Decentralized Local Failure Detection in Dynamic Distributed Systems

Nigamanth Sridhar
Electrical and Computer Engineering, Cleveland State University
2121 Euclid Ave, Cleveland OH 44115 USA
n.sridhar1@csuohio.edu

Abstract

A failure detector is an important building block when constructing fault-tolerant distributed systems. In asynchronous distributed systems, failed processes are often indistinguishable from slow processes. A failure detector is an oracle that can intelligently suspect processes to have failed. Different classes of failure detectors have been proposed to solve different kinds of problems. Almost all of this work is focussed on global failure detection, and moreover, in systems that do not contain mobile nodes or include dynamic topologies. In this paper, we present $\diamond\mathcal{P}_\ell^m$ — a local failure detector that can tolerate mobility and topology changes. This means that $\diamond\mathcal{P}_\ell^m$ can distinguish between a failed process and a process that has moved away from its original location. We also establish an upper bound on the duration for which a process wrongly suspects a node that has moved away from its neighborhood. We support our theoretical results with experimental findings from an implementation of this algorithm for sensor networks.

1 Introduction

Failure detectors are important building blocks for constructing fault-tolerant distributed systems. Introduced first by Chandra and Toueg in [5] as a way of overcoming the "FLP result" [13], failure detectors have evolved over the years to be used in a variety of different ways to solve various problems in distributed systems. In order to solve problems such as consensus, atomic broadcast, etc., each process needs to determine if every other process in the entire network is alive. The failure detectors presented in [1, 3, 16, 17, 25] are all focused on this problem. These failure detectors solve the problem of *global failure detection*. The location of a node p does not impact whether other processes in the network are interested in p's health or not. Regardless of where p is in the network, all other processes need to accurately determine if p has failed.

Motivating Example. However, there are applications that require only *local failure detection*—the health of only those nodes in the immediate neighborhood are of interest. *Failure locality* is a metric that measures the impact of a fault in a single node on the rest of a distributed system. Failure locality is measured in the number of hops from the failed node. For example, if a fault in a node affects nodes that are n hops from it (its n-*neighborhood*), the failure locality of the algorithm is n.

Failure locality has been studied in the context of dining philosophers algorithms. In purely asynchronous distributed systems, the optimal failure locality that can be achieved by any dining philosophers algorithm for resource allocation is two [7]. This optimal locality is not satisfying, however, because the number of nodes affected by a failed node is exponential in the max degree of the graph.

In [22], an algorithm using a $\diamond\mathcal{P}$ failure detector is presented that transforms the underlying dining philosophers algorithm to failure locality 1. The basic idea is that if a node detects a failure in its immediate neighborhood, it then sacrifices its own local progress in the interest of global progress of the entire network. In this transformation, the function of the failure detector module is strictly restricted to the local neighborhood of each process. If a link between two processes p and q failed, perhaps due to q moving out of the transmission range of p, the failure detector at p should no longer keep track of q.

Why is this such a big deal? What is the problem with p continuing to suspect q? The reason is that if p suspects q to be failed, it is going to sacrifice local progress. However, in reality, q is still alive, but is no longer p's neighbor. The correct way of dealing with this, is for p to distinguish between the process q failing, and the link between p and q failing. Consider Figure 1. In both cases, the nodes inside the circle are in each other's neighborhood. In the picture on the left side, node e has failed, and nodes c, d, and g recognize this failure using the $\diamond\mathcal{P}$ failure detector and sacrifice their claim to the resource, thereby allowing nodes a and b to make progress. In the case on the right side of Figure 1, there is no failure. Node e has simply moved out of range

25th IEEE Symposium on Reliable Distributed Systems (SRDS'06)
0-7695-2677-2/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

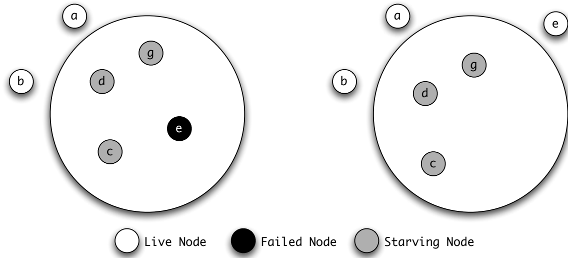


Figure 1: Nodes unable to distinguish failure from mobility

of c, d, and g. However, if the failure detector is based on message passing, the detector may suspect e to have failed. Accordingly, c, d, and g again go into the starve mode. This case, however, is unnecessary (and wrong), and is caused simply by virtue of the fact that the failure detector is not able to distinguish between a failure and mobility.

Context and Contributions. One solution to this problem is to simply require global failure detection always. In that case, regardless of whether e is within the 1-neighborhood of d or not, d's failure detector module will learn that e is still alive, thereby allowing d to make progress as in the normal failure-free case. However, for networks with a large number of nodes, and limited resources available at each node, this is not practical. These constraints are typical of wireless sensor networks [9, 18]. The goal in any typical sensor network application is *dense* instrumentation of the physical world (resulting in a large number of nodes). Economies of scale mandate that the amount of memory resources available to each node be minimal. Moreover, a vast majority of sensor network applications are *localized computations* [10], and should not be required to depend on global knowledge. What we need is a way to store failure information about local neighbors alone, but still be able to detect mobility. *This is our main contribution. We present $\diamond\mathcal{P}_\ell^m$ —an eventually perfect local failure detector that can detect failures locally and distinguish mobility from failure. We show that $\diamond\mathcal{P}_\ell^m$ is correct with respect to strong local completeness and eventual strong local accuracy. We also analyze the quality of service of $\diamond\mathcal{P}_\ell^m$.*

Paper organization. The rest of the paper is organized as follows. Section 2 describes the system model, and introduces the notation we use throughout the paper. In Section 3, we present the design of our local failure detector $\diamond\mathcal{P}_\ell^m$ and prove that it meets the correctness specification in systems with dynamic topologies. In Section 4, we analyze the quality of service guarantees that $\diamond\mathcal{P}_\ell^m$ can provide. In Section 5, we discuss some of the tradeoffs and performance characteristics of our design in the context of an implementation for wireless sensor networks. After presenting some related work in Section 6, we conclude with a summary of our contributions in Section 7.

2 System Model

We consider the system to be comprised of a set Π of nodes (processes). The processes are organized in a multi-hop network. We use $nbrs_p$ to denote the set of neighbors of a process p. Further, we use $\mathcal{N}_p$ to denote the set of processes that p *thinks* are its neighbors. The nodes in the system form a communication graph $\mathbb{G} = (V, E)$ where $V = \Pi$, and E is represents the set of communication links in among nodes in Π. The topology of the graph is dynamic. We denote the diameter of $\mathbb{G}$ as δ.

Each node in the network has its own clock, and there is no global clock in the system. However, for analysis purposes alone, we use a global clock $\mathcal{T}$. We do not assume clock synchronization in the system. Processes communicate with each other by sending messages. Messages may not always be delivered, and may get lost with a probability of p_{ml}. The average message transmission delay is τ^m.

Nodes in the network may be mobile. They may move around, and in so doing, may move in and out of the neighborhood of other nodes. The model of mobility we consider is *passive mobility*—the node that is moving does not know that it is moving. Hence, it cannot notify anyone of its mobility. Also, the nodes move slowly. A node fails by stopping to function—the node stops sending messages to its neighbors. When a node u leaves the neighborhood of another node v, v is not able to distinguish this situation from a situation where u has failed.

The time that the local failure detector module takes to detect that one of its neighbors has failed is T_D. We use td_p to denote the time at which a process p is suspected to have failed. This is a *local* timestamp. We use now to denote the current local timestamp at any process. Note that the value of now at two different processes p and q may be different at the same global time. ts_p is the amount of time that has elapsed since process p was last suspected. Thus at any time, $ts_p = now - td_p$. We assume that although there is no global clock, each process uses the same time unit. Further, we use $\mathcal{WT}_{u,v}$ to denote the amount of time process v waits between communications from u before suspecting u.

We use CORRECT to refer to the set of all correct processes in the system, and CRASHED to refer to the set of all crashed processes in the system. For a given process p, $\mathcal{S}_p$ refers to the set of processes that the failure detector module at p currently suspects to have failed. We say that a process p is in a *good state* (*alive*) if $\mathcal{S}_p$ is empty, and is in a *bad state* (*starving*) if it suspects at least one of its neighbors to have failed. In general, a process can make local progress only if it is in a good state.

IEEE COMPUTER SOCIETY

3 The Design of $\Diamond\mathcal{P}_\ell^m$

$\Diamond\mathcal{P}_\ell^m$ must meet the following specification:

Strong Local Completeness: there is a time after which every process p that crashes is permanently suspected by every correct neighboring process q. Formally:

$$\exists t_{GST} \in \mathcal{T}, \forall p \in \text{CRASHED}, \quad \forall q \in \text{CORRECT}, \\ \forall t \geq t_{GST} : p \in nbrs_q \Rightarrow p \in \mathcal{S}_q$$

Eventual Strong Local Accuracy: there is a time after which correct processes are not suspected by any correct process in the neighborhood. Each process corrects its view of who its neighbors are periodically. Formally:

$$\exists t_{GST} \in \mathcal{T}, \forall t \geq t_{GST}, \forall p, q \in \text{CORRECT} : \\ p \in nbrs_q \Rightarrow p \notin \mathcal{S}_q \wedge \\ \forall q \in \text{CORRECT} : \Box\neg\Box(nbrs_q \neq \mathcal{N}_q)$$

Suspicion Locality: there is a time after which correct processes only suspect processes that are in the local neighborhood. Formally:

$$\exists t_{GST} \in \mathcal{T}, \forall t \geq t_{GST}, \forall p, q \in \text{CORRECT} : \\ q \in \mathcal{S}_p \Rightarrow q \in \mathcal{N}_p$$

Our implementation of $\Diamond\mathcal{P}_\ell^m$ consists of two independent layers. The first layer is the local failure detector that builds a suspect list from among the neighbors of a given node. We call this the *Local Failure Detection Layer* (LFD). The second layer is the one that detects mobility of nodes across the network. We call this the *Mobility Detection Layer* (MD). Together, the two layers satisfy the above specification in mobile networks. We denote the composition of the two layers as $\Diamond\mathcal{P}_\ell^m = \text{LFD} \circ \text{MD}$.

3.1 Local Failure Detection Layer

At each node p, the failure detector's LFD layer keeps track of which processes in $nbrs_p$ are suspected to be failed. Moreover, it also keeps track of when each of these processes were most recently suspected. We denote this time as td_q for suspected process q. Note that this timestamp is a *local* timestamp. This layer works like any other failure detector does—at any point, the process can query its failure detector module for the current list of suspects.

The LFD layer can be implemented using any of several known $\Diamond\mathcal{P}$ algorithms for failure detection. Depending upon the needs and constraints of a given application, the designer may choose from one of several strategies for building the suspect list from among a node's neighbors. This layer is not concerned with mobility of nodes. The main concern of this layer is to satisfy the correctness specification as though there were a static topology. Some popular strategies are listed below:

Heartbeats [1]. Every node sends out an *"I am alive"* message to every one of its neighbors. Every node also maintains a list of its neighbors that have "checked in" by way of an *"I am alive"* message. If a node p does not hear from q for a specified period of time, p assumes that q has failed, and therefore adds q to its suspect list. Supposing at a later point in time p does receive q's heartbeat message, p realizes its mistake, and removes q from the suspect list.

Adaptive Timeouts [12]. Each node p maintains an *adaptive timeout* for each neighboring process q. If p suspects q wrongly after, say, time $\mathcal{WT}_{q,p}$, and later hears from q after $delay_q$, then p updates the timeout period for q to be long enough so that this mistake is not repeated. (The new $\mathcal{WT}_{q,p}$ is now at least $\mathcal{WT}_{q,p} + delay_q$.) Each process thereby keeps extending timeouts until such a time when the timeouts are long enough to account for all forms of incidental delays.

Pinging [16]. When a process queries its failure detector module for the current suspect list, the failure detector sends to each neighboring process an *"Are you alive?"* ping message. If it receives a response in the form of an *"I am alive"* message within a specified time, the neighbor is not added to the suspect list, otherwise it is added. The message complexity of this strategy is twice that of the heartbeat strategy, except that the number of times the detection cycle has to occur can be greatly reduced, thereby reducing the overall message complexity.

Leases [4]. In applications in which nodes sleep for most of the time (as with sensornets), none of the strategies listed above make sense. In this context, the roles could be reversed. Each process p sends to each neighbor q an *"I am alive"* message. In addition, p also sends to q a request for a *lease* for some duration ld_p. Now, p can go to sleep, and all it has to do is wake up some time before ld_p expires, and send a request for lease renewal to all of its neighbors. This is the strategy we use in this paper to implement local failure detection .

3.1.1 Implementing LFD

The LFD layer can use any implementation $\mathcal{FD}$ of a correct local failure detector. The only requirement that the MD layer imposes on LFD in addition is that when processes are added to the suspect list, they also be time-stamped. Supposing the particular implementation that is available does not time-stamp suspects, this needs to be added to the $\mathcal{FD}$ implementation. This transformation is a monotonic addition of functionality—it does not modify $\mathcal{FD}$. A simple version of this transformation is presented in Figure 2.

Lemma 1. *The transformation* TimeStamp *preserves the correctness of the underlying failure detector* $\mathcal{FD}$*.*

Proof. The transformation action does not update any of the variables in $\mathcal{FD}$. Further, it does not in any way influ-

IEEE COMPUTER SOCIETY

```
    Transformation TimeStamp_u
1: S_u^ts := ∅

2: upon suspecting v do
3:     S_u^ts := S_u^ts ∪ ⟨v, now⟩
   end
```

Figure 2: Transformation to time-stamp suspects

ence the decisions that $\mathcal{FD}$ makes in order to determine if a neighbor should be suspected or not. Therefore, *TimeStamp* is a correctness-preserving transformation of $\mathcal{FD}$. □

Lemma 2. *The* LFD *layer does not influence the way the underlying failure detector* $\mathcal{FD}$ *behaves.*

Proof. The LFD layer is a wrapper that only augments the suspect list with extra information about suspected neighbors. There is no change to the existing actions that $\mathcal{FD}$ takes. Therefore, the list of suspects that LFD outputs contains exactly those processes that $\mathcal{FD}$ outputs. □

3.2 Mobility Detection Layer

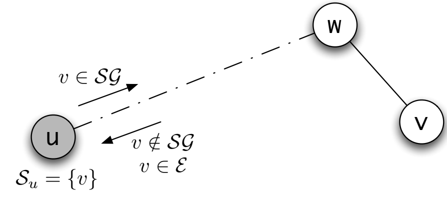
We now discuss the second layer of our $\diamond\mathcal{P}_\ell^m$ implementation. Every process u in Π executes this layer in order to share *its view of failed nodes* with the rest of the network, and to correct its suspect set based on what other nodes in the network can see.

Figure 3 shows the implementation of one round in the $SuspectSharing$ component in the MD layer of $\diamond\mathcal{P}_\ell^m$. Every so often, some process u initiates a gossip diffusing computation [8]. Note that there is only one active gossip in the network at any given time. At the time of deployment, some node is designated as the initiator. The initiator for subsequent rounds is nominated at the end of each round, as described later in this section. This is shown in lines 2—7. The message that is sent out is a pair that comprises:

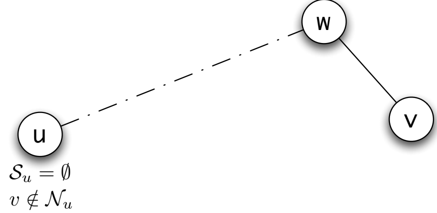
$\mathcal{SG}$ — a tuple containing (i) the set of suspects x that the process u maintains, (ii) the duration (ts_x) for which each process has been suspected, (iii) the id of the process that suspects x (u in this case), and (iv) the number of hops for which this process x has been in $\mathcal{SG}$ (0 initially), and

$\mathcal{E}$ — a set of exonerated processes (initially empty).

When an idle process u receives the gossip message from some neighbor v (lines 8—25), u sets v to be its parent, and transitions into the active state. It then examines the suspect group that is contained in the message ($\mathcal{SG}$). It compares the incoming suspect set with its own ($\mathcal{S}_u$) to see if there are any conflicting entries. If $\mathcal{SG}$ contains any process x that is in



(a) u suspects v, who is now a neighbor of w



(b) u no longer suspects v, who has been exonerated by w

Figure 4: v is exonerated and u is restored to good state

u's neighborhood, but is not in $\mathcal{S}_u$, then u looks at how long the process has been suspected[1]. Based on this, the failure detector module at u decides whether x should remain in $\mathcal{SG}$ or not. If the time elapsed since the last communication[2] u received from x is smaller than ts_x and $\mathcal{WT}_{x,u}$, then u *exonerates* x. This process x is removed from $\mathcal{SG}$ and placed in $\mathcal{E}$ (lines 12—15). This situation can occur when x has moved from the neighborhood of the suspector σ into the neighborhood of u, and is shown in Figure 4.

In the figure, the solid line represents direct neighborhood, while the dashed line represents a multi-hop path. So v and w are neighbors, while there is a path from u to w in the graph. Figure 4a shows u in a bad state, since it suspects v. This means that at some point in the past (and after the last round of gossip finished), u and v were neighbors. After v moved away, u being unable to distinguish that from v's failure begins to suspect v. When w sees that u suspects v to be failed, it exonerates v because it knows v to still be alive. When the message diffuses back to u, it can transition back to a good state after removing v from $\mathcal{S}_u$.

Further, if the suspector process σ is not u's neighbor, the distance from the suspector (d_σ) is incremented by 1 (line 17). If x is in $\mathcal{S}_u$, the entry for x in $\mathcal{SG}$ is updated with u's information — u is the "most recent suspector".

Once u has updated its local suspect set based on $\mathcal{SG}$, it adds the processes in (the newly updated) $\mathcal{S}_u$ to $\mathcal{SG}$. For each suspect, the duration for which it has been suspected along with u's own identity are also added (lines 18—19).

[1] Note that the suspect group only carries around the *duration* for which a process has been suspected. Therefore it is not required that all nodes in the network be synchronized, as long as all nodes use the same *time unit*.

[2] This can be obtained from the LFD layer.

IEEE COMPUTER SOCIETY

```
    Program SuspectSharing_u
 1: initially state_u = idle ∧ parent_u = u ∧ C_u = ∅

 2: if initiator then                                              {* initiator starts gossip *}
 3:     state_u := active
 4:     SG := {x : x ∈ S_u : ⟨x, ts_x, u, 0⟩}                      {* Prepare suspect group *}
 5:     E := ∅
 6:     C_u := N_u − S_u
 7:     send ⟨SG, E⟩ to all w :: w ∈ N_u                {* Send gossip message to all neighbors *}

 8: upon (receive ⟨SG, E⟩ from v) and (state_u = idle) do
 9:     parent_u := v
10:     state_u := active
11:     for each ⟨x, ts_x, σ, d_σ⟩ ∈ SG do
12:         if (x ∈ N_u) ∧ (x ∉ S_u) then         {* Some other process suspects a live neighbor *}
13:             if (now − lhf_x < ts_x) then                                  {* x is alive *}
14:                 SG := SG − ⟨x, ts_x, σ, d_σ⟩                  {* Remove x from suspect group *}
15:                 E := E ∪ {x}                                               {* Exonerate x *}
16:         if x ∈ S_u then σ := u; d_σ := 0
17:         else if σ ∉ N_u then d_σ := d_σ + 1                {* Update distance from suspector *}
18:     for each ⟨x, td_x⟩ ∈ S_u do
19:         if x ∉ SG ∧ x ∉ E then SG := SG ∪ ⟨x, ts_x, u, 0⟩ {* Add local suspects to suspect group *}
20:     if (N_u − {parent_u}) ≠ ∅ then
21:         C_u := {w :: w ∈ N_u ∧ w ∉ S_u ∧ w ≠ parent_u}
22:         send ⟨SG, E⟩ to all w :: w ∈ C_u ∨ w ∈ S_u                        {* Propagate gossip *}
23:     else                                           {* Leaf node, so respond to parent immediately *}
24:         state_u := complete
25:         if ¬initiator then send ⟨SG, E⟩ to parent_u

26: upon (receive ⟨SG, E⟩ from v) and (state_u = active) do
27:     if v ∈ C_u then C_u := C_u − {v}
28:     for each x ∈ E do
29:         if x ∈ S_u then
30:             S_u := S_u − {x}                        {* Remove exonerated nodes from suspect list*}
31:             N_u := N_u − {x}                                            {* x is not a neighbor *}
32:     for each ⟨x, ts_x, σ, d_σ⟩ ∈ SG do
33:         if (x ∈ N_u) ∧ (u ≠ σ) ∧ (d_σ > 2) then               {* x is suspected more than 2 hops away *}
34:             N_u := N_u − {x}                                            {* x is not a neighbor *}
35:             if x ∈ S_u then S_u := S_u − {x}                            {* x is not a suspect *}
36:     C_u = {x :: x ∈ C_u ∧ x ∉ S_u}                   {* Update set of live neighbors yet to respond *}
37:     if C_u = ∅ ∧ ¬initiator then             {* Heard back from all live neighbors; respond to parent *}
38:         state_u := complete
39:         send ⟨SG, E⟩ to parent_u
```

Figure 3: Algorithm for sharing suspect information in the MD layer executing at every process u

IEEE COMPUTER SOCIETY

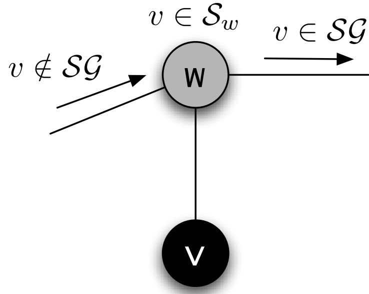


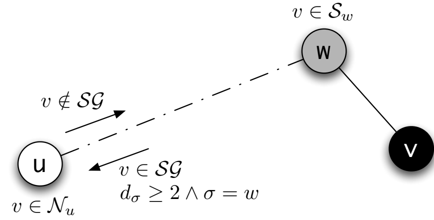
Figure 5: w adds v, which it suspects, to $\mathcal{SG}$ before propagating $\mathcal{SG}$

This situation is shown in Figure 5.

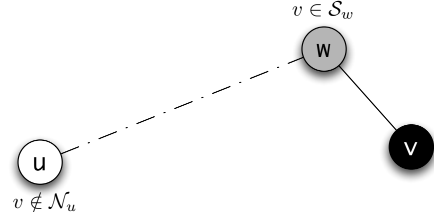
The diffusing computation eventually reaches the edge of the communication graph. When a node receives the gossip message, but does not have any other neighbors to send it to, it sends the updated gossip message back to the neighbor from whence it received the message (its *parent* in the current round of gossip) as shown in line 25. Once again, each node updates its suspect set with respect to $\mathcal{SG}$ and $\mathcal{E}$ contained in the message. Once a node has heard back from all of its children (for that round of gossip), it sends the updated gossip message to its parent. Note that each node only waits for responses from correct neighbors. The round of gossip ends when the process that initiated it has heard from its immediate neighborhood. So at the end of each round of gossip, each node has an updated view of its local neighborhood and who is alive, and who isn't.

The growing phase of the gossip is used to exonerate processes—a process w lower in the gossip propagation tree exonerates a process v that is suspected by one of its ancestors u. Then, during the shrinking phase, u corrects its suspect list and its neighbor list to remove v (as shown in Figure 4). However, even if v has not been exonerated, u would still need to remove v from its neighbor list. Otherwise, LFD_u will continue to suspect v, and u will remain in the bad state unable to make local progress even if all of its other neighbors are alive. Consider Figure 6. u considers v to be its neighbor, but since the last time v communicated with u, it has moved away. Suppose that v entered some other node w's neighborhood, and then failed. w now suspects u, and if the $\mathcal{SG}$ that it receives in the gossip message does not contain v, it adds it. In the shrinking phase, when u sees that v is being suspected by a different process (perhaps for a time that is shorter than $\mathcal{WT}_{v,u}$, which is why LFD_u may not suspect v yet), u realizes that v is no longer its neighbor. So even though v may have actually crashed, u can remove v from $\mathcal{N}_u$, thereby preventing LFD_u from ever suspecting v and putting u in a bad state.

When the initiator completes a round of gossip, it nominates one of its correct neighbors to initiate the next round.



(a) u does not suspect v yet, but w does



(b) u has realized v is not a neighbor anymore, and so will not suspect v

Figure 6: Process u discovers that although v is crashed, v is no longer a neighbor, and hence u can stay in good state

3.3 Correctness of $\diamond\mathcal{P}_\ell^m$

We begin our proof of correctness of $\diamond\mathcal{P}_\ell^m$ with some preliminary lemmas.

Lemma 3. *The* LFD *layer satisfies strong local completeness and eventual strong local accuracy.*

Proof. The LFD layer is essentially any known $\diamond\mathcal{P}$ failure detection algorithm. [19] presents correct local failure detection algorithms. The only modification that LFD makes is to add *TimeStamp*. According to Lemma 1, *TimeStamp* is a correctness-preserving transformation. Therefore, as long as $\mathcal{FD}$ satisfies strong local completeness and eventual strong local accuracy, so does LFD. □

Lemma 4. *The* MD *layer terminates.*

Proof. At its core, the MD layer includes an implementation of gossip. The only messages that are exchanged in the MD layer are messages that diffuse the gossip message through the network. The gossip algorithm itself does not depend on the actual content of the message being diffused. We refer the reader to [8, 23] for a proof that the diffusing computation terminates. Given that the mechanism for diffusing the messages terminates, and since the MD layer does not (i) generate any new messages of its own, nor does it (ii) destroy any messages that distribute the gossip, we can conclude that the MD layer terminates. Moreover, the termination of gossip is not thwarted by failed or suspected

IEEE COMPUTER SOCIETY

processes. Each node only waits to hear from its correct neighbors before transitioning to the complete state.

What happens when messages are lost? The proofs above assume reliable channels. If messages may be lost, then gossip will not terminate. However, if a particular round of gossip does not terminate, then the same initiator starts the next round of gossip. This does not affect the correctness of the failure detector; only the time to correct mistakes is increased. More details are in Section 4. □

The algorithm presented in this paper (Figure 3) has been simplified to assume that gossip always terminates. In reality, the algorithm needs to account for message losses. If a node u does not hear from one of its children v, but knows that v is not a suspect, then u sends a repeat gossip message. The re-transmissions are purely an artifact of improving the quality of service, and do not affect the correctness of the algorithm. In order to further improve QoS, other strategies [14] can be used to make gossip more efficient.

Lemma 5. *The composition* $\mathsf{LFD} \circ \mathsf{MD}$ *satisfies strong local completeness in systems with mobile nodes.*

Proof. In order for us to prove this statement, we need to show that for all processes u:

L5.1. MD_u does not remove any crashed neighbors from $\mathcal{S}_u$.

L5.2. all crashed processes in $\mathcal{N}_u$ are in $\mathcal{S}_u$, and

There are only two places in MD_u where a process x is removed from the suspect set $\mathcal{S}_u$ (Line 30 and line 35). The first is when the process u finds out that a process x that is suspected to have failed, has been exonerated by some other process. This means that x was originally in the neighborhood of u, and then later moved away into the neighborhood of some other process w. In this case, it is safe to remove x from the suspect set $\mathcal{S}_u$ since x should not have been added to $\mathcal{S}_u$ in the first place. Further, since u has now learned that x is no longer its neighbor, x is also removed from $\mathcal{N}_u$. This guarantees that the same mistake of LFD_u adding x back into $\mathcal{S}_u$ will not be repeated.

In order to see the other situation, consider a scenario where a process q communicates that it is alive to u. It then moves away from u's neighborhood to a different part of the network, and introduces itself to some other process w. Following this, w suspects q based on its local detection rules. In the meantime, u also suspects q (since it has not heard from q for longer than $\mathcal{WT}_{q,u}$). Now, if u receives a message with $\mathcal{SG}$ containing the failed process q, and moreover, if the process σ that suspects q is at least two hops away from u, then u can conclude that q is no longer in its neighborhood. Further, it can remove q from $\mathcal{S}_u$ since q is no longer a neighbor. Again, no crashed *neighbors* are removed from $\mathcal{S}_u$ by MD_u. Thus, **L5.1.** is satisfied.

If the implementation that is being used for LFD_u is one in which each process has a different set of timeouts for each of its neighbors[3], then it could so happen that a process x that is in the neighborhood of u is *not* in $\mathcal{S}_u$ but *is* in $\mathcal{SG}$. What this means is that q has failed, but u has not detected this failure as yet. Consider the following scenario.

The process q communicates with u at some time t_1. Immediately after this, q moves away from u, and enters the neighborhood of some other process w, and initiates communication with w at time t_2. Suppose then that at time $t_2 + \mathcal{WT}_{q,w}$, w adds q to its suspect set $\mathcal{S}_w$. At this point, if the gossip message with $\mathcal{SG}$ comes along, w would add q to $\mathcal{SG}$. The message will then diffuse and at some later point t_3, arrive at u. Now, if $t_3 - t_1 < \mathcal{WT}_{q,u}$, u will still not have added q to its own suspect set $\mathcal{S}_u$. However, u may not have enough information to conclude either that q has failed, or that q has moved away. This is because if w were in the 1- or 2-neighborhood of u, then q may still be a valid neighbor. So MD_u in this case, does not do anything about q. At time $t_1 + \mathcal{WT}_{q,u}$, if u has still not heard from q, then q would be added to $\mathcal{S}_u$ by LFD_u. If q is suspected by a process more than two hops away ($d_\sigma \geq 2$), then q can be removed from $\mathcal{N}_u$. Thus, **L5.2.** is also satisfied. □

Lemma 6. *The composition* $\mathsf{LFD} \circ \mathsf{MD}$ *satisfies eventual strong local accuracy in mobile systems.*

Proof. In order for us to prove this statement, we have to show the following are true for every process u:

L6.1. MD_u does not add correct neighbors to $\mathcal{S}_u$, and

L6.2. MD_u prevents LFD_u from suspecting nodes that are no longer in $\mathcal{N}_u$.

We observe that since MD_u does not include any actions that add processes to $\mathcal{S}_u$, **L6.1.** is trivially true.

LFD_u only adds processes to $\mathcal{S}_u$ that are in the neighborhood (in $\mathcal{N}_u$) and that are suspected. Suppose that u suspects a process x. Upon receiving the gossip message, say that u finds x in $\mathcal{E}$—the set of exonerated processes. This means that although u originally placed x in $\mathcal{SG}$, some other process w in the system has moved x from $\mathcal{SG}$ to $\mathcal{E}$. This means that x is no longer in $nbrs_u$, and must be removed from $\mathcal{N}_u$. Once x has been removed from $\mathcal{N}_u$, it will not be suspected by LFD_u. Thus, **L6.2.** is satisfied. □

Lemma 7. *The* MD *layer ensures that* $(\forall p, q \in \Pi :: q \in \mathcal{N}_p \wedge q \notin nbrs_p)$ *is transient.*

[3] An example of such an implementation is one that uses adaptive timeouts. The amount of time that a process p spends waiting for two different process q and r may be different. Moreover, the wait times for the same process q may be different at two other processes x and y.

IEEE COMPUTER SOCIETY

Proof. Any node that is known not to be a neighbor of a process p is removed from the neighbor list at the end of each gossip round (lines 31 and 34). At the end of each round of gossip the view that p has of who its neighbors are is consistent with its real neighbor set ($\mathcal{N}_p = nbrs_p$). □

From Lemma 7, we can conclude the following, since $\mathcal{N}_p$ is corrected at the end of every round of gossip to be consistent with $nbrs_p$.

Corollary 8. *The* MD *layer ensures that* $(\forall p \in \Pi :: \Box\neg\Box(nbrs_p \neq \mathcal{N}_p))$.

Lemma 9. *The* MD *layer ensures that* $(\forall p, q \in \Pi :: q \in \mathcal{S}_p \wedge q \notin \mathcal{N}_p)$ *is transient.*

Proof. Any node that is known not to be a neighbor of a process p is removed from the suspect list at the end of each gossip round (lines 30 and 35 in Figure 3). Therefore at the end of each round of gossip the suspect set is corrected to include only neighboring nodes that are suspects. □

Theorem 10. $\Diamond\mathcal{P}_\ell^m$ *is a correct eventually perfect local failure detector for mobile systems.*

Proof. Follows from Lemmas 5, 6, and 9 and Cor. 8. □

4 Analyzing the Quality of Service of $\Diamond\mathcal{P}_\ell^m$

Thus far, we have presented the design of our $\Diamond\mathcal{P}_\ell^m$ local failure detector, and established that it meets its specification of strong local completeness and eventual strong local accuracy. In this section, we analyze the quality of service (QoS) of $\Diamond\mathcal{P}_\ell^m$ based on the QoS metrics for failure detectors presented in [6].

In [6], Chen *et al.* define the following primary QoS metrics for failure detectors:

Detection time (T_D). This measures the speed of detection of a failure detector.

Mistake recurrence time (T_{MR}). This measures the time between two consecutive mistakes.

Mistake duration (T_M). This measures the time it takes a failure detector to correct a mistake.

The detection time of $\Diamond\mathcal{P}_\ell^m$ is a local measure, and is completely driven by the choice of failure detector implementation for the LFD layer. Since we have already shown that $\Diamond\mathcal{P}_\ell^m$ does not influence the decision-making of the LFD implementation about which neighbors have failed (Lemma 2), there is no change in detection time either.

Theorem 11. $\Diamond\mathcal{P}_\ell^m$ *preserves the detection speed of the underlying failure detector* $\mathcal{FD}$ *used in the* LFD *layer, i.e.,* $T_D(\Diamond\mathcal{P}_\ell^m) = T_D(\mathcal{FD})$.

Proof. The LFD and MD layers do not add any actions to the underlying failure detector $\mathcal{FD}$ that affect how $\mathcal{FD}$ adds neighbors to the suspect set $\mathcal{S}$. Therefore, the composition LFD $\circ$ MD preserves the decision-making of $\mathcal{FD}$. Thus, the detection speed is preserved. □

The accuracy metrics are more interesting in the case of $\Diamond\mathcal{P}_\ell^m$ considering that the main contribution of $\Diamond\mathcal{P}_\ell^m$ is its ability to correct mistakes caused by mobility. The first accuracy metric—T_{MR}—is also a property of the underlying failure detector in the LFD layer.

Theorem 12. $\Diamond\mathcal{P}_\ell^m$ *preserves the mistake recurrence time of the underlying failure detector* $\mathcal{FD}$ *used in the* LFD *layer, i.e.,* $T_{MR}(\Diamond\mathcal{P}_\ell^m) = T_{MR}(\mathcal{FD})$.

Proof. This theorem holds for the same reasons that Theorem 11 holds—$\mathcal{FD}$ is not modified. Recurrence of mistakes made at the local level are the concern of $\mathcal{FD}$, and hence are not affected by mobility detection. Therefore, the mistake recurrence time is preserved by the composition. □

The mistake duration of the failure detector is affected by the mobility detection layer. In the absence mobility detection, the mistake duration of a local failure detector $\mathcal{FD}$ is ∞. If p moves out of the neighborhood of q, there is no way for q to correct that mistake. However, the MD layer improves this to some finite duration. In the following, ρ_g is the *gossip recurrence time*—the duration of time between two rounds of gossip in the MD layer. This will be a user-supplied parameter, and is typically much smaller than average message rate, *i.e.,* $\rho_g \ll \rho_m$, where ρ_m is the average rate of application messages.

Theorem 13. *The average mistake duration of* $\Diamond\mathcal{P}_\ell^m$ *is bounded above by* $\frac{\rho_g + 2\cdot\delta\cdot\tau^m}{1-p_g} + T_D(\mathcal{FD})$.

Proof. Without loss of generality, let us suppose that the process who wants to correct a mistake is the one that initiates the gossip computation to share suspect lists in the system. For this process u to learn that some wrongly suspected process v is alive in a different part of the system, the entire gossip computation needs to terminate. Suppose that the probability that a round of gossip terminates is p_g.

The process w that can exonerate v could potentially be at the other end of the graph (δ (diameter of graph) hops away. In such a case, the message has to traverse the entire graph twice (once during the growing phase and again when shrinking). Given that τ^m is the average-case message transmission delay at any hop, and the the average-case termination duration for the gossip algorithm is $2 \cdot \delta \cdot \tau^m$, if there were no message loss. In the presence of message losses, the time taken for the exoneration message to get back to the initiator once gossip has been initiated is $\frac{2\cdot\delta\cdot\tau^m}{1-p_g}$. Further, the process may take up to $T_D(\mathcal{FD})$ to detect that v is failed in its neighborhood.

IEEE COMPUTER SOCIETY

The fact that a node p has moved away from the neighborhood of another node q can be detected in a single round of gossip. In the worst case, p could move away just when one round of gossip finishes, meaning that q will have to wait until the next round of gossip terminates before it can realize its mistake in suspecting p. Therefore, $T_M(\diamond\mathcal{P}_\ell^m) \leq \frac{\rho_g + 2\cdot\delta\cdot\tau^m}{1-p_g} + T_D(\mathcal{FD})$. □

The mistake duration can be tuned to whatever level of service the application needs by varying the value of ρ_g. Increasing the rate of executing gossip in the MD layer has the effect of reducing T_M.

The $\diamond\mathcal{P}_\ell^m$ failure detector is based on message passing. Therefore, another important measure of QoS is the extra message complexity introduced by the failure detector.

Theorem 14. *The message complexity of the $\diamond\mathcal{P}_\ell^m$ failure detector is $O(n \cdot e)$ for each round of gossip, where n is the number of nodes, and e is the number of edges.*

Proof. During each round of gossip, the messages sent around in the network form a spanning tree rooted at the initiator node. The gossip message visits every node in the network. Therefore, the additional message complexity introduced by the $\diamond\mathcal{P}_\ell^m$ failure detector is $O(n \cdot e)$. □

5 Discussion

We now take a brief look at the scalability of $\diamond\mathcal{P}_\ell^m$ in real network deployments. First we note that $\diamond\mathcal{P}_\ell^m$ does not increase the number of messages substantially beyond a local failure detector, since the rate of gossip is low, and there is only one gossip round active at any time.

Every time the MD layer initiates a round of gossip the message that is diffused contains the list of *all* the processes that are suspected to be failed. This could be a substantial number. Depending on the target network scenario, an implementation may impose bounds on how many failures it can tolerate (as a percentage of all nodes in the network). A key point is that if these suspected nodes are really crashed, then they are never removed from the suspect list.

One approach to solving this problem could be employ a periodic clean-up protocol that can use some global knowledge in sweeping through the system to remove these crashed processes from the suspect lists at each correct process. The rate at which this operation would be performed (ρ_c) would be much smaller than the gossip rate, *i.e.*, $\rho_c \ll \rho_g \ll \rho_m$. Such rare reconfiguration is not uncommon in typical wireless network deployments, where application may be re-tasked *in situ*. For example, in a wireless sensor network application of the sort described in [21], the average message rate is on the order of once every several minutes. If we wanted to account for mobility of nodes in such an application, then we could configure ρ_g to perhaps be once every several hours, and the clean-up could execute once every month. A further optimization would be to piggy-back the clean-up as part of any network-level re-tasking operation.

As of now, the mobility detection layer assumes that the communication graph remains connected. The algorithm does not handle partitionable networks. This is a direction for our future work. [2] and [11] present some techniques for dealing with failure detection in partitionable systems. We expect to try to adapt some of these techniques to $\diamond\mathcal{P}_\ell^m$.

We have implemented the $\diamond\mathcal{P}_\ell^m$ failure detector as a middleware service for wireless sensor networks. The service has been implemented using nesC [15] for Berkeley motes running TinyOS [18]. While the full details of this implementation, and complete experimental analysis of the implementation are outside the scope of this paper, we have sufficient proof-of-concept results from our experiments that match our analytical predictions. We refer the interested reader to [26] for more details.

6 Related Work

Research in the area of failure detectors started with the work on unreliable failure detectors by Chandra and Toueg [5]. In this work, the authors describe different classes of failure detectors and define the correctness specification of detectors in terms of completeness and accuracy. Since then, there has been a lot of research in various kinds of implementations of failure detectors [1,3,16,17,25]. All of this work has been focussed on global failure detection.

In [19], Hutle and Widder present two time-free self-stabilizing algorithms for local failure detection. Their results are presented in the context of sparse networks. However, the algorithms apply equally well to dense networks. The first algorithm they propose requires unbounded amount of space in each process, and the second algorithm (the more realistic one) can do with bounded space if there is a known upper bound on the number of messages in the system. Their work, however, assumes a static topology, and does not tolerate mobility of nodes.

The problem of failure detection in partitionable networks is relevant to our work on mobile networks. In [11], Fetzer and Högstedt present a protocol for failure detection in partitionable systems. They consider systems in which a gateway node that connects a section of the network to another section fails. Their work uses the concept of software rejuvenation [20]—using a Rejuvenation Server to rejuvenate a gateway server if it is detected to be failed. Another approach to failure detection in partitionable networks is presented in [2], where Aguilera *et al.* use the heartbeat failure detector [1] to solve consensus in partitionable networks. Again, their work does not consider mobility of nodes and mistakes caused by mobility.

The work that is closest to our work is the work by Temal and Conan on disconnection detectors presented in [24].

IEEE COMPUTER SOCIETY

They consider both voluntary and involuntary disconnection of nodes in a network. The focus of the work is on achieving global failure detection; the algorithm is a way of solving consensus in the presence of disconnections.

7 Conclusion

In this paper, we have presented a solution to the problem of failure detection in the presence of mobility in distributed systems. Most research on failure detectors so far has been targeted at global failure detection—each process p keeps track of the health of every other process q in the system. However, in some deployment contexts, such as in wireless sensor networks, global failure detection is too resource-intensive, and is hence not practical. Although there has been some research in local failure detection, all of this work ignores mobility in systems. They all assume static communication graphs.

In this paper, we have presented the design of an eventually perfect local failure detector that can tolerate mobility of nodes, as long as such mobility does not partition the network. We have shown that our detector is correct; it satisfies strong local completeness and eventual strong local accuracy. Moreover, the $\diamond\mathcal{P}_\ell^m$ detector only keeps information about its immediate neighborhood, thereby reducing the amount of memory needed (which is a scarce resource in sensor nodes). We have conducted a few proof-of-concept experiments to validate $\diamond\mathcal{P}_\ell^m$ and are now in the process of further experimentation to fine-tune the systems aspects.

Acknowledgments. This work has been supported by a grant from Ohio ICE. The author would like to thank Jason Hallstrom and Hamza Zia for their help with this paper.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *WDAG '97*, pages 126–140, London, UK, 1997.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30, 1999.

[3] C. Almeida and P. Veríssimo. Timing failure detection and real-time group communication in real-time systems. In *8th Euromicro Wksp. on Real-Time Systems*, June 1996.

[4] R. Boichat, P. Dutta, and R. Guerraoui. Asynchronous leasing. In *7th IEEE Intl. Wksp. on Object-Oriented Real-Time Dependable Systems (WORDS '02)*, pages 180–187, 2002.

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

[6] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE ToC*, 51(1):13–32, 2002.

[7] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *ACM Symposium on Theory of Computing*, pages 593–602, 1992.

[8] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[9] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[10] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Proc. MOBICOM '99*, pages 263–270, 1999.

[11] C. Fetzer and K. Högstedt. Rejuvenation and failure detection in partitionable systems. In *PRDC '01*, page 154, Washington, DC, USA, 2001. IEEE Comp. Soc.

[12] C. Fetzer, U. Schmid, and M. Susskraut. On the possibility of consensus in asynchronous systems with finite average response times. In *ICDCS '05*, pages 271–280, 2005.

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

[14] B. Garbinato, F. Pedone, and R. Schmidt. An adaptive algorithm for efficient message diffusion in unreliable environments. In *DSN '04*, page 507. IEEE CS, 2004.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03*, pages 1–11, 2003.

[16] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC '01*, pages 170–179, New York, NY, USA, 2001.

[17] N. Hayashibara, A. Cherif, and T. Katayama. Failure detectors for large-scale distributed systems. In *SRDS'02*, 2002.

[18] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX*, pages 93–104. ACM Press, 2000.

[19] M. Hutle and J. Widder. Time free self-stabilizing local failure detection. Research Report 33/2004, TU Wien, Vienna, Austria, 2004.

[20] N. Kolettis and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[21] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02*, pages 88–97, New York, NY, USA, 2002.

[22] S. M. Pike and P. A. Sivilotti. Dining philosophers with crash locality 1. In *ICDCS '04*, pages 22–29. IEEE, 2004.

[23] P. A. G. Sivilotti. Introduction to distributed systems. Lecture Notes. Computer Science and Engineering, The Ohio State University, 2004.

[24] L. Temal and D. Conan. Failure, connectivity and disconnection detectors. In *UbiMob '04: Proc. 1st French-speaking conf. on Mobility and ubiquity computing*, pages 90–97, New York, NY, USA, 2004. ACM Press.

[25] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. In *Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*, 1996.

[26] H. A. Zia, N. Sridhar, and S. Sastry. Abstractions for detecting failures in wireless sensor-actuator networks. Technical report, Electrical and Computer Engineering, Cleveland State University, 2006.

IEEE COMPUTER SOCIETY