Research by Experimentation
for Dependability on the
Internet of Things

D-1.3 – Report on Runtime Assurance

Grant Agreement no: 317826
www.relyonit.eu

Date: November 30, 2014

Author(s) and affiliation: James Brown, John Vidler, Ibrahim Ethem Bagci, Utz Roedig (ULANCS); Carlo Alberto Boano, Felix Jonathan Oppermann, Marcel Baunach, Kay Römer (TUG); Marco Antonio Zuniga, Faisal Aslam, Koen Langendoen (TUD).

Work package/task: WP1
Document status: Final
Dissemination level: Public
Keywords: Wireless sensor networks, temperature, interference, received signal strength, timing, analytical models, runtime assurance.

Abstract This document presents (i) the generic RELYonIT runtime assurance framework and (ii) instances of this framework for the temperature and radio interference models derived in Deliverable D-1.1. The generic runtime assurance framework is composed of three elements: violation detection, violation verification, as well as reporting and remediation. After describing in detail each of these elements, we show how the framework can be used to build a runtime assurance system for environments in which temperature and radio interference represent a major challenge. We consider these two environmental factors individually and examine lightweight detection techniques which can infer potential violations from existing information held in the system. We then look at verification methods showing how to compare data collected at run-time to instantiate the models devised in Deliverable D-1.1 with pre-deployment models used to parametrize communication protocols. We finally examine reporting and remediation and show how alarms can be triggered as soon as a model violation occurs as well as how data recorded during verification can be reported to aid remediation.

Disclaimer

Contents

List of Figures

List of Tables

Executive Summary

Predicting and optimising the performance of systems IoT systems to a given environment enables performance guarantees to be offered to the application. However, these predictions assume that the model of the environment is correct for the lifetime of the application.

In many situations the environment may change over time, and these changes may invalidate the environmental model and any predictions or optimisations that are based upon it. It is the purpose of Runtime Assurance to monitor the performance of the system and to confirm the environmental models when performance degradation occurs and to raise alarms when the environmental model is no longer representative.

Towards realising a runtime assurance system we have created a generic runtime assurance framework composed of three elements; violation detection, violation verification, and finally reporting and remediation. We examine the use of this framework with two environmental models: temperature and radio interference.

For each of these two aspects we have examined lightweight detection techniques which can infer potential violations from existing information held in the system, and methods for verification describing how data can be collected at run-time to instantiate models derived in Deliverable D-1.1 and compare with pre-deployment model instances. Finally, reporting and remediation is discussed showing how alarms can be triggered on model violation, and how data recorded during verification can be reported to aid remediation.

Runtime assurance is a small integral component of the RELYonIT system. It builds up on models and tools for parameterisation developed in the earlier tasks of WP1. Close links with Protocol Models (WP2) and their parameterisation is required to gain the relevant understanding of how protocols are affected by the environmental effects so the correct performance metrics can be monitored for model violation detection. Runtime assurance systems must also collect and provide the relevant information to Runtime Adaptation (WP3) and potentially to the Protocol Selection and Parameterisation stages in situations where environmental changes are significant.

1 Introduction

In Deliverable D-1.1 [10] we have devised environmental and platform models allowing us to capture and predict how specific environmental conditions vary with time and how their variation affects the platforms that support IoT applications. This work was extended in Deliverable D-1.2 [5], where we developed tools to capture application-specific environmental data prior deployment to parametrise these models. These models and tools can be used in conjunction with WP2 (Environment-Aware Protocols) to predict the performance of wireless sensor networks protocols [1, 6] as well as with work undertaken in WP3 (Configuration and Runtime Support for Dependable Applications) to automatically obtain an optimal protocol configuration.

Predicting and optimising the performance of a sensor network in a given environment enables applications to satisfy certain performance guarantees. However, these predictions assume that the model of the environment is correct. In some scenarios the environment may change, e.g., when a new Wi-Fi access point is installed inside a building, or in case sensor nodes are deployed in an area exposed to extraordinary climate conditions. Such a change may invalidate the model and any predictions or optimisation based thereon.

It would be beneficial to monitor the environment during runtime to gain knowledge as to when the environment changes such that remediation can take place prior to the application being adversely effected. Techniques for runtime assurance are needed to monitor for changes in an environment that violate the model parameterised pre-deployment and to raise an alarm. This document examines such techniques in detail.

The rest of the document is structured as follows. Chapter 2 provides a description of the RELYonIT runtime assurance framework and of its three building blocks: detection, verification, and reporting. Chapter 3 presents an instance of this framework for the temperature models derived in D-1.1 [10] and shows a system that verifies adherence to the temperature environmental model captured before deployment. Chapter 4 focuses on radio interference; first examining a number of data sources that can be used to infer violations before then looking at verification; comprised of model instantiation and model comparison stages. The final chapter presents our conclusions for this deliverable.

2 Framework

The responsibilities of the runtime assurance component are threefold:

1. Monitoring aspects of systems performance for indications of possible volitions of the environmental model;
2. Verifying that model violations exist;
3. Reporting violations with data collected to be used for remediation.

The component is intended to operate on wireless sensor nodes at runtime, and should therefore share system resources with the application.

As there are potential problems with the inherent non-deterministic run length of some of these operations, we minimise any adverse effect on the application through cooperating closely with the application. To this end, we leave it to the application to schedule operations during runtime between its own tasks such that any impact is mitigated or minimised to the greatest degree.

Facilitating close cooperation between the application and the runtime assurance component requires a well-defined framework and set of interfaces. In this section we will document the framework devised during the project, and the three subcomponents of which it is composed, namely; *detection*, *verification* and *reporting*.

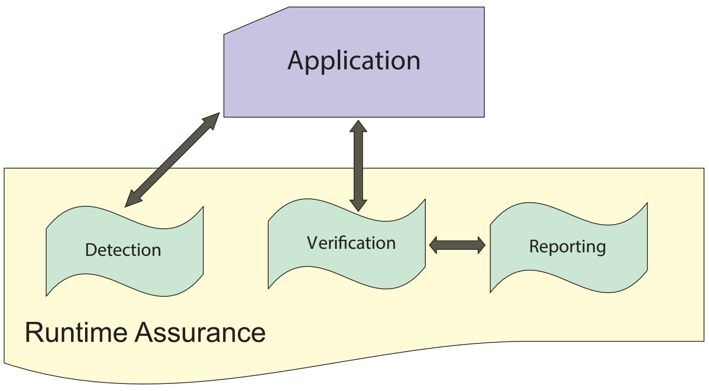


Figure 2.1: Framework Diagram

2.1 Violation Detection

The first, and most frequently used element of the runtime assurance framework is the *violation detection* subcomponent. The detection module will run alongside the application and regularly checks for signs that the environmental model has been violated.

As embedded systems are often resource constrained, it is important to ensure that the method used for violation detection is lightweight as to limit any potentially adverse effect on the application. Whilst the amount of RAM and ROM the detection module requires should be limited, it is the processing requirements that are actively constrained. Determining exactly when this component is scheduled is highly application dependant, and the amount of processor time (the run-time for the component) should be carefully restricted to minimise the possibility of affecting the application's own scheduled operations.

The detection module could be scheduled to run periodically based on a timer, however the effect on the application of doing so would be difficult to predict. Taking the example of a simple minimum/maximum temperature model, the task of taking regular measurements of the ambient temperature is quite a lightweight, fast task, and could easily be worked in to the runtime of the application at any point. Interference modelling, on the other hand, requires high frequency sampling of the channel, along with real-time processing, both of which are time consuming and very resource-heavy. Performing such a heavy-weight process alongside the application at runtime would detrimentally affect the performance of the application. A safer policy is to allow the application to schedule the detection process around it's own operations.

One simple approach to test if a violation may have occurred is to execute the tools developed in the previous deliverable (D1.2) during runtime. Data could be collected to create environmental models which could then be compared against the model collected prior to runtime, and any significant differences would signify a violation has occurred. This approach would depend on the data requirements and the processing overheads of re-running the environmental model generation code.

Another approach is to infer a violation has occurred by measuring other properties of the system. In WP2 much work has studied how changes in aspects of the environment may affect certain system performance metrics. A number of aspects of the system may degrade if the environmental model is violated - the energy consumption of the device may increase with radio interference due to additional idle listening, for example. Communication metrics may also change with the environment such as a drop in signal strength with temperature or a drop in PRR with increased interference - changes in either of which could signal to the runtime assurance component that the environmental model may have been violated.

The advantage of evaluating these effects is that often it would require minimal processing and memory overheads as many of these metrics are recorded by the system during normal operations, although this approach would not offer the highest detection accuracy as performance may also be affected by non-environmental influences.

2.1.1 Interface

To support the functionality outlined above, the detection module interface provides a single function outlined below.

```
int relyonit_rta_detection(void *) ;
```

The function is intended to be invoked periodically by the application to check for potential environmental model violation. The function will do no heavy processing at this stage and will return control to the application in shortest possible time.

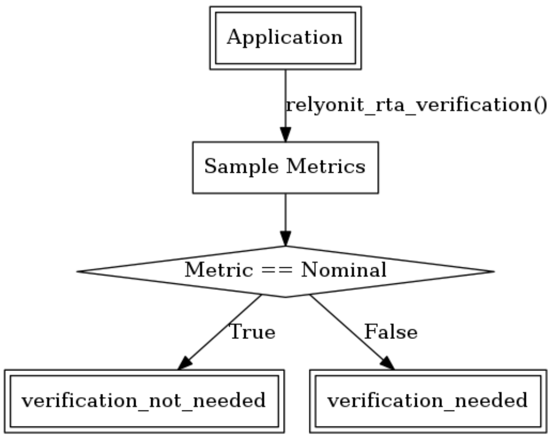


Figure 2.2: The logical model for the *relyonit_rta_detection* function, error states are omitted for clarity

The application should call this function at a sufficient frequency to detect changes and to enable rapid adaptation. The minimal frequency will depend on the environmental aspect being monitored and how rapid that aspect can change. As well as periodically, the application should also call the function at points where the performance of the system is suspected to be degraded such as after a run of failed packet transmissions.

The function will return one of three different codes; *verification_needed*, which will be represented by a 1, *verification_not_needed*, which will be represented by a 0, and *error* which will be represented by a −1, as shown in Figure 2.2. In the event that the function returns *verification_needed*, the application should invoke the verification function at its earliest convenience.

2.2 Verification

The second subcomponent of the runtime assurance framework is *verification*. This module is executed by the application after the detection module detects a potential environmental model violation (as indicated by a *verification_needed* result from the detection module). The module is responsible for verifying that a violation has actually occurred and then to notify the reporting component to signal the violation.

Verification of a violation will often involve re-running the environmental data capture tool for the specific environmental aspect derived in D1.2. The data recorded will be used to instantiate a real-time instance of the model that will be compared to the one captured prior to application deployment. In the event that the deviation between the two models is above a model-defined threshold then a violation is considered to have occurred.

The processing and memory overheads of model violation verification will depend on the model being verified, although are generally expected to be significantly higher than that of the *detection* subcomponent for non-trivial models. Again taking the temperature model example, the minimum and maximum temperature observed over a time period may be sufficient and would require limited resources, whereas for interference modelling, the model in the form of a *PDF* of the observed interference may need to be captured, which is far more resource intensive.

As the resource requirements for verification can vary anywhere from minor to extreme, scheduling when the verification algorithm should run is under the control of the application. The application will be notified through the result of the detection module documented previously, and if further verification is required, the application can then finalise any priority tasks before calling this component.

2.2.1 Interface

To support the functionality of the verification module outlined above, the module interface provides a single function outlined below.

```
int relyonit_rta_verification(void *) ;
```

Invoked by the application after the detection module has signalled the presence of a potential violation, the function attempts to re-create the model with new data and compare it to the current model, and unlike the detection function which simply attempted to infer a violation by using information present in the system, the verification system will actually check a violation has occurred.

Because generating a model is both model-specific and likely to be computationally and resource-expensive, no predictions can be made for the run-time of this function. Therefore, the application should, before invoking the verification function, first complete any high priority tasks.

Although no guarantees on required processing time are defined in the interface specification, some indication of the expected runtime for a given model should be given the application developer.

The function will return three different codes; *violation*, which will be represented by a 1, *no_violation*, which will be represented by a 0 and *error*, which will be represented by a -1, all as shown in Figure 2.3. The return states of *violation* and *no_violation* are purely to inform the host application of the result, and can be used to adjust application specific state, if required.

On the detection of a violation, *before* returning the state to the application the violation function will signal to the reporting system that a violation has occurred. The reporting mechanism is discussed in the next section.

Depending on the application, it may be possible for the process to continue during a violated state in a best-effort capacity until remediation can take place. The application should continue calling the runtime detection and verification as per normal operations, to enable the runtime assurance processes to inform the application that the state has returned to normal, or alternatively to allow further information to be gathered and reported during an ongoing violation.

2.3 Reporting and Remediation

The reporting and remediation module of the runtime assurance framework is responsible for both signalling an alarm, and initiating any remediation operations when a violation of the

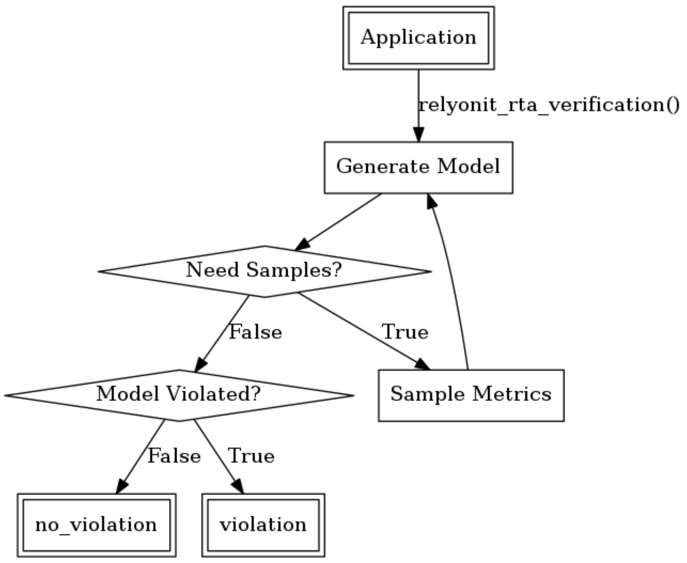


Figure 2.3: The logical model for the *relyonit_rta_verification* function, error states are omitted for clarity

environmental model has occurred.

The implementation to signal the event to a controller is left up to the implementer, but may, for example take the form of a header bit flag piggy-backed in existing application packets sent to the sink. Alternatively, an entirely separate message may be sent to signal the event. When beneficial, the module should also transmit information gathered by the verification system that can be used for remediation. Information such as the current temperature or the generated interference model instance could also be forwarded to aid in remediating the current performance issues, and may provide hints to Runtime adaptation of Protocol Parameters (Task 3.3) on how to adapt the system. In situations where runtime adaptation is not possible the information may be used as input to rerun Protocol Selection and Parametrization (Task 3.2).

The module will be directly interfaced with the verification system and as such has no user callable functions.

2.4 Case Study

The implementation of runtime assurance will be dependent on the environmental model that runtime assurance is validating. Each implementation will apply the framework defined in the previous sections. In this section will examine a case study to show the framework in action and demonstrate runtime assurance for the Car Parking scenario used for the final integrated demo. The demonstrator will focus on energy consumption of the radio which is heavily affected by radio interference. As such runtime assurance in this case must monitor the interference environmental model. A more detailed examination of runtime assurance for inference and the effect of interference on energy is given in Section 4.

The demonstrator will focus on idle energy consumption. The radio idle energy model defined in D2.2/2.3 will be used to define the expected energy consumption of the device. This model employs a simplified version of the interference model using the channel busy-to-idle ratio rather than the complete idle and busy PDF's. Runtime assurance must ensure the actual interference observed by the device does not deviate significantly from this model and if so an alarm must be raised.

2.4.1 Detection

To infer if interference has changed, this function checks if the energy consumption of the device has deviated from that predicted by the model. ContikiOS by default records the on-time, and thus, energy consumption of various components of the system, and these statistics can be accessed by the detection subcomponent for comparison. For our purposes, the on-time of the radio receiver is monitored and compared with the predicted value that is generated at compile time and stored on the device. If a significant deviation (greater than 5%) is seen, we infer that a violation of the interference model has occurred and signal that verification is required by means of return code of $verification\_needed$.

The function is called by the application every minute via the use of a simple timer (in our specific case, Contiki's *etimer*). When the etimer expires the application will first finish any priority tasks before calling the detection function. A minute was selected as this provided enough time that the effects of small fluctuations in interference are averaged and less detection checks result in $verification\_needed$.

The function meets the goal of being as lightweight as possible, and is implemented as a simple test comparing two integers, namely the predicted radio on-time and then actual, and returns an appropriate integer status code. In our example the function is executed once per minute, although there are no side effects to calling it at a higher frequency.

2.4.2 Verification

The verification function is executed by the application after the detection module has signalled verification being needed. The application should first perform any tasks of high priority before calling the verification function to prevent scheduling conflicts with the verification subcomponent. The function itself takes approximately one minute to execute.

The implementation of the verification module borrows from that of the interference data collection tool outline in D1.2. The function executes for one minute, sampling RSSI at a high frequency of 30Khz. If the observed RSSI sample is above the threshold of -77 dbm then a busy counter is incremented otherwise an idle counter is incremented. The sampling is done in real-time with no breaks and thus requires the full use of the processor with interrupts disabled. At the end of the RSSI sampling period, the channel state, i.e., the percentage that the channel is busy is calculated and compared with the value recorded prior to deployment stored on the device at compile time. If the value deviates by more than 10 percentage points then a violation has occurred.

When a violation has occurred the verification function will call the reporting function providing the measured channel state and then return *violation* to the application signifying a violation has occurred. If no violation is detected and no reporting is necessary, a return value of *no_violation* is given to the application signifying no violation has actually occurred.

When the system is in a violation state, the demo application will attempt to continue to operate as normal whilst any remediation action is taking place. The detection subcomponent will continue to be executed once a minute which will re-trigger the verification process whilst the violation continues to be present.

2.4.3 Reporting and Remediation

When a violation of the model is verified by the verification system, the reporting function is invoked. The function will generate a message that will be sent via the RIME stack to the control system. The packet will contain the ID of the system along with a flag signifying that a model violation has occurred. No additional reliable mechanisms on top the existing RIME mechanism are implemented to ensure the packet reaches the control system, and as the application will rerun verification every minute during a violated state, the violation report will continue to be generated and transmitted until remediation is performed.

To aid the remediation system in reevaluating the interference model, the channel state as evaluated by the verification process is added to the data in this message.

3 Temperature

This section presents an instance of the runtime assurance framework for the temperature model derived in WP1 [5, 10]. Before describing the implementation of the three elements composing our framework (violation detection, violation verification, and reporting/remediation), we first shortly summarize how the temperature model was conceived and its exploitation in the newly designed environment-aware protocols.

3.1 Parameters of Interest

In deliverable D-1.1 [10], we have captured the behaviour of the environment $\mathcal{E}$ in terms of four thermal properties: hotness, periodicity, change of rate, and maximum and minimum temperature range. As we argued in [3, 11], the most important of these properties are the **maximum and minimum temperatures** recorded on each node i (Equation 3.1), as they bound the network performance. Indeed, denoting f_i as the time series of the temperature observed by a node i, the knowledge of the maximum and minimum temperature at each node:

$$(\max_i, \min_i) = (\max\{f_i\}, \min\{f_i\}) \tag{3.1}$$

can be used to predict the worse-case attenuation of received signal strength that a wireless link experiences in the presence of temperature fluctuations. This information, together with the platform model devised in WP1 capturing the signal strength attenuation for a specific hardware platform [2, 10], can be used by every sensor node to predict the expected attenuation of the signal-to-noise ratio (SNR) and derive the parameters of our newly designed temperature-aware MAC protocol [11, 12].

Such a prediction, however, is only valid if the values used to compute Equation 3.1 faithfully represent the characteristics of the environment surrounding the nodes. In case the characteristics of the time series collected to derive the environmental model used to parametrize a given communication protocol differ from the temperature fluctuations actually occurring at the deployment location, the environmental model becomes invalid. Such violation needs to be reported to the system, and a remedation action needs to be taken (e.g., re-parametrization of the communication protocol).

3.2 Violation Detection and Verification

Each sensor node keeps track of the maximum and minimum temperatures recorded prior deployment. To detect a violation of the environmental model, these values need to be compared against the current *on-board*[1] temperature. This requires each sensor node to periodically

[1]Notice that the *on-board* temperature of sensor nodes is often higher than *air* temperature measured by traditional weather stations, as wireless sensor nodes are exposed to direct sunlight or are enclosed into

monitor this value by using a dedicated on-board temperature sensor[2] and by comparing this value with the maximum and minimum temperature computed prior deployment.

Frequency. The on-board temperature of sensor nodes typically varies slowly over time (a variation of 1°C cannot occur at time-scales of micro- or milliseconds), and therefore the violation detection feature can be implemented as a low-priority task executing every P seconds.

To derive P, we use the maximum rate of change on a node i captured prior deployment as explained in [10] by identifying the steepest slope of the temperature series f_i:

$$R_i = \max_t (f_i(t + \Delta t) - f_i(t)) \tag{3.2}$$

We have observed in common outdoor deployments that the highest rate of change typically occurs when a node receives the first sun-rays at the beginning of the day. In such scenarios, a node can experience an increase in its on-board temperature up to $R_i = 2°\text{C/minute}$ [4]. The knowledge of R_i together with the effect of temperature on transmitted and received power of the platform of interest (α and β, respectively, see [10]) allows us to compute the maximum variation in dB of the SNR on a time-frame P.

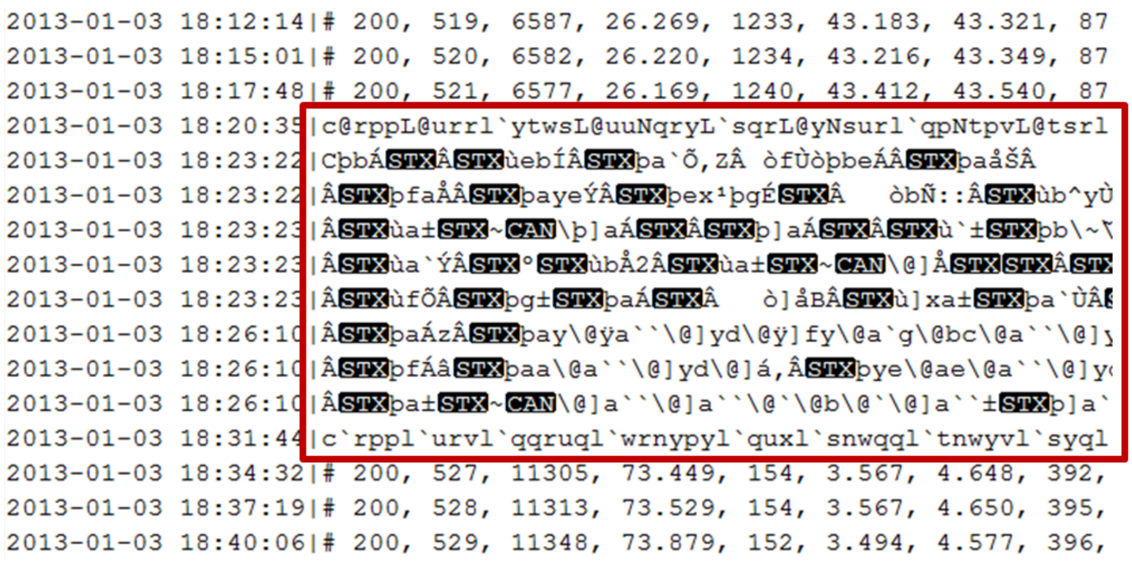
For example, when two Maxfor MTMCM5000MSP sensor nodes (TelosB replicas with $\alpha = 0.08$ and $\beta = 0.08$ [3]) experience an increase of temperature at the beginning of the day of $R_i = 2°\text{C/minute}$, one can expect an attenuation of the signal up to 0.32 dB within one minute. Selecting $P = 20$ seconds will therefore allow us to make sure that any violation of the model will be detected before the SNR can be influenced by 0.1 dB, a value sufficiently low not to harm packet reception.

Integrity of logs. Measuring the on-board temperature periodically every P seconds also gives the system the possibility to prevent corruption of any log information sent over the USB back-channel. During our experiments, we have indeed observed that common USB serial connections used for data logging and node programming are unable to cope with very fast temperature fluctuations, as they often result in de-synchronization of the USB sender and receiver. In the presence of such variations, the USB serial port looses synchronization with the sensor node and the characters forwarded to the USB back-channel become temporarily unreadable, as shown in Fig. 3.1. Since standard wireless sensor nodes do not handle this issue autonomously, a node has the possibility to re-initializes the USB port if the temperature between two consecutive temperature readings changed by more than $TEMP_{DCOSYNCH}$ °C.

Verification mode. Data retrieved from inexpensive sensors is known to be often brittle, i.e., a sensor reading generating a violation may be the result of an outlier or of a fault in the sensing process. Therefore, as soon as a violation of the model is detected, we repeat the reading $K = 3$ consecutive times and carry out a majority voting. If the model has actually been violated, the system enters the reporting and remediation mode.

transparent packaging absorbing infra-red radiation [4].

[2]The vast majority of wireless sensor nodes embed a dedicated on-board temperature sensor. If the latter is not available, several low-power micro-controllers such as the MSP430 offer the possibility to obtain a rough estimate of the on-board temperature from a built-in temperature sensor using a specific input of the ADC.



```
2013-01-03 18:12:14|# 200, 519, 6587, 26.269, 1233, 43.183, 43.321, 87
2013-01-03 18:15:01|# 200, 520, 6582, 26.220, 1234, 43.216, 43.349, 87
2013-01-03 18:17:48|# 200, 521, 6577, 26.169, 1240, 43.412, 43.540, 87
2013-01-03 18:20:35|c@rppL@urrl`ytwsL@uuNqryL`sqrL@yNsurl`qpNtpvL@tsrl
2013-01-03 18:23:22|CþbÁSTXÂSTXùebÍÂSTXpa`Õ,ZÂ òfÙòþbeÁÂSTXpaåŠÂ
2013-01-03 18:23:22|ÂSTXþfaÅÂSTXþayeÝÂSTXþex¹þgÉSTXÂ   òbÑ::ÂSTXùb^yÙ
2013-01-03 18:23:23|ÂSTXùa±STX~CAN\þ]aÁSTXÂSTXþ]aÁSTXÂSTXù`±STXþb\~
2013-01-03 18:23:23|ÂSTXùa`ÝÂSTX°STXùbÅ2ÂSTXùa±STX~CAN\@]ÅSTXSTXÂ
2013-01-03 18:23:23|ÂSTXùfÕÂSTXþg±STXþaÁSTXÂ   ò]åBÂSTXù]xa±STXþa`ÙÂ
2013-01-03 18:26:10|ÂSTXþaÁzÂSTXþay\@ÿa``\@]yd\@ÿ]fy\@a`g\@bc\@a``\@]
2013-01-03 18:26:10|ÂSTXþfÁâSTXþaa\@a``\@]yd\@]á,ÂSTXþye\@ae\@a``\@]y
2013-01-03 18:26:10|ÂSTXþa±STX~CAN\@]a``\@]a``\@`\@b\@`\@]a``±STXþ]a`
2013-01-03 18:31:44|c`rppl`urvl`qqruql`wrnypyl`quxl`snwqql`tnwyvl`syql
2013-01-03 18:34:32|# 200, 527, 11305, 73.449, 154, 3.567, 4.648, 392,
2013-01-03 18:37:19|# 200, 528, 11313, 73.529, 154, 3.567, 4.650, 395,
2013-01-03 18:40:06|# 200, 529, 11348, 73.879, 152, 3.494, 4.577, 396,
```

Figure 3.1: Unreadable serial output in the presence of sudden thermal variations.

3.3 Reporting and Remediation

Once a violation is detected, it is reported to the central system such that remediative actions can take place.

As will be later explained in Section 4.4, there are two available options: the runtime assurance alarms can either be sent as an explicit message, or piggybacked on existing communications (an optimal solution in high throughput applications where packets are frequently generated).

Upon receiving the alarm messages, each node can either try to compensate for the violation by adapting runtime parameters (using the Runtime Adaptation of Protocol Parameters module) or, if the change in environment is quite significant, it will re-compute the selection of protocols and their parametrization.

Implementation. As part of the complete implementation of runtime assurance for the 2nd integrated demo, we implemented reporting as an explicit message. However, as data retrieved from inexpensive sensors is known to be often brittle (i.e., a sensor reading generating a violation may be the result of an outlier or of a fault in the sensing process), nodes do not rely on single sensed values. Instead, as soon as a violation of the model is detected, each node repeats the reading $K = 3$ consecutive times and carries out a majority voting: if the model has actually been violated, an explicit message is sent so that the system enters the reporting and remediation mode. The implementation is in detail and evaluated as part of the 2nd integrated demo deliverable (D4.4.).

4 Radio Interference

In this section we will present an instance of runtime assurance for radio interference. Interference measurement consideration will be examined before looking at each of the elements of the runtime assurance framework.

4.1 Measurement Considerations

As described in Deliverable D-1.1 [10], the radio interference model is formed by computing a *CDF* of the idle and busy periods over time. An idle period is a period where there is no interference or where interference is below a predetermined threshold, whereas a busy period is one where interference is contiguously above the threshold. The threshold is defined as the maximum value where interference has no effect on a transmission, and normally, this *CCA* threshold is set to -77 dBm.

Continuous sampling of the channel over a specific time period is required to gather the required data for the model. If the sampling is too slow or if there are spaces in sample recordings, periods of interference will be missed. To maximise the accuracy of the model we chose a high rate of contiguous scanning, recording one sample every radio symbol period, which equates to approximately 30 kHz.

As described in Deliverable D-1.2 [5], for a system with the resources of a typical embedded system such as a t-mote [7, 8], to scan at this rate, the clock must be set to maximum rate, with interrupts disabled to provide the required performance, and the system cannot execute any other task during sampling or spaces in sampling will occur.

We suggested a scanning period of 5 minutes for each individual model, for reasons as discussed in Deliverable D-1.2 [5]. This value was selected after experimental investigation and was adequate to collect a representative view of the channel.

Due to the high system requirements during this measurement and sampling phase, sampling on a node at runtime should be avoided to reduce any adverse effects on the application, and will additionally impact on both the detection and verification subcomponents' operation.

4.2 Violation Detection Methods

As described in the previous subsection, sampling radio interference and instantiating the *CDF* interference model consumes significant resources and would have an adverse effect on any application. The detection component of the runtime assurance framework should be as lightweight as possible, and as such, performing any resource intensive operations is not a viable option. Instead, the detection module should infer from other relevant data present in the system to detect violation of the model that may have occurred.

There are many ways to infer changes in interference from performance data collected by the system during normal operation. In this section we will examine three examples that have been investigate for infering changes in interference.

4.2.1 Time

Many environments are cyclic, with multiple distinct periods of operation. In a typical office setting for instance, two main periods can be expected; one during office hours, and another other during non-office hours. During office hours interference will usually be more significant than during non-office hours (due to usage of Wi-Fi access points and microwave ovens), and therefore any model recorded in one time period would be violated when the second time period was entered. In examples such as this, time is an acceptable trigger of the verification system. Figure 4.1 is captured from a university office and illustrates this cyclic behaviour.

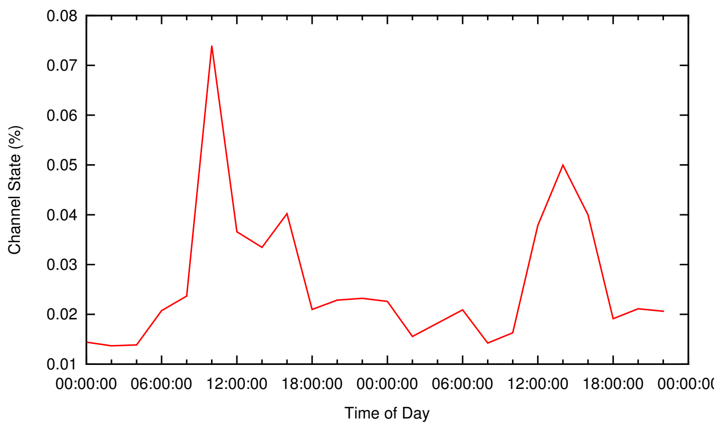


Figure 4.1: Cyclic behaviour of office environment.

In this situation, there are two possible modes of operation. The first is to use a simple timer where after a fixed period the need for verification is signalled. The timer is set to check every n minutes, and will depend on how quickly the device needs to be able to detect and flag a violation. This will depend on factors such as the application requirements, the differences in the two periods and the cost of verification. The second is to use the time of day - this method is more relevant in an environment whose cyclic behaviour is based on time of day. At these set specific times of the day the system would signal that verification was required.

4.2.2 Energy Usage

A second data type that can be used to infer changes in interference is the energy usage of the radio, or system as a whole. Interference can affect communication in a number of different ways, such as increasing the number of required transmissions or increasing idle listening.

The radio is typically the most dominant energy consumer of an embedded system. Many communication protocols must duty cycle to conserve sufficient energy to enable adequate system lifetime, and duty cycling systems must first synchronise before data can be exchanged.

In modern non-TDMA based systems, MAC protocols utilise packet strobing and energy detection. The transmitter will repeatedly send out a packet with a fixed spacing until an acknowledgement is received, or after a configured number of strobes without ACKs signals a failure. The receiver will regularly wake up and enable its radio for a short period of time to sample the energy of the channel, taking a pair of samples with configured schedule such that if a strobe is present one of the samples will detect it. If a strobe is detected, the receiver will keep its radio on to receive the following strobe in full.

Using this method, as the detection of energy within the channel is used to infer the transmission of a strobe, any additional interference may be falsely detected as a strobe and will cause the device extend listening and thus, use more energy.

Figure 4.2 illustrates this affect. As the interference level increases so does the radio on-time and thus energy of the device. The figure shows that this is the case in both an idle scenario - where no packets are being exchanged with the node - and also in an active scenario where there is a transmission every 2 seconds.

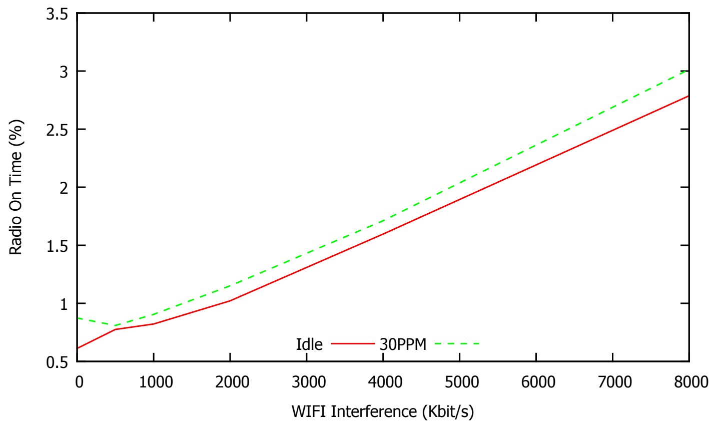


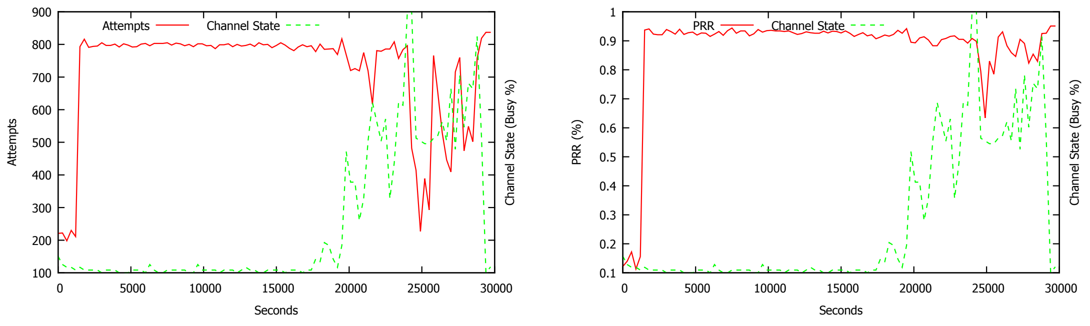
Figure 4.2: Radio On time using ContikiMAC under different interference loads.

This demonstrates that energy usage of a system can infer changes in interference patterns in some situations, and this is especially useful as energy usage is easily computed (and even stored by default) in many systems such as ContikiOS and TinyOS. Whilst changes in energy usage may not always be caused by changes in interference, it would be a significant signal that the environment has changed and a more in-depth analysis and verification is needed.

Using energy to infer possible violations of the interference model is a lightweight solution that fulfils the requirements of the framework for detection.

4.2.3 Communication Statistics

Communication statistics are another source that can be used to infer violations in the interference model. The most significant effect that interference has on radio transmissions is in causing collisions. Collisions will reduce the packet reception rate in an unreliable system or increase the number of retransmission in a reliable system. Figure 4.3(b) shows an example of this, such that as interference becomes more dominate PRR drops.



(a) Transmission Attempts (b) PRR

Figure 4.3: interference affects communication statistics.

Monitoring both of these statistics - which are recorded by embedded network stacks by default - is a lightweight mechanism to detect changes in interference.

A second communication statistic that is closely linked to interference is the number or frequency of transmission back-offs.

Many MAC protocols will check the channel is clear before transmitting. If a channel is detected as clear, the transmission will continue, otherwise the transmission will be postponed. Because of this, if the interference levels change, then the rate of back-off will change proportionally.

Monitoring the frequency of back-offs by the MAC protocol is a lightweight approach to inferring the channel conditions. Figure 4.3(a) illustrates an example of this, such that as interference becomes more dominant, the number of successful attempts where the channel was measured as clear over a fixed time period decreases.

4.2.4 Implementation

Each of the triggers defined above can be used to infer changes in interference. However, for a particular instance of runtime assurance, it is better to choose the trigger that links best to the requirements of the application. If an application has stringent communication requirements, a trigger based on communication statistics is more suitable.

In addition to the examination of the individual triggers, we choose to examine triggering on energy changes as part of a complete implementation of runtime assurance. As part of the car parking scenario which was discussed in Section 2.4 a detection module that utilises an energy trigger has been implemented. This implementation will be examined in detail and evaluated as part of the 2nd integrated demo with results presented in D-4.4.

4.3 Violation Verification

Whilst the aim of the detection system is to use lightweight mechanisms to detect possible violations of the environmental model, the verification system is not resource constrained, and instead must offer high accuracy in violation verification. To accurately verify a violation has occurred, data needs to be collected to generate the environmental model, and compare it to the model derived prior to deployment. If a violation has occurred this must then be flagged with the central system, which is examined in more detail in the following section.

For interference there are two environmental models; the idle/busy *CDF* which aims to capture the shape of interference and a simpler busy percentage model. In this section we will examine how each model can be instantiated during run time and how they can be compared with earlier derived models.

4.3.1 Model Instantiation

The same process utilised during pre-deployment should be used to gather the required data to generate the new model.

For interference, this process involved high frequency sampling of *RSSI* taken by the radio. Each sample was compared with a defined threshold to evaluate if the channel was busy or idle.

For the *CDF* model, the consecutive number of samples in the same state is important as this represents the period length. Each of these period lengths is recorded in a frequency table which is used to produce the corresponding *CDF* model. This is opposed to the busy channel state model, where instead of measuring the number of consecutive samples in the same state, a simple count of samples above the threshold with the total number of samples is needed.

The implementation of this mechanism has been described in detail in Deliverable D-1.2 [5]. It requires exclusive use of the main processor of the system during execution, as any reduction in sampling frequency or period will produce errors, and as such, interrupts must be disabled to reduce such errors. It is important that the application takes this into account when scheduling the task. Whilst high resources are required, runtime assurance should provide bounds to how long the process will take to aid in scheduling decisions.

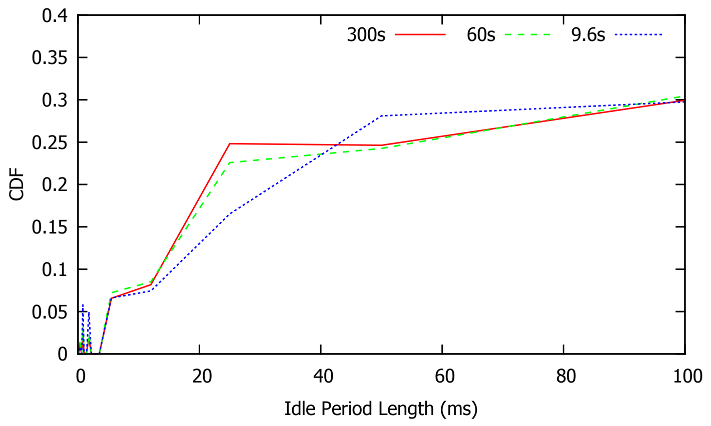


Figure 4.4: Comparison of *IDLE-CDF* generated with different sampling durations.

The available runtime for sampling will be application dependant, and for some application a short verification process will be essential. For interference, Deliverable D-1.2 [5] recommended a sampling duration of five minutes during pre-deployment data collection as this was shown to best capture the interference within the channel.

Our earlier work also recommends that the model be captured multiple times to increase its accuracy. During run time, running the model for long durations is not feasible, and instead, shorter sampling durations are needed to reduce any outages of the application, however a shorter sampling duration may lead to lower accuracy.

Figure 4.4 shows an *IDLE-CDF* captured for a point in time with different sampling durations (300, 60, and 9.6 seconds). As it can be seen the shape of the distribution for 300 seconds is similar to 60 seconds, however for 9.6 seconds a large deviation is seen. This suggests that sampling for 60 seconds may offer a similar representation of interference as provided by 300 seconds whereas 9.6 seconds may be too short. Whilst a sampling duration similar to that used for during pre-deployment is recommended, this shows that if necessary a shorter sampling duration may be acceptable.

4.3.2 Model Comparison

The model derived from information gathered at runtime must be compared to that generated pre-deployment to evaluate if a violation has occurred. This operation is highly dependant on the model in question, and can vary greatly in complexity.

For a model that represents interference as a percentage that the channel is busy, comparing the two models is simple. The earlier model will be stored on the device at compile time with an acceptable error threshold. The new model can thus be checked against this range and if within the model has not been violated, otherwise where the new model falls outside of the range there has been a violation.

$$D_{KL}(P||Q) = \sum_i P(i) ln \frac{P(i)}{Q(i)} \tag{4.1}$$

The more complex idle/busy *CDF* model presents a more complex operation, as comparing two *CDFs* is a more involved operation. There are a number of statistical tools that can be used to compare two probability distributions - however, the comparison must be made on the node with scarce processing resources.

One such tool which requires limited processing is the Kullback-Leibler (*KL*) divergence [9] presented in Equation 4.1 which measures the information lost when approximating one probability distribution with another. As with the less complex model above, the *CDF* model derived at deployment time with an acceptable KL divergence can be stored on the device at compile time, then the *KL* divergence between the *CDF* recorded at deployment time and the one taken at runtime can then be computed by the node. The calculated value can then be compared with the acceptable divergence range, and if the model is within this range no violation is considered to have occurred, otherwise a violation must have happened.

An example of using the *KL* divergence can be seen in Figure 4.5 which presents three different *IDLE-CDFs* recorded by the device over the course of a day. If the *CDF* recorded at 02:00 was used as the basis of the model, the divergence from this by the *IDLE-CDFs* recorded at 10:00 and 18:00 could be calculated. For the 10:00 the *KL* divergence was 0.00114 whereas for 18:00 it was higher at 0.029, thus; depending on the acceptable deviation, the model derived at 10:00 could be within acceptable tolerances, whereas that taken at 18:00 may result in a violation.

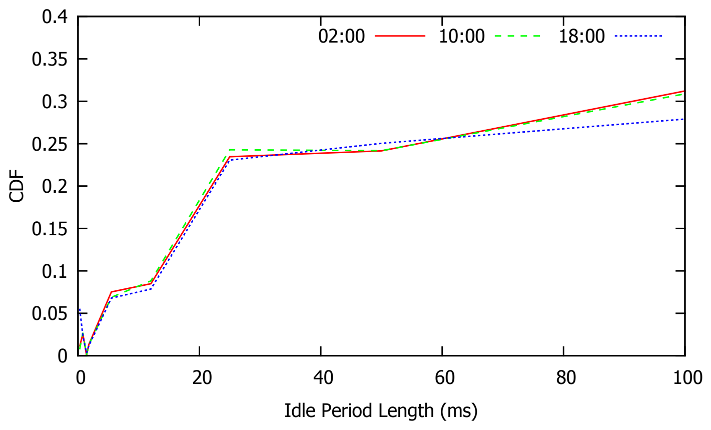


Figure 4.5: Three *IDLE-CDF* recorded at different times of the day

4.3.3 Implementation

In addition to our investigations into individual sub-components such as model instantiation and model comparison a complete instance of run-time assurance has been implemented as part of the 2nd integrated demo. As discussed in Section 2.4 the demo which focus on device life-time utilises the radio energy protocol model presented in [12]. This protocol model make use of the simpler busy percentage model for radio interference. For runtime assurance, our implementation of the verification module records high frequency samples of RSSI to compute the interference model which is compared with the one stored to verify model violations. This implementation will be examined in detail and evaluated as part of the 2nd integrated demo with results presented in D-4.

4.4 Reporting and Remediation

When a violation is detected it must be reported to the central system such that remediative actions can take place. Unlike detection and verification, the reporting module will essentially be the same regardless of the environmental model.

There are two available options; the report can be sent as an explicit message, or tacked on to existing communications. In high throughput application where packets are frequently generated, runtime assurance alarms can be aggregated with application data. An additional field added to the packet header to contain an alarm flag in this case is sufficent.

When a violation is detected, the system would set the flag to cause every packet transmitted to carry the alarm flag. This offers the least overhead as the alarm is sent with application data. However, additional information such as the derived model could not be incorporated.

The second option is to create an explicit message that is sent to the central system to report the violation. With this method, space is avaliable such that data captured during verification can be included in message, as this information may aid in remediation techniques. The controller can then choose to either try to compensate for the violation by adapting runtime parameters using the Runtime Adaptation of Protocol Parameters module or if the change in environment is too significant for runtime adaptation, to re-compute protocol selection and their parametrization and to deploy new configurations.

4.4.1 Implementation

As part of the complete implementation of runtime assurance for the 2nd integrated demo, we implemented reporting as an explicit message whose performance will be examined in more detail. When a violation of the interference model is violated, an explicit message is created that is sent to the control system. Within this alarm message, the current interference model instance is also included for remediation. This implementation will be examined in detail and evaluated as part of the 2nd integrated demo with results presented in D-4.4.

5 Conclusions

In Deliverable D-1.1 we developed environmental and platform models which can be used to estimate the performance of sensor net deployments with the main focus oin the environmental aspects of temperature and radio interference. This work was expanded in Deliverable D-1.2 where we presented tools that collected the relevant data before deployment to parametrise the developed models.

Since the publication of these deliverables we have continued this work and have focused on developing techniques for runtime assurance to check during normal application operation if the models we instantiated pre-deployment still hold.

In this document, we have presented a generic runtime assurance framework which consists of three elements; detection, verification and reporting and remediation. The detection subcomponent aims to provide lightweight mechanisms to infer model violations from information already present in the system. The verification subcomponent accurately verifies the violation detection by reusing tools developed in Deliverable D-1.2, and finally the reporting subcomponent, which reports the model violation to the central system. We have further shown how this framework can be used for the two environmental aspects which we have focused on in previous deliverables, temperature and radio interference. We examined Runtime Assurance for each of these aspects with focus on triggers for lightweight detection and tools to compare model instances.

Moving forward, we plan to use the work explored here, and in the previous Deliverables D-1.1 and D1.2 for the integrated demonstration, in which we will examine the connections between the work in WP1 with WP2 and WP3 in a real-world evaluation.

Bibliography

[1] C. A. Boano, M. A. Zúñiga, K. Römer, and T. Voigt, "JAG: Reliable and predictable wireless agreement under external radio interference," in *Proceedings of the* 33rd *IEEE International Real-Time Systems Symposium (RTSS).* IEEE, Dec. 2012, pp. 315–326.

[2] C. A. Boano, H. Wennerström, M. A. Zúñiga, J. Brown, C. Keppitiyagama, F. J. Oppermann, U. Roedig, L.-Å. Nordén, T. Voigt, and K. Römer, "Hot Packets: A systematic evaluation of the effect of temperature on low power wireless transceivers," in *Proceedings of the* 5th *Extreme Conference on Communication (ExtremeCom).* ACM, Aug. 2013, pp. 7–12.

[3] C. A. Boano, K. Römer, and N. Tsiftes, "Mitigating the adverse effects of temperature on low-power wireless protocols," in *Proceedings of the* 11th *International Conference on Mobile Ad hoc and Sensor Systems (MASS).* IEEE, Oct. 2014.

[4] C. A. Boano, M. A. Zúñiga, J. Brown, U. Roedig, C. Keppitiyagama, and K. Römer, "TempLab: A testbed infrastructure to study the impact of temperature on wireless sensor networks," in *Proceedings of the* 13th *International Conference on Information Processing in Sensor Networks (IPSN).* IEEE, Apr. 2014, pp. 95–106.

[5] J. Brown, I. E. Bagci, U. Roedig, M. A. Zúñiga, C. A. Boano, N. Tsiftes, K. Römer, T. Voigt, and K. Langendoen, "D-1.2 - Report on Learning Models Parameters," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Grant Agreement no: 317826, Tech. Rep., Nov. 2013.

[6] J. Brown, U. Roedig, C. A. Boano, and K. Römer, "Estimating packet reception rate in noisy environments," in *Proceedings of the* 9th *International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp).* IEEE, Sep. 2014, pp. 583–591.

[7] Crossbow Inc., *TelosB datasheet, Revision B*, 2004.

[8] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling ultra-low power wireless research," in *Proceedings of the* 4th *International Symposium on Information Processing in Sensor Networks (IPSN)*, Apr. 2005, pp. 364–369.

[9] Wikipedia, "Kullback-Leibler divergence," http://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence, 2014, [Online; accessed 24-Nov-2014].

[10] M. A. Zúñiga, C. A. Boano, J. Brown, C. Keppitiyagama, F. J. Oppermann, P. Alcock, N. Tsiftes, U. Roedig, K. Römer, T. Voigt, and K. Langendoen, "D-1.1 - Report on Environmental and Platform Models," http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Grant Agreement no: 317826, Tech. Rep., Jun. 2013.

[11] M. A. Zúñiga, F. Aslam, I. Protonotoarios, K. Langendoen, C. A. Boano, K. Römer, J. Brown, U. Roedig, N. Tsiftes, and T. Voigt, “D-2.1 - Report on Optimized and Newly Designed Protocols,” http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Grant Agreement no: 317826, Tech. Rep., May 2014.

[12] M. A. Zúñiga, I. Protonotoarios, S. Li, K. Langendoen, C. A. Boano, F. J. Oppermann, K. Römer, J. Brown, U. Roedig, L. Mottola, and T. Voigt, “D-2.2 & D-2.3 - Report on Protocol Models & Validation and Verification,” http://www.relyonit.eu/, RELYonIT: Research by Experimentation for Dependability on the Internet of Things, Grant Agreement no: 317826, Tech. Rep., Nov. 2014.