Timed automata based modeling and verification of denial of service attacks in wireless sensor networks

Youcef HAMMAL[*], Quentin MONNET[†], Lynda MOKDAD[†], Jalel BEN-OTHMAN[‡] and Abdelkarim ABDELLI[*]

[*] *LSI, Department of Computer Science*
USTHB University, Algeria

[†] *Lab. LACL, Université Paris-Est*
LACL (EA 4219), UPEC, F-94010 Créteil, France
E-mails: quentin.monnet@lacl.fr, lynda.mokdad@u-pec.fr

[‡] *Lab. L2TI, Université Paris 13*
L2TI (EA 3043), UP13, F-93430 Villetaneuse, France
E-mail: jbo@univ-paris13.fr

Abstract. *Sensor networks have been increasingly deployed for civil and military applications over the last years. Due to their low resources, sensors come along with new issues regarding network security and energy consumption.*

Focusing on the network availability, previous studies proposed to protect the network against denial of service attacks with the use of traffic monitoring agents. Working on the election process, we try to enhance this solution by introducing an energy-aware and secure method to dynamically select these "cNodes" in a clustered WSN: nodes with the higher residual energy get elected. We discuss limitations of this deterministic selection and suggest to designate new control nodes, "vNodes", to monitor the cNodes by periodically enquiring about their remaining energy, thus ensuring that they do not lie during the election process in attempt to keep their role.

Validation is first carried out with a formal specification of our proposal using the UPPAAL model-checker. We model nodes by means of communicating timed automata, logical clocks and timing constraints. Through Computation Tree Logic we express and check properties for the election processes, related to energy and presence of greedy or jamming nodes.

Studia Informatica Universalis.

Finally, we present some experimental results obtained with the ns-3 simulator in order to analyze the impact of our proposal on the energy repartition in the network.

Keywords: *Wireless sensor networks; Reliability, availability, and serviceability; Energy-aware systems; Model-checking; Simulation*

Introduction

Collecting data has been a fast increasing concern over the last decades. Today, all is to be measured, or watched over. One wants to study the pollution degree of seas. Other people intend to measure seismic activity, or regulate traffic on roads, or regulate the humidity level under greenhouses... Examples of measures to be done over wide areas are countless. Some of those areas of interest may be hard to access (mountain, dense forests...). Obviously it is not possible to send humans bearing measuring tools to all those places, all the time. This is why wireless sensor networks (often abbreviated as WSNs) have been deployed in a growing fashion. Those networks are made of small devices, sometimes dropped on the spot by helicopter, tasked with gathering data on their physical environment. Sensors embed radio emitters and receivers, and are able to exchange data through wireless communication. Meaningful data is typically directed to a base station, which acts as an interface between the network and the user.

Sensors may be numerous, so they must be cheap. This means that they have low computing capabilities, and little available memory. They also run on limited energy, drawn directly from their battery. It is not always feasible to reach the sensors once the network is deployed, and hence to change their batteries, so the energy consumption must be handle with great care. For these reasons algorithms run by the sensors must be as less requiring as possible. And yet in the same time, the sensors, once left to their own devices, must self-organize and form a consistent network without outside help.

Some of the various applications of WSNs are very sensitive. They may be deployed for forest fire detection, or for measuring the nuclear activity degree in sensitive areas. Military operations may also involve

them for communications, or for detection (be it for detecting enemies or biological or chemical agents). In this context, bringing security guaranties to the network becomes a primary issue. Availability is the capacity of the network to carry out the services it was designed for, whatever happens. It is related to resistance to failures or, this is the point in this paper, to what we call denial of service attacks, which aim at bringing the network down.

Based on former proposals (see Section 1), the present study relies on the use of monitoring nodes (or "*cNodes*") to protect a clustered wireless sensor network again various denial of service attacks. Specifically, it focuses on the selection process of those control nodes. Our approach consists in taking energy into account at this step, so as to obtain a better load balancing. We propose to designate the sensors for the *cNode* position according to their residual energy, but we show that several problems occur with deterministic election. Indeed compromised nodes could see a flaw to exploit in order to take over the *cNode* role and decrease the odds of being detected by announcing high residual energy. We address this issue by introducing a second role of surveillance: we choose "*vNodes*" responsible for watching over the *cNodes* and for matching their announced consumption against a mathematical model. We also recommend that every node in the cluster be monitored by at least one *cNode* to prevent all the *cNodes* to be elected inside the same spatial area of the cluster at each election iteration.

We intend to validate our approach through two different processes. First we propose a formal specification of our proposal using timed automaton of the UPPAAL model-checker. Cluster heads, communication mediums, different kinds of nodes in our WSN as well as compromised nodes are modeled by means of communicating timed automata that are augmented with logical clocks and timing constraints. Furthermore, we use CTL (Computation Tree Logic) logic to express and check many properties related to *cNodes* and *vNodes* election process in accordance with the defined criteria related to energy consumption and presence of compromised nodes which may have greedy or jamming behaviors. The second aspect of this validation consists in a software simulation of our proposal.

Hence the rest of the paper is organized as follows: first we present a quick overview of related works in Section 1. After we introduce a new *cNode* selection method in Section 2, we present a formal specification and model-checking of our solution through the timed automata of the UPPAAL model-checker in Section 3. We then describe and discuss the simulation of the algorithm that we have done using ns-3 in Section 4, while last Section eventually summarizes the main contributions of the paper and gives some directions of future works.

1. Related works

Deploying wireless sensor networks for sensitive operations requires that all aspects of network security are reviewed. Hence numerous research investigations have been undertaken on ways to improve security in WSNs.

Some of these studies led to new protocols to provide confidentiality in WSNs, *i.e.* to prevent unauthorized intruders to access to the meaningful data that nodes exchange in the network. In [1] for example, the authors introduce a method to ensure data confidentiality against parasitic adversaries. The use a very simple key management scheme, roughly consisting in sharing a key between each node and the base station. They also introduce en route encryption mechanism to reinforce the confidentiality against compromised nodes: a special subset of nodes in the network re-encrypt the message on its way to the base station. A periodic renewing of the keys of the node is deployed, to prevent attackers to use compromised keys at will. The authors claim a low energy consumption in regard with classic public key encryption schemes. But encryption itself is not always necessary: another example concerning confidentiality is [2], where we proposed several ways to split messages and to send them through several distinct paths in the network to avoid using systematical strong encryption. Depending on the importance of the data contained in each packet, we propose to send it using one of the three: the SDMP method [3], a secret sharing scheme such as Shamir's [4], or standard encryption.

Another issue consists in ensuring that every node actually is who it pretends to be (thus preventing address spoofing). This is called au-

thentication. It often comes along with data integrity. For instance the proposals in [5] are based on two symmetric key-based security building blocks. The first block is called Secure Network Encryption Protocol. It provides data confidentiality, two-party data authentication and data freshness. The second block is called μTESLA (for "micro" version of the Timed, Efficient, Streaming, Loss-tolerant Authentication Protocol) and assumes authenticated broadcast using one-way key chains constructed with secure hash functions. Many other mechanisms related to authentication are resumed and compared in surveys such as [6].

Other studies have been realized in an attempt to secure specific operations occurring in WSNs, and one can find secure ways to compress or aggregate data in WSNs [7, 8]. But confidentiality, authentication or secure aggregation are of no use if the good functioning of the network gets disrupted. What is the point in ciphered and authenticated messages which will never reach their destination? This brings us to resistance against denial of service (DoS) attacks, which is the topic of this paper.

Indeed there are many existing attacks able to compromise the good working of a wireless sensor network. Several mechanisms have been proposed to detect it and to provide countermeasures [9, 10]. Even restricted to the data link, media access control and network layers of the network, there are many different existing DoS attacks. Consequently, even more solutions have been proposed to counter them. Various studies have been realized in the purpose of countering one specific kind of attack, sometimes on specific protocols, thus leading to the proposals of hardened versions of AODV [11] or DSR [12] for instance. Some researchers prefer to ensure that nodes are not physically withdrawn from the network to get modified [13], considering any returning node as compromised (but not detecting nodes modified on site), or to verify by means of mobile agents that the binary code run by the nodes has not been modified [14], although this solution makes use of cryptography mechanisms and can limit the evolution of the network.

Many other systems, often more convenient to deploy, consist in the implementation of trust based mechanisms [15, 16] with agents applying set of rules [17] on traffic to attribute a trust value to each of the nodes in the network. Each enforced rule is expected to protect the net-

work against one kind of attacks: intruders are detected on breaking the rules. We are particularly interested in the solution of Lai and Chen [18] who proposed to elect control nodes to monitor the traffic in clustered networks so as to detect and react to denial of service attacks. In such a scheme, clustered networks are partitioned into clusters *via* algorithms such as LEACH [19] or VSR [20]. One cluster head (CH) per cluster is responsible for gathering data from its peers, for aggregating and sending it to the base station. The CHs are the only nodes to use long range (and expensive) transmissions to reach the base station. Clustering enables to preserve energy for the sensing nodes and offers an easier management of the nodes. In our case, all nodes of a cluster can reach their cluster head directly (*1-hop transmission*), as on Figure 1. Control

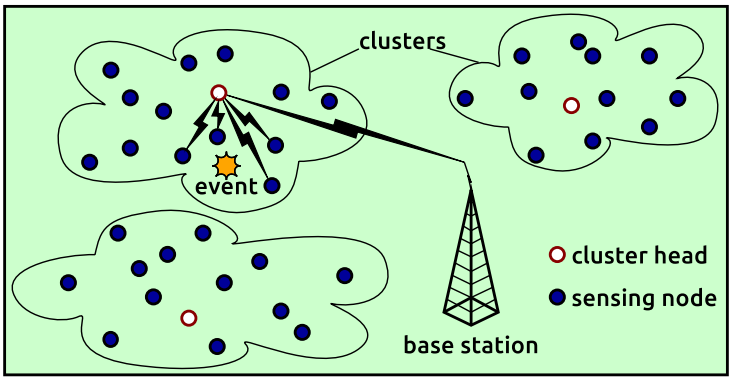

Figure 1: Clustered wireless sensor networks scheme

nodes are elected among the non-cluster head nodes of a cluster. We call them *cNodes* from now on. They are responsible for listening to the traffic and detecting nodes whose emitted traffic exceeds a given threshold value. These abnormal behaviors are reported to the cluster head. On reception of reports from several distinct *cNodes* (to prevent false denunciation from a compromised node), the CH virtually excludes the suspicious node from the cluster. Although the original method is efficient for detecting rogue nodes, the authors do not give details of the election mechanism to choose the *cNodes*. Also, there is no mention in their study of renewing the election in time, which causes the appointed *cNodes* to endorse a heavier energy consumption on a long time.

In a first attempt to bring load balancing to this solution, we propose in former papers to reiterate the election periodically [21, 22]. Results coming from the simulation show a better load repartition among the cluster. But in previous work our focus was not on designing an energy efficient election process for the *cNodes*. Contrary to the present study, we had assumed that the election would be led by the nodes themselves, each node picking a pseudo-random number and matching it to a predefined probability to decide whether it would be or not *cNode* for the coming cycle (this is roughly the same principle as the self-election of the cluster heads in LEACH clustering protocol).

Beside being energy-efficient, it is important for an algorithm to *just work*, and thus for researchers to provide guaranties of its functioning. Formal method may help in this, and one of the most interesting techniques is model-checking. It is a formal method based technique for verifying finite-state-concurrent systems, and can automatically extract performance or safety properties from a model and to ensure they are asserted at all time in the network. Additionally, were a system not to work properly, model-checking tools provide a trace to help finding the source of the error. Contrary to testing or simulation, results obtained *via* model-checking do not vary between distinct running instances; more important, formal specification enables one to verify the system against every single execution trace, where testing would only raise some encountered errors (maybe not all existing errors). Model-checking is used in a wide area of applications, from the validation of network protocols to the verification of safety properties in regard with the non-collision between platoon vehicles [23], for example.

One of the two aforementioned papers already makes use of formal methods, including model-checking, to evaluate its proposal [22]. A first attempt is described in the study to model it with CTMC (Continuous Time Markov Chains), but limitations due to the period over which the detection is performed led to put away Markovian processes. Instead the nodes and the monitoring agents are modeled with eGSPN (extended Generalized Stochastic Petri Networks), a class of Petri networks including times transitions and suitable for representing stochastic processes. From those eGSPNs we could extract a number of performance and dependability properties formally expressed in terms of

the Hybrid Automata Stochastic Logic (HASL [24]) and relying on Linear Hybrid Automata (LHA). These properties were eventually asserted with the COSMOS statistical model-checker [25].

Other works include model-checking of WSN-addressed systems, though we found few of them concerning protection against DoS attacks. In [26] for example, three security protocols are analyzed through model-checking, nominally TinySec, allowing authentication and encryption, and LEAP and TinyPK, both used for key management. The authors used the AVISTA model-checker, designed for verifying security protocols. In our study, we prefer to work with the more general UPPAAL model-checking tool.

2. cnodes selection mechanism

Control nodes watching over the network traffic allow the detection of various types of denial of service attacks. This is achieved with agents running on the *cNodes* and applying specific rules on overheard traffic [17]. Each rule is used to fight against one (sometimes a few) specific attack(s): jamming, tampering, black hole attacks, and so on. Each time a *cNode* notices that a rule is broken by a node, it raises a bad behaviour for this node, and send an alert to the cluster head. Following are some example rules:

– Rate rule: assuming that minimal and maximal rates for data each node sends are enforced, a bad behaviour will be reported if those rates are not respected. With this rule, monitoring agents should be able to detect negligence (if minimal rate is not reached) or flooding (if maximal rate is exceeded) attacks.

– Retransmission rule: a *cNode* overhearing a packet supposed to be retransmitted by one of its neighbor (the neighbor node is not the final destination for this packet). If the concerned neighbor does not forward the packet, it may be undertaking a black-hole (full dismissal of packets) or a selective forwarding attack.

– Integrity rule: a bad behaviour will be raised if a neighbor of the node running the monitoring agent tampers a packet before forwarding it. Applying this rule assumes that the nodes are not expected to proceed

either to data aggregation or compression before forwarding.

– Delay rule: forwarding a packet should not exceed a threshold delay.

– Replay rule: a message should be sent no more than a limited number of times.

– Jamming rule: an unusually high number of collisions (compared to average, or concerning only some nodes) may be related to the presence of a jamming node. If jamming is done with random noise, without legitimate packets containing a node identifier, it may be difficult to detect the source of it, but several cooperating agents should be able to detect it.

– Radio transmission range rule: a node sending messages with an unexpectedly high power may be trying to launch a hello flood (it tries to appear in the neighbor list of as many nodes as possible) or wormhole attack (it redirects a part of the overheard traffic to another part of the network). Hence it may be consider as a bad behaviour.

In the rest of this study, we will not describe in details each one of the mentioned attacks, nor will we detail the associated solutions to counter them. When details are needed, we will consider only one example: flooding attacks. The model of a flooding attack is the following: a malicious node sends a high amount of data to prevent legitimate nodes from communicating by saturating the medium, or by establishing too many connections with the receiver node [27]. In wireless sensor networks, it is also used to drain the energy of neighbor nodes. *cNodes* are responsible for listening to the traffic of their surrounding nodes: if a sensor is to generate more traffic in the network than a predetermined threshold, it is considered as potentially compromised and trying to flood the network. At this time a report is sent to the cluster head. On reception of reports coming from multiple *cNodes*, the CH considers that traffic from the suspicious node must no more be considered. Information about distrust is passed on to normal nodes which stop listening to the packets coming from the attacker. We work under the following assumptions: firstly, the cluster head is not a compromised node (the use of *cNodes* to detect a malicious CH is described in [18], but it is not considered in this paper). Secondly, we do not consider the case of several malicious nodes cooperating with one another.

Electing the *cNodes* is not an easy task. In [22] we expose and compare three ways to elect them:

– pseudo-random election by the base station;

– pseudo-random election by the cluster head;

– pseudo-random election by the nodes themselves (all nodes have a probability p to become *cNode*; they pick a pseudo-random number x, and endorse the *cNode* role if and only if x is lower than p).

We assumed that election should be random so that compromised nodes would not be aware of which node could control the traffic. In that previous study however we do not consider the remaining energy during the *cNodes* election. But monitoring the traffic implies to keep listening for wireless transmission without interruption. Hence *cNodes* will have a greater energy consumption than normal nodes. Given that preserving energy is an essential issue in the network, we prefer to ensure load balancing rather than assuring a pseudo-random election, and thus we mean to consider the residual energy of the nodes during the election. This choice also raises new issues and makes us define a new role for the nodes in the cluster.

2.1. *Using vNodes to ensure a secured deterministic election*

There is no way to measure remotely the residual energy of a node, so the only way to get its value is to "ask" this node. The election algorithm we propose is described as follows:

1) During first step, each node evaluates its residual energy and sends the value to the cluster head;

2) Having received the residual energy of all nodes in the cluster, the cluster head picks the n nodes with the highest residual energy (where n is the desired number of *cNodes* during each cycle) and returns them a message to assign them the role of *cNode*.

It is a deterministic selection algorithm which eliminates any random aspect from the process. The rule is simple: nodes possessing (locally) the highest residual energy will be elected. Given that the *cNode* role implies consuming more energy (*cNodes* listen to surrounding commu-

nications most of the time), rotation of the roles is theoretically assured. But the deterministic aspect is also a flaw that may be exploited by compromised nodes. This is a crucial issue: we can not neglect compromised nodes as the whole *cNodes* mechanism is deployed in the sole purpose to detect them!

More precisely, the problem may be stated as follows. Compromised nodes will be interested in endorsing a *cNode* role, as it enables them:

– to reduce the number of legitimate *cNodes* able to detect them;

– to advertise the cluster head about "innocent" sensing nodes to have them revoked.

When a pseudo-random election algorithm is applied, a compromised node (or even several ones) can be elected during a cycle, but it will loose its role further in time, for later cycles. Even with a self-election process (based on LEACH [19] model for instance, such as shortly described above), compromised nodes can keep their *cNode* role as long as they want, but they can not prevent other (legitimate) nodes to elect themselves, too. With deterministic election however, they can monopolize most of the available *cNode* roles. They only have to announce the highest residual energy value at the first step of the election to get assured to win. If there are enough compromised nodes to occupy all of the n available *cNode* roles, then they become virtually immune to potential detection.

To prevent nodes from lying when announcing their residual energy, we propose to assign a new role to some of the neighbors of each *cNode*. Those nodes — we call them *vNodes*, as for *verification* nodes — are responsible for the surveillance of the monitoring nodes. Once the *cNodes* election is over, each neighbor to a *cNode* decides with a given probability whether it will be a *vNode* for this *cNode* or not. A given node can act as a *vNode* for several *cNode* (in other words, it can survey several neighbor *cNode*).

If this role consumes too much energy, it is not worth deploying *vNodes*: we should rather use pseudo-random election for the *cNodes*. So *vNodes* must not stay awake and listen most of the time, as *cNodes* do. Instead they send, from time to time, requests to the *cNode* they watch

over, asking it for its residual energy. They wait for the answer, and keep the value in memory.

Once they have gathered enough data, *vNodes* try to correlate the theoretical model of consumption of the *cNode* they survey and its announced consumption, deduced from broadcast messages (during elections) and answers to requests from *vNodes*. Four distinct cases may occur:

1) The announced consumption does not correlate (at all) with the theoretical model: there is a high probability the node is compromised and seeks to take over *cNode* role. It is reported to the cluster head;

2) The announced consumption correlates *exactly* with the theoretical model: the node is probably a compromised node trying to get elected while escaping to detection (in other words, the rogue *cNode* adapts its behavior regarding to the previous point). It is easy to detect the subterfuge as values received from the rogue node and the ones computed by the *vNodes* are exactly the same. It is reported to the cluster head;

3) The announced consumption correlates roughly with the theoretical model, but does not evolve in the same way (regarding to the model) than the real consumption locally observed by the *vNodes* (local (in time) evolution of the announced consumption does not "stick" to the one of the surrounding *vNodes*, which should roughly rise or decrease during the same periods). The node is probably compromised, trying to escape detection by decreasing its announced energy with random values. It is reported to the CH;

4) The announced consumption correlates roughly with theoretical model, and evolves in the same way as the traffic observed by *vNodes*. Whether the node is compromised or not, it has a normal behavior, and is allowed to act as a *cNode*.

If a given *vNode* is in fact a malicious node, it could lie about integrity of the *cNode* it watches. To prevent that, the cluster head must receive multiple reports (their number exceeding a predetermined threshold) from distinct *vNodes* before actually considering a *cNode* as compromised. To some extent, this also makes the scheme resilient to errors from the *vNodes*.

In that way, nodes are allowed to act as *cNodes* only if they announce plausible amounts of residual energy. Assuming that this role consumes more energy than sensing only, the nodes elected as *cNodes* will sooner or later see their residual energy drop below the reserve of normal sensing nodes, which implies that they will not get re-elected at the next election. Note that the cases 2 and 3 make a compromised node decrement its announced energy as the time goes by. Even if inconsistency may be noticed and the compromise detected, this simple behavior ensures that the rogue node will stop to get elected at one point in the time.

Thus, the interest of *vNodes* can be summarized as follows: a compromised node can not ensure the takeover of the *cNode* role at each cycle without cheating when announcing residual energy, and hence being detected by the *vNodes*. Detecting rogue *cNodes*, or forcing them to give up their role for later cycles, are the two purposes of the *vNodes*. The *vNode* role does not prevent a node to process to its normal sensing activity (requests to *cNodes* must not occur often, otherwise it will drain too much power from the *vNodes*). The state machine of the nodes is presented in Figure 2.

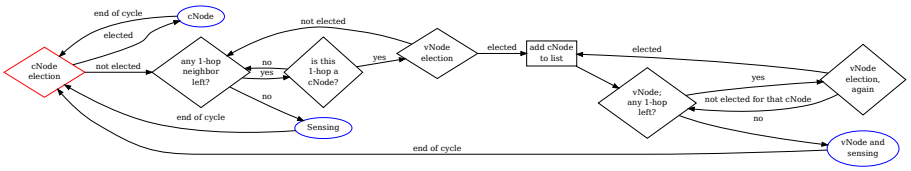

Figure 2: State machine of the (non-CH) nodes

2.2. *Cluster coverage in case of heterogeneous activity*

Deterministic election of the *cNodes* does not only introduce a flaw that compromised nodes could try to exploit. There is a second problem, independent from the nodes behavior, that could prevent the detection of compromised nodes. If a region of the cluster happens to produce more traffic activity than the other parts of the cluster, the energy of its nodes will be drawn faster. In consequence, none of the n nodes with the

highest residual energy (n being the desired number of *cNodes* during each cycle) will be located inside this region, and some nodes may not be covered for surveillance as long as traffic do not fade, possibly for all cycles. Figure 3 illustrates this problem.

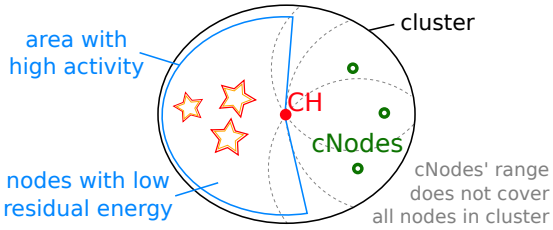

Figure 3: Illustrative scheme: *cNodes* are elected inside the area with less activity (thus with more residual energy) and do not cover nodes from the opposite side of the network.

To address this issue we need to ensure that every node in the network is covered by at least one *cNode*. So the election process we presented in 2.1 needs to be modified. The correct version is as follows:

1) During first step, each node evaluates its residual energy and broadcasts the value;

2) The cluster head listens to all values. Other nodes also register all messages they hear into memory;

3) All nodes send to the CH the list of their 1-hop neighbors [1];

4) The CH picks the n nodes among those with the highest residual energy, such that the n nodes cover all other nodes in range [2]. If needed, it selects some additional nodes to cover all the cluster;

5) The CH returns to selected nodes a message to assign them the role of *cNode*.

Note that some clustering algorithms (such as HEED [28] for example) provide other election mechanisms (for cluster heads, but that can also be used for selecting *cNodes*) based on residual energy. We do not

1. We do not deal with the case of compromised nodes cheating at this step of the process. Indeed they could announce extra virtual neighbors to try to escape from coverage.
2. The details of the algorithm executed by the cluster head at this step are not given in this study.

want to use it because energy only takes part in the process as a factor for probability that the nodes declare themselves elected. Instead we prefer nodes to broadcast their residual energy in order to enable surveillance by the *vNodes*.

2.3. *Observations*

cNodes apply a very basic trust based scheme to the cluster: when a sensor node breaks a rule, for example by exceeding a given threshold for transmitted packets, it is considered as untrustworthy. There are many other trust based schemes in literature, most of them more advanced than this one (see Section 1). The *cNodes* could implement several other trust mechanisms (by lowering a reputation score on bad behaviour for each node for instance). As more complex mechanism would create additional overhead, we prefer to limit our study to this simple method.

3. Formal Specification and Model-Checking of the System

In this section, we present a formal specification and model-checking of our system made up of a network of wireless sensors (WSN) which have to operate under DoS attacks. The language we use to modeling cluster heads, mediums, and nodes of such systems is the timed automata of the model-checker UPPAAL [29].

Timed automata, first introduced in [30], are flattened automata augmented with time constraints over logical clocks. However, UPPAAL modeling language offers additional features such as bounded integer variables and urgency. The query language of UPPAAL, used to specify properties to be checked, is a subset of CTL (Computation Tree Logic) [31–33].

Accordingly, before our system could be model-checked with UPPAAL, it has first to be specified as timed automata. We first give the definition of standard automata and show how these are augmented with time annotations and timed semantics to give rise to the modeling language of UPPAAL. Thereafter, we explain through our study case how

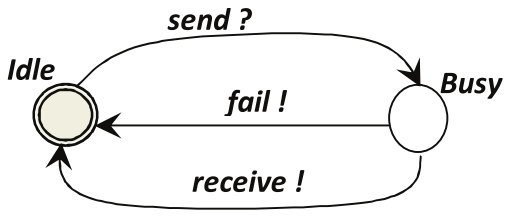

Figure 4: Classical untimed automaton

the components of the system are described by means of these timed automata.

3.1. *Timed Automata*

We present below the definition of classical finite state automaton which are used to specify intended (untimed) behaviors of processes (*i.e.*, active components of communicating systems such as nodes and cluster heads of WSN). Obviously, each of these automata does not take into account time constraints and only describes its process statuses and control moves between them along with their triggers (see Figure 4).

Definition 3.1. An untimed automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \hookrightarrow, q_0 \rangle$ where:

– Q is the set of locations (untimed states) of the process automaton (depicted as graph nodes). For instance, $Idle$ and $Busy$ are locations in the automaton given in Figure 4.

– Σ is the set of interactions (*i.e.*, events) that the specified process could perform synchronously with the related interactions of its cooperating processes. An interaction occurs over an abstract channel whose symbol (let it be a) is a part of its name. Henceforth, the symbols $a!$ and $a?$ denote respectively *send* and *receipt* interactions.

– $\hookrightarrow \in Q \times \Sigma \times Q$ is the set of edges (automaton transition arcs) between locations. As illustrated in Figure 4, the occurrence of a $send?$ event leads the control to move through the edge labeled with this event from $Idle$ location to $Busy$ location.

– q_0 is the initial location (denoted by a double circle such as for $Idle$ location in Figure 4).

However when analyzing a timed system, designers of its reactive components would actually want to express onto their automata the timing constraints that govern most of the interactions' occurrences between the components. Hence, occurrences of send or receipt actions have to abide to timing restrictions in order to correctly provide component services to its environment. Formally, this approach consists in expressing time constraints by means of boolean formulas over logical clocks. Such variables express time progress but their values could be initialized and tested.

Definition 3.2. (timing constraint) Let χ be a finite set of clocks ranging over $\mathcal{R}^{\geq 0}$ (set of non negative real numbers). The set $\Psi(\chi)$ of timing constraints on χ is defined by the following syntax: $\psi ::= true \mid x \ll c \mid x - y \ll c \mid not(\psi) \mid \psi \wedge \psi$ where $x, y \in \chi$, $c \in \mathcal{R}^{\geq 0}$ and $\ll \in \{<, \leq\}$. Other assertions such as, $x > 3$, $2 \leq x < y + 5$, $\psi \vee \psi$ can be defined as abbreviations.

A valuation $v \in V$ of the clocks is a function that assigns a non-negative real value $v(x) \in \mathcal{R}^{\geq 0}$ to each clock $x \in \chi$. We say that v satisfies a constraint $\psi \in \Psi$ if $\psi(v)$ evaluates to $true$. For $v \in V$ and $X \subseteq \chi$, we define $v[X := 0]$ to be the valuation $v' \in V$ such that $v'(x) = 0$ if $x \in X$, and $v'(x) = v(x)$ otherwise. For $\delta \in \mathcal{R}^{\geq 0}$, we define $v + \delta$ to be the valuation $v' \in V$ such that $v'(x) = v(x) + \delta$ for all $x \in \chi$.

Roughly, a timed automaton [30] is a finite directed graph annotated with conditions and resets over non-negative real valued clocks. We thus enhance previous untimed graphs with timing constrains by adding three mappings I, G and Z as follows:

Definition 3.3. (Timed Automata) The timed version of an automaton $\mathcal{A} = \langle Q, \hookrightarrow, \Sigma, q_0 \rangle$ is an extended graph $\mathcal{A}_{\mathcal{T}} = \langle \mathcal{A}, \chi, I, G, Z \rangle$ where:

– χ is a finite set of clocks.
– $I : Q \longrightarrow \Psi(\chi)$,
– $G : \hookrightarrow \longrightarrow \Psi(\chi)$, and
– $Z : \hookrightarrow \longrightarrow 2^{\chi}$

The first mapping I assigns to each location $q \in Q$ of the untimed automaton a sojourn or activity condition called *invariant* (denoted $I(q)$)

which may be the boolean constant $true$. The second mapping G assigns to each edge $(e \in \hookrightarrow)$ a timing guard which should be true to let the edge be taken (*i.e.*, to let the transition fire). The mapping Z associates with each edge a set of clocks initializations which may be empty (see Figure 5).

A state of a timed automaton $\mathcal{A}_{\mathcal{T}}$ depicts a configuration of the automaton at some instant. Formally, it is a pair (q, v) defined by a location q and a clock valuation v. At any state, $\mathcal{A}_{\mathcal{T}}$ can evolve either by a discrete state change corresponding to a move through an edge that may change the location and reset some of the clocks, or by a continuous state change due to the progress of time at a location. For $a \in \Sigma$ and for $\delta \in \mathcal{R}^{\geq 0}$ we define the relation $\xrightarrow{a} \subseteq (Q \times V)^2$ and $\xrightarrow{\delta} \subseteq (Q \times V)^2$ characterizing respectively the discrete and the continuous state changes as follows:

$$\frac{e = (q, a, q') \in \hookrightarrow, G(e)(v)}{(q, v) \xrightarrow{a} (q', v[Z(e) := 0])} \qquad \frac{\forall \delta' \in \mathcal{R}^{\geq 0}, \delta' \leq \delta . I(q)(v + \delta')}{(q, v) \xrightarrow{\delta} (q', v + \delta)}$$

The first inference rule states that a discrete transition could be taken (given in the rule conclusion) if the guard $G(e)$ of the related edge e evaluates to $true$ (as stated in the rule premise). Some clocks $Z(e)$ (if any) shall be reset once the edge e is taken. On the other hand, the second rule shows how time can progress with some amount δ in some location q by increasing clocks valuations (from v to $v + \delta$). This rule is called *time closure* and stipulates that this time progress could occur if the location invariant $I(q)$ continuously remains true for all valuations $v + \delta'$ such that $\delta' \leq \delta$.

The role of invariants is important. Indeed as time progresses, the values of the clocks increase providing the state satisfies the invariant. For states that do not satisfy the invariant, the progress of time is "stopped". This mechanism allows the specification of hard deadlines: when for some action the deadline specified by the invariant is reached, the continuous flow of time is interrupted. Therefore, the action becomes urgent and it is "forced" to occur if it is enabled.

A deadlock status will be thus any timed configuration of the automaton whose the related active location has a false activity condition

and all its outgoing transitions have false timing guards. In other terms, the time can not progress in such a state where no action is enabled in respect of its timing guard. So, no way is available to leave this state to enable time progress again.

3.2. *Modeling Language of UPPAAL*

The modeling language of UPPAAL [29] consists in networks of timed automata. In fact, a system is modeled as a network of timed automata in parallel, each of which is related to at least one of the system components. As aforementioned, a timed automaton is a flattened finite-state machine extended with clock variables. It uses a dense-time model where a clock variable evaluates to a real number. All the clocks progress synchronously in such a way that all automata clocks valuations simultaneously increase with the same amount of time. Otherwise, when the progress of one clock is blocked because some timing constraint becomes unsatisfied at a given instant then all the clocks become blocked too. The model is further extended with bounded discrete variables that are part of the state. These variables are used as in programming languages: they are read, written, and are subject to common arithmetic operations or manipulated within C-like functions we could add as assignments onto the edges. A state of the timed system is defined, as explained in Subsection 3.1, by the locations of all automata, the clock constraints, and the values of the discrete variables. Every automaton may fire an edge separately or synchronize with another automaton, which leads to a new state.

The UPPAAL modeling language extends timed automata with the following additional features [29]:

– *Constants* are declared as *const name=value*. Constants by definition cannot be modified and must have an integer value. Bounded integer *variables* are declared as *int[*min*,*max*] name*, where min and max are the lower and upper bounds, respectively. Guards, invariants, and assignments may contain expressions ranging over bounded integer variables. The bounds are checked upon verification and violating a bound leads to an invalid state that is discarded (at run-time). If the bounds are omitted, the default range of -32768 to 32768 is used. Here

is a declaration of two constants denoting the numbers of respectively nodes and regions in a WSN cluster:

`const int` $NBR\_NODES = 9, NBR\_REGIONS = 2;$

In the same way we declare other constants manipulated in our models, such as $ASLEEP\_DUR, AWAKEN\_DUR, \& \ ELECTION\_TERM$ denoting respectively the asleep mode duration, the awaken mode duration and the period to selecting *cNodes* and *vNodes*.

– *Templates automata* are defined with a set of parameters that can be of any type (*e.g.*, int, chan). These parameters are substituted for a given argument in the process declaration. For instance, the $Node$ template has two arguments related to the identifiers of the node itself and the region it belongs to. These parameters are respectively defined according to the new types idN and IdR declared as follows:

`typedef` $int[0, NBR\_NODES - 1]\ IdN;$
`typedef` $int[0, NBR\_REGIONS - 1]\ IdR;$

– *Binary synchronization channels* (abstract interaction gates) are declared as *chan c*. An edge labeled with $c!$ in one automaton synchronizes with another labeled $c?$ in a cooperating automaton. A synchronization pair is chosen non-deterministically if several combinations are enabled.

– *Broadcast channels* are declared as *broadcast chan* c;. In a broadcast synchronization one sender (*e.g.*, the cluster head CH launching the signal $release!$ to free all mediums) can synchronize with an arbitrary number of receivers (*e.g.*, mediums waiting for $release?$). Any receiver that can synchronize in the current state must do so. If there are no receivers, then the sender can still execute the $release!$ action, *i.e.* broadcast sending is never blocking. For instance, the statement given below defines two broadcast channels the CH use to make all nodes respectively sleep or wake whatever their current statuses:

`broadcast chan` $sleep, wake;$

– *Urgent synchronization channels* are declared by prefixing the channel declaration with the keyword *urgent*. Delays must not occur if a synchronization transition on an urgent channel is enabled. Edges using urgent channels for synchronization cannot have time constraints,

i.e. no clock guards. Urgent locations (denoted with U letter inside its depicting circle) are semantically equivalent to adding an extra clock x, that is reset on all incoming edges, and having an invariant $x \leq 0$ on the location. Hence, time is not allowed to progress when the system is in an urgent location.

– *Committed locations* are even more restrictive on the execution than urgent locations. A state is committed if any of the locations in the state is committed (denoted with the C letter inside its circle). A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

– *Arrays* are allowed for clocks, channels, constants and integer variables. They are defined by appending a size to the variable name, *e.g.* we define some arrays of channels which the nodes of each region use to interact with the related medium to achieve their communications:

`broadcast chan` $send[NBR\_REGIONS], receive[NBR\_REGIONS]$;
`broadcast chan` $ok[NBR\_REGIONS], fail[NBR\_REGIONS]$;
`chan` $rtc\_rts[NBR\_REGIONS]$;
`chan` $free[NBR\_REGIONS], busy[NBR\_REGIONS]$;

The next arrays of channels are used by the cluster head and *vNodes* to request and get the residual energies from nodes of any kind or *cNodes*:

`broadcast chan` $req\_All\_ResidualEnergy, req\_CN\_ResEnergy[NBR\_REGIONS]$;
`chan` $getResidualEnergy[NBR\_NODES]$;
`broadcast chan` $get\_CN\_ResEnergy[NBR\_REGIONS]$;

Next, we define also arrays of other types that allow the cluster head to store collected residual energies of nodes, suspicion claims on them, sets of selected *cNodes* and *vNodes*. Note that $ListOfNodes$ is just a new type derived from int with a limited range $[0..NBR\_NODES - 1]$.

`int` $nodeResidualEnergy[NBR\_NODES]$;
`int` $nbrClaims[NBR\_NODES]$;
$ListOfNodes setOfCNodes,\ setOfVNodes$;
$ListOfNodes\ sortedListOfNodes$;

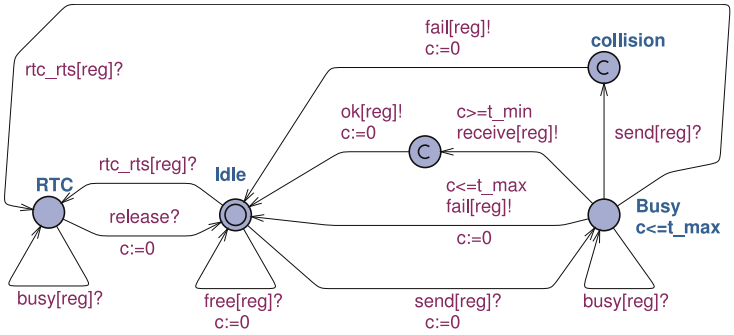

Figure 5: UPPAAL automaton of a region medium template

`bool` $visited[NBR\_NODES]$`; //mark visited nodes.`

`bool` $regCovered[NBR\_REGIONS]$`;//mark regions in which at least one cNode is elected.`

– *Expressions in UPPAAL* range over clocks and integer variables. Expressions are manipulated with the following labels:

- ***Guard*** A guard over an edge is a particular expression satisfying the following conditions: it is side-effect free; it evaluates to a boolean; only clocks, integer variables, and constants are referenced (or arrays of these types); clocks and clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions (disjunctions are allowed over integer conditions).

- ***Synchronization*** A synchronization label is either on the form $Expression!$ or $Expression?$ or is an empty label. The expression must be side-effect free, evaluate to a channel, and only refer to integers, constants and channels.

- ***Assignment*** An assignment label is a comma separated list of expressions with a side-effect; expressions must only refer to clocks, integer variables, and constants and only assign integer values to clocks.

- ***Invariant*** An invariant over a location is an expression that satisfies the following conditions: it is side-effect free; only clock, integer variables, and constants are referenced; it is a conjunction of conditions of the form $x < e$ or $x <= e$ where x is a clock reference and e evaluates to an integer.

3.3. *UPPAAL Models of the System Components*

We are interested in modeling and analyzing all processes in one cluster. Each cluster is split up into many regions, each of which consists of many nodes communicating throughout a shared medium. Hence, two nodes which try to send their data in the same time, would lead a collision that makes their attempts fail. We present below the automata templates of each of the system components: the region medium, the cluster head, parts of a node implementing sensing, *cNode* and *vNode* modes, the whole node, and lastly fragments of compromised nodes.

3.3.1. *Automaton of the medium*

Each region possesses its separate medium allowing all its nodes to communicate with each other. Collisions may thus happen leading to failure of data transmission. The automaton template of a medium is parameterized by its region identifier reg and owns an initial location $Idle$ where the medium is considered as free. That is why, the event $free[reg]?$ may happen at that location in a synchronous way with the achievement of its dual action $free[reg]!$ in any node automaton which is checking the medium status.

When a node automaton tries to execute an action $send[reg]!$, the related medium automaton should have its control in $Idle$ location where it can synchronize with the previous event by taking the edge labeled with its dual action $send[reg]?$ and incoming to $Busy$ location. Therein, the medium can interact with any node that is checking (over the channel $busy[reg]$) whether it is occupied. However, the medium could stay at this location as long as its invariant $c <= t\_max$ is true where t_max is the maximum value of the contention window of transmission. Meanwhile, four cases may occur:

1) The communication may abort, for instance because the receiver was not ready. This case is depicted by an edge labeled with $fail[reg]!$ outgoing back from $Busy$ to the initial location $Idle$.

2) Another node can make a send which leads to collision. This failure is depicted by a sequence of two edges labeled with respectively $send[reg]?$ and $fail[reg]$ with a committed location in-between. At

last, the control returns also back to the initial location $Idle$.

3) The medium may receive from the cluster head a prevailing signal $rtc\_rts[reg]?$ asking it to commute to the ***RTC/RTS***[3] communication mode depicted by the location RTC.

4) If no one among the three cases described above does happen and the clock c valuation comes to exceed t_min delay (minimum value of contention window) then events $receive[reg]!$ and $ok[reg]!$ could be successively broadcast to all nodes of the region and control returns back to $Idle$ location. Note that such actions remain offered as long as the edge guard and the source location $Busy$ invariant are true and no one of the above conflicting transitions does occur.

Last, it should be noticed that the medium may change its status from $Idle$ to RTC location through an edge labeled with the event $rtc\_rts[reg]?$ broadcast by the cluster head CH. On the other hand, control may return back to $Idle$ location once the medium succeeds to perform $release?$ event in a synchronized way with a $release!$ event that CH should have simultaneously broadcast.

3.3.2. *Automaton template of the cluster head*

The cluster head CH may be in either *Awaken* location or *Asleep* location. It shall commute from one status to the other one according to timing constraints using a local clock c_1 and expressed in terms of guards and invariants we added on both these two locations and the edges in-between. For instance, the invariant on the location *Asleep* is $c_1 <= ASLEEP\_DUR$ stating that the control may remain in this location as long as the amount of time elapsed therein (given by c_1 valuation) does not go beyond the sleep duration. Hence, the transition to *Awaken* status becomes urgent exactly once that limit is reached, thereby, broadcasting a $wake!$ signal to reactivate all the system nodes which would have been as well in $Asleep$ mode for the same amount of time. Similarly, the control goes from $Awaken$ to $Asleep$ whenever the clock value reaches the Awaken duration $AWAKEN\_DUR$, deactivating thus the system nodes throughout the broadcast channel $sleep!$.

3. This mode is temporarily used to send control packets in the CSMA/CA protocol.

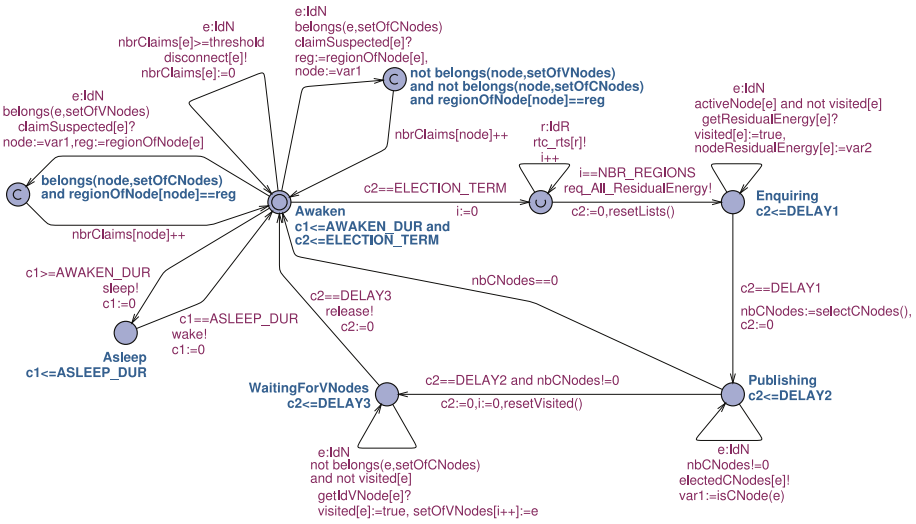

Figure 6: UPPAAL automaton of the cluster head template

However, the invariant at the $Awaken$ location is a conjunction which contains an additional activity condition over a second clock c_2, namely $c_2 <= ELECTION\_TERM$ related to the election cycle of *cNodes* and *vNodes*. In fact, this location could be also left to $Enquiring$ location *via* an edge when the guard $c_2 == ELECTION\_TERM$ becomes true. In this case, once all regions mediums are checked as free the control immediately moves (through a second step outgoing from an urgent location) and the CH thereafter yields a broadcast $req\_All\_ResidualEnergy$! to all the cluster nodes (including both *cNodes* and *vNodes*) in order to trigger the collect of residual energies from these nodes. During this move, the lists related to the *cNodes* and *vNodes* selection shall also be emptied thanks to the function *resetList()*. Notice also the initialization of the clock c_2 in such a way it could be used next in other timing formulae. The rational behind such a reduction of clocks number or an integer variable range is to alleviate the state explosion problem.

When the cluster head control reaches the $Enquiring$ location it stays therein until the clock c_2 reaches some delay during which the cluster head is expected to receive the residual energies of the nodes throughout the broadcast canal $getResidualEnergy$. We take care that

every node does not send its data more than once by using a boolean array $visited$ that records such events. Once c_2 valuation equals some delay 1 the *cNodes* are elected and saved according to our method implemented in the function $selectCNodes()$. Simultaneously to that function call a silent move occurs into the next location $Publishing$ where each node is informed whether it is selected as *cNode* or not. In a similar fashion, the progress of c_2 to delay 2 triggers the transition to *WaitingForVNodes* location where the control stays for some delay 3 during which the cluster head collects the identifiers of the *vNodes*. Notice that some nodes will randomly proclaim themselves as *vNodes* and then communicate their identifiers to the cluster head which stores them into a set of *vNodes*. When the delay 3 expires, the control returns back to the initial location $Awaken$.

Between two election cycles, the cluster head at $Awaken$ location might receive through $claimSuspected$ channels, from time to time, claims about suspected simple sensing nodes or *cNodes*. The cluster head has to check the identity of the $CNode$ which sends the claim about a node that operates within the sensing mode. To this end, the CH checks that the sender belongs to the set of *cNodes* and is situated in the same region as the suspected node. Similarly, the cluster head has to check that any suspension claim about a *cNode* is sent by a *vNode* situated in the same region as the suspected *cNode* and that it has not been sent by another *cNode* which would be considered in such a case as compromised. Every time that a claim on a *cNode* or a sensing node e is deemed a potential treat, then the number of claims on that node (*i.e.*, $nbrClaims[e]$) is incremented. When the number increases up to a given threshold, the cluster head achieves an action $disconnect[e]!$ that will be synchronized with a dual action ($disconnect[e]?$) in the automaton of the suspected node e. There are two edges labeled with the action $disconnect[e]?$, which are outgoing from locations in parts of the node automaton related to the sensing and *cNode* modes and both of them lead to a sink state $Disconnected$ where no more activity is allowed. Note that the election cycle begins with a rtc_rts broadcast signal to get hold of mediums of regions and it finishes with a $release$ broadcast signal to free them again.

3.3.3. *Automaton of the sensing role of WSN nodes*

The automaton template given in Figure 7 illustrates the sensing function mode of a node nd. We notice that initial location is $Awaken$ from which the control may go to $Asleep\_SN$ location and then return back in accordance with the receipt of respectively two signals $sleep?$ and $wake$ originating from the cluster head. Note also that when the node in a sensing mode wakes up it will randomly choose a number n of a new type $NbrMsgs$ defined as follows:

`typedef int`$[0, MAX\_NBR\_MSGS - 1]\ NbrMsgs$`;`

where MAX_NBR_MSGS is an integer constant previously declared. The sensing node shall also be deactivated and change its status from $Awaken$ to $Disconnected$ location through either one of two edges as soon as respectively its residual energy runs out or the signal $diconnect?$ from CH is received.

The sensing node (at $Awaken$ location) can also overhear all data packets sent on the medium of its region reg *via* a $receive[reg]?$ action labeling a loop arc upon $Awaken$ location and decreasing thus the residual energy of nd with some amount RC (constant previously defined). Furthermore, the sensing node can make send actions a number of times equal to the chosen number $nbMsgsToSend$: first, it has to check whether its region medium is free (by an interaction $free[reg]!$ guarded with $nbMsgsToSend > 0$). If this is case, the control moves to a location $ReadyToSend$ where it shall stay for random time less than a previously constant $BACKOFF$. During this time interval, it shall always sense the medium as free otherwise it returns back to initial status $Awaken$ *via* an edge labeled with the event $busy[reg]!$ (in a synchronized manner with a dual event in the medium automaton). It may also leave to $Asleep$ location whenever it receives the CH signal $sleep$.

Once the clock c reaches the timeout value $BACKOFF$ at location $ReadyToSend$, the control will immediately leave to $Waiting$ location throughout an edge labeled with the action $send[reg]!$ that decrements the residual energy with an amount denoted SC. Thereafter, $Waiting$ location could be left by means of one of four outgoing edges respec-

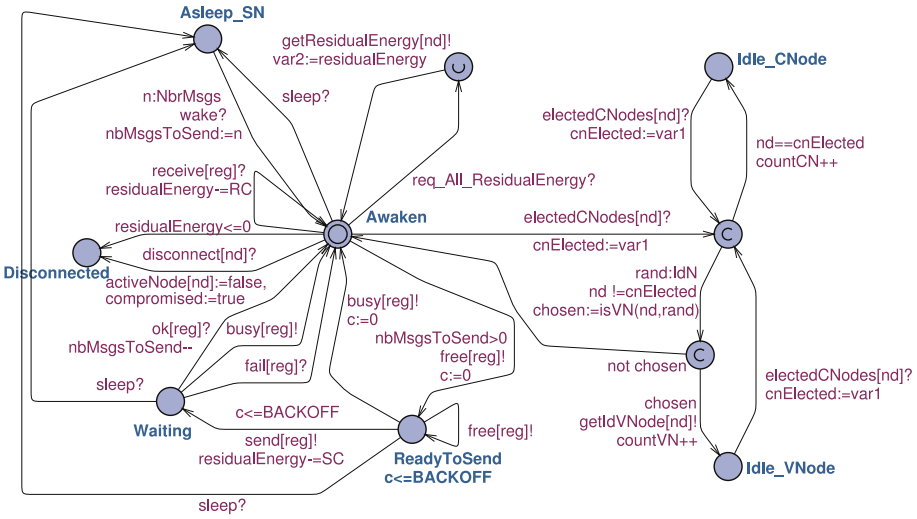

Figure 7: UPPAAL automaton of the sensing role of a WSN node

tively labeled with the following events:

1) $sleep$ broadcast event that makes the node fall into the $Asleep\_SN$ mode,

2) $busy[reg]?$ event signalling that the medium has become occupied and control has to return back to $Awaken$,

3) $fail[reg]?$ event that the medium broadcasts to tell the node about the failure of its send operation (*e.g.*, collision), which leads it to return back to $Awaken$,

4) $ok[reg]$ event that signals a successful send operation which decrements the number of messages to send and makes control return back to $Awaken$.

The cluster head CH periodically broadcasts a request to all active nodes for collecting their residual energies by means of the event $req\_All\_ResidualEnergy?$ to which the node nd shall as well immediately respond (thanks to the urgent location in-between). Hence, the value of $residualEnergy$ is sent to CH through the interaction $getResidualEnergy[nd]$.

Once the collect operation is completed, CH launches the selection cycle by interacting through the canal $electedCNode[nd]$ with

node nd (if always active) which would receive a integer value. If this value corresponds to its identifier then it immediately goes to the location $Idle\_CNode$ where it begins functioning according to the *cNode* mode until it receives again another event $electedCNode[nd]?$ which relaunches a new election cycle. However, if the received value is not nd then the node will call a routine $isVN$ parameterized with a randomly sampled integer $rand$. The result of the routine is stored in a variable $chosen$, which the control transitions depend on as follows: if $chosen$ is false then the control commutes to $Awaken$ location of the simple sensing mode else the edge to $Idle\_VN$ location is taken and the node nd commutes to the *vNode* mode which will last until a new event $electedCNode[nd]?$ is received to relaunch a new election cycle.

3.3.4. *Automaton of the cNode role*

In a similar way to sensing mode, the *cNode* automaton (depicted in Figure 8) may be disconnected and thus deactivated because of a run-out of the battery or a receipt of the $disconnected[nd]?$ event from CH. It shall also go to the asleep mode by entering a separate $Asleep\_CN$ location from which it will be able to return back exactly to the *cNode* mode $Idle\_CN$ location once it receives the awakening event. Furthermore, a *cNode* must immediately send to CH its residual energy through the canal $getResidualEnergy[nd]$ whenever it is requested to do so *via* the signal $get\_All\_ResidualEnergy$. When CH sends the event $electedCNode[nd]$ to the *cNode*, the latter will check the received value to its identifier in order to return back to the sensing mode initial location $Awaken$ or to commute into the *vNode* functioning mode (by entering $Idle\_VNode$ location).

The automaton parts related to the intended mission of the *cNode* are the following (see Figure 8):

– The *cNode* can overhear send actions made by each node e in its region. Every time it senses such an operation it increments a counter $sentMsgs[e]$ related to messages sent by e.

– Since a *cNode* may overhear messages sent throughout its region reg, each action $receive[reg]?$ causes an energy consumption of an amount equal to RC.

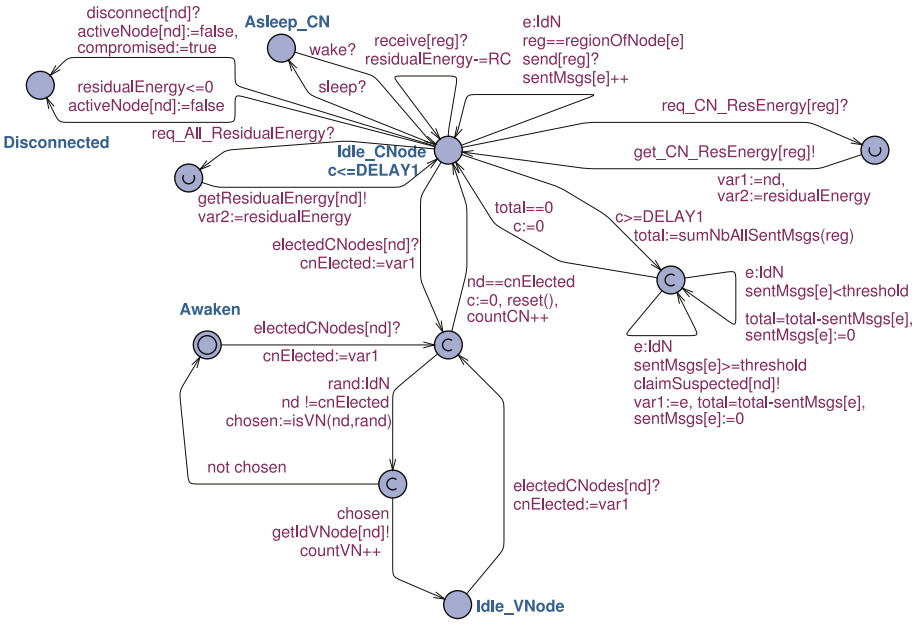

Figure 8: UPPAAL automaton of the *cNode* role

– In fact, the $Idle\_CNode$ is constrained by a timed invariant $c <= DELAY_1$ such that time progresses until the guard $c == DELAY_1$ becomes true making it possible to perform a silent move to a committed location (*via* an edge without a label). At the committed location, the *cNode* checks the number of sent messages $sentMsgs[e]$ of each node e in its region in accordance with some threshold. If $sentMsgs[e]$ does not exceed $threshold$ then the *cNode* simply resets this counter, else it sends a suspicion claim on the node e to the cluster head *via* the event $claimSuspected[nd]!$ which conveys the suspected node identifier e (in the assignment part) and then it resets $sentMsgs[e]$.

– Since the *cNode* behavior may be analyzed by *vNodes* in its region, then the *cNode* could be asked at any time to transmit its residual energy throughout the broadcast channel $req\_CN\_ResEnergy[reg]$. In such a case, it shall immediately send the requested value *via* the event $get\_CN\_ResEnergy[reg]!$.

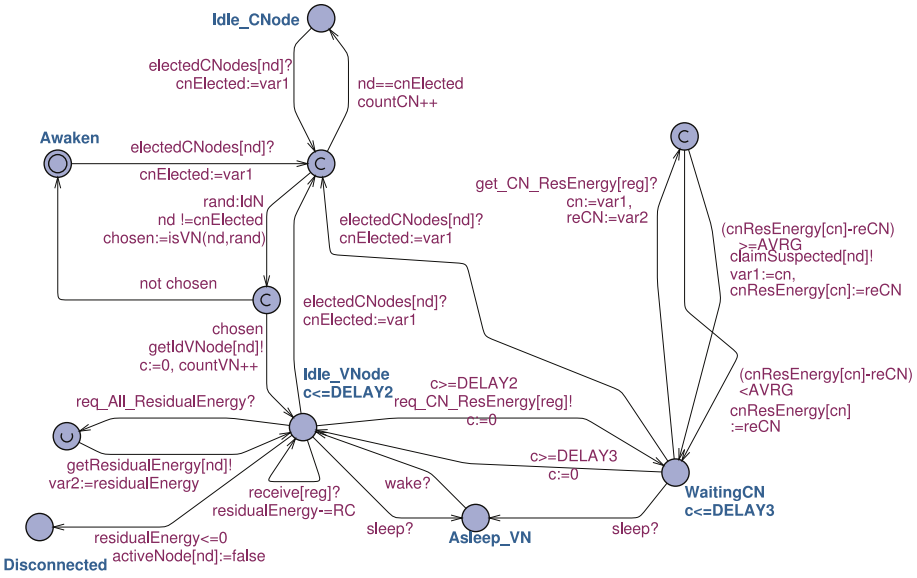

Figure 9: UPPAAL automaton of the *vNode* role

3.3.5. *Automaton of the vNode role*

Similarly to sensing and *cNode* modes, the *vNode* automaton (given in Figure 9) could be disconnected and thus deactivated because of a run-out of the battery or a receipt of the $disconnected[nd]?$ event from CH. Notice that it must go to the asleep mode by entering a separate $Asleep\_VN$ location to be able to return back exactly to the *vNode* mode $Idle\_VN$ location once it receives the awakening event. Likewise, it has to immediately send to CH its residual energy through the canal $getResidualEnergy[nd]$ whenever it is requested to do so *via* the signal $get\_All\_ResidualEnergy$. When CH sends the event $electedCNode[nd]$ to the *vNode*, the latter will check the received value to its identifier in order to either return back to the sensing mode initial location $Awaken$ or to commute into the *cNode* functioning mode (by entering $Idle\_CNode$ location).

The expected mission of our *vNode* automaton is to check the *cNodes* in its region by looking at their energy consumption values and comparing them to a threshold $AVRG$ representing some average of energy consumption of regular *cNodes*. That is why the *vN-*

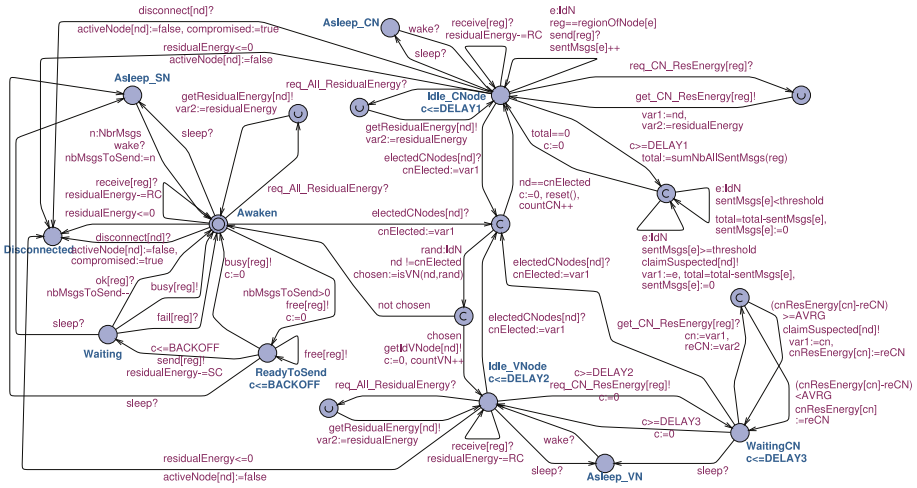

Figure 10: UPPAAL automaton modeling the whole WSN node template

ode has to periodically request from *cNodes* their residual energies. The collect periodicity is implemented by means of a clock c_2, an invariant $c_2 <= DELAY2$ on $Idle\_VNode$ location, and a guard $c_2 >= DELAY_2$ on the edge launching the collect request on a broadcast channel $req\_CN\_ResEnergy[reg]$. Once this signal is broadcast, the control moves into the new location $WaitingCN$ where it can stay for some delay $DELAY_3$ during which it would receive requested values $reCN$ along with *cNodes*' identifiers cn over the channel $get\_CN\_ResEnergy[reg]$. Every value $reCN$ received is stored and used to compute the energy consumption that is compared to the threshold $AVRG$. If it is found greater than $AVRG$, the *vNode* sends a suspicion claim on the *cNode* cn to the cluster head along the channel $claimSuspected[nd]$!. When the period $DELAY_3$ expires (that is, the guard $c_2 >= DELAY_3$ becomes true), the control moves back from $WaitingCN$ to $Idle\_VNode$.

3.3.6. *Whole automaton template of a WSN node*

Given that all functioning modes are modeled, we build the whole model of a WSN node might alternatively operate in sensing, *cNode*, and *vNode* modes. This operation is achieved by putting their au-

tomata together and merging their nodes and edges labeled with the same names. Guards and invariants of such common elements would be combined with the conjunction operator and all of their assignments would be preserved. Figure 10 depicts the automaton of the node template able to function according to the aforementioned modes.

3.3.7. *Automaton of compromised node*

In addition to the aforementioned behavior of a normal node, a compromised node can transmit false values about its residual energy to the cluster head so that this faked information improves its chances to be selected as a *cNode*. Once a compromised node commutes the *cNode* mode, it can send any number of false suspicion claims against any node in its region. Similarly, it can decide to not use the random process to become *vNode*. Instead, it can deterministically decide to commute into the *vNode* functioning mode and thereafter make any number of false suspicion claims on *cNodes* of its regions. Furthermore, a compromised node (as depicted by Figure 11) may have a greedy behavior whereby it tries to get hold on the medium by performing send actions at a high rate and without any delay whenever the medium is sensed as free. This functioning mode will deprive the other nodes from using the medium to send their data. It could also have a jamming behavior where it recurrently makes unlimited send actions labeling a loop arc upon $Awaken$ location) without checking whether the medium is free. Such behavior would certainly leads to collisions and deprive regular nodes from accessing the medium.

3.4. *Model-checking properties*

3.4.1. *System under analysis*

Once the templates have been defined, we build the whole system by instantiating $CompomisedNode$, $Node$ and $Medium$ templates as many times as necessary an then combining the produced processes with the only one process $ClusterHead$ in parallel as shown in the next example:

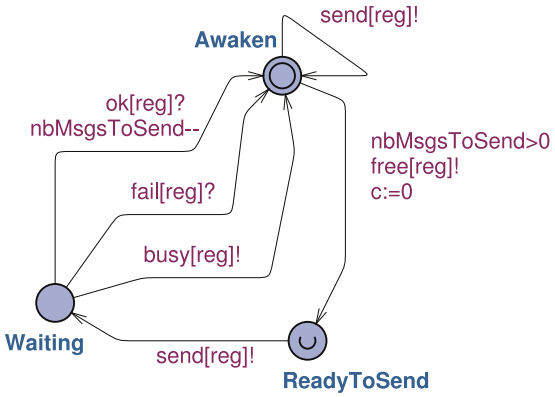

Figure 11: A fragment of UPPAAL automaton modeling a compromised node

```
// template instantiations.
medium0 = Medium(0);
medium1=Medium(1);
CH=ClusterHead();
node0 = Node(0,0);
node1 = Node(1,0);
node2 = Node(2,1);
node3 = Node(3,1);
node4 = Node(4,0);
...
nodeC=CompromisedNode(5,0);
// List one or more processes to be composed into a system.
system CH, medium0,medium1,node0,node1,node2,node3,...,nodeC;
```

3.4.2. *Model Checking Properties*

Given a model of a system, a model checker tests automatically whether this model meets a given specification. Specifications must be expressed in a formally well-defined language such as propositional temporal logics [31–33]. The verification procedure is an exhaustive search of the state space of the system. We use UPPAAL, one of the many model checking tools, which uses a simplified version of CTL (Computation Tree Logic) where nesting of path formulae are dis-

carded. Like in CTL, the query language consists of path formulae and state formulae; state formulae describe individual states, whereas path formulae quantify over paths or traces of the model. Path formulae can be classified into reachability, safety and liveness.

– **State Formulae** A state formula is an expression that can be evaluated for a state without looking at the behavior of the model. For instance, this could be a simple expression, like $c >= BACKOFF$, that is true in a state whenever the clock c is greater or equal than some constant $BACKOFF$. The syntax of state formulae is a superset of that of guards where use of disjunctions is not restricted. It is also possible to test whether a particular process is in a given location using an expression on the form $node0.Disconnected$ where $node0$ is a process and $Disconnected$ is a location. In UPPAAL, deadlock is expressed using a special state formula which simply consists of the keyword $deadlock$ and is satisfied for all deadlock states. A state is a deadlock state if there are no outgoing action transitions neither from the state itself or any of its delay successors. However, it can only be used with reachability path formulae as shown later.

– **Reachability Properties** These are the simplest form of properties. They ask whether a given state formula ϕ, possibly can be satisfied by any reachable state, *i.e.*, does there exist a path starting at the initial state, such that ϕ is eventually satisfied along that path. We express that some state satisfying ϕ should be reachable using the path formula $E <> \phi$. Reachability properties are often used while designing a model to perform sanity checks. For instance, we have checked such properties (*e.g.*, $E <> node0.Idle\_CNode$) to show that within the system model it is possible for $node0$ to be selected as *cNode* at some point in the future.

– **Safety Properties** Safety properties are on the form: "something bad will never happen". For instance, a deadlock should never occur in the model. A variation of this property is that "something will possibly never happen". In UPPAAL these properties are formulated positively, *e.g.* something good is invariantly true. Let ϕ be a state formulae. We express that ϕ should be true in all reachable states with the path formulae $A[\,]\phi$ whereas $E[\,]\phi$ says that there should exist a maximal path such that ϕ is always true. For instance, we have checked using UPPAAL that

the following property is satisfied:

$$A[\,](CH.Enquiring\ imply\ CH.i == NBR\_REGIONS$$

requiring that the cluster head CH gets hold of mediums of all regions before it could launch the selection cycle of *cNodes*.

– **Liveness Properties** Liveness properties are of the form: something will eventually happen. In its simple form, liveness is expressed with the path formula $A <> \phi$, meaning ϕ is eventually satisfied. The more useful form is the $leads\_to$ or $response$ property, written $\phi \dashrightarrow \psi$[4] which is read as whenever ϕ is satisfied, then eventually ψ will be satisfied. We thus checked the two following liveness properties: the first one expresses that whenever the battery of a node runs out, this node would be connected, whereas the second property states that any node in asleep mode shall be awaken later:

$$node0.residualEnergy <= 0 \dashrightarrow node0.Disconnected \texttt{ and } CH.Asleep \dashrightarrow CH.Awaken.$$

We give below a list of other CTL formulae expressing properties we could use to analyse our system:

– would any compromised node $node$ always eventually be detected in the future?:

$nodeC.Awaken \dashrightarrow nodeC.compromised == true$ where $compromised$ is a boolean variable to be set to true when the node is disconnected by CH.

– Is there any risk that a *cNode* be a *vNode* controlling itself at the same time?:

$$E <> node0.Idle\_CNode\ and\ node0.Idle\_VNode.$$

– Is there any risk that a *cNode* be a *vNode* controlling other *cNode* at the same time?:

$$E <> (node0.Idle\_VNode\ and\ node1.Idle\_CNode\ and\ CH.belongs(1, CH.setOfCNodes))$$

where $belongs(e, list)$ is a function we have defined to tell whether a node e belongs to some list of nodes.

4. $\phi \dashrightarrow \psi$ is equivalent to $A[\,](\phi \Rightarrow A <> \psi)$.

– is it possible that a *cNode* be compromised?:

$node0.Idle\_CNode \dashrightarrow node0.compromised == true$

– is it possible that a *vNode* be compromised?:

$node0.Idle\_VNode \dashrightarrow node0.compromised == true$

– does always the residual energy of any node decrease and eventually runs out?:

$node0.Awaken \dashrightarrow node0.Disconnected$

– can CH elect successively the same node *cNode* twice?:

$(CH.c1 == node0.DELAY1\ and\ node0.Idle\_CNode) \dashrightarrow (CH.c1 == 2 * node0.DELAY1\ and\ node0.Idle\_CNode)$

– does the number of times that a node is selected a *cNode* reach a given threshold n within some interval of time d

$(CH.c1 == 0\ and\ node0.Awaken) \dashrightarrow (CH.c1 == d\ and\ node0.countCN >= n)$

where $counterCN$ is an integer variable to be incremented every time a node is selected as *cNode*

4. Selection in practice: results from simulation

Beside formal specification, we have undertaken a more concrete simulation of our proposal regarding the energy consumption in order to compare it with our previous model (the one using pseudo-random election for *cNodes*). We used ns-3 software [34] to proceed.

In the new proposal, the *vNodes* are to model the theoretical consumption of the *cNodes* they watch over. We have chosen to use Rakhmatov and Vrudhula's diffusion model [35] to compute the consumption. This choice was driven by several reasons:

– it provides a pretty accurate approximation of real consumption, taking into account chemical processes internal to the battery such as rate capacity effect and recovery effect;

– it is one of the models already implemented in ns-3. So in our case it is an absolutely perfect theoretical model. It remains "theoretical" as

vNodes use this model to compute the expected behaviour of *cNodes* according to the few packets they sometimes hear. Meanwhile, real *cNodes* consumption computed by ns-3 core takes into account every packet actually sent or received by *cNodes*, also including packets that *vNodes* can not hear (because of distance or sleep schedule). So the values computed by *vNodes* and ns-3 core will not always be the same, which allows us to use the model.

Rakhmatov and Vrudhula's diffusion model refers to the chemical reaction happening inside the battery electrolyte, and is summarized by equation (1).

$$\sigma(t) = \underbrace{\int_0^t i(\tau)\,\mathrm{d}\tau}_{l(t)} + \overbrace{\int_0^t i(\tau)\left(2\sum_{m=1}^{\infty}\exp^{-\beta^2 m^2(t-r)}\right)\mathrm{d}\tau}^{u(t)} \quad (1)$$

where:

– $\sigma(t)$ is the apparent charge lost from the battery at t;

– $l(t)$ is the charge lost to the load ("useful" charge);

– $u(t)$ is the unavailable charge ("lost in battery" charge);

– $i(t)$ is the current at t;

– $\beta = \dfrac{\pi\sqrt{D}}{w}$, where D is the diffusion constant and w the full width of the electrolyte.

In practice, computing the first ten terms of the sum provides a good approximation (this is also the default behavior of ns-3, by the way).

We launched several simulation instances and chose to focus on the energy consumption and load balancing in the cluster. To obtain data about detection rate or false positive values of the *cNodes* scheme, the reader is redirected to our previous work [21, 22]. When we implemented our solution, we set the parameters of the simulation as detailed in Table 1.

We obtained the residual energy values for each node at each minute of the simulation. From this data we draw the average residual energy of the nodes (excluding cluster head) as well as the standard devi-

Table 1: Parameters used for simulations

Parameter	Value
Number of nodes	30 (plus 1 CH)
Number of *cNodes*	4
Probability for *vNodes* selection	33 %
Delay between consecutive elections	1 minute
Simulation length	30 minutes
Cluster shape	Squared box
Cluster length	Diagonal is 2×50 meters
Transmission range	50 meters
Location of the nodes	CH: center; others: random
Mobility of the nodes	Null
Average data sent by normal nodes	1024 bytes every 3 seconds
Data sent by *vNodes* (per target *cNode*)	1024 bytes every 5 seconds

ation. Average residual energy per minute in the batteries of the nodes is displayed on Figure 12. Increasing values at $t = 11$ minutes and $t = 15$ minutes with the use of the proposed solution traduce the recovery effect of the batteries. As expected, our proposal causes an increased global energy consumption. This is due, of course, to the new *vNode* role. *vNodes* have to wake up periodically to send requests to neighbor *cNodes* and to wait for an answer: this is energy-consuming. The estimated overhead for our solution appears on Figure 13.

Standard deviation of residual energy value in the nodes at each minute of the simulation is presented on Figure 14. During the first minutes of simulation, our solution creates a higher disproportion in load balancing due to the introduction of *vNodes* (there are more nodes assuming demanding functions). But after the seven first minutes or so, the standard deviation with our method falls below the standard deviation of previous method. This is the consequence of a better load repartition over the nodes with our solution. The difference between standard deviation with and without our simulation may look small: this is due to the model of the simulation we implemented. Given that we have a good pseudo-random numbers generator, when the number of elections get high, all nodes will roughly assume *cNode* role the same number of

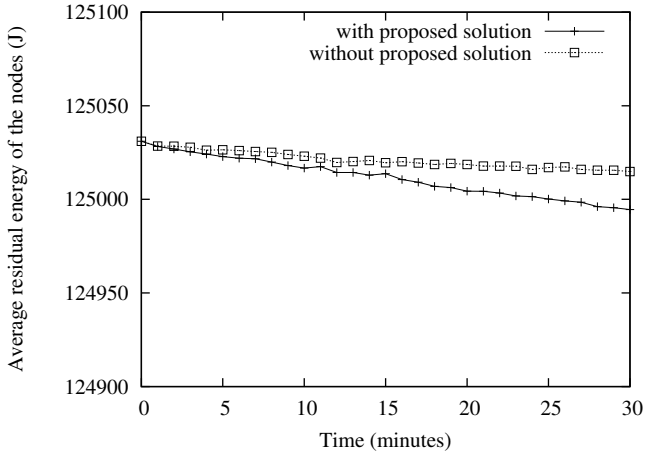

Figure 12: Average residual energy of the nodes (excluding cluster head)

times in simulation *not* using our solution. As sensing nodes all have the same activity, a correct repartition of the *cNode* roles over the time leads to a good energy balance. But in a situation where sensing nodes have different activity levels — for instance, if there is an area in the cluster when measured events occur much more often than in the other parts of the cluster — the consumption would not be equilibrated between all the nodes with the previous method; whereas our solution would deal well with this case, since *cNodes* are elected according to residual energy. Thus simulations show that the use of *vNodes* leads to a higher energy consumption, but electing *cNodes* on residual energy provides a better load repartition in the cluster.

Conclusion

Monitoring agents running on *cNodes* are used in clustered wireless sensor networks to apply rules on traffic of the nodes and to detect several types of denial of service attacks (*e.g.* flooding, black hole attacks). We proposed in this paper a new method to dynamically elect those *cNodes*, which is based on their residual energy. The aim of the proposed

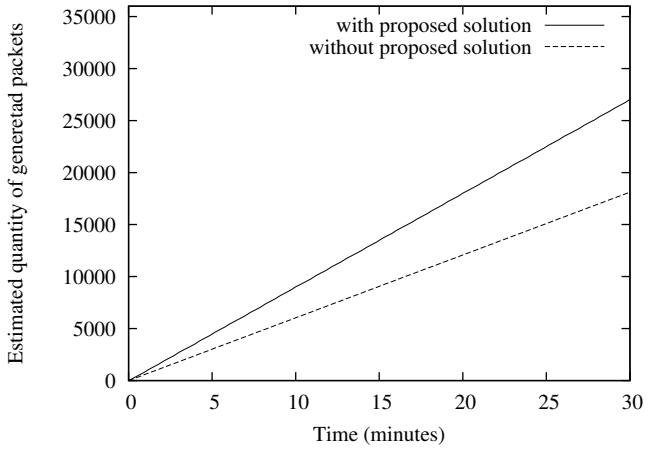

Figure 13: Estimated number of generated packets during the simulation

selection algorithm is to provide a better load balancing in the cluster, assuming that the *cNodes* role leads to a greater consumption.

We have addressed several issues related with the deterministic nature of the selection. Compromised nodes trying to systematically take over the *cNode* role are forced to abandon it for later cycles, or get detected, by *vNodes*. The *vNode* role is a new role we introduced to survey the *cNodes*. They periodically enquire for the residual energy of the *cNodes*, and match the return value with a theoretical model: were a compromised node to pretend having a high residual energy in attempt to be elected for each cycle, it would get detected. The issue of areas of the cluster uncovered by *cNodes*, depending of the activity in the cluster, is addressed by enforcing covering of the whole cluster: the cluster head is to designate additional *cNodes* if needed. Working with clusters ensures a good scalability of the solution. It is also flexible, as *cNodes* can endorse various trust-based model, and monitoring rules can be set to fight against several types of denial of service attacks. And the use of *vNodes* is resilient to a small percentage of compromised *vNodes* (depending on parameters set by user).

We also have presented a timed automata-based approach to modeling and analyzing our method; we modeled behaviors of regular nodes

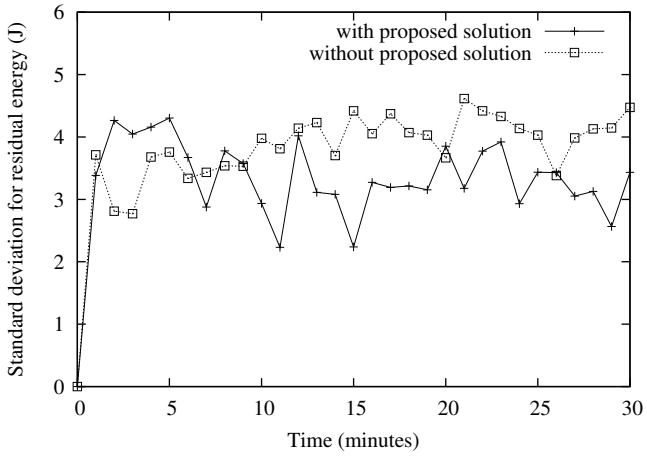

Figure 14: Standard deviation for residual energy of the nodes

that might alternatively operate in sensing, *cNode*, and *vNode* modes along with compromised nodes which act in greedy or jamming ways. Such a rigorous and formal approach helped us fix many design errors of the proposal and overcome flaws that led the system to erroneous statuses. Many properties have been expressed in Computation Tree Logic (CTL) and checked over the system under design.

Eventually, the results we have obtained through simulations show that even though using our simulation causes a higher global consumption of energy in the cluster, it provides a better load repartition between sensors.

Future works include improvements of our solution by adding monitoring of the cluster head, as well as modeling a cluster with areas of different activity levels. Concerning the formal approach, we also plane to explore a probabilistic model checking on our model to evaluate timing limits and performances. To enable such an approach, we have to adorn transitions of our automata with quantitative annotations about transmission success, failure and collision probabilities. Thereafter, more suitable tools have to be used to make use of this information in order to quantitatively deal with liveness properties.

References

[1] Jun Luo, Panagiotis Papadimitratos, and Jean-Pierre Hubaux. GossiCrypt: wireless sensor network data confidentiality against parasitic adversaries. In *Proceedings of the fifth annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON'08)*, pages 441–450, San Francisco, CA, USA, June 2008.

[2] Quentin Monnet, Lynda Mokdad, and Jalel Ben-Othman. Data protection in multipaths WSNs. In *Proceedings of the fifth IEEE international workshop on Performance Evaluation of communications in DIStributed Systems and WEb based Service Architectures (PEDISWESA'13)*, Split, Croatia, July 2013.

[3] Jalel Ben-Othman and Lynda Mokdad. Enhancing data security in ad hoc networks on based multipath routing. *Journal of Parallel and Distributed Computing*, 70(3):309–316, March 2010.

[4] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[5] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, September 2002.

[6] Marcos A. Simplicio, Jr, Bruno T. de Oliveira, Paulo S. L. M. Barreto, Cintia B. Margi, Tereza C. M. B. Carvalho, and Mats Naslund. Comparison of authenticated-encryption schemes in wireless sensor networks. In *Proceedings of the 36th Annual IEEE Conference on Local Computer Networks*, pages 454–461, Bonn, Germany, October 2011.

[7] Kui Wu, Dennis Dreef, Bo Sun, and Yang Xiao. Secure data aggregation without persistent cryptographic operations in wireless sensor networks. *Ad Hoc Networks*, 5(1):100–111, January 2007.

[8] Suat Ozdemir and Yang Xiao. Secure data aggregation in wireless sensor networks: a comprehensive overview. *Computer Networks*, 53(12):2022–2037, August 2009.

[9] Shio Kumar Singh, M. P. Singh, and D. K. Singh. A survey on network security and attack defense mechanism for wireless sensor

networks. *International Journal of Computer Trends and Technology*, May 2011.

[10] Vishal Rathod and Mrudang Mehta. Security in wireless sensor network: a survey. *Ganpat University Journal of Engineering and Technology*, 1(1):35–44, January 2011.

[11] Hongmei Deng, Wei Li, and Dharma P. Agrawal. Routing security in wireless ad hoc networks. *IEEE Communications Magazine*, 40(10):70–75, October 2002.

[12] Benjamin J. Culpepper and H. Chris Tseng. Sinkhole intrusion indicators in DSR MANETs. In *Proceedings of the first International on Broadband Networks BroadNets'04)*, pages 681–688, San José, CA, USA, October 2004.

[13] Jun-Won Ho. Distributed detection of node capture attacks in wireless networks. In Hoang Duc Chinh and Yen Kheng Tan, editors, *Smart Wireless Sensor Networks*, chapter 20, pages 345–360. InTech, December 2010.

[14] Sina Hamedheidari and Reza Rafeh. A novel agent-based approach to detect sinkhole attacks in wireless sensor networks. *Computers and Security*, 37:1–14, September 2013.

[15] Mohammad Momani and Subhash Challa. Survey of trust models in different network domains. *International Journal of Ad Hoc, Sensor and Ubiquitous Computing*, 1(3):1–19, September 2010.

[16] M. Carmen Fernández-Gago, Rodrigo Román, and Javier Lopez. A survey on the applicability of trust management systems for wireless sensor networks. In *Proceedings of the third international workshop on Security, Privacy and trust in Pervasive and Ubiquitous computing (SECPerU'07)*, pages 25–30, Istanbul, Turkey, July 2007.

[17] Mohammad Reza Rohbanian, Mohammad Rafi Kharazmi, Alireza Keshavarz-Haddad, and Manije Keshtgary. Watchdog-LEACH: a new method based on LEACH protocol to secure clustered wireless sensor networks. *Advances in Computer Science: an International Journal*, 2(3):105–117, July 2013.

[18] Gu Hsin Lai and Chia-Mei Chen. Detecting denial of service attacks in sensor networks. *Journal of Computers*, 4(18), January 2008.

[19] M. J. Handy, M. Haase, and D. Timmerman. Low energy adaptive clustering hierarchy with deterministic cluster-head selection. In *Proceedings of the 4th IEEE International Workshop on Mobile and Wireless Communications Networks*, pages 368–372, Stockholm, Sweden, 2002.

[20] Fabrice Theoleyre and Fabrice Valois. VSR: a routing protocol based on a structure of self-organization. *Studia Informatica Universalis*, 6(1):27–57, 2008.

[21] Malek Guechari, Lynda Mokdad, and Sovanna Tan. Dynamic solution for detecting denial of service attacks in wireless sensor networks. In *Proceedings of the 2012 IEEE International Conference on Communications (ICC'12)*, Ottawa, Canada, June 2012.

[22] Paolo Ballarini, Lynda Mokdad, and Quentin Monnet. Modeling tools for detecting DoS attacks in WSNs. *Security and Communication Networks*, 6(4):420–436, April 2013.

[23] Madeleine El-Zaher, Jean-Michel Contet, Pablo Gruer, Franck Gechter, and Abderrafiaa Koukam. VSR: a routing protocol based on a structure of self-organization. *Studia Informatica Universalis*, 10(3):119–141, 2012.

[24] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. HASL: an expressive language for statistical verification of stochastic models. In *Proceedings of the 5th international ICST conference on performance evaluation methodologies and tools (VALUETOOLS'11)*, pages 306–315, Cachan, France, May 2011.

[25] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. COSMOS: a statistical model checker for the hybrid automata stochatic logic. In *Proceedings of the 8th international conference on quantitative evaluation of systems (QEST'11)*, pages 143–144, Aachen, Germany, September 2011.

[26] Llanos Tobarra, Diego Cazorla, Fernando Cuartero, Gregorio Díaz, and Emilia Cambronero. Model checking wireless sensor network security protocols: TinySec + LEAP + TinyPK. *Telecommunication Systems*, 40(3–4):91–99, April 2009.

[27] Jinat Rehana. Security of wireless sensor network. Technical report, Helsinki University of Technology, 2009.

[28] Ossama Younis and Sonia Fahmy. HEED: a Hybrid, Energy-Efficient Distributed clustering approach for ad-hoc sensor networks. *IEEE Transactions on Mobile Computing*, 3(4):366–379, October 2004.

[29] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, volume 3185, pages 200–236. Springer, 2004.

[30] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[31] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for real time systems. In *Proceedings of the fifth Symposium on Logic In Computer Science (LICS'90)*, pages 414–425, 1990.

[32] T. A. Henzinger. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.

[33] G. J. Holzmann. Software model checking. *NATO Summer School, IOS Press Computer and System Sciences, Marktoberdorf Germany*, 180:309–355, August 2000.

[34] The network simulator – ns-3.

[35] D. Rakhmatov and S. Vrudhula. An analytical high-level battery model for use in energy management of portable electronic systems. In *Proceedings of the International Conference on Computer Aided Design (ICCAD'01)*, pages 488–493, San Jose, CA, USA, November 2001.