Asynchronous Failure Detectors

Alejandro Cornejo
Computer Science and Artificial Intelligence Lab, MIT
acornejo@csail.mit.edu

Nancy Lynch
Computer Science and Artificial Intelligence Lab, MIT
lynch@csail.mit.edu

Srikanth Sastry
Computer Science and Artificial Intelligence Lab, MIT
sastry@csail.mit.edu

ABSTRACT

Failure detectors — oracles that provide information about process crashes — are an important abstraction for crash tolerance in distributed systems. Although current failure-detector theory provides great generality and expressiveness, it also poses significant challenges in developing a robust hierarchy of failure detectors. We address some of these challenges by proposing a variant of failure detectors called *asynchronous failure detectors* and an associated modeling framework. Unlike the traditional failure-detector framework, our framework eschews real time completely. We show that asynchronous failure detectors are sufficiently expressive to include several popular failure detectors. Additionally, we show that asynchronous failure detectors satisfy many desirable properties: they are self-implementable, guarantee that stronger asynchronous failure detectors solve more problems, and ensure that their outputs encode no information other than process crashes. We introduce the notion of a failure detector being *representative* of a problem to capture the idea that some problems encode the same information about process crashes as their weakest failure detectors do. We show that a large class of problems, called *finite problems*, do not have representative failure detectors.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*computability theory*

General Terms

Algorithms, Reliability, Theory

Keywords

Asynchronous System, Fault-Tolerance, Asynchronous Failure Detector, I/O Automata

PODC'12, July 16–18, 2012, Madeira, Portugal.

1. INTRODUCTION

Failure detectors [5] are a popular mechanism for designing asynchronous distributed algorithms for crash-prone systems. Conceptually, they provide (potentially unreliable) information about process crashes in the system. This information may be leveraged by asynchronous algorithms for crash tolerance. Technically, failure detectors are specified by constraints on their possible outputs, called *histories*, relative to the actual process crashes in the system, called the *fault pattern*. The fault pattern is the 'reality', and the history is an 'approximation' of that reality. A failure detector is a function that maps every fault pattern (the 'reality') to a set of admissible histories (the 'approximations'). The *stronger* a failure detector is, the closer its admissible 'approximations' are to the 'reality'.

We explore the modeling choices made in the traditional failure-detector framework, and we focus on a variant of failure detectors called *asynchronous failure detectors*. We also offer an alternative modeling framework to study the properties of asynchronous failure detectors. Briefly, asynchronous failure detectors are a variant of failure detectors that can be specified without the use of real time, are self-implementable, and interact with the asynchronous processes *unilaterally*; in unilateral interaction, the failure detector provides outputs to the processes continually without any queries from the processes. We show that asynchronous failure detectors retain sufficient expressiveness to include many popular and *realistic* [7] failure detectors while satisfying several desirable properties.

1.1 Background and Motivation

The canonical works [5, 4] pioneered the theory of failure detectors. Results in [5] showed how *sufficient* information about process crashes can be encoded in failure detectors to solve problems in asynchronous systems. Complementary work in [4] showed that some information about crashes is actually *necessary*; in particular, they showed that Ω is a "weakest" failure detector to solve consensus in crash-prone asynchronous systems. Their proposed proof technique has been used to demonstrate weakest failure detectors for many problems in crash-prone asynchronous systems (cf. [8, 24, 11, 14]). Recent results have shown that a large class of problems have a weakest failure detector [17] while yet another class of problems do not have a weakest failure detector [3].

From a modeling perspective, failure detectors mark a departure from conventional descriptions of distributed systems. Conventionally, the behavior of all the entities in a distributed system model — processes, channels, and other

entities — are either all asynchronous or are all constrained by the passage of real time. In contrast, in the failure-detector model, only the failure-detector behavior is constrained by real time, whereas the behavior of all other entities is asynchronous. The differences between the two styles of models have been the subject of recent work [6, 17] which has brought the theory of failure detectors under additional scrutiny. We discuss five aspects of failure-detector theory that remain unresolved: self-implementability, interaction mechanism, the kind of information provided by a failure detector, comparing failure-detector strengths, and the relationship between weakest failure detectors and partial synchrony.

Self-Implementability. Failure detectors need not be *self-implementable*. That is, there exist failure detectors (say) $\mathcal{D}$ such that it is not possible for any asynchronous distributed algorithm to implement an admissible behavior of $\mathcal{D}$ despite having access to the outputs from $\mathcal{D}$. Since a failure detector D' is stronger then a failure detector D iff D' can implement D, we arrive at an unexpected result that a failure detector D need not be comparable to itself.

Jayanti et. al. resolve the issue of self-implementability in [17] by separating the notion of a failure detector from an *implementation of a failure detector*. A failure detector provides outputs to each process at each time instant, but a failure-detector implementation provides outputs only upon being queried. An implementation of a failure detector $\mathcal{D}$ is said to be correct if, for every query, the output of the implementation is a valid output of $\mathcal{D}$ for some time in the interval between the query and the output. In effect, the definition of "implementing a failure detector" in [17] collapses multiple classes of distinct failure detectors into a single equivalence class.[1] The broader impact of results from [17] on the landscape of failure-detector theory remains unexplored.

Interaction Mechanism. The mechanism proposed in [17] explicitly requires that failure-detector implementations interact with processes via a query-based interface. An alternative interface is one in which failure-detector implementations provide outputs to processes unilaterally and continually, without queries. To our knowledge, the motivation for choosing either interface has not been adequately elucidated despite non-trivial consequences of the choice. For instance, recall that self-implementability of a failure detector in [17] depends critically on the query-based interface. Also, the so-called 'lazy' implementations of failure detectors [10] depend on a query-based interface to ensure communication efficiency; an analogous optimization is not known with a unilateral interface. Therefore, the significance and consequences of the interaction model merit investigation.

Information About Crashes Alone. Whether or not failure detectors can provide information about events other than process crashes has a significant impact on the weakest failure detectors for problems such as Non-Blocking Atomic Commit [14, 15] and Uniform Reliable Broadcast [1, 16]. In order to restrict failure detectors to the ones that give information only about crashes, the authors in [1] consider failure detectors that are exclusively a function of the fault pattern. In [14], the authors further restrict the universe of failure detectors to *timeless* failure detectors, which provide information only about the set of processes that crash, and no information about *when* they crash. To our knowledge, the necessary and sufficient conditions for failure detectors to provide information about crashes alone remains unresolved.

Comparing Failure Detectors. Not all information provided by failure detectors may be useful in an asynchronous system. For instance, if a failure detector provides the current real-time in its outputs (in addition to other information), the processes cannot use this information because passage of real time is not modeled in an asynchronous system. Suppose we consider two failure detectors D and D' where D is timeless, and D' provides all the information provided by D; additionally D' provides the current real time as well. Clearly, D' is strictly stronger than D. However, since the asynchronous system cannot use the information about real time provided by D', there exist no problems that can be solved in an asynchronous system with D', but that cannot be solved with D. This leads to a curious conclusion: there exist failure detectors (say) D and D' such that D' is strictly stronger than D, and yet D' cannot solve a harder problem than D. This begs the following question: what does the relative strength of failure detectors tell us about the relative hardness of problems they solve?

Weakest Failure Detectors and Partial Synchrony. Failure detectors are often viewed as distributed objects that encode information about the temporal constraints on computation and communication necessary for their implementation; the popular perception is that several failure detectors are substitutable for partial synchrony in distributed systems [19, 21, 20]. Therefore, if a failure detector D is the weakest to solve a problem P, then a natural question follows: is the synchronism encoded in the outputs of D the minimal synchronism necessary to solve P in a crash-prone partially synchronous system? Work to date suggests that the answer is affirmative for some problems [19, 22] and negative for others [6]. To our knowledge, there is no characterization of the problems for which the aforementioned question is answered in the affirmative or in the negative.

Summary. Based on our understanding of the state of the art, we see that failure-detector theory is a very general theory of crash tolerance with important results and novel methods. These results and methods provide a qualitative understanding of the amount of information about crashes necessary and sufficient to solve various problems in asynchronous systems. However, the generality of the theory makes it difficult to develop a robust hierarchy of failure detectors and to determine the relative hardness of solving problems in crash-prone asynchronous systems.

1.2 Contribution

In this paper, we examine a new variant of failure detectors called *asynchronous failure detectors* (AFDs) and we show that they satisfy many desirable properties. We define AFDs through a set of basic properties that we expect any "reasonable" failure detector to satisfy. We demonstrate the expressiveness of AFDs by defining many traditional failure detectors as AFDs. Restricting our focus to AFDs offers several advantages.

[1] For example, consider the instantaneously perfect failure detector $\mathcal{P}^+$ [6] which always outputs the exactly the set of crashed processes and the perfect failure detector $\mathcal{P}$ [5] which never suspects live processes and eventually and permanently suspects crashed processes. Under the definition of "implementing a failure detector" from [17], an implementation of $\mathcal{P}^+$ is indistinguishable from an implementation of $\mathcal{P}$.

First, AFDs are self-implementable and their specification does not require real time. Therefore, unlike current failure-detector models, all the entities in the distributed system are asynchronous. In order to specify AFDs, we propose a new modeling framework that completely eschews real time, which allows us to view failure detectors as problems within the asynchronous model. This allows us to compare failure detectors as we compare problems; it also allows us to compare problems with failure detectors, and vice versa.

Second, AFDs provide outputs to the processes unilaterally, without queries. Therefore we preserve the advantages offered by the framework in [17] while ensuring failure detectors provide information only about process crashes.

Third, the hierarchy of AFDs ordered by their relative strength induces an analogous hierarchy of problems ordered by their relative hardness. In fact, if an AFD D is strictly stronger than another AFD D', then we show that the set of problems solvable with D is a strict superset of the set of problems solvable by D.

Fourth, AFDs clarify a relationship between a weakest failure detector to solve a problem and the minimal synchronism that is necessary and sufficient to solve the same problem. We introduce the concept of representative AFDs for a problem. Briefly, an AFD D is "representative" of a problem P iff D is sufficient to solve P and D can be extracted from a (blackbox) solution to P. By construction, the synchronism encoded by the outputs of a representative AFD for a problem P is also the minimal synchronism sufficient to solve P. We show that finite problems (such as consensus and set agreement) do not have a representative AFD, but they have a weakest failure detector [17].

2. I/O AUTOMATA

We use the I/O Automata framework [18] for specifying the system model and failure detectors. Briefly, in the I/O framework each component of a distributed system is modeled as a state machine, where different components interact with each other through input and output actions. This section provides an overview of I/O-Automata-related definitions used in this paper. See [18, Chapter 8] for a thorough description of the I/O Automata framework.

2.1 Definitions

An I/O automaton (or simply, an automaton) is a (possibly infinite) state machine. Formally, an I/O automaton consists of five components: a signature, a set of states, a set of initial states, a state-transition relation, and a set of tasks. We describe these components next.

Actions, Signature, and Tasks. The state transitions in an automaton are associated with named *actions*; the set of actions of an automaton A is denoted $act(A)$. Actions are classified as *input*, *output*, or *internal*, and they constitute the *signature* of the automaton. The set of input, output, and internal actions of an automaton A are denoted $input(A)$, $output(A)$, and $internal(A)$, respectively. Input and output actions are collectively called *external* actions, and output and internal actions are collectively called *locally controlled* actions. The locally controlled actions of an automaton are partitioned into *tasks*.

Internal actions of an automaton are visible only to the automaton itself whereas external actions are visible to other automata as well; automata interact with each other through external actions. Unlike locally controlled actions, input actions arrive from the outside and are assumed not to be under the automaton's control.

States. The set of states of an automaton A is denoted $states(A)$. A non-empty subset $init(A) \subseteq states(A)$ is designated to be the set of *initial states*.

State-Transition relation. The state transitions in an automaton A are restricted by a *state-transition relation*, denoted $trans(A)$, which is a set of tuples of the form (s, a, s') where $s, s' \in states(A)$ and $a \in act(A)$. Each such tuple (s, a, s') is a *transition*, or a *step*, of A.

For a given state s and an action a, if $trans(A)$ has some step of the form (s, a, s'), then a is said to be *enabled* in s. Every input action in A is enabled in all the states of A. A task C, which consists of a set of locally controlled actions, is said to be *enabled* in a state s iff some action in C is enabled in state s.

Intuitively, each step of the form (s, a, s') denotes the following behavior: the automaton A, in state s, performs action a and changes its state to s'.

2.2 Executions And Traces

Now we describe how an automaton executes. An *execution fragment* of an automaton A is a finite sequence $s_0, a_1, s_1, a_2, \ldots, s_{k-1}, a_k, s_k$, or an infinite sequence $s_0, a_1, s_1, a_2, \ldots, s_{k-1}, a_k, s_k, \ldots$, of alternating states and actions of A such that for every $k \geq 0$, action a_{k+1} is enabled in state s_k. An execution fragment that starts with an initial state is called an *execution*. Each occurrence of an action in an execution fragment is said to be an *event*.

A *trace* of an execution denotes only the externally observable behavior. Formally, the trace t of an execution α is the subsequence of α consisting of all the external actions. We say that t is a trace of an automaton A if t is the trace of some execution of A. When referring to specific events in a trace, we use the following convention: if t contains at least x events, then $t[x]$ denotes the x^{th} event in the trace t, and otherwise, $t[x] = \perp$. Throughout this article, we assume that no action is named $\perp$.

It is useful to consider subsequences of traces that contain only certain events. We accomplish this through the notion of a *projection*. Given a sequence of actions t and a set of actions B, the projection of t over B, denoted $t|_B$, is the subsequence of t consisting of exactly the events from B.

2.3 Composing I/O Automata

A collection of I/O automata may be composed by matching output actions of some automata with the same-named input actions of others. Specifically, each output of an automaton may be matched with same-named input of any number of other automata. Upon composition, all the actions with the same name are performed together.

2.4 Fairness

When considering executions of a composition of I/O automata, we are interested in the executions in which all the automata get fair turns to perform steps; such executions are called fair executions.

Recall that in each automaton, the locally controlled actions are partitioned into tasks. An execution fragment α of an automaton A is said to be a *fair execution fragment* iff the following two conditions hold for every task C in A. (1) If α is finite, then no action in C is enabled in the final state of α. (2) If α is infinite, then either (a) α contains

infinitely many events from C, or (b) α contains infinitely many occurrences of states in which C is not enabled.

A trace t of A is said to be a *fair trace* if t is the trace of a fair execution of A.

2.5 Deterministic Automata

We define an action a (of an automaton A) to be *deterministic* iff for every state s, there exists at most one transition of the form (s, a, s') in $trans(A)$. We define an automaton A to be *task deterministic* iff (1) for every task C and every state s of A, at most one action in C is enabled in s, and (2) all the actions in A are deterministic. An automaton is said to be *deterministic* iff it is task deterministic, has exactly one task, and has a unique start state.

3. CRASH PROBLEMS

This section provides definitions of problems, distributed problems, crashes, crash problems and asynchronous failure detectors.

3.1 Problems

A *problem* P is a tuple (I_P, O_P, T_P) where I_P and O_P are disjoint sets of actions and T_P is a set of (finite or infinite) sequences over these actions.

Distributed Problems. Here, we introduce a fixed finite set Π of n location IDs; we assume that Π does not contain the element $\perp$.

For a problem P, we define a mapping $loc : I_P \cup O_P \rightarrow \Pi \cup \{\perp\}$ which associates an action to a location ID or $\perp$. For an action a, if $loc(a) = i$ and $i \in \Pi$, then a is said to *occur at* i. Problem P is said to be *distributed* over Π if, for every action $a \in I_P \cup O_P$, $loc(a) \in \Pi$.

For convenience, the location of each action is included in the name of the action as a subscript; for instance, if an action a occurs at i, then the action is named a_i.

Crash Problems. We posit the existence of a set of actions $\{crash_i | i \in \Pi\}$, denoted $\hat{I}$; according to our conventions $loc(crash_i) = i$. A problem $P \equiv (I_P, O_P, T_P)$ that is distributed over Π, is said to be a *crash problem* iff, for each $i \in \Pi$, $crash_i$ is an action in I_P; that is, $\hat{I} \subseteq I_P$.

Given a sequence $t \in T_P$, $faulty(t)$ denotes the set of locations at which a $crash$ event occurs in t. Similarly, $live(t)$ denotes the set of locations for which a $crash$ event does not occur in t. The locations in $faulty(t)$ are said to be *faulty* in t, and the locations in $live(t)$ are said to be *live* in t.

For convenience, we assume that for any two distinct crash problems $P \equiv (I_P, O_P, T_P)$ and $P' \equiv (I_{P'}, O_{P'}, T_{P'})$, $(I_P \cup O_P) \cap (I_{P'} \cup O_{P'}) = \hat{I}$. The foregoing assumption simplifies the issues involving composition of automata; we discuss these in Section 5.

3.2 Asynchronous Failure Detectors

Recall that a failure detector is an oracle that provides information about crash failures. In our modeling framework, we view failure detectors as a special type of crash problems and are called *asynchronous failure detectors*. A necessary condition for a crash problem $P \equiv (I_P, O_P, T_P)$ to be an asynchronous failure detector is *crash exclusivity*, which states that $I_P = \hat{I}$; that is, the actions I_P are exactly the $crash$ actions. Crash exclusivity guarantees that the only inputs to a failure detector are the $crash$ events, and hence, failure detectors provide information only about crashes. An asynchronous failure detector also satisfies additional properties, but before describing these properties formally we need some auxiliary definitions.

Let $D \equiv (\hat{I}, O_D, T_D)$ be a crash problem. For each $i \in \Pi$, $\mathcal{F}_i$ is the set of actions in O_D at i; thus, $O_D = \cup_{i \in \Pi} \mathcal{F}_i$. We begin by defining the following terms. Let t be an arbitrary sequence over $\hat{I} \cup O_D$.

Valid sequences. The sequence t is said to be *valid* iff (1) for every $i \in \Pi$, no event in O_D occurs at i after a $crash_i$ event in t, and (2) if no $crash_i$ event occurs in t, then t contains infinitely many events in O_D at i.

Sampling. A sequence t' is a *sampling* of t iff (1) t' is a subsequence t, (2) for every location $i \in \Pi$, (a) if i is live in t, then $t'|_{\mathcal{F}_i} = t|_{\mathcal{F}_i}$, and (b) if i is faulty in t, then i is faulty in t' and $t'|_{\mathcal{F}_i}$ is a prefix of $t|_{\mathcal{F}_i}$.

Constrained reordering. Let t' be a permutation of events in t; t' is constrained reordering of t iff, for every pair of events e and e', if (1) e precedes e' in t and (2) either $loc(e) = loc(e')$, or $e \in \hat{I}$, then e precedes e' in t' as well.

Now we define an asynchronous failure detector. A crash problem of the form $D \equiv (\hat{I}, O_D, T_D)$ (which satisfies crash exclusivity) is an *asynchronous failure detector* (AFD, for short) iff D satisfies the following properties.

Validity. Every sequence $t \in T_D$ is valid.

Closure Under Sampling. For every sequence $t \in T_D$, every sampling of t is in T_D.

Closure Under Constrained Reordering. For every sequence $t \in T_D$, every constrained reordering of t is in T_D.

A brief motivation for the above properties is in order. The validity property ensures that after a location crashes, no outputs occur at that location, and if a location does not crash, outputs occur infinitely often at that location. Closure under sampling permits a failure detector to 'skip' or 'miss' any suffix of outputs at a faulty location. Finally, closure under constrained reordering permits 'delaying' output events at any location.

3.3 Examples of AFDs

Here, we specify some of the failure detectors that are most widely used and cited in literature, as AFDs.

The Leader Election Oracle. Informally, Ω continually outputs a location ID at each location; eventually and permanently, Ω outputs the ID of a unique live location at all the live locations.

We specify our version of $\Omega \equiv (\hat{I}, O_\Omega, T_\Omega)$ as follows. The action set $O_\Omega = \cup_{i \in \Pi} \mathcal{F}_i$, where, for each $i \in \Pi$, $\mathcal{F}_i = \{FD\text{-}\Omega(j)_i | j \in \Pi\}$. T_Ω is the set of all valid sequences t over $\hat{I} \cup O_\Omega$ that satisfy the following property: *if* $live(t) \neq \emptyset$, *then there exists a location* $l \in live(t)$ *and a suffix* t_{suff} *of* t *such that,* $t_{suff}|_{O_\Omega}$ *is a sequence over the set* $\{FD\text{-}\Omega(l)_i | i \in live(t)\}$.

Perfect and Eventually Perfect Failure Detectors. Here we specify two popular failure detectors among the canonical failure detector from [5]: the perfect failure detector $\mathcal{P}$ and the eventually perfect failure detector $\Diamond\mathcal{P}$. Informally, $\mathcal{P}$ never suspects any location (say) i until event $crash_i$ occurs, and it eventually and permanently suspects crashed locations; $\Diamond\mathcal{P}$ eventually and permanently never suspects live locations and eventually and permanently suspects faulty locations.

We specify our version of $\mathcal{P} \equiv (\hat{I}, O_\mathcal{P}, T_\mathcal{P})$ as follows. The action set $O_\mathcal{P} = \cup_{i \in \Pi} \mathcal{F}_i$, where, for each $i \in \Pi$, $\mathcal{F}_i = \{FD\text{-}\mathcal{P}(S)_i | S \in 2^\Pi\}$. $T_\mathcal{P}$ is the set of all valid sequences

t over $\hat{I} \cup O_{\mathcal{P}}$ that satisfy the following two properties. (1) For every prefix t_{pre} of t, if $i \in live(t_{pre})$, then for each $j \in \Pi$ and for every event of the form $FD\text{-}\mathcal{P}(S)_j$ in t_{pre}, $i \notin S$. (2) There exists a suffix t_{sus} of t such that, for every $i \in faulty(t)$, for each $j \in \Pi$, and for every event of the form $FD\text{-}\mathcal{P}(S)_j$ in t_{sus}, $i \in S$.

We specify our version $\diamond\mathcal{P} \equiv (\hat{I}, O_{\diamond\mathcal{P}}, T_{\diamond\mathcal{P}})$ as follows. The action set $O_{\diamond\mathcal{P}} = \cup_{i\in\Pi}\mathcal{F}_i$, where, for each $i \in \Pi$, $\mathcal{F}_i = \{FD\text{-}\diamond\mathcal{P}(S)_i | S \in 2^{\Pi}\}$. $T_{\diamond\mathcal{P}}$ is the set of all valid sequences t over $\hat{I} \cup O_{\diamond\mathcal{P}}$ that satisfy the following two properties. (1) There exists a suffix t_{trust} of t such that, for every pair of locations $i, j \in live(t)$, and for every event of the form $FD\text{-}\diamond\mathcal{P}(S)_j$ in t_{trust}, $i \notin S$. (2) There exists a suffix t_{sus} of t such that, for every $i \in faulty(t)$, for each $j \in live(t)$, and for every event of the form $FD\text{-}\diamond\mathcal{P}(S)_j$ in t_{sus}, $i \in S$.

It is easy to see that $\Omega \equiv (\hat{I}, O_\Omega, T_\Omega)$, $\mathcal{P} \equiv (\hat{I}, \hat{O}, T_{\mathcal{P}})$ and $\diamond\mathcal{P} \equiv (\hat{I}, \hat{O}, T_{\diamond\mathcal{P}})$ satisfy all the properties of an AFD and the proof of the aforementioned assertion is left as an exercise for the reader. Similarly, it is straightforward to specify failure detectors like Ω^k and Ψ_k as AFDs.

4. SYSTEM MODEL AND DEFINITIONS

An asynchronous system is modeled as the composition of a collection of the following I/O automata: process automata, channel automata, a crash automaton, and possibly other automata (including a failure-detector automata).

Process Automata. The system contains a collection of n process automata: one process automaton at each location. Each process automaton is a deterministic automaton whose actions occur at a single location. A process automaton whose actions occur at i is denoted $proc(i)$. It has an input action $crash_i$ which is an output from the crash automaton; when $crash_i$ occurs, it permanently disables all locally controlled actions of $proc(i)$. The process automaton $proc(i)$ sends and receives messages through a set of output actions $\{send(m, j)_i | m \in \mathcal{M} \wedge j \in \Pi \backslash \{i\}\}$, and a set of input actions $\{receive(m, j)_i | m \in \mathcal{M} \wedge j \in \Pi \setminus \{i\}\}$, respectively. In addition, process automata may interact with the environment automaton and other automata through additional actions.

A *distributed algorithm* A is a collection of process automata, one at each location; for convenience, we write A_i for the process automaton $proc(i)$ at i.

Channel Automata. For every ordered pair (i, j) of distinct locations, the system contains a channel automaton $C_{i,j}$. The input actions are $\{send(m, j)_i | m \in \mathcal{M}\}$, which are outputs from the process automaton at i. The output actions are $\{receive(m, i)_j | \ m \in \mathcal{M}\}$, which are inputs to the process automaton at j. Each such channel automaton implements a *reliable FIFO link*.

Crash Automaton. The crash automaton contains the set $\{crash_i | i \in \Pi\} \equiv \hat{I}$ of output actions and no input actions. Every sequence over $\hat{I}$ is a fair trace of the crash automaton.

Environment Automaton. The environment automaton, denoted $\mathcal{E}$, models the external world with which the distributed system interacts. The external signature of the environment matches the input and output actions of the process automata that do not interact with other automata in the system. The set of fair traces that constitute the externally observable behavior of $\mathcal{E}$ specifies "well-formedness" restrictions, which vary from one system to another.

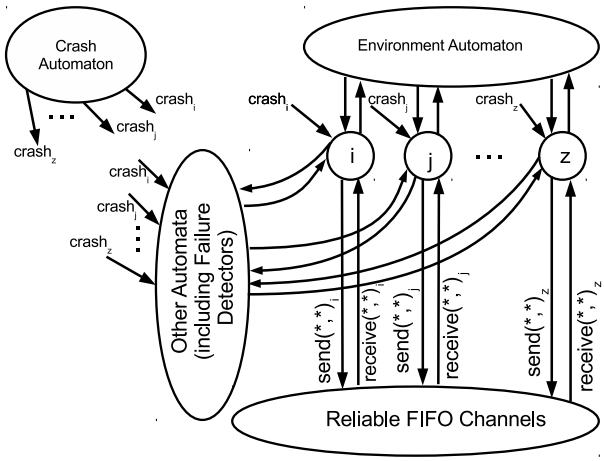

Figure 1: Interaction diagram for a message-passing asynchronous distributed system augmented with a failure detector automaton.

Other Automata. The system may contain other automata with which the process automata and the crash automaton interact. Typically, these automata *solve* a crash problem, as defined in the next section.

5. SOLVING PROBLEMS

In this section, we define what it means for an automaton to solve a crash problem and for a distributed algorithm to solve a crash problem. We also define what it means for a system to solve a crash problem P using another crash problem P'. We use the aforementioned definitions to define what it means for an AFD to be sufficient to solve a crash problem, and vice versa.

5.1 Solving a Crash Problem

An automaton U *solves* a crash problem $P \equiv (I_P, O_P, T_P)$ in an environment $\mathcal{E}$, if (1) the input actions of U are I_P, and the output actions of U are O_P, (2) the input actions of $\mathcal{E}$ are O_P, and the output actions of $\mathcal{E}$ are $I_P \setminus \hat{I}$, and (3) the set of fair traces of the composition of U, $\mathcal{E}$, and the crash automaton is a subset of T_P.

A distributed algorithm A *solves a crash problem* P *in an environment* $\mathcal{E}$ (or, A solves P in $\mathcal{E}$), iff the automaton $\hat{A}$, which is obtained by composing A with the channel automata, solves P in $\mathcal{E}$. A crash problem P is said to be *solvable* in an environment $\mathcal{E}$, iff there exists a distributed algorithm A such that A solves P in $\mathcal{E}$. If a crash problem is not solvable in $\mathcal{E}$, then it is said to be *unsolvable* in $\mathcal{E}$.

5.2 Using One Crash Problem to Solve Another

Often, an unsolvable crash problem P may be solvable in a system that contains an automaton that solves some *other* unsolvable crash problem P'. We describe the relationship between P and P' as follows.

A distributed algorithm A *solves a crash problem* P *using another crash problem* P' *in an environment* $\mathcal{E}$ (or succinctly, A solves P using P' in $\mathcal{E}$), iff the following is true. Let $\hat{A}$ be the composition of A with the channel automata,

the crash automaton, and the environment $\mathcal{E}$. For every fair trace t of $\hat{A}$, if $t|_{I_{P'} \cup O_{P'}} \in T_{P'}$, then $t|_{I_P \cup O_P} \in T_P$.

We say that a crash problem $P' \equiv (I_{P'}, O_{P'}, T_{P'})$ *is sufficient to solve* a crash problem $P \equiv (I_P.O_P, T_P)$, in environment $\mathcal{E}$, denoted $P' \succeq_{\mathcal{E}} P$ iff there exists a distributed algorithm A that solves P using P' in $\mathcal{E}$. If $P' \succeq_{\mathcal{E}} P$, then also we say that P *is solvable using* P' in $\mathcal{E}$. If no such distributed algorithm exists, then we state that P is *unsolvable* using P' in $\mathcal{E}$, and we denote it as $P' \not\succeq_{\mathcal{E}} P$.

It is worth noting that in the foregoing definition, the problems P and P' must be distinct in order for automata composition to be applicable. However, it is useful to consider problems that are "sufficient to solve themselves"; that is, given a crash problem P and an environment $\mathcal{E}$, it is useful to define the following relation: $P \succeq_{\mathcal{E}} P$. We do so using the notion of renaming.

5.2.1 Renaming and Self-Implementability

A crash problem $P' \equiv (I_{P'}, O_{P'}, T_{P'})$ is said to be a *renaming* of a crash problem $P \equiv (I_P, O_P, T_P)$ iff (1) $(I_P \cup O_P) \cap (I_{P'} \cup O_{P'}) = \hat{I}$, and there exist bijections $r_{IO} : I_P \cup O_P \to I_{P'} \cup O_{P'}$ and $r_T : T_D \to T_{D'}$ such that, (1) for each $a \in \hat{I}$, $r_{IO}(a) = a$, for each $a \in I_P \setminus \hat{I}$, $r_{IO}(a) \in I_{P'} \setminus \hat{I}$, for each $a \in O_P$, $r_{IO}(a) \in O_{P'}$, (2) for each action $a \in I_P \cup O_P$, $loc(a) = loc(r_{IO}(a))$, and (3) for each $t \in T_P$ and for each $x \in \mathbb{N}^+$, if $t[x] \neq \bot$, then $r_T(t)[x] = r_{IO}(t[x])$.

Now, we can define the solvability of a crash problem P using itself as follows. We say that a crash problem P is *self-implementable* in environment $\mathcal{E}$, denoted $P \succeq_{\mathcal{E}} P$, iff there exists a renaming P' of P such that $P \succeq_{\mathcal{E}} P'$.

5.3 Using and Solving AFDs

Since an AFD is simply a kind of crash problem, we have automatic definitions for the following notions. (1) A distributed algorithm A solves an AFD D in environment $\mathcal{E}$. (2) A distributed algorithm A solves a crash problem P using an AFD D in environment $\mathcal{E}$. (3) An AFD D is sufficient to solve a crash problem P in environment $\mathcal{E}$. (4) A distributed algorithm A solves an AFD D using a crash problem P in environment $\mathcal{E}$. (5) A crash problem P is sufficient to solve an AFD D in environment $\mathcal{E}$. (6) A distributed algorithm A solves an AFD D' using another AFD D in environment $\mathcal{E}$. (7) An AFD D is sufficient to solve another AFD D' in environment $\mathcal{E}$. (8) An AFD D is self-implementable in environment $\mathcal{E}$.

We remark that when we talk about solving an AFD, the environment $\mathcal{E}$ has no output actions because the AFD has no input actions except for $\hat{I}$, which are inputs from the crash automaton. Therefore, we have the following lemma.

LEMMA 1. *For a crash-problem P, an AFD D, and an environment $\mathcal{E}$, if $P \succeq_{\mathcal{E}} D$, then for any other environment $\mathcal{E}'$ with the same external signature as $\mathcal{E}$, $P \succeq_{\mathcal{E}'} D$.*

Consequently, when we refer to an AFD D being solvable using a crash problem (or an AFD) P, we generally omit the reference to the environment automaton and simply say that P is sufficient to solve D; we denote this relationship by $P \succeq D$. Analogously, when we refer to a D being unsolvable using P, we denote this relationship by $P \not\succeq D$.

Finally, if an AFD D is sufficient to solve another AFD D', then we state that D *is stronger than* D', and we denote that $D \succeq D'$. If $D \succeq D'$, but $D' \not\succeq D$, then we say that D is *strictly stronger* than D', and we denote that $D \succ D'$.

Next, we consider reflexivity of the $\succeq$ relation between AFDs. We show that for every AFD D, $D \succeq D$ must be true; that is, every AFD is self-implementable.

6. SELF-IMPLEMENTABILITY OF AFDS

Within the traditional definitions of failure detectors, it is well known that not all failure detectors self-implementable (see [6] for a detailed discussion). In contrast we show that every AFD is self-implementable. Recall that an AFD D is self-implementable, denoted $D \succeq D$, iff there exists a renaming D' of D such that $D \succeq D'$.

Algorithm For Self-Implementability. We provide a distributed algorithm $\mathcal{A}^{self}$ that demonstrates self implementability of an arbitrary AFD D. First, we fix an arbitrary AFD $D \equiv (\hat{I}, O_D, T_D)$. Let $D' \equiv (\hat{I}, O_{D'}, T_{D'})$ be a renaming of D. Let $r_{IO} : O_D \to O_{D'}$ and $r_T : T_D \to T_{D'}$ be the bijections that define the renaming. That is, for each $t \in T_D$ and for each $x \in \mathbb{N}^+$, if $t[x] \neq \bot$, then $r_T(t)[x] = r_{IO}(t[x])$. The $\mathcal{A}^{self}$ automaton leverages the information provided by AFD D to solve D'.

The distributed algorithm $\mathcal{A}^{self}$ is a collection of automata $\mathcal{A}_i^{self}$, one for each location $i \in \Pi$. Each automaton $\mathcal{A}_i^{self}$ has the following signature. (1) An input action $crash_i$ which is the output action from the crash automaton. (2) The set of input actions $\mathcal{F}_i = \{d | d \in O_D \wedge (loc(d) = i)\}$ which are outputs of the failure-detector automaton D. (3) The set of output actions $\mathcal{F}'_i = \{r_{IO}(d) | d \in \mathcal{F}_i\}$.

At each location i, $\mathcal{A}_i^{self}$ maintains a queue fdq of elements from the range O_D; fdq is initially empty. When event $d \in \mathcal{F}_i$ occurs at location i, $\mathcal{A}_i^{self}$ adds d to the queue fdq. The precondition for action $d' \in \mathcal{F}'_i$ at i is that the head of the queue fdq at i is $r_{IO}^{-1}(d')$. When this precondition is satisfied, and event d' occurs at i, the effect of this event is to remove $r_{IO}^{-1}(d')$ from the head of fdq. Finally, when event $crash_i$ occurs, the effect of this event is to disable the output actions $\mathcal{F}'_i$ permanently. The pseudocode for $\mathcal{A}^{self}$ is available in Algorithm 1.

Algorithm 1 Algorithm for showing self-implementability of asynchronous failure-detector.

The automaton $\mathcal{A}_i^{self}$ at each location i.
Signature:
  input $d_i : O_D$ at location i, $crash_i$
  output $d'_i : O_{D'}$ at location i
Variables:
  fdq: queue of elements from O_D, initially empty
  $failed$: Boolean, initially $false$
Actions:
  input $crash$
  effect
    $failed := true$
  input d
  effect
    add d to fdq
  output d'
  precondition
    $(\neg failed) \wedge (fdq \text{ not empty}) \wedge (r_{IO}^{-1}(d') = head(fdq))$
  effect
    delete head of fdq

Correctness. The proof of correctness follows from closure under sampling and closure under constrained reordering, but is omitted due to space constraints.

THEOREM 2. *The distributed algorithm $\mathcal{A}^{self}$ uses AFD D to solve a renaming of D.*

From Theorem 2 we have the following as a corollary.

COROLLARY 3. *Every AFD is self-implementable: for every AFD D, $D \succeq D$.*

An immediate consequence of Corollary 3 is that we can take the union of the relation $\succeq$ between distinct AFDs and the $\succeq$ relation comparing an AFD and claim that the $\succeq$ relation is transitive. This is captured in the following lemma.

LEMMA 4. *Given AFDs D, D', and D'', if $D \succeq D'$ and $D' \succeq D''$, then $D \succeq D''$.*

7. AFDS AND OTHER CRASH PROBLEMS

In this section, we explore the relative solvability among AFDs and the consequences of such relative solvability on other crash problems that can be solved using AFDs. Section 7.1 shows that if an AFD D' is strictly stronger than another AFD D, then the set of problems that D' can solve in a given environment is a strict superset of the set of problems solvable by D in the same environment. Section 7.2 revisits the traditional notion of a *weakest failure detector* for a problem and defines what it means for an AFD to be a *weakest* to solve a crash problem in a given set of environments. We also introduce the notion of an AFD begin *representative* of a problem in a given set of environments. Section 7.3 shows that a large class of problems, which we call *finite problems*, do not have a representative AFD.

7.1 Comparing AFDs

Traditionally, as defined in [4], a failure detector $\mathcal{D}$ is stronger than a failure detector $\mathcal{D}'$ if $\mathcal{D}$ is sufficient to solve $\mathcal{D}'$. This definition immediately implies that every problem solvable in some environment using $\mathcal{D}'$ is also solvable in the same environment using $\mathcal{D}$. However, this definition does not imply the converse; if in every environment every problem solvable using $\mathcal{D}'$ is also solvable using $\mathcal{D}$, then it is not necessarily the case that $\mathcal{D}$ is stronger than $\mathcal{D}'$.

We demonstrate that in our framework, the converse must also be true; that is, given two AFDs D and D', every crash problem solvable using D' in a some environment is also solvable using D in the same environment iff D is stronger than D'. This is captured by the following theorem:

THEOREM 5. *For every pair of AFDs D and D', $D \succeq D'$ iff for every crash problem P, and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$.*

PROOF. The proof is immediate for the case where $D = D'$. For the remainder of the proof we fix D and D' to be distinct AFDs.

Claim 1: Let $D \succeq D'$. Fix P to be a crash problem and $\mathcal{E}$ to be an environment. If $D' \succeq_{\mathcal{E}} P$, then $D \succeq_{\mathcal{E}} P$.

PROOF. Assume $D' \succeq_{\mathcal{E}} P$. There exists a distributed algorithm $\mathcal{A}^P$ such that for every fair trace t of the composition of $\mathcal{A}^P$, with the crash automaton, the channel automata, and $\mathcal{E}$, if $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$, then $t|_{I_P \cup O_P} \in T_P$.

Since $D \succeq D'$, there exists a distributed algorithm $\mathcal{A}^{D'}$ such that for every fair trace t of the composition of $\mathcal{A}^{D'}$ with the crash automaton and the channel automata, $t|_{\hat{I} \cup O_D} \in T_D \Rightarrow t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Let $\mathcal{A}$ be a distributed algorithm where each $\mathcal{A}_i$ at location i is obtained by composing $\mathcal{A}_i^P$ and $\mathcal{A}_i^{D'}$. Let T_A be the set of all fair traces t of the composition of $\mathcal{A}$ with the crash automaton and the channel automata such that $t|_{\hat{I} \cup O_D} \in T_D$. By the construction of $\mathcal{A}^{D'}$, we know that for each such trace t, $t|_{\hat{I} \cup O_{D'}} \in T_{D'}$. Then, by the construction of $\mathcal{A}^P$, we have that $t|_{I_P \cup O_P} \in T_P$, which immediately implies $D \succeq_{\mathcal{E}} P$. □

Claim 2: If, for every crash problem P and every environment $\mathcal{E}$, $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$, then $D \succeq D'$.

PROOF. Suppose $D' \succeq_{\mathcal{E}} P \rightarrow D \succeq_{\mathcal{E}} P$, for every crash problem P and environment $\mathcal{E}$. Specifically, $D' \succeq D' \rightarrow D \succeq D'$. Applying Corollary 3, we conclude $D \succeq D'$. □

The theorem follows directly from Claims 1 and 2. □

COROLLARY 6. *Given two AFDs D and D' where $D \succ D'$, there exists a crash problem P and an environment $\mathcal{E}$ such that $D \succeq_{\mathcal{E}} P$, but $D' \not\succeq_{\mathcal{E}} P$; that is, there exists some problem P and an environment $\mathcal{E}$ such that D is sufficient to solve P in $\mathcal{E}$, but D' is not sufficient to solve P in $\mathcal{E}$.*

PROOF. If $D \succ D'$, then $D' \not\succeq D$. By the contrapositive of Theorem 5, there exists a problem P and an environment $\mathcal{E}$ such that $D \succeq_{\mathcal{E}} P$ and $D' \not\succeq_{\mathcal{E}} P$. □

7.2 Weakest and Representative AFDs

The issue of weakest failure detectors for problems was originally tackled in [4] in which a failure detector D is defined as a *weakest* to solve a problem P if the following two conditions are satisfied: (1) D is sufficient to solve P, and (2) any failure detector D' that is sufficient to solve P is stronger than D. This definition can be directly translated to our framework as follows.

An AFD D is *weakest* for a crash problem P in an environment $\mathcal{E}$ iff (1) $D \succeq_{\mathcal{E}} P$ and (2) for every AFD D' such that $D' \succeq_{\mathcal{E}} P$, $D' \succeq D$. An AFD D is a *weakest* for a crash problem P in a set of environments $\widehat{\mathcal{E}}$ iff for every $\mathcal{E} \in \widehat{\mathcal{E}}$, D is weakest for P in $\mathcal{E}$.

There have been many results that demonstrate weakest failure detectors for various problems. The proof techniques used to demonstrate these results have been of two distinct styles. The first proof technique was first proposed in [4] and is as follows. To show that D_P, which is sufficient to solve P, is the weakest failure detector to solve problem P it considers an arbitrary failure detector D that is sufficient to solve the problem P using an algorithm $\mathcal{A}$. It then constructs a distributed algorithm that exchanges the failure detector D's outputs and then continually simulates runs of $\mathcal{A}$ using the set of D's outputs available so far. From these simulations, an admissible output for D_P is extracted. This proof technique has been used to determine a weakest failure detector for the so-called one-shot problems such as consensus [4] and k-set consensus[11].

The second proof technique is simpler and follows from mutual reducibility. To show that D_P, which is sufficient to solve P, is the weakest failure detector to solve problem P, it uses a solution to P as a 'black box' to design a distributed algorithm whose outputs satisfy D_P. This proof technique has been used to determine a weakest failure detector for long-lived problems such as mutual exclusion [9, 2], contention managers [12], and dining philosophers [22].

A natural question is, "does the mutual-reducibility based proof technique work for determining weakest failure detectors for one-shot problems?" We answer this question negatively by introducing the notion of a representative AFD.

Representative AFD. Informally, an AFD is representative of a crash problem if the AFD can be used to solve the crash problem and conversely, a solution to the problem can be used to solve (or implement) the AFD.

Formally, an AFD D is *representative* of a problem P in an environment $\mathcal{E}$ iff $D \succeq_{\mathcal{E}} P$ and $P \succeq D$. An AFD D is *representative* of problem P in a set of environments $\widehat{\mathcal{E}}$ iff for every environment $\mathcal{E} \in \widehat{\mathcal{E}}$, D is representative of P in $\mathcal{E}$.

Observe that if an AFD D is representative of a crash problem P in $\widehat{\mathcal{E}}$, then D is also a weakest AFD to solve P in $\widehat{\mathcal{E}}$. However, the converse need not be true. Specifically if D is a weakest AFD to solve problem P in $\widehat{\mathcal{E}}$, it is not necessary for D to be representative of P in $\widehat{\mathcal{E}}$.

In particular, we highlight that the weakest failure detector results in [23, 22, 13] establish that the eventually perfect failure detector is representative for eventually fair schedulers, dining under eventual weak exclusion, and boosting obstruction-freedom to wait-freedom, respectively.

Next, we show that a large class of problems (which we call finite problems) do not have a representative failure detector despite having a weakest failure detector.

7.3 Finite Problems and Representative AFDs

In this subsection we define the notion of a finite problem, which captures what is often referred to as one-shot problems. Informally speaking, finite problems are those that have a bounded number of interactions with the environment. Examples of finite problems include consensus, leader election, terminating reliable broadcast, and k-set agreement. Examples of problems that are not finite problems include mutual exclusion, Dining Philosophers, synchronizers, and other long-lived problems.

Before we define finite problems we need some auxiliary definitions. A problem P is *crash independent* if, for every finite prefix t_{pre} of a trace $t \in T_P$, $t_{pre}|_{I_P \cup O_P \setminus \hat{I}}$ is a finite prefix of some $t' \in T_P$ such that $t'|_{\hat{I}}$ is empty. In other words, for every prefix t_{pre} of every trace $t \in T_P$, the subsequence of t_{pre} consisting of exactly the non-crash events is a prefix of some crash-free trace in T_P. For each $t \in T_P$, let $len(t)$ denote the length of the subsequence of t that consists of all non-crash events. A problem P has *bounded length* if there exists a $b_P \in \mathbb{N}^+$ such that, for every $t \in T_P$, $len(t) \leq b_P$.

If a problem P is crash independent and has bounded length we say that P is a *finite problem.*

Before we state the main theorem of this section, recall that an *unsolvable problem* is one that cannot be solved in a purely asynchronous system (i.e. without failure detectors).

THEOREM 7. *If P is a finite problem that is unsolvable in an environment $\mathcal{E}$ then P does not have a representative AFD in $\mathcal{E}$.*

Proof sketch. Suppose by contradiction that P is a finite problem that is unsolvable in an environment $\mathcal{E}$, and some AFD D is representative of P in $\mathcal{E}$. Therefore, there exists a distributed algorithm A^P that uses P to solve D, and conversely there exists a distributed algorithm A^D which uses D to solve P in $\mathcal{E}$. First we state the following lemma.

LEMMA 8. *There exists a crash-free finite execution α_{ref} of A having a trace t_{ref} such that (1) $t_{ref}|_{I_P \cup O_P} \in T_P$, (2) there are no messages in transit in the final state of α_{ref}, and (3) for every fair execution α' that extends α_{ref}, the suffix of α' following α_{ref} has no events in $I_P \cup O_P \setminus \hat{I}$.*

Before proving Lemma 8, we show why it implies the theorem. From Lemma 8, and crash independence of P, it follows that for any fair execution α' (and its associated trace t') of A that extends α_{ref} then $t'|_{I_P \cup O_P} \in T_P$. Since A solves D using P we have that $t'|_{I_D \cup O_D} \in T_D$.

For each $i \in \Pi$, let s_i be the state of process automaton at i at the end of α_{ref} and let f_i denote the sequence of events from O_D at location i in α_{ref}. Next, we describe a distributed algorithm A' which, in every fair execution, guarantees that each process i will first output the sequence f_i and then behave as A^P would behave when starting at state s_i.

The distributed algorithm A' which is identical to A^P except in the following ways at each $i \in \Pi$. (1) A'_i has an additional variable fdq_i that is a queue of failure-detector outputs and its initial value is f_i. (2) The initial values of all other variables in A'_i corresponds to the state s_i. (3) For every output action $a_i \in O_D$ at i, A'_i has two actions $int(a_i)$ and a_i: (a) $int(a_i)$ is an internal action whose associated state transitions are the same as action a_i in A^P_i except that, additionally, $int(a_i)$ enqueues the element a_i to fdq_i. (b) a_i is enabled when element a_i is at the head of fdq_i . The effect of a_i is to delete the element a_i from the head of fdq_i. (4) A'_i does not contain any action from $I_P \cup O_P \setminus \hat{I}$.

By construction and the FIFO property of the queues in A' we have the following lemma.

LEMMA 9. *For every fair execution α (and its trace t) of A' with the channel automata and the crash automaton, there exists a fair execution α_{A^P} (and its trace t_{A^P}) of the composition of A^P with the crash automaton, and the channel automaton where $t_{A^P}|_{I_P \cup O_P} \in T_P$ such that the following is true. (1) α_{ref} is a prefix of α_{A^P}. (2) $t|_{\hat{I} \cup O_D}$ is constrained reordering of a sampling of $t_{A^P}|_{\hat{I} \cup O_D}$.*

Lemma 9 implies that any fair execution α of A' produces a trace t such that $t|_{I_D \cup O_D} \in T_D$, and therefore A' solves D. Therefore, by composing A'_i and A^P_i (and their respective channel automata) at each location i , we obtain a distributed algorithm that solves P in $\mathcal{E}$; that is, P is solvable in $\mathcal{E}$. But P is assumed to be unsolvable in $\mathcal{E}$. Thus, we have a contradiction, and that completes the proof of Theorem 7. □

Proof of Lemma 8. Let σ be the set of all fair executions of A such that for any trace t produced by an execution in σ it is true that $t|_{I_P \cup O_P} \in T_P$. Let α_{max} be an execution in σ which produces the trace t_{max} that maximizes $len(t_{max}|_{I_P \cup O_P})$.

Let $\alpha_{s.pre}$ be the shortest prefix of the execution α_{max} which contains all events of $I_P \cup O_P$. Since P is bounded length it follows that such a prefix exists and is finite, and furthermore, any extension of $\alpha_{s.pre}$ does not include any events from $I_P \cup O_P \setminus \hat{I}$ because $len(t_{max}|_{I_P \cup O_P})$ is maximal. We extend $\alpha_{s.pre}$ to another finite execution α_{pre} by appending *receive* events for every message that is in transit at the end of $\alpha_{s.pre}$ such so that no message is in transit (and the channels are 'quiescent') at the end of α_{pre}

Let Π_C be the set of crashed locations in α_{pre}, and observe that by assumption after the first $crash_i$ event in α_{pre},

$proc(i)$ does not perform any outputs in α_{pre}. Let α_{ref} be identical to α_{pre} except that all crash events have been removed. For a location $i \notin \Pi_C$ the executions α_{pre} and α_{ref} are indistinguishable, and therefore $proc(i)$ must produce the same output in both executions. For a location $i \in \Pi_C$ the executions α_{pre} and α_{ref} are indistinguishable up to the point where the first event $crash_i$ occurs in α_{pre}, and after that point there is no other output at i in α_{pre}; therefore $proc(i)$ must produce the same output in both executions. Thus, α_{ref} is a finite crash-free execution that fulfills the requirements for the lemma. □

8. WEAKEST AFD FOR CONSENSUS

In a seminal result [4], Chandra et. al. established that Ω is a weakest failure detector to solve crash-tolerant binary consensus. Recasting the arguments from [4] in our modeling framework yields a simpler proof. The proof is split into two parts, which we discuss separately.

In the first part, as in [4], we construct a tree of possible executions of an AFD-based solution to consensus. However, in [4], each edge of such a tree corresponds to a single event whereas in our framework, each edge corresponds to a task, which represents a collection of events. Therefore, we reduce the number of cases for which we have analyze the tree. Specifically, we look for transitions from a bivalent to a monovalent execution.[2] Furthermore, the proof in [4] considers a forest of executions, where each tree in the forest corresponds to a single configuration of the inputs to consensus. In contrast, our framework treats inputs for consensus as events that are performed by the environment automaton. Therefore, we need analyze only a single tree of executions. These, two factors simplify the analysis of AFD-based consensus significantly and yield the following (paraphrased) claim, which may be of independent interest.

Claim. In the tree of all possible executions of a system solving consensus using an AFD, the events responsible for the transition from a bivalent to a univalent execution occur at a live location.

The second part of the proof uses the above claim to show that Ω is a weakest AFD to solve consensus. The arguments are similar to the ones presented in [4], but are simplified by the above claim.

As in [4], we present a distributed algorithm A^Ω which receives the outputs from the AFD D (which is sufficient to solve consensus) and solves Ω. The process automata exchange the AFD outputs among each other. Based on their current knowledge of the AFD outputs at various locations, A_i^Ω at each location i continually determines a finite "canonical" FD sequence, denoted t_i, which is a prefix of some sequence in T_D. Furthermore, as the execution proceeds, A_i^Ω at each location i obtains increasingly longer sequences of AFD outputs from other locations. Thus, at each live location i, A_i^Ω constructs increasingly longer canonical FD sequences t_i. Eventually, at each live location i, t_i converges to some unique sequence in $t_{ref} \in T_D$. More importantly, for any finite prefix t_{pre} of t_{ref}, eventually and permanently, the canonical sequences t_i at each live location i are extensions of t_{pre}.

[2]Briefly, an execution of the system is v-valent (where v is either 0 or 1) if the only possible decision at each location, in the execution or any fair extension of the execution, is v. A v-valent execution is monovalent. If an execution is not monovalent, then it is bivalent.

Periodically, at each location i, A_i^Ω uses its canonical sequence t_i to construct a finite tree of executions of depth d_i, where d_i is the length of t_i. From this tree, it determines the "earliest" transition from a bivalent execution to a monovalent execution of consensus. The location of the process associated with this transition is provided as the output of Ω at i. Note that the earliest such transition in the tree of executions is determined uniquely by the nodes within some finite depth (say) d of the tree. Let $t_{pre.d}$ be the prefix of t_{ref} of length d. Eventually and permanently, the canonical sequence t_i at each live location are extensions of $t_{pre.d}$. Therefore, eventually and permanently, A_i^Ω at every live location i determines the same "earliest" transition from a bivalent execution to a monovalent execution of consensus. From the claim established in the first part, we know that the the events responsible for the "earliest" transition from a bivalent to a univalent execution occur at a some live location (say) l. Therefore, eventually and permanently, A_i^Ω at every live location i determines l to be the output of the Ω AFD, which is a unique correct location. Thus A^Ω implements the Ω AFD using D. Thus, we show that Ω is a weakest AFD for consensus.

9. DISCUSSION

Query-Based Failure Detectors. Our framework models failure detectors as crash problems that interact with process automata unilaterally. In contrast, many traditional models of failure detectors employ a query-based interaction [4, 17]. Since the inputs to AFDs are only the crash events, the information provided by AFDs can only be about process crashes. In contrast, query-based failure detectors receive inputs from the crash events and the process automata. The inputs from process automata may "leak" information about other events in the system to the failure detectors We illustrate the ability of query-based failure detectors to provide such additional information with the following example.

Applying Theorem 7 we know that consensus does not have representative failure detectors. However, if we consider the universe of query-based failure detectors, we see that consensus has a representative query-based failure detector, which we call a *participant failure detector*. A participant failure detector outputs the same location ID to all queries at all times and guarantees that the process automaton whose associated ID is output has queried the failure detector at least once (observe that this does not imply that said location does not crash, just that the location was not crashed initially).

It is easy to see how we can solve consensus using the participant failure detector. Each process automaton sends its proposal to all the process automata before querying the failure detector. The output of the failure detector must be a location whose process automaton has already sent its proposal to all the process automata. Therefore, each process automaton simply waits to receive the proposal from the process automaton whose associated location ID is output by the failure detector and then decide on that proposal.

Similarly, solving participant failure detector from a solution to consensus is also straightforward. The failure detector implementation is as follows. Upon receiving a query, the process automaton inputs its location ID as the proposal to the solution to consensus. Eventually, the consensus solution decides on some proposed location ID, and therefore, the ID of some location whose process automaton queried the fail-

ure detector implementation. In response to all queries, the implementation simply returns the location ID decided by the consensus solution.

Thus, we see that query-based failure detectors may provide information about events other than crashes. Furthermore, unlike representative failure detectors, a representative query-based failure detector for some problem P is not guaranteed to be a weakest failure detector for problem P. In conclusion, we argue that unilateral interaction for failure detectors is more reasonable than a query-based interaction.

Future Work. Our work introduces AFDs, but the larger impact of AFD-based framework on the existing results from traditional failure-detector theory needs to be assessed. The exact set of failure detectors than can be specified as AFDs remains to determined. It remains to be seen if weakest failure detectors for various problems are specifiable as AFDs, and if not, then the weakest AFDs to solve these problems are yet to be determined. We are yet to investigate if the results in [17] hold true for AFDs and if every problem (as defined in [17]) has a weakest AFD. The exact characterization of problems that have a representative AFD and the problems that do not have a representative AFD is unknown.

10. ACKNOWLEDGMENTS

This work is supported in part by NSF Award Numbers CCF-0726514, CCF-0937274, and CNS-1035199, and AFOSR Award Number FA9550-08-1-0159. This work is also partially supported by Center for Science of Information (CSoI), an NSF Science and Technology Center, under grant agreement CCF-0939370.

11. REFERENCES

[1] M. K. Aguilera, S. Toueg, and B. Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *Proc. of 13th International Symposium on Distributed Computing*, pages 19–34, 1999.

[2] V. Bhatt, N. Christman, and P. Jayanti. Extracting quorum failure detectors. In *Proc. of 28th ACM symposium on Principles of distributed computing*, pages 73–82, 2009.

[3] V. Bhatt and P. Jayanti. On the existence of weakest failure detectors for mutual exclusion and k-exclusion. In *Proc. of the 23rd International Symposium on Distributed Computing*, pages 311–325, 2009.

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

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

[6] B. Charron-Bost, M. Hutle, and J. Widder. In search of lost time. *Information Processing Letters*, 2010.

[7] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *Proc. of International Conference on Dependable Systems and Networks*, pages 345–353, 2002.

[8] 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 *Proc. of 23rd ACM Symposium on Principles of Distributed Computing*, pages 338–346, 2004.

[9] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and P. Kouznetsov. Mutual exclusion in asynchronous systems with failure detectors. *Journal of Parallel and Distributed Computing*, pages 492–505, 2005.

[10] C. Fetzer, F. Tronel, and M. Raynal. An adaptive failure detection protocol. In *Proc. of the Pacific Rim International Symposium on Dependable Computing*, pages 146–153, 2001.

[11] E. Gafni and P. Kuznetsov. The weakest failure detector for solving k-set agreement. In *Proc. of 28th ACM symposium on Principles of distributed computing*, pages 83–91, 2009.

[12] R. Guerraoui, M. Kapalka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, pages 415–433, 2008.

[13] R. Guerraoui, M. Kapalka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. *Distributed Computing*, pages 415–433, 2008.

[14] R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *Proc. of 17th IFIP World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science: Foundations of Information Technology in the Era of Networking and Mobile Computing*, pages 461–473, 2002.

[15] R. Guerraoui and P. Kouznetsov. The weakest failure detector for non-blocking atomic commit. Technical report, EPFL, 2003.

[16] J. Y. Halpern and A. Ricciardi. A knowledge-theoretic analysis of uniform distributed coordination and failure detectors. In *Proc. of 18th ACM symposium on Principles of distributed computing*, pages 73–82, 1999.

[17] P. Jayanti and S. Toueg. Every problem has a weakest failure detector. In *Proc. of 27th ACM symposium on Principles of distributed computing*, pages 75–84, 2008.

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

[19] S. M. Pike, S. Sastry, and J. L. Welch. Failure detectors encapsulate fairness. In *14th International Conference Principles of Distributed Systems*, pages 173–188, 2010.

[20] S. Rajsbaum, M. Raynal, and C. Travers. Failure detectors as schedulers (an algorithmically-reasoned characterization). Technical Report 1838, IRISA, Université de Rennes, France, 2007.

[21] S. Rajsbaum, M. Raynal, and C. Travers. The iterated restricted immediate snapshot model. In *Proc of 14th International Conference on Computing and Combinatorics*, pages 487–497, 2008.

[22] S. Sastry, S. M. Pike, and J. L. Welch. The weakest failure detector for wait-free dining under eventual weak exclusion. In *Proc. of 21st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 111–120, 2009.

[23] Y. Song, S. M. Pike, and S. Sastry. The weakest failure detector for wait-free, eventually fair mutual exclusion. Technical Report TAMU-CS-TR-2007-2-2, Texas A&M University, 2007.

[24] N. C. Vibhor Bhatt and and P. Jayanti. Extracting quorum failure detectors. In *Proc. of 28th ACM Symposium on Principles of Distributed Computing*, pages 73–82, 2009.