A

DICE: Monitoring Global Invariants with Wireless Sensor Networks

ŞTEFAN GUNĂ, University of Trento, Italy
LUCA MOTTOLA, Swedish Institute of Computer Science
GIAN PIETRO PICCO, University of Trento, Italy

Wireless sensor networks (WSNs) enable decentralized architectures to monitor the behavior of physical processes and to detect deviations from a specified "safe" behavior, e.g., to check the operation of control loops. Such correct behavior is typically expressed by *global* invariants over the state of different sensors or actuators. Nevertheless, to leverage the computing capabilities of WSN nodes, the application intelligence needs to reside inside the network. The task of ensuring that the monitored processes behave safely thus becomes inherently distributed, and hence more complex. In this paper we present DICE, a system enabling WSN-based distributed monitoring of global invariants. A DICE invariant is expressed by predicates defined over the state of multiple WSN nodes, e.g., the expected state of actuators based on given sensed environmental conditions. Our modular design allows two alternative protocols for detecting invariant violations: both perform in-network aggregation but with different degrees of decentralization, therefore supporting scenarios with different network and data dynamics. We characterize and compare the two protocols using large-scale simulations and a real-world testbed. Our results indicate that invariant violations are detected in a timely and energy-efficient manner. For instance, in a 225-node 15-hop network, invariant violations are detected in less than a second and with only a few packets sent by each node.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms: Algorithms, Design

Additional Key Words and Phrases: System architectures, Programming models and languages, Network protocols, In-network processing and aggregation

ACM Reference Format:
Gună, Ş, Mottola, M., and Picco, G. P. 2013. DICE: Monitoring Global Invariants with Wireless Sensor Networks *ACM Trans. Sensor Netw.* V, N, Article A (January YYYY), 33 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The untethered computing power and autonomous operation of wireless sensor networks (WSNs) enable novel applications to monitor the behavior of physical processes. A prominent example is monitoring the operation of control loops by checking that the state of the physical process remains within the safety bounds dictated by the control laws. Most often, such correct behavior is expressed by *global* invariants over the state of different computational units attached to sensors or actuators.

Authors's addresses: Ş. Gună and G. P. Picco, Dipartimento di Ingegneria e Scienza dell'Informazione Via Sommarive, 14 I-38123 Povo (TN), Italy; Luca Mottola Swedish Insitute of Computer Science, Isafjordsgatan 22, 16440 Kista, Stockholm, Sweden. Luca Mottola is now also with Politecnico di Milano, Italy, Via Golgi, 42, 20133 Milano, Italy.

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

To achieve autonomous monitoring of physical processes, a prominent trend motivated by efficiency and flexibility reasons is to leverage increasing degrees of decentralization [Stankovic et al. 2005]. As a result, WSN nodes increasingly host a significant portion of the application intelligence. Such degrees of distribution, however, drastically complicate the system operation.

Example scenarios and motivation. Consider a demand-driven building ventilation system. Sensors feed periodic CO_2 and pressure readings to a Heating, Ventilation, and Air-conditioning Controller (HVAC). This operates fans in the building to maintain the inhabitants' comfort.

An example invariant that building operators require to check concerns the relation between the state of sensors and actuators dictated by control laws. Such sample invariant can be specified as:

> (**I1**) *Whenever any sensor detects a CO_2 reading above a given threshold, at least one fan must be active.*

Nevertheless, monitoring the correct HVAC operation may be subject to further constraints. For instance, a non-uniform pressure profile in the building may indicate a malfunctioning HVAC. This condition can be checked as:

> (**I2**) *The difference in pressure between any two sampling points must remain below a given threshold.*

A different example scenario is the transportation of perishable items [Hoppough 2006]. Currently-used RFID tags only allow tracking the location of items at specific points along the supply chain. WSNs nodes may enable continuous, fine-grained monitoring of the storage conditions, both at warehouses and during transportation. For instance, when at a warehouse, the system could monitor an invariant similar to I1, stating that when temperature readings from any nodes are above threshold, at least one fan should be active. On the other hand, when packages are in a container a non-uniform load may cause stability problems during transportation. WSN nodes may be installed to ensure that the difference in load between any two sampling points in a container must remain below a threshold, similar to I2.

The example invariants above are *i) global*, i.e., they cannot be evaluated based on the state of a node alone, and *ii)* they possibly change over time. Similar needs for distributed monitoring of global invariants are likely to arise in several application scenarios, such as factory automation and health-care [Stankovic et al. 2005]. Indeed, as WSNs foster decentralized and increasingly autonomous applications, the mechanisms ensuring their correct operation under any circumstance must also become distributed. The latter vision motivates our work.

Contribution and Road-map. We present DICE (Distributed Invariant CheckEr), a system to monitor global invariants in a distributed fashion using WSN nodes. A DICE invariant is expressed by combining predicates defined over the state of multiple WSN nodes, e.g., the current CO_2 readings at any sensor node against the state of fans attached to actuator nodes. Solving this problem in general requires global knowledge of the system state [Garg and Waldecker 1994] and is further complicated by the resource-scarce nature of WSNs.

To tackle this problem for DICE invariants, we identify a reduced set of global state elements sufficient for detecting invariant violations—the *local view*. Based on this concept, we design, implement, and evaluate two distributed protocols for monitoring violations in scenarios with different environment and network dynamics, and with different degrees of redundancy in detection:

— our fully-decentralized FLAT protocol allows *any* node to detect invariant violations, achieving increased failure tolerance against node crashes. It is particularly efficient for monitoring slowly-changing processes and amenable to deployment on relatively dynamic network topologies;
— our TREE protocol induces some degree of centralization by allowing only a *subset* of nodes to detect invariant violations, but it does so very efficiently in scenarios with high churn in application data and relatively stable network topologies.

The choice of the appropriate protocol depends on the specifics of the target scenario. For instance, from a networking standpoint TREE may be the most appropriate solution in the relatively static HVAC scenario, while FLAT may be a better choice for the logistics scenario, where the handling of packages is likely to induce node churn. Nevertheless, both protocols above realize invariant monitoring efficiently: in a 225-node network with a 15-hop diameter, both FLAT and TREE detect violations in less than a second and with only a few packets sent per node.

The rest of the paper unfolds as follows. Section 2 analyzes the types of invariants we target and describes the language to specify them. Section 3 defines the local view concept and the assumptions we base our work upon, e.g., in terms of asynchronous distributed processing. Section 4 illustrates the DICE system architecture, including our dedicated tool-chain that automatically encodes invariants in their binary form and generates their corresponding local view data structures, based on the type of invariant monitored. Section 5 describes the two protocols for invariant monitoring based on the notion of local view, and discusses how the specifics of distribution, e.g., relative to communication delays, may affect their operation, along with our remedies to these issues. Section 6 reports quantitatively on the performance and correctness of the two protocols, using both simulations to verify the behavior in large-scale networks and a lab testbed to profile the traffic patterns using real-world sensed data. Section 7 contains a concise survey of related work. Section 8 ends the paper with brief remarks.

2. INVARIANTS

We describe how DICE invariants are formulated and the declarative language we design to specify them.

2.1. Formulation

DICE invariants are combinations of *atomic predicates*[1] over variables whose value is a function of network nodes. A DICE invariant is thus of the form:

$$Q_1 x_1, \ldots Q_r x_r : P_1(x_1, \ldots, x_r) \circ \ldots \circ P_s(x_1, \ldots, x_r)$$

where $Q_i \in \{\forall, \exists\}$ $(1 \leq i \leq r)$, $P_j(x_1, \ldots, x_r)$ $(1 \leq j \leq s)$ is an atomic predicate whose value depends on variables at nodes $x_1, \ldots, x_r$, and $\circ \in \{\wedge, \vee, \rightarrow\}$. Note that existential and universal quantifiers apply *only* to terms representing network nodes. These, however, are solely used for addressing, and do not directly concur to determining the truth value of an invariant. As such, a DICE invariant is simply analogous to a Boolean condition in mainstream programming languages, with variables possibly residing on different nodes.

We specifically focus on two types of global invariants characteristic of our target applications:

[1]An atomic predicate is one with no deeper propositional structure [Whitesitt 1995]. This entails that a predicate belonging to a DICE invariant cannot be further decomposed in more elementary predicates.

— in *local-predicate (*LP*) invariants*, each predicate involves only one node variable. The invariant resulting from combining the predicates with $\circ \in \{\wedge, \vee, \rightarrow\}$ is however global. LP invariants can be equivalently written as:

$$Q_1 x_1 : P_1(x_1) \circ \ldots \circ Q_r x_r : P_s(x_r), \quad r = s.$$

— in *distributed-predicate (*DP*) invariants* there is only one predicate (i.e., $s = 1$) that involves multiple node variables. DP invariants can be equivalently written as:

$$Q_1 x_1, \ldots, Q_r x_r : P(x_1, \ldots, x_r).$$

In practice, undesired deviations from specified behaviors are often expressed as comparisons against known thresholds [Lipták 1995], as in our motivating scenarios in the introduction. Therefore, we consider invariants where $P(x_1, \ldots, x_r)$ is a Boolean predicate or an inequality $f(x_1, \ldots, x_r) < T$ or $f(x_1, \ldots, x_r) > T$, where f is a linear function of the values of variables at $x_1, \ldots, x_r$ and T is a numerical constant.

2.2. Invariant Language

We design a declarative language to specify DICE invariants. The language is input to a dedicated tool-chain, described in Section 4, which automatically generates the data structures and executable code necessary for their distributed monitoring.

Attributes. Physical processes are monitored through WSN nodes, which directly report either sensed data or the state of actuators. DICE *attributes* conceptually link the WSN node and the physical process under control. Each node is characterized by one or more attributes, each a typed mapping between a name and value. Different nodes can have different attributes. DICE supports three kinds of attributes. *Constant* attributes are set by the programmer and remain unchanged. In the HVAC scenario,

```
attribute int type = FAN;
```

declares an attribute representing the type of node, in this case one controlling a fan. The value of *periodic* attributes, instead, is automatically updated by the system at a programmer-specified rate. For instance,

```
attribute int co2 every 3;
```

declares an attribute for a CO_2 reading, whose value is refreshed every 3 s. However, polling the value of slow-changing attributes can be inefficient. Therefore, we also allow declarations such as

```
attribute bool isActive on event;
```

where an update to the active status (e.g., of a fan) is triggered by the control logic, outside DICE, as discussed in Section 4.

Periodic and event-triggered attributes give developers the knobs to trade resource consumption, e.g., in terms of processing, against the latency to make the state of a physical phenomena available to DICE. The latter has an impact on the latency to detect invariant violations, as these can be detected only once DICE is aware of the environment state. We discuss these aspects and their implications in Section 3.

Invariants. DICE invariants are specified using existential or universal quantifiers over WSN nodes and the `@` operator to select which of a node's attributes is referenced in the invariant. As an example, the invariants in the introduction can be specified as:

```
invariant I1 {forall m: type@m = CO2 and co2@m > T
           -> exists n: type@n = FAN and isActive@n}

invariant I2 {forall m,n: pressure@m - pressure@n <T}
```

invariant = **invariant** *inv-name* **{** *predicate-exp* **}** [*tolerate-exp*] ;
predicate-exp = (*predicate* | **not (***predicate***)**) [(**and** | **or** | **->**) *predicate-exp*] ;
tolerate-exp = **tolerate** *const time-unit* [**for** *const* **times in** *const time-unit*] **;** ;
predicate = (**forall** | **exists**) *node-variable* : bool-exp
| (**forall** | **exists**) *quant-list* **:** *linear-exp* (**<** | **>**) *const* ;
bool-exp = *term* [(**=** | **<** | **>** | **!=**) *const*] [(**and** | **or** | **->**) *bool-exp*] ;
quant-list = *node-variable* [**,** *quant-list*] ;
linear-exp = [**−**] ([*const* *****] *term* | *const*) [(**+** | **−**) *linear-exp*] ;
term = *attribute-name* **@** (*node-variable* | *const*) ;

Fig. 1. Invariant language grammar.

Notably, `I1` is an example of `LP` invariant: both the antecedent and the consequent in the logical implication individually refer to a single node, `m` and `n` respectively. On the other hand, invariant `I2` is of type `DP`, as the only predicate in the invariant checks an inequality between the values of attributes at different nodes.

Figure 1 shows the grammar for the invariant language, which determines the invariants one can possibly specify. DICE invariants can have an arbitrary number of node variables. Invariants can also refer to a specific node by using its identifier, as in `co2@52` for node `52`. Besides the quantifiers `forall` and `exists`, the usual logical operators `and`, `or`, `not` also apply.

An example invariant the grammar in Figure 1 prevents from specifying is $\forall m, n : x@m + x@n = T$. Checking violations to this invariant indeed requires non-polynomial time, as it is an instance of the NP-hard Boolean satisfiability problem [Whitesitt 1995], and would likely be prohibitively expensive to solve in a distributed fashion on a network of resource constrained nodes. Similar observations hold for $\forall m, n : x@m + x@n \neq T$, which the grammar also prevents.

Transient violations. Short-term invariant violations are inevitable in some scenarios. For instance, a sudden gathering of people may trigger a violation of I1 before the fan is activated. However, if the HVAC operates correctly, the CO_2 readings should eventually return below `T`. Using the `tolerate` clause, DICE allows to specify transient deviations along two dimensions: time and occurrences. For instance,

```
invariant I1 {forall m: type@m = CO2 and co2@m > T
           -> exists n: type@n = FAN and isActive@n}
              tolerate 10 min for 5 times in 24 h;
```

is a variation of invariant I1 where the CO_2 readings are allowed to cross the threshold for at most 10 minutes, supposedly enough for the HVAC to react. However, this transient violation should not happen repeatedly, as this may indicate a failure in the control system. Thus, the invariant also specifies that transient violations are allowed at most five times per day, e.g., based on the building usage patterns.

3. MONITORING INVARIANTS

DICE invariants express properties that must hold *simultaneously* at multiple nodes. Temporal operators, allowing relations among states occurring at different times, are intentionally not included in the grammar in Figure 1. Their monitoring would require tracking the environment evolution over time, which clashes with the memory limitations of typical WSN nodes.

Moreover, in abstract terms, an invariant violation is determined when a given combination of values of *physical* quantities simultaneously occurs in the environment. However, in practice, these physical values become available to *any* software monitor-

ing system only when they are *observed* through some device (e.g., a sensor) and stored as program data—in the case of DICE, the node attributes. Our goal is to design a solution that efficiently detects invariant violations, determined by changes in the DICE attribute values at multiple nodes, if and only if they occur.

Based on the formulation above, detecting invariant violations can be regarded as an instance of distributed predicate detection [Garg and Waldecker 1994]. In a perfectly synchronized system with infinite communication and processing resources, a centralized solution that maintains global knowledge about the system state suffices to solve the problem [Garg and Waldecker 1994]. On the other hand, in absence of specific assumptions on system delays, absolute correctness—a system capturing invariant violations if and only if they occur—is impossible to achieve [Garg and Waldecker 1994].

WSNs are asynchronous systems, characterized by scarce communication and processing resources, and non-deterministic system delays. Therefore, WSNs pose multiple challenges:

(1) Maintaining global state knowledge is prohibitively expensive, e.g., in terms of energy for communication, as any change of state at any node in the network may require communication to a central unit.
(2) Due to the asynchronous nature of WSNs, node failures are generally not decidable. Besides potentially leading to missed invariant violations, this generally causes problems also for detecting when an invariant is complied with.
(3) Non-negligible delays, e.g., in sensing and communication, may prevent the system from capturing all and only the invariant violations (as discussed above), yielding both false positives and false negatives.

To tackle challenge (1), we reduce the amount of global information necessary at each node by defining a notion of *local view*, i.e., a reduced set of information enabling *local* detection of *global* invariant violations. In this sense, the local view represents a slice of the global system state sufficient for checking global invariant violations. In the rest of this section, we describe how the local view is populated at any given node, depending on two types of invariants: local-predicate (LP) and distributed-predicate (DP) invariants. Due to their specific structure, LP invariants allow for a monitoring strategy, described in Section 3.1, that relies on the computation and dissemination of Boolean values representing the value of predicates local to nodes. DP invariants, on the other hand, require the processing and dissemination of individual attribute values, as illustrated in Section 3.2.

Maintenance of the local view at each node, however, must be complemented by protocols performing the efficient dissemination of its relevant changes, further complicated by the presence of faults and non-deterministic system delays. To put the dissemination protocols in context, we precede their discussion with the illustration of the DICE compiler and run-time support in Section 4, as these aspects are key to understand the way dissemination is performed. Our protocols, described in Section 5, address challenge (2) by striking different trade-offs w.r.t. the resilience to node and communication faults, and tackle challenge (3) with mechanisms that allow the reconstruction of the order of events, greatly reducing the impact of communication delays on the correctness of detection of invariant violations. We quantitatively assess the impact of our design choices in Section 6.

3.1. Local-predicate (LP) Invariants

We exemplify the monitoring of LP invariants to provide an intuition of how the local view can be efficiently populated. Next, we formally present the general algorithm.

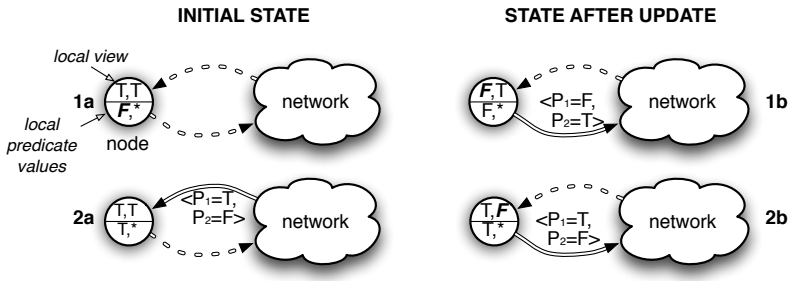


Fig. 2. Local view processing for a LP invariant $\forall m, n : P_1(m) \circ P_2(n), \circ \in \{\wedge, \vee\}$. Value changes are shown in bold. The node considered in the picture only affects the truth value of P_1 and it is excluded from monitoring P_2, e.g., as it happens for a CO_2 sensor node monitoring $type@n = FAN \wedge isActive@n$.

Intuition. For simpler illustration, consider a slight variation of invariant I1 where only universal quantifiers are used:

$$(\forall m : type@m = CO2 \ \wedge \ co2@m > T) \Rightarrow (\forall n : type@n = FAN \ \Rightarrow \ isActive@n)$$

To detect violations of LP invariants and accordingly define the content of the local view, we take inspiration from the notion of *communication silence* [Zhu and Sivakumar 2005]. Let us assume that the example invariant above holds. Then, every node attached to a CO_2 sensor does not send any information to other nodes as long as its reading is *above* T. The rest of the system implicitly takes the "collective silence" of CO_2 nodes as an indication that the predicate they monitor holds true. Similarly, nodes controlling a fan do not send any information as long as the fans are *active*, which implicitly indicates that every such node is currently operating the fan. Therefore, if the entire system remains silent, it means that the invariant is complied with.

Whenever either the CO_2 reading drops below T or a fan becomes inactive, the corresponding node notifies this event. Breaking the silence reveals a change in the truth value of some predicate. Since the two predicates in I1 are joined by implication, the invariant can be violated only when a notification arrives from a node controlling a fan (i.e., there exists at least one inactive fan) while CO_2 nodes remain silent (i.e., their reading is still above threshold).

In general, based on the logical operators connecting two predicates $\forall x_i : P_h(x_i)$ and $\forall x_j : P_k(x_j)$, a violation is detected when *i)* the two predicates are in disjunction and both node x_i and x_j send a notification that the corresponding predicate has become false; or *ii)* the two predicates are in conjunction and either node x_i or x_j sends a notification. In all other cases the invariant is complied with. Existential quantifiers and logical implications are straightforwardly mapped to the cases above using known transformations of logical formulae [Whitesitt 1995]. Note, however, that the system performance is ultimately dictated only by the content of the local view, which is independent of the predicate representation.

To realize this scheme, the local view for LP invariants contains, for each predicate, a Boolean value representing its truth value.

Algorithm. The technique above is applicable to any LP invariant. Consider, without loss of generality as mentioned above, predicates of the form $\forall x_i : P_h(x_i)$.

Figure 2 illustrates the interplay between the local view on a node and the rest of the network, focusing on the local view processing for a sample monitored invariant $\forall m, n : P_1(m) \circ P_2(n), \circ \in \{\wedge, \vee\}$. For the sake of illustration, we assume that the node in the picture can affect the value of P_1, but not of P_2, e.g., because it is a CO_2 sensor

Algorithm 1: Monitoring LP invariant $\forall x_1 : P_1(x_1) \circ \ldots \circ \forall x_s : P_s(x_s)$ on node m. LV_m is the local view at node m. $LV_m[i]$ is the i^{th} entry in the local view, corresponding to predicate P_i. The functions `value(e)` and `source(e)` return the Boolean value and the node identifier in local view entry `e`, respectively. Function `disseminate()` triggers the dissemination of local view changes, using the dissemination strategies illustrated in Section 5. Function `evaluate()` checks whether the current local view determines an invariant violation. The size of the local view is lv_size.

```
1  event onAttributeValueChange(attribute a, value v)
2      checkLocalAttributeValue(a, v);
3      if LV_m updated then disseminate(LV_m); evaluate();
4  event onLocalViewReceive(local view LV_n received from node n)
5      for i ⟵ 1 to lv_size do
6          if value(LV_m[i]) = true and value(LV_n[i]) = false or
7            source(LV_m[i]) = source(LV_n[i]) then
8              LV_m[i] ⟵ LV_n[i];
9              foreach attribute a and local value v for a do
10                 checkLocalAttributeValue(a, v);
11     if LV_m updated then disseminate(LV_m); evaluate();
12 procedure checkLocalAttributeValue(attribute a, value v)
13     foreach predicate P_i ∈ {P_1, ..., P_s} that references a do
14         if value(LV_m[i])=true and P_i|_{a=v} = false or source(LV_m[i])= m then
15             value(LV_m[i]) ⟵ P_i|_{a=v};
16             source(LV_m[i]) ⟵ m;
```

and therefore does not have a fan attached[2]. This invariant requires a local view consisting of two Boolean values, one for each predicate. Assume the invariant initially holds. The node remains silent as long as predicate P_1 holds. When a change occurs, two cases are possible:

(1) the truth value of the locally monitored predicate P_1 changes from true to false (1a). In this case, the local view is updated and propagated to the rest of the network (1b);
(2) a local view update from the network arrives, e.g., reporting that the value of the predicate P_2 monitored by a remote node changed from true to false (2a). Again, the local view is updated and propagated to the rest of the network (2b).

Every time the local view changes, a node also re-evaluates the global invariant to check whether the new information determines a violation.

Algorithm 1 illustrates the generic processing for a node m. Whenever a local attribute a changes to a value v, we execute the procedure `checkLocalAttributeValue` (line 1). This procedure checks, for every predicate P_i that references the attribute a given as parameter (line 13), whether (line 14) *i)* value v yields P_i false while the corresponding local view entry is true, or *ii)* m is the node that last updated the value of P_i in the local view. The first condition is the case where m "breaks the silence": the discrepancy between the value of P_i in the local predicate (false) and in the local view (true) allows node m to set the corresponding entry in the local view to false, regardless of which node last modified it. In the second condition, node m is the node that last changed a value in the local view, and is thus responsible for the same entry at other nodes, irrespective of the truth value. In both

[2]The cases when a predicate is trivially false on a node because of values of constant attributes are automatically identified by the DICE compiler and excluded from processing.

cases, we update the local view with the new value for P_i and m as the associated node identifier (line 15-16). If these operations cause a change in the local view, we trigger the dissemination of the latter and check whether the new local view content determines an invariant violation (line 3).

Upon receiving a local view update from another node n (line 4), node m checks its local view (line 5-10) searching for entries that either became false based on information received from the network, or that were updated by the node that last changed them (line 6-7). The two conditions mirror those of line 14 in a distributed setting. The former condition is the case of a *remote* node "breaking the silence" because it locally detected a predicate to be false; the latter is the case of a remote node responsible for an entry in the local view, regardless of the truth value. In both cases, m updates the local view (line 8). In addition, node m checks all attributes a against their local values (lines 9-10). This is necessary whenever an entry in the local view is reverted to true by a remote node responsible for it, but the values at node m still force the same entry to hold false. Due to the condition in line 14, the (local) false value immediately replaces the one value in the local view, necessarily causing another dissemination of the new local view where node m now responsible for the updated entry (line 11). As before, the dissemination is triggered by any change in the local view, along with a check of potential invariant violations.

Note that generally a node remaining silent may also indicate failure of that node. There is no remedy to this if the failed node is the only one allowing a violation to be detected. This is unlikely in the scenarios we target, where WSN nodes are expected to be deployed in large numbers, thus providing redundancy, and the monitored phenomena span significant portions of space, thus involving several nodes.

3.2. Distributed-predicate (DP) Invariants

As illustrated above, monitoring violations of LP invariants can be achieved by disseminating solely the truth value of the constituting predicates. We can determine such value locally, as each predicate is a function of only one node variable.

This observation does not apply to DP invariants, whose (single) predicate involves multiple node variables, in a sense generalizing the predicates seen in LP invariants. As result, the truth value of the predicate can no longer be determined locally, and dissemination of the actual attribute values becomes necessary.

For this case, as above, we start by presenting an example to quickly grasp the rationale behind our strategy and how we build the local view. The general algorithm is presented next.

Intuition. Consider invariant I2:

$$\forall m, n : pressure@m - pressure@n < T$$

To detect violations, one should consider all combinations of pressure readings at any two nodes. This is unnecessary if one identifies the *worst-case* combination, i.e., the two nodes corresponding to the highest pressure difference. If this is below the threshold, then the invariant is complied with, because any other pair of nodes has a smaller pressure difference. Otherwise, the invariant is violated. In the case of I2, the highest pressure difference is determined by the two nodes sensing the maximum and minimum pressure.

The local view for a DP invariant thus needs to include, for every involved attribute, the number of system-wide maximum and minimum values necessary to determine the worst-case combination. If these values do not determine a violation, the invariant is necessarily complied with. The key idea is not new per se: in a sense, comparing the worst-case combination of maximum and minimum values at n nodes against a threshold resembles the application of discrepancy functions to $(n + 1)$-dimensional

Quantifier	Threshold	Local view
$\forall x_i$	$f(x_1, \ldots, x_i, \ldots, x_n) < T$	$max(x_i)$
	$f(x_1, \ldots, x_i, \ldots, x_n) > T$	$min(x_i)$
$\exists x_i$	$f(x_1, \ldots, x_i, \ldots, x_n) < T$	$min(x_i)$
	$f(x_1, \ldots, x_i, \ldots, x_n) > T$	$max(x_i)$

Fig. 3. DP invariants: information to insert in the local view for a positively correlated attribute x_i.

spaces [Tsitsiklis 1984]. However, while the latter aims at analytically describing how closely a structural model matches the observed data, we use a similar concept to capture global invariant violations with a reduced set of data—the local view—in the energy-constrained setting of WSNs.

Algorithm. The intuition above can be generalized to any DP invariant. Initially, let us consider universally quantified invariants $\forall x_1, \ldots, x_n : f(x_1, \ldots, x_n) < T$. First, we determine if the attributes at nodes $x_1, \ldots, x_n$ are positively correlated with f, i.e., if the evaluation of f increases when the attribute value increases. For instance, in I2 $pressure@m$ is positively correlated. Similarly, $pressure@n$ is negatively correlated, i.e., a decrease in that term causes an increase in f. Based on this information, each node builds a local view containing the network-wide maximum (minimum) values for positively (negatively) correlated attributes. This is sufficient to determine an invariant violation. The same technique is straightforwardly applied when the invariant requires f to be *above* a threshold.

When node variables are existentially quantified, rather than identifying the nodes that violate a property as we do in the case of universally quantified variables, we aim instead at identifying the nodes that comply with the monitored property. Consider $\forall m, \exists n : x@m + y@n < T$. Given the network-wide maximum of x, this invariant is satisfied if we can find a node n where $x@m + y@n < T$. To determine the worst-case combination of nodes m and n, it is sufficient to identify the network-wide minimum value for y. If this value is such that $x@m + y@n \geq T$ when x is maximum, there exists no other node in the network where the value of y satisfies the invariant, therefore we detect a violation.

Figure 3 summarizes the mapping among the quantifiers, the function f, and the information included in the local view w.r.t. a positively correlated attribute. This information is sufficient to detect violations of DP invariants. The case of negatively correlated attributes is dual.

Figure 4 exemplifies for DP invariants the interplay of the local view on a node and the rest of the network, similar to Figure 2. We consider a sample monitored invariant $\forall m, n : x@m + x@n < T$, which requires a local view including the two network-wide maxima of attribute x. Four cases are possible:

(1) a local value update does not affect the current local view (1a). In this case, no further processing is needed. The local view on the other nodes can indeed remain the same, and no communication is required (1b);
(2) a local value update must replace a value in the local view (2a). In this case, the local view is updated and propagated to the rest of the network through the dissemination manager (2b);
(3) the values in a local view update from the network do not affect the current local view (3a). No further processing is required in this case, as the local view remains unaltered (3b);
(4) a local view update from the network carries at least one value that must replace an entry in the current local view (4a). The new values are merged into the local view and the latter is disseminated further to reach system-wide convergence (4b).

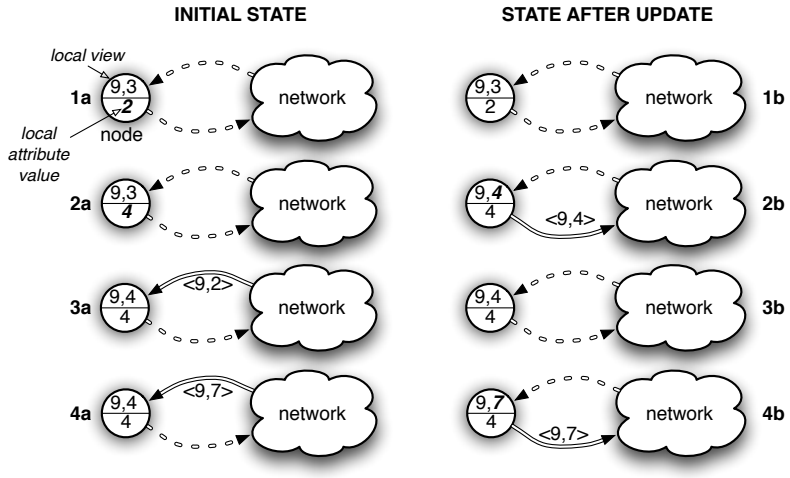


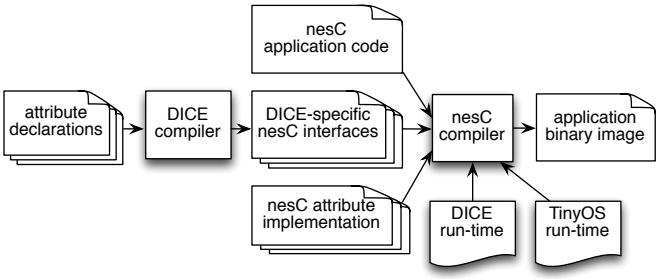
Fig. 4. Local view processing for a DP invariant $\forall m, n : x@m + x@n < T$ whose local view includes the two network-wide maxima of attribute x. Value changes are shown in bold.

Algorithm 2: Monitoring of DP invariant $\forall x_1, \ldots, x_s : \sum_{i=1}^{s} a@x_i < T$ on node m, where a is an attribute and T is a constant. $LV_m[i]$ is the i^{th} entry in the local view for a given attribute a. The meaning of `value()`, `source()`, `disseminate()`, `evaluate()`, and lv_size are the same as in Algorithm 1. Additionally, `min`(LV_m) returns the local view entry whose value is minimum.

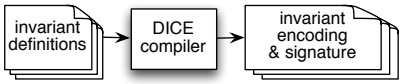
```
1  event onAttributeValueChange(attribute a, value v)
2      updateLocalView(a, v, m);
3      if LV_m updated then disseminate(LV_m); evaluate();
4  event onLocalViewReceive(local view LV_n for attribute a received from node n)
5      for i ⟵ 1 to lv_size do updateLocalView(a, value(LV_n[i]), source(LV_n[i]));
6      if LV_m updated then disseminate(LV_m); evaluate();
7  procedure updateLocalView(attribute a, value v, source n)
8      for i ⟵ 1 to lv_size do
9          if n = source(LV_m[i]) then
10             value(LV_m[i]) ⟵ v; return;
11     if v > value(min(LV_m)) then
12         value(min(LV_m)) ⟵ v; source(min(LV_m)) ⟵ n;
```

Algorithm 2 describes the generic processing for DP invariants at a node m, for the case where the s maximum values of attribute a are included in the local view[3]. We distinguish the case where the attribute value change originates locally (lines 1-3) and when local view updates are received from the network (lines 4-5). In both cases, the local view is first updated with the proper source node identifier (lines 2 and 5, respectively), and in case of changes to the local view we trigger the local view dissemination

[3]The further generalization of Algorithm 2 for different attributes a_i and the case of negative correlation do not present technical complexity, but their presentation would be tedious and lengthy without contributing any additional insight. Therefore, we omit them for brevity.



(a) Attributes and application.



(b) Invariants.

Fig. 5. The DICE tool chain.

and check whether the change determines a violation (lines 3 and 6, respectively). The core of the processing occurs inside `updateLocalView` (lines 7-12). For each local view entry, we update the corresponding value in case the node n given as parameter is found among those contributing to the local view (lines 8-10). In addition, and most importantly, if the value v given as parameter is greater than the minimum value in the current local view, i.e., the smallest of the maxima stored in it (line 11), v must replace the latter (line 12).

4. SYSTEM ARCHITECTURE

We describe the tool-chain and run-time architecture of DICE. Our prototype targets TinyOS [Hill et al. 2000], and relies on CTP [Gnawali et al. 2009] for the tree-based forwarding necessary to the TREE dissemination strategy. Nevertheless, the techniques we describe do not depend on either.

4.1. Compiler

In our example scenario, invariant I1 instructs DICE to check whether the fan is active when CO_2 readings at any sensor node are above a threshold. However, the control logic actually operating the fan runs within the application code of the actuator node, external to DICE. Attributes connect DICE with the application, enabling the former to become aware of the status of the latter relevant to invariant monitoring.

Figure 5 outlines the tool-chain supporting this design. As illustrated in Figure 5(a), the DICE compiler generates one nesC interface for each attribute declaration, thus providing the connection between the source of attribute values and the DICE run-time. The latter accesses the interface through a compiler-generated component that periodically polls data from it or, for `on event` attributes, awaits the signaling of the corresponding events. The attribute interfaces, the components providing these interfaces, the DICE run-time, and the TinyOS libraries are input to the nesC compiler, which yields the binary image to upload on the nodes.

Invariant specifications do not require integration into a binary image. Thus, they can be changed freely *without* reprogramming the WSN nodes. To support this design, as shown in Figure 5(b), the compiler generates a *binary encoding* for every invariant. This describes the logical relations between the predicates in an invariant, used

Algorithm 3: Generating the signature of a DP invariant $\forall x_1, \ldots, x_r : P(x_1, \ldots, x_r)$: S is the invariant signature, composed of a tuple $\langle h, (\texttt{pos}(h), \texttt{neg}(h))\rangle$ where h is an attribute name and $\texttt{pos}(h)$ ($\texttt{neg}(h)$) is the number of times attribute h appears positively (negatively) correlated with the function f in P, as described in Section 3.2. $\texttt{pos}(h)$ $\texttt{neg}(h)$ are initially zero for all attributes.

```
1  function signature(P)
2      S ⟵ ∅;
3      distribute quantifiers in P;
       /* e.g., transform "c · (h@m − k@n)" to "c · h@m − c · k@n"        */
4      foreach h referenced by P, any node m, any quantifier c do
5          if c = ∀ then
6              pos(h) ⟵ pos(h) + number of terms "h@m" preceded by '+';
7              neg(h) ⟵ neg(h) + number of terms "h@m" preceded by '−';
8          if c = ∃ then
9              pos(h) ⟵ pos(h) + number of terms "h@m" preceded by '−';
10             neg(h) ⟵ neg(h) + number of terms "h@m" preceded by '+';
11         S ⟵ S ∪ {⟨h, (pos(h), neg(h))⟩}
12     return S;
```

to check violations given the truth value of the predicates themselves. The compiler encodes invariants using their postfix notation, with names replaced by numerical identifiers to reduce space. The run-time relies on a simple stack machine to interpret invariants encoded this way.

For DP invariants, the compiler also automatically generates an invariant *signature*, which determines what values need to be collected to detect violations. Algorithm 3 describes how the signature is generated. Consider invariant $\forall x, z\ \exists y : a@x + a@y - a@z < T$ as an example. Following a preprocessing step (line 3) where quantifiers are moved close to the quantified terms, for every attribute h appearing in P we search for the number of times h appears with a plus or minus sign in the linear function f in P (lines 4-11), as described in Section 3.2. In case of universal quantifiers, a plus (minus) sign indicates positive (negative) correlation (lines 5-7); the dual applies for existential quantifiers (lines 8-10). For our example, Algorithm 3 finds one positively correlated term ($a@x$) and one negatively correlated term ($a@z$) corresponding to universal quantifiers. For existential quantifiers, $a@y$ is the only negatively correlated term. The occurrence of plus and minus signs depending on the quantifier is progressively summed up to build the signature (line 11), which is finally returned when all attributes are examined (line 12). For our example, the compiler generates a signature ($\langle a, (1, 2)\rangle$). Such signature instructs the DICE run-time to collect the single maximum and the two minimum values of a throughout the network.

4.2. Run-time Layer

Figure 6 illustrates the main components of the DICE run-time. At the core is a data structure implementing the local view as defined in Section 3. Every local view entry is associated to a tuple ⟨*name, value, source, timestamp*⟩. The *name* field refers to a predicate for LP invariants, or to an attribute for DP invariants. The type of invariant also determines the content of the *value* field, which stores a Boolean value for LP invariants or a numerical value for DP invariants. The *source* field keeps track of the node that originated the associated local view entry, whereas the *timestamp* field is used by the dissemination manager to ensure a correct processing of local view updates, as described in Section 5.

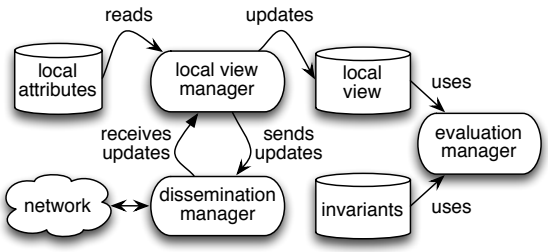


Fig. 6. The architecture of the DICE run-time.

The evaluation manager checks the local view against user-specified invariants, determining their violations. The local view manager determines the appropriate changes to the local view, based either on value changes of the local attributes, or updates received through the network. The latter are performed according to the protocol in the dissemination manager. The same protocol is used to disseminate local view updates, regardless of the invariant type.

Evaluation manager. A change in the local view triggers the evaluation manager to check if any of the monitored invariant is violated, using the invariant encoding. If the current local view indicates a violation and the invariant does not include a `tolerate` clause, the evaluation manager immediately generates a notification. Otherwise, the evaluation manager starts a timer with duration equal to the tolerance period. If a subsequent local view change brings the invariant back to compliance before the timer expires, no violation is notified. Otherwise, the violation is notified as if it were detected at the end of the tolerance period.

Local view manager. Changes to the local view are determined by the local view manager, which executes the algorithms described in Section 3 depending on the type of invariant. The changes are caused either by a local update (i.e., an attribute value change) or by a remote update (i.e., attribute value changes from the network). In turn, these changes may require further dissemination of updates, as dictated by the monitoring algorithms. The dissemination manager, illustrated next, takes care of this.

The current implementation allows monitoring of multiple invariants, whose local view is however disseminated independently. This strategy could be improved upon in the cases where the invariants share some attributes, by eliminating redundant transmissions of the local view updates. The opportunity for such optimization can be automatically detected off-line by the compiler, which would generate a single master signature for all monitored properties.

Dissemination manager. The next section describes two protocols to disseminate local view updates—i.e., determining how they are propagated inside the "network" bubbles of Figure 2 and Figure 4. The FLAT protocol, described in Section 5.1, is designed to cope with failure-prone scenarios. In FLAT, all nodes can detect violations, achieving increased reliability through redundancy. The TREE protocol, described in Section 5.2, is optimized for scenarios with high variability in the application data. In these scenarios, TREE significantly reduces the communication overhead w.r.t. FLAT, at the expense of limiting the detection of violations only to a subset of nodes. Finally, Section 5.3 describes how these protocols deal with the challenges posed by communication delays.

5. LOCAL VIEW DISSEMINATION: PROTOCOLS

In this section we describe the design of two protocols for efficiently disseminating the local view updates enabling global invariant evaluation in DICE. Both protocols ad-

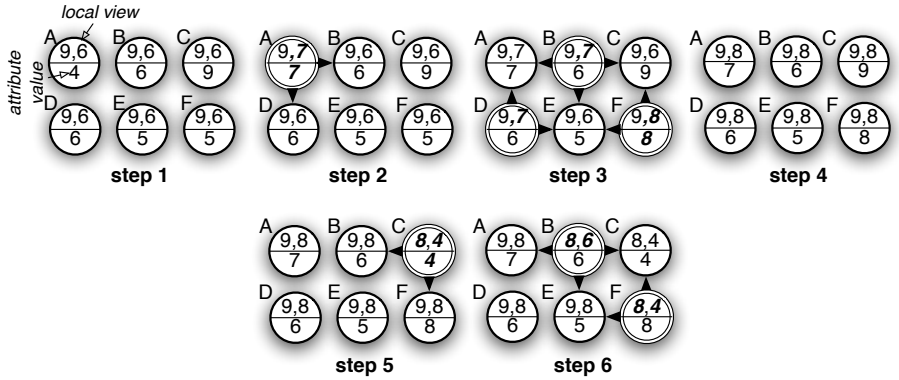


Fig. 7. Disseminating local view updates in FLAT.

dress challenge (2) in Section 3 by striking different trade-offs w.r.t. node and communication failures. In doing so, we target networks of static or quasi-static WSN nodes, where node mobility happens rarely—if at all—as representatives of the applications we target. Highly mobile settings, on the other hand, require different solutions as they introduce new challenges; for example, how to keep track of whether a node contributing a network-wide maximum remains reachable as it roams around. We conceived preliminary solutions for these scenarios as well [Gună 2011], omitted here because of space constraints. We conclude this section by discussing how we address the impact of communication delays on the overall correctness of DICE, tackling challenge (3) in Section 3.

5.1. FLAT

To allow any node to detect violations, all nodes must eventually agree on the most recent local view. To achieve this, we disseminate every local view update to the entire network. This functionality resembles well-known dissemination protocols [Levis et al. 2004; Lin and Levis 2008]. However, these are usually designed to propagate data from a *single* source, one data at a time. In DICE, every node is a possible source and multiple dissemination processes triggered at different nodes may overlap in time. This prevents off-the-shelf re-use of existing protocols.

To tackle the problem in our specific scenario, we adapt the polite gossiping technique [Levis et al. 2004]. At each node, the dissemination manager periodically broadcasts the current local view. It also receives local view updates from other nodes, storing them in a *network cache* containing the last received local view. As long as the network cache is the same as the local view, the broadcast period increases exponentially up to a maximum τ_h to reduce traffic once the network has reached convergence. Otherwise, the dissemination manager informs the local view manager of new data from the network. The local view manager determines whether the received information should be merged with the current local view, according to the processing described in Section 4.2. If so, re-propagation occurs with the broadcast period reset to the minimum τ_l, to speed up dissemination. A node suppresses the periodic broadcast if at least γ neighbors already broadcast the same information. The values for τ_l, τ_h, and γ are set depending on application scenario and network layout.

Update propagation. We deal with concurrent dissemination processes from different nodes by merging local view updates as they propagate. Consider Figure 7 as an example, again with the monitored invariant $\forall m, n : x@m + x@n < T$. We assume all

nodes have the same local view $\langle 9, 6\rangle$ in step 1. In step 2, node A changes its local value to 7 and propagates an updated local view $\langle 9, 7\rangle$, which reaches node B and D. While these further rebroadcast the update, the value of x at F jumps to 8, as in step 3. The two updates originating at A and F "compete" for the propagation. The local view manager stops the propagation of the smaller update from A wherever the larger update from F has already been processed. This would happen at E, if the update from F is received before those from B or D. As a result, the larger value overcomes the other, providing eventual consistency of local views, as in step 4.

Note that, if we were to use "as is" an existing dissemination protocol (e.g., DIP [Lin and Levis 2008]) every node would disseminate one of the two new values in Figure 7. Indeed, these protocols are based on a globally-consistent version number. The dissemination of the two updates would occur with the same version number. Without the ability to process local view updates as they propagate, intermediate nodes would propagate either of the two values, possibly causing inconsistencies. Although inconsistencies can be dealt with by the source of the larger value when it recognizes that its update did not propagate throughout the network (and repeats the dissemination), this would cause higher network overhead. Thus, a data-agnostic protocol such as DIP would likely generate higher traffic than our solution.

Value deletion. The above processing is not sufficient for nodes whose state belongs to the current local view. The last two steps of Figure 7 illustrate the problem. In step 5, the value of x at C changes from 9 to 4. Contrary to the previous case, here one of the values in the local view disappears as a consequence of a state change. In this case, based on local information, C determines that the new maxima are 8 and 4. If this local view at C, LV_C, were propagated as described above, the system would reach an inconsistent state. Indeed, according to the local view information stored in the runtime layer as $LV_C = \langle\langle x, 8, F, t_4\rangle, \langle x, 4, C, t_5\rangle\rangle$, with $t_5 > t_4$, B and F would infer that the value of x at C dropped from 9 to 4. However, once this information is merged with their local views, yielding $LV_B = LV_F = \langle\langle x, 8, F, t_4\rangle, \langle x, 6, B, t_0\rangle\rangle$ and rebroadcast, the receiving A, D, E would obtain no information on the last update at C, and incorrectly conclude that $\langle x, 9, C, t_1\rangle$ ($t_1 > t_0$) is still valid.

We address the problem by appending an *eviction entry* to local view updates, enabling the removal of stale values. The entry is a tuple $\langle$*attribute, source, timestamp*$\rangle$ re-propagated at each hop along with the local view update. In our example, the eviction entry $\langle x, C, t_5\rangle$ allows nodes to determine that entry $\langle x, 9, C, t_1\rangle$ is no longer valid, and eventually converge to $\langle\langle x, 8, F, t_4\rangle, \langle x, 7, A, t_2\rangle\rangle$.

We use the periodic broadcasts of the current local view also to determine if a node is unreachable and its state should be evicted. When a node A misses a given number of consecutive broadcasts from a neighbor B contributing to the local view, A assumes that B failed[4]. Then, A recomputes its local view and propagates it along with an eviction entry, with the same structure and processing discussed earlier. In this case, however, the entry $\langle *, B, t\rangle$ causes the eviction of all attribute values provided by the crashed node. A node joining (or re-joining after failure) starts with an empty local view, eventually made consistent through the periodic broadcast process.

5.2. TREE

In scenarios characterized by high variability in application data, the FLAT protocol is likely to exhibit a significant communication overhead. Therefore, in the TREE protocol we leverage a tree-shaped routing topology to propagate local view updates. In TREE, each node pushes updates only upwards, towards the root, rather than to the entire

[4]To prevent nodes from erroneously considering neighbors as failed because of the broadcast suppression mechanism above, nodes contributing to the local view never suppress their own broadcast transmissions.

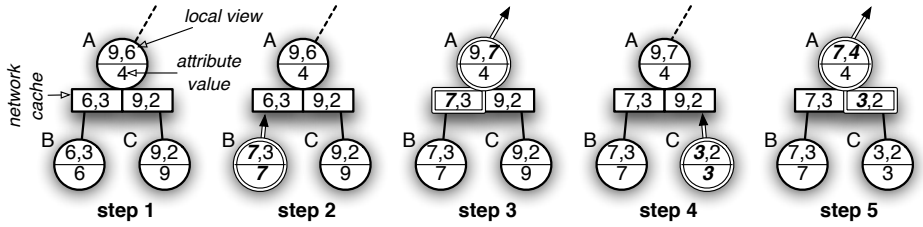


Fig. 8. Disseminating local view updates in TREE.

network. As shown in Section 6, with frequent local view updates this yields a significant reduction in network traffic w.r.t. FLAT. On the other hand, with this technique only the root of the tree is always guaranteed to obtain the information necessary to detect violations. The other nodes may detect violations only whenever the local view updates determining the violation are generated within their own subtrees.

Our solution departs from the traditional use of tree-shaped routing topologies for data collection because of two key aspects:

— Unlike data collection protocols, in our case the root is not physically attached to a base station. Indeed, the tree overlay is meant only to provide *structure* to an otherwise flat network, to improve on communication overhead. Further, other nodes are expected to take over the role of the root, to provide load balancing and therefore increased WSN lifetime.
— We obtain such improvements by limiting the propagation of local view changes upstream based on the content of local caches at every node. Such caches store a summary of relevant values previously forwarded upstream. This form of in-network aggregation improves "by construction" over the straightforward solution whereby every local view change is always forwarded upstream, even when not necessary.

Update propagation. Figure 8 shows an example of update dissemination in TREE. The monitored invariant again requires a local view including the two network-wide maxima. Nodes maintain in their network cache one entry for each of their children, as illustrated for parent A and children B and C in step 1. This entry stores the most recent local view update from the corresponding child, representative of the state of the subtree rooted at it. In step 2, a local value change at node B causes a modification to B's local view, and the subsequent propagation towards the parent. Propagation occurs after a short timeout that allows nodes to process local view updates possibly coming from their own children, further reducing network overhead. Upon receiving the update, A rebuilds its local view based on the content of the network cache and the local attribute values, as in step 3, and propagates the update further, as the local view has changed.

Value deletion. The processing for deleting values from the local view is similar. In step 4, a local attribute value drops at node C. This causes a local view update at C, and the corresponding propagation towards the parent A. This again recomputes the local view based on the new content of the network cache and the local attribute values, as in step 5, and propagates the update further because of the changes in the local view.

The key difference w.r.t. FLAT, easily seen by comparing Figure 7 and 8, is that TREE does *not* bring convergence of the entire network to the same local view. Indeed, local view updates are directed only towards the root, while in FLAT they spread along arbitrary paths in the network. This difference is key to the improvements in communication overhead enjoyed by TREE, which are however counter-balanced by the fact

that only a subset of the nodes (possibly only the root) is guaranteed to detect violation. Moreover, the overlay topology used by TREE makes this protocol less robust in the presence of communication or node failures and induces an uneven load on nodes. Next, we illustrate the mechanisms we employ to tackle these issues.

Network cache consistency. Changes in the routing topology, packet losses, and node failures may render the content of the network cache no longer consistent with the system state. This may cause both false negatives, e.g., when a local view update reporting a new maximum is lost, and false positives, e.g., when a maximum drops and nodes higher in the tree miss the local view update because of a topology change.

We adopt a soft-state approach to maintain the consistency of network caches. Each cache entry is associated with an expiration timeout that causes its removal unless the node responsible for it refreshes the entry within the timeout. To further improve performance in case of changes in the routing topology, upon changing parent the child node sends an explicit eviction message to the former parent. If this is still reachable, the message causes the removal of the child's cache entry before the timeout expires.

An extreme case of node failure is the crash of the tree root, which is a single point of failure in TREE. Nevertheless, when the root stops acknowledging packets, the neighboring nodes can *locally* detect the crash and elect a new root, e.g., using existing protocols [Frank and Römer 2005].

Load balancing. In TREE, the local view at every node filters the updates from the descendants based on aggregate information obtained from the corresponding subtree. As a result, nodes down in the tree are more likely to be relaying local view updates towards their parents, as their local views cover the system state to a lesser extent compared to nodes closer to the root, yielding an uneven load among WSN nodes.

We design a simple load balancing scheme to address this issue. Our scheme rotates the root role among the nodes to achieve a more even energy consumption. The decision to rotate is taken based on an estimation of the current energy budget of the network, as perceived at a given node. We periodically determine the node where this quantity is maximum, and hand over the root role to it.

We estimate the energy budget of a node n as:

$$w_n(i+1) = \frac{1}{2} \cdot w_n(i) + \frac{1}{2} \cdot \frac{\sum_{m \in Neigh(n)} w_m(i)}{|Neigh(n)|} \tag{1}$$

where $w_n(i)$ is the energy budget at node n at the current step i and $Neigh(n)$ denotes the 1-hop neighbors of n. Observe that in Equation 1 every node contributes to the evaluation of $w_n(i+1)$ at every other node, as the formula is recursively applied to the entire network. We achieve this by periodically exchanging the current value $w_n(i)$ with the nodes in $Neigh(n)$ and computing the next value $w_n(i+1)$ until the metric becomes stable (i.e., $w_n(i+1) = w_n(i) \pm \varepsilon$, ε being an approximation constant) or we reach a maximum number of iterations. Note that each iteration requires only a single broadcast of the current $w_n(i)$ from each node n.

Interestingly, identifying the node with the largest stable value of the metric in Equation 1 can be achieved by reusing the same machinery we use to compute in-network the global minima and maxima necessary to monitoring invariants. As a result, the current root eventually knows whether there exists another node n with a larger $w_n(i)$. If so, it hands over the root role to node n using an eventually-consistent dissemination protocol [Levis et al. 2004]. The hand-over message also includes the last sequence number used by the former root to refresh the routing tree. When node n receives such notification, it starts building a new routing tree rooted at itself using a strictly greater sequence number. Based on this information, every other node in the

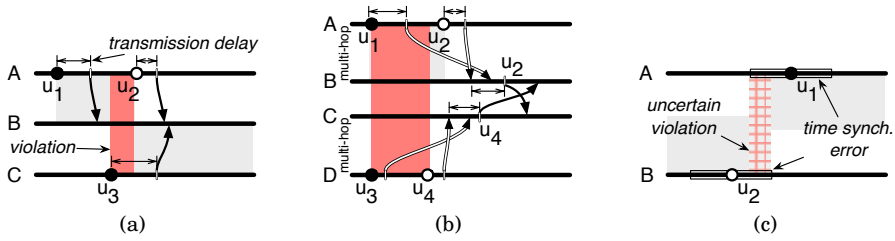


Fig. 9. Motivation and limitations of the history buffer.

network recognizes that a root change has taken place, and stops the propagation of the hand-over message.

5.3. Communication Delays

Communication delays may cause specific interleavings of local view updates that cause either protocol to miss some violations. Figure 9(a) illustrates the problem. Say that updates u_1 and u_3 establish a state violating the monitored invariant, whereas any other system state complies with it. Ideally, nodes A and C would propagate their updates as soon as they occur. In practice, however, the network stack may cause communication delays, e.g., because of collision avoidance mechanisms at the MAC layer. The intermediate node B may then receive the updates in the order u_1, u_2 and u_3. If we used only the most recent update from every node to evaluate the invariant, the system would not detect the violation.

Solution. We tackle the problem above with two combined mechanisms: *i)* a protocol maintaining a consistent ordering among timestamped local view updates, and *ii)* a circular *history* buffer storing recently-received updates, including those that have been superseded by new ones.

Our timestamp synchronization is inspired by [Römer 2001], adapted by neglecting communication and network stack delays. At each hop, the data is sent along with the timestamp of generation and the sender's (physical) clock at send time. At the receiver, the latter is used to convert the data timestamp to the receiver's (physical) clock before inserting the data in the history buffer. With this information, a node can reconstruct the proper sequence of updates and detect violations. For instance, B in Figure 9(a) is able to match u_3 with u_1, both stored in its history.

Limitations. This mechanism is effective only if there is at least one node with access to both u_1 and u_3, and therefore able to reconstruct the correct order causing the violation. This is not always the case in our protocols, which do not guarantee FIFO ordering on multi-hop paths. Consider Figure 9(b), where the monitored invariant is again violated only in the state following u_1 and u_3. The violation may go unnoticed if communication delays and update reordering cause the most recent updates u_2 and u_4 to supersede u_1 and u_3 at intermediate nodes B and C. As further re-propagation of the updates causing the violation is "quenched" at intermediate nodes, it may happen that no node in the network has enough information to detect the violation. However, addressing this problem would require either the propagation of the entire history, or stricter assumptions on communication delays and FIFO ordering—both too expensive in a WSN setting.

In addition, inaccuracies in timestamp synchronization may produce situations where given combinations of local states are *uncertain* [Römer and Ma 2009]. Consider Figure 9(c). Assume the state of B before u_2 at physical time t_2, combined with the state of A after u_1 at physical time t_1, violate the monitored invariant. If the timestamp synchronization error ε_{TS} is greater than $|t_2 - t_1|$, then it is not possible to determine

Component	TREE	FLAT
local view manager	9.8 KB	
dissemination manager	7.4 KB	2.9 KB
load balancing	3.2 KB	n/a
history buffer	2.2 KB	
evaluation manager	6.6 KB	
Total	**28.7 KB**	**25.5 KB**

Fig. 10. Program memory occupation of DICE components.

the correct ordering of such updates when inserting them in the history buffer. Using our timestamp synchronization protocol, ε_{TS} may be as large as 200μs per hop [Römer 2001], although protocols exist to reduce ε_{TS} to a few μs per hop [Maróti et al. 2004]. In practice, this means that we may not detect "instantaneous" violations that exist only for a few hundred milliseconds. However, these violations are not an issue in the scenarios we target where, as we already mentioned, transient violations of much longer duration are commonly accepted, and properly specified in DICE using the `tolerate` clause.

6. EVALUATION

We evaluate the performance and correctness of our DICE prototype based on two complementary approaches.

Section 6.1 focuses on TOSSIM simulations. These allow us to analyze and compare FLAT and TREE by relying on global knowledge of the network state. Moreover, using a simulator simplifies the assignment of value distributions to network nodes, and enables us to easily test different scenarios. Finally, it allows us to analyze relatively large networks, up to 225 nodes, at the price of a less accurate representation of wireless transmission. Therefore, Section 6.2 analyzes data from a real deployment consisting of TelosB motes placed in indoor and outdoor areas of our lab. This serves as a validation of our simulation results, albeit on a smaller scale, and as a real-world testing of our prototype based on actual environmental data.

We use our TinyOS implementation of DICE, described in Section 4, to run both simulation and testbed experiments. Our implementation has a modular design, which allows the user to select which dissemination protocol to employ based on system and application requirements. Figure 10 shows the code memory overhead. In terms of data memory, FLAT occupies $\sim$1.4 KB, while TREE occupies $\sim$4.2 KB. The larger data memory footprint of the latter is due to the buffers internal to CTP. These values are independent of the number and nature of the invariants monitored, each contributing an additional 10 B in the local view, 12 B in the network cache, and 14 B in the history buffer. The invariant specification is very compact thanks to the encoding illustrated in Section 4.1; for instance, the properties I1 and I2 in Section 2 occupy only 194 B.

As discussed in Section 3, the detection procedures depend on the nature of the monitored invariant. However, LP and DP invariants rely on the same underlying dissemination mechanism; they differ only in that the former can leverage "collective silence". This is possible because, for LP invariants, it suffices to disseminate only the (locally determined) truth value of the single predicates, rather than the actual attribute values, as discussed in Section 3.2. This minimizes communication overhead to a great extent. For this reason, in the following we consider only DP invariants.

Metrics. We focus on detection latency and communication overhead. As for the former, we define the *global detection latency* as the time difference between the instant when a change in the environment triggering a violation is available to DICE through the node attributes—i.e., it becomes *observable* by the software—and the instant at

which the violation is *first detected* anywhere in the network. From the application point of view, this indicates how fast the distributed dissemination protocols we devise can detect a violation.

Other metrics are useful to evaluate the "internal" performance of detection. Similar to the global detection latency, we define the *node detection latency* as the time difference between when a state change in the node attribute values triggers a violation, and the time at which *any given* node locally detects the violation. The average value of this metric is an indication of the speed at which the information necessary to evaluate invariants propagates through the network.

Global and node detection latency are related: the former coincides with the node detection latency of the *first* node recognizing the violation, easily computed as the minimum among all node detection latencies. The node detection latency can be computed only for those nodes that actually detect the violation—all in FLAT, typically only a fraction in TREE. Therefore, for the latter protocol we also evaluate the *detection count*, i.e., the number of nodes that detect the violation, as a measure of the redundancy of detection in TREE.

To investigate the overhead imposed on a node, we report on the *average number of local view changes per node* and the *average number of packets sent per node*, measured over the time span between the generation of the state change triggering a violation and the time at which the *last* node in the network detects it. These can be regarded as an indirect measure of the computational overhead and a direct measure of the communication overhead, respectively.

Correctness. In DICE, every state change triggers a brief period during which invariants may be incorrectly reported as violated or being complied with. This is the time required for the update to propagate throughout the network and is assessed as part of the aforementioned latency metrics.

Instead, our focus is on missed violations and false alarms. These can be generated by an incomplete propagation due to message losses. However, in FLAT the periodic rebroadcast guarantees eventual delivery. In TREE, update propagation relies on CTP, which achieves a message delivery ratio up to 99.9% [Gnawali et al. 2009]. Therefore, we do not analyze this aspect further.

The more interesting case is instead whenever an incorrect report about the invariant is caused by the reordering of updates inside the network, as discussed in Section 5.3, caused by an uneven propagation speed and a high churn in the attribute values. These conditions may reorder updates and cause indifferently a missed violation or a false alarm. For simplicity, in our evaluation we use scenarios that generate only the former.

Findings. We summarize as follows the key conclusions we draw based on our evaluation, which serve as a guideline for applying either of our dissemination strategy:

— In applications with frequent topology changes or likely node failures, the FLAT strategy is preferable, because of the unstructured operation that is largely unaffected by topology changes and the redundancy provided by every node's ability to locally detect invariant violations. This materializes both in reduced network overhead and smaller detection latencies.
— In scenarios where attribute values change frequently, the TREE strategy is preferable, as in-network aggregation greatly reduces the network overhead compared to FLAT. On the other hand, with slow attribute value changes, the cost of maintaining the tree topology outweighs the benefits of in-network aggregation, and FLAT becomes preferable.
— As the complexity of the monitored invariant grows, FLAT scales better than TREE, as the in-network aggregation functionality of the latter increasingly becomes in-

effective. This again is reflected in greater network overhead and larger detection latencies with more complex invariants in TREE.

— When the physical values possibly determining an invariant violation are clustered in a limited area (e.g., in case of a heating source) and thus sensed by a subset of nearby WSN nodes, FLAT generally features smaller detection latencies than TREE, as the propagation of local view changes determining the violation is forced by the tree topology and may thus take sub-optimal routes, unlike the unconstrained propagation in FLAT.

Next, we discuss these aspects in detail based on our quantitative results.

6.1. Simulation Experiments

We analyze the behavior of DICE using synthetic distributions of attributes, and evaluate separately the performance of the update dissemination and the likelihood of missed violations.

Nodes are arranged in a square grid[5], with an inter-node space of 10 m. We analyze network configurations ranging from 25 nodes to 225 nodes. Unless otherwise noted, in TREE the root is placed in one of the grid corners. Network connectivity is simulated using the LinkLayerModel tool in the TinyOS distribution.

We configured FLAT and TREE as follows. The polite gossiping employed by the former uses $\tau_l = 200$ ms and $\tau_h = 4$ s as lower and upper bounds of the transmission period, and $\gamma = 5$ as the number of neighbor transmissions triggering a message suppression. In TREE, we use the default CTP parameters of the TinyOS distribution. Moreover, as mentioned in Section 5.2, TREE buffers incoming updates for a small interval, which we set to 200 ms (i.e., same as τ_l) to ensure that the minimum time an update is buffered at a node is the same for FLAT and TREE. Unless otherwise noted, the results of each experiment are averaged over 50 repetitions.

6.1.1. Performance of Update Dissemination. Our protocols are influenced by the complexity of the invariant and specifically by its signature, determining the number of attribute values that must be present in the nodes' local views. To capture this aspect we use five invariants, collectively defined as:

$$\forall n_1, \ldots, n_k : \sum_{i=1}^{k} x@n_i < T, \qquad k \in \{2, 3, 4, 5, 6\}$$

As illustrated in Section 3, each invariant requires the local view to contain the k largest values of the x attribute. To avoid cluttering our charts, in the remainder we show only the results for the extremes, i.e., for $k \in \{2, 6\}$.

We consider two value distributions for x. The first is a 3-dimensional gradient, shown in Figure 11. This distribution simulates a physical phenomenon (e.g., a heating source) where the values sensed in the range $[1, 10]$ are proportional to the inverse of the square of the distance from a source, placed at the center of the grid. As the grid and distribution are perfect, we obtain a set of concentric rings of nodes with the same value for x. In the second distribution, instead, each node assumes a random value in the range $[1, 100]$. Albeit somewhat more artificial, this distribution is interesting in that violations do not follow a pattern and can happen anywhere in the network.

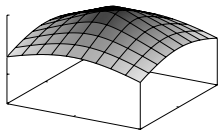
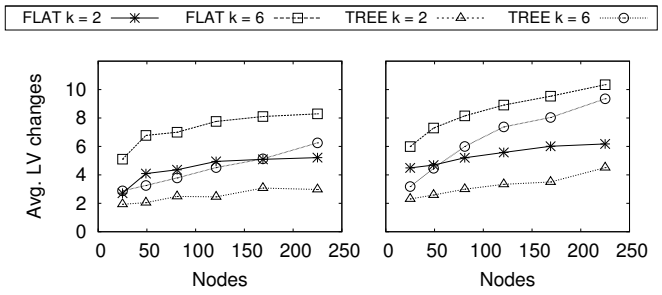


Fig. 11. A gradient distribution for x.

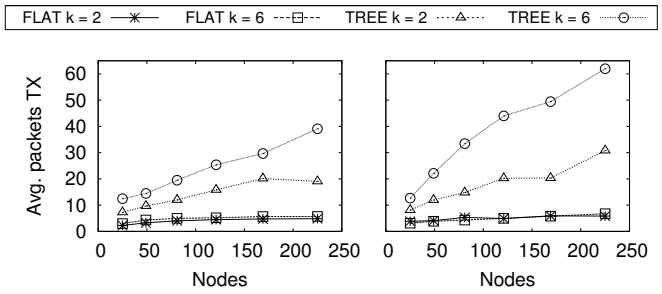
The simulations are executed as follows. In the initial state *i)* all nodes hold a value $x = 0$ *ii)* their local view reflects this global state *iii)* the overlay used by TREE pro-

[5]We also ran simulations with randomly-generated topologies. However, the many sources of randomness (topology, value distribution, timers, etc.) make them much less insightful.



(a) Gradient distribution. (b) Random distribution.

Fig. 12. Average local view changes per node.



(a) Gradient distribution. (b) Random distribution.

Fig. 13. Average number of sent packets per node.

tocol is completely built. This initial stable state is then perturbed with either of the aforementioned gradient and random distribution. The simulation ends when all nodes converge again to the same local view containing the new global maxima.

Unless otherwise noted, all simulations in this section are performed by using the setup above.

Local view changes and packets sent. As shown in Figure 12, the number of local view changes is slightly higher in FLAT than TREE. Indeed, in the latter local view changes are aggregated as they travel upstream: at a parent node, a local view change from one child (representing the state of an entire sub-tree rooted at it) may be superseded by a local view from another child, and result in a single local view change propagated upstream. Instead, in FLAT local view changes propagate in an unstructured fashion: a local view change may still "quench" another at a given node, but because view changes propagate along arbitrary paths, the same view change may reach other nodes along a different path and trigger other view changes.

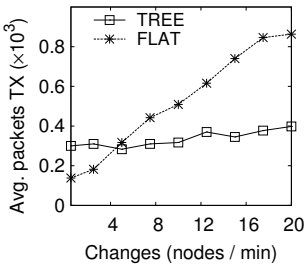
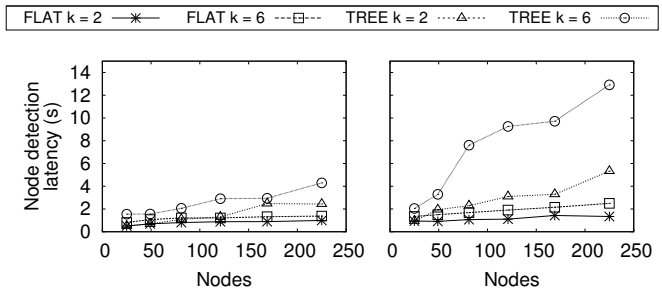


Fig. 14. Communication overhead vs. update rate.

Looking at local view changes alone, TREE would appear as the most convenient approach. However, tables turn when the average number of packets actually transmitted is considered, as shown in Figure 13. The higher value for TREE is determined by the messages necessary to maintain the network caches up-to-date and consistent—therefore, in essence, by the structure induced by TREE. In contrast, FLAT structureless dissemination of updates does not bear this overhead.



(a) Gradient distribution. (b) Random distribution.

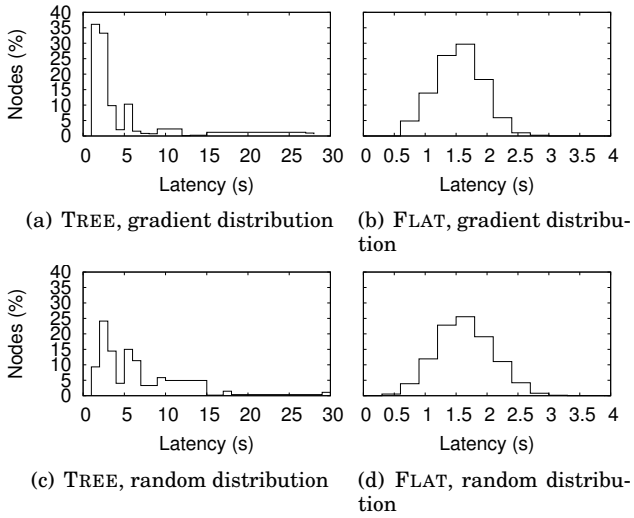
Fig. 15. Average node detection latency.

Impact of update rate. However, the observation above holds only in the case where the frequency at which the attribute values change at each node is relatively low. This is evident in the experiment in Figure 14, where we simulate a 10×10 grid monitoring the invariant $\forall m, n : x@m + x@n < T$. Nodes start from a random value of attribute x. Periodically, a number of randomly-selected nodes change their value to obtain the rate on the x-axis, over a simulated time of 2 hours. Intuitively, if the update rate is low, one or more "sweeps" of the entire network are sufficient in FLAT to inform all nodes about the new local view. If instead there are many concurrent changes, in FLAT they propagate in an unstructured fashion, each potentially causing a new update and therefore a new dissemination competing against the others. This ultimately generates a traffic that grows linearly with the update rate, precisely due to FLAT's inability to aggregate the updates. In contrast, the structure provided by TREE, detrimental at low update rates, becomes an asset when the update rate increases: local view updates can be effectively aggregated in-network and over subtrees, therefore greatly limiting the increase in traffic.

Detection latency. Figure 15 focuses on the average node detection latency. We note that the relative performance of FLAT and TREE is greatly affected by the distribution of attribute values: the behavior of TREE with the random distribution is significantly worse than FLAT—an order of magnitude in the case of $k = 6$. Indeed, in the random distribution the maxima can be anywhere. However, in the case of the gradient distribution, the attribute maxima are located at the center of the grid; a violation can be easily detected by neighbors in either protocol.

The significantly higher latency of TREE is explained by the fact that, in this protocol, *i)* the likelihood of detection is higher in nodes that are closer to the root, and *ii)* the root can be far away from the nodes contributing to the detection. Instead, in FLAT the propagation of maxima can be thought of as a "bubble", whose expansion (caused by the propagation of local view updates) is not restricted to an overlay as in TREE. As a consequence, the violation is detected as soon as the "bubbles" corresponding to the attribute maxima intersect at some node.

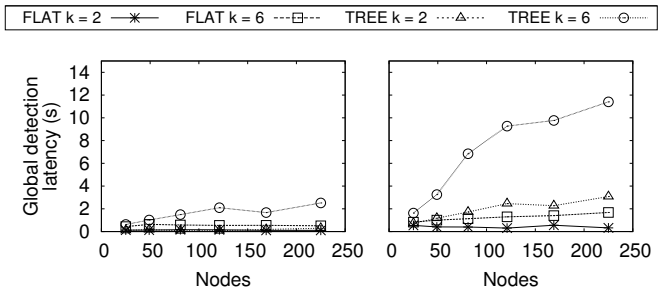
This behavior not only yields a smaller average node detection latency than TREE, but also a different distribution of the detection latencies at each node, as shown in Figure 16 for both gradient and random distribution. In the case of FLAT, the detection latency at each node resembles a Gaussian distribution, with relatively short tails. Instead, in TREE the distribution of latency is much more irregular: the nodes that are closer to the maxima (placed in the middle of the grid) detect the violation with a latency comparable to the slowest FLAT nodes, but others may detect with a latency two orders of magnitude larger. Moreover, the difference when moving from a gradient



(a) TREE, gradient distribution (b) FLAT, gradient distribution

(c) TREE, random distribution (d) FLAT, random distribution

Fig. 16. Detection latency for 225 nodes, $k = 2$. Values on the x-axis have different scales.



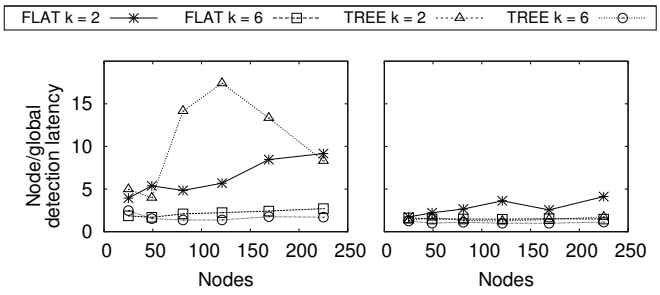
(a) Gradient distribution. (b) Random distribution.

Fig. 17. Global detection latency.

to a random distribution is more marked in TREE than in FLAT, especially w.r.t. the tails of the latency distribution.

In addition to the distribution of attribute values, which in practice is often unknown, the other parameter affecting the relative performance of the two protocols is the complexity of the monitored invariant, which in our experiments is represented by the value of k. As k increases, a node requires data from an increasing number of sources to perform detection. Therefore, the latency increases with k for both FLAT and TREE, as shown in Figure 15. However, the impact of this parameter is significantly larger in the latter.

Finally, Figure 17 shows the global detection latency (i.e., the time to the first detection), while Figure 18 shows the ratio between the average node detection and the global detection. Under challenging conditions, such as invariants with high complexity or scenarios with a random distribution, it is generally more difficult to detect violations. This causes the global latency to be higher and closer to the node latency, as reflected by the low values, and constant ratio between the two, in Figure 18(a) for $k = 6$ and Figure 18(b) for all cases. If the invariant is simple or the changes more localized, as in Figure 18(a) for $k = 2$, the global latency is much smaller than the node latency, with the structure of the overlay inducing bigger differences in TREE.



(a) Gradient distribution. (b) Random distribution.

Fig. 18. Ratio between average node detection and global detection latency.

Detection count. Similar arguments explain the trends of detection count, i.e., the fraction of nodes that detect violation. As already mentioned in Section 6, this metric is meaningful only for TREE, as FLAT guarantees eventual detection at all nodes. As seen in Figure 19, the higher the complexity of an invariant, the lower the detection count. The likelihood of a node performing detection increases closer to the root, and as k increases the phenomenon is exacerbated. A similar reasoning holds w.r.t. the attribute value distribution: the more scattered the values triggering a violation, the fewer the nodes that possess all the required information to enable detection, which explains the lower values for the random distribution.

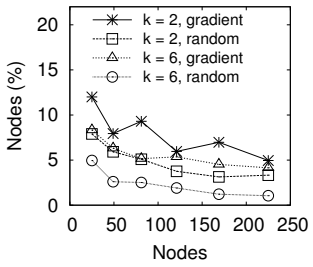
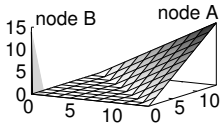


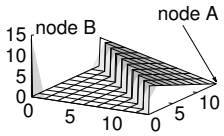
Fig. 19. Nodes detecting violations in TREE.

6.1.2. Assessing the Risk of Missed Violations. In Section 5.3 we discussed the compromises we make w.r.t. the potential of missing a violation; here, we evaluate their usefulness. Firstly, we analyze the effectiveness of the history buffer in preventing missed violations, by reproducing a scenario similar to Figure 9(a). Secondly, we assess how DICE behaves in situations where history buffers cannot help, similar to Figure 9(b). Finally, we evaluate the impact of node failures.

History buffers. We assess the extent to which history buffers help avoiding missed violations by monitoring $\forall m, n : x@m + x@n < 28$ in a 225-node grid. Nodes are initially assigned the distribution of x values in Figure 20(a), which violates the invariant due to the two nodes A and B, in opposite corners, with $x = 14$. After 500 ms (i.e., while the updates caused by the initial distribution are still propagating) this distribution is changed into the final one in Figure 20(b), which satisfies the invariant. Note that in both distributions a fraction of the nodes maintains the value of their attribute at $x = 0$. The change of distribution in the other nodes, instead, triggers a flood of updates. These updates are inserted in the history of all nodes, including A in the upper-right corner, on which we focus our experiment. A is also the root used in the simulations carried out for TREE. Our goal is to see how A can evaluate the invariant, and detect the violation, by combining its history with an older state of node B. Indeed, both A and B hold a value $x = 14$; however, the rapid change in distribution after only 500 ms occurs



(a) Initial distribution.



(b) Final distribution.

Fig. 20. Distributions to assess the impact of history size.

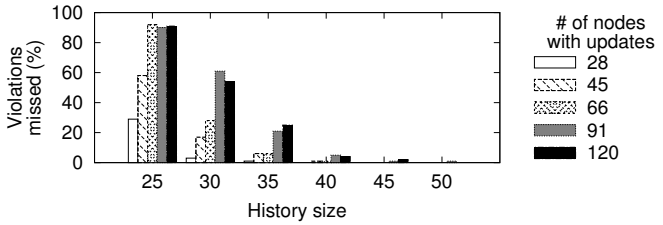


Fig. 21. Impact of history size in FLAT.

before B's value propagates all the way to A. The latter node can detect the violation only if its entry with $x = 14$ is still available in the history when B's $x = 14$ reaches A. Indeed, the high number of concurrent updates near A may fill A's history, flushing old entries.

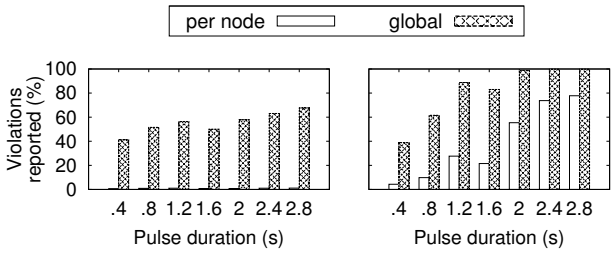
In our experiments, the history size ranges from 25 to 50. For each size, we report the percentage of violations detected by A over 350 runs. To determine the extent to which concurrent updates render the history ineffective in preventing missed violations, we use different sizes for the upper-right triangle of Figure 20(a), ranging from 28 to 120 nodes.

Interestingly, even with a small history size of 25 elements, TREE manages to capture all the violations triggered by all updates. The explanation lies in the shape of the tree topology built by TREE: the average number of children is 1, and the maximum is 5. Thus, each node receives data from a small number of neighbors, and therefore the risks of filling up the history buffer are lower. This is not valid for FLAT, where a node can overhear packets from several neighbors. However, Figure 21 shows that a history of 40 elements is already sufficient to bring the likelihood of missed violations below 4% in all cases, and a relatively small history of 50 elements guarantees detection even with a massive amount of updates, where more than half (120 out 225) of the nodes change their distribution. This confirms that the history buffer is effective in preventing missed violations.

Update reordering. The previous experiments demonstrate the ability of DICE to reconstruct a violation based on old values contained in the history. However, as mentioned in Section 5.3, this may not be enough when updates are propagated along non-FIFO multi-hop paths; this may cause a more recent update to supersede an old one, as shown in Figure 9(b). Situations like these are more likely to happen if the violation persists for a short time; a long duration implies a re-propagation of updates, and therefore increased chances that they are received in the right order.

We reproduce these scenarios as follows. We simulate a 225-node grid monitoring $\forall m, n : x@m + x@n < 10$. The value distribution for x is such that all nodes are placed on a plateau at $x = 1$, except for two opposite vertexes at $x = 3$. As in Figure 9(b), we change simultaneously the value to $x = 6$ on both these vertexes, obtaining short "pulses" that cause the violation of the invariant. Between pulses, we set $x = 3$ for 3 s, during which the updates should propagate throughout the network. This choice yields a worst-case scenario where the number of hops in between the two vertexes is the largest possible, which increases the probability of losing updates due to their reordering along multi-hop paths. For TREE, the root was set to one of the corner nodes other than the two chosen vertexes. The input to experiments is the duration of the pulses. The output is the percentage of violations detected globally and per node. These are reported over 50 runs of 5-minute experiments, for a total of more than 3000 pulses simulated.

Figure 22 shows the results, distinguishing between the number of violations detected globally and the average number of violations detected by any node. The signif-



(a) TREE (b) FLAT

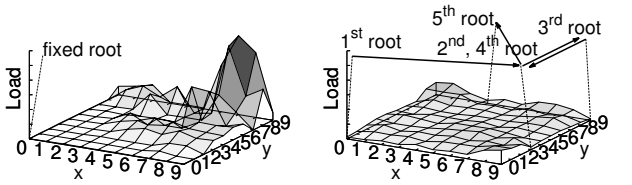
Fig. 22. Detection of fast violation pulses.

icantly better performance of FLAT is determined by its fully decentralized nature: because updates disseminate along multiple arbitrary paths, although some nodes may miss some violations, others can catch them. Even when the pulse duration is 400 ms (i.e., twice the lowest polite gossiping period τ_l) 40% of the violations are detected by at least one node. With a pulse duration of 2 s, more likely to occur in real-world applications, FLAT detects 100% of the violations. The worse performance of TREE is caused by its structure, which not only forces updates to propagate through predefined paths, but also yields larger detection latencies, as already pointed out. Latencies are also the cause for the lower average detection rate per node. Moreover, compared to the gradient scenario in Section 6.1.1, here the maxima are not necessarily close to each other, therefore latency is higher.

Node failures. The overlay exploited by TREE makes it more fragile in comparison with FLAT's structure-less, intrinsically more resilient strategy. To quantify this aspect, we run experiments comparing the two protocols in the presence of node failures. On a 10×10 grid, we monitor the invariant $\forall m, n : x@m + x@n < T$. All nodes change their x to a random value every 2 minutes, over a simulated time of 2 hours. In TREE, the root was set in one of the grid corners. Every 2 minutes we make two random nodes fail at the worst possible moment, i.e., right in between an attribute change and the dissemination of the corresponding local view change. Node failures are not recovered: the system reaches a point where the network is partitioned and no detection is possible. In this challenging scenario, TREE is able to detect violations only in 56% of the induced node failures. Instead, FLAT is able to detect a violation in 84% of the induced node failures, thanks to the inherent resilience of its dissemination.

6.1.3. Load Balancing in TREE*.* In Section 5.2 we described a load balancing scheme for TREE to mitigate the fact that nodes farther from the root bear a bigger fraction of the communication overhead. To evaluate our scheme, we simulate a network of 100 nodes deployed in a 10×10 grid. We monitor the invariant $\forall m, n : x@m + x@n < T$ for 5 simulated days. We randomly change the value of attribute x at each node every 2 minutes. To quantify the load, we count the number of bytes transmitted and received at every node, which approximates the corresponding energy consumption. Figure 23(a) shows the case where no load balancing is performed, and the root remains fixed at coordinates (0,0). The chart confirms the intuition that nodes far from the root bear a much higher load w.r.t. the others. Figure 23(b) shows, in the same scenario, the effect of changing the root at the end of each simulated day based on the scheme described in Section 5.2. The relative standard deviation of the load is 175% in Figure 23(a), and drops to 43% in Figure 23(b).

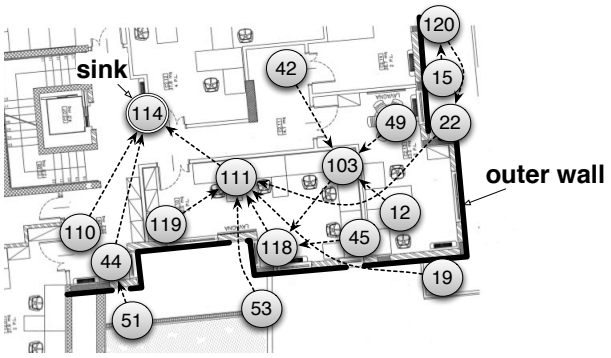
In principle, an even more uniform load could be achieved by increasing the frequency with which the metric w is computed, and therefore a new root is elected. However, this comes at two costs. First, the overhead of informing the next root of its new



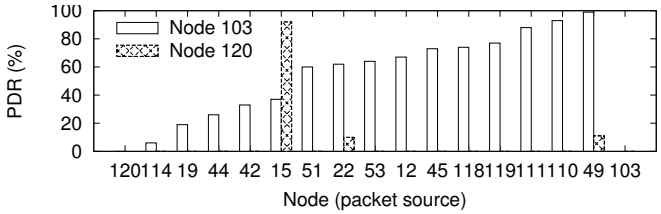
(a) Root is fixed at (0,0).

(b) Root is initially at (0,0), and moves according to our load balancing scheme.

Fig. 23. Load distribution in TREE.



(a) Topology snapshot for TREE.



(b) Packet delivery ratio for nodes 103 and 120.

Fig. 24. Laboratory testbed: setup and connectivity.

role: in the simulations of Figure 23, this averages to 1.7 packets per node for each hand-off. Second, the transfer of responsibility to the new root is not instantaneous. In our simulations, the time elapsed since the old root relinquished its role until all nodes join the tree set up by the new root is on average 1.2 s, and depends on the distance between the two roots. For instance, the transfer from (0,0) to (9,3) takes longer than the one from (9,3) to (9,9). While the tree is being reconfigured, packets containing local view updates may "wander" in the network; we counted an average of 0.44 per node during each root transfer. However, these are not lost: the underlying CTP protocol buffers each received packet at each hop, enabling their correct re-routing towards the new root as soon as its tree is set up.

6.2. Testbed Experiments

To analyze the traffic patterns and investigate the system behavior over time against the dynamics of real-world sensed data, we run a number of tests using 17 TelosB

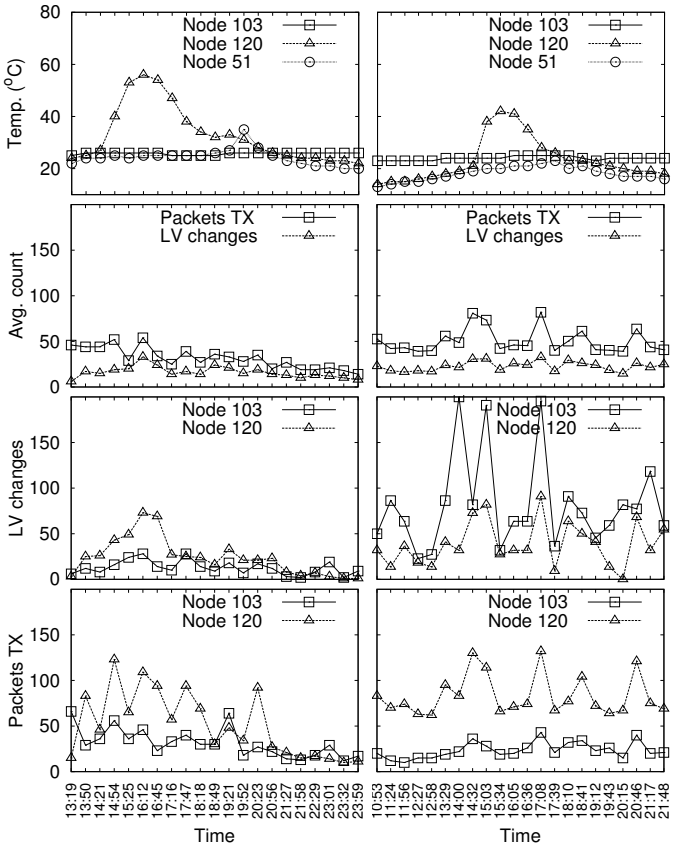


Fig. 25. Laboratory testbed: experimental results.

motes deployed in a lab environment as shown in Figure 24(a). Every node periodically reports statistics to a node connected to a computer. We monitor the invariant $\forall m, n : temp@m - temp@n < 10°\mathrm{C}$ where $temp$ is the temperature, sampled every 2 s using the on-board SHT11 sensor. This rate is overkill for a slowly-changing phenomena such as temperature: we intentionally use it to stress our system. To avoid short-term oscillations in temperature values due to sensor inaccuracies, we fed DICE with a moving average over the three most recent temperature readings. We configured the protocols as mentioned in Section 6.1, and let the system run for about 11 hours.

Figure 25 illustrates the results we gathered. The charts in the top row show the temperature values at three nodes representative of different placements, yielding different temperature changes. As shown in Figure 24(a), node 103 is placed indoors; its reported temperature value is relatively constant. Node 120 is instead exposed to direct sunlight; its readings are greatly influenced by the time of the day. This and similarly-placed nodes are more likely to experience a sudden value change, large enough (compared to nodes in other areas) to trigger violations. Finally, node 51 is almost always in shade, therefore usually reports the lowest temperature.

The charts in the second row from the bottom of Figure 25 illustrate the system performance over time, plotting the local view changes and packets sent per node, aggregated over periods of 30 minutes. On average, we observe about one local view

change every 3.75 minutes for TREE and 1.5 minutes for FLAT. Therefore, as in our simulations, the number of local view changes is higher in FLAT than in TREE. The better performance of TREE corresponds to the simulation settings to the right of the crossing point in Figure 14, where the higher update rate favors TREE over FLAT. However, unlike in simulation, in this case the number of packets is higher than local view changes. The reason is that temperature changes very slowly, therefore the communication overhead is dominated by the keep-alive messages in both protocols, unchanged w.r.t. simulations.

The trends above are illustrated by the charts at the bottom of Figure 25. Node 103 is in the center of the testbed, whereas node 120 is at its fringe. The snapshot of the routing topology[6] for TREE in Figure 24(a) shows that node 103 aggregates and reports data only from indoor nodes, with a relatively constant temperature curve. Additionally, the path from node 120 towards the sink does not include node 103. Thus, one would expect this node to have a lower number of local view changes than node 120, which is instead outdoor and experiencing a temperature increase. Interestingly, this is the case for TREE but not for FLAT. The reason is that in FLAT, as the packet delivery ratio in Figure 24(b) indicates, node 103 has more good neighbors and therefore more update sources, hence the higher number of local view updates. Instead, node 120 can receive updates from fewer sources, hence its lower number of local view changes.

Finally, the effect of polite gossiping in FLAT can be observed by looking at the local view changes and packets transmitted for nodes 103 and 120. In the case of node 103, not every update corresponds to a packet transmitted. As this node has a larger number of neighbors, it is more likely to suppress its broadcast, unlike node 120. For the latter, in the absence of neighbors communicating redundant data, we can see that the number of packets transmitted is considerably higher than the number of local view updates.

7. RELATED WORK

The problem we tackle with DICE is reminiscent of predicate detection in distributed systems. In this field, seminal work investigates the detection of stable predicates [Chandy and Lamport 1985], whose truth value changes only once in the system lifetime. Our work focuses instead on *unstable predicates* [Garg and Waldecker 1994], that is, predicates whose truth value may change repeatedly. A particular class of unstable predicates are linear inequalities in the form $\sum_i x_i > K$, similar to DP invariants. This class has been previously studied by Tomlinson and Garg [1997]. In some respect, their algorithm is analogous to ours, e.g., it identifies the largest values of x_i to decide whether the predicate is satisfied. However, their techniques, as well as most literature on distributed predicate detection, are based on logical time and therefore expensive and hardly applicable *at run-time* in WSNs; post-mortem analysis of global state through a mixture of physical and logical timestamping has indeed been explored by Sookoor et al. [2009]. Our work is instead motivated by the desire to support run-time detection of invariant violations.

In-network aggregation, the technique we employ to reduce traffic and collect network state, is widely referenced in the WSN literature. Various protocols, network overlays and summarization techniques have been proposed in this respect [Madden et al. 2002; Considine et al. 2004; Nath et al. 2004]. However, the general goal of these works focuses on data acquisition and traffic optimization, overlooking issues that are key in monitoring invariants, e.g., the consistency of the gathered state.

[6]The topology is relatively stable, with an average of only 5.70 parent changes per node in the 11-hour experiment.

Moreover, declarative approaches have been proposed as programming [Mottola and Picco 2011] or debugging [Cao et al. 2008] abstractions for WSNs. TinyDB [Madden et al. 2005] is an example of the former, where an SQL-based, database-like abstraction simplifies querying data from a WSN. In contrast with TinyDB and similar programming-oriented approaches, which focus on providing construct to develop the core application functionality, DICE focuses instead on the complementary problem of ensuring that the latter behaves as intended.

In this respect, the work closest to ours is the one on passive distributed assertions (PDA) [Römer and Ma 2009]. As in our system, programmers specify the correct behavior of programs in PDA by using predicates that can be seen as a combination of LP and DP invariants. Nevertheless, the monitoring process occurs in a centralized manner, by relying on a *fixed* monitoring station *outside* the WSN, and based on global knowledge of the system state. The latter is acquired by using a secondary WSN deployed alongside the real one that sniffs and delivers packets to the monitoring station, which in many cases limits practical applicability.

8. CONCLUSIONS

We presented DICE, a system for WSN-based distributed monitoring of global invariants in physical processes. DICE provides a declarative language to specify invariants and a run-time support enabling efficient monitoring of their violations. The run-time can be configured to use either the structure-less FLAT protocol or the TREE protocol, which instead relies on an overlay. FLAT provides increased fault tolerance by allowing *any* node to detect a violation, at the expense of increased overhead in scenarios with high rate of changes in the monitored application state. TREE provides improved performance in this scenario by structuring and optimizing the dissemination of relevant state changes, but this very structure makes the approach more complex, as it requires additional mechanisms to preserve structure in the presence of failures, and limits the ability to detect violation only to a subset of the nodes.

Acknowledgments

The authors wish to thank Christine Julien, Alberto Montresor, and Kay Römer for their comments on early drafts of this paper.

This work is partially supported by the European Union Seventh Framework Programme (FP7-ICT-2009-5) under grant agreement n. 258351 (project make*Sense*).

References

Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. 2008. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *Proc. of the 6th Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, USA, 85–98.

K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (1985), 63–75. DOI:http://dx.doi.org/10.1145/214451.214456

Jeffrey Considine, Feifei Li, George Kollios, and John Byers. 2004. Approximate aggregation techniques for sensor databases. In *Proc. of the 20th Int. Conf. on Data Engineering (ICDE)*. IEEE, New York, NY, USA, 449–460.

Christian Frank and Kay Römer. 2005. Algorithms for generic role assignment in wireless sensor networks. In *Proc. of the 3rd Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, USA, 230–242.

V. K. Garg and B. Waldecker. 1994. Detection of weak unstable predicates in distributed programs. *IEEE Trans. Parallel Distrib. Syst.* 5 (March 1994), 299–307. Issue 3.

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. 2009. Collection tree protocol. In *Proc. of the 7th Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, USA, 1–14.

Ştefan Gună. 2011. *On Neighbors, Groups, and Application Invariants in Mobile Wireless Sensor Networks*. Ph.D. Dissertation. University of Trento.

Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. 2000. System architecture directions for networked sensors. *SIGPLAN Not.* 35 (November 2000), 93–104. Issue 11.

Suzanne Hoppough. 2006. Shelf life. Forbes magazine. (April 2006).

Philip Levis, Neil Patel, David Culler, and Scott Shenker. 2004. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the 1^{st} Symp. on Networked Systems Design and Implementation (NSDI)*. ACM/USENIX, New York, NY, USA, 15–28.

Kaisen Lin and Philip Levis. 2008. Data discovery and dissemination with DIP. In *Proc. of the 7th Int. Conf. on Information Processing in Sensor Networks (IPSN)*. IEEE Computer Society, Washington, DC, USA, 433–444.

Bela Lipták. 1995. *Process Control*. CRC Press Inc., Baton Rouge, IW, USA.

Samuel Madden, Michael Franklin, Joe Hellerstein, and Wei Hong. 2002. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Operating System Review* 36 (December 2002), 131–146.

Samuel Madden, Michael Franklin, Joe Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* 30, 1 (2005), 122–173.

Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. 2004. The flooding time synchronization protocol. In *Proc. of the 2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, USA, 39–49.

Luca Mottola and Gian Pietro Picco. 2011. Programming wireless sensor networks: fundamental concepts and state of the art. *Comput. Surveys* 43, 3 (April 2011), 19:1–19:51.

Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. 2004. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. of the 2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, US, 250–262.

Kay Römer. 2001. Time synchronization in ad hoc networks. In *Proc. of the 2nd Symp. on Mobile ad hoc networking & computing (MobiHoc)*. ACM, New York, NY, USA, 173–182.

Kay Römer and Junyan Ma. 2009. PDA: passive distributed assertions for sensor networks. In *Proc. of the 8th Int. Conf. on Information Processing in Sensor Networks (IPSN)*. IEEE Computer Society, Washington, DC, USA, 337–348.

Tamim Sookoor, Timothy Hnat, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. 2009. Macrodebugging: global views of distributed program execution. In *Proc. of the 7th Int. Conf. on Embedded Networked Sensor Systems (SenSys)*. ACM, New York, NY, USA, 141–154.

John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. 2005. Opportunities and obligations for physical computing systems. *Computer* 38 (2005), 23–31. Issue 11.

Alexander I. Tomlinson and Vijay K. Garg. 1997. Monitoring functions on global states of distributed programs. In *J. Parallel Distrib. Comput.*, Vol. 41. Academic Press, Inc., London, UK, 173–189.

John Tsitsiklis. 1984. *Problems in Decentralized Decision Making and Computation*. Ph.D. Dissertation. MIT.

John Eldon Whitesitt. 1995. *Boolean Algebra and Its Applications*. Dover Publications, Mineola, NY, USA.

Yujie Zhu and Raghupathy Sivakumar. 2005. Challenges: communication through silence in wireless sensor networks. In *Proc. of the 11^{th} Int. Conf. on Mobile Computing and Networking (MobiCom)*. ACM, New York, NY, USA, 140–147.