View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Cleveland-Marshall College of Law

Cleveland State University
EngagedScholarship@CSU

Electrical Engineering & Computer Science Faculty Publications

Electrical Engineering & Computer Science Department

7-2009

Failure Detectors for Wireless Sensor-Actuator Systems

Hamza A. Zia
Cleveland State University

Nigamanth Sridhar
Clevealand State University, n.sridhar1@csuohio.edu

Shivakumar Sastry
University of Akron

Follow this and additional works at: https://engagedscholarship.csuohio.edu/enece_facpub

Part of the Digital Communications and Networking Commons

How does access to this work benefit you? Let us know!

Publisher's Statement

NOTICE: this is the author's version of a work that was accepted for publication in Ad Hoc Networks. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Ad Hoc Networks, 7, 5, (07-01-2009); 10.1016/j.adhoc.2008.09.003

Original Citation

Zia, H. A., Sridhar, N., , & Sastry, S. (2009). Failure detectors for wireless sensor-actuator systems. Ad Hoc Networks, 7(5), 1001-1013. doi:10.1016/j.adhoc.2008.09.003

Repository Citation

Zia, Hamza A.; Sridhar, Nigamanth; and Sastry, Shivakumar, "Failure Detectors for Wireless Sensor-Actuator Systems" (2009). *Electrical Engineering & Computer Science Faculty Publications*. 64.
https://engagedscholarship.csuohio.edu/enece_facpub/64

This Article is brought to you for free and open access by the Electrical Engineering & Computer Science Department at EngagedScholarship@CSU. It has been accepted for inclusion in Electrical Engineering & Computer Science Faculty Publications by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.

Failure detectors for wireless sensor-actuator systems ☆

Hamza A. Zia [a], Nigamanth Sridhar [a,*], Shivakumar Sastry [b]

[a] Electrical and Computer Engineering, Cleveland State University, 332 Stilwell Hall, 2121 Euclid Avenue, Cleveland, OH 44115, United States
[b] Electrical and Computer Engineering, The University of Akron, United States

1. Introduction

Wireless sensor-actuator systems (WSAS) have emerged as an important platform that enable unprecedented levels of fine-grained visibility and control over our environment [1–6]. There are several small and inexpensive devices that are capable of capturing and processing sensor inputs. In addition, the availability of protocols [7–10] and algorithms that address challenges, high failure-rates, limited resources and large-scale, allow us to develop and deploy effective sensornets [11–14]. In addition, there are several competing software platforms for programming sensor network applications [15–21].

Despite such advances, large-scale networks of sensors are incredibly hard to design and deploy. The time and effort required to design and implement software for such systems are not proportional to the size of the software itself: most of these applications are only a few hundreds of lines of source code, but take a disproportionate amount of time (several weeks, even months) to develop. An important reason for this incongruity is that the set of design criteria that developers must think about is quite different in the context of WSAS than in the context of enterprise systems. As opposed to internet-scale network applications, WSAS applications need to consider fault-tolerance as a *primary* design consideration. Failure is not the exception; it is the norm. This necessitates the designer of a WSAS application to include failure while she thinks about the functional behavior of the application. This causes the design space to be cluttered with functional requirements as well as fault-tolerance requirements.

In this paper, we argue that a reusable class of failure detectors can simplify the design and deployment of sensornets. We focus on a class of failures, i.e. loss of communication links between nodes. The hardware platforms that WSAS applications run on are inherently unreliable; the cost of producing each individual node is extremely low, and consequently, nodes are designed to be dispensable. This places a larger burden on someone designing applications that run on these platforms: applications running on such *unreliable* infrastructures must still function in a *reliable* manner. When a collection of nodes are unable to communicate, the WSAS is unlikely to provide an acceptable level of service. To tolerate such failures, it is

necessary to *detect* faults as and when they occur. Failure detection logic is typically included either directly in the application, or as part of a middleware service that provides neighborhood information. Neither of these cases is optimal. The former approach is tedious, error prone, and does not enable reuse of the failure detection logic. While the latter decouples the failure detection scheme from the application, the scheme is still tightly coupled to neighbor discovery logic. The range of failures, and correspondingly the range of failure detection mechanisms, cannot be effectively addressed in such tightly-coupled designs. We present a design that offers a higher degree of separation between independent concerns.

Contributions. The contributions of this paper are:

- A design for a failure detector that distinguishes between the failure of a communication link and mobility of a node in the WSAS.
- A proof sketch to show that the above detector is correct.
- Novel packaging of the above failure detector as a parameterizable middleware service for application designers.
- Experimental results based on two different failure detection schemes designed using the middleware service. These components were implemented in *nesC* [15] for the *TinyOS* [11] platform, and the experiments were conducted on a testbed of *TelosB* [13] motes.

Paper organization. Following the background and related work discussion in Section 2, we present the first failure detector that can detect node failures in Section 3. In Section 4 we describe the second failure detector that successfully detects failures in dynamic topologies, along with a proof of correctness, and a case study example to illustrate the use of the failure detector middleware. In Section 5, we provide an evaluation of our failure detectors' performance using experiments conducted on an 80-node sensor network testbed. Finally, we conclude with a summary of the contributions and some directions for future work in Section 6.

2. Background and related work

2.1. Failure detection algorithms

The use of *unreliable* failure detectors to implement *reliable* asynchronous distributed systems was first proposed by Chandra and Toueg [22]. The authors present four classes of failures and each class has different completeness and accuracy specifications. Since then, several other implementations of failure detectors have been reported in the literature [23–27].

All failure detectors work in roughly the following way. Each node maintains a list of its neighbors, and for each neighbor, the node keeps an account of whether it is perceived to be alive or failed. Several strategies to solve this problem have been proposed.

The simplest of these strategies involves nodes exchanging *heartbeat* messages [23]. Each node periodically sends out an "I am alive" message to its neighbors. If a node *p* does not hear from some neighbor *q* for some specified length of time, *p* adds *q* to its *suspect list*. If, later on, *p* does receive a heartbeat from *q*, *p* realizes its mistake, and removes *q* from the suspect list. One way of ensuring that the same mistake is not made again, is to modify the timeout duration based on past mistakes. When *p* recovers from a mistake, it extends the timeout duration for *q* to be longer than the time it took for *q*'s heartbeat to arrive. For example, an adaptive timeout-based failure detector is reported in [28].

Instead of forcing every node to continually flood the network with heartbeat messages, some failure detectors follow an "on-demand" approach. When a process queries its failure detector module for the current suspect list, the failure detector sends an "Are you alive?" message to its neighbors. Following this, it waits for some specified timeout period at the end of which it declares a neighbor to be a suspect if no response has been received. If, on the other hand, it receives an "I am alive" message, the neighbor is not added to the suspect list. 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 of the failure detector module. Such ping-based implementations are reported in [24].

Unfortunately, none of these strategies are optimal for WSAS because here, the nodes sleep for most of the time, and are only awake for a few minutes at a time. In such a context, it makes sense to reverse the role of the messages. Each node *p* sends a message to its neighbors requesting a *lease* for some duration. A neighboring node *q* now records this lease, and assumes that *p* is alive for the lease duration. As long as *p* sends another lease request before its lease expires, *q* does not suspect *p*. But if the lease expires before *p* sends a request, then *p* is added to *q*'s suspect list. This strategy is described in [29], and is the strategy we use in this paper for local node failure detection in Section 3.

Hutle and Widder [30] present two time-free self-stabilizing algorithms for local failure detection for sparse networks. These 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) executes within bounded space when 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.

Fetzer and Högstedt [31] 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. This work uses the concept of software rejuvenation [32] – 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 [33], where Aguilera et al. use the heartbeat failure detector [23] to solve consensus in partitionable networks. Their work does not consider mobility of nodes (link failures) and mistakes caused by mobility. However, these works on partitionable networks present some nice ideas that can be combined with the work pre-

sented in this paper to allow our mobility detectors accommodate network partitions as well.

2.2. Failure detection in sensor networks

In the recent past, there has been a considerable amount of work on developing abstractions for managing failures in sensor networks. Hood [34] is an abstraction that allows a node in a sensor network to easily access and interact with other nodes in its neighborhood. The Hood abstraction allows a node to easily share its local state with neighboring nodes. In addition to maintaining a list of one-hop neighbors, Hood also allows applications to create neighbor sets based on other attributes. The Hood abstraction can enhance the benefit of our failure detection components, since the application designer will then be able to define arbitrary neighborhoods.

The *Memento* system [35] provides health monitoring services to wireless sensor network applications that are similar to our own. They focus exclusively on failures of immediately neighboring nodes, and failures caused by radio link irregularities. As part of the network management system, Memento includes a local failure detection scheme that uses ping-and-response to monitor the health status of neighboring nodes. Our failure detection components, particularly the Mobility Detector, can be used in place of this local failure detector to further improve the utility of Memento in tolerating both link irregularities and node mobility.

Elhadef et al. [36,37] present a distributed diagnosis protocol for health management in mobile ad hoc networks. Their health management protocol, called *dynamic-DSDP*, is similar to ours in the sense that they also have a way of disseminating failure information from an initiator node to the rest of the network. The nodes organize themselves into a spanning tree of the graph, and send diagnostic information about their local neighbors back to the root of the spanning tree. While the reason why nodes in our algorithm share suspect information is for them to correct their local views, the purpose in their algorithm is to get a global view of failure information from across the network.

2.3. Classes of failure detectors

Of the four classes of failure detectors presented by Chandra and Toueg [22], the class of detectors we focus on is the *eventually perfect* ($\Diamond\mathcal{P}$) class. Failure detectors in this class satisfy the following specification (reproduced from [22]):

Strong completeness: Eventually every process that crashes is permanently suspected by every correct process.

Eventual strong accuracy: There is a time after which correct processes are not suspected by any correct process.

Since we are interested in *local* failure detection – the detection of failures in a node's immediate neighborhood – as opposed to *global* failure detection, we adapt the specification to reflect this. We denote this class of eventually perfect local failure detectors as $\Diamond\mathcal{P}_\ell$.

Strong local completeness: There is a time after which every process p that crashes is permanently suspected by every correct neighboring process 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.

Suspicion locality: There is a time after which correct processes only suspect processes that are in the local neighborhood.

3. Detecting failure of neighboring nodes

The failure detectors we present in this paper implement the bidirectional interface shown in Fig. 1. Applications may query a detector for the number of nodes that are suspected to have failed in a neighborhood (`numSuspects()`), or the suspected nodes (`getSuspects()`). Applications may also check if a particular node is alive or failed via the predicate `isSuspect(p)`. In addition, for given suspect `p`, the failure detector can also report how long it has been since there was any communication from `p`, `lastHeardFrom(p)`. Applications may use additional information to remove a node from the suspect list (`removeSuspect()`[1].). In addition, the failure detector can notify applications when at least one neighbor, or no neighbor, is suspected.

LeaseFD is the first, lease based, failure detector we present. This detector executes the algorithm depicted in Fig. 2 at every node u of the system. Similar to other TinyOS components, LeaseFD, is wired in to the application, and is provided with one parameter – the lease duration, which is best supplied by the application. Once started, the failure detector component periodically sends lease request messages to each neighbor. When a node p receives a lease request from a neighboring node q, the lease is recorded, and for the duration of the lease, the node q is not suspected. In case, q were already in the suspect list (wrongly suspected), q is now removed the suspect list (mistake is corrected).

As indicated above, a component that can provide a neighborhood abstraction is necessary to use LeaseFD. We implemented a simple beacon-based neighbor discovery protocol suitable for the failure detectors[2]. Our neighbor discovery protocol establishes bi-directional neighborhood relations between nodes.

4. Distinguishing node failure from link failure

Consider the situation when a node p does not receive an "I am Alive" message from a neighbor q. It is not correct to assume that node q has failed because the communication link between p and q may have failed, while q continues to remain operational. 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.

[1] We will demonstrate a use for this command in Section 4.

[2] A different neighborhood abstraction (e.g. Hood [34], abstract region [38]) can also be used.

```
1 interface FailureDetector {
2   command Suspects * getSuspects();
3   command bool isSuspect(uint16_t p);
4   command uint16_t lastHeardFrom(uint16_t p);
5   command void removeSuspect(uint16_t p);
6   command uint8_t numSuspects();
7   event error_t noSuspects();
8   event error_t atLeastOneSuspect();
9   command void setLeaseDuration();
10 }
```

Fig. 1. The FailureDetector interface.

```
    Program LeaseFD_u(ld_u)
1:  upon Startup
2:      LeaseTimer.startPeriodic(ld_u * .9)
3:  upon (receive(ld_v) from v) do
4:      Record lease duration for v
5:      If v is a suspect, correct suspect list
6:  upon LeaseTimer.fired() do
7:      Send lease requests to each neighbor
8:  upon Suspect Query do
9:      Check if all neighbors have renewed their leases
10:     Add those that have not to the suspect list
11:     Return suspect list to application
```

Fig. 2. Lease-based failure detector algorithm.

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 Fig. 3. 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 *eventually perfect* ($\diamond\mathscr{P}$) failure detector [22] and sacrifice their claim to the resource, thereby allowing nodes a and b to make progress. In the case on the right side of Fig. 3, there is no failure. Node e has simply moved out of range 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.

In order to deal with this problem, we present our second failure detector. This one is an implementation of the $\diamond\mathscr{P}^m_\ell$ failure detector presented in [39]. An algorithm that this link failure detector implements is presented in Fig. 4. The MobilityAwareFD component implements this algorithm and this is executed at each node u, at a low rate.

4.1. Implementing $\diamond\mathscr{P}^m_\ell$ as MobilityAwareFD

This section presents a brief sketch of design of the $\diamond\mathscr{P}^m_\ell$ failure detector [39]. The algorithm is presented in Fig. 4. The basic idea of the $\diamond\mathscr{P}^m_\ell$ failure detector is that a local failure detector that uses message exchanges in a one-hop neighborhood alone cannot distinguish between a node that is failed in its neighborhood and one that has left its neighborhood. In order to make this distinction, the node uses the knowledge available in the rest of the network. Periodically, some node in the network initiates a gossip diffusing computation [40] in the communication graph. In this message, the initiator node u sends out the list of nodes that are in its local suspect list ($\mathscr{S}_u$). For each suspect v in the suspect list, u also sends the duration of time for which v has been suspected. This is shown in lines 2–7. 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.

Expanding phase. When a node w receives the gossip message for the first time, it examines the suspect group $\mathscr{SG}$ in the message, and performs two sets of actions to modify the suspect group. First, it checks to see if there is some process v that is a correct neighbor of w and yet is a member of $\mathscr{SG}$. In this case, w determines that v has been (wrongly) suspected by some other process, and *exonerates* v (lines 12–15). This is shown in Fig. 5a. In the second set of actions, the process w adds to $\mathscr{SG}$ all the processes that it

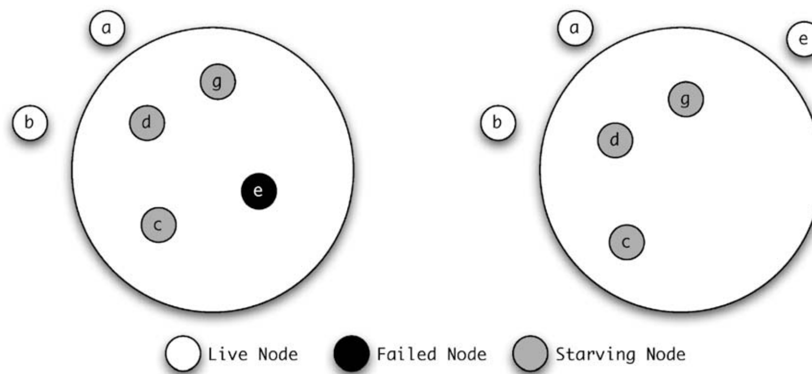


Fig. 3. Nodes unable to distinguish failure from mobility.

Program $SuspectSharing_u$
1: **initially** $state_u = \mathsf{idle} \wedge parent_u = u \wedge \mathcal{C}_u = \emptyset$

2: **if** $initiator$ **then** {* $initiator$ starts gossip *}
3: $state_u := \mathsf{active}$
4: $\mathcal{SG} := \{x : x \in \mathcal{S}_u : \langle x, ts_x, u, 0\rangle\}$ {* Prepare suspect group *}
5: $\mathcal{E} := \emptyset$
6: $\mathcal{C}_u := \mathcal{N}_u - \mathcal{S}_u$
7: send $\langle \mathcal{SG}, \mathcal{E}\rangle$ to all $w :: w \in \mathcal{N}_u$ {* Send gossip message to all neighbors *}

8: **upon** (receive $\langle \mathcal{SG}, \mathcal{E}\rangle$ from v) **and** ($state_u = \mathsf{idle}$) **do**
9: $parent_u := v$
10: $state_u := \mathsf{active}$
11: **for each** $\langle x, ts_x, \sigma, d_\sigma\rangle \in \mathcal{SG}$ **do**
12: **if** $(x \in \mathcal{N}_u) \wedge (x \notin \mathcal{S}_u)$ **then** {* Some other process suspects a live neighbor *}
13: **if** $(now - lhf_x < ts_x)$ **then** {* x is alive *}
14: $\mathcal{SG} := \mathcal{SG} - \langle x, ts_x, \sigma, d_\sigma\rangle$ {* Remove x from suspect group *}
15: $\mathcal{E} := \mathcal{E} \cup \{x\}$ {* Exonerate x *}
16: **if** $x \in \mathcal{S}_u$ **then** $\sigma := u; d_\sigma := 0$
17: **else if** $\sigma \notin \mathcal{N}_u$ **then** $d_\sigma := d_\sigma + 1$ {* Update distance from suspector *}
18: **for each** $\langle x, td_x\rangle \in \mathcal{S}_u$ **do**
19: **if** $x \notin \mathcal{SG} \wedge x \notin \mathcal{E}$ **then** $\mathcal{SG} := \mathcal{SG} \cup \langle x, ts_x, u, 0\rangle$ {* Add local suspects to suspect group *}
20: **if** $(\mathcal{N}_u - \{parent_u\}) \neq \emptyset$ **then**
21: $\mathcal{C}_u := \{w :: w \in \mathcal{N}_u \wedge w \notin \mathcal{S}_u \wedge w \neq parent_u\}$
22: send $\langle \mathcal{SG}, \mathcal{E}\rangle$ to all $w :: w \in \mathcal{C}_u \vee w \in \mathcal{S}_u$ {* Propagate gossip *}
23: **else** {* Leaf node, so respond to parent immediately *}
24: $state_u := \mathsf{complete}$
25: **if** $\neg initiator$ **then** send $\langle \mathcal{SG}, \mathcal{E}\rangle$ to $parent_u$

26: **upon** (receive $\langle \mathcal{SG}, \mathcal{E}\rangle$ from v) **and** ($state_u = \mathsf{active}$) **do**
27: **if** $v \in \mathcal{C}_u$ **then** $\mathcal{C}_u := \mathcal{C}_u - \{v\}$
28: **for each** $x \in \mathcal{E}$ **do**
29: **if** $x \in \mathcal{S}_u$ **then**
30: $\mathcal{S}_u := \mathcal{S}_u - \{x\}$ {* Remove exonerated nodes from suspect list*}
31: $\mathcal{N}_u := \mathcal{N}_u - \{x\}$ {* x is not a neighbor *}
32: **for each** $\langle x, ts_x, \sigma, d_\sigma\rangle \in \mathcal{SG}$ **do**
33: **if** $(x \in \mathcal{N}_u) \wedge (u \neq \sigma) \wedge (d_\sigma > 2)$ **then** {* x is suspected more than 2 hops away *}
34: $\mathcal{N}_u := \mathcal{N}_u - \{x\}$ {* x is not a neighbor *}
35: **if** $x \in \mathcal{S}_u$ **then** $\mathcal{S}_u := \mathcal{S}_u - \{x\}$ {* x is not a suspect *}
36: $\mathcal{C}_u = \{x :: x \in \mathcal{C}_u \wedge x \notin \mathcal{S}_u\}$ {* Update set of live neighbors yet to respond *}
37: **if** $\mathcal{C}_u = \emptyset \wedge \neg initiator$ **then** {* Heard back from all live neighbors; respond to parent *}
38: $state_u := \mathsf{complete}$
39: send $\langle \mathcal{SG}, \mathcal{E}\rangle$ to $parent_u$

Fig. 4. The $\Diamond\mathscr{P}_\ell^m$ algorithm that distinguishes node mobility from node failure.

currently suspects (the contents of $\mathscr{S}_w$) (lines 18–19). This is shown in Fig. 6. After the suspect group $\mathscr{SG}$ in the gossip message has been modified, *w* sends this updated message to each of its neighbors (except the node from whence it received the gossip message; this node is *w*'s parent). This phase is called the *expanding phase* of gossip. During this phase, each process updates the suspect group $\mathscr{SG}$ and the set of exonerated processes $\mathscr{E}$ based on local knowledge.

Shrinking phase. Once the gossip message has reached the edge of the graph (leaf nodes have received the message), the gossip algorithm switches to the *shrinking phase*, where messages are sent back to the initiator. Each leaf node, immediately upon updating $\mathscr{SG}$ and $\mathscr{E}$, sends the updated gossip message back to its parent (line 25). When a non-leaf node has received the gossip message back from each of its children, it examines the exonerated set $\mathscr{E}$ and if it finds some node υ in this $\mathscr{E}$ that is also in $\mathscr{S}_w$ and/or

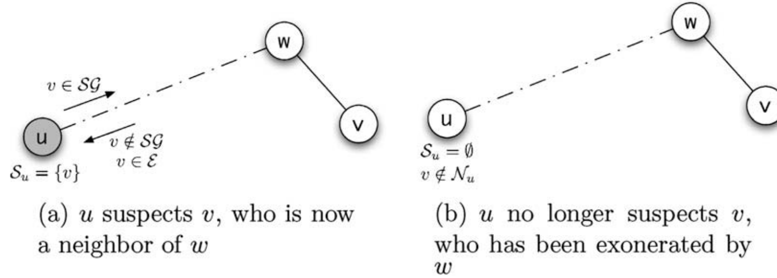


Fig. 5. Process υ is exonerated and *u* is restored to good state.

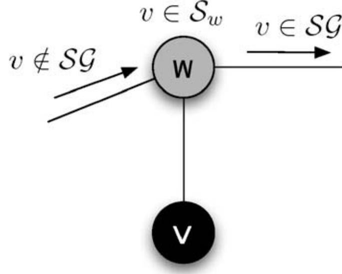


Fig. 6. w adds v, which it suspects, to $\mathcal{SG}$ before propagating $\mathcal{SG}$.

$\mathcal{N}_w$, w removes v from these sets (shown in Fig. 5b). The gossip message is then sent to the parent node.

It is important to note that if a process w sees one of its neighbors v in the exonerated set E, w removes v from its neighborhood ($\mathcal{N}_w$). This prevents w from suspecting v again in the future. This situation is shown in Fig. 7.

The algorithm presented here uses purely local knowledge along with a limited amount of knowledge that the rest of the network shares in order to make (correct) determinations of which nodes have actually failed and which nodes have simply moved out of communication range. The failure detector, while functioning like a *global failure detector* exhibits performance characteristics similar to a *local failure detector*. More details of this are presented in Section 5.

4.2. Proof of correctness

Here, we present a brief proof that the algorithm in the preceding section is indeed correct. The complete rigorous proof can be found in [39].

Claim 1. The mobility detection layer is guaranteed to terminate. The core of the mobility detection layer is a terminating diffusing computation. The algorithm does not add any new messages to this diffusing computation, and as such, does not modify its termination property.

Claim 2. No crashed node is removed from any suspect list incorrectly. There are only two places in suspect sharing algorithm presented in Fig. 4 where a process x is removed from the suspect set $\mathcal{S}_u$ (Line 30 and line 35). In the first case, the node x is removed because some other node in the network (downstream in the gossip tree) exonerated x. This means that x is no longer a neighbor, and is therefore removed from the neighborhood as well. In the second case, node x is indeed failed, but is being suspected by some other node that is further than two hops away. In this case as well, x is no longer a neighbor, and the removal from the suspect list is correct.

Claim 3. No running node is suspected by the mobility detection layer. The function of adding nodes to a suspect list is performed by the local failure detection layer, and not by the mobility detection layer. Therefore this claim is trivially true.

4.3. Case study example: resource allocation using dining philosophers

Let us now consider an example to see the utility of our failure detection components in a WSAS. Consider a network of nodes that use a dining philosophers scheme for distributed resource allocation. One popular algorithm for solving the dining philosophers problem is the *hygienic* algorithm proposed by Chandy and Misra [41]. This is a fork-based scheme; neighboring processes share *forks*, and the set of forks that a node p shares with its neighbors together represent a resource that the node wants to use. This means that if two neighbors share a fork, only one of them can be using the resource at any time. The nodes and the edges that represent forks comprise the *conflict graph*.

The hygienic solution to dining philosophers is based on maintaining a partial order of priority among processes. That is, the edges of the conflict graph are given directions such that the graph is acyclic. A fork held by a neighbor who has higher priority in the partial order is said to be *clean*, while one held by a lower priority neighbor is said to be *dirty*.

When two hungry processes compete for the same fork, the conflict is resolved in favor of the higher-priority process. There is no deadlock because of the acyclicity of the partial order (a "waits-for" cycle cannot form among processes).

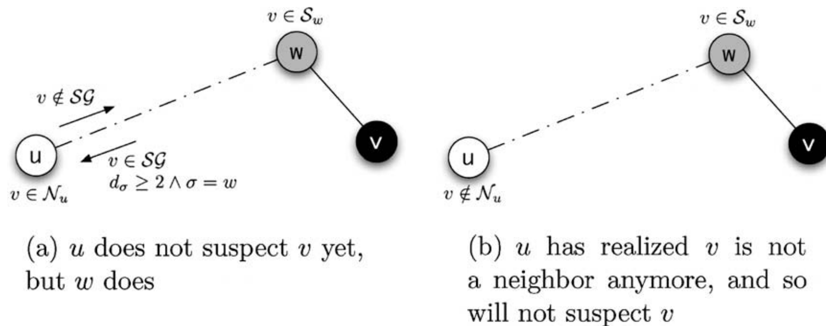


Fig. 7. Process u discovers that although v is crashed, v is no longer a neighbor, and hence u can stay in good state.

```
    Program HygienicDining_u
 1: upon becoming hungry
 2:     if hold all forks, begin eating
 3:     else request all missing forks
 4: upon receiving fork
 5:     flip state of fork (clean/dirty)
 6:     if hold all forks, begin eating
 7: upon receiving fork request
 8:     if fork is dirty and not eating
 9:         send fork
10:         if hungry, re-request fork
11:     else defer request
12: upon finishing eating
13:     all forks become dirty
14:     process deferred fork requests
```

Fig. 8. Hygienic dining philosopher algorithm (adapted from [42]).

There are two key parts to the hygienic solution. The first is that a higher priority hungry process never yields to a lower-priority neighbor (i.e., a hungry process never relinquishes a clean fork). The second is that after a process eats, it lowers its priority below that of all its neighbors. This operation preserves the acyclicity of the conflict graph. Together, these properties are sufficient to ensure that the liveness specification is met. For a complete proof of correctness of this algorithm, please refer to [41]. Fig. 8 shows the pseudocode for the hygienic algorithm.

While this algorithm is extremely simple to understand and implement, and performs well with respect to message complexity and response time, its *failure locality*[3] is poor. In fact, it is as bad as it can get! The failure locality of the hygienic algorithm is d, the diameter of the conflict graph. This means that a *single* node that fails while holding its forks will bring the *entire* network to a halt. As an illustration, consider Fig. 9. In Fig. 9a, node a has failed while holding the fork that it shares with node b. Nodes c and d are higher priority than b, which is hungry. While c and d will not yield immediately when b requests their forks, eventually they will eat and will reduce their priorities below that of b. Therefore, b will collect and hold on to those forks. However, it will never get the fork that a is holding, since a is failed. Since b is still hungry, and is higher in priority than c and d, b will not yield to c or d. Meanwhile, e becomes hungry and is waiting on d's fork. Thus, a single failure has caused the entire network to halt, as shown in Fig. 9(b).

In [42], Pike and Sivilotti propose a transformation of the hygienic algorithm to reduce the failure locality to 1 (Fig. 10). The strategy is as follows. When a node p in the network suspects that one of its neighbors (say, q) is failed, it enters a special state in which it honors all fork requests, regardless of the priority of the requesting node. In this manner, p shields the rest of the network from q's failure; only q's immediate neighbors in the conflict graph are affected by this failure.

If the network topology is static, we could implement this transformation using the LeaseFD component. The failure detector can keep track of each node's neighborhood, and allow the dining algorithm to make progress. When a node q fails, it will fail to renew its lease with its neighbor p. When p checks to see if all of its neighbors have renewed their leases, it will see that q has not, and will suspect q. Consequently, it will set its state to be *skeptical*. A node in the *skeptical* state will always yield forks to other neighbors (Fig. 10). Therefore, the failure locality is reduced to 1.

However, consider that the network's topology is dynamic, and nodes in the network may move around. In this case, the LeaseFD failure detector cannot distinguish between a node that is failed and one that has moved away from range. Consider Fig. 11a for example. In this picture, solid lines denote active links, while dotted lines represent that the two nodes were neighbors at some point in the past. The node a is a mobile node, and it moved around in the network. After a while, nodes b, c, and d are all starving, thinking that a is one of their failed neighbors (see Fig. 12).

Now consider the same transformation (Fig. 10), this time using the MobilityAwareFD failure detector component. This component can distinguish node failure from node mobility. Nodes b, c, and d will suspect a to be failed, but eventually (after the suspect group has been gossiped around), they return back to normal state; node e exonerates a.

5. Evaluation of performance overhead

The experimental evaluation of our failure detection middleware was conducted on the *NESTbed* Tmote Sky testbed at Clemson University [43]. The testbed has 80 Tmote Sky nodes arranged in a 16×5 grid. Even though the physical separation between the motes is only about one foot, the radios on the motes are set to transmit at a low enough power level so as to form multi-hop network topologies.

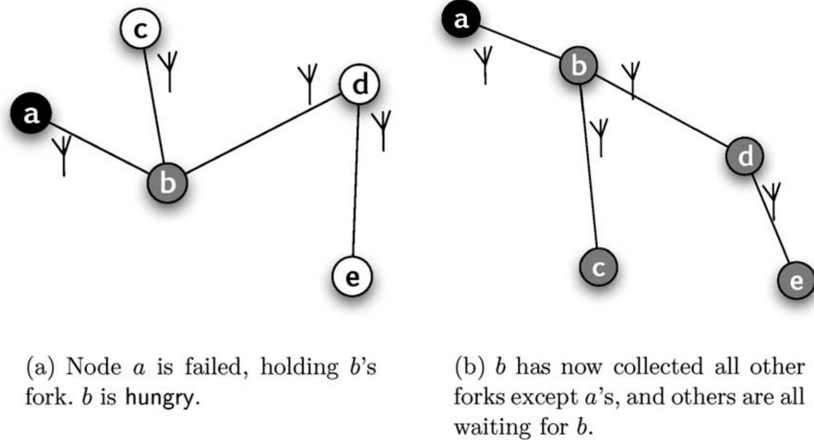
We ran several experiments on the testbed to measure the quality of service provided by our failure detection components. We measure these metrics for both the components presented here – LeaseFD (static topologies) and MobilityAwareFD (dynamic topologies). All of the experiments ran on the entire testbed. Each run of the experiment lasted 10 min. Each experiment was run multiple times, and the results were averaged to account for anomalies.

5.1. Message complexity

The first set of experiments here measure the message complexity of our two failure detection components. The message overhead of the $\diamond\mathcal{P}_\ell^m$ failure detector is the sum of the overhead introduced by the local failure detector, and the overhead of gossip. Consider a WSAS with n nodes, e links between nodes, and maximum number of neighbors of a node δ. As is common with typical WSAN applications, $\delta \ll n$.

In the lease-based local failure detector, each node periodically sends out one lease message to each of its neighbors. If we assume that the lease durations of all nodes

[3] A measure of the impact of a fault in a single node on the rest of a distributed system.



(a) Node a is failed, holding b's fork. b is hungry.

(b) b has now collected all other forks except a's, and others are all waiting for b.

Fig. 9. Hygienic dining philosophers; failure locality is diameter of graph (d).

```
    Program TransformedHygienicDining_u
 1: upon becoming hungry
 2:        if hold all forks, begin eating
 3:        else request all missing forks
 4: upon receiving fork
 5:        flip state of fork (clean/dirty)
 6:        if hold all forks, begin eating
 7: upon receiving fork request
 8:        if fork is dirty and not eating
 9:               send fork
10:               if hungry, re-request fork
11:        else if skeptical and not eating
12:               send fork
13:        else defer request
14: upon becoming skeptical
15:        if not eating
16:               process deferred fork requests
17: upon stopping being skeptical
18:        if hungry
19:               if hold all forks, begin eating
20:               else request all missing forks
21: upon finishing eating
22:        all forks become dirty
23:        process deferred fork requests
```

Fig. 10. Hygienic dining philosopher algorithm, transformed to tolerate crashes (adapted from [42]).

are roughly the same, then the total number of messages sent out in the entire network is $O(n \cdot \delta)$ for each round of leases.

The message complexity of gossip is $O(n \cdot e)$. Each node participates in propagating the message out to the edge of the graph, and then propagating the message back to the initiator in the shrinking phase.

Supposing that there were l rounds of lease messages exchanged among neighbors for each round of gossip in $\diamond\mathscr{P}_\ell^m$, the message overhead in the WSAS is $O(n \cdot e + l \cdot n \cdot \delta)$. The additional overhead introduced by gossip is much smaller than the overhead introduced by the local failure detector alone.

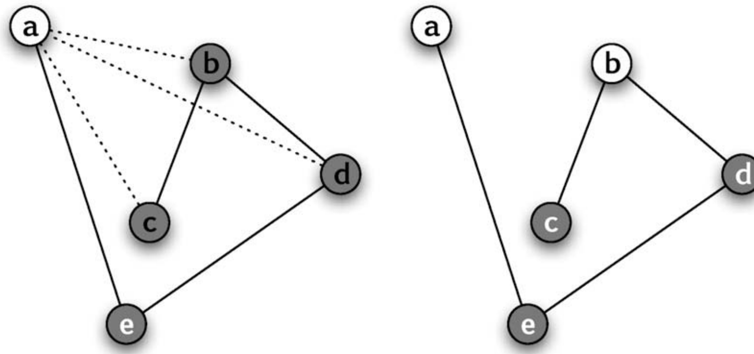
For each component, we measured both the overall load on the entire network (total number of additional messages sent by all nodes in the network), as well as the average number of additional messages sent by each node. The average number of neighbors (average degree) for each node in the network is 6.7. Given this, the measurement of average number of extra message traffic introduced by the LeaseFD component is consistent with the analytical prediction of $O(n \cdot \delta)$ for each round of lease messages exchanged. Fig. 13a shows the average number of messages sent out by LeaseFD per round of leases at each node in the network, for various values of lease durations.

The second part of the message complexity experiments measure the additional message overhead introduced by the MobilityAwareFD component, when running in conjunction with the LeaseFD local failure detector. Again, we measure this for different values of gossip durations (rate at which the suspect sharing algorithm is initiated). Fig. 13b shows the average message overhead introduced by the MobilityAwareFD component.

5.1.1. Speed of failure detection

The speed of failure detection depends on the speed of the local failure detection module (lease in our case). If a node p does not renew its lease with a neighbor q, then q will suspect p when it checks to see if p renewed its lease. In the worst case, p fails the instant after it sends out a lease renewal request. q will detect this failure once the p's lease expires. So if the average lease duration is ld, and the rate at which each process checks on its neighbors is sd, then the speed of failure detection is $O(ld + sd)$.

Our second set of experiments measure how quickly a failure is detected. We measured this on both of our components, LeaseFD and MobilityAwareFD. Since the speed of failure detection is a function of the local failure detector alone, in this case LeaseFD, we only evaluate this component. Fig. 14a shows the speed of failure detection for the LeaseFD component for varying values of lease duration (ld) and suspect duration (sd). Fig. 14(b) confirms that gossip duration has no effect on the speed of failure detection.



(a) Node a is is mobile, and has made b, c, and d think that it is failed. These nodes are now starving.

(b) e exonerates a, and allows b, c, and d to return to normal functioning.

Fig. 11. The MobilityAwareFD component reduces the failure locality of hygienic dining philosophers to 1 in networks that allow node mobility.

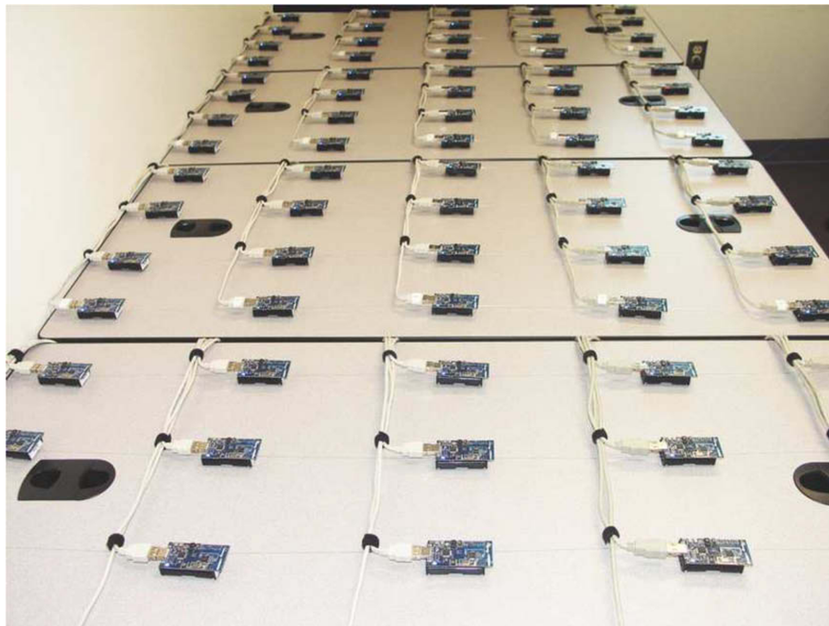


Fig. 12. Experimental setup: The *Tmote Sky* testbed at Clemson University.

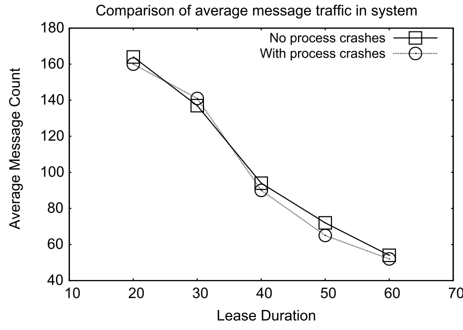
5.1.2. Mistake duration

The $\diamondsuit\mathcal{P}_\ell^m$ detector at process p learns of mistakes it may have made when it sees that some process q that is suspected has been exonerated by someone else in the network. For such exoneration to happen, a round of gossip needs to be executed. At the end of each round of gossip, any mistakes that any node made will be corrected. Therefore, the mistake duration of $\diamondsuit\mathcal{P}_\ell^m$ is $O(n \cdot e \cdot \Delta_m)$ if Δ_m is the average message transmission delay between two nodes.

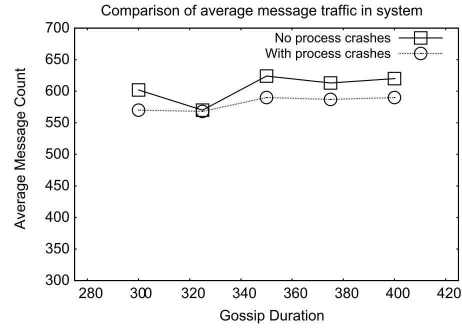
The next metric we measure by experimentation is the mistake duration of our components – the time taken by the failure detector to correct a mistake. This is a function of how often the suspect sharing algorithm executes (gossip duration, ρ_g and how long it takes for a round of the gossip to terminate. Fig. 15b shows the mistake duration of the MobilityAwareFD component for different values of gossip duration. The local failure detector does not have an impact on this metric. Fig. 15a confirms this: the results do not vary much with different values of lease duration or suspect duration.

5.1.3. Mistake recurrence time

The mistake recurrence time of a failure detector measures the time between two consecutive mistakes that

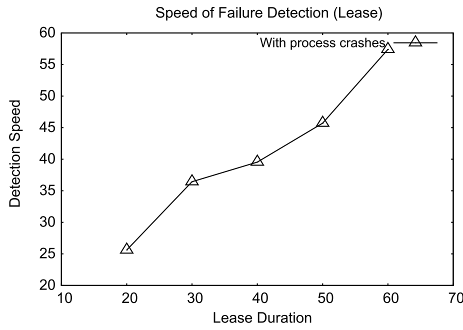


(a) Average number of messages sent out per node with different average lease durations.

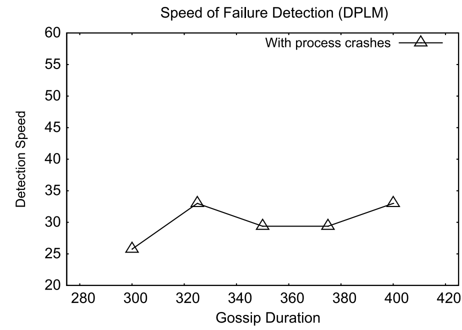


(b) Average number of messages sent out per node with different average gossip durations. (Lease duration is fixed at 30 seconds, and suspect duration is 29 seconds).

Fig. 13. Message complexity of LeaseFD and MobilityAwareFD components.

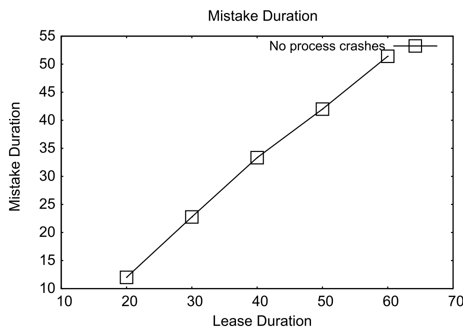


(a) Speed of detection for different average lease durations

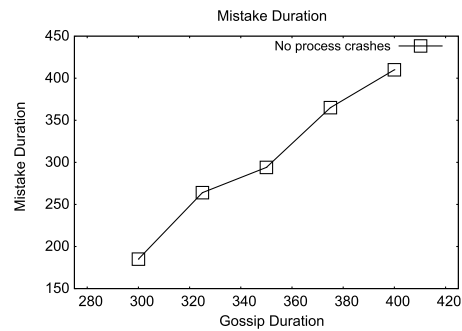


(b) Speed of detection for different average gossip durations

Fig. 14. Speed of detection.



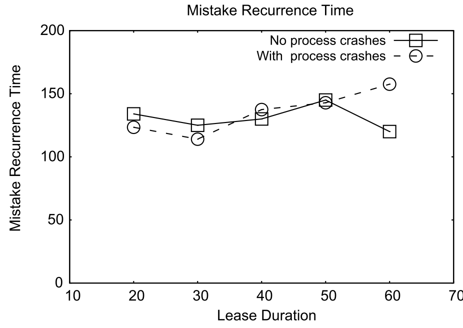
(a) Mistake duration of $\diamond\mathcal{P}_\ell^m$ for different average lease durations.



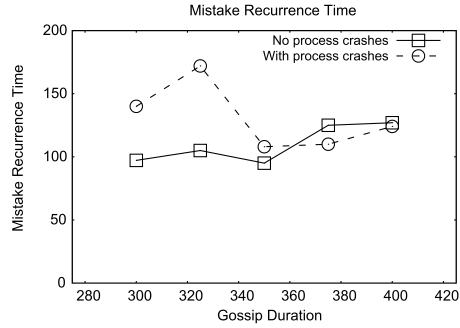
(b) Mistake duration of $\diamond\mathcal{P}_\ell^m$ for different average gossip durations.

Fig. 15. Mistake duration of $\diamond\mathscr{P}_\ell^m$ implementation. This measures how quickly a wrongly-suspected node is exonerated, and the suspecter is returned to good state.

the detector makes. This metric is a function only of the local failure detector, in the case of the LeaseFD detector, depends on the relationship between the suspect duration and lease duration. Fig. 16a shows the mistake recurrence time for different values of lease duration. Fig. 16b shows the mistake recurrence times for different values of gossip

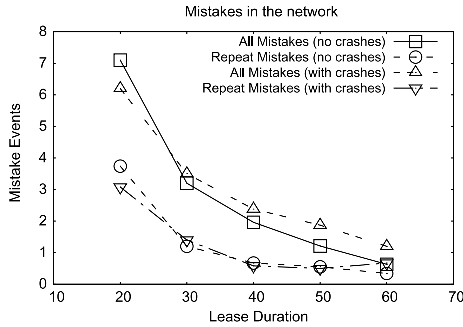


(a) Mistake recurrence time for different average lease durations

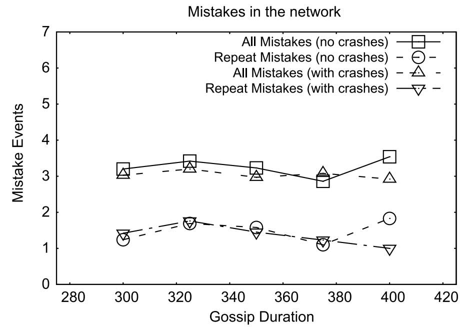


(b) Mistake recurrence time for different average gossip durations

Fig. 16. Mistake recurrence time.



(a) Events detected by the LeaseFD detector



(b) Events detected by the MobilityAwareFD detector

Fig. 17. Events detected in the network.

Application	**RAM**	**ROM**
Hygienic dining, no failure detection	2168	13716
Hygienic dining, with LeaseFD	2390 (10%)	16042 (17%)
Hygienic dining, with MobilityAwareFD	2682 (23%)	20564 (49%)

Fig. 18. Memory footprint for telosb node running $\Diamond\mathcal{P}_\ell^m$.

duration, and confirms that the suspect sharing part has no essential effect on this metric.

5.1.4. Mistake rate

Next, we measured the actual rate of mistakes in the system. We also measured how often a single mistake is repeated by a given node. Fig. 17a shows the comparison of events detected as a function of lease duration in networks with node crashes, and without. It also shows the number of repeat mistakes in these different scenarios. Notice how the rate of mistakes decreases with increasing lease duration. This metric is also a function only of the local failure detector, and the gossip algorithm does not have an effect. Fig. 17b confirms this.

5.1.5. Memory overhead

Finally, we measured the memory overhead that is introduced by the failure detection components we have presented in this paper. We measured this in the context of the case study example of hygienic dining philosophers with and without failure detection. Fig. 18 shows the comparison in RAM and ROM usage for the hygienic dining implementation in TinyOS with and without the failure detection components.

The memory overhead introduced by the failure detection components is not dependent on the underlying application. The increase in ROM is caused by the additional code introduced. The increase in RAM is in account of the state that the failure detection components have to

maintain in order to function. This state size is determined by the size of the local neighborhood of nodes in the network. This size is currently set statically, based on design parameters. An approach recently presented in [44] can be used to optimize the size of neighborhood sets based on network observation at run-time. Such optimization can control the memory overhead to what is actually necessary for the given configuration.

6. Conclusions

In this paper, we presented two failure detectors for WSAS. Since individual nodes in a sensor network deployment are frequently of very low cost, and are hence dispensable, failure considerations are elevated to a first-class level in the design of software for such systems. As a consequence, the ability to detect when failures occur, and react to such failures in the most reasonable manner is important. The components presented in this paper implement efficient solutions to detecting failures.

The failure detectors can be considered as a middleware service and include a way of distinguishing between node failures and link failures. We use this quality of the service in tolerating mobility of nodes in the network, and yet providing consistent *failure localization* when faults do occur. We have presented implementations of our middleware service components in a readily usable form – the components themselves are completely self-contained, and provide an easy-to-use interface that applications can wire to.

Finally, as an aid to developers wishing to use these components, we have provided experimental measurements of the overheads and costs associated with using the middleware service. These measurements will allow developers to be able to make predictions of performance of their own applications while used in conjunction with the failure detection middleware.

References

[1] D. Doolin, N. Sitar, Wireless sensors for wildfire monitoring, in: Proceedings of the SPIE Symposium on Smart Structures and Materials/NDE 2005, SPIE Press, 2005, pp. 477–484.

[2] S. Glaser, Some real-world applications of wireless sensor nodes, in: SPIE Symposium on Smart Structures & Materials/NDE 2004, 2004, pp. 344–355.

[3] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, J. Anderson, Wireless sensor networks for habitat monitoring, in: WSNA'02: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications, ACM Press, New York, NY, USA, 2002, pp. 88–97.

[4] G. Simon, M. Maróti, Á. Lédeczi, G. Balogh, B. Kusy, A. Nádas, G. Pap, J. Sallai, K. Frampton, Sensor network-based countersniper system, in: The Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04), 2004, pp. 1–12.

[5] S. Yang, Redwoods go high tech: researchers use wireless sensors to study california's state tree, 2003. <www.berkeley.edu/news/media/releases/2003/07/28_redwood.shtml>.

[6] N. Hayslip, S. Sastry, J. Gerhardt, Networked embedded automation, Assembly Automation 26 (3) (2006) 235–241.

[7] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, M. Gouda, Y. ri Choi, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, K. Parker, Exscal: elements of an extreme scale wireless sensor network, in: RTCSA'05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 102–108.

[8] G. Zhou, T. He, S. Krishnamurthy, J. Stankovic, Impact of radio irregularity on wireless sensor networks, in: The Second International Conference on Mobile Systems Applications and Services, ACM Press, 2004, pp. 125–138.

[9] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, D. Estrin, Sympathy for the sensor network debugger, in: SenSys'05: Proceedings of the Third International Conference on Embedded Networked Sensor Systems, ACM Press, New York, NY, USA, 2005, pp. 255–267.

[10] A. Woo, T. Tong, D. Culler, Taming the underlying challenges of reliable multihop routing in sensor networks, in: SenSys'03: Proceedings of the First International Conference on Embedded Networked Sensor Systems, ACM Press, New York, NY, USA, 2003, pp. 14–27.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, in: ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 2000, pp. 93–104.

[12] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, D. Culler, Design of a wireless sensor network platform for detecting rare, random, and ephemeral events, in: IPSN'05: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IEEE Press, Piscataway, NJ, USA, 2005, p. 70.

[13] J. Polastre, R. Szewczyk, D. Culler, Telos: enabling ultra-low power wireless research, in: IPSN'05: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, IEEE Press, Piscataway, NJ, USA, 2005, p. 48.

[14] J. Hill, M. Horton, R. Kling, L. Krishnamurthy, The platforms enabling wireless sensor networks, Communications of the ACM 47 (6) (2004) 41–46.

[15] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, D. Culler, The nesc language: a holistic approach to networked embedded systems, in: PLDI'03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, ACM Press, New York, NY, USA, 2003, pp. 1–11.

[16] P. Levis, D. Culler, Maté: a tiny virtual machine for sensor networks, in: ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM Press, New York, NY, USA, 2002, pp. 85–95.

[17] C.-C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: MobiSys'05: Proceedings of the Third International Conference on Mobile Systems, Applications, and Services, ACM Press, New York, NY, USA, 2005, pp. 163–176.

[18] A. Dunkels, B. Gronvall, T. Voigt, Contiki – a lightweight and flexible operating system for tiny networked sensors, in: LCN'04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 455–462.

[19] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms, Mobile Networks and Applications 10 (4) (2005) 563–579.

[20] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E.A. Brewer, D.E. Culler, The emergence of networking abstractions and techniques in tinyos, in: NSDI, 2004, pp. 1–14.

[21] A. Arora, M. Gouda, W. Leal, J.O. Hallstrom, T. Herman, N. Sridhar, A state-based language for sensor-actuator networks, in: Proceedings of WWSNA 2007, 2007.

[22] T.D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of ACM 43 (2) (1996) 225–267.

[23] M.K. Aguilera, W. Chen, S. Toueg, Heartbeat: a timeout-free failure detector for quiescent reliable communication, in: WDAG'97: Proceedings of the 11th International Workshop on Dist. Alg., Springer-Verlag, London, UK, 1997, pp. 126–140.

[24] I. Gupta, T.D. Chandra, G.S. Goldszmidt, On scalable and efficient distributed failure detectors, in: PODC'01: Proceedings of the 20th ACM symposium on Principles of Distributed Computing, ACM Press, New York, NY, USA, 2001, pp. 170–179.

[25] C. Almeida, P. Veríssimo, Timing failure detection and real-time group communication in real-time systems, in: Proceedings of the Eighth Euromicro Workshop on Real-time Systems, 1996, pp. 230–235.

[26] R.V. Renesse, Y. Minsky, M. Hayden, A gossip-style failure detection service, Tech. rep., Cornell University, Ithaca, NY, USA, 1998.

[27] N. Hayashibara, A. Cherif, T. Katayama, Failure detectors for large-scale distributed systems, in: SRDS'02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02), IEEE Computer Society, Washington, DC, USA, 2002, p. 404.
[28] C. Fetzer, U. Schmid, M. Susskraut, On the possibility of consensus in asynchronous systems with finite average response times, in: 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), 2005, pp. 271–280.
[29] R. Boichat, P. Dutta, R. Guerraoui, Asynchronous leasing, in: Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02), 2002, pp. 180–187.
[30] M. Hutle, J. Widder, Time free self-stabilizing local failure detection, Research Report 33/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria, 2004.
[31] C. Fetzer, K. Högstedt, Rejuvenation and failure detection in partitionable systems, in: PRDC'01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing, IEEE Computer Society, Washington, DC, USA, 2001, p. 154.
[32] N. Kolettis, N.D. Fulton, Software rejuvenation: analysis module and applications, in: FTCS'95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, IEEE Computer Society, Washington, DC, USA, 1995, p. 381.
[33] M.K. Aguilera, W. Chen, S. Toueg, Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks, Theoretical Computer Science 220 (1) (1999) 3–30.
[34] K. Whitehouse, C. Sharp, E. Brewer, D. Culler, Hood: a neighborhood abstraction for sensor networks, in: MobiSys'04: Proceedings of the Second International Conference on Mobile Systems, Applications, and Services, ACM Press, New York, NY, USA, 2004, pp. 99–110.
[35] S. Rost, H. Balakrishnan, Memento: a health monitoring system for wireless sensor networks, in: IEEE SECON, Reston, VA, 2006, pp. 575–584.
[36] M. Elhadef, A. Boukerche, H. Elkadiki, Performance analysis of a distributed comparison-based self-diagnosis protocol for wireless ad-hoc networks, in: MSWiM'06: Proceedings of the Ninth ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems, ACM Press, New York, NY, USA, 2006, pp. 165–172.
[37] M. Elhadef, A. Boukerche, H. Elkadiki, A distributed fault identification protocol for wireless and mobile ad hoc networks, Journal of Parallel and Distributed Computing 68 (3) (2008) 321–335.
[38] M. Welsh, G. Mainland, Programming sensor networks using abstract regions, in: NSDI, 2004, pp. 29–42.
[39] N. Sridhar, Decentralized local failure detection in dynamic distributed systems, in: SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 143–154.
[40] E.W. Dijkstra, C.S. Scholten, Termination detection for diffusing computations, Information Processing Letters 11 (1) (1980) 1–4.
[41] K.M. Chandy, J. Misra, The drinking philosophers problem, ACM Transactions on Programming Languages and Systems 6 (4) (1984) 632–646.
[42] S.M. Pike, P.A. Sivilotti, Dining philosophers with crash locality 1, in: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS), IEEE, 2004, pp. 22–29.
[43] A.R. Dalton, J.O. Hallstrom, An interactive, server-centric testbed for wireless sensor systems, International Journal of Distributed Sensor Networks, in revision.
[44] S.K. Wahba, S. Dandamudi, A.R. Dalton, J.O. Hallstrom, Neptune: Optimizing sensor networks, in: The Proceedings of The 17th International Conference on Computer Communications and Networks (IC3N'08), IEEE Computer Society, Washington, DC, USA, 2008.

Post-print prepared by MSL Academic Endeavors, the imprint of the Michael Schwartz Library at Cleveland State University (2014)