FAIRNESS PROPERTIES OF THE TRUSTING FAILURE DETECTOR

An Undergraduate Research Scholars Thesis

by

IAN MATSON

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jennifer Welch

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Ian Matson, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Fairness Properties of the Trusting Failure Detector

Ian Matson
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jennifer Welch
Department of Computer Science and Engineering
Texas A&M University

In 1985 it was shown by Fischer et al. that consensus, a fundamental problem in distributed computing, was impossible in asynchronous distributed systems in the presence of even just one process failure. This result prompted a search for alternative system models that were capable of solving such problems and culminated in the development of two helpful constructs: partially synchronous system models and failure detectors.

Partially synchronous system models seek to solve the problem of identifying process crashes by constraining the real-time behavior of the underlying system. In the resulting models, crashed processes can be detected indirectly through the use of timeouts. Failure detectors, on the other hand, address process crashes by directly providing (potentially inaccurate) information on failures. As a result, failure detectors were viewed as abstractions of real-time information. Pike et al. proposed a different perspective on failure detectors; as abstracting fairness properties.

Fairness in a system imposes bounds on the relative frequencies of communication and execution between processes in a system, and it was shown that four frequently-used failure detectors from the Chandra-Toueg hierarchy ($\mathcal{P}$, $\Diamond\mathcal{P}$, $\mathcal{S}$, $\Diamond\mathcal{S}$) encapsulate these fairness properties. This discovery suggests that failure detectors may be better understood as abstractions of fairness rather

than real-time properties as well as demonstrates the possibility to communicate results between systems augmented with failure detectors and partially synchronous system models.

In this thesis, we will be discussing an extension of the Pike et al. result to the trusting failure detector ($\mathcal{T}$). The trusting failure detector is the weakest failure detector to implement the problem of fault-tolerant mutual exclusion: a fundamental primitive for distributed computing.

ACKNOWLEDGMENTS

Contributors

I would first like to take this space to give a special thanks to Dr. Welch, my research advisor for this thesis. Her support and guidance throughout this entire process has been invaluable, and I owe much of what I accomplished to her encouragement. During the many points wherein the scope of the thesis needed to be changed, sometimes entirely, Dr. Welch never hesitated to provide valuable insight and direction. I am very grateful to have had the opportunity to study under her guidance and hope her the best as she concludes her time at Texas A&M.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Lastly, I would like to thank my friends for their unwavering support through the somewhat tumultuous periods that such theoretical research brought. Their many kind words and continued presence in my life provide a continual source of gratitude.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

No funding was received for this work.

1. INTRODUCTION

As computing continues to become more widespread and interconnected than ever, the study of distributed systems has become an area of particular interest. In general, distributed computing concerns the practice of utilizing multiple independent hosts to pursue some common objective. As these very principles form the core of developing technologies such as cloud computing, blockchain protocols, and network design, distributed systems and their related challenges such as communication and coordination among processes have become increasingly relevant. This thesis will focus primarily on two methods to overcome the challenges posed by faulty processes in distributed systems - failure detectors and fairness properties - and attempt to build on a technique presented in [1] to further unite the two topics.

1.1 Synchrony in Distributed Computing

When working with a system of distributed computers, it is important to define and understand how *synchronous* the underlying system is. The concept of synchrony can encompass a number of things about a system such as how frequently processes execute, the expedience with which communication is handled, or the bounds on process execution time, but in general, synchrony refers to knowledge or restrictions concerning temporal data or events. One extreme form of synchrony is lock-step synchrony: all processes execute at a fixed rate and messages are sent with a known delay. In such a system, execution is extremely precise and timing-related failures are trivial to detect. On the other end of the spectrum lies total asynchrony, wherein there are no assumed timing bounds on execution speed or message delivery. Processes may take arbitrarily long to execute and messages can be delayed for some unbounded but finite period. This form of a distributed system, of course, greatly increases the difficulty of solving even basic problems in the presence of process failures, as it is impossible to distinguish between a process that has permanently crashed and a process that is simply taking a significant amount of time to execute or communicate. In fact, in a well-known 1985 paper [2], it was demonstrated that achieving consensus on a single value among a group of processes was impossible even in the presence of a single failure in a completely asynchronous system. This result was dubbed the FLP Impossibility Result and has served as a key motivator for devising approaches for fault detection and more realistic partially-synchronous system models.

As briefly mentioned above, partially synchronous models were primarily formed as an attempt to more accurately represent conditions encountered in real-world contexts. For example, while it may be true that a message from one process to another might take arbitrarily long to be received, there is likely some bound within which the message can be expected to arrive in the absence of anomalous behavior. In physical settings, therefore, heuristics like this can be readily utilized to gather information on failures that simply aren't represented in completely asynchronous system models. In an effort to reflect this behavior, researchers have developed various models that commonly use timing constraints to enforce varying levels of synchrony. A few common approaches include bounding relative process execution speeds, bounding relative message delays, providing shared memory, and limiting the types and frequency of failures in the system. A few examples of partially synchronous system models include the timed asynchronous distributed system model [3], the asynchronous bounded-cycle model [4], and the Archimedean model [5] [6]. A survey of various forms of synchrony and their corresponding implications on solvability of the aforementioned consensus problem can be found in [7].

1.2 Fault Detection in Distributed Systems

Following the publishing of the FLP-impossibility result, it became clear that solving most problems using a purely asynchronous system was impractical in the presence of faults. One response to this development was to pivot to system representations that included some level of synchrony in an effort to more adequately reflect conditions present in physical systems. Another response was the introduction of a concept known as *failure detectors*. Failure detectors were first introduced in [8] by Chandra et al. as oracles providing failure-related information to each process in the system. The physical implementation of these failure detectors is left undefined, making failure detectors more of an abstraction of failure-detecting capabilities in a system than a top-tobottom implementation. Generally, failure detectors provide either a set of "trusted" or "suspected" processes with some set of restrictions on how timely or accurate those predictions may be. While other forms certainly exist, the bulk of failure detectors adhere to this pattern. The ability to gather information regarding failures, even if inaccurate, directly addressed the issue of distinguishing a slow process from a crashed one, and enabled systems augmented with these failure detectors to bypass the FLP-impossibility result. Consequentially, in the following years, much time was spent pursuing the "weakest" failure detectors to solve various distributed computing problems. This yielded the introduction of a variety of failure detectors- the majority of which are outside the scope of this paper. For the sake of this discussion, we will focus on the eight failure detectors introduced in [8]: the Chandra-Toueg hierarchy. These eight failure detectors have been shown in [8] to be reducible to the following four failure detectors:

- **Perfect Failure Detector ($\mathcal{P}$):** No process is ever suspected prior to crashing, and eventually all faulty processes will be suspected. The perfect failure detector is the weakest failure detector to solve terminating reliable broadcast [8].
- **Eventually Perfect Failure Detector ($\diamondsuit\mathcal{P}$):** There exists some time after which no correct process will be suspected, and eventually all faulty processes will be suspected. The eventually perfect failure detector is the weakest failure detector to solve wait-free eventually bounded-fair dining philosophers [9] and wait-free contention managers [1] [10].
- **Strong Failure Detector ($\mathcal{S}$):** Some correct process is never suspected, and eventually all faulty processes will be suspected. The strong failure detector is the weakest failure detector to solve consensus [8] and atomic broadcast [8].
- **Eventually Strong Failure Detector ($\diamondsuit\mathcal{S}$):** There exists some time after which some correct process is never suspected, and eventually all faulty processes will be suspected. The eventually strong failure detector is the weakest failure detector to solve consensus [2] [11] and atomic broadcast [2] [8].

[1] in shared memory systems

[2] given a majority of correct processes

In terms of relative strength, it was demonstrated in [8] that an asynchronous system augmented with $\mathcal{P}$ is sufficient to implement $\mathcal{S}$, and likewise a system augmented with $\diamondsuit\mathcal{P}$ is sufficient to implement $\diamondsuit\mathcal{S}$. Another, perhaps more succinct way to describe the attributes of each of these failure detectors are through the properties of *completeness* and *accuracy*. These properties were also introduced in [8] and are widely used to describe and define the behavior of failure detectors.

- **Strong Completeness** states that eventually all faulty processes will be permanently suspected by all correct processes.
- **Weak Completeness** states that eventually some faulty processes will be suspected by all correct processes.
- **Strong Accuracy** states that no process will be suspected prior to crashing.
- **Weak Accuracy** states that some correct process will never be suspected.

Since [8] demonstrates that weak completeness can be transformed into strong completeness in a system with all-to-all communication, we only consider strong completeness. When defining the failure detectors in terms of completeness and accuracy, the (eventually) perfect failure detector provides strong completeness and (eventually) strong accuracy, and the (eventually) strong failure detector provides strong completeness and (eventually) weak accuracy.

While the failure detectors presented Chandra-Toueg hierarchy were the first proposed instance of failure-detecting oracles, many additional failure detectors have been proposed and used to implement various primitives in otherwise asynchronous systems. Below we introduce three failure detectors of particular significance:

- The **quorum** (Σ) failure detector was introduced in [12] and outputs a set of processes at each process with the constraint that any two outputs (at each process and time) have a nonempty intersection and eventually every set consists of only correct processes. The quorum failure detector is the weakest failure detector to implement an atomic register [12].
- The **leader** (Ω) failure detector was first introduced in [11] and outputs a single process id at each process with the constraint that eventually all outputs will consist of the same correct process. The leader failure detector was shown to be equivalent to $\diamondsuit\mathcal{S}$ and is therefore the weakest to implement consensus [8] and atomic broadcast [8].

- The **trusting** ($\mathcal{T}$) failure detector was introduced in [13] and outputs a set of trusted processes at each time. There exists some time after which no faulty process will be trusted, and eventually all correct processes will be trusted. Additionally, if any process is ever untrusted, it has crashed. The trusting failure detector is the weakest failure detector to solve fault-tolerant mutual exclusion [13].

1.3 Notable Problems in Distributed Computing

To provide context for the development of various failure detectors, we will seek to define a few key problems and describe the weakest failure detector-augmented system capable of solving them.

- The **consensus** problem is one of the most widely discussed problems in distributed computing and is often used as a base metric when discussing the capabilities of a particular system. Generally, the consensus problem requires a set of processes to decide on a single value with a few restrictions: any two correct processes must decide on the same value and any final value must have been originally proposed. The weakest failure detector to solve this problem in message passing systems with any number of faulty processes is the pair $(\Omega + \Sigma)$ [14] [12].
- The **dining philosophers problem** is a prevalent problem in distributed computing that mainly deals with synchronization. Generally, the dining philosophers problem is related to local exclusion between processes that share resources. This is, in fact, a weaker form of complete mutual exclusion, mentioned below. It was proven in [9] that $\diamond\mathcal{P}$ is the weakest failure detector to implement this problem in asynchronous message passing systems.
- The **fault-tolerant mutual exclusion problem** is in fact a stronger version of the aforementioned dining philosophers problem, as exclusion is both global and perpetual. The problem itself consists of a set of processes sharing resources wherein only a single process may access a particular resource at any time. The weakest failure detector to implement this problem was proven in [13] to be the trusting failure detector ($\mathcal{T}$) in asynchronous message passing systems.

1.4 Fairness and Failure Detectors

As both failure detectors and partially synchronous system models serve to address the problem posed by the FLP-impossibility result, they naturally share a few key similarities. While failure detectors address the FLP-impossibility result by providing direct information about crashed processes, partially synchronous systems place restrictions on the temporal behavior of the overall system, indirectly prohibiting certain fault patterns and providing information about failures. Intuitively, one might notice that a system augmented with failure detectors is simply an alternative form of partial synchrony. This intuition was captured and rigorously defined in [1] using the concept of *fairness*. In particular, Pike et al. defined two different forms of fairness: process fairness and communication fairness. Both forms of fairness are defined as constraints on the relative frequency and ordering of process execution and communication. We will define these terms more rigorously in a later section, but for the time being, note that some process i is *computationally fair* with respect to another process j if some bound Φ exists such that i will take at least one step for every $\Phi + 1$ steps taken by j. Similarly, a process i is *communicationally fair* with respect to some process j if some bound Φ exists such that for every message sent from i to j, j will take no more than Φ steps before receiving the message.

These fairness constraints lend themselves quite handily to defining various partially synchronous models. In [1], Pike et al. proved that four failure detectors, $\mathcal{P}$, $\diamond\mathcal{P}$, $\mathcal{S}$, and $\diamond\mathcal{S}$ encapsulate these fairness constraints and that various partially synchronous system models defined using these constraints are sufficient to emulate these failure detectors. The most significant implication of this development is that fairness may very well be a fundamental unit of partial synchrony. While the concept of failure detectors remains a useful construct as it directly relates to failure detection properties found in physical systems, understanding the corresponding fairness properties inherent to various failure detectors would enable research to progress on a more universal front. Additionally, much research has been conducted concerning the weakest failure detectors to solve certain problems such as consensus [14], dining philosophers [9], and fault-tolerant mutual exclusion [13]. The identification of fairness properties present in these critical failure detectors could enable the application of these results to related partially synchronous system models utilizing these fairness properties.

1.4.1 Fairness-Based System Models

As briefly mentioned in the previous section, the fairness properties introduced in [1] lend themselves quite handily to defining various fairness-based partially synchronous system models. In [1], four such models are proposed and equated with various failure detectors in the Chandra-Toueg hierarchy. As the majority of work presented in this thesis builds off of these definitions, we will highlight them below.

- The **All Fair** ($\mathcal{AF}$) system model is an asynchronous system model wherein during every run, all processes are k-proc-fair and d-com-fair for known *k* and *d*. This system model was shown to functionally equivalent to the perfect failure detector.
- The **Some Fair** ($\mathcal{SF}$) system model is an asynchronous system model wherein during every run, some process is k-proc-fair and d-com-fair for known *k* and *d*. This system model was shown to functionally equivalent to the strong failure detector.
- The **Eventually All Fair** ($\diamondsuit\mathcal{AF}$) system model is an asynchronous system model wherein during every run, all processes are eventually k-proc-fair and d-com-fair for known *k* and *d*. This system model was shown to functionally equivalent to the eventually perfect failure detector.
- The **Eventually Some Fair** ($\diamondsuit\mathcal{SF}$) system model is an asynchronous system model wherein during every run, some process is eventually k-proc-fair and d-com-fair for known *k* and *d*. This system model was shown to functionally equivalent to the eventually strong failure detector.

In this thesis, we introduce a new fairness-based system model- the **Active Fair** ($\triangledown\mathcal{AF}$) system model. The active fair system model is a system model where each process will *execute* fairly, that is, every process may behave unfairly for some initial period until executing, after which each process will remain k-proc-fair and d-com-fair until it crashes. More precisely, some system is considered to provide the Active Fair fairness property if and only if for every process i and time

t, if i has taken a step prior to t, then i is both k-proc-fair and d-com-fair for all $t' > t$, for known k and d.

1.4.2 Trusting Failure Detector

The trusting failure detector ($\mathcal{T}$) is fairly similar to the four failure detectors in the Chandra-Toueg hierarchy with a few minor caveats. Rather than outputting a list of suspected processes, the trusting failure detector outputs a list of trusted processes at each process in the system. Much like $\diamond\mathcal{P}$, $\mathcal{T}$ provides both strong completeness and eventually strong accuracy. Additionally, $\mathcal{T}$ also provides a property we will call *trusting accuracy*: if the trusting failure detector ever stops trusting a process, that process has crashed. This has the implication that once a process becomes trusted, it will never become untrusted prior to crashing. It is relatively clear to see that $\mathcal{T}$ must be strictly stronger than $\diamond\mathcal{P}$ [13] as it provides all guarantees of $\diamond\mathcal{P}$ in addition to monotonically increasing trust. Furthermore, $\mathcal{T}$ is strictly weaker than $\mathcal{P}$ [13], giving the relation $\diamond\mathcal{P} \preceq_{\mathcal{E}} \mathcal{T} \preceq_{\mathcal{E}} \mathcal{P}$, where $\mathcal{E}$ is any environment wherein the number of correct processes is nonzero ($\mathcal{D}_1 \preceq_{\mathcal{E}} \mathcal{D}_2$ for failure detectors $\mathcal{D}_1$ and $\mathcal{D}_2$ if $\mathcal{D}_2$ can implement $\mathcal{D}_1$ in environment $\mathcal{E}$).

In [13], it was proved that the *trusting failure detector* is the weakest failure detector to solve the problem of *fault tolerant mutual exclusion* (FTME) in distributed systems provided that a majority of processes are correct. Additionally, it was shown that in a system with a minority of correct processes, the trusting failure detector in conjunction with the strong failure detector ($\mathcal{T} + S$) is both sufficient and necessary to solve FTME. Consequently, identifying the intrinsic fairness properties of the trusting failure detector is of significant interest as it would both enable the construction of a minimal partially synchronous system model capable of solving FTME as well as further develop understanding concerning the relationship between failure detectors and fairness properties.

1.5 Objectives

The goal of this paper is to utilize the approach used in [1] to identify a fairness-based partially synchronous model that encapsulates the fairness properties inherent to the trusting failure detector. We will demonstrate that these fairness properties are equivalent to the trusting failure detector using the following approach:

- Assuming that a asynchronous system model is augmented with some failure detector $\mathcal{D}$, create a scheduler that ensures the corresponding fairness properties are met.
- Assuming some system model that adheres to a set of fairness properties, emulate the output of the corresponding failure detector $\mathcal{D}$.

By demonstrating that the trusting failure detector is sufficient to implement a fairness-based partially synchronous system, we demonstrate that $\mathcal{T}$ encapsulates *at least* as much fairness as that of the proposed system model. By then demonstrating the the proposed system model implements $\mathcal{T}$, we demonstrate that the the trusting failure detector contains *no more* fairness that that of the proposed system model.

2. SYSTEM DEFINITION

In the following section, we will define the system we will be using for the remainder of the discussion, building heavily on the standard asynchronous system models described in [1], [2], [8].

The system is assumed to be a collection of n processes $\Pi = \{1, .., n\}$. The system is capable of all-to-all communication, and a broadcast is considered to be an atomic step. Communication is semi-reliable: messages cannot be duplicated or corrupted, but they may be dropped upon process crashes or delivered out of order. Process executions and message transmissions may take an arbitrary but finite amount of time to complete.

2.1 Global Time

We assume the existence of a discrete global time with range $T = \mathbb{N}$. The existence of this clock is merely theoretical- it serves only the purpose of description and no processes have access to its values. The time value is not a measure of absolute physical time, but rather counts the number of discrete events that occur within the system.

2.2 Faults and Fault Patterns

The only form of failures considered in this thesis is crashing. A process crashes by permanently and unexpectedly halting. Processes that crash at some point during a given system run are considered *faulty* while all non-faulty processes are considered *correct*. A *fault pattern* is some function $F \in \mathcal{F}$ mapping $T \rightarrow 2^{\Pi}$. $F(t)$ provides a list of all crashed processes at some time t. Note that crashed processes cannot recover, so $\forall t, t' \in T, t < t' \implies F(t) \subseteq F(t')$. We define *faulty*$(F) = \cup_{\forall t \in T} F(t)$ and *correct*$(F) = \Pi -$ *faulty*(F). If for some $p \in \Pi$, $p \in$ *faulty(F)*, we say that process p is *faulty* in F. Likewise, if $p \in$ *correct*(F), we say that p is correct in F. For all fault patterns F, we assume the existence of at least one correct process, that is $\forall F \in \mathcal{F},$ *correct*$(F) \neq \emptyset$.

2.3 Failure Detectors

As defined in [8], a failure detector is some oracle that provides information regarding failures. Each process in Π is considered to augmented with a local failure detector module. In this paper, we consider two forms of failure detectors: those that output a set of suspected processes when queried and those that output a set of trusted processes. Note that the latter is essentially the complement of the former, that is, for any failure detector output, the set of suspected processes is the complement of the set of trusted processes. This allows for comparisons to be drawn across both forms of failure detectors.

We consider a *failure detector history* $H \in \mathcal{H}$ to represent the output of some failure detector at each process over time. More precisely, $H \in \mathcal{H}$ is some function mapping $T \times \Pi \to 2^{\Pi}$, and $H(p,t)$ is the output of the failure detector module at process p at some time t. Note that two processes may not necessarily agree on the set of suspected or trusted processes at some time t, that is $H(p,t)$ may not be equivalent to $H(q,t)$ for some $p, q \in \Pi, t \in T$.

A failure detector $\mathcal{D}$ provides (potentially incorrect) information regarding some fault pattern F during a system execution. Formally, $\mathcal{D}$ is some function mapping from $\mathcal{F} \to 2^{\mathcal{H}}$, and $\mathcal{D}(F)$ is the set of valid failure detector histories for the fault pattern $F \in \mathcal{F}$. For the duration of this thesis, we focus primarily on the *trusting failure detector* as defined in [13]. While not explicitly stated in [13], it is assumed that the initial output of $\mathcal{T}$ during any run is the empty set, that is $\forall F \in \mathcal{F}, \forall A \in \mathcal{T}(F)$, the first output in A is $\emptyset$.

2.4 Processes

Each process can be modeled as a state machine consisting of various discrete states. Processes can transition between these states through atomic steps. Each atomic operation takes as an input the current state of the process, the set of messages that have been received from other processes, and the output of the failure detector. Each atomic operation provides as an output the new state for the process and the set of messages to be sent to other processes. These messages can then be sent over asynchronous communication channels in which messages may or may not be delivered in order. When delivered, messages are stored in a local receive buffer and are considered to be *received* by the process when it takes the next atomic step.

2.5 Configurations and Runs

A system *configuration* describes the current state of each process in the system as well as the set of messages that have been sent but not received. A system run is a sequence of alternating configurations and steps of the form $\alpha = C_0 s_1 C_1 s_2 ...$ and is defined with respect to a set of processes Π, a fault pattern F, and a history H of the trusted failure detector ($H \in \mathcal{T}(F)$). Note that each step s_i is defined to be a step of an individual process. For each configuration C_i, C_i can be obtained by applying s_i to configuration C_{i-1}. Note that no process other than the process taking step s_i changes state, though messages that are in-transit may be delivered and messages sent by the process taking step s_i may be added to the set of in transit messages.

Time value $i \in T$ corresponds to the configuration C_i in α. For the remainder of this thesis, we will refer to any such time as t_i rather than i for the sake of notational clarity.

Messages that are in transit from some process i to j at some time t are guaranteed to be delivered provided that i and j are both live at t. This has the implication that messages sent from i may be dropped after being in transit if process i crashes.

3. EXTRACTING FAIRNESS FROM THE TRUSTING FAILURE DETECTOR

In the following section, we present a distributed scheduler that uses an asynchronous system augmented with the trusting failure detector to provide the fairness guarantees necessary to implement the Actively Fair system model.

3.1 Interface Between Scheduler and Application

The scheduler-application interface presented in Algorithms 1 and 2 is modeled after those presented in [1]. The scheduler interacts with the application through the procedures presented in Algorithm 2. The scheduler can allow the underlying application to take a single application step through the use of the EXECUTEAPP() procedure, enabling the application to take a single step. If there happen to be multiple enabled actions available to the application, it is assumed that the scheduler makes a random choice subject to the constraint that every continuously enabled action will be executed after some finite time.

The application receives and sends messages through the use of RECEIVEAPP() and SENDAPP(), respectively. Both of these procedures are buffered in a local send and receive buffer. During any application step, the application can invoke RECEIVEAPP() to receive all locally buffered messages as well as invoke SENDAPP() to insert some message to the local send buffer. These messages will then be sent by the scheduler to the destination processes.

3.2 Shared Data Structures

By [13], the trusting failure detector is strictly stronger than the eventually perfect failure detector; that is, the trusting failure detector is capable of implementing the eventually perfect failure detector in the presence of any number of failures. As the eventually perfect failure detector can implement consensus in a system with a majority of correct processes [8], this implies that the trusting failure detector can also implement consensus provided a minority of process crashes. In [15], it was demonstrated that if some system is capable of implementing consensus, then it is also possible to construct a linearizable implementation of any sequential data structure. Consequently, we utilize various linearizable shared data structures in Algorithm 1.

Algorithm 1 utilizes a number of various shared data structures that are accessible at each local process. Each data structure is one of two types: *shared set* and *shared list*. Each type is considered to be capable of a number of linearizable operations that we define below.

- *Shared set:*
    1. Union operator: For two sets A and B, performing the operation $A \cup B$ will return some set C such that C is the union of the two sets A and B.
    2. Difference operator: For two sets A and B, performing the operation $A - B$ will return some set C such that C is the set of all $a \in A$ where $a \notin B$.
- *Shared list:*
    1. *l.remove(a)*: removes all elements of value a list l.
    2. *l.append(a)*: appends the element a to the back of list l.
    3. *l.peek()*: returns the value of the first element in l.
    4. *l.rotate()*: removes the first element in l and adds it to the back of l.

3.3 Algorithm Description

Algorithm 1 shows the scheduler running locally at each process *i*. The scheduler utilizes the trusted failure detector to provide the active fairness property to the system of processes. On a high level, the scheduler functions as a distributed round-robin. As the shared list $ordering$ is initially empty and application steps can only be taken in Action 3 when $ordering.peek() = i$, initially no process in the system is able to take a step. During this phase, the process may simply monitor its local failure detector and update the corresponding shared sets $trust_j$ while waiting to become trusted by $\lfloor n/2 \rfloor + 1$ other processes. Note that as the system must contain a majority of correct processes, upon some process i becoming trusted by $\lfloor n/2 \rfloor + 1$ process, i must have been trusted by at least 1 correct process. This fact serves to make the scheduler non-blocking- if some process i starts taking application steps and later crashes, we have by the trusting accuracy property of the trusting failure detector that i will eventually be untrusted by whatever correct process initially trusted it allowing i to removed from $ordering$ and preventing halting. In essence, through requiring that each process be trusted by a majority prior to being added to $ordering$, we can guarantee a correct "witness" for each executing process.

Once a process becomes trusted by some majority of processes, it may execute Action 2 and append itself to the list $ordering$. Each process may execute Action 3 and take an application step upon reaching the front of $ordering$ and will move itself to the end of ordering after having done so. As $ordering$ is a linearizable shared data structure, the precondition of Action 3 may be satisfied for at most one process during any given system configuration. This fact enforces a strict round-robin ordering on process executions, as only the front process in $ordering$ may take an application step and will be moved to the back of the list upon doing so.

Application to application communication in Algorithm 1 is performed through the use of a shared message buffer $messages$. To send an application message m from some process i to some process j, process i will first add m to its local send buffer. After concluding the procedure EXECUTEAPP(), i will append the element $(i, j, M = i.sendBuffer_j)$ to the shared list $messages$, indicating that M is some set of messages from i to j. Prior to taking an application step, j will retrieve this element from $messages$ and add it to its local receive buffer, allowing it to be processed during the procedure EXECUTEAPP().

3.4 Fairness Guarantees Provided

We claim that when run using the previously defined asynchronous system model provided a majority of correct processes, Algorithm 1 provides the *Active Fair* fairness guarantee with respect the ordering of application steps taken by each process and timing of application to application communication. In particular, we claim that Algorithm 1 provides the following to each application:

- *Local Progress*: Every correct process is scheduled to take an application step infinitely often.
- *Process Fairness*: Every process is actively 1-proc-fair. That is, for some processes $i, j \in \Pi$, if i takes an application step at t_i, then for any interval $[t_j, t'_j]$ where j takes application steps

at t_j and t'_j and $t_j < t'_j$, i takes at least one application step in the interval $[t_j, t'_j]$.

- *Communication Fairness*: Every process is actively 1-com-fair. That is, for some processes $i, j \in \Pi$, if i takes an application step at t_i, then for any message m sent to process i at time $t, t > t_i$, process i will take no more than 1 step before receiving m.

Algorithm 1 Actions for scheduler at process i

```
    Shared data structures available at process $i$
1:  **shared list** ordering $\leftarrow \emptyset$                ▷ Ordering of process executions
2:  **shared list** messages $\leftarrow \emptyset$                ▷ Common pool of in-transit messages
3:  **for all** $j \in \Pi$ **do**
4:      **shared set** trust$_j \leftarrow \emptyset$             ▷ Set of processes that have trusted $j$
    Local data structures available at process $i$
5:  **set** $\mathcal{T}_0 \leftarrow \emptyset$                   ▷ Prior value of $\mathcal{T}$, used to determine changes over time
6:  **for all** $j \in \Pi$ **do**
7:      **set** sendBuffer$_j \leftarrow \emptyset$                ▷ Local send buffer for messages sent to $j$
8:      **set** receiveBuffer$_j \leftarrow \emptyset$             ▷ Local receive buffer for messages from $j$

9:  {**upon** $\mathcal{T} \neq \mathcal{T}_0$}                    ▷ Action 1
10:     **for all** $j \in \mathcal{T} - \mathcal{T}_0$ **do**     ▷ There are new elements in $\mathcal{T}$
11:         trust$_j \leftarrow$ trust$_j \cup \{i\}$              ▷ Indicate that $i$ trusts $j$
12:     **for all** $j \in \mathcal{T}_0 - \mathcal{T}$ **do**     ▷ Processes have been untrusted
13:         ordering.$remove(j)$                                   ▷ Remove $j$ from ordering
14:     $\mathcal{T}_0 \leftarrow \mathcal{T}$                     ▷ Reset value of $\mathcal{T}_0$

15: {**upon** $(|trust_i| \geq \lfloor n/2 \rfloor + 1) \wedge (i \notin$ ordering$)$}    ▷ Action 2
16:     ordering.$append(i)$                                       ▷ Majority trusts $i$, safe to add to ordering

17: {**upon** $ordering.peek() = i$}                               ▷ Action 3
18:     **for all** $(j, k, M) \in messages$ **where** $k = i$ **do**    ▷ For any message sent to $i$
19:         receiveBuffer$_j \leftarrow$ receiveBuffer$_j \cup M$  ▷ Add message to local buffer
20:         messages.$remove((j, k, M))$                           ▷ Remove message from shared buffer
21:     EXECUTEAPP()                                               ▷ Execute an enabled application step
22:     **for all** $j \in \Pi - \{i\}$ **do**
23:         $M \leftarrow$ sendBuffer$_j$                          ▷ Populate $M$ with buffered messages
24:         sendBuffer$_j \leftarrow \emptyset$                    ▷ Empty send buffer
25:         messages.$append((i, j, M))$                           ▷ Add buffered messages to shared buffer
26:     ordering.$rotate()$                                        ▷ Move $i$ to the back of ordering
```

Algorithm 2 Interaction between the scheduler and the application at process i

```
procedure EXECUTEAPP
    RECEIVEAPP()
    Execute an enabled application step
    Application step invokes SENDAPP(m,j) to send message $m$ to process $j$.
procedure RECEIVEAPP
    $returnValue \leftarrow \cup_{\forall j \in \Pi - \{i\}} \{(i.\text{receiveBuffer}_j, j)\}$
    for all $j \in \Pi - \{i\}$ do
        $i.\text{receiveBuffer}_j \leftarrow \emptyset$
    return $returnValue$
procedure SENDAPP($m, j$)
    $i.\text{sendBuffer}_j \leftarrow i.\text{sendBuffer}_j \cup \{m\}$
```

3.5 Proof of Correctness

In this section we present a series of proofs concerning the correctness of Algorithm 1. More particularly, we show that Algorithm 1 provides *local progress*, *process fairness*, and *communication fairness* to each process.

Lemma 1. *For any system run α and correct processes $i, j \in \Pi$, if i trusts j at some time t_i, there exists some later time t' such that for all $t > t'$, $i \in trust_j$.*

Proof. Let time t_i be some time at which process i begins to trust process j, that is, $j \notin i.\mathcal{T}$ at $t_i - 1$ and $j \in i.\mathcal{T}$ at t_i. First, note that $i.\mathcal{T}_0$ must always contain some prior output of $i.\mathcal{T}$ as both are initially equal to $\emptyset$ and $i.\mathcal{T}_0$ is only set to $i.\mathcal{T}$ during Action 1. Recall that any process that is ever untrusted by the trusting failure detector has crashed, so t_i must be the first time that i trusted j. Consequently, at time t_i $i.\mathcal{T} \neq i.\mathcal{T}_0$ and $j \in i.\mathcal{T} - i.\mathcal{T}_0$. Note that as j will never be added to $i.\mathcal{T}_0$ until Action 1 is executed and j will never be untrusted by $i.\mathcal{T}$ since j is correct, Action 1 will remain continuously enabled until executed.

When process i eventually executes Action 1, because $j \in i.\mathcal{T} - i.\mathcal{T}_0$, i will add itself to $trust_j$, satisfying the lemma. □

Theorem 1. *In Algorithm 1, every correct process will take an application step infinitely often.*

Proof. Consider some process correct i and system run α. To demonstrate that i will take an application step infinitely often, we must show that Action 3 is enabled infinitely often. We will first show that there exists some time t_i such that for all $t > t_i$, $i \in ordering$.

By the eventually strong accuracy property of the trusting failure detector in conjunction with Lemma 1, we have that every correct process will eventually trust i and add itself to $trust_i$. As there are at least $\lfloor n/2 \rfloor + 1$ correct processes in the system, this implies that eventually $|trust_i| \geq \lfloor n/2 \rfloor + 1$ and Action 2 will remain continuously enabled at i. Eventually, i will execute Action 2 and append i to $ordering$. As processes are only removed from $ordering$ upon being untrusted by some process in Action 1 and by the trusting accuracy property of the trusting failure detector processes will only become untrusted after having crashed, i will not be removed form $ordering$.

Now we have shown that for every correct process i, i will eventually be appended $ordering$ and not be removed. We will now demonstrate that any process $i \in ordering$ will execute Action 3 infinitely often.

Consider some process j such that $ordering.peek() = j$ at some time t. Note that at least 1 correct process has trusted j prior to t, as if $j \in ordering$ then at some time prior j has executed Action 2. As Action 2 requires trust from a majority of processes and there exists a majority of correct processes, at least one of the processes that trusted j must be correct. While j is live, Action 3 will remain continuously enabled until executed. Note that the last line in Action 3 will remove j from the front of $ordering$ and add it to the back. Recall that at least one correct process k has trusted j prior to t, so if j crashes prior to executing Action 3 it will eventually be untrusted by k and removed from $ordering$ when k executes Action 1. Consequently, we have that for any j where $j = ordering.peek()$, j will eventually be removed or moved to the back of $ordering$.

We have now shown that any correct process i will eventually be a member in $ordering$ and that the front of $ordering$ will eventually either be removed upon crashing or execute Action 3 and move to the back of $ordering$. This implies that i will reach the front of $ordering$ infinitely often and take an application step by executing Action 3. □

Theorem 2. *In Algorithm 1, every process is actively 1-proc-fair.*

Proof. To demonstrate that every process is actively 1-proc-fair, we must show that for any two processes $i, j \in \Pi$ and time t, if i has taken an application step at time t_i, then for any interval $I = [t_j, t'_j]$ beginning after t_i where j takes a step at both t_j and t'_j, i must take at least 1 application step in I or i has crashed prior to t'_j.

First, note that each process will appear in $ordering$ at most once, as Action 2 is the only action adding some process k to $ordering$ and requires that k is not already present in $ordering$. Now, for the purposes of contradiction, assume that process i has taken an application step prior to t_j, but does not take an application step in the interval $[t_j, t'_j]$. If i has taken an application step, it must be true that $i \in ordering$ as application steps are only taken when $ordering.peek() = i$ and Action 3 is executed by i. Now examine time t_j. When j takes an application step during Action 3, it calls $ordering.rotate()$, moving itself to the back of $ordering$. Recall that process j can only be in $ordering$ once, so this implies that for every process $k \in ordering$ at t_j, k must be removed or execute Action 3 before j reaches the front of $ordering$. Consequently, when j executes Action 3 again and takes an application step at time t'_j, either i has taken executed Action 3 in the interval $[t_j, t'_j]$ or i has been removed from $ordering$. As i will only be removed from $ordering$ if it is untrusted by some process, this implies that i has crashed by the trusting accuracy property of the trusting failure detector. Thus we have that either i will take an application step in the interval $[t_j, t'_j]$ or crash prior to t'_j. □

Theorem 3. *In Algorithm 1, every process is actively 1-com-fair.*

Proof. Consider some run α and processes $i, j \in \Pi$. To demonstrate that i is actively 1-com-fair, we must show that if i has taken an application step at some time t_i, then for any message m sent sometime after t_i from i to j, j will receive m prior to taking an application step or i has crashed.

Let the time that j first runs EXECUTEAPP() after t_i be t_j. Examine any message m sent from i to j using SENDAPP() at some time t_m, where $t_i < t_m < t_j$. Note that SENDAPP() adds m to $i.sendBuffer_j$ at time t_m. As SENDAPP() is called during EXECUTEAPP, which in turn is called during Action 3, it must be true that after calling EXECUTEAPP(), $m \in sendBuffer_j$.

After EXECUTEAPP() is called, i will then set $M \leftarrow sendBuffer_j$ and append the element (i, j, M) to $messages$, where $m \in M$, provided that i has not crashed.

Now examine t_j. Prior to calling EXECUTEAPP(), j will add any element of the form $(i, j, M) \in messages$ to $receiveBuffer_i$. As $t_i < t_j$, this means m will be added to $receiveBuffer_i$. Consequently, when RECEIVEAPP(), the application at j will receive m prior to taking an enabled application step. □

4. SIMULATING T USING ACTIVE FAIRNESS

In the following section, we present an algorithm implementing the trusting failure detector when run using the Active Fair system model. This construction demonstrates that the amount of synchronism provided by the trusting failure detector is encapsulated by the Active Fair system model.

4.1 Algorithm Description

Algorithm 3 shows the actions available at each process i. All processes in the system are subject to the constraints provided by the Active Fair system model; namely that each process is actively 1-proc-fair and actively 1-com-fair. The algorithm itself is identical to Algorithm 3 in [1] with the exception of a few notational modifications. In general, the algorithm functions by maintaining a list $trusted$ at each process, where $trusted$ the output of $\mathcal{T}$. At each step, process i sends a heartbeat message to all other processes, checks for the presence of incoming heartbeats, and modifies the contents of $trusted$ accordingly. Some process j is added to the set $i.trusted$ when i receives a heartbeat from j and removed from $trusted$ if $i.timer_j$ expires by reaching 0. We claim that for any run α and process i, the contents of $i.trusted$ satisfies the output constraints of the trusting failure detector, namely *eventually strong completeness*, *eventually strong accuracy*, and *trusting accuracy* [13]. For convenience, we have listed these constraints below.

- *Eventually strong completeness* states that eventually, no crashed process is trusted by any correct process.
- *Eventually strong accuracy* states that eventually, every correct process is permanently trusted by every correct process.
- *Trusting accuracy* states that every process j that stops being trusted by process i is crashed.

4.2 Proof of Correctness

In this section, we will establish a proof of correctness for Algorithm 3 by demonstrating that the set $trusted$ satisfies eventually strong completeness, eventually strong accuracy, and trusting accuracy. These proofs are comparable to those used in [1].

Algorithm 3 Using the Actively Fair system model to implement $\mathcal{T}$

```
1: set trusted ← $\emptyset$
2: const integer timeout ← $1$
3: for all $j \in \Pi$ do
4:     integer timer$_j$ ← $0$

5: $\{true\}$:                                                  ▷ Action 1
6:     receive $msgSet$
7:     for all $j \in \Pi$ do
8:         send $(HB, i)$ to $j$
9:         if $(HB, j) \in msgSet$ then
10:            timer$_j$ ← timeout
11:            trusted ← trusted $\cup\, j$
12:        if timer$_j = 0$ then
13:            trusted ← trusted $-\, j$
14:        timer ← $max(0,$ timer$_j - 1)$
```

Theorem 4. *Algorithm 3 satisfies eventually strong completeness; that is, there exists a time after which every crashed process is permanently suspected.*

Proof. Let j be some process that has crashed at t_j and let i be any live process. As j has crashed and will take no more steps, it will no longer send any messages to i after t_j. Eventually, all messages of the form (HB, j) sent from j to i prior to t_j will be received by i, implying that $i.timer_j$ will no longer increase. By the local progress property of the Active Fair system model, i will continue to execute infinitely often after t_j, and at each step i will reduce the value of $i.timer_j$ by 1. Consequently, $i.timer_j$ will eventually equal zero, and i will remove j from $i.trusted$. □

Lemma 2. *For any run α and processes $i, j \in \Pi$, if i takes a step at time t_i and j receives (HB, i) at t_j, $t_j > t_i$, then for all $t > t_j$, $i \in j.trusted$ or i has crashed.*

Proof. As i has executed at time t_i, we have that i is 1-proc-fair; that is, for any two successive steps by j, i will take a step prior to the second step. If j received (HB, i) at time t_j, then at t_j process j sets $timer_i = 1$ and adds i to $j.trusted$. Note that at the conclusion of 5, j decrements $timer_i$, meaning $timer_i = 0$. As i is 1-proc-fair, i will take a step and send (HB, i) to j or crash before j takes another step. Because i is also 1-proc-fair, j will receive this message before taking 1 step. Thus, we have that during its next step, provided that i has not crashed, j will receive (HB, i) from i and set $timer_i = 1$. As $timer_i$ is set to 1 prior to line 12, the condition necessary to remove i from $j.trusted$ will not be met and i will remain in $j.trusted$ until i crashes. □

Theorem 5. *Algorithm 3 satisfies eventually strong accuracy; that is, there exists a time after which every correct process is permanently trusted by every correct process.*

Proof. Let processes $i, j \in \Pi$ be any pair of correct processes. By the local progress property of the trusted failure detector, j will eventually take an application step and send (HB, j) to i. Eventually, this message will be received by i during 5. By Lemma 2, j will remain in $i.trusted$ until j crashes. As j is correct and will never crash, we have that for any pair of correct processes $i, j \in \Pi$, i will eventually permanently trust j. □

Theorem 6. *Algorithm 3 satisfies trusting accuracy; that is, every process j that stops being trusted by process i has crashed.*

Proof. Consider processes $i, j \in \Pi$. If process i has trusted process j, it must be true that at some time t process i received (HB, j) from j. Additionally, j must have taken a step at some time t', where $t' < t$. Consequently, Lemma 2 applies and we have that j will remain in $i.trusted$ until j crashes. Consequently, if j ever becomes untrusted by i, then j has crashed. □

5. CONCLUSION

5.1 Fairness and the Trusting Failure Detector

Throughout the course of this thesis, we have demonstrated the presence of fairness properties that are intrinsic to the trusting failure detector. As in [1], this further suggests that failure detectors are best viewed as an abstraction of relative temporal constraints such as the relative frequency and ordering of system events rather than as constraints on the real-time behavior of a system. In particular, we have shown that the trusting failure detector is sufficient to implement the Active Fair ($\triangledown\mathcal{AF}$) system model in an environment with a minority of process crashes, and that $\triangledown\mathcal{AF}$ is sufficient to implement the trusted failure detector.

An important detail to draw attention to is that this relationship does not necessarily imply that $\triangledown\mathcal{AF}$ is the weakest system model to implement $\mathcal{T}$. While this is certainly possible, it must first be shown that $\mathcal{T}$ can implement $\triangledown\mathcal{AF}$ even in an environment with a majority of process crashes. The reason for this distinction is that the established relationship shows that $\mathcal{T}$ *in conjunction with a majority correct environment* encapsulates at least as much fairness as $\triangledown\mathcal{AF}$, not necessarily that $\mathcal{T}$ alone contains as much fairness as $\triangledown\mathcal{AF}$.

5.2 Relevance of Results to ($\mathcal{T}$ + $\mathcal{S}$)

Recall that in [13], it was demonstrated that the failure detector $\mathcal{T}$ is the weakest failure detector to implement fault-tolerant mutual exclusion in a majority-correct environment, and that the failure detector $(\mathcal{T} + \mathcal{S})$ is the weakest failure detector to implement fault-tolerant mutual exclusion in *any* environment. Thus far, we have shown that $\mathcal{T}$ is sufficient to implement $\triangledown\mathcal{AF}$ in a majority-correct environment and that $\triangledown\mathcal{AF}$ is sufficient to implement $\mathcal{T}$ in any environment. It is only natural, then, to wonder if $(\mathcal{T} + \mathcal{S})$ is sufficient to implement $\triangledown\mathcal{AF}$ in *any* environment.

5.2.1 Implementing $\triangledown\mathcal{AF}$ using $(\mathcal{T} + \mathcal{S})$

Recall that in Algorithm 1, the requirement for a majority of correct processes was utilized during Action 2 to guarantee some correct "witness" process for any process that is appended to the shared list $ordering$. In this manner, crashed processes will invariably be removed from $ordering$

when they are untrusted by some "witness" process, preventing the system from becoming blocked. Now consider some system that has access to the strong failure detector in addition to the trusting failure detector. If the precondition for Action 2 is replaced with $trust_i \supseteq (\Pi - \mathcal{S})$, then we have by the weak accuracy property of $\mathcal{S}$ that i will have some correct witness before appending itself to $ordering$, even in some environment with a majority of process crashes. Thus, we have that the failure detector $(\mathcal{T} + \mathcal{S})$ is sufficient to implement $\nabla\mathcal{AF}$ in any environment.

Before continuing, it is important to highlight that the use of shared data structures in Algorithm 1 is still feasible in a system augmented with $(\mathcal{T} + \mathcal{S})$, even in a minority-correct environment. This is simply because $\mathcal{S}$ is capable of implementing consensus in any environment [8], preserving the validity of linearizable sequential data structures in a system augmented with $(\mathcal{T} + \mathcal{S})$ [15].

5.2.2 Implementing $(\mathcal{T} + \mathcal{S})$ using $\nabla\mathcal{AF}$

Although it is possible to implement $\nabla\mathcal{AF}$ using the failure detector $(\mathcal{T} + \mathcal{S})$, it is unclear how to implement $\mathcal{S}$ using $\nabla\mathcal{AF}$, and thereby to implement $(\mathcal{T} + \mathcal{S})$. The reasoning behind this difficulty is relatively straightforward: information is only gained using $\nabla\mathcal{AF}$ when a process takes an application step, but implementing $\mathcal{S}$ would require some process to be trusted prior to taking an application step. It appears as if is no way to satisfy this constraint other than simply trusting each process initially, which would prevent processes that crash prior to taking a step from becoming untrusted, violating strong completeness.

Despite this difficulty, the strong failure detector is capable of implementing the Some Fair ($\mathcal{SF}$) system model, which in turn is capable of implementing $\mathcal{S}$ [1]. Consequently, we have that the pair $(\mathcal{T} + \mathcal{S})$ can implement the system model ($\mathcal{SF}$ + $\nabla\mathcal{AF}$), and that the system model ($\mathcal{SF}$ + $\nabla\mathcal{AF}$) can in turn implement $(\mathcal{T} + \mathcal{S})$. This, unlike the relationship between $\mathcal{T}$ and $\nabla\mathcal{AF}$, implies that ($\mathcal{SF}$ + $\nabla\mathcal{AF}$) is the weakest system model to implement $(\mathcal{T} + \mathcal{S})$.

5.3 Future Work

In this section, we present a number of unresolved questions that remain after the results of this thesis.

5.3.1 Weakest System Model to Implement $\mathcal{T}$

Perhaps the most visible avenue in which future effort can be directed is the development of an algorithm capable of implementing $\nabla\mathcal{AF}$ in any environment. Such a development would demonstrate that $\nabla\mathcal{AF}$ is, in fact, the weakest system model capable of implementing $\mathcal{T}$.

5.3.2 Fairness and the Quorum Failure Detector

The quorum failure detector (Σ) is a failure detector of no little significance in the field of distributed computing. As mentioned in Section 1, the quorum failure detector was shown to be the weakest failure detector to implement a shared register [12] and thereby serves as a critical link between shared memory and message passing distributed systems. Additionally, the failure detector $(\Sigma, \mathcal{S})$ is the weakest failure detector to implement consensus in any environment. Clearly, capturing the fairness properties encapsulated by the quorum failure detector would provide significant insights into the relationship between shared memory and message passing as well as providing the weakest system model capable of implementing consensus.

5.3.3 Minimalistic Scheduler Implementation

Although the scheduler shown in Algorithm 1 is correct, its specific implementation is more tailored towards demonstrating a theoretical concept rather than towards efficiency of the underlying message passing system. The use of shared memory constructs in Algorithm 1 may quickly become burdensome and unwieldy, as each data structure depends on a consensus protocol to maintain synchronization between processes. To this end, we have constructed a second, more minimalistic scheduler that does not depend on shared memory constructs. Currently, however, the correctness of this scheduler has not been proven. For the interested reader, this scheduler and a brief description are included in the appendix.

5.3.4 Fault Patterns and Fairness

As demonstrated in [13], the trusting failure detector is capable of implementing fault-tolerant mutual exclusion in a majority-correct environment, but to implement FTME in *any* environment, the trusting failure detector must be supplemented with the strong failure detector.

This subtle distinction carries with it the interesting implication that fault patterns may be relatable to fairness properties. Intuitively, it would appear that a majority of correct processes provides some amount of fairness weaker than $\mathcal{SF}$ and stronger than $\Diamond\mathcal{SF}$. This intuition is captured in the fact that ($\mathcal{SF}$ + $\triangledown\mathcal{AF}$) is the weakest system model to implement $(\mathcal{T} + \mathcal{S})$, while $\triangledown\mathcal{AF}$ *may* be the weakest system model to implement $\mathcal{T}$. More research is needed, however, to draw any definite conclusions in this area.

REFERENCES

[1] S. M. Pike, S. Sastry, and J. L. Welch, “Failure detectors encapsulate fairness,” *Distributed Comput.*, vol. 25, no. 4, pp. 313–333, 2012.

[2] M. J. Fischer, N. A. Lynch, and M. Paterson, “Impossibility of distributed consensus with one faulty process,” *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[3] F. Cristian and C. Fetzer, “The timed asynchronous distributed system model,” *IEEE Trans. Parallel Distributed Syst.*, vol. 10, no. 6, pp. 642–657, 1999.

[4] P. Robinson and U. Schmid, “The asynchronous bounded-cycle model,” *Theor. Comput. Sci.*, vol. 412, no. 40, pp. 5580–5601, 2011.

[5] P. M. B. Vitányi, “Distributed elections in an archimedean ring of processors,” *CoRR*, vol. abs/0906.0731, 2009.

[6] P. M. B. Vitányi, “Distributed elections in an archimedean ring of processors (preliminary version),” in *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA* (R. A. DeMillo, ed.), pp. 542–547, ACM, 1984.

[7] D. Dolev, C. Dwork, and L. J. Stockmeyer, “On the minimal synchronism needed for distributed consensus,” *J. ACM*, vol. 34, no. 1, pp. 77–97, 1987.

[8] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[9] Y. Song, *The Weakest Failure Detector for Solving Wait-Free, Eventually Bounded-Fair Dining Philosophers*. PhD thesis, Texas A&M University, College Station, USA, 2010.

[10] R. Guerraoui, M. Kapalka, and P. Kouznetsov, “The weakest failure detectors to boost obstruction-freedom,” *Distributed Comput.*, vol. 20, no. 6, pp. 415–433, 2008.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *J. ACM*, vol. 43, no. 4, pp. 685–722, 1996.

[12] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui, "Shared memory vs message passing," tech. rep., EPFL, Lausanne, 12 2003.

[13] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov, "Mutual exclusion in asynchronous systems with failure detectors," *J. Parallel Distributed Comput.*, vol. 65, no. 4, pp. 492–505, 2005.

[14] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg, "The weakest failure detectors to solve certain fundamental problems in distributed computing," in *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004* (S. Chaudhuri and S. Kutten, eds.), pp. 338–346, ACM, 2004.

[15] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

APPENDIX: MINIMALISTIC SCHEDULER

Algorithm Description

Algorithm 4 shows the scheduler running locally at each process i. The scheduler utilizes the trusted failure detector to provide the active fairness property to the system of processes, and is implemented as a hybrid combining the communication logic found in [1] and the execution procedures found in [13]. On a high level, the scheduler functions as a distributed round-robin. As each process begins with $state = waiting$ and application steps can only be taken in Action 7 when $state = active$, initially no process in the system is able to take a step. During this phase, the process may simply send and receive messages while waiting to accumulate $\lfloor n/2 \rfloor + 1$ permits from other processes. Processes may only send permits to other processes upon the other process becoming trusted by the local failure detector module. Note that as the system must contain a majority of correct processes, this implies that upon some process i obtaining $\lfloor n/2 \rfloor + 1$ permits, i has been trusted by at least 1 correct process. This fact serves to make the scheduler non-blocking - if some process i starts taking application steps and later crashes, we have by the trusting accuracy property of the trusting failure detector that i will eventually be untrusted by whatever correct process initially trusted it, allowing knowledge of i's crash to be broadcast throughout the system and preventing halting. In essence, through requiring a majority of permits and a majority of correct processes, we can provide a correct "witness" for each executing process.

Once a process obtains a majority of permits, it may broadcast to all processes that it is ready execute. Upon receiving any such broadcast, each process will add the broadcasting process to its local variable $pool$ and wait on the broadcasting process by proxy of the local variable $waitlist$ before taking a step. In this fashion, each newly broadcasting process is added to the round-robin protocol in a way that preserves process execution fairness.

Application to application communication in Algorithm 4 is performed in a nearly identical manner to that of the scheduler in [1]. When a process i becomes $active$, it sends a request to receive messages to all processes and waits for a response from each process in $i.pool$. In this fashion, i is guaranteed to have received all relevant messages from any process that has taken a step prior to when i becomes active and is still live.

Total-Order Broadcast

As shown in [13], the trusting failure detector is strictly stronger than $\Diamond P$ in any environment. Consequently, with either a majority of correct processes and the $\mathcal{T}$ or the pair $(\mathcal{T}, \mathcal{S})$, we can implement a total order broadcast subroutine [8]. We denote this routine by $broadcast_{T.O.}$, where messages can be received through $receive_{T.O.}$. It is assumed that $receive_{T.O.}$ is a *state* rather than an *action*, that is, $receive_{T.O.}$ is set at some process i whenever a message resulting from a $broadcast_{T.O.}$ is ready to be received by i. The total ordered broadcast subroutine satisfies the following properties:

- If some process runs $broadcast_{T.O.}(m)$, then eventually every correct process will $receive_{T.O.}(m)$.
- Some process may only $receive_{T.O.}(m)$ if some process has previously run $broadcast_{T.O.}(m)$, and $broadcast_{T.O.}(m)$ will be received at most once at each process.
- If some process receives some message m at t and some message m' at t' where $t < t'$, no other process may receive m' without having already received m.

Algorithm 4 Actions for scheduler at process i

```
1: enum {$waiting, active$} : state ← $waiting$          ▷ Initially set state variable to waiting
2: **integer** seq ← 0                                  ▷ Sequence number to track messages/steps
3: **set** trusted ← ∅                                  ▷ Processes that trust $i$
4: **set** pool ← ∅                                     ▷ Currently queued executions in the system
5: **set** waitlist ← ∅
6: **set** crashed ← ∅                                  ▷ Local view of processes known to have crashed
7: **set** $\mathcal{T}_0$ ← ∅                          ▷ Prior value of $\mathcal{T}$, used to determine changes over time
8: **for all** $j \in \Pi$ **do**
9:     **integer** maxAck$_j$ ← 0                       ▷ Maximum $seq$ in messages from $j$
10:    **set** sendBuffer$_j$ ← ∅                       ▷ Local send buffer for messages sent to $j$
11:    **set** receiveBuffer$_j$ ← ∅                    ▷ Local receive buffer for messages from $j$
```

```
12: {upon $\mathcal{T} \neq \mathcal{T}_0$}                                    ▷ Action 1
13:     for all $j \in \mathcal{T} - \mathcal{T}_0$ do                 ▷ There are new elements in $\mathcal{T}$
14:         send($trust$) to $j$                       ▷ Send permits to newly trusted processes
15:     for all $k \in \mathcal{T}_0 - \mathcal{T}$ do                         ▷ Processes have been untrusted
16:         for all $j \in \Pi$ do
17:             send($crashed, k$) to $j$        ▷ Inform other processes in system of crash
18:     $\mathcal{T}_0 \leftarrow \mathcal{T}$                                            ▷ Reset value of $\mathcal{T}_0$
```

```
19: {upon receive $(trust)$ from $j$}                                   ▷ Action 2
20:     trusted $\leftarrow$ trusted $\cup j$
21:     if $|trusted| = \lfloor n/2 \rfloor + 1$ then        ▷ $i$ is trusted by at least 1 correct process
22:         $broadcast_{T.O.}(ready)$                               ▷ Queue for new execution
```

```
23: {upon receive $(crashed, k)$ from $j$}                              ▷ Action 3
24:     crashed $\leftarrow$ crashed $\cup \{k\}$                          ▷ Add to local set crashed
25:     pool $\leftarrow$ pool $- \{k\}$
26:     waitlist $\leftarrow$ waitlist $- \{k\}$
```

```
27: {upon receive$_{\mathbf{T.O.}}(ready)$ from $j$}                           ▷ Action 4
28:     if $j \notin crashed$ then
29:         pool $\leftarrow$ pool $\cup \{j\}$
30:     if $j = i$ then
31:         waitlist $\leftarrow$ pool
```

```
32: {upon $state = waiting \wedge waitlist = \{i\}$}                         ▷ Action 5
33:     state ← $active$
34:     $seq \leftarrow seq + 1$                          ▷ Generate new sequence number
35:     for all $j \in \Pi - \{i\}$ do
36:         send($getMsg, seq$) to $j$        ▷ Send message requests tagged with $i.seq$
```

```
37: {(state = $active$) $\wedge$ ($\forall j \in \Pi - \{i\}$ :: ((maxAck$_j$ = seq) $\vee$ ($j \notin$ pool)))}   ▷ Action 6
38:     $executeAPP()$              ▷ Allow application to take enabled application step
39:     waitlist ← pool
40:     state ← $waiting$     ▷ Return state to $waiting$ so $i$ cannot immediately execute again
41:     for all $j \in waitlist - \{i\}$ do
42:         send($stepTaken$) to $j$
```

```
43: {upon receive ($stepTaken$) from $j$}                                     ▷ Action 7
44:     waitlist ← waitlist $- j$
```

```
45: {upon receive ($getMsg, num$) from $j$}                                   ▷ Action 8
46:     $S$ ← sendBuffer$_j$                      ▷ Populate $S$ with buffered messages
47:     sendBuffer$_j$ ← $\emptyset$                                 ▷ Empty send buffer
48:     send($S, num$) to $j$                   ▷ Send $S$ to $j$ tagged with seq number
```

```
49: {upon receive ($S', num$) from $j$}                                       ▷ Action 9
50:     receiveBuffer$_j$ ← receiveBuffer$_j$ $\cup S'$       ▷ Add $S'$ to local receive buffer
51:     maxAck$_j$ ← $max(num,$ maxAck$_j)$        ▷ Update $maxAck_j$ to largest seq so far
```

Algorithm 5 Interaction between the scheduler and the application at process i

```
 1: procedure EXECUTEAPP
 2:     RECEIVEAPP()
 3:     Execute an enabled application step
 4:     Application step invokes SENDAPP(m,j) to send message $m$ to process $j$.
 5: procedure RECEIVEAPP
 6:     $returnValue \leftarrow \cup_{\forall j \in \Pi - \{i\}}\{(i.\text{receiveBuffer}_j, j)\}$
 7:     for all $j \in \Pi - \{i\}$ do
 8:         $i$.receiveBuffer$_j$ ← $\emptyset$
 9:     return $returnValue$
10: procedure SENDAPP($m, j$)
11:     $i$.sendBuffer$_j$ ← $i$.sendBuffer$_j$ $\cup \{m\}$
```