On Consistent Neighborhood Views in Wireless Sensor Networks

Arshad Jhumka
University of Warwick, Coventry, UK
arshad@dcs.warwick.ac.uk

Luca Mottola
Swedish Institute of Computer Science, Sweden
luca@sics.se

***Abstract*—Wireless sensor networks (WSNs) are characterized by *localized interactions*. Indeed, several WSN algorithms and protocols work in a decentralized fashion by coordinating nodes within the wireless communication range, e.g., localization algorithms and MAC protocols. Nevertheless, most often these mechanisms do not address faults that may affect the way wireless neighborhoods are recognized by nodes, e.g., as in the case of data corruption. As the operation of these mechanisms is rooted in the use of topology information, these faults may be a significant detriment to correct and efficient system operation.**

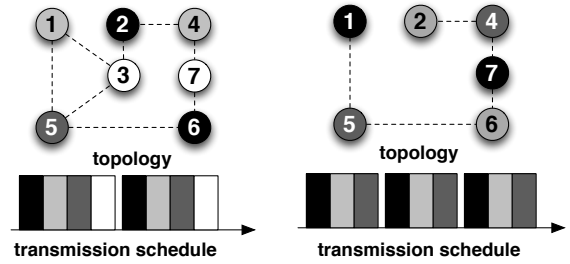
In this paper, we argue that the above issues are particular instances of a general problem of *consistent neighborhood view*. We present three increasingly weaker specifications of the problem. Next, we prove the impossibility of solving the two stronger specifications, and provide an algorithm to solve the weakest specification. In addition, we implement our algorithm in a commonly used WSN network stack, and assess its performance both in simulation and in a real-world testbed. The results show that, when possible, our mechanisms efficiently solve the problem of consistent neighborhood view, providing higher-level mechanisms with a re-usable building block to leverage off.

I. Introduction

Wireless Sensor Networks (WSNs) are distributed systems of battery-powered, resource-constrained computing devices equipped with a wireless communication interface. Characteristics such as ease of deployment have made them a viable solution to sense data in diverse contexts [1], [2].

To achieve energy efficiency, *localized interactions* [3] are usually preferred in WSNs, where algorithms and protocols typically operate by coordinating devices within the 1-hop wireless communication range. Several existing solutions are designed in this vein, e.g., localization algorithms and TDMA (time-division multiple-access) MAC protocols [4]. Figure 1 illustrates an example of the latter functionality. To identify the transmission schedules, TDMA protocols coordinate devices so that no two nodes simultaneously transmit towards the same receiver, a situation causing message losses due to hidden terminal problems [4]. In addition, to reduce message latencies, TDMA MAC protocols aim to minimize the total number of transmission slots used.

In WSNs faults arise in a number of ways, e.g., as *data corruption* due to defective hardware [5], [2] or erroneous sensor readings [6], as *link failures* caused by fluctuating topologies [7], or as *node crashes* due to exhausted batteries or environmental factors [8]. Algorithms and protocols must deal with these situations to maintain good performance. For instance, if node 3 in Figure 1(a) fails, the protocol must recognize the topology change and reconfigure the schedules to further minimize the number of slots used, for instance, as shown in Figure 1(b).



(a) Before failure: 4 transmission slots required.

(b) After failure: 3 transmission slots required.

Fig. 1. TDMA example with node failures. (Dashed lines represent bi-directional communication links).

For an efficient and correct operation of WSN algorithms and protocols in situations like Figure 1, the *physical* network topology must be accurately reflected in the nodes' *logical* states. Particularly, nodes are required to quickly and correctly identify a "consistent" view on their 1-hop neighbors. Consistency here intuitively indicates that, given any two devices n and m, all nodes reachable from both n and m in 1 hop must always appear in both n's and m's logical state, i.e., any 2-hop neighbors must agree on their shared neighborhood. For instance, node 3 in Figure 1(a) must appear in node 1, node 2, and node 5 neighborhoods, whereas it must correctly disappear from every neighborhood in Figure 1(b). If so, the TDMA protocol can correctly reconfigure the communication schedules. Otherwise, inaccurate topology information may yield an incorrect assignment of transmission slots, producing collisions at the physical layer and message losses.

Besides TDMA MAC protocols, which are widely investigated in WSNs [4], similar situations arise in a number of staple WSN functionality. For instance, when deciding on the placement of data operators using distributed heuristics [9], inaccurate topology information may lead to inefficient assignments of operators to nodes. In localization protocols, inconsistent neighborhood views have been shown to cause severe inaccuracies [10]. Finally, topology control mechanisms [11] may fail if nodes are not correctly aware of the underlying physical topology, to the point of creating network partitions that may be impossible to repair.

Nevertheless, this issue is often overlooked. Periodic bea-

cons are normally used to signal the presence of a neighboring node. This technique alone, however, cannot defend against data corruption affecting the neighborhood views and induces long delays before the nodes acquire the correct topology information. To address this issue, this paper investigates the problem of *consistent neighborhood view* from a *theoretical* and a *system* perspective:

- we formally study the problem to determine the conditions under which it can be solved and, if so, we identify suitable algorithms. To achieve this, in Section II we define system and fault models to provide a foundation to develop and analyze algorithms rigorously. Next, in Section III we present three increasingly weaker specifications of the problem at stake. Our investigation leads to an impossibility result for the two stronger problem formulations, whose formal proofs are described in Section IV. On the other hand, in Section V we present a distributed algorithm to solve the weakest problem specification, and prove its correctness.
- we implement our algorithm as a building-block in Rime [12], the network stack employed in the Contiki [13] operating system for WSNs. As described in Section VI, the integration in Rime allows higher-level algorithms and protocols to treat consistent neighborhood views as an operating system service. We assess the effectiveness of our implementation in simulation and in a real-world testbed, as reported in Section VII. Our results indicate that the overhead imposed by our implementation is very limited and, *when possible*, consistency of neighborhood views is rapidly and efficiently achieved with minimum disruption for upper-level algorithms and protocols.

We end the paper with brief concluding remarks and directions for future work in Section VIII.

To the best of our knowledge, we are the first to carry out a similar study in the context of WSNs, where faults and topology discovery have been mainly investigated from a system perspective [14] with little or no formal treatment. Moreover, with a few exceptions [15], [16], the issues stemming from data corruption are seldom taken into consideration.

There exists, however, related work in the area of fault tolerance in traditional distributed systems. For instance, the problem we investigate is reminiscent of seminal work [17] where, however, the authors considers a Byzantine environment, which is instead a failure model we do not study. Self-stabilizing topology discovery and link coloring are also related, and several work exist in this area, such as [18], [19]. However, our work involves maintaining a consistent neighborhood view among 1-hop nodes, whereas topology discovery normally involves the entire network. Group membership and communication have been studied in both asynchronous distributed systems [20] and peer-to-peer systems [21]. However, the hardware limitations of typical WSN devices make these results hardly applicable in our context.

II. Model

This section serves as a stepping stone by describing a model of WSNs and defining some concepts used next.

Topology. We define a WSN node as a computing device equipped with a wireless interface and associated to a unique identifier. Communication in wireless networks is typically modeled with a circular communication range centered on the node, and assuming all nodes have the same communication range. With this model, a node is thought as able to exchange data with all devices within its communication range.

In reality, however, communication between two WSN devices may be temporarily disrupted by a number of factors, e.g., co-location with other wireless technologies and environmental noise [7]. Thus, in this work we define the *physical neighborhood* of node n as the set of devices that node n can exchange data with at a given point in time, and denote this set by N. Based on this notion, we model the *physical topology* of a WSN as $S = (\Gamma, \Sigma)$, where Γ is a set of node ids, and Σ is a set of (node id, physical neighborhood) pairs, i.e., $\forall n \in \Gamma : (n, N) \in \Sigma$. Notice that in this formulation the physical neighborhoods may *change* over time.

Information on the devices a node can exchange data with are stored in the node *logical* state and, in a sense, represent how algorithms and protocols perceive the underlying physical topology. We refer to this notion as the *logical neighborhood* of a node n, denoted by N^l. Similarly as above, given a physical topology $S = (\Gamma, \Sigma)$ we denote by $S^l = (\Gamma^l, \Sigma^l)$ the corresponding *logical topology*, such that $\forall n \in \Gamma^l : (n, N^l) \in \Sigma^l$, and $\Gamma^l \subseteq \Gamma$.

Ideally, Γ^l and Σ^l should always be the same as Γ and Σ, respectively. However, Γ and Σ evolve over time because of faults such as data corruption, node crashes, and link failures. Thus, Γ^l, Σ^l may be different from Γ, Σ. We call physical (logical) neighborhood *view* a snapshot of the physical (logical) neighborhoods at a given point in time.

Programs. We model the processing on a WSN node as a *process*[1] containing non-empty sets of *variables* and *actions*. A *program* is a non-empty set of processes.

Variables take values from a fixed domain. We denote a variable v of process n by $n.v$. Each process has a special *channel* variable, denoted by *ch*, modeling a FIFO queue of incoming data sent by other nodes. This variable is defined over the set of (possibly infinite) message sequences. Every variable of every process, including the channel variable, has a set of *initial* values. The *state* of a program is an assignment to variables of values from their respective domains. The set of *initial states* is the set of all possible assignments of initial values to variables of the program. A state is called *initial* if it is in the set of initial states. The *state space* of the program is the set of all possible value assignments to variables.

An action has the form $\langle name \rangle :: \langle guard \rangle \rightarrow \langle command \rangle$. In general, a *guard* is a state predicate defined over the set of variables. When *guard* evaluates to true, the *command* can be executed, which takes the program from one

[1] We use the term node and process interchangeably.

state to another. A *command* is a sequence of assignment and branching statements. Also, a guard or command can contain universal or existential quantifiers of the form: $(\langle quantifier \rangle \langle boundvariables \rangle : \langle range \rangle : \langle term \rangle)$, where *range, term* are boolean constructs. When a guard evaluates to true in a certain program state, then the corresponding action is *enabled* in that state.

A special **timeout**($timer$) guard evaluates to true when a *timer* variable reaches zero, i.e., *timer* expires. A **set**($timer, value$) command can be used to initialize the timer variable to a specified value.

Semantics. We use guarded commands with interleaving semantics for simplicity. Among program actions that are enabled, one is chosen and its command is executed atomically. A *computation* of a program is a sequence $s_0 \cdot s_1 \cdot s_2 \ldots s_i \cdot s_{i+1} \ldots$ of states of the program where state s_{i+1} is reached from s_i by executing an action that is enabled in s_i, i.e., execution of actions is atomic. A tuple (s_i, s_{i+1}) is called a *transition*. We assume the computation model to be *synchronous*, in the sense that there is a known upper bound on the time it takes processes to take a step. This assumption is not unreasonable as WSNs are often time-synchronized [22] to either correlate sensor readings from different devices or for time-based protocols such as TDMA [4].

A *specification* is a set of computations. Program P satisfies specification ¶ if every computation of P is in ¶. Alpern and Schneider [23] stated that every specification can be described as the conjunction of a safety and liveness property. Intuitively, safety states that something bad should not happen, whereas liveness states that something good will eventually happen. Formally, the safety specification identifies a set of finite computation prefixes that should not appear in any computation. A liveness specification identifies a set of computation suffixes such that every computation has a suffix in this set.

Communication. An action with a **rcv**(msg,$sender$) guard is enabled when there is a message at the head of the channel variable ch of that process. Executing the corresponding action causes the message to be dequeued from the process' channel, while msg and $sender$ are bound to the content of the message and the identifier of the sender node.

Differently, the **send**(msg,$dest$) command causes message msg to be attached to the tail of the channel variable ch of processes in the $dest$ set. The semantics of **send** executed on node n differs depending on the processes in $dest$:

- if all nodes in $dest$ are in the physical neighborhood of node n, i.e., $\forall i \in dest : i \in N$, then message msg is appended to the tail of the channel variable ch at the same time at all processes in $dest$;
- if $dest$ is a predefined value $BCAST$, then message msg is simultaneously appended to the tail of the channel variable ch of all processes that are in n's *physical* neighborhood N;
- if at least one node in $dest$ is *not* in N, then message msg is appended to the tail of the channel variable ch at all processes in $dest$ possibly at different times. This models multi-hop communication among nodes.

The above distinctions are required to model the broadcast nature of the wireless medium used in WSNs. Specifically, we need to render the fact that transmissions towards a broadcast address deliver data also to nodes that are not necessarily known to the sender, i.e., they are not in its logical neighborhood. We assume a synchronous communication model, i.e., there is a known upper bound on message transmission delays.

Faults. A fault model stipulates the way programs may fail. We consider two types of failures: i) *transient* failures, and ii) *pseudo-crash* failures. The former are failures that corrupt the program state by arbitrarily altering values of variables. Thus, they model data corruption due to, e.g., bit-flips caused by defective hardware [5]. Note that transient failures may affect the channel variable *ch* as well, modeling data corruption during message transmission. We assume that transient failures do not occur infinitely often, otherwise system liveness may be compromised. Differently, we say that a node n_1 has *pseudo-crashed* if there is a node n_2 that can no longer receive messages from n_1. Thus, n_1 appears as crashed to n_2. This may happen for several reasons, e.g., the directed communication link between n_1 and n_2 failed, or because node n_1 crashed due to energy depletion. We also assume a (correct) boot phase wherein nodes acquire their 2-hop neighborhood before failures start occurring.

Formally, our fault model is a set F of faulty actions [24]. These are similar to program actions, as they may modify the variables of programs and thus alter the program state. We say that a fault *occurs* if a fault action is executed. Fault actions can interleave program actions and they might or might not be executed when enabled. We say a computation is *F-affected* if the computation contains program transitions and transitions from fault model F.

In the following, we consider two sets F_{tr} and F_{pc} of faulty actions to model transient and pseudo-crash failures, respectively. A program with transient failures contains an action that assigns to any of its variable an arbitrary value from its domain. As logical neighborhoods are part of the program state, transient failures may alter them as well. Differently, a pseudo-crash is modeled by an action that removes node entries from information describing the physical neighborhoods as soon as these information enter the program state.

Definitions. To complete our modeling, we provide formal definitions of some concepts used in the following sections. Firstly, we define how nodes remove from their logical neighborhoods the devices that they assume to be unable to exchange data with:

Definition 1 (Remove): Consider a physical topology $S = (\Gamma, \Sigma)$ and a logical topology $S^l = (\Gamma^l, \Sigma^l)$ of S. We say that *a node* $p \in \Gamma^l$ *removes a node* $q \in \Gamma^l$ if $(q \in P^l) \wedge (\Sigma'^l = \Sigma^l \oplus \{(p, P^l \setminus \{q\})\})$, where Σ'^l represents the updated version of Σ^l and $\oplus$ represents an update function [25].

Finally, we detail two additional terminologies we make use of in the rest of the paper:

Definition 2 (Faulty nodes): A node n_1 is said to be *faulty*

if its state has been altered by a transient fault. Otherwise, the node is *non-faulty*.

Definition 3 (Localized algorithm): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, problem specification ¶ for S^l, and an algorithm A that solves ¶, algorithm A is said to be *localized* if the message complexity of algorithm A varies with the size of a neighborhood. It is *global* if the message complexity varies with the network size.

III. Problem Specification

We define three increasingly weaker specifications of neighborhood view consistency in terms of safety and liveness properties. Note that the first two specifications are analogous to perfect and eventually perfect failure detectors [26] respectively, although in a different context. Recall that the physical (logical) neighborhood of node n is denoted by N (N^l).

Definition 4 (Strong view consistency): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, and two nodes $n, m \in \Gamma^l$, a program *solves the strong neighborhood view consistency problem for* S^l if every computation of the program satisfies the following:

- (Safety) If a node n removes a node m, then m has pseudo-crashed.
- (Liveness) If node m has pseudo-crashed, then eventually $\forall n : n \in M : n$ removes m.

The strong view consistency of neighborhood states that, whenever a node n removes a pseudo-crashed neighbor m, every node that has m in its physical neighborhood eventually removes it. In addition, the safety specification rules out nodes mistakenly removing other nodes from their neighborhood.

Definition 5 (Stabilizing strong view consistency): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, and two nodes $n, m \in \Gamma^l$, a program *solves the stabilizing [27] strong view consistency problem for* S^l if every program computation satisfies the following:

- (Safety) Eventually, if a node n removes a node m, then m has pseudo-crashed.
- (Liveness) If node m has pseudo-crashed, then eventually $\forall n : n \in M : n$ removes m.

Because transient faults can arbitrarily corrupt the program state, nodes may end up wrongly removing non-pseudo-crashed devices. The stabilizing strong view consistency ensures that eventually strong view consistency is established again. Thus, it has similar liveness property as strong view consistency. However, nodes are allowed to finitely make mistakes by removing nodes that have *not* pseudo-crashed from their logical neighborhoods.

Definition 6 (Weak view consistency): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, and two nodes $n, m \in \Gamma^l$, a program *solves the weak neighborhood view consistency problem for* S^l if every computation of the program satisfies the following:

- (Safety) Eventually, if a node n removes a node m, then m has pseudo-crashed.
- (Liveness) If node m has pseudo-crashed, then eventually $\forall n : n \in M : n$ removes m, or a fault is eventually detected.
- (Validity) The fault is signalled only if there are faulty processes in the network.

Therefore, weak view consistency attempts to enforce stabilizing consistency but raises a fault signal if this is not possible. However, the fault is only raised if it actually occurred[2].

IV. Impossibility Results

In this section, we show that i) there exists no algorithm to solve the strong view consistency problem, and ii) there exists no *localized* algorithm to solve the stabilizing strong view consistency problem. Nevertheless, we provide a global algorithm that solves the stabilizing view consistency problem.

A. Strong View Consistency

Intuition. The safety specification of the strong view consistency problem prohibits a node n_1 from removing a node n_2 from its logical neighborhood unless n_2 has pseudo-crashed. However, in the presence of both transient and pseudo-crash failures, it is generally impossible to distinguish the former from the latter. For instance, a transient fault may corrupt the memory content and make a node believe that communication is not possible towards another device, even though this is not the case. This intuition leads to the following:

Theorem 1 (Impossibility): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, a transient fault model F_{tr}, and a pseudo-crash fault model F_{pc}, there exists no algorithm that can solve the strong view consistency problem in the presence of F_{tr} and F_{pc}.

Proof: Assume that an algorithm A exists that solves the strong view consistency problem.

Consider a F_{tr}-free, F_{pc}-affected computation C_1 of A and a state s_i in C_1 where a node n removes a node m. Since A solves the strong view consistency problem, according to the safety property, m has pseudo-crashed. According to the liveness property of the strong view consistency problem, eventually, there is a state s_j in C_1, $j > i$, where algorithm A causes every node that is a neighbor of m to remove m, i.e., $\forall k \in M : k$ removes m.

Now we construct a computation $C_2 = s_0 \cdot s_1 \ldots$ of A which is F_{tr}-affected and F_{pc}-free, and is similar to C_1, in that in state s_i, a node n removes a node m because of a transient fault. Since $C_2 = C_1$ up to and including state s_i, algorithm A again ensures that in state s_j, for every node $k \in M$, k removes m. This violates the safety property of the strong view consistency problem, since the computation is F_{pc}-free.

Thus, A cannot distinguish between computations C_1 and C_2. Hence, A cannot exist. ■

Note that this result applies to both localized and global algorithm, although the proof deals with the localized case, as it is based on neighborhood information. For a global algorithm that computes the global network topology [18], the

[2]Henceforth, we will only say that "a program solves the strong/stabilizing strong/weak view consistency problem" whenever S^l is obvious from the context.

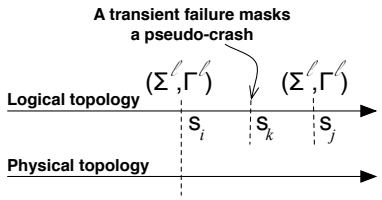


Fig. 2. A transient failure masks a pseudo-crash.

topology information may still be corrupted due to incorrect information coming from various processes, and the strong view consistency specification cannot be satisfied.

The key requirement of the strong view consistency is that no node may mistakenly remove a non-pseudo-crashed node. In the next section, we relax this requirement and study the stabilizing version of the strong view consistency problem.

B. Stabilizing Strong View Consistency

Intuition. The stabilizing strong view consistency problem allows nodes to finitely make mistakes when removing nodes, i.e., nodes that have not pseudo-crashed are allowed to be wrongly removed. On the other hand, we still require pseudo-crashed nodes to be eventually removed. However, in the presence of transient faults, a pseudo-crash can go undetected if it is masked by a transient fault that clears from a node's logical state all evidence of the pseudo-crashing neighbor, i.e., the logical neighborhood of a node n is corrupted such that a pseudo-crashing neighbor m is deleted from n's logical neighborhood. Then, as illustrated in Figure 2, it is *as if* node n has never seen the pseudo-crashed neighbor m and, consequently, it will never remove it. In a such a situation, the liveness property of the stabilizing strong view consistency problem is violated, which leads to the following:

Theorem 2 (Impossibility): Given a logical topology $S^l = (\Gamma^l, \Sigma^l)$, a transient fault model F_{tr}, and a pseudo-crash fault model F_{pc}, there exists no localized algorithm that can solve the stabilizing strong view consistency problem in the presence of F_{tr} and F_{pc}.

Proof: Assume that a localized algorithm A exists that solves the stabilizing strong view consistency problem.

We assume a F_{tr}-free and F_{pc}-free computation $C_1 = s_0 \cdot s_1 \ldots$ of A. According to the liveness property of the stabilizing strong view consistency problem, every logical neighborhood remains unchanged in C_1.

Now we construct a computation $C_2 = s_0 \cdot s_1 \ldots$ of A which is F_{tr}-affected and F_{pc}-affected, and is similar to C_1, in the sense that, in any given state s_j of C_2, there is no node n that removes another node m because transient faults mask every pseudo-crash, as we illustrated in Figure 2. The construction is as follows: assume $C_1 = s_0 \cdot s_1 \ldots s_i \cdot s_{i+1} \ldots$, and $C_2 = s_0 \cdot s_1 \ldots s_i \cdot s'_{i+1} \ldots$, and a node m that pseudo-crashes when C_2 is in s_i. Node m can only be removed in state $s_j, j > i$. Now, in state $s_k, i \leq k < j$, a transient failure occurs at node n such that $m \notin N^l$. In state $s_j, j > k$, n does not remove m.

Since C_2 is the same as C_1, algorithm A keeps all logical neighborhoods unchanged. This violates the liveness property of the stabilizing strong view consistency problem, since C_2 is an F_{pc}-affected computation.

Thus, A cannot distinguish between computations C_1 and C_2. Hence, A cannot exist. ■

```
process j
variables
    start: boolean, initially true;

    % the network topology returned by the discovery algorithm
    top: set of tuples, initially {(j, J)}

    % a timer variable for periodic topology discovery
    discover: timer init Δ;

actions
    init:: start → start := false;
    discovery:: timeout(discover)→
                        top:=topology_discovery();
                        set(discover,Δ);
    detect:: (∃q : j ∈ Q ∧ (q ∉ J)) → ∀i : i ∈ Q :
                        send(j cannot detect q,i);
    remove:: rcv(p cannot detect b, j)→ top:= top⊕{(j, J \ {b})};
```

Fig. 3. Algorithm *GStrongC* solving stabilizing strong view consistency.

Note that, differently from the impossibility result for the strong view consistency problem in Section IV-A, this impossibility result applies only to localized algorithms, since only local knowledge of pseudo-crashes is available. With complete knowledge of the logical network topology, it is possible to develop an algorithm that solves the stabilizing strong view consistency problem. The topology of the network can be obtained using a stabilizing topology discovering algorithm [18]. The topology algorithm returns a set of $\{(node, neighborhood)\}$ pairs that conveys not only information about pseudo-crashed nodes, but also about correct nodes.

A global algorithm satisfying the stabilizing strong view consistency specification is *GStrongC*, described in Figure 3. Note that, according to Definition 3, algorithm *GStrongC* is a global algorithm since its message complexity varies with the network size. Since the topology discovery algorithm is self-stabilizing, it means that eventually the topology is correct. Algorithm *GStrongC* then identifies pseudo-crashes even if the node that would have detected it has no recollection of the pseudo-crashed neighbor. If there exists a node q which contains a node j in its logical neighborhood, but not vice-versa (as specified in the guard of action *detect*), algorithm *GStrongC* makes j send a notification about this to all neighbors of q. Upon receiving this notification, all neighbors of q remove q. Thus, we state that:

Theorem 3 (Global algorithm): Algorithm *GStrongC* solves stabilizing strong view consistency.

Proof: (Outline). When faults stop, *top* is eventually correct because of the stabilizing property of the topology discovery algorithm that returns a set of nodes and their respective neighborhoods, i.e., $\{n, N\}$. In this situation, process j can determine every node that it cannot detect, and send notification messages to all neighbors of these undetected nodes (action *detect*). Upon eventually receiving the notification, these nodes remove the undetected devices. Stabilization of *GStrongC* follows trivially from the stabilization of the topology discovery algorithm of [18]. ■

process j
variables
start: boolean, initially true;

% logical neighborhoods of j's logical neighbors
% notice that $N[j]$ implements J^l
N[]: array of set of ids, initially $N[j] = J$;

% identifier of nodes detected during a round
live: set of ids, initially J^l;

% a timer that sets the beacon period
beacon: timer init Δ;

% a timer that sets the detection period, $\Theta < \Delta$
detect: timer init Θ;

actions
init::*start* $\rightarrow$
start, live := false, $\emptyset$;
send$(\langle j, N[j]\rangle, BCAST)$;
dissem:: **timeout**(*beacon*)$\rightarrow$
send$(\langle j, N[j]\rangle, BCAST)$;
set(*beacon*,Δ);
set(*detect*,Θ);
compute:: **rcv**$((\langle p, P\rangle, r) \rightarrow$
live:= *live* $\cup\{p\}$;
N[p]:= P;
detect:: **timeout**(*detect*)$\rightarrow$
$\forall p : p \in (N[j] \backslash live)$: $\forall q : q \in N[p]$:
send(j cannot detect p,q);
$\exists p : p \in (N[j] \backslash live) \wedge (N[p] = \emptyset)$:
send(fault,$BCAST$);
N[j]:= *live*; *live*:= $\emptyset$;
remove:: **rcv**(p cannot detect b,j)$\rightarrow$
if $(b \in N[j])$ **then**
$N[j] := N[j] \setminus \{b\}$;
else send(fault,$BCAST$); **fi**
skip:: **rcv**(fault, r)$\rightarrow$ skip;

Fig. 4. Algorithm *WeakC* that solves the weak view consistency problem.

However, topology discovery algorithms are computationally expensive. Further, to solve the stabilizing strong view consistency problem, the topology discovery algorithm needs to be executed periodically so as to eventually provide a correct topology, after faults stop occurring. Therefore, this solution would be impractical on resource-constrained devices such as those employed in WSNs.

To redress this problem, in the following section we study the weak view consistency problem and ultimately describe a localized algorithm that satisfies its specification, along with a correctness proof.

V. Solving Weak View Consistency

Since Theorem 1 demonstrates that it is impossible to prevent nodes from mistakenly removing devices that have not pseudo-crashed, and Theorem 2 demonstrates that it is impossible to guarantee that pseudo-crashed nodes are eventually removed in the presence of transient faults and using only local knowledge, the weak view consistency problem specifies that a *fault* is signaled whenever such a fault is detected and only if the fault occurred.

An algorithm providing the above functionality is described in Figure 4. During initialization, every node knows its physical neighborhood. This is achieved by executing a round of beaconing. Next, the nodes periodically exchange *logical* neighborhood information with their *physical* neighbors using the **send** with $dest$ set to $BCAST$ (action *dissem*). Once the dissemination phase is over, the *beacon* timer is set for the next phase of neighborhood exchange. Within a time period Θ greater than the message latency but smaller than the beacon period Δ, nodes collect and process neighborhood messages. This way, every node eventually holds information about the 2-hops logical neighbors within a given time period. Based on this, every node compares its current 1-hop logical neighborhood with the newly collected one (action *compute*). Every pseudo-crashed node n is flagged, and a notification message is sent to every neighbor of n (action *detect)*. Upon receiving a notification, a node proceeds to remove the undetected neighbors from its logical neighborhood (action *remove*). Whenever transient faults occur, *WeakC* signals a fault which can act as an indication to higher-level algorithms and protocols. This is modeled by a *fault* message sent to destination $BCAST$.

In the following, we state four lemmas that we use to derive the correctness of algorithm *WeakC*.

Lemma 1: Consider a logical topology $S^l = (\Gamma^l, \Sigma^l)$. The following predicate is an invariant of *WeakC*:
$(\exists j : j \in \Gamma^l : j.ch \neq \langle\rangle) \vee$
$(\exists j : j \in \Gamma^l : (J^l \setminus j.live = \emptyset) \wedge (j.ch = \langle\rangle)) \vee$
$(\exists j : j \in \Gamma^l : \forall k \in J : (fault) \in k.ch) \vee$
$(\forall j : (J \setminus j.live \neq \emptyset) \wedge (j.beacon = 0))$
$(\forall k \in (J \setminus j.live)) :$
$\forall i : i \in K : (k \notin I) \vee ((j \text{ cannot detect } k) \in i.ch)$

Proof: To prove that the predicate is an invariant, we show that the predicate holds in the initial state, and that the predicate is closed under every program action, i.e., the execution of every action of any process do not invalidate the predicate.

First, we show that the predicate holds in the initial state. The predicate is satisfied as the second disjunct trivially holds (initially $J^l = J$ and $j.live = J \Rightarrow J^l \setminus j.live = \emptyset$ and all channels are empty). Actions *init* and *dissem* modify the channel content, by adding to the channel. All channels are therefore non-empty, satisfying the first disjunct. Hence the predicate is closed under actions *init* and *dissem*. During *compute*, $j.ch$ is non-empty, satisfying the first disjunct. Execution of action *detect* satisfies either the third or the fourth disjunct. Hence, the predicate is closed under *detect*. Similarly, execution of *remove* causes either the third or the fourth disjunct to be satisfied.

Executing action *skip* affects $j.ch$. The first time a fault message is sent is during *detect*. Thus, the fault message necessarily appears after any notification message, and no notification message will appear after a fault message. When the fault message is removed, either $j.ch$ becomes empty or $j.ch$ contains more fault messages. If there are more fault messages to process, the first disjunct is satisfied. If there are no more fault messages to process, then $j.ch$ is empty, and the fourth disjunct is satisfied.

Thus, the predicate is true in the initial state, and is

preserved by *WeakC* actions, hence is an invariant of *WeakC*. ■

The following lemma proves that either liveness of stabilizing strong view consistency is obtained or a fault is signaled:

Lemma 2: If a computation of *WeakC* contains a state s_i where, for process p and process $n \in P^l$, $n \in P^l \setminus p.live$, then there is a state $s_j, j > i$ in the computation where either a fault is signaled or all neighbors of n have removed n.

Proof: As shown above, the predicate in Lemma 1 is an invariant of *WeakC*. When $J^l \setminus j.live \neq \emptyset$, for some process j, either a fault message is sent, or, upon timeout, all neighbors will remove the pseudo-crashed node. ■

Symmetrically, the following lemma demonstrates the safety specification:

Lemma 3: If a computation of *WeakC* contains a state s_n where, for some processes p and k, p has removed k, then there is a state $s_m, m < n$ in the computation where some process j considers k as pseudo-crashed.

Proof: When p removes k from P^l in state s_n, it implies that $k \in P^l$ in some state $s_m, m < n$. This implies that in state s_m, $p.ch$ contains a message of the form (j cannot detect k). This message is sent during action *detect*, when $k \in J^l \setminus j.live$, and thus k appeared as pseudo-crashed. ■

Finally, we prove the validity condition in the specification of weak view consistency:

Lemma 4: A fault message is issued if there is a transient fault in the system.

Proof: A fault message is sent under two different conditions: i) in action *detect*, whenever a node is not detected, but no neighborhood information of the node is held, indicating a fault[3], ii) in action *remove*, whenever a node j receives a notification message to remove a node that it is not found in its neighborhood, which indicates a fault in J^l or in the information held at the sender of the notification. ■

As a consequence, we can state the following regarding algorithm *WeakC* and weak view consistency:

Theorem 4: Algorithm *WeakC* satisfies the weak view consistency specification.

Proof: Follows from Lemma 2, Lemma 3, and Lemma 4. ■

VI. Implementation

We implement algorithm *WeakC* in Rime [12], the network stack of the Contiki [13] WSN operating system. Our implementation is very lightweight, as it only occupies 1.2 KBytes of program memory and 296 bytes of data memory. Optimizing these figures is pivotal when the target hardware platform is provided with only a few KBytes of memory, e.g., as in the case of the widespread TMote Sky node [28].

In addition to the mechanisms in Figure 4, in the implementation we use a neighborhood view identifier to indicate changes in the logical neighborhood. For instance, in the topology of Figure 1(a), if the link between node 4 and node 7 disappears and no other failures occur, node 2 removes node 4 and proceeds to re-acquire it at the next exchange of neighborhood information. In doing so, the neighborhood identifier is changed as well. Thus, although the logical neighborhood at node 2 appears the same as the one before the link failure, higher-level protocols may recognize the topology change based on a different view identifier.

Next, we describe the two constituents of our implementation: the API available to higher-level algorithms and protocols, and the network support required.

API. Two primitives are made available:

```
uint8_t getNeighborhood(uint16_t* ids,
                        uint8_t* num);
bool isNeighbor(uint16_t id);
```

The former is used to query the current logical neighborhood. Memory areas to hold the set of corresponding node identifiers and their number are given as input parameters. Callers obtain these information along with the neighborhood view identifier as return parameter. Differently, **`isNeighbor`** checks whether the node identifier given as parameter belongs to the current logical neighborhood view.

Network Support. To implement the periodic exchange of neighborhood information, we straightforwardly rely on the 1-hop broadcast functionality of Rime, embedding within messages both the identifier of the sender node and its current logical neighborhood view.

Differently, messages carrying neighborhood notifications require reliable multi-hop multicast to reach their destinations. This could be implemented using explicit routing tables. As in our algorithms all nodes may be sources or destinations of multicast messages, this would entail maintaining large routing tables, with considerable use of memory resources. In addition, routing tables may be unusable when nodes or link fail, i.e., exactly when our algorithm needs multicast functionality. However, we observe that in our scenario multicast messages most often target a subset of the 2-hop neighbors of a device. As WSNs tend to be dense networks [7], [29], in case of node failures the distance to the former 2-hop neighbors is very unlikely to increase drastically, if not at all.

Based on this observation, we opt for an expanding-ring n-hop flooding for routing, embedding the list of destinations within messages. Starting with $n = 2$, we periodically retransmit the message by doubling the number of hops traversed until we receive an acknowledgment from all destination nodes. Acknowledgments are sent along the reverse path towards the originator using reliable 1-hop unicast. This mechanism provides reliable multicast with very little memory overhead compared to existing solutions [30] and generates no control traffic for route maintenance.

VII. Performance Evaluation

The objective of our evaluation is to verify that our implementation of *WeakC* solves the weak view consistency problem efficiently and with minimum disruption for upper-level

[3]This does not apply to the boot phase, when we assume no fault may occur

Parameter	Range	Default value	Step
Average neighbors D_p	4 nodes—20 nodes	10 nodes	2 nodes
Node failure P_{nf}	2%—10%	6%	2%
Link failure P_{lf}	2%—10%	6%	2%
Data corruption P_c	2%—10%	2%	2%

Fig. 5. Simulation settings.

algorithms and protocols. To this end, we use both simulation experiments and a real-world testbed. The former enables fine-grained control and inspection of the system operation, the latter assesses the effectiveness of our implementation in realistic environments.

Metrics. We study the following figures:

- the *latency* to establish consistent neighborhood views, i.e., the time from when some change in the physical topology is detected to when all nodes affected re-gain consistent neighborhood views or a fault message is sent;
- the *network overhead*, defined as the total number of packets exchanged at the physical level to carry out a view change, including retransmissions for reliability;
- the total *energy* spent during a view change, measured using Contiki's energy estimation mechanism [31].

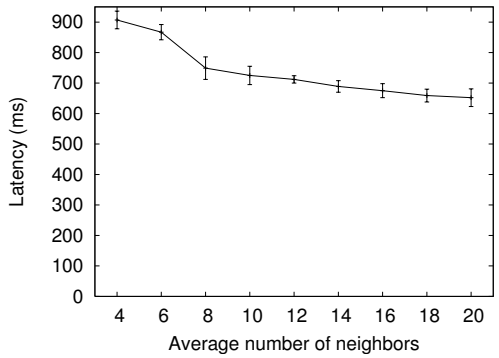
The latency figure provides a measure of the time spent by higher-level functionality with inconsistent topology information. This may be source of inefficient behaviors, and must thus be minimized. The network overhead caters for an indication of the network resources employed. Network traffic may generate collisions and thus message losses, possibly affecting the operation of upper-level mechanisms. This metric must therefore be minimized as well. Finally, energy is a fundamental metric in WSNs, as these systems are mostly battery-powered. The less the energy spent by our algorithm, the longer the system operates.

A. Simulation

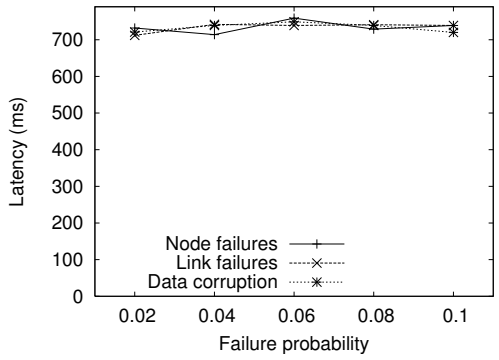
Settings. We use Cooja [32], the Contiki's simulator. We randomly deploy 100 nodes with varying densities. Note that our algorithm is localized. Thus, it is the network density that determines the complexity of the processing. We quantify network density with the average number of nodes D_p in the physical neighborhood of a device. Simulations are divided in rounds of 30 seconds. The first round is reserved as boot phase to acquire the initial topology information, and no faults occur. During the following rounds, we independently simulate different types of faults:

- to simulate node crashes, each node is associated a probability P_{nf} of being shut down by the simulator. To keep the network density constant, at each round new nodes appear in the simulation with the same probability.
- to model link failures, every link has probability P_{lf} of disappearing. Failed links are restored after 2 rounds to maintain, on average, the same network density.
- to model data corruption, we randomly modify an entry in a node's logical neighborhood with probability P_c.

The values used for the various settings are summarized in Figure 5. The values for D_p reflect the network densities in real deployments [1]. Differently, the failure probabilities overestimate the occurrence of similar situations in real-world scenarios. For instance, in a network of 100 nodes, a node failure probability $P_{nf} = 6\%$ per round entails that 6 devices fail every 30 seconds. We intentionally pushed on these values to stress our implementation.



(a) Latency against node density D_p.



(b) Latency against failure probability for different failures.

Fig. 6. Latency.

We use Cooja's free-space radio propagation model and Contiki's implementation of the XMAC protocol [33] working with the default settings. We configure our network support, described in Section VI, to exchange neighborhood information every 5 seconds, and we set the acknowledgement time out in our reliable multicast protocol to 300 ms. Our simulations explore all combinations of the settings in Figure 5 by performing 100 repetitions with different random topologies for each combination. The results presented hereafter are averages over these repetitions.

Results. The latency results we obtained are shown in Figure 6. Figure 6(a) illustrates the latency to re-establish consistent views with varying network densities and default failure probabilities. The absolute values at stake are very limited, being always within one second. This is at least one order of magnitude smaller than the dynamics of typical WSN applications where, for instance, sensed data is reported to the user once every 30 seconds [1]. Therefore, higher-level MAC and routing protocols are very likely not to be affected by the operation of our algorithm. The results also illustrate a correlation between latency and network density. In sparse networks the expanding ring technique we employ for multicast may repeat the transmission with increased maximum hops to reach the target nodes. This situation becomes very

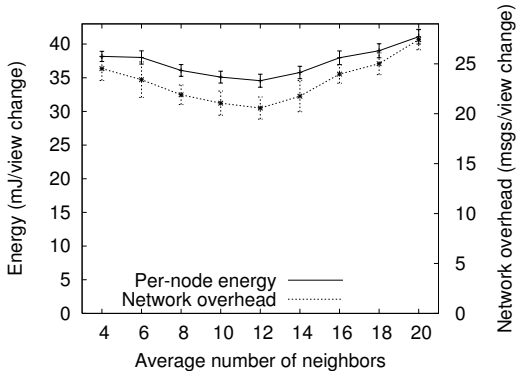


Fig. 7. Per-node energy consumption and network overhead.

unlikely with more than 8 neighbors per each device. In these cases, a single 2-hop flooding is usually sufficient.

Figure 6(b) illustrates the latency for specific classes of failures against failure probability. In every case, the remaining simulation parameters are set to their default values. The trends are almost constant in this case and essentially do not differ for different types of failures. This highlights the ability of our implementation to deal effectively with diverse failures in highly unreliable scenarios. We also verified that combinations of failures probabilities outside the ranges of Figure 5 do exist that generate a sharp latency increase, up to about two seconds. For instance, one such combination is with $P_{nf} = P_{lf} = 30\%$ and $P_c = 10\%$. With these values failures occur so often that the control traffic generates frequent collisions at the physical layer, and consequently multiple retransmissions are required for reliable delivery. We do believe, however, that similar scenarios are quite unrealistic.

The chart in Figure 7 depicts the trends in network overhead against node density. As already observed, when the network is sparse, the expanding-ring technique requires more retransmissions. Conversely, with dense networks the number of possible destinations increases and so does the probability that multiple nodes simultaneously send acknowledgements back. In this situation, acknowledgments are likely to collide at the original sender, a problem known as ack implosion [7]. As these messages are sent using Rime's reliable unicast, which is based on multiple retransmission, more packets are sent to recover such collisions. In all cases, however, the number of messages being exchanged is very limited. Therefore, upper-level protocols perceive very little disruption. The trend in energy consumption against network density, still in Figure 7, resemble the network overhead. This was expected, as radio communication dominates energy consumption in WSNs [34]. The additional energy consumption due to our implementation is negligible, especially considering the energy budget that modern WSNs are equipped with [2].

We also verified that both network overhead and energy consumption are essentially constant against varying failure probabilities. Indeed, during single view changes the network overhead —and hence energy consumption— is affected only by the the shape of the physical topology when faults occur, and not by the frequency of faults.

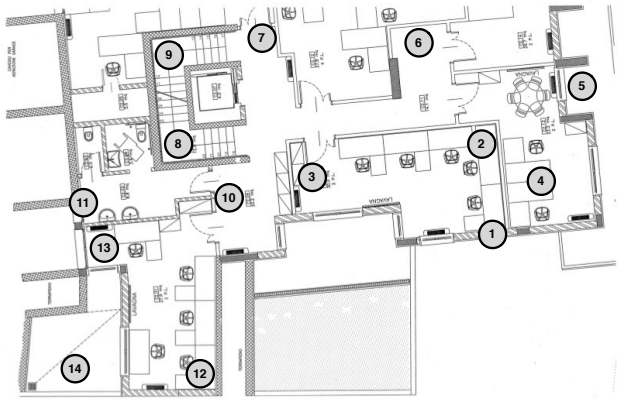


Fig. 8. Testbed deployment.

B. Testbed

Settings. We use our lab testbed, illustrated in Figure 8, to evaluate our implementation in a real-world setting. The testbed is composed of 14 TMote Sky nodes [28] installed in an office environment. With nominal transmission power, every node typically has an average of 5 neighbors.

We deploy on all nodes the same implementation as in our simulation experiments. To simulate node failures, every node randomly turns off the radio with probability $P_{nf} = 6\%$ every 30 seconds, and reactivates it after two minutes. Data corruption is triggered within the same period with artificial changes in the entries of a node's logical neighborhood with probability $P_c = 2\%$. Differently from the simulation experiments, however, we do *not* cause artificial link failures, and just let the wireless links fluctuate as they naturally do in office-like environments [7]. We time synchronize the nodes using Contiki's time synchronization protocol, and dump a timestamped description of all events of interests on flash memory. We let the nodes run for 24 hours and retrieve these data at the end of the experiment for off-line analysis.

Results. The average latency to re-establish consistent neighborhood views observed in our testbed is of 923.13 ms with a standard deviation of 23.31 ms across different nodes. This is comparable to the results we obtained in simulation, shown in Figure 6(a), confirming the validity of our argument in a real-world environment.

Nevertheless, the network-level behavior is different. On average, 40.35 messages are transmitted per view change, which is higher than what we showed in Figure 7. This was expected, as simulators can reproduce the behavior of real-world wireless transmissions only to some extent [7]. Interestingly, however, increased network overhead is not a general trend, but it is caused by failures at specific devices. Figure 9 plots this metric depending on the node that somehow triggers the view change, i.e., because it turns off the radio or a link to it stops transmitting. The number of messages exchanged is markedly higher when it is one among node 4, 5, 10, and 11 to trigger our algorithm. Most probably, the devices involved in the view change when one such node fails have poor connectivity with each other, which requires more retransmissions for reliable delivery of multicast messages.

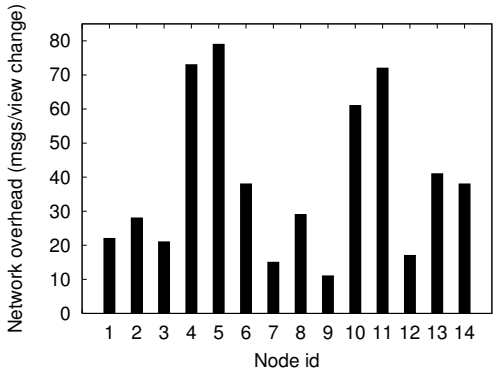


Fig. 9. Network overhead depending on the node triggering the view change.

The same reasoning applies to energy consumption, as in WSNs this is dominated by the radio operation.

VIII. Conclusion and Future Work

We presented three increasingly weaker specifications for the problem of neighborhood view consistency in WSNs. We proved that the two stronger specifications cannot be solved using only local knowledge, and presented an algorithm to solve the weakest specification, along with a correctness proof. We implemented the algorithm in a commonly used WSN network stack and assessed its performance both in simulation and in a real-world testbed. Our implementation can be used as a building block in higher-level algorithms and protocols.

Our ongoing investigation includes theoretical work to study how Byzantine failures may impact the results we obtained, alongside with further implementation effort to make the solutions we will devise in this context available as building blocks in the WSN stack.

Acknowledgements. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, under EU contract FP7-2007-2-224053; by VINNOVA, the Swedish Agency for Innovation Systems; and by SSF, the Swedish Foundation for Strategic Research.

References

[1] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proc. of the 1^{st} ACM Int. Wkshp. on Wireless Sensor Networks and Applications (WSNA)*, 2002.

[2] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proc. of 7^{th} Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.

[3] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: scalable coordination in sensor networks," in *Proc. of the 5^{th} Int. Conf. on Mobile Computing and Networking (MOBICOM)*, 1999.

[4] I. Demirkol, C. Ersoy, and F. Alagoz, "MAC protocols for wireless sensor networks: A survey," *IEEE Communications Magazine*, vol. 44, no. 4, 2006.

[5] N. Finne, J. Eriksson, A. Dunkels, and T. Voigt, "Experiences from two sensor network deployments self-monitoring and self-configuration keys to success," in *Proc. of Int. Conf. on Wired/Wireless Internet Communications (WWIC)*, 2008.

[6] A. Sharma, L. Golubchik, and R. Govindan, "On the prevalence of sensor faults in real-world deployments," in *Proc. of the IEEE Int. Conf. on Sensor and Ad-hoc Communications and Networks (SECON)*, 2007.

[7] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *Proc. of the 1^{st} Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2003.

[8] J. Beutel *et al.*, "Operating a sensor network at 3500 m above sea level," in *Proc. of the 8^{th} Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2009.

[9] B. J. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *Proc. of 2^{nd} Int. Wkshp. on Information Processing in Sensor Networks (IPSN)*, 2003.

[10] K. Langendoen and N. Reijers, "Distributed localization in wireless sensor networks: A quantitative comparison," *Computer Networks*, vol. 43, no. 4, 2003.

[11] X.-Y. Li, P.-J. Wan, Y. Wang, and O. Frieder, "Sparse power efficient topology for wireless networks," in *Proc. of the 35^{th} Annual Hawaii International Conference on System Sciences (HICSS)*, 2002.

[12] A. Dunkels, F. Oŝterlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *Proc. of the 5^{th} Conf. on Networked Sensor Systems (SENSYS)*, 2007.

[13] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. of 1^{st} Wkshp. on Embedded Networked Sensors*, 2004.

[14] L. Paradis and Q. Han, "A survey of fault management in wireless sensor networks," *Journal Networkvi System Management*, vol. 15, no. 2, 2007.

[15] Z. Ma and A. Krings, "Dynamic hybrid fault models and the applications to wireless sensor networks," in *Proc. of the 11^{th} Int. Symp. on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, 2008.

[16] S. Kulkarni and M. Arumugam, *SSTDMA: A self-stabilizing MAC for sensor networks*. IEEE Press, 2006.

[17] M. Pease, R. Shostak, and L. Lamport, "Reaching agreements in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, 1980.

[18] T. Masuzawa, "A fault-tolerant and self-stabilizing protocol for topology problem." in *Proc. Workshop on Self-stabilizing Systems*, 1995.

[19] T. Masuzawa and S. Tixeuil, "A self-stabiliizng link-coloring algorithm resilient to unbounded byzantine faults in arbitrary networks," in *Proc. OPODIS*, 2005.

[20] G. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: A comprehensive study," *ACM Computing Surveys*, vol. 33, no. 4, 2001.

[21] A. Ganesh, A. Kermarrec, and L. Massoulié, "Peer-to-peer membership management for gossip-based protocols," *IEEE Transactions on Computers*, vol. 52, no. 2, 2003.

[22] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, "Clock synchronization for wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 3, no. 3, 2005.

[23] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, 1985.

[24] A. Arora and S. S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components," in *Proc. of the 18^{th} Int. Conf. on Distributed Computing Systems (ICDCS)*, 1998.

[25] B. Potter, J. Sinclair, and D. Till, *An introduction to formal specification and Z*. Prentice Hall, 1996.

[26] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, 1996.

[27] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[28] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proc. of the 5^{th} Int. Conf. on Information Processing in Sensor Networks (IPSN)*, 2005.

[29] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proc. of the 22^{th} Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.

[30] Q. Cao, T. He, and T. Abdelzaher, "ucast: Unified connectionless multicast for energy efficient content distribution in sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 2, 2007.

[31] A. Dunkels, F. Österlind, N. Tsiftes, and Z. He, "Software-based online energy estimation for sensor nodes," in *Proc. of the 4^{th} Wrkshp. on Embedded Networked Sensors (Emnets IV)*, 2007.

[32] T. Voigt, N. Finne, J. Eriksson, A. Dunkels, and F. Oŝterlind, "Cross-level sensor network simulation with COOJA," in *Proc. of the 1^{st} Int. Wrkshp. on Practical Issues in Building Sensor Network Applications (SenseApp)*, 2006.

[33] M. Buettner, G. Yee, E. Anderson, and R. Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *Proc. of the 3^{rd} Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*, 2006.

[34] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *IEEE Communication Mag.*, vol. 40, no. 8, 2002.