Neighborhood View Consistency in Wireless Sensor Networks

ARSHAD JHUMKA, University of Warwick, Coventry, UK
LUCA MOTTOLA, Politecnico di Milano, Italy and SICS Swedish ICT

Wireless sensor networks (WSNs) are characterized by *localized interactions*, that is, protocols are often based on message exchanges within a node's direct radio range. We recognize that for these protocols to work effectively, nodes must have consistent information about their shared neighborhoods. Different types of faults, however, can affect this information, severely impacting a protocol's performance. We factor this problem out of existing WSN protocols and argue that a notion of *neighborhood view consistency* (NVC) can be embedded within existing designs to improve their performance. To this end, we study the problem from both a theoretical and a system perspective. We prove that the problem cannot be solved in an asynchronous system using any of Chandra and Toueg's failure detectors. Because of this, we introduce a new software device called *pseudocrash failure detector* (PCD), study its properties, and identify necessary and sufficient conditions for solving NVC with PCDs. We prove that, in the presence of transient faults, NVC is impossible to solve with any PCDs, and thus define two weaker specifications of the problem. We develop a global algorithm that satisfies both specifications in the presence of unidirectional links, and a localized algorithm that solves the weakest specification in networks of bidirectional links. We implement the latter atop two different WSN operating systems, integrate our implementations with four different WSN protocols, and run extensive micro-benchmarks and full-stack experiments on a real 90-node WSN testbed. Our results show that the performance significantly improves for NVC-equipped protocols; for example, the Collection Tree Protocol (CTP) halves energy consumption with higher data delivery.

1. INTRODUCTION

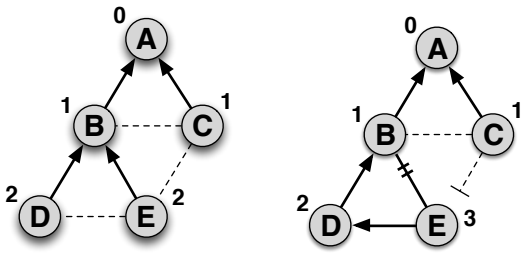
Wireless Sensor Networks (WSNs) are distributed systems of resource-constrained embedded nodes. Because of their characteristics, such as ease of deployment, WSNs have become a viable tool to harvest fine-grained data from the physical world and to act on it.

Problem. Different types of faults may occur in WSNs. A node's memory may be *corrupted* because of defective hardware [Werner-Allen et al. 2006; Finne et al. 2008], erroneous sensor readings [Sharma et al. 2007], or software bugs such as buffer overflows [Chen et al. 2009; Huang et al. 2012; Cooprider et al. 2007]. The nodes may *crash* because of environmental factors or exhausted batteries [Werner-Allen et al. 2006; Beutel et al. 2009; Barrenetxea et al. 2008; Hnat et al. 2011]. Moreover, due to the limited power that can be invested in radio transmissions, wireless links are *prone to failure* due to, for example, external interference, environmental noise, and collisions on the wireless channel [Baccour et al. 2012; Srinivasan et al. 2010].

To achieve better scalability and energy efficiency, the design of WSN protocols often favors *localized interactions* [Estrin et al. 1999]. This entails that a protocol's operation is mainly based on interactions among nodes within direct radio range, in what is known as a node's *neighborhood*. As example, Figure 1 illustrates the operation of a simplified data collection protocol. These protocols support many-to-one traffic by building a tree-shaped routing topology towards a sink node. The paths are determined based on a routing metric that every node periodically advertises to its neighbors. In this example, we consider for simplicity the number of hops to the sink as the routing metric. In Figure 1(a), every node chooses its next hop to minimize the hop distance to the sink.

Author's addresses: A. Jhumka, Department of Computer Science, University of Warwick, UK; L. Mottola, Department of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy and SICS Swedish ICT.

DOI : http://dx.doi.org/10.1145/0000000.0000000



(a) Before a link failure and a data corruption.

(b) After a link failure and a data corruption occur.

Fig. 1. Example operation of a data collection protocol with node A as sink. Dashed lines are bidirectional links. Thick lines are links chosen for routing. Numbers indicate the perceived hop distance from A.

WSN protocols must effectively deal with the possible faults to maintain good performance. If the link between node B and E fails in Figure 1(a), the protocol must recognize the topology change and reconfigure the paths accordingly. However, if the information on a node's neighborhood is incorrectly represented in the program's state, the protocol operation may be misguided. For example, say that data advertised by node C is corrupted at node E and in a way that C appears unreachable from E. If the protocol is unable to recognize the data corruption, the routes may be eventually sub-optimally reconfigured as in Figure 1(b).

As we elaborate in Section 2, erroneous neighborhood information may generate severe problems in data collection protocols, such as routing loops [Iova et al. 2013; Gnawali et al. 2009; Romer and Ma 2009; Keller et al. 2012; Burri et al. 2007]. These protocols are not the only example: neighborhood information are at the core of most WSN protocols. For example, incorrect information on a node's neighbors may yield systematic packet collisions in MAC protocols [Saifullah et al. 2010; Saifullah et al. 2011; Demirkol et al. 2006; Song et al. 2009; Rhee et al. 2008]; it may generate poor performance when using in-network processing algorithms due to sub-optimal allocation of processing functions [Bonfils and Bonnet 2003]; and tends to cause drastic inaccuracies in localization protocols [Langendoen and Reijers 2003].

Unfortunately, as we also discuss in Section 2, the design of WSN protocols often overlooks these issues, adopting fairly simplistic techniques to manage neighborhood information. Some form of un-coordinated beaconing from every node is typically used to signal the reachability of a device [Ko et al. 2011; Hansen et al. 2011; Levis et al. 2004; Dunkels et al. 2011; Gnawali et al. 2009]. These techniques show several weaknesses: *i)* they induce long delays before the nodes possibly acquire the correct topology information, *ii)* they cannot detect data corruption affecting neighborhood information, and *iii)* they are unable to distinguish between different types of faults, such as data corruption as opposed to node crashes or link failures, which may require distinct corrective actions.

Contribution and road-map. To remedy the issues above, we must ensure that the *physical* neighborhood of a node—defined in terms of the underlying physical connectivity—is accurately reflected in that node's *logical* states, that is, in the program that governs a protocol's operation. To achieve this, every time a fault occurs, nodes must quickly and accurately identify a "consistent view" on their 1-hop neighbors. Consistency here intuitively indicates that:

> Given any two nodes n and m, all nodes within a single hop from both n and m must always appear in both n's and m's logical states, namely, any 2-hop neighbors must agree on their shared neighborhoods.

To the best of our knowledge, we are the first to recognize this problem—which we call *neighborhood view consistency (NVC)*—and to factor it out the existing WSN literature. NVC indeed represents a primitive notion at the core of many WSN protocols. We claim that efficiently tackling NVC can benefit a vast fraction of such protocols, and that one can embed algorithms providing NVC

within existing protocols to improve their performance. As an example, as soon as neighborhood consistency is achieved at every node in Figure 1(b), node C would correctly appear in node E's logical state. Hence, node E would realize C is a better next hop than D.

In this paper, we investigate NVC from both a theoretical and a system perspective. In Section 3, we define the necessary conceptual framework. Then, we provide the following contributions:

— We formally define in Section 4 the NVC problem in terms of safety and liveness, and prove that the problem is impossible to solve using traditional *failure detectors* [Chandra and Toueg 1996] combined with *localized interactions* in an *asynchronous* system. These results lay the basis for the remainder of the work.
— Next, we split the problem along two dimensions: *i)* neighborhood monitoring, and *ii)* view consistency enforcement:
  — To solve *i)*, Section 5 presents a (software) device stronger than traditional failure detectors, which we call *pseudocrash failure detector* (PCD). We study its properties, and prove that they can only be *eventually* guaranteed in the general case. We provide an algorithm implementing such PCD, and show that PCDs are in general necessary, but not sufficient to provide NVC: a *synchronous* system is also required.
  — To solve *ii)*, Section 6 investigates how to enforce NVC based on the output of PCDs, proving again that NVC is impossible with the feasible PCDs. We thus define two weaker specifications of NVC and: *i)* demonstrate that none of them can be solved with a localized algorithm if unidirectional links are present, and *ii)* provide a global algorithm that solves both specifications in the presence of unidirectional links, as well as a localized algorithm that solves the weakest specification when links are bidirectional.

The last algorithm, which we call *WeakC*, can replace the un-coordinated beaconing used in most WSN protocols based on localized interactions, which mostly operate over bidirectional links [Ko et al. 2011; Gnawali et al. 2009; Kim et al. 2007a; Mottola and Picco 2011; Voigt and Österlind 2008; Burri et al. 2007]. Compared to existing solutions to manage neighborhood information, upon the occurrence of faults, *WeakC* can reconcile the neighborhood information much earlier, reducing the time a protocol operates in the absence of NVC. Moreover, it can do so in a consistent manner across all involved nodes, easing a protocol's reconfiguration in that the nodes' neighborhood are readily sound compared to each other. Finally, *WeakC* separates data corruption from node crashes or link failures, giving protocols a chance to react differently depending on the fault.

As described in Section 7, we implement *WeakC* atop two different WSN operating systems. Our implementations are designed to be sufficiently flexible to replace the neighborhood management component in many WSN protocols with only limited memory overhead. To demonstrate this as well as the benefits of NVC, we integrate our *WeakC* implementation with four different WSN protocols and run extensive micro-benchmarks and full-stack experiments in a real-world 90-node testbed, as reported in Section 8. The results indicate that: *i) WeakC* efficiently deals with different types of faults, and *ii)* NVC significantly improves the performance of existing WSN protocols. For example, NVC allows the Collection Tree Protocol (CTP) [Gnawali et al. 2009] to deliver more data by *halving* the energy consumption.

We end the paper by surveying the existing related literature in Section 9, and with brief concluding remarks in Section 10.

2. MOTIVATION AND STATE OF THE ART

Ample evidence exists of the role that efficiently and correctly detecting a node's neighbors plays in WSNs. Despite the advancements achieved on the *efficiency* side, however, solutions to ensure the *correctness* of these information are much less developed. Existing approaches tackle the problem at the system level, yet theoretical analysis of the problem are also largely missing.

Applications. The literature includes plenty of experiences from WSN deployments that demonstrate the issues in managing neighborhood information against possible faults. Since early anec-

dotal evidence [Langendoen et al. 2006], these experiences ultimately resulted in entire works reporting on the corresponding lessons learned, both in outdoor [Barrenetxea et al. 2008] and residential [Hnat et al. 2011] settings. For example, Barrenetxea et al. [2008] describe their experiences in *seven* environmental monitoring deployments, each lasting several months, observing that an overhearing strategy to maintain the nodes' neighbor tables is efficient in terms of energy consumption, as it does not require proactively transmitting beacons. However, by doing so, the representation of the physical topology in a node's state may lag behind topology changes, yielding reliability problems [Barrenetxea et al. 2008].

Additional evidence from real-world WSN installations comes from Dawson-Haggerty et al. [2012]. Based on data collected from a 500 node indoor deployment of power metering devices over one year, they observe that the vast majority of links are intermittent even in such a benign environment. Using a form of periodic un-coordinated beaconing, a node's neighborhood information is thus seldom on par with the underlying physical topology. Dawson-Haggerty et al. [2012] observe that this may cause an increase in routing stretch *by a factor of two*, that is, twice the number of messages are generated to reach the destination. This ultimately impacts the system's scalability.

Arora et al. [2004] as well as Dutta et al. [2006] report similar observations, cast in the operation of target tracking applications employing 500+ nodes. To offer sufficient accuracy in localizing the target, the nodes must be precisely aware of their neighboring relations; otherwise, targets may be represented as duplicates or false positives may be reported. To deal with inconsistent neighborhood information, Arora et al. [2004] define a dedicated notion of logical neighbor based on the long-term stability of wireless links. Even if this solves the specific problem they face in the given application, it also results in ruling out many links that may be usefully employed if a dedicated solution to manage neighborhoods against faults would be deployed.

In the volcano expedition of Werner-Allen et al. [2006], for about 37% of the time *at least one node out of nineteen* was reported as crashed, and often multiple nodes were not responding. This created continuous changes in the neighborhood information. In the same deployment, due to the complexity of the software running on the nodes and the bugs therein, data corruptions occurred frequently, hampering the operation of the system until a bug was fixed. The bug probably affected neighborhood information as well, as the operation of the data collection protocol, in turn heavily based on neighborhood information, was significantly impaired [Werner-Allen et al. 2006].

We also have first-hand experience of the impact of incorrectly maintaining neighborhood information. To obtain a quantitative insight, we analyze the logs available from *five* distinct deployments we carried out in past research efforts [Ceriotti et al. 2009; Mottola et al. 2010; Ceriotti et al. 2011]. Note that we did *not* obtain these logs explicitly for studying the NVC problem; thus, they only include partial information compared to our goals in this paper and the following figures are probably quite optimistic[1]. In the Torre Aquila deployment [Ceriotti et al. 2009], we find out that about 43% of the packet lost on the way to the sink could be definitely traced back to nodes reasoning upon inconsistent neighborhood information. The same figure amounts to about 41% for our road tunnel deployments [Mottola et al. 2010; Ceriotti et al. 2011] and to about 40% for our vineyard deployment [Mottola et al. 2010].

As technology evolves, we expect these issues to keep existing and possibly to worsen. For example, as energy harvesting and wireless energy transfer find their way in WSNs [Bhatti and Mottola 2016; Sudevalayam and Kulkarni 2011; K. et al. 2015], the operating modes will drastically mutate. Nodes will enter some form of deep hibernation to survive periods of energy unavailability, and will later *resume* the previous state—as opposed to restarting from scratch—as soon as ambient energy is newly available [Ransford et al. 2011]. This will require revisiting many assumptions about discovering and communicating with neighbors [Pannuto et al. 2014]. Note that hibernating a node is, in fact, analogous to a node crashing and later recovering. Maintaining consistent neighborhood information in this setting will present similar issues as those we tackle here.

[1]The deployments are unfortunately no longer accessible for further experimentation.

Systems. Neighborhood information are at the core of many system functionality. A plethora of works exist in this area, which however focus on achieving efficiency rather than correctness.

Besides the four protocols [Mottola and Picco 2011; Gnawali et al. 2009; Ko et al. 2011; Voigt and Österlind 2008] we employ in Section 8, many other examples exist whose functioning relies upon an accurate logical representation of a node's physical neighborhood [Burri et al. 2007; Kim et al. 2007a; Iova et al. 2013; Ko et al. 2011; Schmid et al. 2010; Choi et al. 2009; Saifullah et al. 2010; Saifullah et al. 2011]. One example is that of IP-enabled low-power wireless protocols, expected to provide the communication backbone for the emerging "Internet of Things". Multiple works generally observe that the efficiency of RPL-based networks is also a function of the "coherence" of neighborhood information across nodes [Ko et al. 2011; Iova et al. 2013]. Another example are the issues in computing the packet schedules in WirelessHART networks, due to conciliating different neighborhood information at different nodes [Saifullah et al. 2010; Saifullah et al. 2011]. Some works try and lessen the role of neighborhood information in the protocol operation; for example, by employing opportunistic transmissions schemes [Duquennoy et al. 2013], by avoiding the use of beacons for neighbor discovery [Puccinelli et al. 2012], or by employing multi-hop network-wide flooding as the only communication primitive [Ferrari et al. 2012]. We aim at formally and practically solving the NVC problem to improve the performance of protocols relying on this information.

Consistent neighborhood information is instrumental not just to the operation of networking protocols, but also to underpin the operation of higher-level programming systems and applications. Whitehouse et al. [2004] design a neighborhood abstraction to replicate data between 1-hop nodes, whereas Costa et al. [2007] build a shared memory space across neighboring nodes. As node crash, link fail, and data corruption occur, these programming systems need to be aware of these changes, so that application programmers are minimally affected. Cases also exist where maintaining neighborhood information is the application itself. Examples are mechanisms for quickly computing the neighborhoods' cardinality [Cattani et al. 2014], and neighbor discovery protocols for mobile WSNs; for example, used in wildlife monitoring scenarios [Pásztor et al. 2010]. In the latter, several solutions exist [Dutta and Culler 2008; Kandhalu et al. 2010; Zhang et al. 2012] that allow designers to use radios as proximity sensors.

Two other research lines provide further evidence of the importance of the problem. On one hand, the design of WSN monitoring and debugging tools [Romer and Ma 2009; Keller et al. 2012] often centers on the ability to inspect the neighborhood information at a node. Reasoning upon this information is regarded as a fundamental part of the testing process, as many bugs are ultimately a function of how neighborhood information is acquired and maintained [Romer and Ma 2009; Keller et al. 2012]. On the other hand, owing to the energy cost of managing multiple beaconing processes of different protocols, software architectures exist to reduce such overhead [Dunkels et al. 2011; Hansen et al. 2011]. Here again, the focus is on efficiency, especially in terms of energy consumption, rather than on the correctness of the neighborhood information.

Neighborhood view consistency. We argue that applying a notion of NVC in many of these systems would provide significant benefits. Many of them employ network functionality that closely resembles the four protocols we consider in Section 8. For these, we quantitatively demonstrate remarkable performance improvements, for example, in data yield and energy consumption. The cost to gain these benefits using our implementations is a small increase in data and program memory due to replacing the neighborhood management component with an implementation of *WeakC*, along with a limited network overhead. Importantly, the latter is only required whenever NVC must be re-established: in the absence of faults, *WeakC* causes no additional traffic.

On the other hand, we also maintain that many of the systems we surveyed did not originally employ a notion of NVC essentially because the problem was insufficiently understood. Especially in real-world deployments, obtaining reasonable—albeit greatly sub-optimal—performance is often deemed sufficient even when room for improvements is clearly available [Barrenetxea et al. 2008; Hnat et al. 2011]. Formally fixing the notion of NVC, as we do in Section 4, provides the necessary conceptual foundations. Moreover, splitting the problem between neighborhood monitoring and

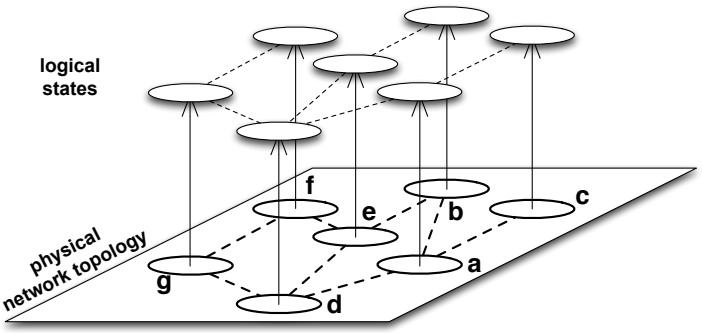


Fig. 2. Intuitive representation of physical and logical neighborhoods. Thick dashed lines represent links in the physical topology. Thin dashed lines represent how those links are reflected in the nodes' logical states. The physical neighborhood of node a is $A = \{b, c, d\}$. However, due to faults, $A^l = \{c, d\} \neq A$. Therefore, WSN protocols running at a would think they cannot communicate with b, although this is physically possible.

view consistency enforcement, which we discuss separately in Section 5 and Section 6, offers the opportunity to employ selected parts of our work, depending on the specific application setting.

Even though our work focuses on WSNs, worth noticing is that the problem of managing neighborhood information, and thus possibly a notion of NVC, is not only germane to WSNs. In mobile ad-hoc networks, for example, several protocols operate based on information about the 1-hop nodes [Cornejo et al. 2014]. Specific solutions thus exist to ensure that these information is efficiently collected despite mobility [Iyer et al. 2011; Cornejo et al. 2014].

3. PRELIMINARIES

We provide the necessary formal background, including the models we use, the program syntax and semantics, the computation and communication models, as well as the types of failures we consider.

3.1. System Model

Topology and processes. A WSN node is a computing device associated to a unique identifier. Communication in WSNs is typically modeled with a circular communication range centered on a 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, communication between two nodes may be temporarily disrupted by a number of factors; for example, interference from co-located wireless networks and environmental noise [Baccour et al. 2012; Srinivasan et al. 2010]. Hence, the network topology changes with time.

In graph-theoretic terms, we represent a WSN as a *directed* graph $G = (V, A)$ with a set V of vertices representing the nodes, and a set A of arcs representing the directed communication links between pairs of nodes. We denote by γ the number of nodes in the network, that is, $\gamma = |V|$. To model network dynamics, we denote the topology of the network at a time t by $G_t = (V_t, A_t)$. We assume, for simplicity, that all links are initially bidirectional, that is, the directed graph is initially symmetric. We also assume that the network remains connected throughout, otherwise enforcing NVC becomes impossible in the general case.

Localized interactions in WSN protocols require accurate physical neighborhood information. On the other hand, a program running on a node only keeps—in its *logical* states—information about its "perception" of the physical neighborhood. We call this the program's *logical* neighborhood. We indicate as N_t^l node n's logical neighborhood at time t, that is, the set of nodes that the program running at n believes are neighbors of n at time t. Differently, for a given node n, we denote its *physical* neighborhood at time t by N_t, namely, N_t is the set of nodes reachable from n with a

single transmission at time t, independently of whether they appear in n's logical states. We will simply denote the physical (logical) neighborhood of a node n by N (N^l) when time is immaterial.

The physical and logical neighborhoods of a node may differ. Figure 2 intuitively represents the concept. Node a's physical neighborhood is $A = \{b, c, d\}$. However, because of different types of faults, this information may be incorrectly reflected in node a's logical states. For example, data corruption may make node c appear as not reachable from a in a single transmission, hence the logical neighborhood at a may differ from the physical one: $A^l = \{c, d\} \neq A$. This is, in essence, the source of the NVC problem we study.

The actions taken by the individual nodes are dictated by the *process* running on it. The system thus consists of a finite set Π of $\gamma > 0$ processes $p_1 \dots p_\gamma$, where each node in the network runs a process[2]. Adjacent processes, defined by the physical topology, are linked by unreliable wireless channels, where a finite number of messages may be unpredictably lost. No spurious messages are delivered. Each process contains a non-empty set of variables and actions, also called steps, depending on an algorithm $\mathcal{A}$. We denote a variable v of process p by $p.v$. An assignment of values to variables in a program is called a *state*.

Program syntax and semantics. We write programs in the guarded command notation [Dijkstra 1974]. Hence, an action has the form $\langle name \rangle :: \langle guard \rangle \rightarrow \langle command \rangle$. In general, a *guard* is a predicate defined over the set of a process' variables. When a *guard* evaluates to true, the *command* can be executed, which takes the program from one state to another. When the state transition is complete, we say that event $\langle name \rangle$ has occurred. A *command* is a sequence of assignments and branching statements. 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* and *term* are Boolean constructs. When a guard evaluates to true in a state, the corresponding action is *enabled* in that state. A special **timeout**($timer$) guard evaluates to true when a *timer* variable reaches zero. A **set**($timer$, $value$) command sets the timer variable to a specified value.

We choose this notation for several reasons. First, it is usually simpler to formally reason on a program execution in terms of what guards become true at a given point in the execution, rather than following a specific control flow. Moreover, the guarded command notation makes programs more compact, compared to the more traditional state transition or procedural representations. Finally, the guarded command notation matches the programming style of many WSN software platforms, which tends to be event driven [Hill et al. 2000]. In these cases, the binding of events to their handlers is, in a sense, corresponding to the evaluation of guards.

The execution of a step of an algorithm $\mathcal{A}$ causes the process to update one or more variables and moves the system from one state to another in one atomic step. In a given state s, several processes may be enabled, and a decision is needed about which one(s) to execute. The subset of processes that take a step when possible is chosen according to different scheduling policies. To ensure the system makes progress, a notion of fairness is also required. Whenever any of the enabled processes can take a step independent of all others, that is, a continuously enabled action is eventually executed, we say the system is weakly fair [Dolev 2000] and runs in an *asynchronous* manner. This entails there is *no bound* on relative process speeds and message transmission time. Differently, we say the system is *synchronous* whenever *all* enabled processes take a step. This captures the fact that processes execute in lock-step, thus, process speeds and message latency are *bounded* and *known*. The notion of weak fairness never extends to faults.

Communication. Each process has a special *channel* variable, denoted by *ch*, modeling a FIFO queue of incoming messages sent by other processes. This variable is defined over the set of (possibly infinite) message sequences. An action with a **rcv**(msg,$sender$) guard is enabled when there is a message at the head of the channel variable ch of a process. Executing the corresponding action causes the message at the head of the channel to be dequeued, while msg and $sender$ are bound to the content of the message and the sender identifier. Differently, the **send**(msg,$dest$) command

[2]We will use the terms node and process interchangeably where no ambiguity can arise.

causes message msg to be attached to the tail of the channel variable ch of processes in the $dest$ set. To capture the broadcast multi-hop nature of WSNs, the semantics of **send** when executed on node n depends on the processes in $dest$:

— if all nodes in $dest$ are in the physical neighborhood N of node n, that is, $\forall i \in dest : i \in N$, then msg is simultaneously appended to the tail of the channel variable 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: this implies the message reaches processes that may not appear in the sender's logical states, modelling the semantics of physical broadcast in WSNs;
— 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, modeling multi-hop transmissions.

3.2. Faults

We consider three types of faults: *i)* process crash faults, *ii)* omission faults, and *iii)* transient faults. These are schematically shown in Figure 3 together with their *symptoms*, that is, the way they manifest from a program's perspective.

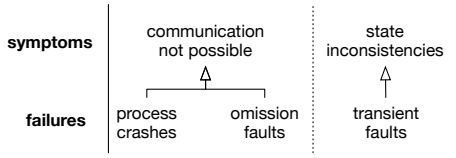


Fig. 3. Types of failures and their symptoms.

A process crash occurs when a process stops executing, for example, because a node runs out of energy or is physically damaged [Werner-Allen et al. 2006; Beutel et al. 2009; Barrenetxea et al. 2008; Hnat et al. 2011], whereas an omission fault occurs when a node sends a message that fails to be delivered to at least one of the intended recipients. This occurs due to, for example, interference and message collisions [Baccour et al. 2012; Srinivasan et al. 2010]. Both types of failures manifest with the same symptom: communication is prevented from one node to another. These failures are thus conceptually similar, and so is their impact in determining the conditions under which we can provide NVC. In the rest of the paper, we say a node n *pseudocrashes* if n crashes or n undergoes an omission fault.

Transient faults corrupt a process state by arbitrarily altering values of variables. They model memory corruption due to, for example, bit-flips caused by defective hardware [Finne et al. 2008; Werner-Allen et al. 2006] or software bugs such as buffer overflows [Chen et al. 2009; Cooprider et al. 2007; Huang et al. 2012; McCartney and Sridhar 2006]. Due to the lack of memory protection in most WSN operating systems, the latter are particularly frequent [Cooprider et al. 2007]. Transient faults may also affect messages in transit. We assume that transient faults do not occur infinitely often, otherwise the system's liveness may be compromised. Transient faults manifest as state inconsistencies in a program, that is, a given state in a program cannot be the result of any execution of an algorithm $\mathcal{A}$. For example, a state inconsistency may be due to conflicting values in different variables, locally to a process or across different processes. As discussed next, transient faults bear a unique effect on the conditions leading to solvable instances of NVC.

Worth noticing is that we do not make any assumption on the spatial or temporal correlation of faults. The theoretical results and algorithms we discuss next do not rely on any such premise, neither do the implementations we describe in Section 7, used to obtain the results in Section 8.

Pseudocrash failure patterns. To model pseudocrashes occurring during system execution, we define a failure pattern $\mathcal{F}$ to be a function from $\mathcal{T}$ to 2^{Π}, where $\mathcal{T}$ represents the range of output from a fictional global clock[3]. Intuitively, $\mathcal{F}(t)$ denotes the set of nodes that are pseudocrashed at time t. We say a node n is *working* at time t if $n \notin \mathcal{F}(t)$ and we say n is *not working* at time t if $n \in \mathcal{F}(t)$. Hence, a node n *pseudocrashes* at time t if n is working at time $t' < t$ and not working at time t. Dually, we say that a node n *recovers* at time t if n is not working at time $t' < t$ and n

[3]The notion of a global clock simply serves to assign a timestamp to the output of a failure pattern, yet there is no actual global clock in the system. This is simply a tool to simplify the presentation without compromising its formal coherence.

is working at time t. We also consider a function $\mathcal{F}'$ from $\mathcal{T}$ to bag Π that denotes the number of times the nodes have pseudocrashed at time t, or how many pseudocrashes occurred in the network until t. Thus, a node n can be in any of these states in a failure pattern $\mathcal{F}$:

— Always working: node n never pseudocrashes in $\mathcal{F}$.
— Eventually working: node n pseudocrashes at least once, but a time t exists when n recovers in $\mathcal{F}$.
— Eventually not working: node n permanently pseudocrashes in $\mathcal{F}$ after a given time t.

When a node is eventually working, we do not require the node to work permanently after time t. Rather, we require that it keeps working for a long enough time to allow progress to be made. A pseudocrash failure can last for an arbitrary length of time. However, if a node alternates between working and not working infinitely often, in what is an "unstable" state, then progress might be compromised. Thus, to ensure liveness, we require the duration of a pseudocrash to be of a certain minimum length. This means that the time between a node pseudocrashing and recovering is lower bounded. In this model, a node crash is a pseudocrash with infinite duration.

We also define two sets including the processes that are always working in $\mathcal{F}$ and those that are eventually not working in $\mathcal{F}$, namely:

$$workingA(\mathcal{F}) \equiv \{n \mid \forall t \geq 0 : n \notin \mathcal{F}(t)\}$$
$$notWorkingE(\mathcal{F}) \equiv \{n \mid \exists t. \forall t' \geq t : n \in \mathcal{F}(t')\}$$

Failure detectors. A failure detector is a (software) device responsible for the detection of node crashes in a distributed system [Chandra and Toueg 1996]. A failure detector can be queried at any time $t \in \mathcal{T}$. It returns the set of processes it *suspects* to have crashed at a time t. A failure detector history H is a mapping from $\Pi \times \mathcal{T}$ to 2^{Π}, where $H(m,t)$ denotes the value of the failure detector for process m at time t. In other means, if the failure detector at process m is queried at time t, then $H(m,t)$ contains the set of processes that m suspects to have crashed at time t. A failure detector $\mathcal{D}$ maps a failure pattern $\mathcal{F}$ to a set of failure detector histories.

However, not all possible failure detector histories are useful or accurate. For example, failure detector histories where node crashes are *not* detected are not accurate. In fact, there is not just one type of failure detector, but many possible types depending on what properties they provide, that is, depending on their strength. To reason on how useful a failure detector history is, failure detectors are required to satisfy certain properties [Chandra and Toueg 1996]. The *completeness* property captures the ability to detect crashes. On the other hand, the *accuracy* property captures the ability to avoid wrong suspicions. Due to the uncertainty in an asynchronous system, these properties cannot be taken for granted, giving rise to different degrees of completeness and accuracy. For example, in a WSN, uncertainty can arise due to wireless interference. In such a case, a failure detector may report that a node crashed when it is actually correct. Extending failure detectors to pseudocrashes, we define *strong* accuracy and *strong* completeness as follows:

— Strong accuracy: no process is suspected before it pseudocrashes. Specifically,
$\forall \mathcal{F}. \forall H \in \mathcal{D}(\mathcal{F}). \forall t \in \mathcal{T}. \forall m, n \in \Pi \setminus \mathcal{F}(t). m \notin H(n,t)$
— Strong completeness: a pseudocrashed process is eventually permanently suspected by all correct neighbor processes. Specifically,
$\forall \mathcal{F}. \forall H \in \mathcal{D}(\mathcal{F}). \exists t \in \mathcal{T}. \forall m \in notWorkingE(\mathcal{F}). \forall n \in workingA(\mathcal{F}), m \in N.$
$\forall t' \geq t. m \in H(n,t')$

These properties characterize the strength of the completeness and accuracy properties of a specific failure detector. The strong specification of these properties also entail that it is the strongest formulation achievable, that is, a failure detector cannot offer a completeness (accuracy) property stronger than strong completeness (strong accuracy). A failure detector that provides both strong accuracy and strong completeness is called *perfect* failure detector. A perfect failure detector permanently suspects a pseudocrashed node until it recovers.

3.3. Algorithms and Computation

Chandra and Toueg [1996] define a computation to be a tuple $C = (\mathcal{F}, \mathcal{D}, I, S, T)$ where $\mathcal{F}$ is a failure pattern, $\mathcal{D}$ a failure detector, I is the system's initial state, S is a sequence of algorithm steps, and T is a sequence of increasing time values when these algorithm steps are taken. In this paper, we study algorithms that use devices similar to failure detectors. However, our definition of computation will be slightly different [Gärtner and Pleisch 2002], but equivalent to that of Chandra and Toueg [1996]. We define two functions: a step function A_s from $\mathcal{T}$ to the set of all algorithm steps, and a process function A_p from $\mathcal{T}$ to Π. In other words, function $A_p(t)$ denotes the process that takes a step at time t and $A_s(t)$ identifies the step that was taken.

To account for the possibility of transient faults, we augment our notion of computation with a special process called *environment* and denoted by ϵ, which causes a further set F_{tr} of steps to become possible. The actions in F_{tr} model transient failures [Arora and Kulkarni 1998]. Without loss of generality, we assume that at any time, at most one process, including the environment, takes a step. If no process takes a step at time t, both the step function and the process function evaluate to $\perp$. A computation in the presence of transient faults is thus $C = (\mathcal{F}, \mathcal{D}, I, A_s \cup F_{tr}, A_p \cup \{\epsilon\})$.

A *specification* is a set of computations. A program P satisfies a specification ¶ if every computation of P is in ¶; Alpern and Schneider [1985] state that every specification can be described as the conjunction of a safety and liveness property. Intuitively, safety states that something bad should not happen; 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 that every computation should include.

4. NEIGHBORHOOD VIEW CONSISTENCY

As intuitively presented in Figure 2, the NVC problem arises when the physical neighborhood is incorrectly or partially reflected in the nodes' logical states. To solve this problem, the system must detect what type of fault happens when. In this section, we show that this is in general impossible to achieve using any of Chandra and Toueg's failure detectors [Chandra and Toueg 1996].

First, we define how a node removes from its logical neighborhood the devices it suspects to have pseudocrashed.

Definition 4.1 (*Remove*). Consider a physical topology $G = (V, A)$, a node $n \in V$ and a logical neighborhood N^l of n. We say that node n *removes* a node $q \in N^l$ if $N'^l = N^l \setminus \{q\}$ where N'^l represents the updated value of N^l.

Next, we define the notion of localized algorithm, which is often only informally described in the existing literature [Estrin et al. 1999]. Intuitively, an algorithm is said to be *local* whenever its input data reside at nodes within some bounded hop distance from each other. For example, if an algorithm running at a given node definitely employs information gathered at most within the 2-hop neighborhood, then the algorithm is local. Differently, the algorithm is *global* whenever such bound cannot be determined and, in principle, the algorithm may make use of data residing at any node in the system, independent of the hop distance. We formally fix this notion as follows.

Definition 4.2 (*Localized algorithm*). Given a topology $G = (V, A)$, problem specification ¶ for G, and an algorithm $\mathcal{A}$ that solves ¶ in G, algorithm $\mathcal{A}$ is said to be local if the complexity of $\mathcal{A}$ varies with the size of an n-hop neighborhood; $\mathcal{A}$ is global otherwise.

With these definitions, we formally define the NVC problem, which we call *strong view consistency* in its most general formulation.

Definition 4.3 (*Strong view consistency*). Given a network $G = (V, A)$, and two nodes $n, m \in V$, a program provides *strong NVC* for G if every computation satisfies:

— (Safety): A working node is never removed.

— (Liveness): Every time a node m pseudocrashes, then eventually $\forall n : m \in N : n$ removes m.

The liveness property of strong NVC states that every time a node m pseudocrashes, every other node that lists m in its logical neighborhood eventually removes it. Differently, the safety specification rules out nodes mistakenly removing working nodes from their logical neighborhoods. The eventual removal of the liveness property basically means that the removal does not need to be instant, but because of the safety aspect, it certainly needs to happen before a possible recovery. Otherwise, a working node would be removed, violating the safety property.

This specification formally fixes the intuitive formulation of consistency given in Section 1, including the mutual agreement of 2-hop neighbors on their shared neighborhoods. This notion applies to arbitrary networks by ensuring that Definition 4.3 applies on every possibly 2-hop slice of the network, independent of the overall depth. Note that strong NVC represents an "ideal" situation, in that an algorithm providing NVC—provided one exists—would be able to perfectly reflect the physical topology in the nodes' logical states. Worth considering is also that the notion of strong NVC as specified in Definition 4.3 is *memoryless*, that is, it does not depend on past occurrences of failures. The specification only advocates the removal of pseudocrashed nodes from the logical state of the program whenever such faults occur. From this point of view, the cases of a new node joining the system and of an existing node recovering are analogous.

We initially investigate the problem of NVC in an asynchronous system. We prove that, in this setting, it is impossible to provide strong NVC with any of Chandra and Toueg's failure detectors [Chandra and Toueg 1996]. The intuition behind this result is that, for an algorithm solving strong NVC to exist, every node must be able to remove a pseudocrashed neighbor before it recovers, that is, the removal needs to occur within a given upper bound. Therefore, whenever a node pseudocrashes and then recovers, every neighbor of the pseudocrashed node must remove it before it recovers, or removing the neighbor after it recovers would entail the removal of a working node, violating the safety specification. However, assuming an upper bound for a process' action implies considering a synchronous system.

THEOREM 4.4 (IMPOSSIBILITY OF NVC WITH FAILURE DETECTORS). *There exists no algorithm that provides strong NVC with any failure detector $\mathcal{D}$ in an asynchronous system.*

PROOF. We prove this by contradiction: we assume that there exists an algorithm $\mathcal{A}$ that solves strong NVC with some failure detector $\mathcal{D}$ in an asynchronous system, and show a contradiction.

Consider first a failure pattern $\mathcal{F}_n$ where no pseudocrash occurs. Now, consider a run $R_n = (\mathcal{F}_n, \mathcal{D}(\mathcal{F}_n), I, S_s, S_p)$ of $\mathcal{A}$. Since $\mathcal{A}$ solves strong NVC, at every time instant, all nodes will output $\emptyset$. Because $\mathcal{A}$ is correct and no pseudocrash occurs, all nodes remove no nodes.

Now, consider a failure pattern $\mathcal{F}_1$ where a node n pseudocrashes at time t and recovers at time $t' > t$. Now, consider a run $R_1 = (\mathcal{F}_1, \mathcal{D}(\mathcal{F}_1), I, S_s, S_p)$ of $\mathcal{A}$ with same step function S_s and process function S_p as R_n. Hence, R_1 and R_n are indistinguishable. Since $\mathcal{A}$ solves strong NVC, then in R_1 all neighbor nodes of n must have removed n by t' and must output $\{n\}$. However, if the neighbor nodes of n do not remove n by t', then they cannot ever do so. Indeed, after t', node n is working again and algorithm $\mathcal{A}$, correctly solving strong NVC, cannot remove a working node due to the safety property.

This leads to a contradiction, because requiring that all neighbor nodes of n to remove n by t' entails the system is synchronous, contradicting the initial conjecture. □

This proof, as well as several others in the following, uses the standard technique of proof by contradiction to show impossibility results, as explained by Lynch [1996]. In such type of proofs, it is shown that if something; for example, an algorithm $\mathcal{A}$, exists, then two mutually contradictory instances—that is, two different runs of $\mathcal{A}$—would have the same result. This demonstrates that $\mathcal{A}$ cannot exist [Lynch 1996]. When applying this technique, the key aspect is the construction of the instances that show the contradiction.

5. PSEUDOCRASH FAILURE DETECTORS

Based on the fundamental result of Section 4, we split the problem along two dimensions: *i)* neighborhood monitoring and *ii)* view consistency enforcement. For neighborhood monitoring, the key issue is the detection of pseudocrashed nodes *and* of their 1-hop neighbors. Thus, we conceive a device strictly stronger than a perfect failure detector, called a *pseudocrash failure detector* (PCD). The PCD at a process j returns information on how j perceives its underlying physical neighborhood J and on the neighborhoods of every node in J, that is, j's 2-hop neighborhood. In contrast, a perfect failure detector would only return information on a per-node basis, that is, the PCD gives strictly more information than a perfect failure detector, as it also reaches further than the 1-hop neighborhood of j. The process at j uses this information to update the logical neighborhood J^l.

We formally define the concept of PCD, study its properties, and prove that, in the presence of transient faults, its properties can only be *eventually* guaranteed. We provide an actual algorithm implementing a PCD with eventual guarantees, and show that PCDs are in general *necessary*, but *not sufficient* to provide NVC: a *synchronous system* is also required. We tackle the view enforcement problem in Section 6.

5.1. Definitions

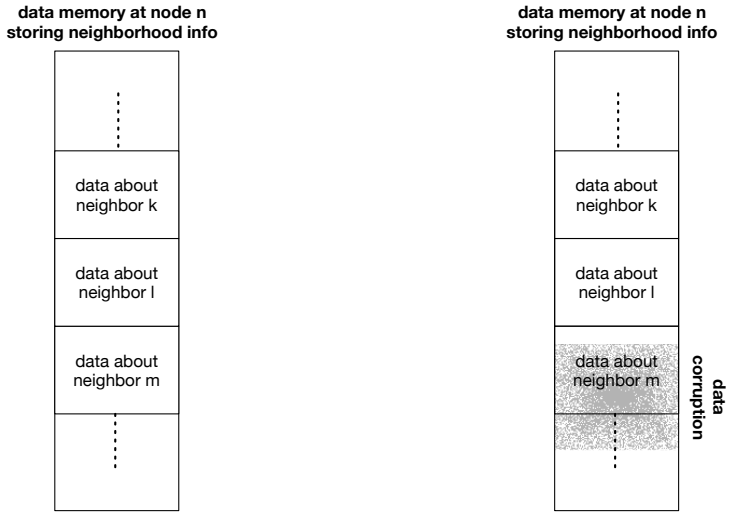
Each process has access to a *local* PCD device providing (possibly incorrect) information about the failure pattern that occurs in an execution. A process p can query its local PCD at any time. The local PCD returns a set of tuples $(n, \hat{N})$ called *suspects*, containing the set of processes the PCD at p suspects to have pseudocrashed at a given time, together with the *most recent (logical) neighborhood* $\hat{N}$ of each of the suspected processes. The most recent neighborhood $\hat{N}$ of a node n at a time t captures the most updated information n has about its neighborhood at time t', which equals the last known neighborhood at t' only if no node is removed in the meantime. In other words, for a node n at time t, $\hat{N}$ is equal to the original neighborhood of n except the nodes pseudocrashed at time t'. This notion is formally defined as follows.

Definition 5.1 (*Most recent neighborhood*). The most recent neighborhood mrn of a process n at time t, which returns the set $\hat{N}$, is

$$mrn(n,t) : \exists t' \leq t.((\hat{N} = N \setminus \mathcal{F}(t') \wedge \mathcal{F}(t') \neq \emptyset) \\ \wedge (\forall t'' : t' \leq t'' \leq t. \forall x \in \hat{N}. n \text{ does not remove } x)$$

In Definition 5.1, the set $\hat{N} = N \setminus \mathcal{F}(t')$ builds $\hat{N}$ by removing from N the nodes pseudocrashed at time t', as $\mathcal{F}(t') \neq \emptyset$ and so there is indeed at least one pseudocrashed node at time t'. Moreover, time t' is the last time that node n removes any neighbor from its logical states, because for any time t'' ($t' \leq t'' \leq t$), node n does not perform any remove action for any node x in $\hat{N}$. As a result, $mrn(n,t)$ is the most recent neighborhood for node n at time t.

Let $\mathcal{R}$ denote the set of all possible tuples that can be returned by a PCD. A PCD history H_{PCD} with range $\mathcal{R}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$, where $H_{PCD}(p,t)$ is the output value of the PCD of process p at time t. $H_{PCD}(p,t)$ denotes the set of processes that p suspects have pseudocrashed at time t, along with their neighborhoods, and thus captures how p's suspicions evolve over time. In short, we say that *p suspects q with a neighborhood Q at time t* if $(q, Q) \in H_{PCD}(p,t)$. Therefore, a PCD is a function that maps each failure pattern $\mathcal{F}$ to a set of PCD histories with range $\mathcal{R}_{PCD}$, where $\mathcal{R}_{PCD}$ denotes the range of output of the PCD. $PCD(\mathcal{F})$ denotes the set of possible PCD histories permitted by the PCD for the failure pattern $\mathcal{F}$.



(a) Before the fault. Node n stores, somewhere in its data memory, information about its 1-hop neighbor m.

(b) A data corruption occurs that affects the same memory segments that would change as a result of m's pseudocrash.

Fig. 4. Pictorial example of a masking transient fault. Node n and m are 1-hop neighbors.

5.2. PCD Properties

A PCD that can be used to solve the strong NVC problem would satisfy the following two properties, where $count(n, \mathcal{F}'(t))$ returns the number of times node n has crashed in $\mathcal{F}$ until t:

— (Strong accuracy) No working process n is incorrectly suspected to have pseudocrashed with neighborhood $\hat{N}$. Formally,
$\forall \mathcal{F}.\ \forall H_{PCD} \in PCD(\mathcal{F}).\ \forall t \in \mathcal{T}.\ \forall m \notin \mathcal{F}(t) : (n, \hat{N}) \in H_{PCD}(m, t) \Rightarrow \exists t', t' < t.\ n \in \mathcal{F}(t') \wedge \hat{N} = mrn(n, t') \wedge count(n, \mathcal{F}'(t')) = count(n, \mathcal{F}'(t))$

— (Strong completeness) Every time a process n pseudocrashes, it will eventually be suspected with some neighborhood $\hat{N}$. Formally,
$\forall \mathcal{F}.\ \forall H_{PCD} \in PCD(\mathcal{F}).\ \forall t \in \mathcal{T}.n \in \mathcal{F}(t) \Rightarrow\ \exists t', t \leq t'.\ \exists m \notin \mathcal{F}(t'), n \in M.(n, \hat{N}) \in H_{PCD}(m, t') \wedge \hat{N} \neq \emptyset \wedge count(n, \mathcal{F}'(t)) = count(n, \mathcal{F}'(t'))$

Intuitively, strong accuracy requires that no working node is wrongly suspected with a possibly wrong neighborhood, whereas strong completeness requires that every node that pseudocrashes is eventually suspected before its next pseudocrash, that is, every pseudocrash is suspected. We call a PCD that satisfies both properties a *perfect* PCD.

It is, however, impossible to implement such perfect PCD in the presence of transient faults. The intuitive reason for this is that *it is possible* to construct executions where transient faults may deceive a PCD to suspect a node even though that has not pseudocrashed. Dually, even if a PCD may correctly suspect some node with the correct neighborhood, transient faults can corrupt this information, creating a situation where it would appear the pseudocrash never occurred.

Figure 4 shows a graphical example. Node n and m are 1-hop neighbors; node n stores, somewhere in its memory, information about its neighbor m as shown in Figure 4(a). Consider m pseudocrashes due to the link between n and m failing. Now, node m should be removed from n's logical neighborhood. However, a data corruption happens that overlaps with the memory area where information about m is stored, as shown in Figure 4(b). This fault modifies data structures such that it

appears as if m is still there. This causes n to keep m rather than removing it. Thus, we say that a transient fault is "masking" the pseudocrash; we say transient faults are "non-masking" otherwise.

These situations are a result of a transient fault's ability to arbitrarily corrupt the state of a program, including ways that affect the same data that should be modified as a result of pseudocrashes. In practice, these occurrences represent a quite unlucky situation: a pseudocrash *and* a transient fault occur that impact *overlapping* data segments; plus, the transient fault "intelligently" corrupts the state in a way that the pseudocrash goes undetected. Intuitively, this should represent a quite rare situation. We analytically prove this statement in Appendix A.

Considering the definition of strong NVC, however, if masking transient faults may occur, the pseudocrashed node may never be suspected, as it happens in Figure 4, violating strong completeness of the perfect PCD. Hence, the following holds.

THEOREM 5.2 (IMPOSSIBILITY OF PERFECT PCD). *In the presence of transient faults, it is impossible to implement a perfect PCD.*

For reason of space, we refer the reader to an extended technical report for the detailed proof [Jhumka and Mottola 2014].

There may be different means to circumvent this result. A possible strategy is to consider guarantees that can be provided once transients faults stop occurring, called *eventual* guarantees. Thus, we weaken the definition of both strong accuracy and strong completeness to capture eventual properties. We denote by Σ such *eventually perfect* PCD:

— (Eventual strong accuracy) There is a time after which no working process n is incorrectly suspected with neighborhood $\hat{N}$. Formally,
$\forall \mathcal{F}.\ \forall H_\Sigma \in \Sigma(\mathcal{F}).\ \exists t' \in \tau.\ \forall t > t'.\ \forall m \notin \mathcal{F}(t) : (n, \hat{N}) \in H_\Sigma(m,t) \Rightarrow \exists t'', t' \leq t'' < t,\ n \in \mathcal{F}(t'') \wedge \hat{N} = mrn(n,t'') \wedge count(n, \mathcal{F}'(t'')) = count(n, \mathcal{F}'(t))$

— (Eventual strong completeness) There is a time after which, when a process n pseudocrashes, eventually n will be suspected with some neighborhood $\hat{N}$. Formally:
$\forall \mathcal{F}.\ \forall H_\Sigma \in \Sigma(\mathcal{F}).\ \exists t' \in \tau.\ \forall t \geq t'.n \in \mathcal{F}(t) \Rightarrow \exists t'', t \leq t''.\ \exists m \notin \mathcal{F}(t''), n \in M.(n, \hat{N}) \in H_\Sigma(m,t'') \wedge count(n, \mathcal{F}'(t)) = count(n, \mathcal{F}'(t''))$

It turns out that there exists a synchronous implementation of a PCD that is eventually perfect, shown in Figure 5. This property is relevant for WSNs because, in the absence of transient faults, the PCD is perfect. The key idea is to periodically exchange neighborhood information among 1-hop nodes so that all of them are eventually aware of their respective *2-hop* neighborhoods. Once transient faults stop happening, this information is sufficient to identify the suspected processes.

In action *dissem* of Figure 5 process j updates its *logical* neighborhood with the identifiers of processes it believes to be still working; then it advertises its identifier and its current *logical* neighborhood to the nodes in the *physical* neighborhood. Indeed, the **send** command with destination *BCAST* reaches also 1-hop neighbors that may be unknown to process j, as outlined in Section 3. Next, the algorithm sets a timeout Δ for the next round of advertisement and a shorter timeout Θ to process the received information. In action *compute*, process j collects the neighborhood information. When the *detect* timeout expires, process j computes the set of suspects as the neighbors it did not hear in the last round of advertisement, along with their logical neighborhoods.

Another possibility to circumvent the impossibility result of Theorem 5.2 is to consider a stronger fault setting. One such option is to only allow transient faults that *cannot* mask pseudocrashes, that is, we only consider non-masking faults. This rules out all situations akin to Figure 4. As demonstrated in Appendix A, these cases are actually the vast majority. Under non-masking transient failures, it can be proven that the PCD in Figure 5 satisfies strong completeness and eventual strong accuracy. Strong completeness is guaranteed because in the absence of masking transient faults, then all pseudocrashes are eventually detected, as there is no way for a transient fault to "hide" a pseudocrash. As a result, the PCD is strong complete. The eventual strong accuracy follows from the

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

% set of pseudocrashed nodes and their 1-hop neighborhood
suspects: set of (id, neighborhood) init $\emptyset$;

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

% timers for exchanging neighborhoods and detection, $\Theta < \Delta$
neighborhoods, detect: timer init Δ, Θ;

actions
dissem:: **timeout**(*neighborhoods*)$\rightarrow$
N[j], live, suspects := *live*, $\emptyset, \emptyset$;
send($\langle j, N[j] \rangle, BCAST$);
set($neighborhoods$,Δ);
set($detect$,Θ);

compute:: **rcv**($\langle p, P \rangle$,r) $\rightarrow$
live, N[p]:= *live* $\cup \{p\}, P$;

detect:: **timeout**(*detect*)$\rightarrow$
suspects := $\{(i, N[i]) | i \in N[j] \setminus live\}$

Fig. 5. A synchronous implementation of a PCD algorithm that is *i)* eventually perfect in the presence of transient faults, and *ii)* perfect in their absence. Moreover, this PCD satisfies strong completeness with non-masking transient faults.

fact that the PCD of Figure 5 is eventually perfect. Strong accuracy, however, may be prevented as a transient fault may still lead to wrong suspicions. The formal derivation of these results is available in an accompanying technical report [Jhumka and Mottola 2014].

5.3. Solving Strong NVC with PCDs

We now study necessary and sufficient conditions for solving strong NVC with PCDs. For the former, we prove that, if we can solve strong NVC, then we can implement a perfect PCD.

THEOREM 5.3 (PCDs NECESSARY FOR NVC). *A perfect PCD is necessary for strong NVC.*

PROOF. The strong accuracy property of the PCD follows from the safety property of the strong NVC problem (no node is wrongly removed) and from the fact that all neighbors eventually remove the pseudocrashed node (part of liveness). The strong completeness property of the perfect PCD derives from the liveness property of strong NVC. Hence, if strong NVC can be solved, we can also implement a perfect PCD. □

As a matter of fact, Figure 6 shows an algorithm that emulates the output of a perfect PCD, given the output of an algorithm solving strong NVC that we assume to exist. The output is stored in the *remove* set, which contains the information about the pseudocrashed node to remove and about the process that detects the pseudocrash. In action *inform*, process j tells process i, which detected the pseudocrash, about its intention to remove the pseudocrashed node b. When the process detecting the pseudocrash receives this message in action *result*, it keeps track of the pseudocrashed node b it suspected, together with the set of its neighbors. The latter set is incrementally constructed as more of these messages are received from neighbors of the pseudocrashed process.

Even though the PCD is a powerful device, it is unfortunately not sufficient to solve strong NVC. The following result, which is dual to the necessary condition in Theorem 5.3, concludes that PCDs cannot be used to provide strong NVC. This is captured as follows.

process j
variables
% logical neighborhoods of j, $N[j]$ implements J^l
N[]: array of set of ids, initially $N[j] = J$;

% id of node to be removed and id of node that detected pseudocrash
remove: set of (id, id);

% output mimics the result of a pseudocrash failure detector
output: set of (id, neighborhood) init $\emptyset$;

actions
inform :: *remove* $\neq \emptyset \rightarrow$
$\forall(b, i) \in remove$ **do**
send($\langle b, j\rangle, i$) ;
remove $:= \emptyset$;

result :: **rcv**($\langle b, i\rangle$,j) $\rightarrow$
output:= *output* $\oplus\{(b, N[b] \cup \{i\})\}$;

Fig. 6. Emulating a perfect PCD using the output of an algorithm solving strong NVC.

THEOREM 5.4 (PCDS INSUFFICIENT FOR STRONG NVC). *It is impossible to solve strong NVC in an asynchronous system equipped with a perfect PCD.*

PROOF. We prove this by contradiction. We assume that an algorithm $\mathcal{A}$ that solves strong NVC with a perfect PCD in an asynchronous system exists, and show that no such $\mathcal{A}$ can exist.

Assume a failure pattern $\mathcal{F}_1$ where $n \in \mathcal{F}_1(t)$ but $n \notin \mathcal{F}_1(t'')$ with $t'' > t$. In a computation C of $\mathcal{A}$, assume that n is suspected, together with its correct neighborhood, at $t', t < t' < t''$ and n is eventually removed at t''' Since $\mathcal{A}$ is correct, then no working node is ever removed, and it must necessarily be that $t''' < t''$.

Now, assume the same failure pattern but with a different A_s and A_p in another computation C' of $\mathcal{A}$, where n is suspected at t' (as in C) but is removed exactly at t''. Both computations are feasible for $\mathcal{A}$. However, because $\mathcal{A}$ is correct, n should be removed before t'', because n recovers at t''. This entails a lower bound on processes' execution speed, hence the system must be synchronous, leading to a contradiction. □

The problem here is that the removal of a pseudocrashed node needs to happen before a node recovers or pseudocrashes next. Since the duration of a pseudocrash is lower bounded, then the system should be synchronous to guarantee that the time to remove a pseudocrashed node is upper bounded. This result is fundamental, in that it establishes that strong NVC can only be offered in a synchronous system. The next section builds upon this result.

6. VIEW CONSISTENCY ENFORCEMENT

We know that a synchronous system is required to solve NVC, and we are equipped with an eventually perfect PCD for such type of system, illustrated in Figure 5. Considering a synchronous setting is not unreasonable for most real-world WSNs. Often, deployed systems rely on some form of time synchronization. This is the case, for example, whenever sensor readings gathered at different nodes need to be aligned to a common clock [Werner-Allen et al. 2006; Ceriotti et al. 2009; Kim et al. 2007b]. Time synchronization is nowadays achieved in WSNs through efficient protocols providing errors in the microsecond range [Sundararaman et al. 2005; Maróti et al. 2004; Ferrari et al. 2011]; the corresponding implementations are also quite stable and often part of the standard distribution of WSN operating systems [Dunkels et al. 2004; Hill et al. 2000]. The synchronized clocks that such protocols provide can straightforwardly be used to run the system in a synchronous fashion.

First, we present two increasingly weaker specifications of NVC, which capture the impossibility result for strong NVC in the presence of transient faults and the practical need to inform higher-level protocols whenever such transient faults occur. Next, we present a global algorithm that solves both specifications in networks where unidirectional links may be present, and a localized algorithm that solves the weakest specification in networks with bidirectional links.

6.1. View Consistency Specifications

Strong view consistency. The strong NVC specification of Section 4 ensures that the *logical* view of the topology is an accurate reflection of the relevant working part of the *physical* network. Specifically, the safety specification prohibits a node n from removing a working node m from its neighborhood, whereas the liveness specification states that any pseudocrashed node is eventually removed.

However, as in Figure 4, in the presence of transient faults it is generally impossible to discern real pseudocrashes from transient failures, and thus to avoid wrong detections. For instance, a transient fault may corrupt a node's memory and make a node believe that communication is not possible towards another device, even though this is not the case. This is as simple as overwriting some memory space in a node's neighbor table because of a software bug, a situation not unlikely in WSNs [Chen et al. 2009; Cooprider et al. 2007]. This intuition, which precisely corresponds to the example of Figure 1, leads to the following.

THEOREM 6.1 (IMPOSSIBILITY OF STRONG NVC). *Given a network $G = (V, A)$ where all nodes are equipped with an eventually perfect PCD, there exists no algorithm providing strong NVC when both transient faults and pseudocrashes can occur.*

The intuition behind this result is based on the ability of transient failures to arbitrarily corrupt a node's state. When a pseudocrash is detected, the relevant nodes need to be informed so the pseudocrashed node can be removed from their logical neighborhoods. However, this notification may be wrong since the information about which node to inform may be corrupted due to transient faults. This means that the detecting node may end up informing the wrong nodes; for example, nodes that do not list the pseudocrashed node among their 1-hop neighbors. This evidently leads to a violation of the problem specification.

This result applies to both localized and global algorithms, although the intuition above deals with the localized case. For a global algorithm that has access to complete topology information [Masuzawa 1995], this information itself may be corrupted due to incorrect information at multiple processes, and thus strong NVC cannot be achieved even in this case. Additional details and formal proofs are also available [Jhumka and Mottola 2014].

Stabilizing strong view consistency. To remedy this result, we allow nodes to finitely make mistakes by removing working nodes. This leads to the weaker specification of *stabilizing strong NVC*.

Definition 6.2 (*Stabilizing strong view consistency*). Given a network $G = (V, A)$, and two nodes $n, m \in V$, a program provides *stabilizing strong NVC* for G if every computation of the program satisfies:

— (Eventual safety): There exists a time after which no working node is removed.
— (Liveness): Every time a node m pseudocrashes, eventually $\forall n : m \in N : n$ removes m.

Intuitively, stabilizing strong NVC ensures that strong view consistency is eventually established again. However, the issue with stabilizing strong NVC is that, from the perspective of higher-level protocols, these protocols cannot adapt their behavior when transient faults occur; for example, to obtain a sub-optimal, yet efficient configuration. This is because no feedback is provided to the protocols whenever strong NVC cannot be achieved.

process j
variables
% 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
discovery:: **timeout**(*discover*)$\rightarrow$
top:=topology_discovery();
set($discover$,Δ);

detect:: $(\exists(q, Q) \in top : j \in Q \wedge (q \notin J)) \rightarrow \forall i : i \in Q \wedge q \in I$:
send(j cannot detect q,i);

remove:: **rcv**(p cannot detect b, j)$\rightarrow$
top:= *top*$\oplus\{(j, J \setminus \{b\})\}$;

Fig. 7. A global algorithm that solves stabilizing strong NVC in a synchronous system and in the presence of unidirectional links and transient failures.

Weak view consistency. To address this shortcoming, we present an even weaker problem specification, called *weak NVC*, which informs higher level protocols of a transient fault by raising a $\langle fault \rangle$ flag whenever that is possibly detected.

Definition 6.3 (*Weak view consistency*). Given a network $G = (V, A)$, and two nodes $n, m \in V$, a program provides *weak NVC* for G if every computation of the program satisfies:

— (Eventual safety): There exists a time after which no working node is removed.
— (Weak liveness): Every time a node m pseudocrashes, then eventually $\forall n : m \in N : n$ removes m or a $\langle fault \rangle$ flag is raised.
— (Validity): A $\langle fault \rangle$ flag is raised only if there is a fault in the network.

Weak NVC, in essence, attempts to achieve strong NVC whenever possible. However, state inconsistencies induced by transient faults may threaten the efficiency of the network. To remedy this problem, it is beneficial that higher-level protocols are made aware of these inconsistencies. Thus, we adopt a two-pronged approach: either there is no inconsistency and strong NVC can be achieved or there is an inconsistency and a fault is detected. Thus, a $\langle fault \rangle$ flag is raised only if a fault exists in the network, and higher-level protocols can react to such notification by taking appropriate countermeasures. The specific actions to take are, in general, protocol-specific. Section 8 describes cases where simple corrections, triggered by the $\langle fault \rangle$ flag, already provide significant improvements.

6.2. Stabilizing Strong NVC with Unidirectional Links

In WSNs, due to issues such as background noise and interference, links can become unidirectional [Baccour et al. 2012]. This entails that only one of the nodes will be suspected by the other. This creates an asymmetry in the pseudocrash suspicions that, as far as a node's neighbors are concerned, cannot be distinguished from a transient failure. We thus seek to understand the impact of such link asymmetry on NVC.

THEOREM 6.4 (UNIDIRECTIONAL LINKS). *There exists no localized algorithm that solves weak NVC in a network with unidirectional links and using an eventually perfect PCD in the presence of transient faults.*

PROOF. We assume a localized algorithm $\mathcal{A}$ that solves the weak view consistency exists and show a contradiction.

Consider a failure pattern $\mathcal{F}$ such that a node i pseudocrashes at time t_1 and t_2, where $t_2 > t_1$. Assume there is a link (i, j) in the network. At t_1, the link becomes unidirectional, in the sense that j can transmit to i, but not viceversa. Eventually, with the eventual strong completeness of the PCD at j, $(i, \hat{I}) \in H_\Sigma(j, t'), t_1 \leq t' \leq t_2$.

Now, consider a computation C of $\mathcal{A}$. Since $\mathcal{A}$ satisfies weak NVC, there exists a time $t'', t' \leq t'' \leq t_2$ where all neighbors of m, such that $i \in M$, remove i by t''. At t_2, i crashes and so, as far as its logical neighbors are concerned, it pseudocrashes again and at $t''' > t_2$ there will be a node h that suspects i, thus $\exists h.(i, \hat{I}) \in H_\Sigma(h, t''')$. At t_2, $j \in I$ but $i \notin J$, since the link (i, j) becomes unidirectional at t_1.

Since $\mathcal{A}$ provides weak NVC, there exists a time $t_3, t_3 > t_2$ where all nodes m, such that $i \in M$, will remove i or raise a $\langle fault \rangle$ flag by t_3 in C. The suspicion $H_\Sigma(h, t''')$ at h has indeed caused all neighbors of i to do so. However, since $j \in I$ but $i \notin J$, node j cannot remove i by t_3; j thus raises a $\langle fault \rangle$ flag to signal a transient fault. However, there is no transient fault in the computation C. Hence a contradiction. □

Theorem 6.4 incidentally indicates that specifications stronger than weak NVC cannot be solved in networks with unidirectional links using localized algorithms. To address this problem, all the neighbor nodes of a pseudocrashed node need to be informed about any potentially unidirectional links, which cannot be achieved only using localized interactions. This hints at global algorithms, able to reason on overall knowledge of the network topology.

We provide one such algorithm in Figure 7, which solves stabilizing strong NVC in the presence of unidirectional links. The algorithm generates notifications for all nodes suspected to have pseudocrashed, based on a global view of the network topology obtained in action *discovery*. Such global topology information, only eventually correct, is processed in action *detect* to generate pseudocrash notifications. We prove the correctness of the algorithm in Figure 7 in an extended technical report [Jhumka and Mottola 2014].

Global algorithms are rarely used in practice, in that localized ones tend to be more energy efficient [Estrin et al. 1999]. Because of this, our interest in the algorithm of Figure 7 is mainly theoretical: it provides evidence that stabilizing strong NVC can be solved in the presence of unidirectional links. In a system perspective, we focus on localized algorithms, as we illustrate in the following.

6.3. Weak NVC with Bidirectional Links

In networks of bidirectional links, either a link exists between two nodes or not. If an edge (a, b) existed at some point in time and then node a appears pseudocrashed to node b; for example, due to a link failure, then node b will also appear pseudocrashed at a. Therefore, in the absence of transient faults, the eventually perfect PCDs at these two nodes suspect each other at some time t, that is, $(a, \hat{A}) \in H_\Sigma(b, t)$ and $(b, \hat{B}) \in H_\Sigma(a, t)$.

As a matter of fact, this is the setting that most WSN protocols base their operation upon. For example, many of them employ packet retransmission schemes based on Automatic Repeat Request (ARQ), that is, by sending explicit acknowledgments in case of a successful transmission. Similar schemes, however, cannot easily work with unidirectional links. This means that such protocols need to use only bidirectional links, and must rule out unidirectional ones [Ko et al. 2011; Gnawali et al. 2009; Kim et al. 2007a; Mottola and Picco 2011; Voigt and Österlind 2008; Burri et al. 2007].

***WeakC* algorithm.** Figure 8 describes a *localized* algorithm that solves weak NVC, using the eventually perfect PCD of Figure 5 in networks of bidirectional links. We call this algorithm *WeakC*. In action *notify* the PCD is queried for suspects, and either a $\langle fault \rangle$ flag is raised if strong NVC cannot be established due to a transient fault, or a pseudocrash notification is sent to the neighbors of the suspected process. The notifications are processed in action *remove*. Depending on the logical state at the receiver, the notification may indicate a transient fault, signaled by raising a $\langle fault \rangle$ flag, or be added to the local suspects. When no more messages are in the incoming queue, the algorithm enforces NVC in action *update* by removing the suspects from the local neighborhood.

process j
variables
% logical neighborhoods of j, $N[j]$ implements J^l
N[]: array of set of ids, initially $N[j] = J$;

% set of pseudocrashed nodes and their 1-hop neighborhood
suspects: set of (id, neighborhood) init $\emptyset$;

actions
% the PCD returns some suspects
notify:: *suspects*$\neq \emptyset \rightarrow$
$\exists(i, \emptyset) \in suspects$: **send** (fault, BCAST);
$\forall(i, I) \in suspects. \forall n \in I$: **send** (j cannot detect i, n);

% adding the pseudocrashed node as suspects
remove:: **rcv**(p cannot detect b, j)$\rightarrow$
if $(b \notin N[j])$ **then**
send (fault, BCAST);
elseif $(b \notin \{x|(x, X) \in suspects\})$ **then**
suspects := *suspects* $\cup \{(b, \perp)\}$;
fi

% dequeuing fault message
skip:: **rcv**(fault, r)$\rightarrow$ skip;

% no more message, enforce NVC
update:: **rcv**$\langle\rangle \rightarrow N[j] := N[j] \setminus \{i|(i, I) \in suspects\}$;

Fig. 8. *WeakC*: a synchronous localized algorithm to solve weak NVC in a network of bidirectional links, using the output of the PCD of Figure 5.

THEOREM 6.5 (ALGORITHM WEAKC). *Algorithm WeakC of Figure 8 provides weak NVC.*

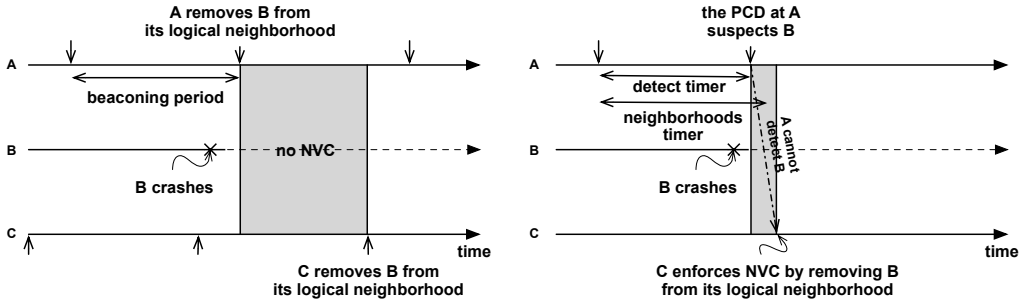
A formal proof for Theorem 6.5 is included in an extended technical report [Jhumka and Mottola 2014]. Intuitively, the correctness of *WeakC* descends from the properties of the PCD algorithm in Figure 5, used here to obtain the list of suspected pseudocrashes. The eventual safety property of *WeakC* follows from the eventual strong accuracy property of the PCD. The weak liveness property of *WeakC* follows from the strong completeness property of the same PCD. Finally, *WeakC* only raises a flag when transient faults are detected in action *notify* or *remove*.

Transient faults are detected whenever nodes have conflicting states. Thus, when a state inconsistency is detected in *WeakC*, it is certainly because of a transient fault. The $\langle fault \rangle$ flag is a way to inform higher-level protocols of this problem. Then, the protocols can use specific mechanism to recover from this situation. This is one of the key differences between *WeakC* and existing solutions such as periodic un-coordinated beaconing, which cannot discern transient faults from other kinds of faults, even though they may require different countermeasures.

Despite the increasingly weaker definitions of NVC introduced in Section 6.1, note that the problem remains *memoryless*. According to Definition 6.3, weak NVC indeed only advocates the removal of pseudocrashed nodes from a program's logical state whenever such faults occur, or the signalling of a $\langle fault \rangle$ flag if a transient fault does happen. This operation is independent of past failures. As a result, algorithm *WeakC* keeps no history of past faults, and both new nodes joining and existing nodes recovering are processed the same way[4].

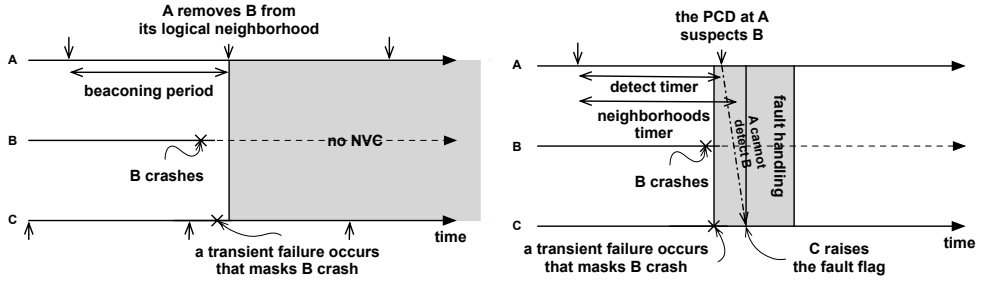
As we show next through sample executions, the processing of *WeakC* is sufficiently simple to replace the neighborhood management functionality in many existing WSN protocols, including

[4]If applications require failure histories to be kept, this information has to be stored separately from *WeakC*; for example, by keeping track of the number of times the $\langle fault \rangle$ flag is raised.



(a) Periodic un-coordinated beaconing. (b) *WeakC*.

Fig. 9. Comparison of periodic un-coordinated beaconing with *WeakC* in case of a node crash (times not to scale).



(a) Periodic un-coordinated beaconing. (b) *WeakC*.

Fig. 10. Comparison of periodic un-coordinated beaconing with *WeakC* in case of a masking failure (times not to scale).

those we discuss in Section 2. We design the APIs of the corresponding implementations, described in Section 7, aiming at facilitating the integration of *WeakC* in existing code bases.

Sample executions. Figure 9 compares (simplified) executions of periodic un-coordinated beaconing and *WeakC* in the case of a node crash with no transient faults, whereas Figure 10 shows the same comparison in case of a transient fault. Node B is a shared neighbor of both node A and C.

Figure 9(a) shows a case with no transient faults. Node A and node C can remove node B from their logical neighborhoods only at the end of the beaconing period, once they realize they miss the beacon from node B. As the two beaconing periods may be arbitrarily aligned, the time the system runs in the absence of NVC—shown by the shaded area in the picture—is *upper bound by the beaconing period*. In contrast, once the *detect* timer of Figure 5 expires at node A in Figure 9(b), the PCD at A starts suspecting node B, together with its logical neighborhood including both node A and C. As a result, this makes the guard in action *notify* of Figure 8 evaluate to true, thus node A sends a message "A cannot detect B" to node C and to itself. Because of the synchronous operation, both nodes realize that no more messages are coming and execute action *update*. Thus, they remove node B from their logical neighborhoods. The time the system spends in the absence of NVC is *upper bound by the message transmission time*. This is likely orders of magnitude lower than the beaconing period, which is typically set to tens of seconds or even minutes for saving energy [Gnawali et al. 2009; Mottola and Picco 2011; Dunkels et al. 2011; Hansen et al. 2011; Levis et al. 2004].

In a case of transient failures, shown Figure 10(a), node A removes node B from its logical neighborhood similar to Figure 9(a). However, node C is subject to a transient fault that masks the pseudocrash of node B; for example, the identifier of node B in node C's program state is rewritten so that, as far as node C is concerned, node B never existed. Thus, node B cannot be removed from node C's logical neighborhood. From the point in time when node A removes B, the system is run-

ning without NVC. This condition persists *until the transient fault on node C eventually clears* and the *periodic un-coordinated beaconing eventually restores the correct neighborhoods*. In contrast, in Figure 10(b) *WeakC* executes the same as in Figure 9(b) at node A. However, whenever node C receives "A cannot detect B", it recognizes a state inconsistency: node C is not aware of any node B, and it raises a $\langle fault \rangle$ flag. Node C proactively reacts to redress the problem. Thus, the time the system spends in the absence of NVC is *upper bound by the message transmission time* plus *the time to react to the fault.*

Unlike Figure 9, a direct comparison between the times spent in the absence of NVC is more difficult for Figure 10. However, let us consider the most benign transient fault, that is, one that clears already at the next round of beaconing. In this case, it would take again tens of seconds or even minutes until periodic un-coordinated beaconing redresses the situation in Figure 10(a). In contrast, let us consider the most elementary fault handling mechanism, that is, rebooting the node, even though more sophisticated mechanisms already exist [Chen et al. 2009]. Upon rebooting, techniques such as Trickle timers used in many modern protocols, will acquire complete neighborhood information in a few seconds [Levis et al. 2004]. In the most unfavorable case for *WeakC*, again the time the system spends in the absence of NVC is one order of magnitude smaller. This reasoning applies also to the example of Figure 1(b), if the data collection protocol at all chooses to route through node D because *WeakC* has not reacted yet, it will be for a very short amount of time. As a matter of fact, in Section 8 we observe that transient faults rarely clear so quickly.

These simplified examples highlight one of the key differences between periodic un-coordinated beaconing and *WeakC*. Once a fault is detected by the PCD of Figure 5, *WeakC* acts proactively. Differently, periodic un-coordinated beaconing simply proceeds the same, eventually restoring NVC if possible at all. This is the origin of most of the performance gains we observe in Section 8. The corresponding impact becomes larger as more complex fault patterns and transient faults emerge.

7. IMPLEMENTATION

In a system perspective, we focus on algorithm *WeakC* in that, compared to the global algorithm of Figure 7, its localized nature enables its integration with most existing WSN protocols. We implement *WeakC* as in Figure 8, including the eventually perfect PCD of Figure 5, on top of both the Contiki [Dunkels et al. 2004] and the TinyOS [Hill et al. 2000] operating systems. For Contiki, we integrate our implementation with both Contiki's IPv6 [Ko et al. 2011] and the Rime [Dunkels et al. 2007a] stacks. In TinyOS, our *WeakC* implementation targets the ActiveMessage stack.

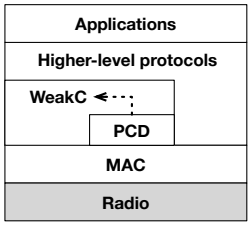


Fig. 11. Network stack when *WeakC* is integrated with higher-level protocols.

Our implementations are lightweight. In Contiki, they occupy 1.3 KBytes of program memory and 352 bytes of data memory when used with the IPv6 stack, or 1.2 KBytes of program memory and 312 bytes of data memory with Contiki's Rime stack. Our TinyOS implementation takes 1.45 KBytes of program memory and 328 bytes of data memory. The amount of data to keep track of a neighbor is 12 bytes in all cases but in the IPv6 stack, where it is 18 bytes due to larger addresses. This allows our implementations to scale efficiently also in dense networks.

Layering and APIs. Independently of the platform, *WeakC* sits between the higher-level protocols that rely on NVC and the MAC layer, as shown in Figure 11. We choose this layering because of two reasons. By sitting atop the MAC layer, the periodic exchange of neighborhood information required by the PCD as well as the messages generated by *WeakC* can take advantage of the radio duty-cycling and collision avoidance mechanisms the MAC layer offers. As we demonstrate in Section 8, a simple CSMA schema with random back-off suffices to handle packet collisions possibly caused by and between our algorithms. In other words, randomly staggering the transmissions in time makes collisions occur sparingly and, most importantly, not systematically that they can be misinterpreted as pseudocrashes by the PCD. As a result, their impact on the performance be-

comes negligible. This layering also minimizes the disruption due to the network traffic generated by *WeakC* against other concurrently-running protocols on the same node, as access to the radio remains mediated by a single MAC protocol [Polastre et al. 2005].

Designing the API for higher-level protocols to rely on *WeakC* needs to strike a balance between minimality—motivated by the resource-scarcity of the target devices—and ease of integration. We eventually settle on six primitives:

```
uint8_t getCurrentViewId();
uint8_t getNeighborhood(uint8_t view_id, uint16_t* node_ids);
bool isNeighbor(uint8_t view_id, uint16_t id);

void setPeriod(uint16_t period);
void setPayload(uint8_t size, uint8_t* data);
void registerNeighborhoodInfo(void (*callback) (uint16_t neighbor,
                  uint16_t number, uint16_t* node_ids, uint8_t* data));
```

The first three operations are used to manage neighborhood views. Function **`getCurrentViewId`** returns an identifier of the current neighborhood view. Higher-level protocols use this to detect neighborhood view changes and as parameter to **`getNeighborhood`** and **`isNeighbor`**. The former is used to query a neighborhood view; our *WeakC* implementation caches the current and the past **`T`** neighborhoods views, **`T`** being a compile-time parameter. This is provided merely as a convenience to higher-level protocols, as they often need to keep track of past neighborhoods views; for example, to identify nodes that joined or disappeared. Rather than delegating this functionality to the protocols, possibly duplicating it across concurrently-executing components, we choose to implement it within *WeakC* and to make it available through our API. However, *WeakC* makes no use of it, as the NVC problem is memoryless, as discussed in Section 6.3.

The last three operations serve to use *WeakC* as a replacement of the neighborhood management functionality in existing protocols. These call **`setPeriod`** to control the rate of neighborhood information exchange, that is, to dynamically set the value of Δ in the PCD of Figure 5. This is fundamental whenever a protocol uses adaptive beacon timers [Gnawali et al. 2009; Ko et al. 2011; Levis et al. 2004]. Any additional data that needs to be exchanged is given with **`setPayload`**. Whenever neighborhood information is received at a node, higher-level protocols are informed through a callback they previously registered using **`registerNeighborhoodInfo`**. The callback receives as a parameter the identifier of the neighbor node, as well as the number and identifiers of its 1-hop neighbors, plus any payload set at the sender side.

Network support. In action *notify* of Figure 8, *WeakC* requires to send pseudocrash notifications to all 1-hop neighbors of the pseudocrashed node. This functionality needs to fulfill two requirements. First, the notifications must be transmitted *reliably* for NVC to be re-established correctly and quickly. If some of the notifications are lost on the way; for example, because of interference, the destinations may take much longer to re-establish NVC, or possibly never do so in the case of transient failures. Moreover, the destinations may be more than two hops away, as exemplified in Figure 12. If node B crashes and node A first detects the pseudocrash, the notification needs to reach node C so that it can *remove* node B. However, node C is no longer reachable from A in two hops, as node B was the only intermediate device. In the general case, the destination of pseudocrash notifications may lie at any number of hops away from the node detecting the pseudocrash. Therefore, we require *multi-hop* relaying of notifications.

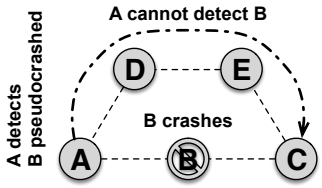


Fig. 12. A case where pseudocrash notifications need to travel more than two hops to reach the 1-hop neighbors of a pseudocrashed node.

We address both requirements using a simple expanding-ring n-hop flooding. Delivery to the intended receivers is determined by a list of destinations embedded within the packet. Processing starts with $n = 2$ to cover the 2-hop physical neighborhood. The destinations reply with an acknowledgment sent along the reverse path towards the sender, using reliable 1-hop unicast. If some

destinations do not acknowledge the packet, either because they lie beyond the n hops or the packet is lost, the originator repeats the process by exponentially increasing n until all destinations send an acknowledgement, or up to a maximum number of retries.

This mechanism provides reliable multicast with no control traffic and little memory overhead compared to existing multicast protocols [Cao et al. 2007; Flury and Wattenhofer 2007]. These use explicit routing tables to maintain originator-destination paths, which may become corrupted. In our case, all nodes may be originators or destinations, which would yield significant memory overhead. Routing tables also require constant maintenance as the may become unusable when nodes or links fail, namely, precisely when *WeakC* needs reliable multicast.

On the other hand, embedding the list of destinations within the packet might become an issue in case their number is large. As these notifications are solely sent to the *1-hop neighbors* of a pseudocrashed node, however, we expect their number to remain within practical limits. Moreover, there is plenty of space in the notification packets, as they only contain little control information, besides the list of destinations. The influence of increased packet length on the reliability of transmissions does not pose practical problems, as demonstrated in Section 8.

Our scheme may also occasionally experience high latency if the destinations are many hops away, due to the many iterations until n reaches a sufficient value. Increasing n exponentially rather than linearly [Jhumka and Mottola 2009] ameliorates the problem in exchange of a little energy overhead. We also observe that pseudocrash notifications most often target a subset of the initial 2-hop neighbors of the originator. Indeed, as WSNs tend to be dense networks [Intanagonwiwat et al. 2002], the distance to the former 2-hop neighbors is unlikely to increase drastically, if at all. Thus, an increase of n beyond 2 is rarely needed, as we observe ourselves in Section 8.

8. SYSTEM EVALUATION

We analyze first the performance of our *WeakC* implementation in isolation. The study is instrumental to understand the impact of using NVC in a full stack. To assess this, we modify four existing higher-level protocols to use *WeakC* for neighborhood management, and compare their performance against the original versions. We carry out all experiments on Twist [Handziski et al. 2006], a real-world WSN testbed of 90 TMote Sky nodes deployed in a university building, and hence naturally subject to external phenomena such as interference from co-located WiFi networks and changes in wireless propagation due to moving objects and persons. We use a -7 dBm transmit power, which in Twist yields a 4 hop network depth. With this configuration, every node in Twist features between 5 and 25 neighbors, resulting in a challenging setting for our algorithms. Every data point is the result of at least five 1-hour long runs. We totalled more than 700 hours of experiments in Twist. We divide each run in 1-minute rounds. We start measuring after the first 5 rounds to let the system acquire initial topology information.

8.1. Micro-Benchmarks

As the overhead of *WeakC* in the absence of faults is the same as periodic un-coordinated beaconing, our analysis concentrates on the performance in handling view changes in response to faults.

Metrics. We consider the following metrics: *i)* the *latency* to establish NVC, that is, the time from when a change in the physical topology is detected to when all affected nodes re-gain NVC or a $\langle fault \rangle$ message is sent: this is proportional to the time the system spends in the absence of NVC, and is thus to be minimized; *ii)* the *network overhead*, defined as the total number of packets exchanged at the physical level to carry out a view change, including retransmissions: the more traffic *WeakC* generates, the higher impact it has on the operation of higher-level protocols, therefore, this metric needs to be minimized too; and *iii)* the total *energy* spent during a view change, accounting for both processing and communication [Demirkol et al. 2006]: the energy spent by *WeakC* during a view change may influence the system lifetime, and it is hence to be minimized as well.

To measure these quantities, we rely on a time-synchronized USB back-channel available in Twist. Throughout the analysis, we use Contiki's implementation of *WeakC* together with the Rime

stack, which is functionally equivalent to the Contiki/IPv6 and TinyOS implementations. We measure energy consumption using Contiki's energy estimation mechanism [Dunkels et al. 2007b].

Settings. We explore three different failure scenarios, as follows:

(1) We artificially inject node crashes, link failures, and data corruption independently at every node with probabilities from 2% to 16%. These probabilities apply on a per-round basis and on a system-wide scale; for example, with a failure probability of 8%, every 1-minute round there is an 0.08 probability that a *uniformly* chosen node crashes, or a link fails, or data is corrupted in a node's program state. These failure probabilities are not uncommon in real deployments, as discussed in Section 2. To maintain a constant network density, node crashes and link failures are artificially recovered with the same probabilities, starting with the round following the one where the failure occurs.

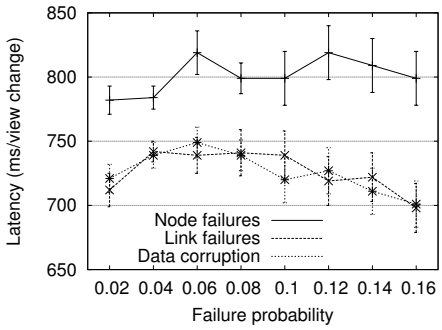
(2) We artificially create situations where a failure impacts several nearby nodes. This models the case of wireless interference originating from WiFi access points [Srinivasan et al. 2010]; of physical accidents, such as fire in a room, that may cause multiple nearby nodes to crash almost simultaneously; and of software bugs occurring because of conditions that affect close nodes, such as a buffer overflow in the neighbor table caused by the addition of a new device [Cooprider et al. 2007]. To emulate WiFi interference, we use the technique by Boano et al. [2009], which allows us to create realistic repeatable interference patterns. With the same probabilities above, we pick a random node and run Boano's interferer on it for a round. Typically, this affects most of the links among the 1-hop neighbors of the interferer, including a good fraction of those in between these. To emulate multiple node crashes and co-located cases of data corruption, again we select a random node with the same probabilities as above, and either force a shut down on it and all of its current 1-hop neighbors, or inject a data corruption on the same nodes.

(3) We only consider the "natural" failures that would normally happen in Twist. To detect them, at every round we checkpoint the state of every node and dump that on a base-station via the USB back-channel [Österlind et al. 2009]. The base-station detects natural failures by comparing consecutive checkpoints at the same node. Note that we cannot prevent natural failures from happening in the first and second failure scenario as well. In these cases, the base-station records the occurrence of the natural failures and instructs the affected node to skip the generation of an equal number of artificial failures. This maintains the overall failure probabilities constant.

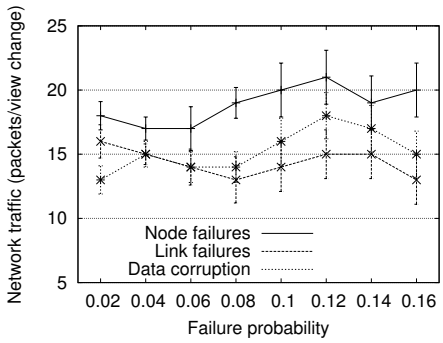
We use ContikiMAC as MAC layer, with the default parameter setting. Because of the layering shown in Figure 11, ContikiMAC also takes care of handling collisions among the packets generated by our algorithms at different nodes, using a CSMA with random backoff. Exchange of neighborhood information occurs every 5 seconds, that is, $\Delta = 5$ s in Figure 5. We consider a link as failed whenever at least 5 consecutive packets are lost over that link. Because at every node at most one failure occurs in a 1-minute round, $\Delta = 5$ s ensures that failures are eventually detected before they are possibly recovered in the next round, matching the conceptual framework of Section 3.2. The timeout before re-trying the multicast send of pseudocrash notifications is 300 ms.

Failure scenario 1: spatially-uncorrelated artificial failures. Figure 13 summarizes the results. Figure 13(a) shows the view change latency we measure depending on the type of failure, against their probability. The absolute values are limited and always within one second, with little variability across different runs. As demonstrated in Section 8.2, this allows protocols to quickly re-gain NVC, which is beneficial to their performance. The values in Figure 13(a) do include the latency that ContikiMAC introduces due to radio duty-cycle. With the default parameters, every packet transmission takes an average 62 ms with ContikiMAC. A minimum of two packet transmissions are necessary for every view change, counting one pseudocrash notification and the corresponding acknowledgement. In practice, we count about 4 to 6 packet transmissions for every view change. This means that 248 ms to 372 ms out of the values in Figure 13(a) are due to ContikiMAC alone.

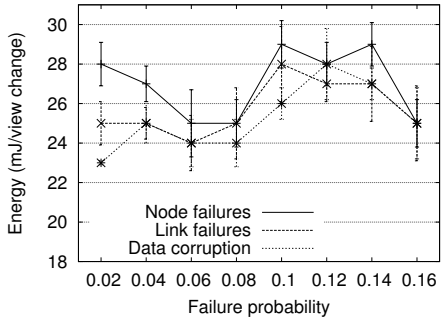
Figure 13(a) also demonstrates that the failure probability bears no appreciable impact. The system recovers from failures sufficiently rapidly that view changes are almost never concur-



(a) View changes are rapid; higher-level protocols quickly re-gain consistent neighborhood information.



(b) The network overhead is minimal; higher-level protocols are unlikely to be affected significantly.



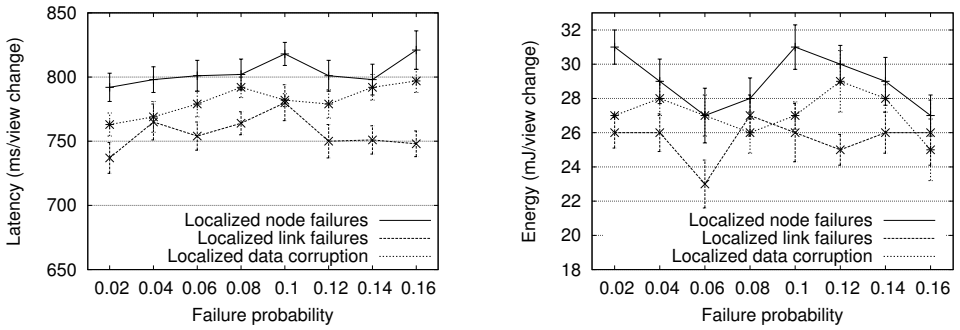
(c) The impact of *WeakC* on overall system lifetime is negligible; it is comparable to transmitting a few additional radio packets.

Fig. 13. Performance of *WeakC* per view change: spatially-uncorrelated artificial failures.

rent, and hence the corresponding network traffic seldom interferes. Unlike our earlier simulation results [Jhumka and Mottola 2009], the view latency is markedly higher in case of node failures. We attribute this to the network topology in Twist, fairly irregular because of physical constraints [Handziski et al. 2006]. Because of this, when a node fails, it may take more time to route notifications around the "hole" created by the failed node. This effect was not evident in simulations, where nodes were more uniformly deployed. Our measures of network overhead, illustrated in Figure 13(b), confirm this reasoning. The failure probability again appears not to influence the performance. In case of node crashes, about 22% more packets are transmitted around the "hole" created by the failed node. The higher variability across runs also confirms that this behavior may or may not manifest depending on where in the network a crash occurs.

Figure 13(c) reports the energy consumption for a view change, including computation and communication. Given the values at stake, we argue that the impact of NVC on the overall system lifetime is, in fact, negligible. To place this argument in context, consider that transmitting a single 50 byte packet with a TMote Sky node takes about 4 mJ at maximum power. Notably, the higher network overhead for node failures seen in Figure 13(b) is, based on our experiment logs, spread across more nodes than with other types of failures. This phenomena balances out the higher number of transmitted messages, resulting in an energy consumption comparable to other types of failures.

Failure scenario 2: localized artificial failures. Figure 14 reports the results in the case of localized failures. Compared to the performance with spatially-uncorrelated artificial failures, the latency is only marginally higher, as shown in Figure 14(a). This is most likely caused by the additional pro-



(a) The performance is only marginally worse than Figure 13(a); the system suffers slightly more from link failures than from other failures if these are localized.

(b) Energy consumption is only slightly higher than Figure 13(c); the system suffers more from localized node failures, as more devices are involved at once.

Fig. 14. Performance of *WeakC* per view change: localized failures.

cessing required when the PCD of Figure 5 suspects multiple neighbors at once. In relative terms, comparing Figure 13(a) against Figure 14(a) indicates that the added overhead for link failures and data corruption is higher than for node crashes. In the latter case, multiple nearby nodes simultaneously crashing create a situation similar to the one of the spatially-uncorrelated case: again a "hole" in the network is created, which is however larger than before, because it involves more nodes. Only the nodes on the fringes of such a hole are involved in the view changes. Differently, localized link failures create a variegated scenario, in that some packets may still go through some of the affected links if these happen to be located in a favorable position compared to the interferer.

Our measures of network traffic, which we omit here for brevity, are comparable to Figure 13(b). Even though the packets required to handle multiple concurrent view changes overlap in space, possibly creating collisions at the physical layer, the network support described in Section 7 and ContikiMAC handle the situation with no performance degradation. Figure 14(b) instead illustrates the energy consumption for a view change in the case of localized failures. Again, the performance is only marginally worse than in Figure 13(c). The relative performance penalty that a comparison with Figure 13(c) indicates is here higher for the case of node failures. As the "hole" is larger than in the case of spatially-uncorrelated failures, more nodes are involved in every view change, and each of them needs to send packets and process information, thus making the impact of a single failure more significant on the network-wide figure.

Failure scenario 3: natural failures. Some preliminary observations are in order about how natural failures occur in Twist. First, we confirm that link failures occur routinely in WSNs [Baccour et al. 2012; Srinivasan et al. 2010], and are almost exclusively localized. We conjecture that they are mainly caused by WiFi interference in the university building [Handziski et al. 2006]. Moreover, we record a per-node 0.01% probability that a data corruption occurs at any round. Twist cannot be physically accessed, so we are unable to trace with certainty the source of these faults. However, the code on the nodes is minimal: it only includes the Contiki OS, ContikiMAC, our implementation of *WeakC*, and custom logging functionality. Therefore, we conjecture that the data corruptions we observe are due to phenomena such as bit flips; indeed, the nodes in Twist are totally unprotected from electromagnetic radiations and static currents. Finally, unlike in real deployments, in Twist all nodes are powered through the USB back-channel, and so we detect only two node crashes during our tests, probably due to physical detaching of the node from the USB cable. We regard these cases as statistically irrelevant. Compared to many WSN installations, Twist is anyways a fairly benign environment. As a matter of fact, the failure rates above, especially for what concerns node crashes, do underestimate the nature of real deployments, as illustrated in Section 2.

Failure	Latency (ms)	Network traffic (packets)	Energy consumption (mJ)
Link	768	15.04	25.32
Data corruption	722	14.89	25.89

Fig. 15. Performance of *WeakC* per view change: natural failures.

Figure 15 reports on the performance of *WeakC* in this setting. The absolute numbers are inline with the previous results. As expected, the performance in the case of link failures resembles more the results of Figure 14(a)—obtained with (artificial) localized failures—than those of Figure 13(a), produced with uniformly distributed failures. The performance in the case of data corruption actually indicates the opposite, as the values in Figure 15 are closer to those in Figure 13(c) than Figure 14(b). Also according to the checkpoints collected at the sink, natural data corruptions tend not to show any spatial correlations. Based on the same information, we also confirm that data corruption faults actually persist for up to tens of minutes, that is, our worst-case analysis of Section 6.3 does underestimate the reduction of the time spent without NVC when using our algorithms.

8.2. Full-stack Performance

We assess the benefits brought by NVC to higher-level protocols. We replace the existing neighborhood management in four existing routing protocols with our *WeakC* implementation, and compare their performance with the original designs.

Metrics. We consider two metrics commonly used for evaluating WSN protocols [Gnawali et al. 2009]: *i) data yield*, that is, the fraction of application packets successfully received at the destination(s) over those sent, and *ii) radio duty cycle*, that is, the fraction of time a node keeps the radio on to deliver packets to the destination(s). The former is an indication of the level of service provided to applications, whereas the latter provides a measure of a protocol's energy efficiency. We measure energy consumption using Contiki's energy estimation functionality [Dunkels et al. 2007b] and a similar mechanism in TinyOS.

Protocols and settings. We use four markedly different protocols to show the general applicability of NVC and to experiment with different traffic patterns and loads. Figure 16 summarizes the settings. Note that the performance of the protocols we consider cannot be directly compared with each other, because every protocol addresses a *sharply* different set of application requirements. Rather, we aim at demonstrating that equipping a diverse set of protocols with NVC is beneficial. The comparison is thus between the original design and the NVC-equipped version of the same protocol.

Protocol	#sources	#sinks	packet interval
CTP	89	1	1 min
MUSTER	18	4	1 min
RPL	10	10	5 min
COREDAC	10	1	5 min

Fig. 16. Tested protocol settings.

The Collection Tree Protocol (CTP) [Gnawali et al. 2009] for TinyOS is a staple data collection protocol. CTP builds *many-to-one* routes by minimizing a routing metric based on ETX [Baccour et al. 2012]. Differently, MUSTER [Mottola and Picco 2011] is a *many-to-many* TinyOS protocol that builds multi-hop routes by minimizing a routing metric that combines ETX and the expected node lifetime. In MUSTER, the pairing of sources and sinks is decided randomly and differently for every run. Both CTP and MUSTER run atop the Low-Power Listening MAC layer found in the TinyOS distribution, using settings in line with earlier experiments with either protocol [Gnawali et al. 2009; Mottola and Picco 2011].

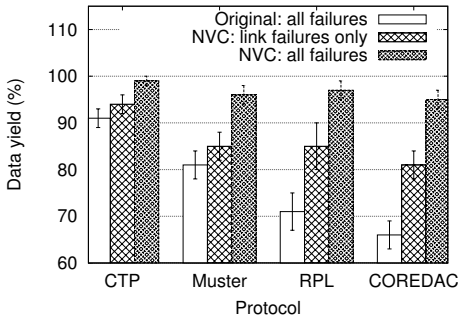
The Routing Protocol for Low-Power and Lossy Networks (RPL) [Ko et al. 2011] with the IPv6 stack runs on Contiki. RPL is a IETF-standardized distance-vector protocol for IP-based communications in low-power wireless. We use RPL in a *one-to-one* scenario, randomly choosing ten disjoint source-sink pairs. The higher packet interval compared to CTP and MUSTER is because RPL was not able to sustain higher traffic loads in our experiments, independently of the use of NVC. COREDAC [Voigt and Österlind 2008] also runs on Contiki with the Rime stack. Unlike the other protocols—all based on opportunistic medium access—COREDAC supports *many-to-one* traffic

with TDMA scheduling. In COREDAC, only ten sources generate packets; a higher load easily generates queue overflows at the intermediate nodes, regardless of the use of our algorithms.

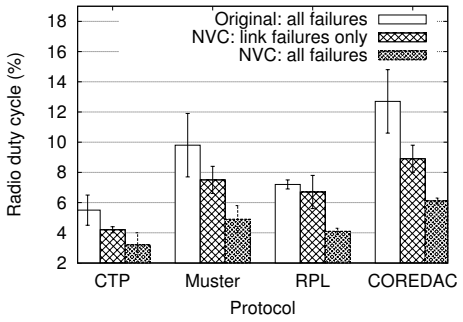
All protocols heavily rely on neighborhood information, maintaining knowledge on both the current next-hop nodes and about potential alternatives; for example, in case of a link failure. They all employ similar schemes for managing neighborhood information, all based on uncoordinated beaconing. Because of different traffic patterns and routing techniques, however, every protocol exhibits peculiar behaviors. The resulting diversity shows the general applicability of NVC. The integration of *WeakC* in all protocols follows the layering shown in Figure 11; the existing MAC layer handles the traffic generated by our algorithms in addition to the traffic generated by the original protocols. Specifically, the former only amounts to the notification messages possibly generated by *WeakC* upon suspecting a fault. The network traffic, as measured in number of packets, is otherwise equal to periodic un-coordinated beaconing.

All protocols run with their default parameters. The parameters of *WeakC* are as in Section 8.1 but Δ, which is driven by the protocol that employs *WeakC* using the API described in Section 7. For example, CTP exchanges neighborhood information using Trickle timers [Gnawali et al. 2009]. In all cases, $\Delta < 1$ min, ensuring that failures never go undetected before they are possibly recovered already in the next round, in accordance with the framework of Section 3.2. As already mentioned, the protocols we study, as well as the majority of WSN protocols, apply techniques to rule out unidirectional links. Therefore, the link failures we discuss next are to be considered as failures that prevent communication in both directions. If a link failed in only one direction, the protocol would just exclude the link from processing and pretend it did not exist.

In all protocols, sending or receiving a $\langle fault \rangle$ message in *WeakC* triggers a node reboot, as the state is now considered untrusted. We choose this strategy merely for simplicity. Note, however, how this places the NVC-equipped protocols in the worst case: whenever a fault is detected, the entire state is lost and needs to be reacquired. As a result, any gain we demonstrate by employing NVC is to be considered as a *lower bound*. Protocol-specific mechanisms to deal with data corruption may, indeed, determine that only a part of the state needs to be discarded and that rebooting is unnecessary [Chen et al. 2009]. In these cases, the NVC-equipped versions would likely show even greater performance improvements compared to the original designs.



(a) NVC significantly improves data yield.



(b) NVC-equipped protocols consume far less energy.

Fig. 17. Performance of NVC-equipped protocols against the original designs.

We let the nodes be subject *only* to natural failures to analyze the most realistic settings. These notably include localized phenomena, such as interference from WiFi access points, that impacts multiple nearby nodes at once. We keep recording the occurrence of failures as in Section 8.1, to provide an indication of what failures occurred and with what frequency.

Results. The experiments again confirm that link failures routinely occur in WSNs. Moreover, similar to Section 8.1, we detect only one node failure in Twist, which we deem statistically immaterial. Differently, in these experiments data corruption happens with an overall $\sim$0.1% probability on a per-round and per-node basis. The same figure was $\sim$0.01% in Section 8.1, which prompts further investigation. We eventually find out that RPL and COREDAC contain bugs that overwrite

random portions of the memory, including neighborhood information[5]. Not considering these two protocols in the estimation of the data corruption probability brings this figure back to the one of Section 8.1. Incidentally, this strengthens the motivation and value of our work. It is sad reality that WSN implementations tend to be buggy; many of these bugs are memory violations [Chen et al. 2009; Cooprider et al. 2007] that corrupt data in a node's memory.

Figure 17(a) shows the protocols' data yield with link failures only, as well as when both link failures and data corruption occur. Already in the first case, NVC-equipped protocols perform markedly better: the RPL protocol, for example, yields over 20% more data. Compared to the original way of managing neighborhood information, indeed, NVC proactively deals with link failures. This entails that the system gains updated knowledge of the physical topology both earlier and in a consistent manner across the involved nodes. This reduces the time the system spends in the absence of NVC and facilitates reconfigurations involving several nodes at once, in that these can readily reason on the *same* set of information. In addition, the ability to discern data corruption allows protocols to apply ad-hoc countermeasures—a simple reboot in our case—to discard corrupted state. As such, the NVC-equipped RPL protocol yields over 36% more data in the presence of transient faults.

The gains in data yield also correspond to a general reduction of radio duty-cycle, shown in Figure 17(b). *WeakC* allows protocols to operate with more accurate information, and hence their control traffic reduces. In MUSTER, for example, the radio duty-cycle is more than halved. Again, when data is corrupted, it is often more efficient for a protocol to perform a node reboot than to continue with corrupted neighborhood information.

Based on the logs of our experiments we can also draw protocol-specific considerations:

— in CTP, corruption in neighborhood information often causes a node to keep using unreliable links although better links would be available; it takes significant time before CTP re-acquires correct information. *WeakC* triggers a node reboot in this case, and hence the routing tables start afresh and are rapidly filled with correct information due to CTP's Trickle timers [Gnawali et al. 2009; Levis et al. 2004] for route discovery.
— in MUSTER, link failures and data corruption tend to misguide the route construction; it happens that MUSTER concentrates too many routes on the same nodes and packets are lost because of queue overflows. Using NVC greatly helps build near-optimal routes, which prevents queue overflows. We indeed record *no* queue overflows for the NVC-equipped version of MUSTER.
— in RPL, data corruption significantly affects the route maintenance: it causes a ripple effect whereby almost every node downstream of the data corruption changes upstream node. Using *WeakC* triggers a node reboot that, as in CTP, results in more efficient performance than the original behavior. A single node rebooting indeed typically causes only a few links to reconfigure.
— in COREDAC, data corruption and link failures likely lead to incorrect synchronization of transmissions from different nodes, and packets are lost because of systematic collisions. Using *WeakC*, COREDAC recovers from these situations more quickly than with the original mechanisms, which are unable to recognize the data corruption.

Ultimately, these observations demonstrate that by equipping protocols with NVC, and yet by applying simple counter-measures against transient faults, one can reap significant performance improvements across diverse scenarios, substantiating our initial claims.

9. RELATED WORK

Our work spans different fields, ranging from WSNs to fault detection and self-stabilizing algorithms, that arguably seldom cross-fertilize.

[5]The main developers of either protocol have confirmed the existence and extent of these bugs, as well as the fact that a simple fix is not straightforwardly identifiable.

WSN fault detection and diagnosis. To the best of our knowledge, we are the first to recognize the NVC problem in WSNs, even though a body of work exists in the field of fault detection and diagnosis for WSNs [Paradis and Han 2007; Yu et al. 2007].

In WSNs, several works provide surveys and general assessments on a variety of fault tolerance techniques. For example, Yu et al. [2007] and Paradis and Han [2007] present a general survey of failure management, including detection and recovery, while a survey of fault diagnosis is due to Mahapatro and Khilar [2013]. On the other hand, Alwan and Agarwal [2009] present a survey of fault-tolerant routing in WSNs. Cinque et al. [2013] present an overview of reliability assessment techniques for WSNs, considering various failure models, such as battery exhaustion, to evaluate the resiliency of WSNs. Munir et al. [2015] conduct simulations to analyze the synergy between fault detection and fault tolerance and develop Markov models to characterize the reliability of WSNs.

Specific works cover a range of aspects. For example, Lee and Choi [2008] use a failure diagnosis technique to detect faulty nodes by comparing their behavior with their neighbors. They show that permanently faulty nodes can be identified with high accuracy, which is intuitive since the continuous "bad" behavior of the node becomes a prominent feature in a node's history. Ould-Ahmed-Vall et al. [2012] develop a fault-tolerant event detection approach that enable nodes to detect erroneous decisions by leveraging the decisions of their neighbors. We share the same distributed approach at identifying faults; for example, in the way we recognize transient failures in *WeakC*.

In our work, a message failure is considered as a pseudocrash. This detection is possible as a node continuously monitors its neighbors [Mahapatro and Khilar 2013; Jhumka et al. 2014]. The failure may stem from several factors but we do not discriminate between them, since they all display the same symptom. For example, we do not determine whether the lack of a message arrival is due to a node crash or a message collision. This significantly eases the problem of view enforcement, since we can use a simple periodic exchange of neighborhood information to detect pseudocrashes.

Data faults. Techniques to detect pseudocrashes do not extend to error detection as the nodes or the network may display a normal behavior except for the sensor readings. In general, these types of errors are handled through an after-deployment calibration [Balzano and Nowak. 2007]. This is possible as the data is typically expected to fit a given model [Ni et al. 2009; Sharma et al. 2010]. On the other hand, Sharma et al. [2010] also mention the use of time-series analysis, estimation methods, and learning techniques at run-time to detect errors in sensor readings.

In FIND, Guo et al. [2014] focus on the detection of data errors that are either biased or random, in what can be termed termed as Byzantine data faults. They propose a detection technique based on the discrepancy between a sensor data rank and the distance rank, that is, FIND ranks the nodes based on their readings and their physical distances from the event and any significant discrepancy between these two metrics is characterized as a fault. Thus, this technique is only applicable for systems where the measured signal attenuates with distance. Such a technique does not extend to deal with WSN protocols, which is the focus of our work.

System approaches. Several works investigate fault-tolerant mechanisms at the operating system level. Cooprider et al. [2007] develop SafeTinyOS: a set of compile-time tools to instrument standard TinyOS code so that memory corruption errors are explicitly signaled to the programmer; for example, by blinking a node's LEDs. Chen et al. [2009] augment the TinyOS operating system with mechanisms to restore selected parts of the application upon recognizing state inconsistencies. These works are complementary to ours. For example, in Section 8, we choose to completely reboot the node whenever our *WeakC* implementation raises a $\langle fault \rangle$ flag. Using the work by Chen et al. [2009], we could restore only the relevant part of the application, possibly further improving performance in NVC-equipped systems.

Topology control protocols [Santi 2005], which adjust a radio's transmission power to fulfill given connectivity requirement, are also complementary to solving the neighborhood issues. While the latter stem from an inaccurate perception of the underlying connectivity, topology control may proactively manipulate the connectivity to reduce the chances that neighborhood issues arise.

Failure detectors and fault-tolerant protocols. In this work, we consider three fault models, namely: *i)* node crashes, *ii)* message omissions, and *iii)* transients faults. There exist several works that address the problem of tolerating permanent failures and transient faults of a *subset* of processes, as opposed to self-stabilization approaches, which focus on tolerating transient faults on *all* processes [Anagnostou and Hadzilacos 1993; Dolev 2000; Beauquier and Kekkonen-Moneta 1997]. In other related work, Beauquier et al. [1998] focus on detecting transient failures in a self-stabilizing way based on temporal and spatial locality. In contrast, we consider pseudocrashes and transient faults. On the other hand, Nesterenko and Arora [2002] study the problem of dining philosophers in the presence of malicious crashes, which is a special form of crash and transient failures where the latter eventually lead to a crash. Compared to their work, we tackle NVC as opposed to predicate detection, and do so under a different fault model.

Failure detectors are extensively used in the study of fundamental problems in distributed system, such as consensus [Chandra and Toueg 1996]. Gärtner and Pleisch [2002] show that failure detectors are inadequate to solve the global predicate detection problem in the presence of crash failures. The authors then introduce a stronger device, called a failure detector sequencer, which provides sufficient information to solve global predicate detection. Delporte-Gallet et al. [2005] introduce a device named Ω that solves the consensus problem in the presence of both crash and message omission failures. Aguilera et al. [2000] prove the limitations of failure detectors in solving consensus in the crash-recovery model. Our work is similar in that we also recognize that perfect failure detectors are insufficient to provide NVC and develop a new device to address the problem.

10. CONCLUSION

We identified and studied the NVC problem in WSNs. We proved that strong NVC cannot be enforced, even with a perfect failure detector. We then analyzed the problem from two angles: *i)* neighborhood monitoring, and *ii)* view enforcement. We developed a novel device called PCD and analyzed its properties, especially with transient faults. We then studied the problem of view enforcement in a synchronous system where nodes are equipped with the relevant PCDs. We provided a local algorithm that solves weak view consistency where higher-level applications are notified of failures. We showed that our approach improves the performance of existing protocols; for example, CTP [Gnawali et al. 2009] shows increased yield, with half the average energy usage.

A. MASKING VS. NON-MASKING TRANSIENT FAILURES

We back up our focus on non-masking transient faults in Sec. 4.2 by showing that this type of fault is much more likely to occur than masking ones; hence, the chances that a transient fault makes the system miss a pseudocrash are low overall.

Given a network $G = (V, A)$, we assume that transient faults follow a Poisson distribution with rate λ_{tf} and pseudocrashes follow a Poisson distribution with rate λ_{pc}. We focus on a given round with duration ψ and assume that a single pseudocrash occurs in that round. We denote the size of a neighborhood by η. For a PCD to miss the detection of a pseudocrashed node n, n should be in the *live* variable in the algorithm of Figure 5:

$$\begin{aligned}
&\Pr\{\text{detection of } n \text{ missed}\} \\
&\quad = \Pr\{n \text{ pseudocrashed } \&\exists \text{ entry in } \textit{m.live} \text{ corrupted to } n\} = \\
&\quad = \Pr\{n \text{ pseudocrashed}\} \cdot \Pr\{\text{an entry in } \textit{m.live} \text{ corrupted to } n\} \\
&\quad \approx \left(\frac{e^{-\psi\lambda_{pc}}\psi\lambda_{pc}}{\gamma}\right)\left(\eta\frac{e^{-\psi\lambda_{tf}}\psi\lambda_{tf}}{\gamma^2}\right) \quad (1)
\end{aligned}$$

Differently, the probability that node n has pseudocrashed and a non-masking transient fault occurred is equal to:

$$\Pr\{\text{n pseudocrashed \& non-masking transient fault occurred}\}$$
$$\approx \Pr\{\text{a pseudocrash and a transient fault occurs in } \psi\})$$
$$\approx (\psi^2 \lambda_{pc} \lambda_{tf} e^{-\psi(\lambda_{tf}+\lambda_{pc})}) \quad (2)$$

It is, indeed, definitely the case that $(1) \gg (2)$.

REFERENCES

M. K. Aguilera and others. 2000. Failure detection and consensus in the crash recovery model. *Distributed Computing* 13, 2 (2000).

B. Alpern and F. B. Schneider. 1985. Defining liveness. *Inform. Process. Lett.* 21 (1985).

H. Alwan and A. Agarwal. 2009. A survey on fault-tolerant routing techniques in wireless sensor networks. In *Proceedings International Conference on Sensor Technologies and Applications*.

E. Anagnostou and V. Hadzilacos. 1993. Tolerating transient and permanent failures. In *Proceedings of the* 7^{th} *International Workshop on Distributed Algorithms (WDAG)*.

A. Arora and others. 2004. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks* 46, 5 (2004).

A. Arora and S. S. Kulkarni. 1998. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the* 18^{th} *International Conference on Distributed Computing Systems (ICDCS)*.

N. Baccour and others. 2012. Radio link quality estimation in wireless sensor networks: A survey. *ACM Trans. Sens. Netw.* 8, 4 (2012).

L. Balzano and R. Nowak. 2007. Blind calibration of sensor networks. In *Proceedings of the* 6^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

G. Barrenetxea and others. 2008. The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the* 6^{th} *Conference on Embedded Network Sensor Systems (SENSYS)*.

J. Beauquier and others. 1998. Transient fault detectors. In *Proceedings of the* 12^{th} *International Symposium on Distributed Computing (DISC)*.

J. Beauquier and S. Kekkonen-Moneta. 1997. On FTSS-solvable distributed problems. In *Proceedings of the* 16^{th} *Symposium on Principles of Distributed Computing (DISC)*.

J. Beutel and others. 2009. Operating a sensor network at 3500 m above sea level. In *Proceedings of the* 8^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

N. Bhatti and L. Mottola. 2016. Efficient state retention for transiently-powered embedded sensing. In *Proceedings of the* 13^{th} *ACM International Conference on Embedded Wireless Systems and Networks (EWSN)*.

C. A. Boano and others. 2009. Controllable radio interference for experimental and testing purposes in wireless sensor networks. In *Proceedings of the* 4^{th} *International Workshop on Practical Issues in Building Sensor Network Applications (SENSEAPP)*.

B. J. Bonfils and P. Bonnet. 2003. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of* 2^{nd} *International Workshop on Information Processing in Sensor Networks (IPSN)*.

N. Burri, P. von Rickenbach, and R. Wattenhofer. 2007. Dozer: Ultra-low power data gathering in sensor networks. In *Proceedings of the* 6^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

Q. Cao, T. He, and T. Abdelzaher. 2007. uCast: Unified connectionless multicast for energy efficient content distribution in sensor networks. *IEEE Transactions on Parallel and Distributed Systems* 18, 2 (2007).

M. Cattani and others. 2014. Lightweight neighborhood cardinality estimation in dynamic wireless networks. In *Proceedings of the* 13^{th} *International Symposium on Information Processing in Sensor Networks (IPSN)*.

M. Ceriotti and others. 2009. Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment. In *Proceedings of the* 8^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

M. Ceriotti and others. 2011. Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels. In *Proceedings of the* 10^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

T. D. Chandra and S. Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM* 43, 2 (1996).

Y. Chen and others. 2009. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS* 22^{nd} *Symposium on Operating Systems Principles (SOSP)*.

J. Choi, M. Kazandjieva, M. Jain, and P. Levis. 2009. The case for a network protocol isolation layer. In *Proceedings of the* 7^{th} *Conference on Embedded Networked Sensor Systems (SENSYS)*.

M. Cinque, A. Coronato, A. Testa, and C. Di Martino. 2013. A survey on resiliency assessment techniques for wireless sensor networks. In *Proceedings of MobiWAC*.

N. Cooprider and others. 2007. Efficient memory safety for TinyOS. In *Proceedings of the* 5^{th} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

A. Cornejo and others. 2014. Reliable neighbor discovery for mobile ad-hoc networks. *Ad Hoc Networks* 12 (2014).

P. Costa and others. 2007. Programming wireless sensor networks with the TeenyLIME middleware. In *Proceedings of the* 8^{th} *ACM/USENIX International Middleware Conference*.

S. Dawson-Haggerty and others. 2012. @Scale: Insights from a large, long-lived appliance energy WSN. In *Proceedings of the* 11^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

C. Delporte-Gallet and others. 2005. Revisiting failure detection and consensus in omission failure environments. In *Proceedings of ICTAC*.

I. Demirkol and others. 2006. MAC protocols for wireless sensor networks: A survey. *IEEE Communications Magazine* 44, 4 (2006).

E. W. Dijkstra. 1974. Self stabilizing systems in spite of distributed control. *Commun. ACM* 17, 11 (1974).

S. Dolev. 2000. *Self-stabilization*. MIT Press.

A. Dunkels and others. 2004. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of* 1^{st} *Workshop on Embedded Networked Sensors*.

A. Dunkels and others. 2007a. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the* 5^{th} *Conference on Networked Sensor Systems (SENSYS)*.

A. Dunkels and others. 2007b. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the* 4^{th} *International Workshop on Embedded Networked Sensors*.

A. Dunkels and others. 2011. Announcements: Efficient protocol concurrency in low-power wireless networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*.

S. Duquennoy and others. 2013. Let the tree bloom: Scalable opportunistic routing with ORPL. In *Proceedings of the* 11^{th} *Conference on Embedded Networked Sensor Systems(SENSYS)*.

P. Dutta and others. 2006. Trio: Enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proceedings of the* 5^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

P. Dutta and D. Culler. 2008. Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications. In *Proceedings of the* 6^{th} *Conference on Embedded Network Sensor Systems (SENSYS)*.

D. Estrin and others. 1999. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the* 5^{th} *International Conference on Mobile Computing and Networking (MOBICOM)*.

F. Ferrari and others. 2011. Efficient network flooding and time synchronization with Glossy. In *Proceedings of the* $12^{t}h$ *International Conference on Information Processing in Sensor Networks (IPSN)*.

F. Ferrari and others. 2012. Low-power wireless bus. In *Proceedings of the* 10^{th} *Conference on Embedded Networked Sensor Systems (SENSYS)*.

N. Finne and others. 2008. Experiences from two sensor network deployments self-monitoring and self-configuration keys to success. In *Proceedings of the International Conference on Wired/Wireless Internet Communications (WWIC)*.

R. Flury and R. Wattenhofer. 2007. Routing, anycast, and multicast for mesh and sensor networks. In *Proceedings of the* 5^{th} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

F. C. Gärtner and S. Pleisch. 2002. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *Proceedings of the* 22^{th} *Symposium on Principles of Distributed Computing (DISC)*.

O. Gnawali and others. 2009. Collection tree protocol. In *Proceedings of the* 8^{th} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

S. Guo, H. ZHang, Z. Zhong, J. Cheng, Q. Cao, and T. He. 2014. Detecting faulty nodes with data errors for wireless sensor networks. In *ACM Transactions on Sensor Networks*.

V. Handziski and others. 2006. TWIST: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *REALMAN*.

M. Hansen and others. 2011. Unified broadcast in sensor networks. In *Proceedings of the* 10^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

J. Hill and others. 2000. System architecture directions for networked sensors. In *Proceedings of the* 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

T. W. Hnat and others. 2011. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the* 9^{th} *Conference on Embedded Networked Sensor Systems (SENSYS)*.

R. Huang and others. 2012. Real-World sensor network for long-term volcano monitoring: Design and findings. *IEEE Transactions on Parallel and Distributed Systems* 23 (2012).

C. Intanagonwiwat and others. 2002. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the* 22^{th} *International Conference on Distributed Computing Systems (ICDCS)*.

O. Iova and others. 2013. Stability and efficiency of RPL under realistic conditions in wireless sensor networks. In *Proceedings of the* 24^{th} *International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*.

V. Iyer and others. 2011. NetDetect: Neighborhood discovery in wireless networks using adaptive beacons. In *Proceedings of the* 5^{th} *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*.

A. Jhumka, M. Bradbury, and S. Saginbekov. 2014. Efficient fault-tolerant collision-free data aggregation scheduling for wireless sensor networks. *J. Parallel Distrib. Comput* 74, 1 (2014), 1789–1801.

A. Jhumka and L. Mottola. 2009. On consistent neighborhood views in wireless sensor networks. In *Proceedings of the* 28^{th} *International Symposium on Reliable Distributed Systems (SRDS)*.

A. Jhumka and L. Mottola. 2014. Neighborhood Monitoring and View Consistency Enforcement in Wireless Sensor Networks. Technical Report TR-2014-12, Politecnico di Milano (Italy). Available at home.deib.polimi.it/mottola/techRep/trPoli2014-12.pdf. (2014).

Zahid K. and others. 2015. Energizing wireless sensor networks by energy harvesting systems: Scopes, challenges and approaches. *Renewable and Sustainable Energy Reviews* 38 (2015).

A. Kandhalu and others. 2010. U-connect: A low-latency energy-efficient asynchronous neighbor discovery protocol. In *Proceedings of the* 9^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

M. Keller and others. 2012. How was your journey? Uncovering routing dynamics in deployed sensor networks with multi-hop network tomography. In *Proceedings of the* 10^{th} *Conference on Embedded Network Sensor Systems (SENSYS)*.

S. Kim and others. 2007a. Flush: A reliable bulk transport protocol for multi-hop wireless networks. In *Proceedings of the* 5^{th} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

S. Kim and others. 2007b. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the* 6^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

J. Ko and others. 2011. Beyond interoperability: Pushing the performance of sensornet IP stacks. In *Proceedings of the* 11^{th} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

K. Langendoen and others. 2006. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the* 20^{th} *International Conference on Parallel and Distributed Processing (IPDPS)*.

K. Langendoen and N. Reijers. 2003. Distributed localization in wireless sensor networks: A quantitative comparison. *Computer Nets.* 43, 4 (2003).

M.-H. Lee and Y.-H. Choi. 2008. Fault detection of wireless sensor networks. *Computer Communications Journal (Elsevier)* 31, 14 (2008), 3469–3475.

P. Levis and others. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the* 1^{st} *Conf. on Networked Systems Design and Implementation (NSDI)*.

N. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA.

A. Mahapatro and P. M. Khilar. 2013. Fault diagnosis in wireless sensor networks: A survey. *IEEE Communications Surverys and Tutorials* 15, 4 (2013).

M. Maróti and others. 2004. The flooding time synchronization protocol. In *Proceedings of the* 2^{nd} *International Conference on Embedded Networked Sensor Systems (SENSYS)*.

T. Masuzawa. 1995. A fault-tolerant and self-stabilizing protocol for topology problem.. In *Proceedings of the Workshop on Self-stabilizing Systems*.

W. P. McCartney and N. Sridhar. 2006. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the* 4^{th} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*.

L. Mottola and others. 2010. Not all wireless sensor networks are created equal: A comparative study on tunnels. *ACM Transactions on Sensor Networks* 7, 2 (2010).

L. Mottola and G. P. Picco. 2011. MUSTER: Adaptive energy-aware multi-sink routing in wireless sensor networks. *IEEE Transactions on Mobile Computing* 10, 12 (2011).

A. Munir, J. Antoon, and A. Gordon-Ross. 2015. Modeling and analysis of fault detection and fault tolerance in wireless sensor networks. *ACM Transactions on Embedded Computing Systems* 14, 1 (2015).

M. Nesterenko and A. Arora. 2002. Dining philosophers that tolerate malicious crashes. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*.

K. Ni and others. 2009. Sensor networks data fault types. *ACM Transactions on Sensor Networks* 5, 3 (2009).

F. Österlind and others. 2009. Sensornet checkpointing: Enabling repeatability in testbeds and realism in simulations. In *Proceedings of the* 6^{th} *European Conference on Wireless Sensor Networks (EWSN)*.

E. Ould-Ahmed-Vall, B. Ferri, and G. Riley. 2012. Distributed fault-tolerance for event detection using heterogeneous wireless sensor networks. *IEEE Transactions on Mobile Computing* 11, 12 (2012).

P. Pannuto and others. 2014. A networked embedded system platform for the post-mote era. In *Proceedings of the* 12^{th} *Conference on Embedded Network Sensor Systems (SENSYS)*.

L. Paradis and Q. Han. 2007. A survey of fault management in wireless sensor networks. *Journal Networked System Management* 15, 2 (2007).

B. Pásztor and others. 2010. Selective reprogramming of mobile sensor networks through social community detection. In *Proceedings of the* 7^{th} *European Conference on Wireless Sensor Networks (EWSN)*.

J. Polastre and others. 2005. A unifying link abstraction for wireless sensor networks. In *Proceedings of the* 3^{rd} *Int. Conf. on Embedded Networked Sensor Systems (SENSYS)*.

D. Puccinelli and others. 2012. Broadcast-free collection protocol. In *Proceedings of the* 10^{th} *Conference on Embedded Network Sensor Systems*.

B. Ransford and others. 2011. MementOS: System support for long-running computation on RFID-scale Devices. *SIGARCH Computer Architecture News* 39 (2011).

I. Rhee and others. 2008. Z-MAC: A hybrid MAC for wireless sensor networks. *IEEE/ACM Transactions on Networking* 16, 3 (2008).

K. Romer and J. Ma. 2009. PDA: Passive distributed assertions for sensor networks. In *Proceedings of the* 8^{th} *International Conference on Information Processing in Sensor Networks (IPSN)*.

A. Saifullah and others. 2010. Real-Time scheduling for WirelessHART networks. In *Proceedings of the Real-Time Systems Symposium (RTSS)*.

A. Saifullah and others. 2011. End-to-end delay analysis for fixed priority scheduling in WirelessHART networks. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

P. Santi. 2005. Topology control in wireless ad-hoc and sensor networks. *Comput. Surveys* 37, 2 (2005).

T. Schmid and others. 2010. A case against routing-integrated time synchronization. In *Proceedings of the* 8^{th} *Conference on Embedded Networked Sensor Systems (SENSYS)*.

A. Sharma and others. 2007. On the prevalence of sensor faults in real-world deployments. In *Proceedings of the International Conference on Sensor and Ad-hoc Communications and Networks (SECON)*.

A. Sharma, L. Golubchik, and R. Govindan. 2010. Sensor faults: Detection methods and prevalence in real-world datasets. *ACM Transactions on Sensor Networks* 6, 3 (2010).

W.-Z. Song and others. 2009. TreeMAC: Localized TDMA MAC protocol for real-time high data-rate sensor networks. *Pervasive Mobile Computing* 5, 6 (2009).

K. Srinivasan and others. 2010. An empirical study of low-power wireless. *ACM Transaction on Sensor Networks* 6, 2 (2010).

S. Sudevalayam and P. Kulkarni. 2011. Energy harvesting sensor nodes: Survey and implications. *Communications Surveys Tutorials, IEEE* 13 (2011).

B. Sundararaman and others. 2005. Clock synchronization for wireless sensor networks: A survey. *Ad Hoc Networks* 3, 3 (2005).

T. Voigt and F. Österlind. 2008. CoReDac: Collision-free command-response data collection. In *Proceedings of the International Conference on Emerging Technologies and Factory Automation (ETFA)*.

G. Werner-Allen and others. 2006. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*.

K. Whitehouse and others. 2004. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the* 2^{nd} *International Conference on Mobile systems, applications, and services (MOBISYS)*.

M. Yu and others. 2007. A Survey of fault management in wireless sensor networks. (2007).

D. Zhang and others. 2012. Acc: Generic on-demand accelerations for neighbor discovery in mobile applications. In *Proceedings of the* 10^{th} *Conference on Embedded Network Sensor Systems (SENSYS)*.