamond\mathcal{P}$, and thus to implement $\diamond\mathcal{S}$. Subsequently, it is shown in [4] that $\diamond\mathcal{S}$ can be implemented in a system model where all processes execute in lock-step synchrony and there exists some correct process whose links are eventually timely; that is, eventually there is an upper bound on the message delay on these links. In later work, focus shifts to the weakest system model to implement $\diamond\mathcal{S}$ (or the failure detector Ω [10][5] which is equivalent to $\diamond\mathcal{S}$) in environments where up to f processes may crash. Furthermore, [30] shows that Ω (and hence, $\diamond\mathcal{S}$) can be implemented in system models where eventually some correct process has f bidirectional links at all times. Note that the set of f timely links need not be fixed and may vary throughout the execution. Independently, it is shown in [3] that Ω can be implemented in systems where some f outgoing links at some correct process are eventually timely. The latter two results are superseded by [27] which shows that Ω can be implemented in systems where eventually some correct process has f timely outgoing links and the set of f timely links can vary throughout the execution.

Similarly, consider $\diamond\mathcal{P}$ implementations. It is shown in [11] that a system with unknown and/or eventual bounds on relative process speeds and message delay is sufficient to implement $\diamond\mathcal{P}$. Subsequently, it is shown in [20] that some upper bound on the *average* delay of messages was sufficient to implement $\diamond\mathcal{P}$ in a system with an unknown upper bound on absolute process speed, as long as the system uses stubborn links [25]. The model is weakened to accommodate infinite message loss in [40] but still assumes a lower bound on absolute process speed. The bound on absolute process speeds is relaxed in [41] to permit arbitrary process speeds while maintaining a bound on relative process speeds.

[4] The popularity of $\diamond\mathcal{S}$ and $\diamond\mathcal{P}$ is not just incidental. Despite apparently weak guarantees on crash fault detection, $\diamond\mathcal{S}$ has been shown to solve consensus and other related problems [11, 10], and $\diamond\mathcal{P}$ has been shown to solve problems including dining philosophers [35, 36], stable leader election [2], quiescent reliable communication [1], and contention management [24].

[5] The leader-election failure detector Ω outputs the id of a process at each process. There is a time after which it outputs the id of the same correct process at all correct processes.

Under the second approach which constrains the relative ordering of certain events, consider $\Diamond\mathcal{S}$ implementations. The first fairness-based system model to implement $\Diamond\mathcal{S}$ is proposed in [19]. The models proposed in [19], called Message Classification Models or MCM, classify messages as either 'fast' or 'slow' based on their delays measured by some global clock that need not measure passage of real time[6]. In these models, a message may be deliberately delayed to make it 'slow'. Let S denote the set of system models where there exists a correct process p and a (potentially unknown) time t such that every message sent to p, but not deliberately delayed, is a 'fast' message. The results in [19] demonstrate the system models in S are sufficiently powerful to implement $\Diamond\mathcal{S}$.

The next significant fairness-based model sufficient to implement $\Diamond\mathcal{S}$ is proposed in [32] (extended in [33]) 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 fixed subset consisting of f processes receive a message from i in each of their rounds. Subsequently, a weaker system model (and weakest-to-date) is proposed in [5] which permits this subset consisting of f processes to vary over time, as long as (eventually) at all times such a subset exists.

Similarly, among $\Diamond\mathcal{P}$ implementations, the weakest to-date fairness-based message-passing models that are sufficient for implementing $\Diamond\mathcal{P}$ are the Θ-model [26] and the ABC model [39]. The Θ-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. However, all $\Diamond\mathcal{P}$ implementations in these models require at least two processes to be correct. Incidentally, the Θ-model and the ABC model may be viewed as special cases of the MCM models from [19].

Note that all the above proposed system models, while claiming to be weakest to-date to implement their respective failure detectors, do not claim to be *the weakest* to do so. The closest result to the 'weakest' message-passing system model to implement $\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$, and other eventually accurate failure detectors is [6] which follows an approach intermediate between the real-time-based and fairness-based approaches. The results in [6] demonstrate that with respect to solvability $\Diamond\mathcal{S}$, $\Diamond\mathcal{P}$, and other failure detectors are "equivalent" to various partially synchronous models. The authors of [6] are aware that their transformations do not preserve bounds on real-time message delay. They claim that the bounds on message delay are preserved in a 'relativistic' sense (in the extended technical report [7]), 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.*

Weakest Models for Failure Detectors in Shared Memory. The notion of 'capturing the power' of a failure detector is explored in [37, 38] for shared-memory systems. The results in [37, 38] show that the 'power' of *limited scope accuracy*[7] [42] failure detectors in systems with single-writer/multi-reader atomic registers can be expressed as restrictions on the number of read/write operations by each process in every round (in other words, *fairness* in the number of read/write operations per process per round). Our work, which focuses on message-passing systems, deviates from [38] in three significant ways.

First, the weakest failure detectors for solving problems in message-passing systems may be different from shared-memory systems. For example, consider wait-free consensus. The weakest failure detector for this problem in asynchronous shared-memory systems is Ω [29] whereas the weakest failure detector to solve the same problem in message-passing systems is (Ω, Σ)[8]; that is, process have access to two failure

[6] For instance, a message could be classified as 'slow' if the message experiences a delay of at most (say) two times the delay experienced by a 'fast' message

[7] Limited scope accuracy is a version of weak accuracy wherein a correct process need not be trusted by all other processes but only by a subset of the processes that are ostensibly 'near' the correct process. Limited scope accuracy captures the idea that a process may detect failures reliably on the same local-area network, but less reliably over a wide-area network.

[8] Incidentally, the reason for the discrepancy between shared memory and message passing systems is the following. The Ω failure detector is necessary to ensure the progress property of consensus: every correct process eventually

detectors Ω and Σ [15]. Similarly, for solving wait-free k-set agreement, the weakest failure detector in shared-memory systems is anti-Ω_k[9][23], but the weakest failure detector in message-passing systems remains an open problem [8, 9]. Therefore, 'capturing the power' of a failure detector in message-passing system merits a separate investigation.

Second, asynchronous systems under message passing are 'less synchronous' than asynchronous systems under shared-memory. In fact, an asynchronous shared-memory system is equivalent to an asynchronous system under message passing that is augmented with the quorum failure detector Σ [15]. Consequently, the results from [38] need not (and do not) carry over to message-passing systems.

Third, the results in [38] demonstrate fairness constraints only for classes of eventually accurate oracles; in contrast, our results address the synchronism captured by fairness constraints for both eventually accurate oracles and perpetually accurate oracles. To our knowledge, our work is the first to establish such equivalence between partial synchrony and failure detectors with perpetual accuracy. As a consequence of our results, we answer a question implicitly posed in [13]: Given that synchronous systems are 'more synchronous' than the perfect failure detector $\mathcal{P}$, what is the 'gap in synchronism' between $\mathcal{P}$ and synchronous systems? We answer this question in Sect. 8.

3 Definitions

This section provides the formal framework and definitions used in the rest of the paper.

We first specify our system model. It is based on the asynchronous system model in [22], but differs from it in two respects: (1) each process has access to a failure detector, and (2) messages sent are only guaranteed to be delivered if both the sender and the receiver are correct. The same formalism can also be used to model a system in which processes do not have access to failure detectors by having the failure detectors return no information. In Sect. 3.2, we define four popular failure detectors, first introduced in [11]. Subsequently, our fairness constraints are presented in Sect. 3.3 and used in Sect. 3.4 to specify fairness-based partially synchronous system models. Subsequently, these fairness-based partially synchronous system models are shown to be equivalent to the failure detectors from Sect. 3.2. Finally, in Sect. 7, we use the definitions from Sect. 3.2 to specify the failure detectors from [6], and we use the definitions from Sect. 3.3, to specify their 'equivalent' fairness-based system models.

3.1 Asynchronous System Model

The asynchronous system model consists of a finite set of processes Π and a set of communication links that allow each process to send and receive messages from each other process in the system.

Global time. We posit the existence of a discrete global time base whose range of values is the natural numbers $\mathbb{N}$. Informally, global time simply counts the events that occur in the system; global time is not a measure of the real-time duration between two events. That is, the real-time duration that elapses 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.

Faults and fault patterns. A process can fail only by *crashing*, which happens when the process ceases execution without warning and never recovers. The processes that crash are said to be *faulty* and the processes that do not crash are said to be *correct* (or non-faulty). A *fault pattern* is a function F that returns

decides. However, Ω, in isolation, is not sufficient to ensure the safety property: no two processes decide differently. Therefore, given Ω, additional 'synchronism' is necessary to establish the safety property, and this has to be obtained from other sources. In the case of shared memory systems, this additional 'synchronism' is provided by read/write atomic registers; in the case of message passing systems, the additional 'synchronism' is provided either by restricting the fault environment to majority-correct ones or by assuming access to a stronger failure detector Σ. Failure detector Σ outputs a set of processes at each process. Any two sets (output at any times and by any processes) intersect, and eventually every set output at correct processes consists only of correct processes.

[9] The failure detector anti-Ω_k outputs, at each process and each time, a set of $n-k$ processes. Anti-Ω_k guarantees that there is a time after which some correct process is never output.

the set of crashed processes at any given time. That is, $F : \mathbb{N} \to 2^{\Pi}$; $F(t)$ denotes the set of processes that are crashed at time t. Since crashed processes never recover, $\forall t, t' \in \mathbb{N}, t < t' : F(t) \subseteq F(t')$. We define $faulty(F) = \cup_{\forall t \in \mathbb{N}} F(t)$ and $correct(F) = \Pi - faulty(F)$; that is, $faulty(F)$ denotes all the processes that crash in F and $correct(F)$ denotes all the processes that are correct in F. A process that has not crashed at time t is said to be *live*. We consider only fault patterns F in which at least one process is correct; that is, $correct(F) \neq \emptyset$; let the set of all such fault patterns be denoted $\mathcal{F}$.

Failure detectors. A failure detector [11] is a distributed oracle that can be queried for (potentially incorrect) information about crash faults in Π. Each process in Π is assumed to have access to a local failure detector module which outputs a subset of Π currently suspected as having crashed.

Informally, a *failure-detector history* describes the output of a failure detector during an execution. Formally, a failure-detector history H is a function that maps $\Pi \times \mathbb{N}$ to 2^{Π}; $H(p, t)$ is the set of processes output by a failure detector to process p at time t. Let $\mathcal{H}$ denote the set of all possible failure-detector histories.

A *failure detector* $\mathcal{D}$ is defined as a function $\mathcal{D} : \mathcal{F} \to 2^{\mathcal{H}} - \emptyset$; that is, $\mathcal{D}$ maps every fault pattern F to a non-empty set of failure detector histories. In other words, $\mathcal{D}(F)$ denotes the set of all possible histories that may be output by $\mathcal{D}$ when the fault pattern is F.

Note that it is not necessary for algorithms to have access to failure detectors. In such cases, we assume that the algorithms have access to a NULL failure detector which always outputs $\emptyset$.

Steps. Each process is modeled as a (possibly infinite) state machine. Certain states are identified as initial states. Each transition of the state machine — or *step* of the process — takes as input the current state of the process, a set consisting of zero or more messages from each other process (the "received" messages), and the output from the failure detector; it produces as output a new state for the process and a set of messages consisting of zero or more messages to each other process (the "sent" messages).[10]

It is important to note that the set of messages received by a process is not under the control of the process. In other words, a process cannot 'choose' whether or not to receive messages. Messages that are delivered to a process are stored in a receive buffer at the process. When the process takes the next atomic step, all the messages in the receive buffer are said to have been *received.*

Configuration. A configuration of the system consists of a state for each process and the set of all messages that have been sent but not yet received, called the in-transit messages (we assume each message can be uniquely identified).

Runs. A run is defined with respect to a set of processes Π, a fault pattern F, and a history H of a failure detector $\mathcal{D}$ (that is, $H \in \mathcal{D}(F)$). A *run* of the system is an infinite sequence of alternating configurations and steps of the form $\alpha = C_0 s_1 C_1 s_2 \ldots$. The sequence must satisfy the following properties:

- C_0 is an initial configuration (every process is in an initial state and no messages are in transit).
- For each $i \geq 1$, the global time i is associated with s_i. For convenience in notation, we denote the global time i associated with s_i by t_i.[11]
- For each $i \geq 1$, s_i must be *applicable* to C_{i-1}, meaning:
  - All the messages to be received during s_i are in transit in C_{i-1}.
  - The failure detector output that is used as input to the transition indicated by s_i is $H(p_i, t_i)$, where p_i is the process taking a step at s_i and t_i is the time associated with s_i.
  - The process executing s_i (say, p_i) is live at time t_i; that is, $p_i \notin F(t_i)$.
- For each $i \geq 1$, C_i is the result of applying step s_i to C_{i-1}: the state of the process executing s_i changes according to the transition function of the process, no other processes change state, the messages received during s_i are removed from the set of in-transit messages, and the messages sent during s_i are added to the set of in-transit messages.
- Every correct process takes an infinite number of steps. To model algorithms that terminate, a correct process can enter a final state S_f in a run so that subsequently the process takes only dummy steps (executes a no-op action) that do not send any messages and keep the process in the state S_f. A no-op

[10] A detailed discussion of the various definitions of an *atomic step* follows in Sect.8.

[11] A consequence of this property is that for all finite intervals of time, every process in Π executes only a finite number of steps. This property is ensures that "Zeno behavior" is prohibited in a valid run; that is, processes are not permitted to accelerate such that they execute an infinite number of steps in finite time.

action is enabled only when no other action is enabled at the process. Recall that other actions may become enabled when the process receives messages. Therefore, if a process takes a dummy step, then it implies that no other action was enabled at the process despite receiving a (possibly empty) set of messages.
– Every message that is sent from (say) process i to (say) process j is guaranteed to be received by i iff both i and j are non-faulty. This assumption implies the following: (a) process crashes can never partition the system, and (b) all the messages sent by i that are in transit when i crashes may be dropped. If process i crashes at time t while a set of messages M, sent by i, are in transit at time t, then messages in M may or may not be delivered.

3.2 Failure Detectors

As mentioned previously, failure detectors can be characterized by the restrictions on their histories for various fault patterns. Failure detectors are classified into various classes based on certain restrictions on their histories. These restrictions are specified by two abstract properties: *completeness* and *accuracy*. The original definition of failure detectors [11] considers two completeness properties *Weak Completeness* and *Strong Completeness*. However, [11] shows that under all-to-all communication, weak completeness can be transformed to strong completeness while preserving accuracy. Therefore, we consider only **strong completeness**, which states that *eventually every faulty process is permanently suspected by every correct process*; that is, a failure detector $\mathcal{D}$ satisfies strong completeness iff $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \exists t \in \mathbb{N} : \forall t' > t, \forall i \in faulty(F), \forall j \in correct(F) : i \in H(j,t')$.

There are four accuracy properties specified for the canonical failure detector classes in [11]: *Strong Accuracy*, *Weak Accuracy*, *Eventually Strong Accuracy*, and *Eventually Weak Accuracy*.

– **Strong Accuracy** states that *no process is suspected before it crashes*; that is, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \forall t \in \mathbb{N}, \forall i,j \in \Pi - F(t) :: i \notin H(j,t)$.
– **Weak Accuracy** states that *some correct process is never suspected*; that is, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \exists i \in correct(F), \forall t \in \mathbb{N}, \forall j \in \Pi - F(t) :: i \notin H(j,t)$.
– **Eventual Strong Accuracy** states that *correct processes are eventually never suspected by any correct process*; that is, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \forall i,j \in correct(F), \exists t \in \mathbb{N} : \forall t' > t : i \notin H(j,t')$.
– **Eventual Weak Accuracy** states that *some correct process eventually is never suspected by any correct process*; that is, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F), \exists i \in correct(F), \forall j \in correct(F), \exists t \in \mathbb{N} : \forall t' > t : i \notin H(j,t')$.

The completeness and accuracy properties stated above define four failure detector classes [11]:

– The *Perfect failure detector* ($\mathcal{P}$) satisfies strong completeness and strong accuracy.
– The *Strong failure detector* ($\mathcal{S}$) satisfies strong completeness and weak accuracy.
– The *Eventually Perfect failure detector* ($\diamond\mathcal{P}$) satisfies strong completeness and eventual strong accuracy.
– The *Eventually Strong failure detector* ($\diamond\mathcal{S}$) satisfies strong completeness and eventual weak accuracy.

Alternate Definitions. In order to facilitate an understanding of how these failure detectors encapsulate fairness, we propose alternate (but equivalent) definitions of these failure detector classes. These alternate definitions are based on the definition of a *distinguished* process. Informally, a process i is said to be *distinguished* if i is never suspected until it crashes, and after crashing, i is eventually suspected by all live processes and remains suspected forever thereafter. Similarly, a process i is said to be *eventually distinguished* if there is a time t (which may or may not be known) after which i is *distinguished*. In other words, an eventually distinguished process may be falsely suspected before (some potentially unknown) time t. Note that every *distinguished* process is also an *eventually distinguished* process where the time t is 0.

Formally, a process i is said to be *distinguished* with respect to a failure detector $\mathcal{D}$ if, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F)$, the following properties are satisfied:

– $\forall t \in \mathbb{N}, \forall j \in \Pi - F(t) :: i \notin F(t) \Rightarrow i \notin H(j,t)$.
– $\exists t \in \mathbb{N} :: (i \in F(t)) \Rightarrow (\forall t' \geq t, \forall q \in \Pi - F(t'), i \in H(q,t'))$

Similarly, a process i is said to be *eventually distinguished* with respect to a failure detector $\mathcal{D}$ if, $\forall F \in \mathcal{F}, \forall H \in \mathcal{D}(F)$, the following properties are satisfied:

- $\forall j \in correct(F), \exists t \in \mathbb{N} : \forall t' > t : i \notin F(t') \Rightarrow i \notin H(j, t')$.
- $\exists t \in \mathbb{N} :: (i \in F(t)) \Rightarrow (\forall t' \geq t, \forall q \in \Pi - F(t'), i \in H(q, t'))$

Based on this definition of a *distinguished* process (and the auxiliary definition of an *eventually distinguished* process), we redefine the four failure detectors classes as follows:

- $\mathcal{P}$ is a failure detector for which every process is distinguished.
- $\mathcal{S}$ is a failure detector for which some correct process is distinguished, and all faulty processes are eventually distinguished.
- $\Diamond\mathcal{P}$ is a failure detector for which every process is eventually distinguished.
- $\Diamond\mathcal{S}$ is a failure detector for which some correct process is eventually distinguished, and all faulty processes are eventually distinguished.

Note that in all the four failure detector classes, faulty processes are eventually distinguished. This corresponds to *strong completeness* [11] which states that there exists a time after which every crashed process is permanently suspected by all correct processes.

3.3 Fairness Constraints

We claim that Chandra-Toueg failure detectors encapsulate *fairness* guarantees of the underlying system. 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* [17]. A system is said to have a bound Φ on relative process speeds if the following holds. In each run α of the system, and the associated fault pattern F, for each process i, in each time interval of the form $[t_1, t_2]$ in which process i takes $\Phi + 1$ steps, all the processes not in $F(t_2)$ are guaranteed to take at least 1 step. Note that this fairness property is symmetric in the following sense. In a system where relative process speeds are bounded by Φ, let i and j be two processes. In any duration where process i takes $\Phi + 1$ steps, if process j is live during the entire duration, then j is guaranteed to take at least one step. This behavior holds even when the roles of i and j are reversed. That is, in any duration where j takes $\Phi + 1$ steps, if i is live during the entire duration, then i is guaranteed to take at least one step. However, it is possible to define computational fairness properties that are asymmetric.

We next give our definition of computational fairness. A process i is said to be *k-proc-fair* (where k is a non-negative integer) in a run α, if, for all processes $j \in \Pi$, in every segment of α in which j takes exactly $k + 1$ steps, either (1) i takes at least one step, or (2) i has crashed before the end of the segment. Similarly, a process i is said to be *eventually k-proc-fair* in α, if, for all processes $j \in \Pi$, there exists a (potentially unknown) time t_{gst} such that, in all execution segments of α that begin after t_{gst} in which j takes exactly $k + 1$ steps, either (1) i takes at least one step, or (2) i has crashed before the end of the segment. That is, i is *k-proc-fair* in α from time t_{gst} onwards.

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. This is an important distinction between computational fairness and bounded relative process speeds defined in [18, 17]. Bounded relative process speeds may be viewed as a special case where every process is (eventually) k-proc-fair.

Communicational Fairness. Constraining 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 run α, if, for all processes $j \in \Pi$, for each message m sent from i to j in α, during the segment of α that starts from the step in which m is sent, contains exactly d steps by j, and ends with a step by j, either (1) m is received by j, or (2) i has crashed before the end of the segment.

Similarly, a process i is said to be *eventually* d-*com-fair* (where d is a non-negative integer) in α if, there exists a (potentially unknown) time t_{gst} such that, for all processes $j \in \Pi$, for each message m sent from i to j in α after time t_{gst}, during the segment of α that starts from the step in which m is sent, contains exactly d steps by j, and ends with a step by j, either (1) m is received by j, or (2) i has crashed before the end of the segment. That is, i is d-com-fair from time t_{gst} onwards.

In traditional partially synchronous models [18, 17] the bounds on message delay are measured in real-time units and the de facto upper bound of one step per unit time on process speeds imposes an upper bound on the number of steps taken by both the sender and the recipient of the message. In contrast, we measure the bounds on communicational fairness as the number of steps taken by the recipient, and not the sender. The reason for such discrepancy is the following. 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

We present four partially synchronous system 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-fair and d-com-fair, for known k and d.
2. *Some Fair* ($\mathcal{SF}$) is an asynchronous system model where, in every run, some correct process is both k-proc-fair and d-com-fair, for known k and d.
3. *Eventually All Fair* ($\Diamond\mathcal{AF}$) is an asynchronous system model where, in every run, all the processes are both eventually k-proc-fair and eventually d-com-fair, for known k and d.
4. *Eventually Some Fair* ($\Diamond\mathcal{SF}$) is an asynchronous system model where, in every run, some correct process is both eventually k-proc-fair and eventually d-com-fair, for known k and d.

Next we describe the methodology used to prove our results.

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 $\mathcal{AF}$, $\mathcal{SF}$, $\Diamond\mathcal{AF}$, 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 called a *scheduler* (described in Sect. 5) that queries 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 from Chandra-Toueg Failure Detectors

In this section, we present a distributed scheduler that 'extracts' the fairness encapsulated by the Chandra-Toueg failure detectors. The scheduler is assumed to serve a distributed application; it determines the times at which the application modules at each process execute their respective steps such that appropriate fairness constraints on the relative ordering of steps and message receipts are maintained.

5.1 Interface Between Scheduler and Application

The scheduler interacts with the application through $executeAPP()$, $receiveAPP()$, and $sendAPP()$ interfaces (specified in Alg. 1.2). 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 through the $receiveAPP()$ interface. 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 takes a step, the scheduler delivers the messages in the receive buffer to the application through $receiveAPP()$. Note that the application does not have control over the invocation of $receiveAPP()$; this is performed by the scheduler to ensure that the application receives messages 'on time'.

The application sends messages via 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 at i then sends the messages to destination processes where they are stored in the receive buffers of the destination processes. These messages are then received by the respective recipient processes when the scheduler modules at the latter processes invoke $receiveAPP()$.

5.2 Fairness Guarantees Provided by the Scheduler

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 to the application being scheduled. For example, if the application module at (a live) process (say) i is guaranteed to be scheduled to take at least one step in all durations where other processes (that are live in that duration) have taken at least $k+1$ steps, then the scheduler is said to provide k-proc-fairness for process i. Similarly, while a message m is is in transit to process i, if the application module at i is guaranteed to be scheduled to take fewer than d steps and guaranteed to receive m within d steps, then the scheduler is said to provide d-com-fairness for process i.

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 permitted to take steps. Upon becoming *active*, the scheduler module permits the associated application module to execute a single enabled step; we assume that the application at each process always has some enabled step that it can take. After the application takes exactly one step, 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 scheduler provides the $\mathcal{AF}$ system model guarantees to the scheduled application.
- If the distributed scheduler uses $\mathcal{S}$, then the scheduler provides the $\mathcal{SF}$ system model guarantees to the scheduled application.
- If the distributed scheduler uses $\Diamond\mathcal{P}$, then the scheduler provides the $\Diamond\mathcal{AF}$ system model guarantees to the scheduled application.
- If the distributed scheduler uses $\Diamond\mathcal{S}$, then the scheduler provides the $\Diamond\mathcal{SF}$ system model guarantees to the scheduled application.

5.3 Algorithm Description

The algorithm in Alg. 1.1 and 1.2 implements a distributed scheduler with dynamic heights and permits. Alg. 1.1 shows the actions of the scheduler and Alg. 1.2 shows the interface between the scheduler and the scheduled application. The idea of dynamic heights (also called *priorities*) and permits (also called *forks*) is borrowed from the algorithms to solve the dining philosophers problem in [12] and [36]. Each process is assigned a static unique 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. Each process i also uses sequence numbers to tag its message requests and the sequence number is stored in the variable $s_i.\mathsf{seq}$. For each other 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 (or simply *token*) to request a permit from j is currently at i, (c) $s_i.\mathsf{ht}_j$ which stores the last received value of j's height (in permits and request messages), and (d) $s_i.\mathsf{maxAck}_j$ which stores the application-message request with latest sequence number for which i has received application messages from j.

Every pair of processes i and j share a permit and a token. Between every pair of processes, initially, permits are held by the higher-id process and the tokens are held by the lower-id process. All processes start in the *waiting* state. 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/deferred or sent based on the same condition mentioned previously.

Once a *waiting* process receives all shared permits from processes not suspected by the failure detector $\mathcal{D}$, the process becomes *active* in Action 4. When a process i becomes *active*, it solicits any messages for which the active process is the recipient by sending an application-message request to all the processes in the system. Each such message contains a unique sequence number $s_i.\mathsf{seq}$. Process i tracks the receipt of responses to its application-message requests by storing the largest sequence number for which application messages have been received in the variable $s_i.\mathsf{maxAck}_j$.

Application-message requests are received in Action 5. When a process j receives such a request from a process i with a sequence number *num*, process j sends the contents of its *local send buffer* for process i in response with the same sequence number *num*. Process i receives this response to its request in Action 6,

```
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 $i$
integer $s_i.\mathsf{seq} \leftarrow 0$        Generates a new sequence number to solicit messages from other processes
$\forall j \in \Pi - \{i\}$ :
  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 $s_i.\mathsf{maxAck}_j \leftarrow 0$        The highest sequence number among the messages received from $j$
  set $s_i.\mathsf{send\_buffer}_j \leftarrow \emptyset$        The send buffer through which application at $i$ sends messages to $j$
  set $s_i.\mathsf{receive\_buffer}_j \leftarrow \emptyset$        The receive buffer from which application at $i$ receives messages from $j$
--------------------------------------------------------------------------------
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 $\langle request, \mathsf{ht}\rangle$ 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 $\langle permit, \mathsf{ht}\rangle$ from $s_j\} \longrightarrow$        Action 3
10 :   $s_i.\mathsf{permit} \leftarrow true$
11 :   $s_i.\mathsf{ht}_j \leftarrow ht$
12 :   if $(s_i.\mathsf{req}_j \wedge (s_i.\mathsf{state} = waiting) \wedge ((ht > s_i.\mathsf{ht}) \vee ((ht = s_i.\mathsf{ht}) \wedge (i < j)))$        Send permit if $s_i$ is waiting
13 :       send $\langle permit, s_i.\mathsf{ht}\rangle$ to $s_j$; $s_i.\mathsf{permit}_j \leftarrow false$        and $s_j$ has higher priority
--------------------------------------------------------------------------------
14 : $\{(s_i.\mathsf{state} = waiting) \wedge (\forall j \notin \mathcal{D} \setminus \{i\} :: s_i.\mathsf{permit}_j)\} \longrightarrow$        Action 4 (Note: $\mathcal{D}$ is queried)
15 :   $s_i.\mathsf{state} \leftarrow active$        Active upon holding permits from trusted processes
16 :   increment $s_i.\mathsf{seq}$ by 1        Generate a new seq. no. to tag a request message
17 :   foreach $j$ in $\Pi - \{i\}$
18 :       send $\langle getMsg, s_i.\mathsf{seq}\rangle$ to $s_j$        Send a request for messages to all processes
--------------------------------------------------------------------------------
19 : $\{$upon receive $\langle getMsg, num\rangle$ from $s_j\} \longrightarrow$        Action 5
20 :   $S \leftarrow s_i.\mathsf{send\_buffer}_j$        Received a request for msgs
21 :   $s_i.\mathsf{send\_buffer}_j \leftarrow \emptyset$
22 :   send $\langle S, num\rangle$ to $s_j$        Send the contents of the local send buffer
--------------------------------------------------------------------------------
23 : $\{($upon receive $\langle S', num\rangle$ from $s_j)\} \longrightarrow$        Action 6
24 :   $s_i.\mathsf{receive\_buffer}_j \leftarrow s_i.\mathsf{receive\_buffer}_j \cup S'$        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 : $\{(s_i.\mathsf{state} = active) \wedge (\forall j \in \Pi - \{i\} :: ((s_i.\mathsf{maxAck}_j = s_i.\mathsf{seq}) \vee (j \in \mathcal{D})))\} \longrightarrow$        Action 7 (Note: $\mathcal{D}$ is queried)
27 :   $executeAPP()$        Execute an application step; $executeAPP()$ is specified in Alg. 1.2
28 :   $s_i.\mathsf{ht} \leftarrow \min(\forall j \in \Pi - \{i\} :: s_i.\mathsf{ht}_j, s_i.\mathsf{ht}) - 1$
29 :   $\forall j \in \Pi - \{i\}$ where $(s_i.\mathsf{permit}_j)$        Reduce height below all the neighbors whose heights are 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 application step
```

Alg. 1.1. Actions for scheduler at process i.

```
procedure $executeAPP()$
    execute an enabled application step where
        $receiveAPP()$ delivers messages to the application
        application step 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 at process i.

process i takes all the received messages and adds them its *local receive buffer*; it also updates $s_i.\mathsf{maxAck}_j$ if *num* is the largest sequence number for which i has received application messages from j.

If the active process i has received application messages for its latest application-message request from all the processes it doesn't suspect, then Action 7 is enabled at i. In Action 7, i invokes $executeAPP()$ to execute an application step before transiting to the *waiting* state. When the process executes an application step, the application is given all the messages in the *local receive buffer* via the $receiveAPP()$ interface described in Alg. 1.2, and the application step sends messages by invoking $sendAPP()$ described in Alg. 1.2 which simply adds the message to the *local send buffer*. These messages are not sent until an application-message request is received from the scheduler module at the recipient of these messages.

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

5.4 Proof of correctness

In this section we prove that the distributed scheduler in Alg. 1.1 satisfies the local progress and fairness properties specified in Sect. 5. For the purpose of the proof, consider an arbitrary run α of Alg. 1.1.

Lost request tokens or permits can prevent progress, while duplicated request tokens or permits can compromise fairness. First we prove that every pair of processes share a unique permit and a unique request token. We use the following notation to denote that a message of type y is in transit from process i to j: $M^{y}_{i \to j}$.

Lemma 1. *For all configurations in α, there exists exactly one request token between each pair of live processes; that is, for all pairs of processes (i, j), exactly one of the following four expressions is true: (1) $s_i.\mathsf{req}_j = true$, (2) $s_j.\mathsf{req}_i = true$, (3) $M^{request}_{i \to j} = true$, and (4) $M^{request}_{j \to i} = true$.*

Proof. For each pair of processes, the initialization code creates a unique request token at the lower-priority process. Since communication channels are reliable, this token is neither lost nor duplicated while in transit. Only Actions 1 and 2 can modify the token variables. No token is lost, because every token received is locally stored (Action 2), and no token is locally removed unless it is sent (Action 1). No token is duplicated, because every token sent is locally removed, and no absent token is ever sent (Action 1). Thus, token uniqueness is preserved. □

Lemma 2. *For all configurations in α, there exists exactly one permit between each pair of live processes; that is, for all pairs of processes (i, j), exactly one of the following four expressions is true: (1) $s_i.\mathsf{permit}_j = true$, (2) $s_j.\mathsf{permit}_i = true$, (3) $M^{permit}_{i \to j} = true$, and (4) $M^{permit}_{j \to i} = true$.*

Proof. For each pair of processes, the initialization code creates a unique permit at the higher-priority process. Since communication channels are reliable, this permit is neither lost nor duplicated while in transit. Only

Actions 2, 3, and 7 modify the permit variables. No permit is lost, because every permit received is locally stored (Action 3), and no permit is locally removed unless it is sent (Actions 2, 3, & 7). No permit is duplicated, because every permit sent is locally removed, and no absent permit is ever sent (Actions 2, 3, & 7). Thus, permit uniqueness is preserved. □

In order to prove local progress, we are required to show that every correct process is guaranteed to take application steps infinitely many times. This proof is established in two steps. Note that a process can execute its application action only when it is *active*. So, in the first step (Lemmas 3 and 4), we show that a correct process is *active* only for a finite duration. In the second step (Lemma 5 and Theorem 1), given that a correct process is *active* only for a finite time, we establish that every correct *waiting* process eventually becomes *active*. Since a correct process starts *waiting* when it stops being *active*, it follows that a correct process becomes *active* infinitely many times, and therefore takes application steps infinitely many times.

Lemma 3. *For all configurations in α, for all pairs of processes (i, j) where $i \neq j$, $s_i.\mathsf{maxAck}_j$ never exceeds $s_i.\mathsf{seq}$; that is, $\forall i, j \in \Pi : i \neq j : s_i.\mathsf{maxAck}_j \leq s_i.\mathsf{seq}$.*

Proof. Initially, $s_i.\mathsf{seq} = s_i.\mathsf{maxAck}_j = 0$, therefore the lemma is true initially. Note that the only action that changes the value of $s_i.\mathsf{seq}$ is Action 4, and Action 4 increments the value by 1. Therefore, if the lemma was true before i executed Action 4, then the lemma is true upon executing Action 4 as well.

Note that the only action that changes the value of $s_i.\mathsf{maxAck}_j$ is Action 6. If Action 6 increases the value of $s_i.\mathsf{maxAck}_j$, then the increased value *num* is received by i in a message $\langle msgSet', num\rangle$ from j. But note that j sends $\langle msgSet', num\rangle$ to i only upon receiving $\langle getMsg, num\rangle$ from i (Action 5). But in the message $\langle getMsg, num\rangle$ sent by i to j (at time t'), the value of *num* (in line 25, Action 6, Alg. 1.1) is $s_i.\mathsf{seq}$ at time t'. Inspection of the algorithm reveals that $s_i.\mathsf{seq}$ is non-decreasing. Therefore, the new $s_i.\mathsf{maxAck}_j$ is either the current or a previous value of $s_i.\mathsf{seq}$. Therefore, if the lemma was true before i executed Action 4, then the lemma is true upon executing Action 4 as well.

Thus, the lemma is true initially, and the lemma is true after executing any action that changes the values of $s_i.\mathsf{seq}$ and $s_i.\mathsf{MaxAck}_j$; thus proved. □

Now we are ready to show that all correct processes are *active* only for finite durations.

Lemma 4. *Let C be a configuration in α at time t in which a process i is active. Then in some configuration C' at time $t' > t$, either i is crashed or i is waiting.*

Proof. Given that process i is active in configuration C at time t, let the system be in configuration C'' at time $t'' \leq t$ such that i is active in all the configurations in the interval $[t'', t]$ and i is not active at time $t'' - 1$. In other words, process i becomes *active* in the step that results in configuration C'' at time t'' (in Action 4), and i remains *active* through time $t \geq t''$ in configuration C. If i is faulty, then i crashes at some time $t' > t$, thus satisfying the lemma.

However, if i is correct, then to prove the lemma we have to show that there exists a time after t at which i is *waiting*. Let the value of $s_i.\mathsf{seq}$ in C'' be *num*. From the code in Alg. 1.1, we see that the value of $s_i.\mathsf{seq}$ changes from *num* only in Action 4, which is enabled only when i is *waiting*. Therefore, the value of $s_i.\mathsf{seq}$ does not change from *num* until i transits from *active* to *waiting*. Also, from Action 4, we see that i sends the message $\langle getMsg, num\rangle$ to all other processes in the step that i takes immediately preceding C''. For each correct process j, j receives the message $\langle getMsg, num\rangle$ from i, executes Action 5, and sends $\langle msgSet, num\rangle$ to i. The message $\langle msgSet, num\rangle$ is eventually received by i (since i is still live) in Action 6 at time (say) $t_{rj} > t''$, and i sets the value of $s_i.\mathsf{maxAck}_j$ to *num*.

We know from Lemma 3 that $s_i.\mathsf{maxAck}_j$ is always at most $s_i.\mathsf{seq}$, and we know that $num = s_i.\mathsf{seq} = s_i.\mathsf{maxAck}_j$ at time t_{rj}. From the code in Alg. 1.1, we see that the value of $s_i.\mathsf{maxAck}_j$ is non-decreasing, and we have already established that the value of $s_i.\mathsf{seq}$ does not change until i transits from *active* to *waiting*. Therefore, from time t_{rj} until i transits to *waiting*, $s_i.\mathsf{maxAck}_j = s_i.\mathsf{seq}$.

For each faulty process j, one the following is true: (1) eventually $s_i.\mathsf{maxAck}_j = s_i.\mathsf{seq}$ and remains so until i transits to *waiting*, or (2) j crashes and by strong completeness, j is eventually and permanently suspected by the failure detector $\mathcal{D}$.

Therefore, eventually for all processes $j \in \Pi - \{i\}$, either $s_i.\mathsf{maxAck}_j = s_i.\mathsf{seq}$ or j is suspected by $\mathcal{D}$. That is, eventually, Action 7 is continuously enabled at i until Action 7 is executed, and after Action 7 is executed, i starts *waiting*. Thus, we showed that if a process i is *active* in configuration C at time t, then at some future configuration C' at time $t' > t$ either i is crashed or i is *waiting*. □

In order to prove progress, we need to show that every *waiting* process eventually becomes *active*. For this purpose, we introduce some definitions to construct a metric function on configurations of α. First, we measure the priority distance between any two processes i and j in a configuration as:

$$dist(i,j) = \begin{cases} 0, & \text{if } (s_i.\mathsf{ht} < s_j.\mathsf{ht}) \\ s_i.\mathsf{ht} - s_j.\mathsf{ht}. & \text{if } ((s_i.\mathsf{ht} \geq s_j.\mathsf{ht}) \\ & \quad \wedge (i < j)) \\ s_i.\mathsf{ht} - s_j.\mathsf{ht} + 1, & \text{if } ((s_i.\mathsf{ht} \geq s_j.\mathsf{ht}) \\ & \quad \wedge (i > j)) \end{cases}$$

For any pair of processes i and j, in some configuration where j is *waiting*, suppose that $dist(i,j) = d$. While j remains *waiting*, $s_j.\mathsf{ht}$ remains unchanged. Also, recall from Action 7 that each process reduces its height (below all the processes whose shared permits it holds) when exiting the *active* state. Consequently, d is an upper bound on the maximum number of times that process i can overtake process j and become *active* before either j becomes *active* or $s_i.\mathsf{ht} < s_j.\mathsf{ht}$. Now we define a metric function $M : \Pi \rightarrow \mathbb{N}$ for each process $j \in \Pi$ as follows:

$$M(j) = \sum_{\forall i \in \Pi : i \neq j} dist(i,j)$$

Note that M is bounded below by 0, and that $M(j) = 0$ iff j currently has the highest priority value among all processes in Π. In general, the value of $M(j)$ depends only on processes that currently have a higher priority than j. This is because $dist(i,j) = 0$ for any process i with lower height than j or equal height as j but lower process id. If $M(j) = b$, then b is an upper bound on how many times any higher-priority process can become *active* before either j becomes *active* or j is the process with highest priority.

Also note that the metric value of each process in a given configuration is unique: $(i \neq j) \Rightarrow M(i) \neq M(j)$. Moreover, $M(i) < M(j) \Leftrightarrow ((s_i.\mathsf{ht} < s_j.\mathsf{ht}) \vee ((s_i.\mathsf{ht} = s_j.\mathsf{ht}) \wedge (i < j))$. These properties follow from the fact that priorities are totally ordered.

Finally, the metric value $M(j)$ never increases while process j is *waiting*. $M(j)$ can only increase by reducing the height $s_j.\mathsf{ht}$ in Action 7 while exiting the *active* state. Importantly, if j is the process with the largest height in the system, this change in relative priority actually causes the metric values of all other processes to decrease.

We now state and prove the following helper lemma for progress:

Lemma 5. *Let C be any configuration in α with at least one live* waiting *process. Let j be the live* waiting *process in C with minimal metric. Then there is a later configuration C' in α such that: (1) j is active in C', or (2) j is crashed in C', or (3) some other process i is live and* waiting *and $M(i) < M(j)$ in C'.*

Proof. Assume in contradiction that in every configuration after C, j is live and *waiting* and has the minimal metric. We will show that eventually j is *active*, a contradiction.

Let C'' be a configuration after C in α in which all faulty processes have crashed and by strong completeness of $\mathcal{D}$, all such crashed processes are permanently suspected. After C'', j only needs to collect permits from correct processes. We show that j succeeds in collecting and keeping all these permits, and thus, j will become *active*.

Let i be any correct process other than j. First we show that j will not lose the permit it holds with i. By hypothesis, j is *waiting* and has higher priority than any correct process from configuration C onwards (recall that $M(j)$ never increases while j is *waiting*; hence, j will continue to be the highest priority process until it becomes *active*), so any request token received by j in Action 2 will be deferred. Note that it is possible for j to receive an 'old' request token from i which has higher priority value, thereby causing j to give up its shared permit. However, j will send the request token to i in Action 1 right after sending the permit, and this time i will have to return the permit to j because j has higher priority. Thereafter, eventually, j defers the request token from i until j becomes *active*.

Now we show that j will eventually acquire the permit shared with i. By Lemma 2, j shares a unique request token with i. All permits that were in transit to j when j started *waiting* are delivered in finite time. For any missing permits, if j holds the request token, then j will eventually send the corresponding token.

However, if j has neither the request token nor the shared permit upon transiting to *waiting*, then eventually the shared permit and the request token are either at i or at j. We now show that eventually j receives either the request token or the shared permit. For the purposes of contradiction, suppose eventually i holds both the permit and the request token permanently. If i is *active*, then i eventually starts *waiting* (by Lemma 4) and sends the permit to j in Action 5. If i is *waiting*, then depending on the order in which the request token and the permit arrived at i, process i executes either Action 2 or Action 3. Since priorities are non-increasing, the priority encoded in the token and the permit received by i must be at least as high as j's current priority. We have already established that j has the highest priority in the system. Therefore, in both Action 2 and Action 3, i sends the shared permit to j.

If j (eventually) receives the request token, then j sends this request token to i in Action 1. Recall that by the hypothesis, j has higher priority than i; consequently, this permit request must be honored unless i is currently *active*. In the latter case, we know from Lemma 4 that i eventually exits to be *waiting*; therefore, the requested permit will be sent when i starts *waiting* in Action 5.

Thus, we conclude that if j remains *waiting* indefinitely, then j eventually suspects each faulty process and eventually holds the shared permit with each correct process. By Line 14, the guard on Action 4 is enabled. So j becomes *active*. □

Thus, we see that a waiting correct process with the minimal metric eventually either becomes active or no longer has the minimal metric in the system. Since the set of processes in the system is finite and fixed, we conclude that eventually some waiting correct process (say) i with minimal metric becomes active and takes an application step. As a result, i reduces its priority, and consequently, the metric values of all other correct processes in the system decrease. Furthermore, when i transitions to waiting (again), it no longer has the minimal metric in the system (assuming there are other correct processes in the system); that is, some other correct process in the system has the minimal metric. Thus, by inductively applying this argument to all correct processes, we see that every correct process takes infinitely many steps. We prove this result in Theorem 1.

Theorem 1. *Algorithm 1.1 satisfies local progress: every correct process takes infinitely many application steps.*

Proof. Note that to prove the theorem, it is sufficient to prove the following claim: For every k, every configuration C of α, and every correct process j, if $M(j) = k$ in C, then there is a later configuration in which j is *active*. We prove this by a complete (strong) induction on metric values.

Base Case: $k = 0$. Suppose $M(j) = 0$ in configuration C. Since 0 is the smallest possible value that the metric can have and j is correct, Lemma 5 implies that in some subsequent configuration C', either j is *active* or there is another live *waiting* process i whose metric is smaller than j's metric in C'.

However, since j's metric can never increase while j is *waiting*, and it is not possible for a process to have a metric less than 0, no such live *waiting* process i exists. So, j eventually becomes *active*.

Inductive Case: $k > 0$. Suppose for every $k' < k$, every configuration C of α, and every correct process j, if $M(j) = k'$ in C, then there is a later configuration in which j is *active*. We must show that for every configuration C and every correct process j, if $M(j) = k$ in C, then there is a later configuration in which j is *active*.

Let C be a configuration, and let j be a correct *waiting* process in C with $M(j) = k$. Suppose that k is the minimal metric value among all correct *waiting* processes in C. Then Lemma 5 applies to j, so we conclude that j eventually becomes *active*, or some correct process i with $M(i) < M(j)$ starts *waiting*. Alternatively, suppose that k is not the minimal metric value among all correct *waiting* processes in C. Then some (other) correct *waiting* process i with $M(i) < k$ already exists. Either way, we conclude that j eventually becomes *active* or the inductive hypothesis applies to some correct *waiting* process i with $M(i) < k$. In the latter case, process i becomes *active*. By Lemma 4, i eventually exits the *active* state by executing Action 5, which thereby lowers the height $s_i.\mathsf{ht}$ and decreases $dist(i, j)$ by at least 1. Recall that while j remains *waiting*, $M(j)$ does not increase. Thus, any decrease in $dist(i, j)$ will cause the metric value of $M(j)$ to become less than k. Since j is now a correct *waiting* process with $M(j) < k$, the inductive hypothesis applies directly to j. Thus, we conclude that j eventually becomes *active*.

By Lemma 4 and Action 7, we know that every time j becomes *active*, it executes an application step, and j eventually exits. Upon exiting j starts *waiting* again. Thus, we show that Alg. 1.1 satisfies local progress by complete induction. □

To establish the proof for computational fairness, we make use of the notion of *distinguished* processes. Recall that a distinguished process is never suspected until it crashes and is suspected forever thereafter. An informal argument for computational fairness of distinguished processes is as follows. Given a distinguished process i, no process in the system suspects i until i crashes. Therefore, when any other process j becomes active while i is live, j is guaranteed to hold the permit it shares with i. Therefore, when j transits back to waiting, j is guaranteed to reduce its height below i's height. Since i is not suspected until i crashes, as long as i does not crash, i is guaranteed to become active before j. We formalize this argument in Theorem 2 and its proof.

Theorem 2. *In Alg. 1.1, each distinguished (respectively, eventually distinguished) process is* 2*-proc-fair (respectively, eventually* 2*-proc-fair).*[12]

Proof. Consider an arbitrary run α and let i be any process that is eventually distinguished in α. Let the earliest time after which i is distinguished be t_i. That is, from time t_i onwards, if i is live at time $\hat{t} \geq t_i$, then i is trusted by the failure detector (at all live processes) at time $\hat{t}$, and on the other hand, if i is crashed at time $\hat{t} \geq t_i$, then for some $\tilde{t} \geq \hat{t}$, i is suspected by the failure detector (at all live processes) in the interval $[\tilde{t}, \infty)$. We must show that for all j, in every interval starting after t_i in which j takes three application steps, either i takes at least one application step or i is crashed.

Consider a process $j \neq i$ that takes three application steps after t_i, say at times t, t', and t'' and suppose that i is live through t''. We must show that i takes an application step at least once between t and t''. Since i is a distinguished process from time t_i onwards, the failure detector $\mathcal{D}$ at j never suspects i between t_i and t''. Therefore, j holds the shared permit between i and j at times t, t', and t''.

At time t, let the height of i be ht_i and the height of j be ht_j. There are two cases to consider here:

Case 1. Let $ht_j < ht_i$. Note that j holds the permit it shares with i. Eventually, j sends the permit to i in Action 7, and j includes its height ht_j in the permit. When i receives this permit $ht_j < ht_i$; therefore, i does not relinquish the permit until i becomes *active*. Since j becomes *active* at time t' (and takes an application step) and j does not suspect i, j holds the shared permit at time t'. Since i sends the permit only after i becomes *active*, i is guaranteed to become *active* before time t' (and take its application step); thus the theorem is satisfied.

Case 2. Let $ht_j > ht_i$. Note that j holds the shared permit. Eventually, j sends the permit to i in Action 7. We know that j takes an application step at time t' and therefore is *active* at time t'. Therefore, j must hold the permit it shares with i at time t'. That is, i sends the permit to j in the interval (t, t').

Note that i sends the permit to j only in Actions 2 and 7. If i executes Action 7 in (t, t'), then i was *active* in the interval (t, t') and the theorem is satisfied. On the other hand, if i sends the permit to j in Action 2, then i is not *active* in the interval (t, t'). Also, i includes its height ht_i in the permit. Therefore, when j is *active* at time t', the value of $s_j.\mathsf{ht}_i$ is ht_i. When j transits to *waiting*, it reduces its height to $ht'_j < ht_i$ and includes this height in the permit sent to i. We know that j takes an application step again at time t'', and so j is *active* again at time t''. Therefore, j must hold the permit it shares with i at time t''. That is, i sends the permit to j in the interval (t', t''). However, since $ht'_j < ht_i$, i does not send the permit to j in Action

[12] In the special case where all the processes are (eventually) distinguished, then a careful analysis shows that all correct distinguished processes are, in fact, (eventually) 1-proc-fair. We omit this here because, for the purposes of our results, it is sufficient to show that (eventually) distinguished processes are (eventually) 2-proc-fair.

2 in the interval (t', t''). The only other action in which i sends the permit to j is Action 7; that is, i must have been *active* at some time in the interval (t', t''), and hence, i must have taken an application step in the interval (t', t'').

In other words, i is eventually 2-proc-fair. However, if $t_i = 0$, then i is 2-proc-fair in the run α; that is, if i is a distinguished process, then i is 2-proc-fair. Thus, we have shown that every distinguished (respectively, eventually distinguished) process is 2-proc-fair (respectively, eventually 2-proc-fair). □

We use the notion of *distinguished* processes to establish communicational fairness as well. The intuitive argument for communicational fairness is as follows. Given a distinguished process i, no process in the system suspects i until i crashes. Therefore, while (1) an application message m sent by i to another process j is in transit and (2) i is live, j waits for $s_j.\mathsf{maxAck}_i$ to be equal to $s_j.\mathsf{seq}$ before executing an application step. However, $s_j.\mathsf{maxAck}_i$ equals $s_j.\mathsf{seq}$ only when j receives the set of application messages from i that were in transit before j became active. Note that m is one such message. Therefore, j takes no more than one application step while m is in transit. We formalize the this argument in Theorem 3 and its proof.

Theorem 3. *In Alg. 1.1, each distinguished (respectively, eventually distinguished) process is* 1*-com-fair (respectively, eventually* 1*-com-fair).*

Proof. Consider an arbitrary run α and let i be any process that is eventually distinguished in α starting at some time t_i. We show that for all j and all application messages m sent from i and received by j after t_i, during the time m is in transit, either j takes at most one step or i is crashed.

Consider a process $j \neq i$ to which i sends an application message m at some time t after t_i. This message is sent by the application when i is *active* and invokes $sendAPP(m, j)$, which causes m to be added to $s_i.\mathsf{send\_buffer}_j$ (the message is actually sent by the scheduler later during the execution). By the assumption that m is received by j, we know that j takes at least one application step after t. Again, note that j executes its application step only when j is *active*. Let $t' > t$ be the earliest time after t that j becomes *active* (by executing Action 4). In the *active* session that starts at time t', j sends $\langle getMsg, s_j.\mathsf{seq}\rangle$ to i in Action 4.

From Lemma 3 we know that $s_j.\mathsf{maxAck}_i \leq s_j.\mathsf{seq}$ before j executes Action 4. But Action 4 increments $s_j.\mathsf{seq}$, therefore, after j executes Action 4, $s_j.\mathsf{maxAck}_i < s_j.\mathsf{seq}$

From Lemma 4, we know that j eventually stops being *active*. Since i is a correct eventually distinguished process, we also know that j does not suspect i. Therefore, if j eventually exits the *active* state by executing Action 7, then eventually $s_j.\mathsf{maxAck}_i = s_j.\mathsf{seq}$.

The above two arguments imply that while j is *active*, the value of $s_j.\mathsf{maxAck}_i$ is updated to $s_j.\mathsf{seq}$. However, the only action that updates $s_j.\mathsf{maxAck}_i$ is Action 6, and Action 6 is executes only upon receiving $\langle msgSet', num\rangle$.

The message $\langle getMsg, s_j.\mathsf{seq}\rangle$ sent by j in Action 4 is eventually received by i in Action 5 (or i is crashed, in which case the lemma is satisfied). Action 5 empties $s_i.\mathsf{send\_buffer}_j$ and sends the messages in the buffer to j. But note that the message m was in $s_i.\mathsf{send\_buffer}_j$ before Action 5 is executed. Therefore, message m is sent to j in the message $\langle msgSet', s_j.\mathsf{seq}\rangle$. This message is eventually received by j in Action 6, and Action 6 puts message m into the receive buffer $s_j.\mathsf{receive\_buffer}_i$ and updates $s_j.\mathsf{maxAck}_i$ to $s_i.\mathsf{seq}$.

Therefore, when j executes Action 7, m is already in $s_j.\mathsf{receive\_buffer}_i$. Note that in Action 7, j executes an application step which will receive all the messages in $s_j.\mathsf{receive\_buffer}_i$ (from $receiveAPP()$ in Alg. 1.2). That is, j takes no more than one step after m is sent and before m is received. Therefore, j is 1-com-fair with respect to i for all processes j from time t_i.

In other words, i is eventually 1-com-fair. In addition, if $t_i = 0$, then i is 1-com-fair in the run α; that is, if i is a distinguished process, then i is 1-com-fair.

Thus, we have shown that every (eventually) distinguished process is (eventually) 1-com-fair. □

By substituting the failure detector $\mathcal{D}$ in Alg. 1.1 with $\mathcal{P}$, $\Diamond\mathcal{P}$, $\mathcal{S}$, and $\Diamond\mathcal{S}$, and applying Theorems 2 and 3, we get the following corollaries.

Corollary 1. *If the failure detector $\mathcal{D}$ in Alg. 1.1 is the perfect failure detector $\mathcal{P}$, then the action system described in Alg. 1.1 provides the All Fair ($\mathcal{AF}$) system model guarantees to the scheduled application.*

Corollary 2. *If the failure detector $\mathcal{D}$ in Alg. 1.1 is the eventually perfect failure detector $\diamond\mathcal{P}$, then the action system described in Alg. 1.1 provides the Eventually All Fair ($\diamond\mathcal{AF}$) system model guarantees to the scheduled application.*

Corollary 3. *If the failure detector $\mathcal{D}$ in Alg. 1.1 is the strong failure detector $\mathcal{S}$, then the action system described in Alg. 1.1 provides the Some Fair ($\mathcal{SF}$) system model guarantees to the scheduled application.*

Corollary 4. *If the failure detector $\mathcal{D}$ in Alg. 1.1 is the eventually strong failure detector $\diamond\mathcal{S}$, then the action system described in Alg. 1.1 provides the Eventually Some Fair ($\diamond\mathcal{SF}$) system model guarantees to the scheduled application.*

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

In this section we show that the system models $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, and $\diamond\mathcal{SF}$ are sufficient to implement the failure detectors $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$, 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 in Alg. 1.3 implements a failure detector under the fairness-based system models described in Sect. 3.4. The failure detector implemented by Alg. 1.3 is determined by the fairness guarantees of the underlying system. Specifically, the algorithm implements $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, or $\diamond\mathcal{S}$, if the underlying system model is $\mathcal{AF}$, $\mathcal{SF}$, $\diamond\mathcal{AF}$, or $\diamond\mathcal{SF}$, respectively.

6.1 Algorithm Description

In Alg. 1.3, the failure detector module at process i maintains a variable $\mathsf{timerValue}_j$ for each process j in the system which counts down from $k + d$ to 0, where, in the various system models described in Sect. 3.4, the bounds on fairness are specified by the existence of k-proc-fair and d-com-fair processes. The value of $\mathsf{timerValue}_j$ is decremented by 1 in each step (line 10). In each step process i receives zero or more messages from all other processes (line 2) and sends a heartbeat to each process j in the system (line 4). If i receives a heartbeat from j, then i deletes j from the set $\mathsf{suspectList}$ (line 6) and resets the timer for j to $k + d$ (line 7). If $\mathsf{timerValue}_j$ is decremented to 0, then i adds j to $\mathsf{suspectList}$ (line 9). Here we assume that the failure detector module output is $\mathsf{suspectList}$. Hence, when i adds j to $\mathsf{suspectList}$, i is said to suspect j as having crashed; when i deletes j from $\mathsf{suspectList}$, i is said to trust j.

6.2 Proof of Correctness

We now show that the action system in Alg. 1.3 satisfies strong completeness and different accuracy properties depending on the underlying system model. For the purpose of the proofs, we consider an arbitrary run α of Alg. 1.3.

Theorem 4. *Alg. 1.3 satisfies* strong completeness*; that is, there exists a time after which every crashed process is permanently suspected.*

Proof. In α, processes send heartbeats in every step. If a process i crashes at time t in α, i stops taking steps after t, and so stops sending heartbeats. Eventually, all the (finite number of) heartbeats sent by i are delivered. Let the last such delivery be at time $t' \geq t$. Inspection of the code reveals that the maximum value of $\mathsf{timerValue}_i$ at a process j at time t' is $k + d$. Thereafter, in every step executed by j after time t', $\mathsf{timerValue}_i$ is decremented (if $\mathsf{timerValue}_i$ is not already 0) until j receives another heartbeat from i. Process j resets $\mathsf{timerValue}_i$ to $k + d$ only upon receiving a heartbeat from i. Since we have established that no such heartbeat are received by j after t', it follows that in at most $k + d + 1$ steps, $\mathsf{timerValue}_i$ is decremented to 0 at all processes j, and so j starts suspecting i (in line 9). Since j does not receive any more heartbeats from i, j suspects i permanently. $\square$

```
constant timeOut ← $k + d$
set suspectList ← $\emptyset$
$\forall j \in \Pi - \{i\}$ :
  integer timerValue$_j$ ← timeOut
--------------------------------------------------------------------------------
1 :  $\{true\} \longrightarrow$                                                  Action 1
2 :      receive $\langle msgSet \rangle$                  Receives zero or more messages from each process
3 :      $\forall j \in \Pi - \{i\}$ do
4 :        send $\langle HB \rangle$ to $j$                                 Send a heartbeat to each process
5 :        if $(\langle HB, j \rangle \in msgSet)$
6 :          suspectList ← suspectList $- \{j\}$                       Trust upon receiving a heartbeat
7 :          timerValue$_j$ ← timeOut                                                   Reset timer
8 :        if (timerValue$_j = 0$)
9 :          suspectList ← suspectList $\cup \{j\}$                            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-fair and d-com-fair

We prove accuracy properties in two steps. In the first step (Lemma 6), we show that a correct process is trusted infinitely often; that is, if a correct process j trusts a correct process i at time t, then there exists a time $t' > t$ such that j trusts i at time t'. Note that this permits j to (falsely) suspect i in the open interval (t, t'). In the second step (Lemma 7), we show that if a process i is trusted after it is k-proc-fair and d-com-fair, then i will be continuously trusted until i crashes. Lemmas 6 and 7 are used to prove the various accuracy properties satisfied by Alg. 1.3, depending on the underlying system model.

Lemma 6. *In a run α of Alg. 1.3, if process i is correct, then every correct process j trusts i infinitely often; that is, $\forall t \in \mathbb{N}$, there exists a time $t' > t$ such that j trusts i at time t'.*

Proof. Consider a run α of Alg. 1.3 where a process i is correct. From lines 5–6 of Alg. 1.3 we know that a correct process $j \neq i$ trusts process i upon receiving a message from i. We also know that i sends heartbeats to all processes in every step (Alg. 1.3). Hence, if i is correct, then i takes steps infinitely often, and sends heartbeats infinitely often. Reliable communication guarantees that no heartbeat is lost. Therefore, all correct processes receive heartbeats from i infinitely often, and hence, execute lines 5–6 of Alg. 1.3 infinitely often. Therefore, all correct processes trust i infinitely often. □

Lemma 7. *In a run α of Alg. 1.3, if process i becomes k-proc-fair and d-com-fair from time t, and the value of* timerValue$_i$ *is $k + d$ at a process $j \neq i$ at time $t' \geq t$, then from time t' onwards, i is never suspected by j until i crashes.*

Proof. Consider a run α of Alg. 1.3. Let i become k-proc-fair and d-com-fair in α from time t. Let the value of timerValue$_i$ be $k+d$ at a process $j \neq i$ at time $t' \geq t$. We know that the value of timerValue$_i$ is decremented by 1 in each step until j receives a message from i. We now show that j is guaranteed to receive a message from i before timerValue$_i$ is decremented to 0.

Note that i sends a heartbeat to j in each action that i executes. Given that i is k-proc-fair and d-com-fair, we know that i will send at least one heartbeat to j before j has taken $k+1$ steps after t', and this heartbeat is received by j before j has taken $k + d + 1$ steps after t'. Recall that at time t', the value of timerValue$_i$ is $k + d$ and is decremented by 1 at every step taken by j. However, j receives at least one heartbeat from i within $k + d$ steps, and so the value of timerValue$_i$ is reset to $k + d$ (in line 7) before it reaches 0.

Note that for j to start suspecting i, timerValue$_i$ must be 0, and we have shown that if j starts trusting i, then the value of timerValue$_i$ is reset to $k + d$ (in line 7) before it reaches 0. Therefore, from time t onwards, i is never suspected by j until i crashes. □

Theorem 5. *If Alg. 1.3 is executed on the $\mathcal{AF}$ system model, Alg. 1.3 implements the perfect failure detector $\mathcal{P}$.*

Proof. The $\mathcal{AF}$ system model guarantees that all processes are k-proc-fair and d-com-fair from time $t = 0$. Also at time $t = 0$, at each process j, the value of $\mathsf{timerValue}_i = k + d$ for every other processes i in the system. Applying Lemma 7 with $t = t' = 0$, we know that i is never suspected by j until i crashes. Since i and j are arbitrary processes in the system, it follows that no process is suspected before it crashes. This, in conjunction with Theorem 4 shows that every process is distinguished; that is, Alg. 1.3 implements the perfect failure detector $\mathcal{P}$. □

Theorem 6. *If the action system in Alg. 1.3 is executed on the $\diamond\mathcal{AF}$ system model, Alg. 1.3 implements the eventually perfect failure detector $\diamond\mathcal{P}$.*

Proof. Consider a pair of correct processes i and j. Recall that the $\diamond\mathcal{AF}$ system model guarantees that i is k-proc-fair and d-com-fair from some (unknown) time t. From Lemma 6 we know that j trusts i infinitely often, which implies that the value of $\mathsf{timerValue}_i$ at j is $k + d$ infinitely often. Applying Lemma 7 we know that eventually j never suspects i. On the other hand, if i is faulty and crashes in finite time, then from Theorem 4 we know that eventually j always suspects i. In other words, i is eventually distinguished. Since i is an arbitrary process in the system, it follows that all the processes are eventually distinguished. That is, Alg. 1.3 implements the eventually perfect failure detector $\diamond\mathcal{P}$. □

Theorem 7. *If the action system in Alg. 1.3 is executed on the $\mathcal{SF}$ system model, Alg. 1.3 implements the strong failure detector $\mathcal{S}$.*

Proof. Recall that the $\mathcal{SF}$ system model guarantees that some correct process i is k-proc-fair and d-com-fair from time $t = 0$. Also at time $t = 0$, at each process j, the value of $\mathsf{timerValue}_i = k + d$. Applying Lemma 7 with $t = t' = 0$, we know that i is never suspected by j. This, in conjunction with Theorem 4 shows that some correct process is distinguished and all faulty processes are eventually distinguished; that is, Alg. 1.3 implements the strong failure detector $\mathcal{S}$. □

Theorem 8. *If the action system in Alg. 1.3 is executed on the $\diamond\mathcal{SF}$ system model, Alg. 1.3 implements the eventually strong failure detector $\diamond\mathcal{S}$.*

Proof. Recall that the $\diamond\mathcal{SF}$ system model guarantees that eventually some correct process i is k-proc-fair and d-com-fair. Let j be a correct process. From Lemma 6 we know that j trusts i infinitely often, which implies that the value of $\mathsf{timerValue}_j$ at i is $k+d$ infinitely often. Applying Lemma 7 we know that eventually j never suspects i. This, in conjunction with Theorem 4 shows that some correct process is distinguished and all faulty processes are eventually distinguished; that is, Alg. 1.3 implements the strong failure detector $\mathcal{S}$. □

7 Failure Detectors from the Extended Chandra-Toueg Hierarchy

We have shown how failure detectors in the Chandra-Toueg hierarchy encapsulate fairness constraints. Now we consider failure detector oracles from the extended Chandra-Toueg hierarchy which consists of all the failure detectors whose output is a subset of the processes in the system. Specifically, we consider the $\mathcal{G}_*$ family of failure detectors from [6].

The $\mathcal{G}_*$ family of failure detectors in [6] output a set of trusted processes (instead of suspected processes), and each member of this family is specified by a parameter c and denoted $\mathcal{G}_c$. The $\mathcal{G}_c$ failure detector satisfies strong completeness (specified in Sect. 3.2) and *c-Eventual Trust*, defined as follows [6].

c-Eventual Trust. In every run, there exists a set Π' consisting of at least c correct processes, such that there exists a time τ after which the failure detector output of all correct processes is a set of correct processes and a superset of Π'. Note that the output of the failure detector may continually change as long as, eventually and permanently, each output is a superset of Π'.

Clearly, the $\mathcal{G}_c$ failure-detector definition is valid only in fault environments containing at least c correct processes. Therefore, for the rest of this section, we only consider fault patterns that contain at least c correct processes.

Using the definition of a distinguished process from Sect. 3.2, we see that for a given (fixed) c, the $\mathcal{G}_c$ failure detector can be redefined as follows:

– For a given c, $\mathcal{G}_c$ is a failure detector for which at least c correct processes are eventually distinguished, and all faulty processes are eventually distinguished.

Consider the following fairness-based partially synchronous model: *Eventually c-Fair* ($\Diamond c$-$\mathcal{F}$) is an asynchronous system model where, for each run, there exists a (potentially unknown) time after which at least c correct processes and all faulty processes are both k-proc-fair and d-com-fair, for known k and d.

We use the constructions from Alg. 1.1 and 1.3 to show that the $\mathcal{G}_c$ failure detector encapsulates the *Eventually c-Fair* system model.

7.1 Extracting Fairness from the $\mathcal{G}_c$ Failure Detector

Consider an arbitrary run α of Alg.1.1 in an asynchronous system augmented with a $\mathcal{G}_c$ failure detector for an arbitrary, but fixed, c. From Theorem 2, we know that every eventually distinguished process is eventually 2-proc-fair, and from Theorem 3, we know that every eventually distinguished process is eventually 1-com-fair. Since $\mathcal{G}_c$ guarantees that at least c correct processes are eventually distinguished, and all faulty processes are eventually distinguished, we have the following corollary.

Corollary 5. *If the failure detector $\mathcal{D}$ in Alg. 1.1 is the $\mathcal{G}_c$ failure detector (for a given c), then the action system described in Alg. 1.1 provides the Eventually c-Fair ($\Diamond c$-$\mathcal{F}$) system model guarantees to the scheduled application.*

7.2 Extracting $\mathcal{G}_c$ from the Eventually c-Fair System Model

Consider an arbitrary run α of Alg. 1.3 in an Eventually c-Fair ($\Diamond c$-$\mathcal{F}$) system model for an arbitrary, but fixed, c. From Theorem 4, we know that Alg. 1.3 eventually and permanently suspects crashed processes.

Let Π' denote a set of c correct processes that are guaranteed to be eventually k-proc-fair and d-com-fair by the $\Diamond c$-$\mathcal{F}$ system model. From Lemma 6, we know that each process $j \in \Pi'$ is trusted by each correct process i infinitely often, which implies that the value of $\mathsf{timerValue}_j$ at i is $k + d$ infinitely often. Applying Lemma 7, we know that eventually i never suspects j. That is, every correct process eventually never suspects processes in Π'. Therefore, eventually and permanently, the outputs of Alg. 1.3 at each correct process trusts only correct processes and is a superset of Π'. Thus, we have the following corollary.

Corollary 6. *If the action system in Alg. 1.3 is executed on the $\Diamond c$-$\mathcal{F}$ system model, Alg. 1.3 implements the $\mathcal{G}_c$ failure detector.*

8 Discussion

Complete Synchrony and $\mathcal{P}$. It was first noted in [13] that there exist time-free problems that are solvable in synchronous systems, but are unsolvable with $\mathcal{P}$. This indicates 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. Given that $\Diamond\mathcal{S}$ is the weakest failure detector to solve consensus in asynchronous systems with a majority of correct processes [10], 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 observation suggests that there is a weaker system model which can implement $\Diamond\mathcal{S}$ in majority-correct environments, but not in all environments.

Real-Time Bounds and Failure Detectors. Although our results argue that failure detectors are better understood as bounds on fairness and not real time, they do not discount the real-time bounds that empirical systems incidentally satisfy. The real-time bounds become useful when considering the performance or the Quality of Service (QoS) [14] provided by these oracles. In other words, our results provide a separation of concerns between the correctness and performance of oracles with respect to the temporal properties of the distributed systems. Specifically, our work shows that correctness of oracles can be determined and understood exclusively through the fairness constraints of the system, and once correctness has been established, the performance of the oracles can be analyzed exclusively through the real-time constraints that the system satisfies.

On the definition of an atomic step. We defined an atomic step in Sect. 3.1 to consist of receiving a finite number of messages, changing the state, and sending a finite number of messages. Other results on asynchronous systems, partial synchrony, and failure detectors have adopted different definitions of an atomic step. For instance, in [22], a process may receive up to one message, make a state transition, and send an arbitrary but finite set of messages to other processes in a single step. On the other hand, in [18], a process may either receive a finite set of messages or send at most one unicast message in a single step, but it cannot do both. In [11], a process may receive at most one message, perform a state transition, and send at most one (unicast) message in a single step. In [10], a process may receive at most one message, perform a state transition, and send at most one message to all processes in a single step. The results in [17] explore solvability of consensus under various definitions of an atomic step including the variants defined above.

Interestingly, the choice of the definition has a significant impact on the validity of our results. The most sensitive aspect of the definition of an atomic step is the number of messages that a process may receive in a single step. Our results require that a process be able to receive multiple messages in a single step. Such sensitivity is a consequence of requiring the establishment of communicational fairness despite the sending process taking steps faster than a receiving process. For instance, if a process i sends one message per step to process j, and the system is 10-proc-fair and 1-com-fair, then i could send up to 10 messages to j between two consecutive steps by j, and to satisfy communicational fairness, j would be required to receive all 10 messages when it takes its next step.

However, our results are not sensitive to the number of messages that a process may send in a single step. The reason for such robustness is that processes can always ‘bundle’ multiple messages to a destination into a single message and effectively send multiple messages to a single process in just one message. Similarly, in systems where processes may send at most one message per step, messages to different processes may be sent with a slow-down of at most n steps (where n is the number of processes in the system) when compared to systems which follow our definition of atomic steps. Note that such a slow down affects *when* a message is sent, but it does not affect the number of steps that the recipient takes *while* the message is in transit. Thus, fairness guarantees may be preserved despite processes sending at most one message (instead of an arbitrary but finite number of messages) per step.

Open Questions. We have argued that several failure detectors encapsulate fairness in executions and provided evidence by demonstrating that all the failure detectors in the Chandra-Toueg hierarchy encapsulate such fairness constraints. This opens a larger question: *do all failure detectors encapsulate fairness?* The answer is arguably *no*. Notable candidates for counterexamples include the failure detectors proposed in [28] 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 we restrict our question to the extended Chandra-Toueg hierarchy (which include oracles like $\mathcal{T}$ [16], and other parametric oracles like the ones in [42, 34, 38]). If it turns out that all oracles that output process ids do encapsulate fairness, then such a result 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 better 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 [29]. In environments where an arbitrary number of processes may crash, the weakest failure detector for the problem is a stronger oracle $(\Diamond\mathcal{S}, \varSigma)$ [15]. Given that $\Diamond\mathcal{S}$ encapsulates some fairness constraints, and $\varSigma$ can be implemented in an asynchronous system with majority correct, we conjecture that $\varSigma$ 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 $\varOmega$, $\Diamond\mathcal{P}$, 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.

Acknowledgments. We thank Michel Raynal for his comments which improved this paper. We also thank the anonymous reviewers for their comments which improved the formal framework and extended the results to the $\mathcal{G}_*$ family of failure detectors.

References

1. Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM Journal on Computing*, 29(6):2040–2073, 2000.
2. Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *Proceedings of the* 15th *International Conference on Distributed Computing*, pages 108–122, London, UK, 2001. Springer-Verlag.
3. Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the* 23rd *ACM Symposium on Principles of Distributed Computing*, pages 328–337, 2004.
4. Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.
5. Antonio Fernández Anta and Michel Raynal. From an asynchronous intermittent rotating star to an eventual leader. *IEEE Transactions on Parallel and Distributed Systems*, 21(9):1290–1303, 2010.
6. Martin Biely, Martin Hutle, Lucia Draque Penso, and Josef Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In *Proceedings of the* 9th *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 4–20, 2007.
7. Martin Biely, Martin Hutle, Lucia Draque Penso, and Josef Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. Technical Report 54/2007, Technische Universität Wien, Institut für Technische Informatik, 2007.
8. Martin Biely, Peter Robinson, and Ulrich Schmid. Weak synchrony models and failure detectors for message passing k-set agreement. In *Proceedings of the* 13th *International Conference on Principles of Distributed Systems*, pages 285–299, 2009.
9. Francois Bonnet and Michel Raynal. Looking for the weakest failure detector for k-set agreement in message-passing systems: Is π_k the end of the road? In *Procceding of* 11th *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 149–164, 2009.
10. Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.
11. Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

12. K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.
13. Bernadette Charron-Bost, Rachid Guerraoui, and Andre Schiper. Synchronous system and perfect failure detector: solvability and efficiency issues. In *International Conference on Dependable Systems and Networks*, pages 523–532, 2000.
14. Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.
15. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the* 23^{rd} *ACM Symposium on Principles of Distributed Computing*, pages 338–346, 2004.
16. Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, and Petr Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, 65(4):492–505, 2005.
17. Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.
18. Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
19. Christof Fetzer. The message classification model. In *Proceedings of the* 17^{th} *ACM Symposium on Principles of Distributed Computing*, pages 153–162, 1998.
20. Christof Fetzer, Ulrich Schmid, and Martin Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the* 25^{th} *IEEE International Conference on Distributed Computing Systems*, pages 271–280, 2005.
21. Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distributed Computing*, 16(2-3):121–163, 2003.
22. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
23. Eli Gafni and Petr Kouznetsov. The weakest failure detector for solving k-set agreement. In *Proceedings of the* 28^{th} *ACM Symposium on Principles of Distributed Computing*, pages 83–91, 2009.
24. Rachid Guerraoui, Michal Kapalka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, 20(6):415–433, April 2008.
25. Rachid Guerraoui, Rui Oliveira, and André Schiper. Stubborn communication channels. Technical Report LSR-REPORT-1998-009, Ecole Polytechnique Federale de Lausanne, July 1998.
26. Jean-François Hermant and Josef Widder. Implementing reliable distributed real-time systems with the Θ-model. In *Proceedings of the* 9^{th} *International Conference on the Principles of Distributed Systems*, pages 334–350, 2005.
27. Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Chasing the weakest system model for implementing Ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, 6(4):269–281, 2009.
28. Prasad Jayanti and Sam Toueg. Every problem has a weakest failure detector. In *Proceedings of the* 27^{th} *ACM Symposium on Principles of Distributed Computing*, pages 75–84, 2008.
29. Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Proceedings of the* 8^{th} *International Workshop on Distributed Algorithms*, pages 280–295, 1994.
30. Dahlia Malkhi, Florin Oprea, and Lidong Zhou. Ω meets Paxos: Leader election and stability without eventual timely links. In *Proceedings of the* 19^{th} *International Symposium on Distributed Computing*, pages 199–213, 2005.
31. Achour Mostefaoui, Eric Mourgaya, and Michel Raynal. An introduction to oracles for asynchronous distributed systems. *Future Gener. Comput. Syst.*, 18(6):757–767, 2002.
32. Achour Mostéfaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the* 33^{rd} *International Conference on Dependable Systems and Networks*, pages 351–360, 2003.
33. Achour Mostéfaoui, Eric Mourgaya, Michel Raynal, and Corentin Travers. A time-free assmption to implement eventual leadership. *Parallel Processing Letters*, 16(2):189–207, 2006.
34. Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. On the computability power and the robustness of set agreement-oriented failure detector classes. *Distributed Computing*, 21(3):201–222, 2008.
35. Scott M. Pike and Paolo A.G. Sivilotti. Dining philosophers with crash locality 1. In *Proceedings of the* 24^{th} *IEEE International Conference on Distributed Computing Systems*, pages 22–29, 2004.
36. Scott M. Pike, Yantao Song, and Srikanth Sastry. Wait-free dining under eventual weak exclusion. In *Proceedings of the* 9^{th} *International Conference on Distributed Computing and Networking*, pages 135–146, 2008.
37. Sergio Rajsbaum, Michel Raynal, and Corentin Travers. Failure detectors as schedulers (an algorithmically-reasoned characterization). Technical Report 1838, IRISA, Université de Rennes, France, 2007.

38. Sergio Rajsbaum, Michel Raynal, and Coretin Travers. The iterated restricted immediate snapshot model. In *Proceedings of* 14th *International Conference on Computing and Combinatorics*, pages 487–497, 2008.
39. Peter Robinson and Ulrich Schmid. The Asynchronous Bounded-Cycle Model. In *Proceedings of the* 10th *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 246–262, 2008.
40. Srikanth Sastry and Scott M. Pike. Eventually perfect failure detection using ADD channels. In *Proceedings of the* 5th *international Symposium on Parallel and Distributed Processing and Applications*, pages 483–496, 2007.
41. Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. Crash fault detection in celerating environments. In *Proceedings of the* 23rd *IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.
42. Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the* 17th *ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998.