Distinguishing Movement from Failure in Dynamic Distributed Systems

Matthew Lang and Paolo A. G. Sivilotti
Computer Science & Engineering
The Ohio State University
Columbus, Ohio 43210
{langma,paolo}@cse.ohio-state.edu

Nigamanth Sridhar
Electrical & Computer Engineering
Cleveland State University
Cleveland, Ohio 44115
n.sridhar1@csuohio.edu

Abstract

One of the main challenges in building fault-tolerant asynchronous systems lies in that, given a finite amount of time, it is not possible to distinguish a slow process from a process that has failed. Failure detectors—oracles that provide a set of processes suspected to have failed—were introduced to overcome this fundamental barrier while still maintaining the separation between algorithm and architecture.

Until recently, failure detector research had not been extended to models of computation with dynamic communication topologies (*e.g.*, sensor networks with mobile nodes). One of the first works to make this extension introduced $\diamond\mathcal{P}_l^m$, a failure detector that distinguishes failure from movement, and provided an implementation of the failure detector for sensor networks.

Unfortunately, the specification of $\diamond\mathcal{P}_l^m$ is unimplementable; the existing algorithms make stronger assumptions about the underlying network than are stated—or even possible—in the model.

In this work, we provide a formal model of computation that incorporates mobility and failure, thereby providing a framework for reasoning about algorithms in environments with dynamic topologies and process failures. We prove the unimplementability of $\diamond\mathcal{P}_l^m$ in our model and suggest an alternate specification for a mobility-tolerant failure detector. An algorithm implementing this alternate specification is presented, followed by an analysis of the expected behavior and cost of the algorithm in different environments.

I. Introduction

In asynchronous distributed systems, there is no bound on the time required for a process to execute an instruction, nor is there a bound on message delay. Because of this, it is impossible to distinguish between processes that have failed and processes that are slow or have sent messages that are yet to be received. As a result, fundamental problems, such as consensus, are impossible to solve [11].

Failure detectors [4] were introduced by Chandra and Toueg to overcome this issue. Failure detectors are oracles that, when queried by an algorithm, give an approximation of the set of processes that have failed. Most of the work surrounding failure detectors focuses on providing this approximation to each process and is framed by a model of computation that assumes a static network topology.

However, there are systems, such as sensor networks, where the network topology may not be static. Nor, in a system such as this, is it necessary for all processes to be aware of all failed processes in the system. In fact, it may not even be feasible. *e.g.*, a sensor network where the storage capacity of nodes is limited and the size of the network is very large.

In [16], Sridhar introduced the notion of a local failure detector for a system where the network topology is dynamic. The failure detector, $\diamond\mathcal{P}_l^m$, allows processes to distinguish between neighboring processes that have failed and those that have moved (and therefore appear to have failed) *without* requiring the failure detector for an individual process to have knowledge of all failed processes in the network. The need for such a failure detector is motivated by noticing that there are fundamental problems, such as the distributed resource allocation problem [7], where it is important to distinguish between a neighboring *process* failing while holding a resource and a *link* between processes failing. In the first case, a process requesting access to the held resource may choose to allow

This work was supported in part by a CAREER grant from the NSF (CNS-0746632).

itself to starve by lowering its priority, thereby allowing its other neighbors to make progress (as in [15], [14]). In the second case, the process may not need to sacrifice its own progress if a resource held by a process that has moved is no longer considered shared.

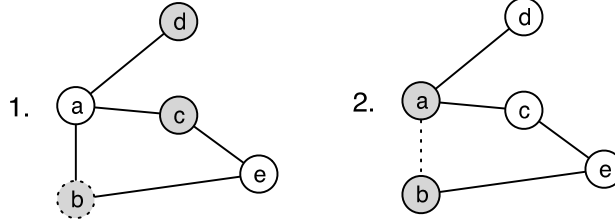

Fig. 1. Local Failure Detection Preserves Progress

To illustrate, consider Figure 1. In the figure, a gray node indicates a process holding a resource. In case one, process b has failed while holding a resource. To allow d and c to make progress, process a allows itself to starve. In case two, the link between a and b has failed. If a is able to detect this link failure, it need not starve itself to preserve the progress of d and c.

Unfortunately, the specification of $\diamond\mathcal{P}_l^m$ is unimplementable: it requires processes to distinguish between network topology changes that are unobservable. In addition, the algorithm presented along with $\diamond\mathcal{P}_l^m$ is flawed—there are circumstances where observable changes in network topology are not recognized by processes.

This work makes the following contributions: we develop a formal model of computation that provides a context to reason about process failure and mobility and allows algorithm designers to have a concise abstract view of environments where failure and mobility are present; we prove the impossibility of $\diamond\mathcal{P}_l^m$ in our model; we present an alternate specification that is implementable but still useful in the context of process mobility; and finally, we present an algorithm implementing our alternate specification, along with a discussion of its expected behavior in practice and possible optimizations.

This paper is organized as follows. Section II discusses related work and presents the informal model described in [16]. Section III introduces a model of computation incorporating failure and mobility. Section IV contains the proof of impossibility of $\diamond\mathcal{P}_l^m$. Section V presents our alternate specification and sections VI and VII contain an implementation and its proof of correctness. Section VIII describes possible optimizations to our algorithm and discusses its behavior under real-world conditions. Finally, section IX concludes the paper.

II. Mobility-Aware Failure Detection

In [16], the need for a computation to distinguish process mobility from process failure is discussed; to meet this need, the specification of an eventually perfect local failure detector that can make this distinction, $\diamond\mathcal{P}_l^m$ is introduced.

The work most closely related to $\diamond\mathcal{P}_l^m$ is that related to partitionable networks [9] and disconnection detectors [17]. Neither thread of research, however, considers the mobility of individual nodes.

The specification of $\diamond\mathcal{P}_l^m$ was given in terms of three layers; here we describe them informally.

Each process p in the system is comprised of four layers: a local computation, a discovery layer $DISC$, a local failure detector LFD, and a mobility detection layer $MOBILE$. Each process has a set of processes it may potentially communicate with: its *communication pattern*. Each process p has an associated *neighbor set* N_p and *suspect set* $S_p \subseteq N_p$; these two sets are the only elements of a process's state that is shared between layers.

- *The Local Computation Layer*
  Each process performs a local computation. The local computation on process p may read from N_p and S_p and exchange messages only with processes in N_p. The local computation uses S_p as an estimate of failed processes in its neighborhood. The other three layers provide guarantees about the accuracy of the contents of N_p and S_p.
  Informally, the local computation layer expects an implementation of $\diamond\mathcal{P}_l^m$ to ensure that all processes in its communication pattern are in its neighbor set and all processes that are outside of its communication pattern

are not in its neighbor set. Furthermore, it expects that all failed processes in its neighborhood are in its suspect set.

- *The Discovery Layer* $DISC$
  The discovery layer on process p, $DISC_p$ adds processes to N_p. The discovery layer ensures that if a correct process q remains in the communication pattern of p long enough, $DISC_p$ will eventually discover q's presence and add q to N_p.
  The discovery layer for p should not be cognizant of p's movement; the location of a process or the set of processes with which it may communicate must be discovered via communication with other processes in the network.
  The exact implementation of $DISC_p$ is network architecture dependent: in a sensor network where channels are defined by the ability of other processes to send and receive radio signals, a straightforward implementation of $DISC_p$ would be to periodically broadcast p's presence and, upon receiving such a signal from $DISC_q$, add q to N_p.
- *The Local Failure Detector* LFD
  Each process p is equipped with a failure detector LFD_p. LFD_p reads from N_p to generate the suspect set S_p.
  The LFD layer satisfies a version of the $\diamond\mathcal{P}$ specification [4] adapted to our system model. Namely, for a correct process p, LFD_p satisfies the following two properties:
  - *Local Completeness:* If a process q crashes and remains in the neighbor set of p indefinitely, then there is a time after which p permanently suspects q.
    If a process q moves outside of the communication pattern of p and remains outside it, then there is a time after which p permanently suspects q.
  - *Eventual Local Accuracy:* If q is correct, there is a time after which q is never suspected by p if q remains in N_p and in the communication pattern of p.

  Like the discovery layer, the LFD layer is network architecture-dependent. The purpose of LFD is to discover processes with which p can no longer communicate (whether as a result of movement or failure). Existing implementations of the $\diamond\mathcal{P}$ failure detector are suitable for this purpose, as a process failing in a static network has the same behavior as a process moving in a dynamic network; neither process may be communicated with by its former neighbors.
  There are many implementations of the $\diamond\mathcal{P}$ specification (such as heartbeats [1], adaptive timeouts [10], pinging [12], leases [2], etc.) that may be used as implementations of LFD, depending on the characteristics and timing properties of a given network architecture.
- *The Mobility Detector* $MOBILE$
  The mobility detection layer determines, for each process $q \in S_p$, whether q has indeed failed or if q is no longer in the communication pattern of p. The purpose of the $MOBILE$ layer is to prevent the local computation from blocking on non-failed neighbors that have moved away (and appear to have failed).
  The $MOBILE$ layer, unlike the $DISC$ and LFD layers, should not be dependent on the network architecture; it should provide guarantees to the local computation on the contents of N_p and S_p regardless of message delay and differences in relative clock speeds.

In the next section, we will formally define a model of computation for process mobility. The design of this model is informed by the separation of process *detection* from process *suspicion* discussed in this section. We will then give the specification of $\diamond\mathcal{P}_l^m$ in our model and show that there are no implementations of $MOBILE$ that satisfy $\diamond\mathcal{P}_l^m$.

III. Model of Computation

Our system model is that of an asynchronous distributed system; message delay, clock drift, and the time required by a process to execute a single step are all finite but unbounded. Our model is similar to that of [3] but adapted to model a network of mobile processes.

Formally, a system is comprised of a set of processes $\Pi = \{p_0 \ldots p_n\}$. All processes are connected pair-wise by a channel; for a pair of processes p and q, the channel $channel(p, q)$ is the set of unreceived messages sent

by p to q.

There is a discrete global clock $\mathcal{T}$ ordered by $<$.

1) Failure and Communication Patterns: Processes are assumed to be *fail stop* and do not recover. A *failure pattern* is a monotone function $\mathcal{F} : \mathcal{T} \longrightarrow \mathcal{P}(\Pi)$ that models the occurrence of failures in an execution[1]. If $p \in \mathcal{F}(t)$ then p may neither send nor receive messages with any of its neighbors at or after time t. Let the function $correct(\mathcal{F})$ denote the set of processes that do not fail in $\mathcal{F}$. *i.e.*, $p \in correct(\mathcal{F})$ if $(\forall t \in \mathcal{T} :: p \notin \mathcal{F}(t))$. Let $crashed(\mathcal{F})$ denote the set of processes $\{q \in \Pi | q \notin correct(\mathcal{F})\}$.

A *communication pattern* models the set of processes that a processes may successfully send a message to at each point in time. A communication pattern is a function $\mathcal{C} : \Pi \times \mathcal{T} \longrightarrow \mathcal{P}(\Pi)$ that satisfies the following two properties:

- The communication pattern is symmetric: p can successfully send a message to q if and only if q can successfully send a message to p. $(\forall p, q \in \Pi :: (\forall t \in \mathcal{T} :: p \in \mathcal{C}(q,t) \Leftrightarrow q \in \mathcal{C}(p,t)))$.
- The communication pattern is irreflexive: $(\forall p \in \Pi :: (\forall t \in \mathcal{T} :: p \notin \mathcal{C}(p,t)))$.
- The graph induced over correct processes by the communication pattern at each time t is connected. $(\forall t \in \mathcal{T} :: (\forall p, q \notin \mathcal{F}(t) : p \neq q : (\exists p_0, p_1, \ldots p_n \in \Pi : p_0 = p \wedge p_n = q : p_1 \in \mathcal{C}(p_0, t) \wedge p_2 \in \mathcal{C}(p_1, t) \wedge \ldots \wedge p_n \in \mathcal{C}(p_{n-1}, t))))$.

Note that the definition of the communication pattern allows processes to move after they have failed.

2) Algorithms and Executions: An *algorithm* $\mathcal{A}$ is a set of deterministic automata, one for each process. We use the notion $\mathcal{A}_p$ to denote the automaton for process p. The basic unit of computation is a *step* of $\mathcal{A}$.

A *configuration* is a tuple $\langle s, c \rangle$ where s is a function giving value to processes' local variables, and c is a function giving value to the channels between all process. Each process p has the sets N_p and S_p as elements of local state. Steps are functions from configurations to configurations.

An execution $E = \langle I, A, \mathcal{F}, \mathcal{C}, T, \mathcal{D}, \mathcal{H} \rangle$ of an algorithm $\mathcal{A}$ is a seven-tuple where I is the initial configuration of the system, A is a sequence of steps in $\mathcal{A}$, $\mathcal{F}$ is a failure pattern, $\mathcal{C}$ is a communication pattern, and T is a function mapping each step in A to a time $t \in \mathcal{T}$ subject to the constraint that subsequent actions in A are assigned increasing time values. $\mathcal{D}$ and $\mathcal{H}$ are *discovery* and *failure detector histories*, respectively.

A *discovery history* is a function $\mathcal{D} : \Pi \times \mathcal{T} \longrightarrow \mathcal{P}(\Pi)$. If $q \in \mathcal{D}(p,t)$ then we say that q is discovered by p at time t. A discovery history $\mathcal{D}$ is valid for an execution E if it satisfies the following:

- If a non-failed process q is in the communication pattern of a correct process p, either q is discovered, q leaves the communication pattern of p, or q fails.
  $(\forall p \in correct(\mathcal{F}) :: (\forall q \in \Pi :: (\forall t \in \mathcal{T} :: q \in \mathcal{C}(p,t) \wedge q \notin \mathcal{F}(t) \Rightarrow (\exists t' \in \mathcal{T} :: q \in \mathcal{D}(p,t') \vee q \notin \mathcal{C}(p,t') \vee q \in \mathcal{F}(t')))))$
- If a process q is discovered by a correct process p, it was in the communication pattern of p at some previous time.
  $(\forall p \in correct(\mathcal{F}) :: (\forall q \in \Pi : p \neq q : (\forall t \in \mathcal{T} :: q \in \mathcal{D}(p,t) \Rightarrow (\exists t' \in \mathcal{T} : t' \leq t : q \in \mathcal{C}(p,t')))))$
- If a process q is permanently outside the communication pattern of a correct process p, there is a time after which q is never discovered by p.
  $(\forall p \in correct(\mathcal{F}) :: (\forall q \in \Pi :: (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \notin \mathcal{C}(p,t'))) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \notin \mathcal{D}(p,t')))))$
- If process q fails, there exists a time after which q is never discovered by any correct process.
  $(\forall q \in crashed(\mathcal{F}) :: (\forall p \in correct(\mathcal{F}) :: (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \notin \mathcal{D}(p,t')))))$

A *failure detector history* is a function $\mathcal{H} : \Pi \times \mathcal{T} \longrightarrow \mathcal{P}(\Pi)$ that models the set of processes believed to be failed by process p at each point in time. For this paper, we stipulate that $\mathcal{H}$ satisfies a version of the $\diamond\mathcal{P}$ specification adapted for a situation in which processes are mobile. A failure detector history $\mathcal{H}$ is valid for an execution E if it satisfies the following:

- *Completeness:* If a process q remains in the communication pattern of a process p after failing, then p eventually and permanently suspects q.

[1]where $\mathcal{P}(\mathcal{A})$ denotes the power set of the set $\mathcal{A}$

$(\forall p \in correct(\mathcal{F}) :: (\forall q \in crashed(\mathcal{F}) :: (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \in \mathcal{C}(p,t'))) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \in \mathcal{H}(p,t')))))$

- *Accuracy:* If a correct process q remains in the communication pattern of a correct process p permanently, then there is a time after which q is never suspected by p.
  $(\forall p, q \in correct(\mathcal{F}) :: (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \in \mathcal{C}(p,t'))) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \notin \mathcal{H}(p,t'))))$
- *Locality:* If there is a time after which a process q is permanently outside of the communication pattern of a correct process p then p permanently suspects q.
  $(\forall p \in correct(\mathcal{F}) :: (\forall q \in \Pi : q \neq p : (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \notin \mathcal{C}(p,t'))) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : q \in \mathcal{H}(p,t')))))$

A step of $\mathcal{A}_p$ consists of a finite number of the following operations:

- *Changes to internal state*. An step may make changes to the process's local state subject to the restriction that it may not add processes to N_p or S_p.
- *Message send operation*. If the step occurs at time t and involves the send of a message m to process q such that $q \in \mathcal{C}(p,t)$, the message is added to the channel: if $channel(p,q) = M$ before the step, then $channel(p,q) = M \cup \{m\}$ afterwards and we say the send operation was *successful*. If p sends a message m to process q such that $q \notin \mathcal{C}(p,t)$ then the message is not added to the channel: the state of the channel after the step is the same as it was before and we say the send operation has *failed*.
- *Message receive operations*. As in [3] message receive operations behave as follows: If the step contains a receive operation, then either a message $m \in channel(q,p)$ is delivered to p and removed from the channel or a *null* message is received. If there does not exist a q such that $channel(q,p) \neq \emptyset$ then a null message is received. Non-determinism is introduced into the model by not constraining the order of message delivery and by allowing null messages to be received when a process's channels contain messages. However, every message in $channel(q,p)$ is eventually delivered; after a message is sent, it is received after a finite but unbounded number of receive operations.
  Note that this definition allows a message to be received by p at time t from some $q \notin \mathcal{C}(p,t)$ if q was in $\mathcal{C}(p,t')$ at some time $t' < t$.

Steps of $\mathcal{A}_p$ may not read $\mathcal{F}$, $\mathcal{C}$, T, $\mathcal{D}$, or $\mathcal{H}$.

We assume the same constraints on executions as [3] with respect to process fairness; each correct process takes an infinite number of steps in A.

If step α in $\mathcal{A}_p$ occurs in A at a time t such that $p \in \mathcal{F}(t)$, then the step has no effect on state. That is, if the step occurs in a configuration $\langle s, c \rangle$, then the configuration after the action occurs is $\langle s, c \rangle$. Actions are atomic—processes do not fail nor does the communication pattern change in the middle of a step.

An execution E begins in the initial configuration I and proceeds by taking steps of $\mathcal{A}$ given by A. A step α_i in A occurs in the configuration given by the result of the previous step α_{i-1} or the initial configuration (in the case of $i = 0$) *and* the value of $\mathcal{D}$ and $\mathcal{H}$ at time $T(\alpha_i)$.

Suppose α_{i-1} resulted in a configuration $\sigma = \langle s, c \rangle$. The configuration that α_i occurs in is $\sigma' = \langle s', c' \rangle$ where $c' = c$ and for all processes p, $s'(a) = s(a)$ for all variables $a \neq N_p$ or $a \neq S_p$. The values of $s'(N_p)$ and $s'(S_p)$ are given by $\mathcal{D}$ and $\mathcal{H}$: $s'(N_p) = s(N_p) \cup \{q | (\exists t \in \mathcal{T} : T(\alpha_{i-1}) < t \leq T(\alpha_i) : q \in \mathcal{D}(p,t))\}$ and $s'(S_p) = s'(N_p) \cap \{q | q \in \mathcal{H}(p, T(\alpha_i))\}$.

We will use the notation $\sigma_i = \langle s_i, c_i \rangle$ to denote the configuration that step α_i occurs in. For an execution E, the trace σ produced by E is the sequence $\sigma = \langle \sigma_0, \sigma_1, \ldots \rangle$.

We will also define $N_p(t)$ to be the value of $s_i(N_p)$ if there exists a step α_i such that $T(\alpha_i) = t$. If there does not exist such a step, let t' be the least time greater than t such that there exists an action α_i where $T(\alpha_i) = t'$ and let $N_p(t) = s_i(N_p)$. Let $S_p(t)$ be defined similarly.

It is important to note that in the operational view of the system described in section II, the discovery layer and local failure detector are components of a process; here, we deliberately make them a part of the model of computation. This separation simplifies reasoning about algorithms that execute within the model. In practice, there will be a components of p much like the ones described in the previous section running on a local process. To

show that these components are correct, one must verify that a particular implementation produces sequences of values of N_p and S_p that are possible sequences of N_p and S_p produced by valid $\mathcal{D}$ and $\mathcal{H}$. For the rest of the paper, a process p suspecting a process q means q is in S_p.

IV. Impossibility of $\Diamond\mathcal{P}_l^m$

In [16], Sridhar proposed the $\Diamond\mathcal{P}_l^m$ specification for a mobility-aware local failure detector. An algorithm $\mathcal{A}$ satisfies $\Diamond\mathcal{P}_l^m$ if the following properties are satisfied[2]:

The following are quantified over all executions:

- *Strong Local Completeness*: There exists a time after which all correct processes suspect all crashed processes in their communication set.
  $(\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p \in \mathit{correct}(\mathcal{F}) :: (\forall q \in \mathit{crashed}(\mathcal{F}) :: q \in \mathcal{C}(p,t') \Rightarrow q \in S_p(t')))))$
- *Eventually Strong Local Accuracy*: There exists a time after which no correct process suspects another correct process in their neighbor set and the neighbor set of all correct processes are infinitely often equal to their communication set.
  - $(\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p,q \in \mathit{correct}(\mathcal{F}) :: q \in N_p(t') \Rightarrow q \notin S_p(t'))))$
  - $(\forall p \in \mathit{correct}(\mathcal{F}) :: (\forall t \in \mathcal{T} :: (\exists t' \in \mathcal{T} : t' \geq t : N_p(t') = \mathcal{C}(p,t'))))$
- *Suspicion Locality*: There exists a time after which all processes suspected by a correct process are in its neighbor set.
  $(\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p \in \mathit{correct}(\mathcal{F}) :: S_p(t') \subseteq N_p(t'))))$

These properties indeed represent a failure detector that can distinguish between mobility and failure. However, these properties are too strong: they require a $MOBILE$ implementation to be able to distinguish between a process that fails and a process that moves and subsequently fails without sending any messages.

A. Impossibility of $\Diamond\mathcal{P}_l^m$

First, we will show that in the general case, $\Diamond\mathcal{P}_l^m$ is unimplementable. Our proof will exploit the properties that $\Diamond\mathcal{P}_l^m$ guarantees about processes that fail at a time t and the communication set of which change at a time later than t. This is sufficient to show that $\Diamond\mathcal{P}_l^m$ is unimplementable; however, the case used to illustrate the impossibility of $\Diamond\mathcal{P}_l^m$ can be ruled out by slightly weakening the specification. To show that the weakened version of the specification is still unimplementable, we will also show that it is not possible to implement $\Diamond\mathcal{P}_l^m$ even if processes' communication patterns do not change after they fail.

Informally, $\Diamond\mathcal{P}_l^m$ is unimplementable because an algorithm implementing it must be able to distinguish between the two executions shown in Figure 2. In the figure, a, b, and c are processes and edges between processes represent the communication pattern.

In the first execution, the communication pattern remains constant throughout the execution; a's communication pattern is b and c and b's communication pattern is a. At some point in time, c fails.

In the second execution, the communication pattern changes; initially a's communication pattern is b and c and initially b's communication pattern is a. Then, at some later time, process c moves and subsequently fails, *without* being discovered by b. Since b did not discover c before it failed (and cannot discover it after it fails), b will never suspect c and behave as it would in the first execution. This violates the Strong Local Completeness property of the $\Diamond\mathcal{P}_l^m$ specification, which requires b to suspect c.

The first proof of impossibility follows this intuition, except c moves after it fails. We will formalize the above intuition when we discuss the impossibility of a slightly weaker version of the $\Diamond\mathcal{P}_l^m$ specification.

Theorem 4.1: There does not exist an algorithm $\mathcal{A}$ such that $\mathcal{A}$ satisfies the $\Diamond\mathcal{P}_l^m$ specification.

Proof: By contradiction. For a contradiction assume there does exist such an $\mathcal{A}$.

Let $\Pi = \{a,b,c\}$ and let $E = \langle I, A, \mathcal{F}, \mathcal{C}, T, \mathcal{D}, \mathcal{H} \rangle$ be an execution of $\mathcal{A}$ satisfies the following:

- $\{a,b\} \in \mathit{correct}(\mathcal{F})$, $c \notin \mathcal{F}(T(\alpha_0))$, and $t_c \in \mathcal{T}$ is the least time such that $c \in \mathcal{F}(t_c)$.
- For all $t \in \mathcal{T}$, $\mathcal{C}(a,t) = \{b,c\}$ and $\mathcal{C}(b,t) = \{a\}$.

[2] These properties are stated in terms of our model, $\Diamond\mathcal{P}_l^m$ was originally stated informally.

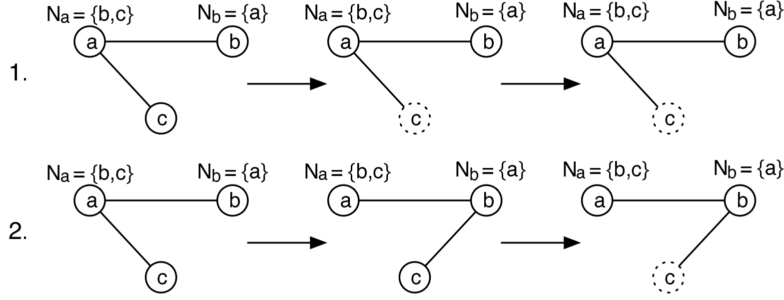

Fig. 2. Indistinguishable Computations

- There exists a time t such that for all $t' \geq t$, $c \in \mathcal{H}(a, t')$ and $c \in \mathcal{H}(b, t')$. This is a valid failure detector history for E as c is eventually permanently suspected by a.

Let σ be the trace produced by E. Since $\mathcal{A}$ satisfies $\diamond\mathcal{P}_l^m$, by the Eventual Strong Local Accuracy property, it is true that $N_b(t)$ is equal to $\mathcal{C}(b,t)$ for an infinite number of values of $t \in \mathcal{T}$. Then, since Suspicion Locality requires $S_b(t) \subseteq N_b(t)$ for all t, there does not exist a time t_b such that for all $t \geq t_b$, $c \in S_b t$.

Now let E' be an execution such that $E' = \langle I, A, \mathcal{F}, \mathcal{C}', T, \mathcal{D}, \mathcal{H} \rangle$. Let t_m be a time greater than t_c and let $\mathcal{C}'$ such that for all t less than t_m and processes p, $\mathcal{C}'(p,t) = \mathcal{C}(p,t)$ and for all t greater than or equal to t_m, $\mathcal{C}(a,t) = \{b\}$ and $\mathcal{C}(b,t) = \{a, c\}$. Since $t_m > t_c$, $\mathcal{D}$ is a valid discovery history for E'. Also, since there is a time after which c is permanently suspected by b, $\mathcal{H}$ is a valid failure detector history for E'.

Now, since $\mathcal{A}$ is correct, by the Strong Local Completeness property of $\diamond\mathcal{P}_l^m$ there is a time t_b' such that for all $t \geq t_b'$, $c \in S_b(t)$.

For a contradiction, we will show that σ is produced by E'.

Since $\mathcal{F}$, A, T, $\mathcal{D}$, and $\mathcal{H}$ are the same in E and E', it suffices to show by induction on i that for all i, σ_i is the configuration in which step α_i occurs in E'.

First, since I is the initial configuration of E' and the values of $\mathcal{F}$, $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{H}$ are the same as in E before t_c, σ_0 is the configuration in which step α_0 occurs in E'.

Now assume σ_i is the configuration in which α_i occurs in E'. We will show that σ_{i+1} is the configuration in which α_{i+1} occurs in E'. Let γ be the configuration that results from α_i in E and let γ' be the configuration that results from σ_i in E'.

Since steps are deterministic, the only way for γ to be a different configuration from γ' is for the value of $\mathcal{C}$ to differ from $\mathcal{C}'$ at time $T(\alpha_i)$ and as a result, a send operation of α_i fails.

There are two cases for the value of $T(\alpha_i)$: $T(\alpha_i) < t_m$ and $T(\alpha_i) \geq t_m$. If $T(\alpha_i) < t_m$, then $\mathcal{C}$ agrees with $\mathcal{C}'$ and $\gamma' = \gamma$. If $T(\alpha_i) \geq t_m$, since $t_m > t_c$ and c has already failed, messages sent to or from c fail regardless of the change in the communication pattern and again it is the case that $\gamma = \gamma'$.

Since $\mathcal{D}$ and $\mathcal{H}$ are valid detection and failure detector histories for E' and $T(\alpha_{i+1})$ is the same value in both E and E', the configurations induced by $\mathcal{D}$ and $\mathcal{H}$ from γ and γ' are equal. So σ_{i+1} is the configuration in which α_{i+1} occurs in E.

Then σ is produced by E'. However, we have shown that both $(\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : c \in S_b(t)))$ holds for σ (since σ is produced by E') and $(\forall t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : c \notin S_b(t)))$ holds for σ (since σ is produced by E), which is a contradiction.

Then there does not exist an algorithm $\mathcal{A}$ such that $\mathcal{A}$ satisfies $\diamond\mathcal{P}_l^m$.

□

This establishes our result; however, one may note that the proof relies on a process's communication pattern changing after it fails (*i.e.*, the process fails and subsequently moves) and that it is unreasonable to expect a process detects the existence of a process that no longer communicates and moved into its communication pattern. This suggests a slightly weaker specification of $\diamond\mathcal{P}_l^m$—one in which processes that do not change their communication pattern after failing must be permanently suspected by members of their communication pattern. This specification is unimplementable as well.

Theorem 4.2: An execution $E = \langle I, A, \mathcal{F}, \mathcal{C}, T, \mathcal{D}, \mathcal{H} \rangle$ of an algorithm $\mathcal{A}$ has *stationary failed processes* if $(\forall p \in crashed(\mathcal{F}) :: (\forall t \in \mathcal{T} :: p \in \mathcal{F}(t) \Rightarrow (\forall t' \in \mathcal{T} : t' \geq t : \mathcal{C}(p,t) = \mathcal{C}(p,t')))).$

There does not exist an algorithm $\mathcal{A}$ such that all executions of $\mathcal{A}$ with stationary failed processes satisfy the $\diamond\mathcal{P}_l^m$ properties.

Proof:

For a contradiction suppose there does exist such an algorithm $\mathcal{A}$. Let $\Pi = \{a, b, c\}$ and let $E = \langle I, A, \mathcal{F}, \mathcal{C}, T, \mathcal{D}, \mathcal{H} \rangle$ be an execution such that E satisfies the following:

- $\{a, b\} = correct(\mathcal{F})$, $c \notin \mathcal{F}(T(\alpha_0))$, and t_c is the least t such that $c \in \mathcal{F}(t)$.
- Let $T(\alpha_{i+1}) = t_c$ and let $T(\alpha_i) = t_p$. Let T be such that there exists a time t_m such that $t_p < t_m < t_c$.
- For all $t \in \mathcal{T}$, $\mathcal{C}(a,t) = \{b, c\}$ and $\mathcal{C}(b,t) = \{a\}$.
- There exists a time t such that for all $t' \geq t$, $c \in \mathcal{H}(a,t')$ and $c \in \mathcal{H}(b,t')$. This is a valid failure detector history for E as c is eventually permanently suspected by a.

Notice that E is an execution with stationary failed processes.

By the definition of discovery histories, there does not exist a time $t \geq t_c$ such that $c \in \mathcal{D}(b,t)$.

As in Theorem 4.1, let σ be the trace produced by E. Since $\mathcal{A}$ satisfies $\diamond\mathcal{P}_l^m$, by the Eventual Strong Local Accuracy property $N_b(t)$ is equal to $\mathcal{C}(b,t)$ for an infinite number of values of $t \in \mathcal{T}$. Then, by Suspicion Locality, there does not exist a time t_b such that for all $t \geq t_b$, $c \in S_b(t)$.

Now let $E' = \langle I, A, \mathcal{F}, \mathcal{C}', T, \mathcal{D}, \mathcal{F} \rangle$ where $\mathcal{C}'$ is such that for all $t < t_m$ and processes p, $\mathcal{C}'(p,t) = \mathcal{C}(p,t)$ and for all $t \geq t_m$, $\mathcal{C}(a,t) = \{b\}$ and $\mathcal{C}(b,t) = \{a, c\}$. Notice that both $\mathcal{D}$ and $\mathcal{H}$ are valid histories for E' since c fails after t_m and both a and b permanently suspects c. Also notice that E' is an execution with stationary failed processes.

As before, since $\mathcal{A}$ is correct, by the Strong Local Completeness property of $\diamond\mathcal{P}_l^m$ there is a time t'_b such that for all $t \geq t'_b$, $c \in S_b(t)$.

We will show that σ is produced by E'; as before, it suffices to show that for all i, σ_i is the configuration in which α_i occurs in E'.

As before, σ_0 is the configuration in which step α_0 occurs in E'.

Now suppose σ_i is the configuration in which step α_i occurs in E'. We will show that σ_{i+1} is the configuration in which α_{i+1} occurs in E'. This follows from similar reasoning to the previous theorem; in this case, however, the only disagreement between $\mathcal{C}$ and $\mathcal{C}'$ occurs at a time immediately prior to c failing. After c fails, no step of $\mathcal{A}_c$ has any effect on state, so no different configuration can result from a step after t_p.

Then σ is produced by E'. As in the previous theorem, we have shown that both $(\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : c \in S_b(t)))$ holds for σ (since σ is produced by E') and $(\forall t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : c \notin S_b(t)))$ holds for σ (since σ is produced by E), which is a contradiction.

Then there does not exist an algorithm $\mathcal{A}$ such that $\mathcal{A}$ satisfies $\diamond\mathcal{P}_l^m$ for all executions with stationary failed processes.

□

It should be noted that $\diamond\mathcal{P}_l^m$ also requires implementations to exhibit correct behavior when the communication pattern is not quiescent. If processes move infinitely often, a correct algorithm must be able to detect this, even though it may never be able to observe the movement.

V. Specification of a Mobility-Aware Failure Detector

Since the specification for $\diamond\mathcal{P}_l^m$ is unimplementable, we propose a mobility-aware failure detector specification that still provides useful guarantees to the computational layer and is implementable; an implementation is given in the next section.

Our specification differs from $\diamond\mathcal{P}_l^m$ in that we weaken the requirement that a process suspect failed processes within its communication pattern; we only require that processes which consider a failed process to be a neighbor suspect it. We also make the completeness and accuracy properties conditional on the communication pattern being quiescent.

In the following, let $\mathcal{Q}(\mathcal{C}) \equiv (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p \in \Pi :: \mathcal{C}(p,t) = \mathcal{C}(p,t'))))$. That is, $\mathcal{Q}(\mathcal{C})$ is true if the communication pattern eventually stabilizes.

Our mobility-aware failure detector specification is as follows:

- *Completeness:* If the communication pattern eventually stabilizes, there is a time after which it is permanently the case that all correct processes suspect incorrect processes that are considered neighbors.
  $\mathcal{Q}(\mathcal{C}) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p \in correct(\mathcal{F}) :: (\forall q \in crashed(\mathcal{F}) :: q \in N_p(t') \Rightarrow q \in S_p(t')))))$
- *Accuracy:* If the communication pattern eventually stabilizes, there is a time after which it is permanently the case that no correct process is suspected by any other correct process.
  $\mathcal{Q}(\mathcal{C}) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p, q \in correct(\mathcal{F}) :: q \notin S_p(t'))))$
- *Locality:*
  1) The suspect set of a correct process is always a subset of its neighbor set. $(\forall p \in correct(\mathcal{F}) :: (\forall t \in \mathcal{T} :: S_p(t) \subseteq N_p(t)))$
  2) If the communication pattern stabilizes, there is a time after which it is permanently the case that all correct processes are in another correct process's neighbor set if and only if they are in its communication pattern.
     $\mathcal{Q}(\mathcal{C}) \Rightarrow (\exists t \in \mathcal{T} :: (\forall t' \in \mathcal{T} : t' \geq t : (\forall p, q \in correct(\mathcal{F}) :: q \in \mathcal{C}(p, t') \Leftrightarrow q \in N_p(t'))))$

Even though this specification provides weaker guarantees to a client, it is still useful—processes are still able to differentiate between movement and failure. Though our specification does not guarantee that processes that move and subsequently fail are removed from a process's neighbor set, it does provide the guarantees necessary for a to make progress in the example given in Figure 1: the difference between b failing and b moving will be detected.

Our requirement of quiescence is similar to the assumptions made when reasoning about self-stabilizing algorithms—that the system is not perturbed while the algorithm is stabilizing—in that it simplifies the specification and simplifies reasoning about implementations. An algorithm implementing the specification provides a correct view of the system after a "sufficiently long" period of quiescence. Stating our properties in terms of an infinitely long period of quiescence eliminates the need to discuss exactly what duration of time is "sufficiently long." It has been noted that in practice, the most common cases of topology perturbation occur when battery sources degrade over time and links begin to fail ([6]). As batteries are replaced, new links are formed. It has been noted that in these types of systems, there are long periods where the topology remains static; it is in these periods that algorithms implementing the specification make corrections to nodes' view of the system.

VI. Algorithm

Our implementation of the mobility-aware failure detector specification, the MD algorithm, is a diffusing computation [8], [13]. The MD algorithm is designed to run periodically. After each successful run of the algorithm, the neighbor set and suspect set of each correct process is updated to remove processes that are suspected of failure by that process but are not suspected of failure by at least one other process that considers the suspected process a neighbor. By doing so, the MD algorithm guarantees that eventually all correct processes are not suspected by any other correct process.

The MD algorithm, in contrast to the algorithm presented in [16], distributes the set of all *correct* processes to every process in the system. This simplifies reasoning about the correctness of the algorithm. We realize that in a setting where the number of failed processes or links is much smaller than the size of the entire network, it may be wasteful to share the list of correct processes throughout the network. We present MD as a general solution and discuss optimizations that can reduce both the number of messages exchanged and the size of messages in Section VIII.

In the following, we refer to N_p as the neighbors of p and S_p as the processes suspected by p.

Processes are in one of three states: `idle`, `collect`, or `distribute`. Each run of the MD algorithm (given in Algorithm 1) begins with each process in the `idle` state and starts with a unique $root$ process initiating a round of the computation by setting its state to `collect` and sending a message to all its neighbors, N_{root}, containing the set of processes not suspected by $root$.

When an `idle` process receives a message from a neighbor, it enters the `collect` state and sets its $parent$ to be the process from which it received the message and its $children$ to all other neighbors. It then initializes a set X to be the set of processes it does not suspect and then sends this set to all of its $children$.

If a process in the `collect` state receives a message from a child, it removes that process from its $children$ and unions the set of processes it received with X. After a process has received messages from all its children, it sends the accumulated set of unsuspected processes to its parent and sets its state to `distribute`.

When the $root$ process has received messages from all processes in its $children$ set, it begins a second round of the computation by sending a message containing the set accumulated in the first round to all its neighbors and enters the `idle` state.

When a process in the `distribute` state receives a message it passes the message on to its neighbors and enters the `idle` state.

The algorithm is complete when all processes are in the `idle` state and the set of processes that at least one process does not suspect has been distributed to every correct process.

In the presentation of the algorithm, we ignore the circumstances under which the algorithm will not terminate. This can occur if processes fail or the communication pattern changes during a run of the algorithm. We also do not describe how a $root$ process is chosen or how a new $root$ process is selected if the $root$ fails. We assume that the algorithm is run periodically by a client that manages the selection of a $root$ process and interrupts the computation to prevent non-termination by setting processes states to `idle` after some network-dependent timeout. The proof of correctness makes clear assumptions about the state of the system during a terminating run of the algorithm. Given the model of computation and the assumption of quiescence, these conditions are eventually met. It is the responsibility of the client to ensure that the algorithm is run under these circumstances.

VII. Proof of Correctness

In the following, let $E = \langle I, A, \mathcal{F}, \mathcal{C}, T, \mathcal{D}, \mathcal{H} \rangle$ be an arbitrary execution of MD.

The Completeness and the first Locality properties follow trivially from the program text, the model of computation, and the definition of $\mathcal{H}$.

To show the second Locality the Accuracy properties, we first introduce a few lemmas.

In the following, if $Q(\mathcal{C})$ is assumed to hold, let $t_m \in \mathcal{T}$ be a time after which the failure and communication patterns remains constant. Such a time is guaranteed to exist by the definition of Q and the fact that Π is finite.

Lemma 7.1: If $Q(\mathcal{C})$ there is a time $t_q \geq t_m$ after which the following properties hold:

1) If a process q has failed and $q \in N_p$ for some correct process p, $q \in S_p$. Formally, $(\forall p \in correct(\mathcal{F}) :: (\forall q \in crashed(\mathcal{F}) :: (\forall t \in \mathcal{T} : t \geq t_q : q \in N_p(t) \Rightarrow q \in S_p(t))))$.
2) If p and q are correct processes and $q \in N_p$, then $q \in S_p$ if and only if q is not in the communication set of p. $(\forall p, q \in correct(\mathcal{F}) :: (\forall t \in \mathcal{T} : t \geq t_q : q \in N_p(t) \Rightarrow (q \in S_p(t) \Leftrightarrow q \notin \mathcal{C}(p,t))))$
3) If p and q are correct processes and q is in the communication set of p, then q is infinitely often added to N_p. $(\forall p, q \in correct(\mathcal{F}) :: (\forall t \in \mathcal{T} : t \geq t_q : (\exists t' \in \mathcal{T} : t' \geq t : q \in \mathcal{C}(p,t') \Rightarrow q \in N_p(t'))))$.

Proof: Assume $Q(\mathcal{C})$. Notice that for all $t \geq t_m$, $\mathcal{C}(p,t) = \mathcal{C}(p,t_m)$ for all p and $\mathcal{F}(t) = \mathcal{F}(t_m)$.

For property one, by the completeness and locality definitions of $\mathcal{H}$, there exists a time $t \geq t_m$ for each failed process q and correct process p such that for all $t' \geq t$ if $q \in \mathcal{C}(p,t')$ then $q \in \mathcal{H}(p,t')$. Then, by the definition of the model of computation and the first Locality property, if $q \in N_p(t')$, $q \in S_p(t')$. Also, by the definition of the model of computation if $q \notin \mathcal{C}(p,t')$ and $q \in N_p(t')$, $q \in S_p(t')$.

Let t_1 be the maximum of all such t. t_1 satisfies property one.

For property two, let p and q be correct processes. By the accuracy property of $\mathcal{H}$ and the model of computation, there exists a time $t \geq t_m$ such that for all $t' \geq t$, if $q \in N_p(t')$ and $q \in S_p(t')$ then $q \notin \mathcal{C}(p,t')$. By the locality property of $\mathcal{H}$ and the model of computation, there exists a time t such that for all $t' \geq t$, if $q \in N_p(t')$ and $q \notin \mathcal{C}(p,t')$ then $q \in S_p(t')$.

Let t_2 be the maximum of all such t. t_2 satisfies property two.

For the third property, let p and q be correct processes such that $q \in \mathcal{C}(p,t_m)$. By the definition of $\mathcal{D}$ and the definition of t_m, for all $t \geq t_m$ there exists a $t' \geq t$ such that $q \in \mathcal{D}(p,t')$. t_m satisfies property three.

Finally, let $t_q = \max(t_1, t_2, t_m)$. t_q satisfies the three properties listed. □

Lemma 7.2: If the $root$ process begins the MD algorithm by entering the `collect` state at time t_{start} then the algorithm terminates with all correct processes p in the `idle` state with $X_p = X_{root}$ at time t_{end}, provided the following hold for all t such that $t_{start} \leq t \leq t_{end}$ and for all p such that $p \notin \mathcal{F}(t_{start})$:

Algorithm 1 Mobility Detection Algorithm for Process p: MD_p

```
Program MD_p
initially state_p = idle
assign
    p = root ∧ state_p = idle ⟶
        state_p := collect
        children_p := N_p \ S_p
        X_p := N_p \ S_p
        forall r ∈ children_p
            send X_p to r

    p = root ∧ state_p = distribute ∧ children_p = ∅ ⟶
        forall r ∈ N_p \ S_p
            send X to r

    state_p = idle ∧ receive N from q ⟶
        parent_p := q
        children_p := N_p \ S_p \ {q}
        X_p := N ∪ (N_p \ S_p)
        forall r ∈ children_p
            send X_p to r
        if children ≠ ∅ then state_p := collect
        else state_p := distribute
            sendX_p to parent_p

    state_p = collect ∧ receive N from q ⟶
        children_p := children_p \ {q}
        X_p := X_p ∪ N
        if children = ∅ then state_p := distribute
            send X_p to parent_p

    state_p = distribute ∧ receive N from q ⟶
        forall r ∈ N_p \ S_p \ {q}
            send N to r
```

- $\mathcal{F}(t) = \mathcal{F}(t_{start})$.
- $\mathcal{C}(p,t) = \mathcal{C}(p,t_{start})$.
- $N_p(t) = N_p(t_{start})$ and $S_p(t) = S_p(t_{start})$
- If $q \in N_p(t) \setminus S_p(t)$ then $q \in \mathcal{C}(p,t)$ and $p \in N_q(t) \setminus S_q(t)$.
- $(\exists\, p_0 \dots p_n \,:\, p_0 = root \wedge p_n = p \,:\, p_1 \in N_{p_0}(t) \setminus S_{p_0}(t) \wedge p_2 \in N_{p_1}(t) \setminus S_{p_1}(t) \wedge \dots \wedge p_n \in N_{p_{n-1}} \setminus S_{p_{n-1}})$.

Proof: Assume the $root$ process begins the MD algorithm by entering the `collect` state at time t_{start} and if at t_{end} all processes are in the `idle` state, the properties stated in the lemma hold.

We will show for all processes $p \notin \mathcal{F}(t_{start})$, transitioning to the `collect` state from the `idle` state and was therefore not in G_1 or G_2 . To enter the `collect` state, p must have received a message from a process q already in the graph. So $parent_p = q$ and p is a leaf, preserving the tree structure. Processes are only removed from

G_2 when they enter the `distribute` state. A process p entering the `distribute` state must have received a message from all its children, including any process q that considers p its parent. Since such a q only sends a message to p after entering the `distribute` state, $q \notin V_2$. Then only leaves are removed from G_2, preserving the tree structure.

Notice that when all process enter the `collect` state, G_1 is a spanning tree of all processes not in $\mathcal{F}(t_{start})$.

To show that the first phase terminates, we make use of a *metric*. Define the metric M as the tuple $(|\{p \in \Pi \setminus \mathcal{F}(t_{start}) | state_p = \texttt{distribute}\}|, |\{p \in \Pi | state_p = \texttt{collect}|)$[3]. Let values of M be lexicographically ordered. Notice M is bounded above (by $(|\Pi \setminus \mathcal{F}(t_{start})|, 0)$).

We show that if $M = (k, j)$ then eventually $M > (k, j)$ if $M \neq (|\Pi \setminus \mathcal{F}(t_start)|, 0)$.

Let $M = (k, j)$ where $j > 0$ and consider a process $p \notin \mathcal{F}(t_{start})$ such that $state_p \neq \texttt{distribute}$. There are two cases:

- p is `idle`. Consider any path to p from the $root$ process following the neighbor sets of processes. From the assumptions there is such a path, and since $root$ initiated the computation there is at least one process on any path in G_2. Let $q \in V_2$ be the last such process on a path. Then there is a process $v \in N_q$ such that $v \notin V_2$ (and therefore is `idle`). Since $q \in V_2$, $state_q = \texttt{collect}$. Then q has sent or will send a message to v. So eventually $v \in V_2$ and the metric increases.
- p is in the `collect` state. Then $p \in V_1$ and $p \in V_2$. If there is another process q such that $state_q = \texttt{idle}$ then the previous case applies. So assume there does not exist such a q. Since G_1 is a tree and G_2 is a sub-tree of G_1, let q be the last process on a path from $root$ through p such that $q \in V_2$. Since q is the last process on the path, any node (if any) which considered q a parent has entered the `distribute` state and therefore sent q a message. Also, since all processes are in the `collect` or `distribute` state, all processes which did not consider q a parent sent q a message. Then q will eventually receive these messages and enter the `distribute` state, increasing the metric.

Then M eventually increases. Since M is bounded above, eventually all processes are in the `distribute` state.

For the second phase of the algorithm, notice that if the $root$ process is in the `distribute` state, $children_{root} = \emptyset$ and $root$ sends X_{root} to its neighbors. When a process p in the `distribute` state receives a message with a set of processes N, it assigns $X_p = N$, sends the message to its neighbors, and enters the `idle` state.

The proof that all processes receive the set of processes sent by $root$ is similar to the proof that the first phase terminates and is omitted. Since all processes eventually receive the set of processes sent by $root$, all processes eventually enter the `idle` state and the MD algorithm terminates. □

Lemma 7.3: Assume the root process begins the MD algorithm (by entering the `collect` state) at time t_{start} and the last process enters the `distribute` at time t_{end} and for all t such that $t_{start} \leq t \leq t_{end}$ and for all p such that $p \notin \mathcal{F}(t_{start})$ the following hold:

- $\mathcal{F}(t) = \mathcal{F}(t_{start})$.
- $\mathcal{C}(p, t) = \mathcal{C}(p, t_{start})$.
- $N_p(t) = N_p(t_{start})$ and $S_p(t) = S_p(t_{start})$.
- If $q \in N_p(t) \setminus S_p(t)$ then $q \in \mathcal{C}(p, t)$ and $p \in N_q(t) \setminus S_q(t)$.
- $(\exists p_0 \ldots p_n : p_0 = root \wedge p_n = p : p_1 \in N_{p_0}(t) \backslash S_{p_0}(t) \wedge p_2 \in N_{p_1}(t) \backslash S_{p_1}(t) \wedge \ldots \wedge p_n \in N_{p_{n-1}} \backslash S_{p_{n-1}})$.

Then $X = \bigcup_{p \in \Pi \setminus \mathcal{F}(t_{start})} N_p(t_{end} \setminus S_p(t_{end}))$ at the root process.

Proof: Assume the root process begins the MD algorithm at time t_{start} and the last process enters the `distribute` at time t_{end} and the properties listed above hold for all t such that $t_{start} \leq t \leq t_{end}$.

First notice, as in the previous Lemma, that the graph induced by the parent relation for processes in the `distribute` state at time t_{end} forms a spanning tree. A *leaf* is a process that no other process considers its parent. A *path* is a sequence of processes starting with a leaf and following the parent relation.

We will show that at time t_{end} the last process p on any path is in a state where $\bigcup_{u \in path} (N_u(t_{end}) \backslash S_u(t_{end})) \subseteq X_p$ by induction on the length of the path. In the following assume the time is t_{end}.

[3]where $|A|$ is the cardinality of a set A

For paths of length 1 observe that a process p assigns $X = N \cup (N_p \setminus S_p)$ before assigning $state =$ `distribute`.

Now assume the property holds for paths of length i. We will show that it holds for paths of length $i{+}1$. Let p be the last process in a path of length $i{+}1$. Since $state_p =$ `distribute`, it has received messages from all its children, including the child process preceding it on the path, say q. Since q is the last process in the path before p, it is the last process on a path of length i. Then, by the induction hypothesis, $\bigcup_{u \in (path \setminus \{p\})} (N_u(t_{end}) \setminus S_u(t_{end})) \subseteq X_q$. Since $state_p =$ `distribute`, q has sent a message containing X_q to p and p has assigned $X_p = X_q \cup (N_p \setminus S_p)$. Then $X_p \supseteq \bigcup_{u \in (path \setminus \{p\})} (N_u(t_{end}) \setminus S_u(t_{end})) \cup (N_p(t_{end}) \setminus S_p(t_{end}))$. Then $\bigcup_{u \in path} (N_u(t_{end}) \setminus S_p(t_{end})) \subseteq X_p$.

Then the property holds for all paths.

The root process, say p, is the last process on all paths. Then for all paths, $\bigcup_{u \in path} (N_u(t_{end}) \setminus S_u(t_{end})) \subseteq X_p$ at time t_{end}.

Observe that the MD algorithm on a process u never adds a process $v \in S_u$ or $v \notin N_u$ to X_u. Then, since N_u and S_u remain constant between t_{start} and t_{end} and all processes are in a path, it is the case that at time t_{end}, $X_p = \bigcup_{u \in \Pi \setminus \mathcal{F}(t_{end})} (N_u(t_{end}) \setminus S_u(t_{end}))$ □

Theorem 7.1: If $Q(\mathcal{C})$ then MD satisfies the second *Locality* property of the mobility-aware failure detector specification.

Proof: Assume $Q(\mathcal{C})$.

$\Rightarrow$ Direction. First note that MD_p only removes a process from N_p if it is in S_p. Therefore, after t_q given by Lemma 7.1, if q is removed from N_p at time $t \geq t_q$ then either $q \in F(t)$ or $q \notin \mathcal{C}(p,t)$. Since by Lemma 7.1 correct processes in the communication set of p are infinitely often added to N_p and these neighbors are never removed, there is a time $t \geq t_q$ such that if $q \in \mathcal{C}(p,t)$ then $q \in N_p(t)$ and remains so thereafter.

$\Leftarrow$ Direction. By contrapositive. We show that there is a time t after which if $q \notin \mathcal{C}(p,t)$ then $q \notin N_p(t)$ for all correct processes p and q.

First, we know by the forward implication that there is a time $t \geq t_q$ such that for each pair of correct processes p and q if $q \in \mathcal{C}(p,t)$ it is the case that $q \in N_p(t)$. Furthermore, since this t is greater than t_q, $q \notin S_p(t)$.

Now, since $Q(\mathcal{C})$ holds and by the definition of $\mathcal{D}$, there is a time t' after which q will not be added to N_p for any p such that $q \notin \mathcal{C}(p,t')$.

If the MD algorithm is initiated after these times described, the conditions listed in Lemma 7.3 are met. Then, by Lemma 7.3, $X = \bigcup_{p \in \Pi \setminus \mathcal{F}(t_{end})} (N_p \setminus S_p)$ at the root when all processes are in the `distribute` state.

Since for all correct processes q there exists a correct process p such that $q \in N_p$ and $q \notin S_p$ at that point, $q \in \bigcup_{p \in \Pi \setminus \mathcal{F}(t_{end})} (N_p \setminus S_p)$. By Lemma 7.2, MD terminates and N is distributed to all correct processes.

Suppose there exists a process p that receives X at time t and it is the case that $q \in N_p$ and $q \notin \mathcal{C}(p,t)$. By Lemma 7.1, $q \in S_p$. By the program text, it is the case that q will be removed from N_p.

Therefore MD satisfies the second Locality property of the mobility-aware failure detector specification algorithm. □

The Accuracy property follows directly from the second Locality property and Lemma 7.1.

Then MD is a correct implementation of a mobility-aware failure detector.

VIII. Discussion

A. Performance of MD

The MD algorithm is an implementation of the mobility-aware failure detector specification. There are several refinements of the MD algorithm that may be better suited for detecting process movement in a network where the cost of communication may be quite high (such as a sensor network).

The MD algorithm is a diffusing computation. As a coarse-grained analysis, notice that for a single instance of the computation (beginning with the *root* process entering the `collect` state and ending with all processes

entering the `idle` state), the message complexity is $O(|\Pi|^2)$ —each process sends each message to all its neighbors. In each phase of the algorithm, the message size is $O(\Pi)$.

The detection time T_D, mistake recurrence time T_{MR}, and mistake duration time T_M [5] of the MD algorithm are the same as the $\diamond\mathcal{P}_l^m$ algorithm presented by Sridhar [16] (with the adjustment in cost for the missing second phase of the $\diamond\mathcal{P}_l^m$ algorithm). If the diameter of the graph induced in the first phase of the MD algorithm is δ, τ^m is the average-case message transmission delay, the duration in time between two consecutive runs of the algorithm is ρ_g, the probability that the MD algorithm terminates in a given run is p_t, and $T_D(\mathcal{H})$ is the detection time of $\mathcal{H}$, then the average mistake duration of MD is $\frac{\rho_g+3\cdot\delta\cdot\tau^m}{1-p_t}+T_D(\mathcal{H})$.

The first opportunity for optimization is the size of messages: since the size of the messages exchanged depends on the size of processes' neighbor sets and suspect sets, the message size may be reduced if the number of suspected processes in the system is low compared to the number of unsuspected processes. For example, if process movement or failure is a rare occurrence and the local failure detector used by each process is reliable (in the sense that its mistake duration time is low and its mistake recurrence time is high), it may be more efficient to exchange the suspect sets of processes and a set of processes "exonerated" by other processes.

The second opportunity for optimization is in reducing the total number of messages sent to perform the computation. Since the algorithm is run periodically, it is possible to pipeline the process of collecting information about processes neighbors with distributing the set collected in the previous run of the algorithm. That is, the second phase of the algorithm may be eliminated if the $root$ process distributes the value of X_{root} collected in the previous run of the first phase of the algorithm when it initiates the computation. This optimization reduces the overall number of messages sent at the cost of increasing message size and increasing the response time of the algorithm: processes must wait until the algorithm is run again before the results of the previous run are distributed.

Another possible optimization is to take into account the amount of time a process has been suspected by another process. Our algorithm may remove processes from its neighbor set which have not failed nor moved; if a process were to only remove a process from its neighbor set if another process has "heard from" the suspected process at a later time, the number of occurrences of this can be reduced. Without a global clock, this can be done by keeping track of the duration of time elapsed between the last time a process heard from a process or began to suspect it and the current time.

B. Expected-Case Behavior

The MD algorithm is correct; it guarantees the weaker properties of the mobility-aware failure detector specification. However, in practice, the observed behavior of the algorithm may be that of the stronger $\diamond\mathcal{P}_l^m$. We saw in section IV that the obstacle that prevented the $\diamond\mathcal{P}_l^m$ specification from being implementable was that processes may change their communication pattern and fail without being detected by other processes. If it can be guaranteed that processes which move are discovered before they fail (*i.e.*, processes exhibit a stronger notion of failure than fail-stop), then the MD algorithm satisfies the stronger specification.

In addition, the algorithm does not require quiescence in order to provide an accurate picture of the network; if the rate of change in the communication pattern is less than the time required for a run of the algorithm *and* the response times of the components implementing the discovery and failure histories have a "fast enough" response time to provide a process with a updated view of which processes a node can communicate with. In practice, these requirements are not difficult to ensure ([18]).

IX. Conclusions

In networks where the communication topology is dynamic and processes may fail, it is important for processes to be able to determine whether or not a process has moved or failed, especially if resources are shared between processes in a neighborhood. However, the previously proposed $\diamond\mathcal{P}_l^m$ failure detector, is not implementable without a stronger model of failure than fail stop or assumptions about the network architecture.

We have presented a model of computation that captures process mobility and failure. Our model is useful not only for reasoning about mobility and failure (as demonstrated by our proof of impossibility of $\diamond\mathcal{P}_l^m$), but also provides algorithm designers with an abstract view of an environment where mobility and failure, as well as mechanisms for detecting mobility and failure, are present.

We also presented a revised specification of a mobility-aware failure detector that weakens the properties of $\diamond\mathcal{P}_l^m$ while still providing client systems with guarantees that can be used to prevent and limit starvation.

Finally, we presented MD, an implementation of a mobility-aware failure detector. The version of MD presented, though simple, may be optimized for different operating conditions and under certain realistic circumstances, exhibits the behavior of $\diamond\mathcal{P}_l^m$.

This is not to say that we completely addressed the issue of detecting the difference between failure and mobility within the model we presented, though. Future work may explore what properties a mobility-aware failure detector can guarantee given more assumptions about the operating environment's network architecture and analyze the performance of refinements of MD in different networks with different models of the cost of communication.

REFERENCES

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Workshop on Distributed Algorithms*, pages 126–140, 1997.

[2] Romain Boichat, Partha Dutta, and Rachid Guerraoui. Asynchronous leasing. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 180, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158, Vancouver, BC, Canada, 1992. ACM Press.

[4] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, page 191, Washington, DC, USA, 2000. IEEE Computer Society.

[6] Andrew R. Dalton, Jason O. Hallstrom, Hamza A. Zia, and Nigamanth Sridhar. Improving network link quality in embedded wireless systems. In *Proceedings of the 3rd Workshop on Dependable Embedded Systems*, pages 43–48, Leeds, UK, Oct 2006.

[7] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.

[8] Edsger W. Dijkstra and C.S.Scholten. Termination detection for diffusing computations. *Inf. Proc. Letters*, 11(1):1–4, 1980.

[9] Christof Fetzer and Karin Högstedt. Rejuvenation and failure detection in partitionable systems. In *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, page 154, Washington, DC, USA, 2001. IEEE Computer Society.

[10] Christof Fetzer, Ulrich Schmid, and Martin Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 271–280, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[12] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors, 2001.

[13] Jayadev Misra and K. M. Chandy. Termination detection of diffusing computations in communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 4(1):37–43, 1982.

[14] Scott M. Pike and Paolo A. G. Sivilotti. Dining philosophers with crash locality 1. *icdcs*, 00:22–29, 2004.

[15] Paolo A.G. Sivilotti, Scott M. Pike, and Nigamanth Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED Internation Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529. IASTED/ACTA Press, November 2000.

[16] Nigamanth Sridhar. Decentralized local failure detection in dynamic distributed systems. In *Proceedings of the 25th IEEE International Symposium on Reliable Distributed Systems (*SRDS '06*)*, pages 143–152, Leeds, UK, October 2006.

[17] Lynda Temal and Denis Conan. Failure, connectivity and disconnection detectors. In *UbiMob '04: Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, pages 90–97, New York, NY, USA, 2004. ACM.

[18] Hamza A. Zia, Nigamanth Sridhar, and Shivakumar Sastry. Failure detectors for wireless sensor-actuator networks. Technical Report CSU-ECE-TR-07-04, Cleveland State University, Electrical and Computer Engineering, 2007.