Addressing Pre- and Post-Deployment Support of Wireless Sensor Networks

by

Veselin Ganev

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

In this work we address the challenges arising when developing, testing and deploying software for Wireless Sensor Networks. We investigate both pre-deployment software design, as well as efficient post-deployment updates. We present a combined pre-deployment framework that simulates the network as it would be deployed to help in the testing, evaluation and fine-tuning of the system. Additionally, we address the need for post-deployment remote updates of the running code in an efficient and reliable manner. We also describe a real-world application that has motivated our work: the Smart Condo™ project, a system for monitoring patient activities in assisted living environments.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

- **FSM**: Finite State Machine
- **ISM band**: Industrial, Scientific and Medical band
- **MQTT**: MQ Telemetry Transport [1]
- **PIR**: Passive InfraRed
- **RFID**: Radio-Frequency Identification
- **VUEE**: Virtual Underlay Execution Engine [8]
- **WSN**: Wireless Sensor Network

Chapter 1

Introduction

1.1 WSN overview

Wireless Sensor Networks (WSNs) consist of a number of sensor nodes, whose purpose is to monitor physical phenomena and communicate their readings wirelessly to a base station for data storage and analysis. A typical sensor node consists of a microcontroller, a radio module, one or more sensors and a battery. The sensors translate physical properties into electrical signals, that can be interpreted by the microcontroller. The microcontroller reads the signals from the sensors, applies any required filtering and transmits the readings using the radio module. Typically, one node is designated to be the *sink*, which collects the information from the other nodes and relays it to the storage and visualization systems.

Due to limited sensing and radio range, WSNs often require a large number of nodes in order to completely cover the target area. This requires the cost of individual nodes to be as low as possible in order for the whole network to be affordable. In addition, the nodes being spread over a large geographic area requires extra care to be taken when designing the communication protocols used for transmitting the data to the base station. Depending on the area that needs to be covered, a WSN may be either single-hop, where every node can transmit to and receive from any other node, or multi-hop, where packets get forwarded by one or more routers before they reach their destination.

Another challenge arises from the fact that the nodes frequently need to be deployed in areas that are either not accessible or are difficult and expensive to access. This often means that replacing the batteries must be done as infrequently as possible, or in some cases it may not be practical at all and therefore, the lifetime of the network would be equal to the lifetime of the batteries powering the nodes. This requires the nodes to be as energy-efficient as possible in order to maximize the life of their batteries. Further, it must be possible for any modifications to the nodes' configuration and software to be done remotely so that having to physically visit each node can be avoided.

The energy usage requirements for WSNs pose design challenges for both the hardware and software design of the sensor nodes. The radio transmissions are typically the biggest contributor to the energy usage of the nodes, which requires that transmissions are minimized as much as possible.

Designing algorithms and communication protocols that reduce the amount and the size of messages being transmitted is one of the most active areas of research in the WSN field. It is often assumed that receiving data over the radio is free, however in reality keeping the radio listening, even when there is nothing to receive, presents a significant energy cost and therefore, it is often necessary to keep the receiver off as much as possible. Reducing energy consumption also requires that the processor running the node's software is kept in a low-power sleep mode for the majority of the time.

Designing software for wireless sensor nodes presents a variety of challenges. The main difficulty, as mentioned above, is minimizing the energy consumed by the node, while still being able to relay the collected data in a timely and reliable manner. A related problem is that the combination of needs for low energy consumption and low cost nodes translates in using hardware that is very limited in terms of processing power and available memory, which further increases the difficulty of developing software for the sensor nodes. And finally, software for WSNs shares many of the challenges found in traditional distributed systems since each sensor node is an independent computing device that needs to work together with the other nodes in order to efficiently accomplish the goal of sensing the properties of interest.

1.2 The need for a simulation at the appropriate level

Once the network is deployed it is often costly to perform major changes to the application. One way to reduce the need for such changes is to use a simulation-based approach when developing and testing of the software of the sensor nodes. The main components that need to be modelled include the sensing characteristics of the nodes, the network environment used for collection of the data and the energy cost of the sensing and networking-related operations (for battery powered nodes) as well as the platform on which the software runs.

An important factor when deciding how to design the simulation environment is the desired fidelity of the simulation. If the models are too simplistic the simulation may not be representative enough to provide useful pre-deployment evaluation of the application. If, on the other hand, the models are too detailed, then, not only does it become difficult to develop and maintain the simulation system itself but also, there will be too many parameters that would all need to be figured out separately for each application, making it too cumbersome to set up the simulation environment for a particular application. Another problem with simulations that are too detailed is that they may be computationally expensive, which limits the usability of the simulation.

On the networking side, a large amount of simulation-based wireless networking research simplifies the environment too much by making very strong assumptions about the way signals propagate [14]. This leads to situations where the behaviour of the networking protocols can be so different from the behaviour in reality that any evaluation of the networking components of the application may not be representative enough to be useful. For example, many simplified models assume a

propagation pattern represented by a circle of a given radius, with the transmitting node in the middle of the circle. The signal in this model is always of the same strength within the circle and it immediately disappears outside of the circle. Such models do not take into account signal fading, the fact that the radio channel quality varies with time, obstacles between the nodes, possible asymmetries in the channel. Using such a simplistic model for pre-deployment simulation risks not providing enough of an opportunity to test the application's capability to handle the challenges of the networking environment.

From the perspective of a pre-deployment simulation, it is beneficial to be able to test how well the software processes readings in the particular environment being sensed, given the capabilities and limitations of the particular sensors being utilized. For this to be possible, a model of the environment and sensors needs to be provided to the simulation. Many sensor networking simulation techniques do not concern themselves with the sensing characteristics and often assume that there will always be readings following a particular distribution. On the other hand, an overly detailed simulation of the environment and sensors, such as those used in radar systems, can be difficult to set up and use.

Energy consumption, which is a critical factor in battery-powered applications, is also an important part of the simulation environment, so that the the energy profile of the application can be evaluated and optimized pre-deployment. Many models for wireless sensor network energy consumption assume that keeping the receiver always enabled is free and only transmissions cost energy. However, in a typical radio, the receiver can draw a significant amount of current (e.g. up to 10-15 mA for the nodes used in the Smart Condo), depending on the receiver. This means that for WSN applications that require long period of operation on a single battery charge, some sort of duty cycling of the receiver will be unavoidable and this must be testable in the simulation as coordination in the presence of such duty cycling can be a significant challenge faced by the WSN application. Similarly, many models assume that the energy cost of the sensors is negligible, however some sensors can consume a substantial amount of energy and this needs to be taken into account by the simulation environment.

Another component of the pre-deployment simulation system is the platform that runs the software. The sensor nodes typically feature low-powered embedded processors. Some simplistic simulators do not allow the actual application code to be simulated but instead require a high-level description of the algorithms running on the node and will perform the simulation based on that. This may be adequate for evaluating the algorithms and parameters but would not provide an appropriate pre-deployment test of the actual code of the applications, which is an important part of minimizing the need for modification post-deployment. On the other extreme, emulators can provide a virtual version of the entire system and run the same machine code that is being run on the nodes. The challenges with this approach are that it requires a considerable effort to replicate each sensor node platform and also it often does not allow many of the traditional debugging and testing tools to

be used, despite the fact that the emulation is running in a standard development environment.

In between the two extremes is the source-level simulation approach, which allows for the application code to remain the same between the real application running on the node and the simulated application. This has the benefit of allowing the traditional debugging and testing tools to be used while developing the application running on the nodes.

1.3 The need for software updates

As mentioned in Section 1.1, WSNs can be deployed in areas that are inaccessible. As a result, often the only way to interact with them is through the wireless network itself. And even if they are accessible, the network can contain a very large number of nodes that makes it impractical to manually visit every one of them. For these reasons, currently programming the nodes is usually done only during the development stage and it is assumed that the code they are running will not change during their lifetime. However, requirements can change and bugs can be discovered after deployment which introduces a need to be able to efficiently and safely update the program running on the node over the network. This is made particularly difficult by the fact that wireless transmissions are very expensive which means that extra care needs to be taken to reduce the size of patch that gets sent over the network. Flash writing is also relatively expensive, so it should be minimized as well. Another issue is that if an update has unexpected effects that cause nodes to malfunction and if there is no fail-safe mechanism to automatically bring the nodes back to a state where they can at least receive further updates then the nodes could become completely useless.

1.4 Scope of the Thesis

The purpose of the thesis is to address the challenges arising when developing, testing and deploying software for WSNs. To this end we recognize two separate steps and corresponding challenges. The first is the pre-deployment software design, and the second is efficient post-deployment updates. While still constrained by energy usage, we attempt to, on one hand, simulate the network as it would be deployed to help in the configuration and fine-tuning of the code and (pre-deployment). On the other hand, we attempt to address the need for efficient and reliable remote updates of the running code (post-deployment). The rest of the thesis reflects these two aspects of WSN software development. However, we first present the environment and applications that motivated this work, namely, the Smart Condo project.

1.5 Previous and Related Work

ns-2 [4] [13] and its successor *ns-3* [2] are general discrete-event network simulators, simulating communication at the packet level. They are commonly used for evaluating network protocols. *ns-2* does not attempt to model the application layer. *ns-3* supports Direct Code Execution (DCE)

[23], which allows real applications to be executed in the simulation environment. DCE can run Linux applications and the Linux kernel in its simulated environment, allowing existing protocol implementations in the Linux kernel to be simulated in *ns-3*.

SensorSim [19] is based on *ns-2* and adds features relevant to simulating sensor networks. It supports modelling of power usage and energy storage and new communication protocols. Further, it allows interaction between the simulated environment and real nodes in order to address difficulties in modelling sensors. SensorSim uses a middleware platform called SensorWare, in which applications are written using the Tcl scripting language, allowing them to run both on the real and on the simulated nodes at the expense of requiring nodes to use higher-power hardware than the microcontrollers that we target.

EmStar [10] is a simulation environment for nodes running Linux on iPAQ-based hardware. Similarly to the hardware targeted by SensorSim, this is significantly higher-power than the type we work with. It allows the same application code to run on both the simulated and real nodes, as they both run on top of a Linux system and the simulated environment provides simulated drivers for the radio and sensors that behave the same as the real drivers.

TOSSIM [17] simulates applications that use the TinyOS [15] operating system and it utilizes the source-level simulation approach, allowing the same application code to run on both the real and simulated nodes. It provides bit-level simulation of the TinyOS communication stack and models for the node's hardware, such as the Analog-to-Digital Converter, the clock, the serial interface (UART), etc. TOSSIM simulates the hardware interrupts that drive the execution of a TinyOS application and uses an event queue that delivers the interrupt events to the application. TinyOS applications use the nesC [12] language, an extension of the C, which provides support for the TinyOS concurrency model. It does not allow for dynamic memory allocation. TOSSIM modifies the nesC compiler to allow compilation of TinyOS applications into the simulation framework.

TOSSF [20] is another simulator for TinyOS applications. It builds upon the Dartmouth Scalable Simulation Framework (DaSSF), a framework for discrete-event simulation and the Simulator for Wireless Ad-Hoc Network (SWAN), which provides models for the RF channel, sensors, the terrain and node mobility.

COOJA [18] is a simulation framework for applications using the Contiki [9] WSN operating system. COOJA provides cross-level simulation by supporting simulations at three different levels: the networking level, the operating system level and the machine code instruction set level. The networking level is the highest level of abstraction, where the application is not simulated and can be used for initial design and prototyping if protocols. The operating system level simulates the application and OS, compiled for the host platform. At the lowest level, the machine code instruction set level, COOJA uses an instruction-level MSP430 emulator to work directly with code compiled for the node's target hardware. Each individual simulated node must be at one of these levels, however a simulation can have nodes at any of the three levels. COOJA also provides radio models

for the communication channel between nodes.

One way to update code running on sensor nodes while keeping the size of updates small and number of flash writes low is to use a virtual machine such as Maté [16]. In that case the program running on the node is specified in a higher level language and it is run inside another application (the virtual machine). This allows programs to be significantly smaller which makes the size of the updates smaller and it also makes it easier to update one part of the code without affecting others. Also the virtual machine can dynamically load the new version of the program, so some of the issues due to address shifts in single-image based applications are addressed with this approach. One of the problems with using a virtual machine is the performance penalty, which can be too much when the hardware is very limited which is typical for sensor nodes. Another problem is that if the virtual machine needs to change then some other mechanism needs to be used.

Wang et all [26] survey various proposed systems for wirelessly reprogramming sensor nodes, however most of them are only sending the entire new program even though only a small portion may have changed and some of them are TinyOS-specific and rely on the nodes using TinyOS. The methods proposed in [22] and [21] produce patch scripts based on two different images, attempting to minimize size of the patches. The approach taken in [22] does not consider the effect of address shifts as a result of code inserted in the middle of a program which is a significant source of differences between two versions of a program due to lack of dynamic relocation. While their reason for not doing that is a valid one (it is dependent on the instruction set), this can result in much bigger patches. The authors in [21] address that issue although they do not mention how the patch is distributed and how failures in a large network would be dealt with.

A feature that appears to be missing from the systems described in these papers is the ability to verify that the patch decoding procedure was successful and the ability to restore a node after a patching operation has failed. This is important because without such functionality a problematic update could render large portion of a WSN completely unusable.

1.6 Summary of Contributions

The contributions of the work described in this thesis has two major components:

- Virtualized environment for realistic pre-deployment simulation
- Efficient and reliable post-deployment code update system

The pre-deployment Wireless Sensor Network simulation framework provides an environment for simulating the three major components of a WSN deployment: the sensors, the network and the application. It can be useful for testing the implementation of the application logic as well as the performance of the WSN deployment using relevant metrics, such as energy consumption, accuracy of inferred information (e.g. localization accuracy), reporting latency and others. The framework provides source-level simulation of the application, allowing the same application code

that runs on the real nodes to run in the simulation environment. It provides sensor models as well as models of the radio communication channels. The radio model used in the framework is significantly more realistic than the simplistic circular models described in Section 1.2 and at the same time it avoids burdening the user with the need to tune too many parameters and to collect too many measurements. These characteristics of the simulation framework provide enough fidelity for testing and evaluating the application behaviour in a realistic manner, addressing the problem of simulation at the appropriate level, described in Section 1.2. Chapter 3 describes the simulation framework in detail.

The post-deployment update system addresses the challenges of creating and distributing code updates in Wireless Sensor Networks. It applies a number of optimizations to reduce the size of the patch that is being transmitted in order to minimize the energy used while distributing the patches. A key optimization employed during the patch generation is disassembling the code and looking for address shifts. Since a single change can cause many similar shifts, the positions and sizes of the shifts are often consistent, allowing the generator to describe a large number of shifts using a short amount of information. The patch processing software running on the nodes also disassembles the old software and applies the address shifts described in the patch. Another important feature of the update system is ability to detect faults at various stages of an update. If needed, it can safely abort the update in a way that leaves the sensor network in a consistent and usable state. This is critical when the nodes in a deployment are not reachable post-deployment where a failed software update could render the network unusable. Chapter 4 describes the design and implementation of the system in detail, including the patch generation algorithm and the update verification mechanisms. Further, it provides an evaluation of the different optimizations employed by the patch generator.

Chapter 2

The Smart Condo

This thesis has been in-part motivated by the Smart Condo, a Wireless Sensor Network system for monitoring patient activities in assisted living environments and aiding hospital discharge teams in evaluating the patients' abilities to function on their own [27] [11] [24]. This application benefits of a wireless sensor network, as the project aims to collect information about how the patient is managing on their own, in a way that minimizes privacy intrusion, while providing data that is easy to process and analyze by software, both of which are difficult to do with video-surveillance-based solutions. The system was deployed in the Independent Living Suite at the Glenrose Rehabilitation Hospital in Edmonton, AB and at the Edmonton Clinic Health Academy at the University of Alberta.

The current version of the Smart Condo architecture is shown in Figure 2.1. The system consists of the following components:

- the **sensor network**, a collection the sensor nodes, monitoring the environment and sending the data wirelessly
- the **bridge**, receiving the data from the sensor network and publishing it on the relevant MQTT topics
- the **database storage component**, which receives readings published on the MQTT topics and stores them for future processing
- the **Localizer component**, which takes raw sensor readings and estimates the location of the patient by applying various localization algorithms
- the **report generation tool**, which analyzes the sensor data and produces reports for the medical staff
- the **Second Life client**, visualizing the inferred patient activities in a Virtual World, either in real-time or based on past traces
- the **diagnostic tools**, which get data from the MQTT stream and produce various diagnostic information about the state of the sensor network

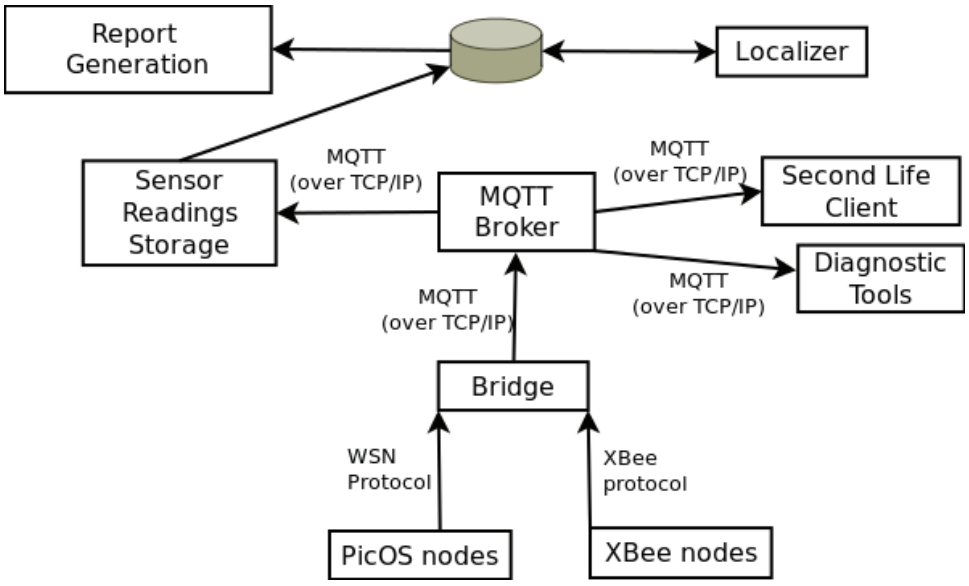

Figure 2.1: The Smart Condo Architecture

- the **simulator**, capable of emulating the sensor network, aiding in the development and evaluation of the system

2.1 Sensor Nodes

The objective of the Smart Condo is to allow for the easy integration of any sensor relevant to assisted living, off-the-shelf, or custom made. Towards this goal, the Smart Condo started with a combination of sensors and sensor technologies but its architecture facilitates the easy integration of new devices.

Currently, each sensor node has one or more of the following sensors:

- Passive InfraRed (PIR) motion sensors in the ceilings detecting patient movement around the suite (Panasonic NaPiOn AMN43121) [3]
- Reed switches on doors, cupboards and drawers detecting opening/closing
- Seat sensors on furniture detecting when and where a patient is sitting
- Bed sensor for monitoring the patients' sleeping patterns
- Electrical current sensors detecting usage of various electrical appliances and lights
- Water flow sensors detecting sinks/shower/toilet usage
- RFID reader
- accelerometers

2.2 Operational Aspects of the Sensor Nodes

Most of the sensor nodes are built around the PicOS operating system for embedded devices detailed in section 3.1. They consist of a TI MSP430 [6]-based microcontroller, a CC1100 [5]-based radio transceiver, operating on the 915 MHz ISM band and on-board flash storage. All of these components are carefully chosen with low power consumption in mind.

At the lowest level a sensor can produce either digital or analog outputs. However, for the purposes of analyzing patient behaviour, all sensors are treated as ultimately digital, indicating whether a certain activity is taking place at any point in time. In the case of analog sensors, the node does some local processing (for example, applying a threshold) of the raw values in order to determine the boolean state.

Digital sensors are configured to trigger interrupts, which can wake up the microcontroller when a change in the state has occurred. This allows the CPU to be in low-power sleep mode, and to be automatically awoken when the state of the sensor is changed. For nodes that have analog sensors, it is usually necessary for the node to wake up periodically, take measurements and decide whether a state change has occurred. This consumes more power, especially when running the analog-to-digital converter is involved, however the node still saves energy by processing the raw values locally and only transmitting messages when actual an actual state change has been determined.

Each sensor reading that is reported has the following information associated with it: timestamp, state and, optionally, raw value. Ultimately, the node needs to transmit the readings, so that they can be processed by the rest of the system. However, the size of each reading is small compared to the overhead per packet and, in the interest of reducing energy usage, it is desirable to reduce the amount of overhead per reading. This is addressed by buffering readings and sending multiple readings per packet. When there is a new sensor reading, it is first stored in a local buffer. The transmission of the readings in the buffer can be triggered by one of two conditions: the buffer is full or a certain (configurable) amount of time has elapsed since the time of the first event in the buffer.

All readings are timestamped (and the clocks of the nodes are frequently synchronized), so this buffering mechanism does not affect the ability to determine when a particular event occurred, even if it is not transmitted right away. However, this does introduce some latency in the reporting of the sensor readings, which might have some detrimental effects, depending on the situation. If the collected data is only used to produce a report at the end of the patient stay, the delays in reporting caused by this buffering are insignificant. In situations where real-time visualization is required, low latency is needed, meaning that we cannot buffer for too long. In order to address different requirements, there is a configurable maximum buffering time, which determines the maximum time between a sensor reading and the time this reading is transmitted.

In the simulation environment it is possible to experiment with different buffering times and different patient traces. The amount of energy saved by this feature depends on how often the same sensors get triggered in a relatively short period of time, which, in turn, is heavily dependant

on the specific patient activity patterns. Thus, while it is possible to use simulated patient traces for experimenting with this feature, it is more useful to use real historical patient activity and run simulations with different buffering times, which can show the trade-off between energy usage and latency.

When a node first starts up, it sends a *POWERON* message, which notifies the bridge that the node wishes to join the network. Then, the bridge sends an *ANNOUNCE* message indicating that the node is recognized and is allowed to join. After that, the node sends a configuration request and the bridge responds with a message containing the current values of the configuration options for this node. At this point the node is considered to be associated with the bridge and it can send sensor readings and receive further configuration updates if necessary. If no responses are received by the node after several *POWERON* messages have been sent, the bridge is considered to be currently unreachable. In that case, the node will go to sleep and will repeat the association attempt at a later time.

In PicOS, acknowledgements and re-transmissions are left up to the application. Therefore, the Smart Condo application also provides this functionality. Whenever a packet is transmitted, the node will wait for an acknowledgement from the bridge and will re-transmit it if no acknowledgement has been received within the allowed time (which is configurable). After a certain (also configurable) amount of failed transmission attempts, the node considers the bridge to be unreachable. In such an event the node will consider itself dis-associated from the bridge and it will go through the standard association process of sending *POWERON* messages, waiting for replies, and so on.

In order to keep energy usage low, a node keeps its transceiver off for the majority of the time and only enables it when needed. This applies not only to the transmitter, but also to the receiver, which consumes a considerable amount of power when enabled. In a standard node, the receiver is only enabled when waiting for acknowledgements from the bridge. This reduces energy usage, however it presents a problem when a node's configuration has been updated. In order to address this, any configuration changes are added to the acknowledgements sent by the bridge. This allows for such changes to be received when the node has sensor readings to transmit, however, for many nodes, there can be a considerable delay between new sensor readings. This is resolved by introducing a configuration parameter *max_tx* and forcing the node to send a message at least once every *max_tx* seconds. If the node does not have any readings, it will send an empty message in order to satisfy this requirement. The acknowledgement of this message will contain any configuration changes, meaning that in the worst case scenario, a node can still be reconfigured at least once every *max_tx* seconds. In addition, this functionality ensures that a node failure can be detected within *max_tx* seconds, because under normal circumstances, a node will send at least one message every *max_tx* seconds and thus, lack of messages from a node in that amount of time can be used as an indicator that there might be something wrong with the node.

In some deployments of the Smart Condo, the sensors are used for some patients but not for

others. This results in the nodes not needing to report anything for extended periods of time. In these cases, further energy usage optimization is provided by a special ”deep sleep” mode. In this mode the nodes turn off their sensors and only periodically communicate with the bridge to find out if they should get out of deep sleep mode at this point in time. The frequency of such communication is configurable and is the determining factor for the overall energy consumption in this mode. The lower the frequency, the less energy will be used during the deep sleep period. However, setting it too low means that there will be a longer delay between the time it is decided to take the nodes out of deep sleep mode and the time all nodes have actually performed this action.

While the majority of the sensor nodes in the Smart Condo are of the PicOS-based type, the system also supports several several off-the-shelf devices. These include some of the electrical current sensors, which utilize XBee transceivers with custom application-layer protocols, as well as several proprietary medication monitoring devices that require obtaining data from third-party back-ends. An important aspect of the Smart Condo architecture is that the differences between these node types are abstracted in such a way that the rest of the system's components do not need to know the details pertaining to different node types and the different communication mechanisms involved. This is, in part, achieved by implementing an unified communication mechanism, as described in the next section.

2.3 Unified communication using MQTT

As described earlier, the Smart Condo system consists of a number of distributed components, in addition to the heterogeneous sensor network. In order to make it easier to develop, test and maintain the system, all communication between components is done using MQTT, a lightweight publish/-subscribe messaging transport protocol [1]. In an MQTT-based system, each client connects to the MQTT broker, subscribes to any topics of interest and publishes messages on other topics. Whenever a message is published on a topic, any client that is subscribed to this topic will receive the message.

MQTT provides three Quality of Service (QoS) levels [1]:

- QoS 0 (At most once delivery): No acknowledgements or packet identifiers (sequence numbers) are employed by the MQTT protocol. Messages may be lost, without the sender's knowledge and messages may be duplicated.

- QoS 1 (At least once delivery): Each message has a packet identifier and must be acknowledged by the receiver before the sender considers the message to be actually sent. Duplication is still possible.

- QoS 2 (Exactly once delivery): In addition to ensuring that the message has been received, this QoS level also makes sure that it is received only once.

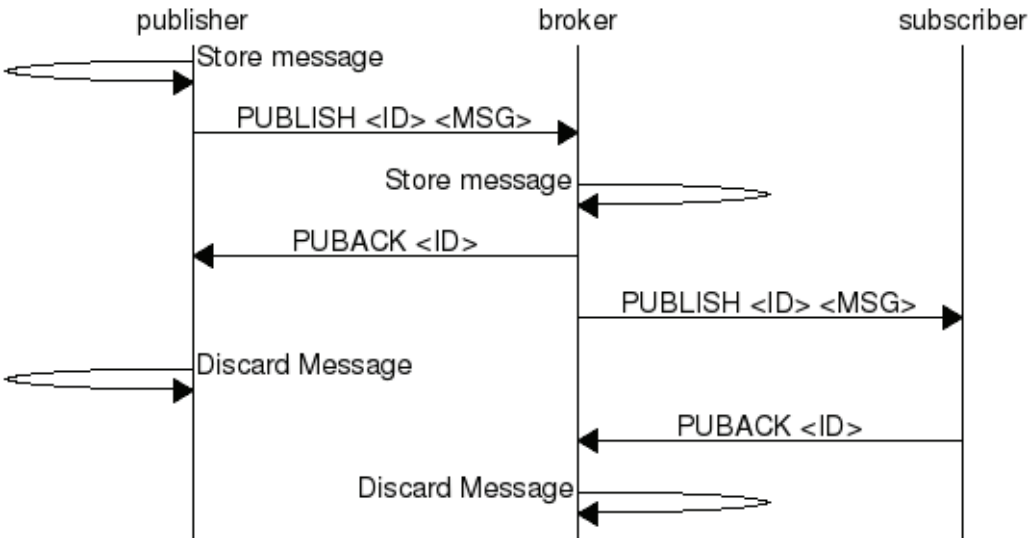

Figure 2.2: Sequence diagram showing the packets sent during the publishing of a MQTT message with QoS 1 [1]

Both the publisher and the subscriber can choose their own QoS when negotiating with the broker. ”Sender” and ”receiver” in the above description can refer to different entities at the different stages of message transmission, i.e. when a client first publishes a message, the publisher is the sender and the broker is the receiver and after that the broker is the sender and the subscriber is the receiver. In both halves the QoS logic is applied based on the QoS level requested by the publisher and the subscriber. A sequence diagram for the publishing of a message with both the sender and receiver using QoS 1 is shown in Figure 2.2.

In the Smart Condo system, the bridge receives the sensor data from each node and publishes it on the topic designated for the particular sensor. Other components subscribe to the topics for the sensors that are relevant to the particular component and process sensor data arriving on these topics. Even though the system always uses MQTT over TCP, it still uses QoS 1. The reason for this is that while TCP ensures that the data (MQTT messages in this case) will be delivered reliably while the connection is still valid, it is still possible for messages to get lost when the connection gets unexpectedly closed (between the time the connection dies and the time the MQTT client learns about the fact) and the bridge must ensure that all readings are transmitted. In addition to brief disconnects, it is also possible that the connectivity between the bridge and the MQTT broker is disrupted for longer period of time. In order to address this, the bridge stores readings locally until they can be published to the broker.

2.4 Localizer

One key functionality of the Smart Condo system is the estimating the location of the patient. The location information is used by the higher-level tools, including the report generation and the Sec-

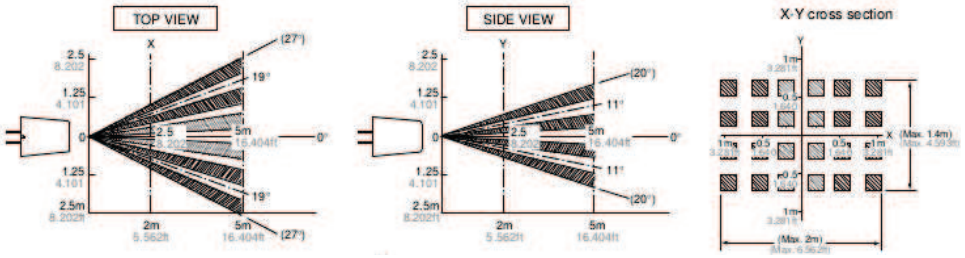

Figure 2.3: PIR motion sensor area for the ”Spot” type NaPiOn sensors [3]

ond Life visualization. The location error (as determined by the difference between the inferred location and the ground truth) is one of the metrics used to evaluate the performance of a particular deployment of the system.

For localization purposes, there are two types of patient activities: (1) those that require the patient to be at a particular position for the activity to occur and (2) motion around the suite. Activities of type (1) include opening/closing doors/cabinets/drawers/etc., using the sinks/shower, flushing the toilet, starting appliances (but not necessarily stopping them, as some may be only run on a timer). When such an event occurs, and assuming that the patient is alone in the suite, the localization process is straightforward, as the the location of the object being manipulated is known. However, often the localization needs to be done based on patient motion only, using readings from PIR motion sensors and RFID readings (if the RFID subsystem is included in the particular deployment). The rest of this section described several methods developed for localization based on motion sensor readings.

The first version of the Smart Condo used a simple centroids intersection method to perform the localization [11]. Given a particular placement of the sensors, the possible centroids of the overlapping sensor coverage areas are pre-computed. Then, as motion sensor readings are processed, the location of the patient is inferred to be the overlap centroid corresponding to the set of currently triggered sensors. In order to address possible faulty readings, single-sensor readings are filtered out, meaning that a location estimate is provided only when at least two sensors are currently being triggered. This method had the advantage of being simple and easy to implement.

The current version of the system uses a more sophisticated localization algorithm [28], which provides a number of improvements over the simple centroids intersection method. It supports arbitrary coverage area and allows the motion sensors used in the Smart Condo, which have a pyramid-shaped sensing area (as shown in Figure 2.3 from [3]), to be modelled more accurately. In addition, it takes into account walls and other obstacles that will affect the actual sensing area. The geometric model of the person being tracked is modelled is more realistic, being considered as a cylinder instead of a single point. It can also handle oscillations in the sensor outputs. The algorithm considers the short-term history of the estimated person locations, which allows the estimates to be refined as more readings become available. The design and implementation of this localization algorithm are

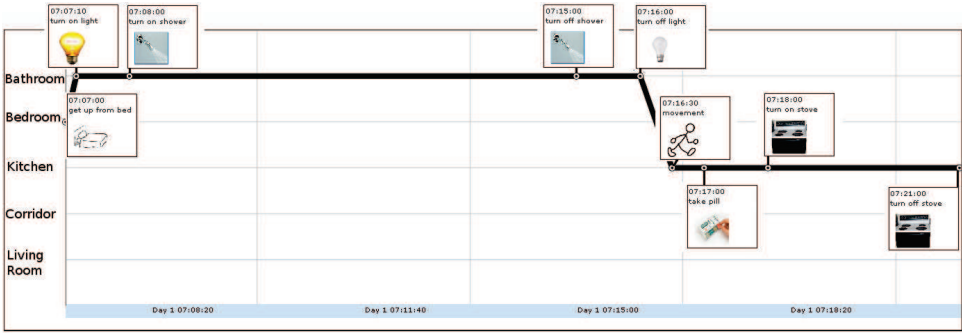

Figure 2.4: Graph of patient activities

described in detail in the thesis of Vosoughpour [28].

2.5 Reports and Visualizations

As described earlier, the purpose of the Smart Condo is to provide staff with information about the patient's stay. To this end, the system can produce both textual and graphical reports that summarize the patient activities of interest. It can also show the patient activities in a Virtual World environment.

The textual reports produce information about patient activities and highlight potential red flags, as requested by the occupational therapists. The following are some examples of such activities and related red flags that are included in the textual report, all associated with the corresponding timestamps marking when the activities are occurring:

- meal preparation activities: usage of stove and other cooking appliances; patient behaviour while the stove is on, e.g. do they stay mostly in the kitchen, do they leave the suite; kitchen sink usage (do they leave the sink on and leave the area)
- bathroom activities: usage of the toilet (is it used too often); sink usage or lack thereof after toilet usage; usage of the shower
- night-time activities: how often does the patient get up at night; how often do they use the bathroom at night; where else in the suite to they spend time at night
- medication activities: do they take their medication on time

The graphical reports provide a time-series chart of the patient activities, as shown in Figure 2.4. The X axis on the chart is time and the Y axis represents the area of the suite that the patient was in. Additionally, the line is annotated with events of interest that have been detected (usage of appliances, shower usage, taking medication, etc.). The graph provides the ability to adjust the time-window, allowing the user to get a broad overview of patient activities as well as to zoom in on a particular time interval and seeing more details about the activities during that time.

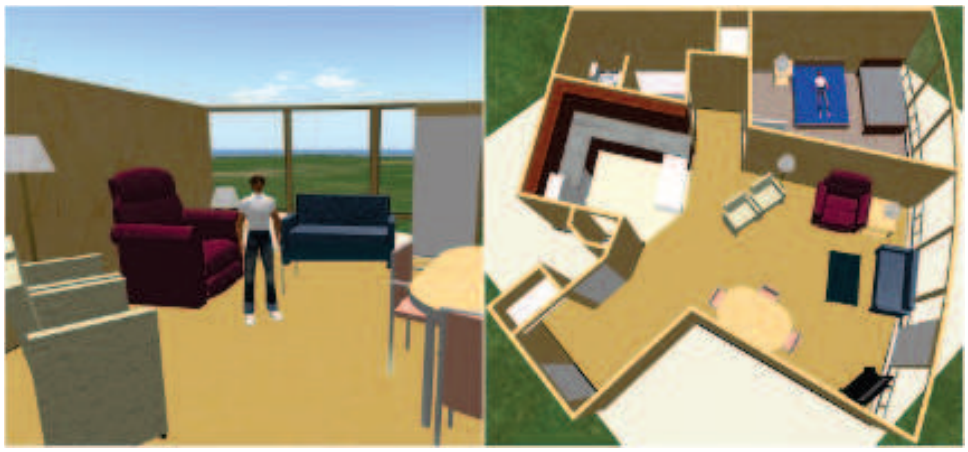

Figure 2.5: Virtual worlds visualization, showing the avatar and the suite from different views

The virtual world visualization uses a 3D model of the suite and provides animation of patient activities having an avatar move and perform the actions that have been inferred after processing the sensor readings [24] [11]. It aims to provide more detailed information about the patient activities at a particular time than the textual and chart reports, while still avoiding the privacy concerns of video surveillance. Figure 2.5 shows two example views of the suite and avatar in the virtual world visualization.

2.6 Contribution

The primary role of the Smart Condo project in this thesis is to provide motivation for the work on the pre-deployment simulation framework and post-deployment update system described in the next chapters. In addition, the following aspects of the Smart Condo have been developed while working on the thesis:

- The MQTT communication unification
- Support for water flow sensors, electrical current sensors, RFID readers and accelerometers
- Support for deep sleep mode
- The sensor readings storage component
- The diagnostic tools
- The report generation tools
- Support for third party sensors using the XBee protocol

Chapter 3

Pre-deployment Simulation

3.1 PicOS

Our approach to simulation at the appropriate level utilizes the PicOS [7] operating system in combination with VUEE [8] (Virtual Underlay Execution Engine) simulation framework.

PicOS is a lightweight operating system, specifically designed for applications such as those typically running on low-power sensor nodes. It provides a form of multitasking implementation for embedded event-based applications that is suitable for devices with very limited memory. Tasks in PicOS are designed to behave like finite state machines (FSM) that change their state in response to events. While a typical application has a number of running tasks, the switching between tasks happens only at state boundaries, meaning that the CPU switches from one task to another only when the current task FSM is transitioning to another state. This greatly simplifies the synchronization between applications, while still allowing concurrency and it also helps in maintaining clarity of the applications.

PicOS also includes a ”sensors” API, which allows the application logic that deals with reading data from sensors to be separated from the hardware-specific sensor drivers. The application can ask to be awoken in a particular state when the value of a sensor has changed and it can then read the sensor value and perform actions based on it. The driver for any particular sensor must implement the waiting and reading operations.

A sample PicOS application is presented below:

```
#include "sysio.h"
#include "tcvphys.h"
#include "phys_cc1100.h"
#include "plug_null.h"
#include "sensors.h"

#define MAX_PACKET_LENGTH       60

typedef struct {
        byte reserved;
        word sens_val;
        word src;
```

```
        word seq_no;
} msg;

int sfd = -1;
word sensor_val;

fsm receiver {
        address rx_pkt;

        entry RX_INIT: {
                tcv_control (sfd, PHYSOPT_RXON, NULL);
        }

        entry RX_GET: {
                rx_pkt = tcv_rnp (RX_GET, sfd);
        }

        entry RX_SHOW: {
                msg* rx_pkt_msg = (msg*)rx_pkt;
                diag("Got packet SRC=%u SEQ=%u SENS=%u RSSI=%d",
                        rx_pkt_msg->src, rx_pkt_msg->seq_no,
                        rx_pkt_msg->sens_val,
                        ((byte*)rx_pkt) [tcv_left (rx_pkt) - 1]);
                tcv_endp (rx_pkt);
                proceed(RX_GET);
        }
}

fsm sender {
        address tx_pkt;
        word tx_pkt_len;
        word seq = 0;

        entry TX_INIT: {
                tcv_control (sfd, PHYSOPT_TXON, NULL);
                tx_pkt_len = sizeof(msg);

                if (tx_pkt_len & 1)
                        tx_pkt_len++; //Round up to even
                //2 bytes for network ID and 2 bytes for CRC
                tx_pkt_len += 4;
                if (tx_pkt_len > MAX_PACKET_LENGTH) {
                        diag("TX packet too long");
                        finish;
                }
        }

        entry TX_WAIT: {
                when(sender, TX_SEND);
                release;
        }

        entry TX_SEND: {
                msg* tx_pkt_msg;
```

```
                diag("TX_SEND[SEQ=%u,SENS=%u]", seq, sensor_val);
                tx_pkt = tcv_wnp (TX_SEND, sfd, tx_pkt_len);
                tx_pkt_msg = (msg*) tx_pkt;
                tx_pkt_msg->src = (word) host_id;
                tx_pkt_msg->seq_no = seq++;
                tx_pkt_msg->sens_val = sensor_val;
                tcv_endp (tx_pkt);

                proceed(TX_WAIT);
        }
}

fsm root {
        entry R_INIT: {
                phys_cc1100 (0, MAX_PACKET_LENGTH);
                tcv_plug (0, &plug_null);
                sfd = tcv_open (WNONE, 0, 0);
                if (sfd < 0) {
                        diag ("Cannot open tcv interface");
                        halt ();
                }

                runfsm receiver;
                runfsm sender;
                proceed(R_NEW_EVENT);
        }

        entry R_WAIT: {
                wait_sensor(0, R_NEW_EVENT);
        }

        entry R_NEW_EVENT: {
                read_sensor(R_NEW_EVENT, 0, &sensor_val);
                trigger(sender);
                proceed(R_WAIT);
        }
}
```

This application performs the following simple tasks:

- Periodically sends a message to the rest of the nodes
- Upon receiving a message from any other node, shows information about it

Three FSMs are used to implement this application: *root*, *sender* and *receiver*.

The *root* FSM is present in all PicOS applications and is started by PicOS when the application starts. It may start additional FSMs as needed. In this application, the root FSM initializes the network interface and starts the sender and receiver FSMs. It also provides an example of using the PicOS sensor API to wait for sensor value changes and to read the value. The *wait_sensor* function tells the OS that when the sensor value changes, the application should be awoken in the requested

state (R_NEW_EVENT in this case). Whenever the value of sensor 0 changes, the FSM reads the new value and signals the sender FSM to send a new message.

The *sender* FSM does some initialization and then waits for signals from other FSMs, indicating that a new message needs to be transmitted. Note that in this case, the sender could just do the waiting for sensor changes itself, however the example aims to illustrate the simple interprocess communication functionality provided by PicOS. Each FSM can wait for a number of conditions to occur by using the *when* function and specifying the identifier of the event to wait for and the state in which it should be woken up when that event has been triggered. Due to implementation specifics, the event identifier should be unique across the entire application. One way to meet this requirement is to use addresses (pointers) as event identifiers. In this case, the address of the FSM function is used. In order to trigger an event, the FSM can use the *trigger* function and specify the event identifier, as done by the *root* FSM in this example application.

The *receiver* FSM waits for new packets from the network interface and shows the sender ID, sequence number and sensor value contained in the message, as well as the Received Signal Strength Indicator, which represents the signal strength when the packet was being received.

The *diag* function is provided by PicOS for displaying diagnostic information. When the application runs on real nodes, the diagnostic messages are sent through the node's serial port (directly, without the usual queueing and involved interrupt handling). When the application is being simulated, the diagnostic messages are being sent to the simulator's standard output, annotated with timestamp and node ID, as shown in the next section.

3.2 VUEE-based node and network simulation for PicOS applications

A feature of PicOS that makes it particularly suitable for pre-deployment simulation of sensor networks is that it integrates with the VUEE [8] simulator in a way that allows for efficient source-level simulation of PicOS applications. PicOS abstracts the hardware layer of the sensor nodes and VUEE provides a simulation of the hardware components, which is accurate enough so that properly designed applications do not need to be aware that they are running in the simulator. This allows for the same application-level code to run on both the real nodes and the simulated nodes in VUEE, without incurring the cost of virtual machine based solutions and without the need for machine code emulation of the processors in the sensor nodes. VUEE simulates the networking environment by taking into account the positions of the nodes, the capabilities of their transceiver and the specified parameters of the RF channel.

The simulation environment for a particular application is defined in one or more XML files. The configuration for the sample application presented in Section 3.1 is provided here. It is split into two XML files: the nodes configuration file and the RF channel configuration file.

The nodes configuration is provided below:

nodes.xml

```
<network nodes="3">
    <grid>0.1m</grid>
    <xi:include href="channel.xml"/>
    <nodes>
        <defaults>
            <memory>2048 bytes</memory>
            <leds number="3">
                <output target="socket"></output>
            </leds>
            <radio>
                <power>7</power>
                <preamble>32 bits</preamble>
                <lbt>
                    delay           4msec
                    threshold       -109.0dBm
                    tries           5
                </lbt>
                <backoff>
                    min         2msec
                    span            63msec
                </backoff>
            </radio>
        </defaults>
        <node number="0">
            <location>1.0 1.0</location>
            <sensors>
                <sensor number="0" vsize="2">4095</sensor>
                <input source="string">
                    T 1
                    S 0 400
                    E 0
                    T 5
                    S 0 10
                    E 0
                </input>
            </sensors>
            <ptracker>
                <output target="device">ptracker_node_0</output>
                <module id="cpu">0.3 0.0077</module>
                <module id="radio">0.0004 16.0 30.7 30.7</module>
            </ptracker>

        </node>
        <node number="1">
            <location>10.0 10.0</location>
            <sensors>
                <sensor number="0" vsize="2">4095</sensor>
                <input source="string">
                    T 2
                    S 0 350
                    E 0
                </input>
            </sensors>
```

```
        </node>
        <node number="2">
            <location>20.0 20.0</location>
            <sensors>
                <sensor number="0" vsize="2">4095</sensor>
                <input source="string">
                    T 3
                    S 0 550
                    E 0
                </input>
            </sensors>
        </node>
    </nodes>
</network>
```

The *defaults* element specifies the default configuration for each node's simulated hardware. Individual parts of it can be overridden in the definition of the individual nodes if necessary. In this example the simulated hardware includes LEDs and the radio. Each node is described in a *node* element. At minimum, each node must be given an ID (specified in the "number" attribute) and an X and Y location.

The node ID/number is made available to the application in the *host_id* global variable. In VUEE, this is handled automatically. When the application runs on a real node, the *host_id* constant is pre-defined with a certain magic value and PicOS provides a script that replaces it in the program image, before loading it on the node. This ensures that the same IDs are used both for simulation and in the real deployment and that this ID can be accessed by the application easily by using the *host_id* variable. The location is used when simulating the RF channel.

As mentioned in Section 3.1, PicOS provides a sensors API, which the example makes use of. This provides an easy way to simulate the sensors in VUEE, in such a way that the same application code that runs on the sensor nodes can run in the simulated environment, provided that the sensors themselves are simulated adequately. In this example we provide sample sensor changes in the configuration file using the *sensors* element. For each simulated sensor (only one in this example) we have a *sensor* element that specifies the sensor number, the size of the return value and the maximum value of the sensor.

The *input* element specifies how simulated data should be provided to the sensor. Normally this is done either through a file or a socket but in some cases, it can be provided directly, as is the case here. There are three types of commands, identified by the first character on each line. The *T* command tells the simulated sensor to wait for some time before processing the remaining commands. "T 1" means it should wait for one second. The *S* command instructs the simulator to set the value of the specified simulated sensor to the given value. "S 0 400" means that the value of sensor 0 should be set to 400. Finally, the *E* command indicates that a "sensor changed" event should be generated. This means that any FSMs waiting for this event (using the *wait_sensor* function) should be awoken in the their respective requested states.

While VUEE provides the simulated sensors API, the sensor models themselves need to be implemented separately. The simulation framework, developed as part of this thesis and described in later sections of this chapter, implements models of some sensors and also provides a way for interfacing with external sensor models. Both of these approaches have been used in simulations of the Smart Condo application described in Chapter 2. The framework then provides VUEE with the corresponding commands in the above-described format in order to manipulate the simulated sensors.

VUEE also provides a power/current consumption tracking module, called *ptracker*. It takes the consumption values for the different components of the node (CPU, radio, sensors, storage, etc.) and it will keep track of the amount consumed as these components get enabled/disabled throughout the simulation. The consumption values are specified in the *ptracker* element, with each simulated module for which power should be tracked specified in a *module* element. The *output* element specifies how the tracked information should be outputted, either a file or a socket (file is specified as *device*). The CPU module takes two values: the consumption when the CPU is running and when it is sleeping. The radio takes four values: when it is idle, when the receiver is on, when the transmitter is on (during packet transmission) and when both the receiver and transmitter are on. The values in the example's configuration are in milliamps and are based on the hardware of the nodes in the Smart Condo (described in Section 2.2).

The RF channel model configuration file is provided below:

channel.xml

```
<channel bn="-110.0dBm">
    <propagation type="shadowing" sigma="1.0dB">
        RP(d) = received power at distance d
        XP    = transmitted power
        X     = lognormal random Gaussian component
        ========================================================
        RP(d)/XP [dB] = -10 x 3.7 x log(d/1.0m) + X(sigma) - 33.5
        ========================================================
    </propagation>
    <cutoff>-115.0dBm</cutoff>
    <ber>
        Interpolated ber table:
        ======================
          SIR         BER
        50.0dB      1.0E-6
        40.0dB      2.0E-6
        30.0dB      5.0E-6
        20.0dB      1.0E-5
        10.0dB      1.0E-4
         5.0dB      1.0E-3
         2.0dB      1.0E-1
         0.0dB      2.0E-1
        -2.0dB      5.0E-1
        -5.0dB      9.9E-1
    </ber>
```

```
    <frame>8 12 0</frame>
    <rates>9600</rates>
    <rssi>0 -202.0 255 53.0</rssi>
    <power>
        0   -30.0dBm
        1   -15.0dBm
        2   -10.0dBm
        3    -5.0dBm
        4     0.0dBm
        5     5.0dBm
        6     7.0dBm
        7    10.0dBm
    </power>
</channel>
```

The *bn* attribute of the *channel* element specifies the background noise level. The *propagation* element configures the radio propagation model, i.e. how a signal transmitted at point A should be received at point B. VUEE uses the propagation information and the background noise level to calculate the strength of the signal received by all nodes in the network, given the location of the transmitting node and the locations of all other nodes in the network. The *cutoff* element specifies the minimum threshold necessary for a signal to be seen at all by the node's receiver. Then, the Bit Error Rate (BER) table, specified in the *ber* element is used to calculate the probability of bit errors in the received packet. The packet delivered to the PicOS network interface will reflect any errors that the simulator introduces here.

The main propagation model used by VUEE is the *shadowing* model. This model is much more realistic than the simple deterministic circular models described in Section 1.2 while still having a manageable number of parameters. It is selected by setting the *type* attribute of the *propagation* element to *shadowing*. The remaining parameters are specified inside the *propagation* element. Note that only the numbers are actually interpreted by VUEE. The rest is only used for documentation and is ignored by VUEE. These parameters include: the loss exponent (3.0 in the example), the reference distance (1.0m) and the calibrated loss at the reference distance (38.0). X(sigma) is a Gaussian random variable with zero mean and standard deviation of sigma (as specified in the *sigma* attribute) of the element. The model parameters in the example configuration are the default VUEE values. In reality, some or all of them can be measured in the actual environment where the network would be deployed and the model can be configured with the measured values in order for the simulation to be more faithful.

The *rssi* element describes how the received power should be converted to an RSSI indicator provided by the simulated transceiver. The *power* element specifies the transmit power levels for each of the power settings provided by the transceiver. For faithful simulation these parameters should be the same as the corresponding values of the real transceivers.

When running the simulation, VUEE provides the output of each node's diagnostic messages (when the application uses the *diag* function, as described in Section 3.1) to the standard output of

the simulator process. An example of such output for the example application is provided below:

```
[1.000000] <0> /sender 24/ DIAG: TX_SEND[SEQ=0,SENS=400]
[1.022240] <1> /receiver 18/ DIAG: Got packet SRC=[0] SEQ=[0]
SENS=[400] RSSI=138
[1.022240] <2> /receiver 13/ DIAG: Got packet SRC=[0] SEQ=[0]
SENS=[400] RSSI=125
[2.000000] <1> /sender 19/ DIAG: TX_SEND[SEQ=0,SENS=350]
[2.022240] <0> /receiver 23/ DIAG: Got packet SRC=[1] SEQ=[0]
SENS=[350] RSSI=136
[2.022240] <2> /receiver 13/ DIAG: Got packet SRC=[1] SEQ=[0]
SENS=[350] RSSI=137
[3.000000] <2> /sender 14/ DIAG: TX_SEND[SEQ=0,SENS=550]
[3.022240] <1> /receiver 18/ DIAG: Got packet SRC=[2] SEQ=[0]
SENS=[550] RSSI=137
[3.022240] <0> /receiver 23/ DIAG: Got packet SRC=[2] SEQ=[0]
SENS=[550] RSSI=124
[5.000000] <0> /sender 24/ DIAG: TX_SEND[SEQ=1,SENS=10]
[5.022240] <1> /receiver 18/ DIAG: Got packet SRC=[0] SEQ=[1]
SENS=[10] RSSI=137
[5.022240] <2> /receiver 13/ DIAG: Got packet SRC=[0] SEQ=[1]
SENS=[10] RSSI=125
```

Each line corresponds to a call to *diag* and includes the following information:

- Timestamp (simulated time)
- Node ID
- Name of FSM that was running when the diagnostic message was generated
- The actual diagnostic message

In addition, VUEE can communicate information about the nodes, as well as the simulated hardware, through several mechanisms, including files and sockets. In this example, this includes the node locations, which can be manipulated at run-time, if needed and the simulated LEDs. VUEE comes with a tool called *udaemon*, which provides a graphical interfaces for interacting with the simulation. One of them is the *ROAMER* interface, which provides a visualization of the node locations and ability to easily move nodes around. Figure 3.1 shows the ROAMER interface for the example application, which has three nodes.

The *ptracker* output for node 0 is shown below:

```
U 0000.000 [0000.000]: 0 0.3004
U 0000.000 [0000.000]: 0 16.3
U 0001.004 [0001.004]: 16.3 31
U 0001.022 [0001.022]: 16.56364 16.3
U 0005.004 [0005.004]: 16.35386 31
U 0005.022 [0005.022]: 16.40732 16.3
```

Each line corresponds to a change in the node's consumption. It contains the following information:

- Time of the change in consumption

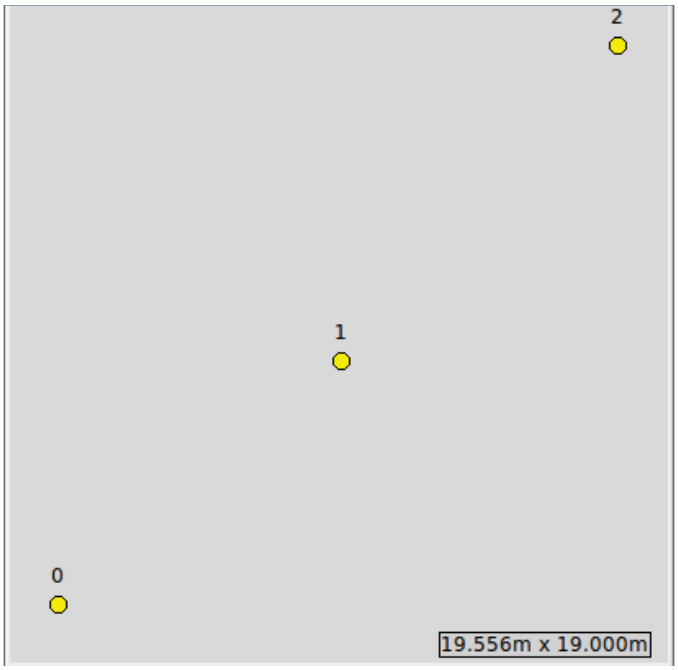

Figure 3.1: An example of the ROAMER user interface for visualizing and manipulating the nodes' locations.

- Time of the change since last reset (in this case there has been no reset, so the two timestamps are the same)
- The accumulated average consumption up to this point
- The current consumption (after the change)

In this example we have the node starting both the CPU and the receiver at time 0, resulting in a current draw of 16.3 mA. Then, between times 1.004 and 1.022, the node is transmitting its first packet. The transmitter is on during this time, leading to consumption of 31 mA. Between 1.022 and 5.004 only the CPU and receiver are on and the consumption is again 16.3 mA. Similarly, we can see the change during the transmission of the second packet, between time 5.004 and 5.022, drawing 31 mA and then going back to 16.3 mA drawn by the CPU and receiver. By time 5.022 we can also see that the average draw since the beginning has been 16.4 mA. Note that this is a very significant amount for low-power WSNs and in this example is caused by leaving the receiver on for the whole time. This allows a node to receive a packet at any time at the cost of large continuous consumption. In real applications, the receiver should be on only for a small portion of the time, as discussed in Chapter 1. For example, in the Smart Condo, the receiver is only on briefly after transmission in order to be able to receive a response (usually an ACK) after a packet has been sent. In section 3.4 we explore an example where an application does need the receiver to be on for longer periods of time and the receiver is being duty cycled to conserve energy.

3.3 Combined Pre-Deployment framework

The comprehensive sensor network simulation framework, developed as part of this thesis, combines simulations of the sensors and the network environment and allows the evaluation of both logic implementation and the performance of the network with any particular configuration of sensors. The framework is based on PicOS and VUEE (described in sections 3.1 and 3.2) and is, in part, motivated by the Smart Condo project (described in Chapter 2).

The main features of the simulation framework include:

- placing nodes and sensors in various configurations in the simulated space
- ability to interface with third party extensions for sensor models, localization algorithms and trace generation
- automatically running simulations with different configurations (placements/algorithms/parameters)
- producing performance reports to evaluate the different configurations

We generally start with the floor plan of the simulated space and proceed to place the sensors in it. The sensor placement can be inputted directly into the pre-deployment framework, or a third party extension can be used to select an optimal placement, such as the Sensor Placement Optimization algorithm described in [25]. In addition, the placement of the wireless sensor nodes needs to be decided. In the simplest case the node placement configuration can be one node per sensor, however it is possible, and often desirable, to have several sensors per node. A key feature of the pre-deployment framework is the ability to evaluate different placements of sensors and nodes in order to help in determining the optimal placement. To this end, the user can provide many different placements, run simulations with each of them and evaluate the performance of the network in each case.

In order to be able to evaluate the system, sensor models need to be chosen for each sensor that is used by the simulated application. The pre-deployment framework provides its own simple sensor models and also allows interfacing with third party models. For some sensors, such as switches, seat sensors, appliance usage sensors, the models are straightforward. However for others, such as motion sensors and RFID readers, they can be more complex. The original version of the framework provided its own simple circular sensing area model for the motion sensors. This model is still available, however it has been mostly superseded by the more accurate models provided as part of the Localizer tool developed by Vosoughpour [28]. The pre-deployment framework can interface with these models automatically by providing them with the sensor placements, getting the sets of triggered sensors at any point in time and using those to provide input to VUEE's simulated sensors (described in Section 3.2).

Another requirement for performing such evaluation is choosing activity traces. A trace is defined as a sequence of movements and/or actions performed by the simulated entity. The pre-deployment framework provides both its own simple trace generation tool and can also interface with more sophisticated external trace generation software. The internal tool can produce movement traces in the form of a sequence of lines, given start/end points and speeds. The external tools, supported by the framework, provide graphical interface for producing the traces. Once the traces have been generated, the pre-deployment framework can run simulations with each of them and can compare results.

The metrics that can be used for evaluation currently include localization error and energy cost, however the framework is designed to be extensible and new metrics can be added as needed. For localization applications, which are the primary target, the framework can interface with tools such as Vosoughpour's [28] Localizer in order to analyze how changes in the WSN communication protocols or the network environment can affect the accuracy of the localization. For evaluating the energy costs, the VUEE ptracker data (described in Section 3.2) is used to determine the energy used by each node during a simulation run.

Once the different parameters (including network protocols/parameters, algorithms, placements, etc.) to be evaluated have been chosen, the pre-deployment framework can automatically run a number of simulations and produce reports in the form of graphs that can be used to evaluate performance based on the metrics being considered. The steps comprising the simulation process are outlined below. This assumes that the Localizer tool is used for sensor models and for localization and that we already have the sensor and node placements (can be many of them if we are evaluating different placements, in which case the steps below will be repeated for each placement) as well as the traces (again, we can have multiple). The steps are:

1. Produce VUEE configuration files given a particular placement of sensors and nodes
2. Produce the sensor configuration in the format required by the Localizer
3. Use the trace and the sensor models to produce a sequence of sensor triggering events
4. Use the sensor events from 3. to produce the VUEE sensor input files in the required format
5. Run VUEE as well as the OSS application that normally runs on the bridge. At this point, the simulated sensor triggering goes through VUEE and the reported sensor readings from the simulation are published to the respective MQTT topics just like they would be in the real deployment.
6. Obtain location estimates from the Localizer and perform error analysis
7. Obtain power consumption information from VUEE's ptracker

8. Produce graphs showing location error and power consumption for the different parameters being evaluated

3.4 Example: Adding High Power Sensors

For most nodes in the Smart Condo (described in chapter 2), the power consumption of the sensor itself is low enough that the sensor can be constantly running without a significant impact on overall consumption. However, there can be sensors that provide better accuracy at the expense of higher consumption, e.g. RFID readers. The RFID technology utilized uses passive tags (without batteries of their own) that communicate with the reader by modulating reflected RF power received from the reader. This means that the reader needs to use a significant amount of power in order to be able to detect a tag.

Specifically, the RFID readers used for the Smart Condo can draw a current of more than 1A when the RF field is on. Normally the required granularity for reporting activities is one second and the reader can still detect the tag by keeping the field on for only a portion of each second. This allows for duty cycling the RF field by keeping it on for example for 100 ms for every second. That reduces the average consumption to 100 mA, however this is still a significant amount. For comparison, the passive infrared motion sensors draw about 100 uA.

For nodes with such high-power sensors, it becomes worthwhile to consider only keeping the sensor powered on for part of the time. This can be addressed in two ways. The first one is duty-cycling the sensor, so that it is only powered on for a portion of the time. The second involves having the node listen to readings sent by nearby nodes, which themselves have low-power sensors (e.g. motion sensors) and only enable their high-power sensor if a person is likely to be near their sensing area in the immediate future. This can reduce the overall consumption of the sensor because in this case it would be only powered on when nearby sensors have been triggered. The second approach is expected to provide lower consumption, however, the receiver itself draws a significant amount of current and the receiver needs to be on for the node to be able to listen to events reported by other nodes. For optimal results, the receiver can be duty-cycled and the node will only listen to others for part of the time.

Using duty-cycling can have negative effects on the accuracy of the localization because the more accurate sensors are no longer able to provide readings during the off-time of each period. This presents a trade-off between power consumption and location accuracy. The tradeoff for a particular sensor configuration and specific models of patient movement can be investigated in the simulated environment. The rest of this section provides a specific example where this trade-off is explored.

In this example, we consider a person walking in a straight line in an area that has both motion sensors and RFID readers. The simulated person walks for 11 meters at a speed of 0.3 m/s and then after 10 minutes walks back to the starting point. The motion sensors are always on and report every

detected event. For the nodes with RFID readers, we consider the duty-cycling approach in the two methods of operation described earlier, i.e. with and without listening to readings reported by nearby nodes. We look at how both the location error and the average current consumption change as the duty cycle is varied.

In Figure 3.2 we can see how the location error decreases as the RFID node's duty cycle increases. In Figure 3.3 we can see how the average current consumption over the 10 minutes varies with the duty cycle for the two different types of duty cycling.

In both cases the period of the cycling is 10 seconds. A duty cycle of 0.1 means that the sensor and/or the receiver was on for 1 second out of every 10 second period.

The "without listening" line corresponds to the first version of the system, where the RFID nodes do not listen for readings from any other nodes but duty cycle their readers all the time. As expected, this leads to large consumption due to the large draw of the RFID reader. The relationship is almost equal to duty cycle times the sensor's draw, meaning that almost all of the energy is spent on the sensor. This is expected because during the 10 minutes, the only transmissions are when the person actually walks, which is about half a minute at the beginning and half a minute at the end and even then the energy spent by the radio is significantly smaller than that spent on the RFID reader.

The "with listening" corresponds to the improved version, where the RFID nodes listen to readings reported by motion sensors and only turn the power-hungry sensor for a few seconds when they overhear such readings from nearby motion sensors. The duty cycle here is of the node's receiver. Here the energy used by the receiver is a much more significant fraction of the total energy spent. We can see that the overall energy efficiency of this method is significantly better and the difference gets bigger as the duty cycle increases because the high-power sensor draws significantly more current when it is on than the receiver (100 mA vs 16 mA in this case)

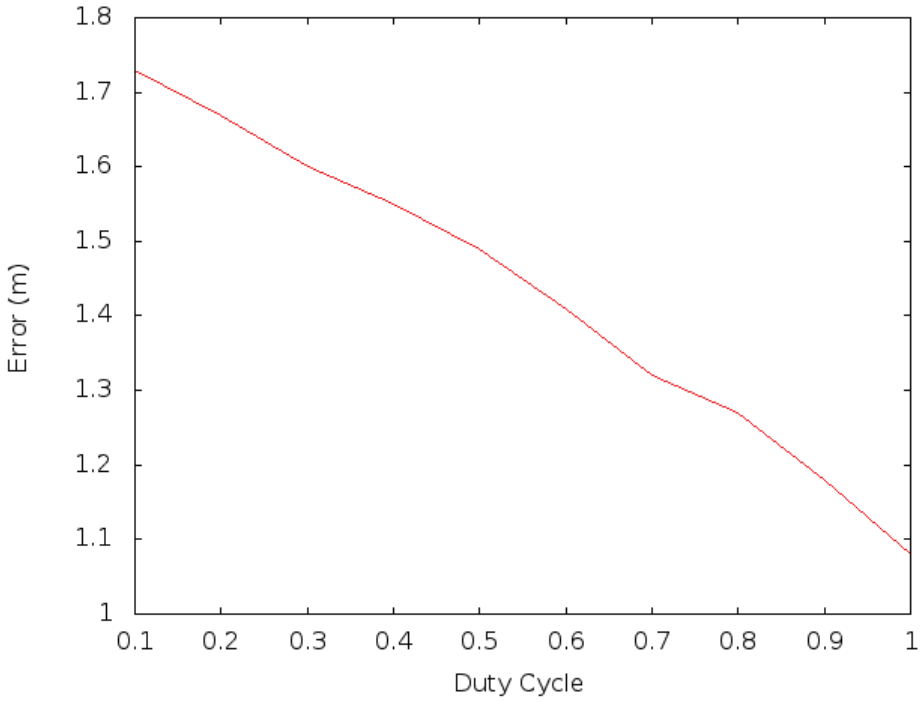

Figure 3.2: Location error versus RFID sensor duty cycle

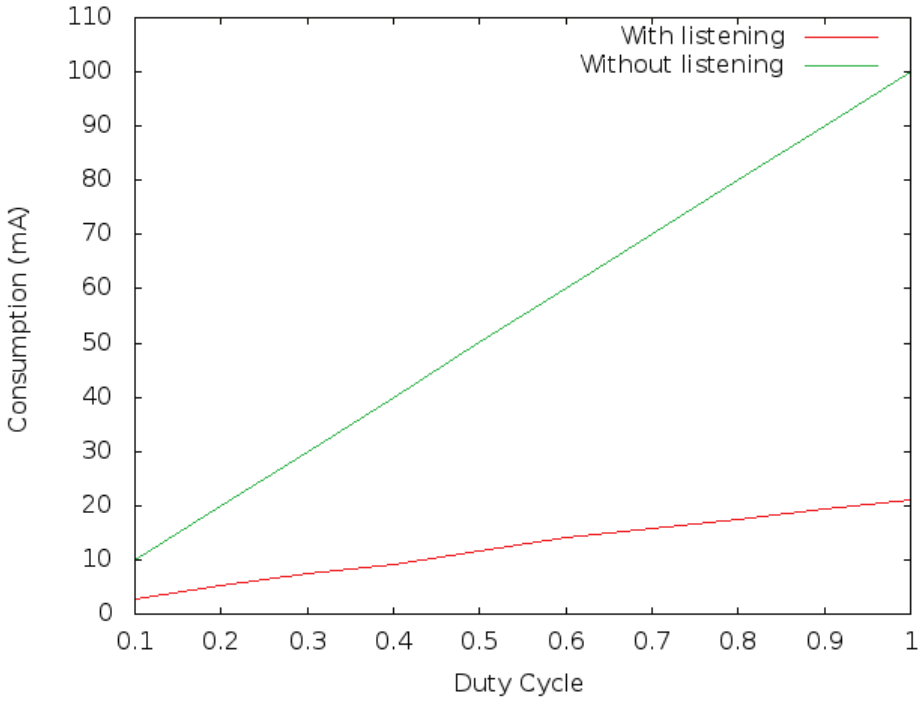

Figure 3.3: Consumption of nodes with RFID sensors versus duty cycle

Chapter 4

Post-deployment Software Updates

The system described here has been developed to address the need for reliable and efficient wireless updates of the software that runs on Wireless Sensor Networks. It generates a patch given two versions of a program and can distribute it to all (or some) nodes in a WSN. This section describes the structure of that system and details some key aspects of the implementation.

The update system consists of three major components: a patch generator, an update manager and a patch processing component. The patch generator is responsible for taking the two versions of a program and generating an efficient patch that can be used by a node that has the first version in order to reproduce the second version. It runs on the ”host”, a regular computer system used by the user to initiate the patching process. The update manager is used for ensuring that the update gets to the nodes and that it is applied successfully. Part of it runs on the host as well and another part runs on the sink. The patch processing component runs on the target nodes that are being updated and is used for accepting a patch from the sink, decoding it and applying it. Most of the code that runs on the sink and nodes is compatible with VUEE in order make testing and debugging easier. However, the actual updating of the running image and then rebooting into a new one only works on the real nodes. In VUEE (and in the encoder, when it verifies the patch) the writing to the code flash is emulated by having it write to a simple byte array that represents the memory space, so that most of the logic before the reboot can be simulated and thus tested in the development environment as opposed to directly on the node.

4.1 Patch Generator

The goal of the patch generator is to utilize the large difference in computing power and available energy between the host machine and the nodes being updated. Specifically, it uses algorithms that aim for producing a patch that is as small as possible (and therefore taking less energy to distribute to the nodes) at the cost of more processing during the generation (which is usually not a problem). It takes two versions of the program, one being the version that the node has and the other — the new version and generates a patch that can be sent to the sink for distributing to the nodes. The

actual generation is split into three main parts: address patching, delta script generation and patch verification, which includes collecting statistics about flash operations.

4.1.1 Address Patching

One of the major reasons for the large differences in the binaries of two versions of a program, even when the change in source code is small, is shifting in addresses, caused by extra code being added in the middle of the program. This causes the addresses of functions and variables that are located after the change to increase and therefore all of the instructions that are referring to these addresses will be automatically changed, even though nothing has changed about them on the source code level. The problem is amplified by the fact that typically a WSN environment lacks facilities such as dynamic linking, so normally the application, the operating system and any libraries being used get compiled and linked together into a single image. One possible approach to this problem is to split the initial program into sections that initially have blank areas, so that they can be expanded partially without affecting the rest of the application. However, there are several issues with that option. First, doing this would significantly decrease space available for the actual application code and since WSN nodes are already very limited in the amount of such space they have, further reductions may not be acceptable. Also, the extra space needed in one section may be more than what such a method would provide, so the update would not be possible even though overall there is enough free space for it.

Another approach, which is taken by my system, is to give the node information about where the shifts occur and to have it go through the machine code and attempt to automatically update the addresses. The encoder runs exactly the same code that the node would in order to get to the same intermediate version (original with addresses patched) as the the node. Then the delta script needs to be produced between that intermediate version and the new version. If the addresses are correctly patched, this will result in a smaller patch than if the delta had to include every single address change. Further, it does not require the insertion of gaps throughout the image, so it does not waste space.

There is, however, one serious difficulty with this approach in many cases it is not possible to determine with certainty whether the argument to the instruction is an address or something else. And while patching non-address words does not affect the correctness (the delta script is always generated after the address patching, so it compensates for such errors) it may actually make the patch bigger due to large number of corrections needed after the address patching, for parts of the image that did not actually change.

In order to reduce such errors, the generator analyzes the binaries using utilities from the GCC toolchain and identifies which parts are functions and which are constants. The idea is that anything in the constants is almost certainly not actual machine code, so trying to do address patching there would most likely have negative effects. Compressed information about these sections is included

with the meta data of the patch. It is possible that the overhead of this meta data may outweigh the benefits of patching some instructions or it may even turn out that for some a particular program, even for the code sections it may be better not to patch some instructions. The encoder deals with that by trying several options and comparing the number erroneous patches in each of them before deciding which should be used in the current patch. The instructions being considered for patching are CALL, MOV and PUSH. CALL instructions are not susceptible to the issues described above since their arguments are always addresses. MOV and PUSH, however, can be either way, so the above rules are used to decide how to process them.

Another optimization done by the patch generator to reduce the address shifts is to place code that is more likely to change closer to the end of the binary. For example, the code of the application (as opposed to that of the OS) is usually more likely to change, so it is placed closer to the end.

4.1.2 Binary Delta Script

The delta script specifies the instructions that a node should execute in order to produce the second version of the program assuming that it has access to the first version. The instructions can be SHIFT(x), COPY(from, len) or ADD(len, byte0, byte1, ...). SHIFT(x) means that the current patch pointer should be moved forward by x bytes. This is used for parts of the program that are identical between the two versions. COPY(from, len) means copy ”len” bytes starting at ”from” byte position in the old version. This is used in order to reduce the patch size when the bytes needed are available somewhere in the old version. And finally ADD is used for bytes that are new and need to be explicitly included in the patch.

Since this part comprises most of the final patch, special care is taken to reduce to footprint of these instructions. Each of them has a type, a length and some have extra data. The length can be either encoded in the same byte as the type or it can take one or two extra bytes, depending on the size of the number representing the length. Also, whenever equivalent information can be represented by different instructions the generator attempts to use the combination that uses the least space. For example a single SHIFT takes less bytes to encode than a single COPY because the COPY has a ”source” attribute and in the SHIFT the source is always the same as the current position. Similarly if there are several SHIFTs in a row or several compatible COPYs they can be compressed into a single SHIFT or COPY instruction. However, if there is a SHIFT between two COPYs and if all three can be replaced by a single COPY then that would be cheaper even though the individual SHIFT is cheaper than the individual COPY, so the delta generator looks for situations like this. Also, in general ADDs are the most expensive and should be avoided if the same information can be expressed with COPYs. However a short COPY between two ADDs might be actually worse than a single continuous ADD because of the extra bytes needed to start the COPY and the second ADD, so the generator also looks for these cases.

	Patch Generator	Without address patching	Whole image
Changing ackSeq from 1 to 2 bytes	322	2,782	35,126
Adding a PicOS state and a short diag statement	260	744	35,140
Adding 20 lines of code	590	1,074	35,140

Table 4.1: Effectiveness of the patch generator

4.1.3 Patch Verification

After the patch has been produced, the generator runs it using the same code that the node would when it gets the patch. One of the reasons for this is so that problems with the algorithms used by the generator or by the decoder can be caught immediately rather than after the patch has been sent to the nodes. Also, it simplifies the debugging process since if the problem is caught at this stage it can be debugged like a regular application.

Another benefit is that when running the code the way a node would, it is possible to keep track of expensive operations such as flash reading/writing/erasing and they can be reported to the user before the patch is sent to the nodes.

4.2 Performance Evaluation

In order to evaluate how the different features described earlier in this chapter contribute to the reduction in size of the patches, we consider several examples:

- a change of a struct member from one to two bytes which causes many address shifts and demonstrates the benefits of address patching
- adding a new state to a PicOS thread and a simple diag statement in that state
- Adding 20 lines of code that implement an additional command in a serial-based UI of an application

Table 4.1 shows how patches produced by the generator for the these examples compare against versions without the address patching and against the whole new image, which would need to be transmitted if the delta scripts were not used.

4.3 Update Manager

The update manager is responsible for making sure that once the patch has been generated, it gets to all of the intended recipients and it coordinates the actual patching in an attempt to ensure consistency of the code running on the nodes, even if some of them fail to apply to patch.

The update manager takes a generated patch, breaks it into packets and sends it to the sink, ensuring that the sink receives it completely. The reason for doing this (as opposed to just having the sink forward packets and the host making all of the decisions) is that this approach can work efficiently even if the link between the host and the sink is unreliable and/or expensive. Currently the only implemented way of communication between the host and the sink is using a serial connection, but the protocol, can work over any physical connection. The current implementation uses the TCV/XRS serial module of PicOS, so the serial link can be shared with other processes if needed. This also makes it possible for ”diag” statements in PicOS to still work during the time the update manager components are communicating with each other.

After the sink has received and stored the patch, it broadcasts an ”announce” packet, which contains meta information about the patch, such as intended recipients, length, checksum (Figure 4.1). Then it expects to receive either an acknowledgement or an error notification from every node that is listed in the recipient list. Note that the protocol does not assume any underlying reliability functionality, so for every packet that the sink sends it expects either an acknowledgement or some other response and if it does not hear from a node in a certain amount of time, it re-sends the packet. Since the idea is that every node is in a consistent state when it comes to the software it runs, the sink does not proceed to the next step unless all of the nodes have replied. Similarly if any node responds with an error, the current patching process terminates and the user needs to decide what to do.

Once the sink has received acknowledgements to the patch announcement, it starts broadcasting the blocks that comprise the actual patch (Figure 4.2). Similarly, after every block, it waits for acknowledgements from every node (with a timeout and re-send) and does not continue to the next block until it has received that acknowledgement.

Once the nodes have all received the entire patch, they verify its checksum and attempt to build the new image by storing modified blocks in their external flash (and not touching the actual running code yet). Then they verify the checksum of the new image and notify the sink of the result (Figure 4.3). If any of these steps fail on any node, that node notifies the sink, which would then terminate the current update and forward the notification to the host for the user to see it. Since the individual packets are validated before being acked, any failures after all the blocks have been received indicate a problem that cannot be resolved automatically, hence the reason for terminating the update and letting the user decide what to do. In order to keep things simple we only have the acknowledgement/timeout/resend functionality on the sink, so in order to deal with the possibility of losing the message that contains this acknowledgement (or the error, if any has occurred) the sink periodically asks the nodes for their status after the last packet has been sent and before all of them have either acknowledged the rebuilding of the image or have successfully send an error notification.

If all nodes have acknowledged that they have successfully built the new image, the sink sends a ”go-ahead” packet to tell the nodes to actually copy the changed blocks to their code flash and

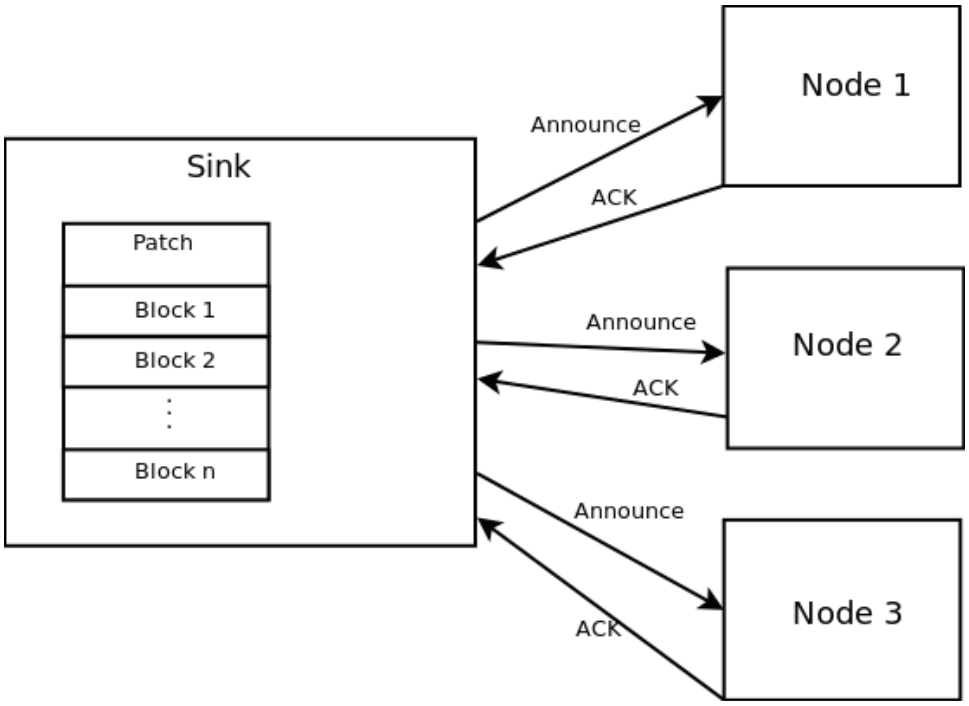

Figure 4.1: Once the sink has the patch, it sends an announce message to all target nodes.

reboot themselves (Figure 4.4). Then the sink waits for the final confirmation, from each node, that the patching has completed successfully (it periodically re-sends the ”go-ahead” until it hears something from each node). The reason for having the nodes wait for the ”go-ahead” message is to ensure that even if some nodes could not rebuild the image (e.g. some nodes are experiencing issues with the flash storage), all nodes would still keep running the original program as opposed to some running the old one and some — the new one. This may be especially important in case the update introduces a change in the communication protocols which if only partially applied could make it impossible for all of the nodes to communicate with each other.

Note that it is possible to patch the sink itself, in which case it stores the patch the same way as a node would but then instead of sending it to other nodes it just applies it the same way the other nodes would. This is currently only used for testing/debugging, however it could be useful in case the sink is not directly connected to the host (e.g. the sink could be in a remote location and communicating over a satellite link).

4.4 Patch Receiving and Processing

This component runs on the target nodes and is responsible for receiving a patch from the sink, verifying it and applying it while coordinating with the sink as needed.

When a node receives an announce packet (see Section 4.3), it first checks whether its ID is one of the recipients. If it is not, it ignores the patch. If it is, it acknowledges the announce, so that the

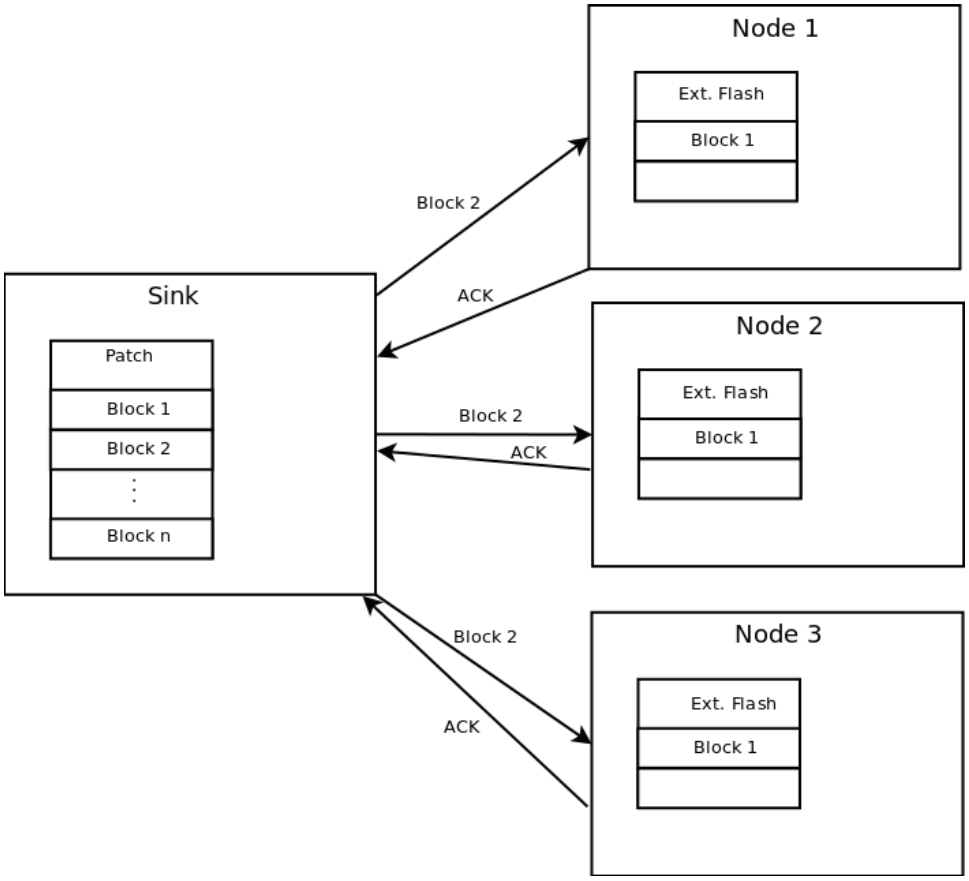

Figure 4.2: The sink sends it to all target nodes, block by block. In this example, three nodes are being updated. They have already received block 1 and currently block 2 is being transmitted. As each node receives it, it sends an ACK and stores the block in its external flash.

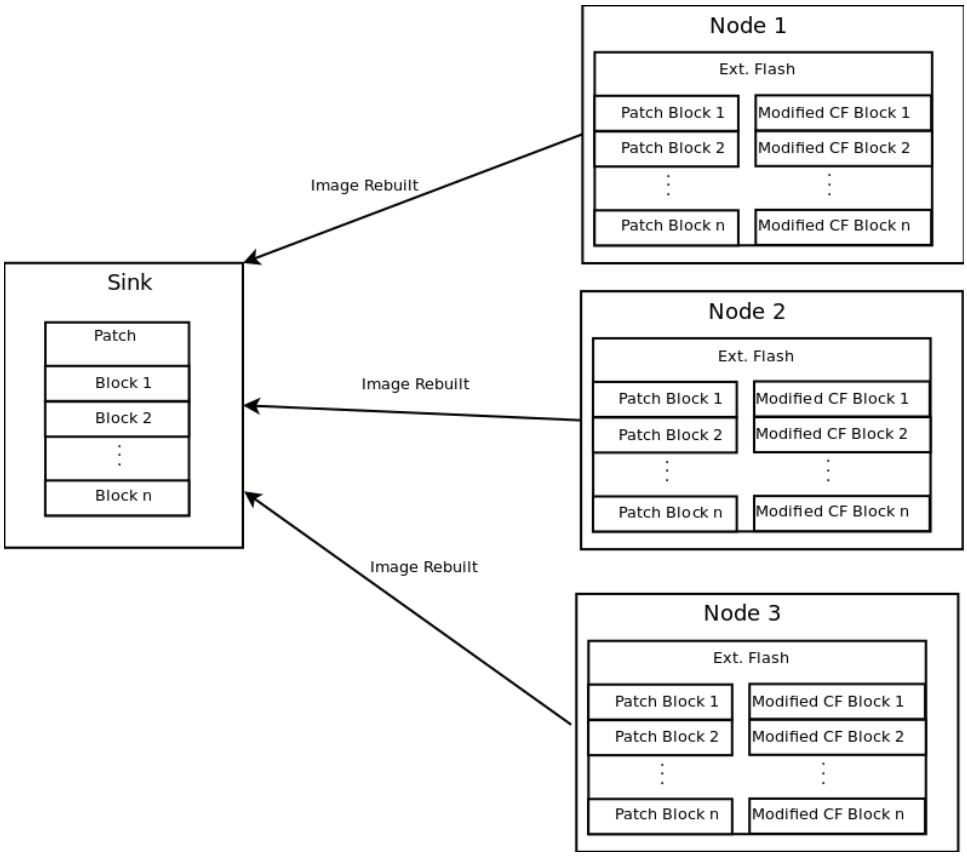

Figure 4.3: Once each node receives all blocks in the patch, it processes it but without modifying the Code Flash yet. Instead, each CF block that must be modified is written into the external flash. At this point, all blocks of the new image are available to the node. Those that are not modified are in the code flash and the modified ones are in external flash. By combining the modified and non-modified blocks, the node calculates and verifies the checksum of the new image. If the checksum matches, the patch processing has been successful and the nodes sent a confirmation to the sink specifying that the image has been rebuilt.

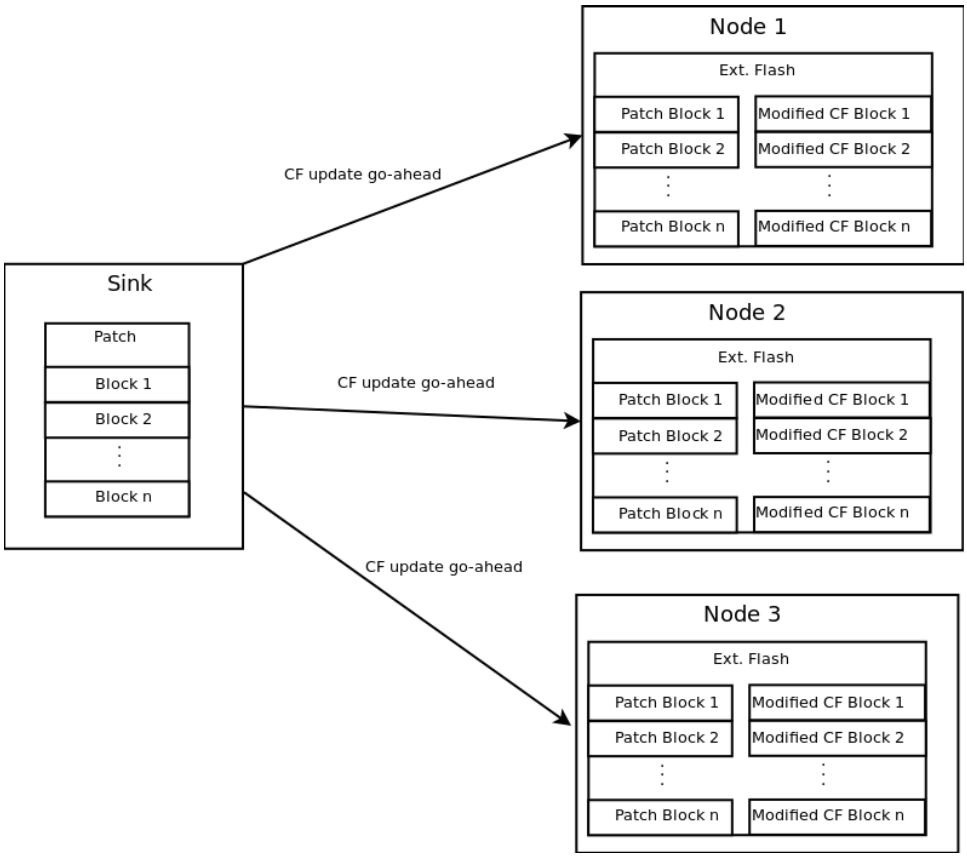

Figure 4.4: After the sink has received the "image rebuilt" confirmation from all nodes, it can tell all of them to go ahead with updating their code flash. This is done in a separate step so that if any of the nodes failed in processing the patch and rebuilding the image, none of them will be updated and all will still have compatible versions of the program.

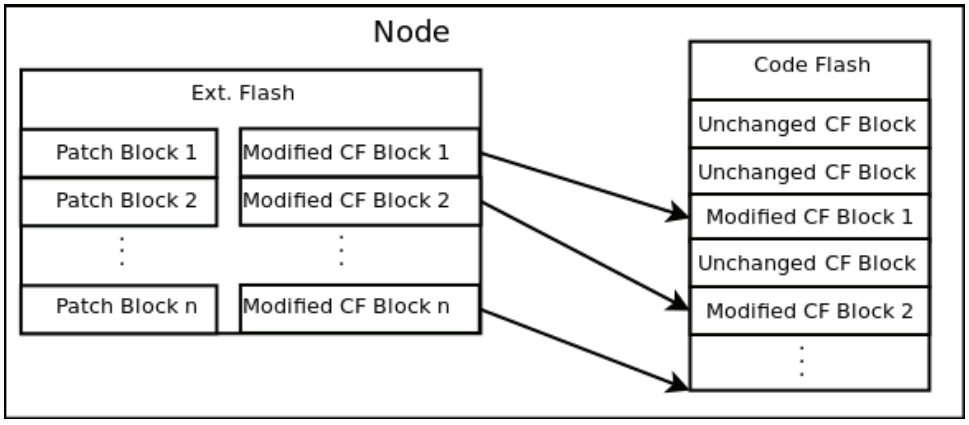

Figure 4.5: After the go-ahead has been received, each node will copy the modified code flash blocks to the actual code flash and will reboot into the new image.

sink knows that this node is ready to start receiving packets (Figure 4.1). In the case that the node cannot receive the patch (e.g. storage hardware error), it responds with an error message so that the sink can notify the host, which would notify the user of the problem. Then as the node receives each block from the sink, it stores it in its External Flash (EF) and acknowledges it (Figure 4.2). It is possible that the same block may be received several times, since some other nodes may not have acknowledged it, in which case the node acknowledges it right away (there is no need for storing it again).

As soon as the last packet has been received, stored and acknowledged, the node first verifies the checksum of the entire patch and then it rebuilds the new image by first doing the address patching according to the rules specified in the meta data (Figure 4.3). For every Code Flash (CF) block that actually needs to be modified as a result of this, the new contents gets written in EF. After that, the node similarly processes the delta script in the patch and any blocks changed as a result are again written to EF (blocks that were updated in the previous stage are rewritten in EF). An important detail here is that whenever a COPY instruction is executed the source of the copy can be either in the CF (if it hasn't been changed) or it can be in EF if it has changed during the previous stage. It can even be partially in CF and partially in EF if the source spans several blocks. These situations are taken into account by the patch processing module.

Once the patch has been processed and the modified blocks have been stored in EF, a checksum of the new image is calculated by reading each block either from CF (if unchanged) or from EF (if changed). Then that checksum is compared with the expected checksum for the new image and the result is sent to the sink (Figure 4.3). If the check was successful, the node waits for a "go-ahead" for copying the changed blocks to CF. When that message is received (i.e. all other nodes have successfully rebuilt the image in their EF) the node copies all of the changed blocks to CF and reboots itself (Figure 4.4).

The patch processing module includes a mechanism to give interested threads a chance to clean up before this final stage begins. This was implemented similarly to the Linux SIGTERM signal — a signal is sent to all threads that have requested such notification and then the system waits for some (configurable) amount of time before it proceeds with the reset. This was originally planned to use the standard PicOS wait/trigger mechanism, however doing it that way turned out to be problematic because in PicOS once a thread is awoken in a state as a result of waiting for an event, all other requests get cleared. Since it is unreasonable to expect processes to keep adding the request for this signal all the time and modifying this behaviour in PicOS was undesirable, a separate, very simple, notification mechanism was implemented for SIGTERM signals. This allows any process to register a state in which it would like to be awoken when such an event occurs. The patch processing module keeps track of PIDs and states for such processes. Then, when it is time to start writing to CF, it forces every such process into the registered state and waits for the configured amount of time before proceeding.

An important limitation during the copying of EF blocks to CF is that none of the code that might be rewritten can be executed. In order to accomplish this goal, first interrupts and the watchdog are completely disabled. Also, the code that does run during this time is placed in a special section at the end of the binary, and anything in that section is not patched (it is simply ignored by the patch generator). Further, this code is not allowed to interact with anything outside of that section since it may be inconsistent. This includes making function calls and accessing constants in object files that are not explicitly placed into the special section. Since code that copies from EF to CF needs to read from EF it must be able to use the storage driver, so the object files for the storage driver are also put in that section. Note that while nothing in this section is allowed to make outside calls, it is permissible for other parts of the code to call functions from this section. Finally, since node IDs should not change and in order to simplify the patch generation process the node ID constant is also placed in that section.

After rebooting, the node verifies the checksum of the data in CF and compares it to the expected checksum for the new image. Then it sends a notification to the sink that the patching has been completed. It is possible for this notification to get lost. This is addressed by having the node remember that it has been patched and if it hears another ”go-ahead”, it knows that the notification was not received and re-sends it.

There is still a possibility that something goes wrong with the new version so that after rebooting the node does not get to the point where it can accept further updates. This could be due to a problem with the internal flash, or possibly the new version of the image does not work on this particular node. In some cases it may not be possible to do anything (without physical access). However, if the basic initialization could be completed and if the execution can get to the special patching section, which should not have been touched during the update, so presumably it should still work, then it should be possible to restore the previous version and then perform another update over the network. A more robust solution would be to have a bootloader-like module located in the special section, which would reduce the chance of an update preventing the node from booting at all. Being able to restore an older version would require that the initial version of the image, before any patching has been done is available in EF and that after patching the old data does not get erased (or at least all of the data for the previous version is still there).

Chapter 5

Conclusion

Developing, testing, debugging and evaluation of software for Wireless Sensor Networks present a number of unique challenges. They include the minimizing of energy usage for long-term battery-powered WSN operation as well as being able to run on very limited hardware. In addition, the application is distributed over a large number of nodes, which introduces challenges that are typical for distributed systems. In this thesis, we presented two approaches to dealing with these challenges: pre-deployment simulation and post-deployment code updates.

The Smart Condo system is a specific real-world WSN application that has motivated parts of this thesis work. It is designed to monitor patient activities in assisted living facilities and to provide staff with information that can be useful in determining the patients' ability to manage on their own. A variety of sensors are utilized instead of cameras in order to address privacy issues associated with video surveillance.

The PicOS operating system is designed for event-based applications running on platforms with very limited hardware resources, which makes it suitable for WSN applications. Further, it provides flavour of multitasking, in which tasks are represented by finite state machines, in a way that makes the code implementing these tasks clear and self-documenting. At the same time, the multitasking model has low overhead and is able to operate on devices with as little as 1KB of RAM. We provided a sample PicOS application in Section 3.1, which includes three FSMs, illustrating this multitasking behaviour.

The VUEE simulator integrates with PicOS and allows for PicOS applications to be simulated by running the same application code that runs on the actual node. This allows PicOS applications to be tested, debugged and evaluated in a standard development environment. VUEE is capable of simulating some of the hardware components on, including sensors and radio transceivers. The radio communication channel between the nodes is simulated in a realistic fashion. In addition, VUEE can keep track of each node's power consumption, which can help evaluating the energy efficiency of an application. In Section 3.2 we demonstrated how our sample application can be simulated with a network consisting of several nodes. We highlighted and explained some the relevant configuration options. In that section we also showed what the output of VUEE is for this sample application with

the specified configuration and how it can be interpreted.

While VUEE allows sensors to be simulated, it does not provide any sensor models and expects that the simulated values are provided to it. Similarly, while it can provide information about the nodes' power consumption, analysis of that information is beyond its scope. Our combined pre-deployment framework, described in Section 3, utilizes VUEE in order to provide a complete simulation environment that integrates sensor models and can provide analysis of both generic metrics, such as energy consumption, and application-specific ones such as location accuracy in localization applications when exploring different application or protocol parameters or different node placements. In Section 3.4, we provided a simple example that illustrates how the pre-deployment framework can be used for evaluating application performance in energy usage and location error when adding a new type of a sensor to a localization application and exploring two different modes of operation and related parameters.

Post-deployment updates in WSN applications present a number of unique challenges. Wireless transmissions can be very expensive, which means that the updates must be as small as possible. In addition, nodes can often be deployed in inaccessible environments, meaning that an update failure can have very damaging effects if it leaves the node unable to communicate, to the point of rendering node useless.

We address these challenges in our update system by applying several methods of reducing the patch size, as well as by providing mechanisms for detecting faults in the update process before committing the node to the new version of the software, in addition to being able to simulate the whole update process prior to executing an update.

A key feature of the update mechanism is looking at the actual code and detecting address shifts. This allows the information for these address shifts to be encoded efficiently. Differences other than address shifts go through a process of generating delta script that can be executed on the node to reconstruct the new version of the code. Great care is taken during the generation of the script to ensure that portions of the new code that are already present in the old version are not included in the patch but instead instructions are included on how to produce the new version from the old one. These techniques provide significant reduction of the patch size, as seen in Table 4.1.

5.1 Future Work

The pre-deployment framework currently incorporates and integrates with sensor models for sensors used in the Smart Condo. The framework is designed to be extensible and in the future, more models, which would be useful for other applications, can be added. Currently the metrics supported by the framework for performance evaluation are energy usage and location error for localization-specific applications. In the future more such metrics can be added.

Several improvements can be made to the post-deployment system. One of them is the more robust handling of the case where the node appeared to have been patched successfully but something

goes wrong after the reboot, as discussed at the end of Chapter 4. Additionally, in the present system, the user needs to always supply both the old and the new image for patch generation, meaning that they need to know what image are the nodes currently running. A useful addition to the patch generator and the update manager would be the ability to query the nodes that are to be updated in order to figure out what version are they currently running and to automatically generate a patch.

Bibliography

[1] MQTT Specification. http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/MQTT_V3.1_Protocol_Specific.pdf.

[2] ns-3 Network Simulator. https://www.nsnam.org/.

[3] Panasonic Passive infrared human detection sensor with built-in amp. http://www3.panasonic.biz/ac/e_download/control/sensor/human/catalog/bltn_eng_mp.pdf.

[4] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[5] TI CC110x Radio Transciever. http://www.ti.com/product/cc1101.

[6] TI MSP430F1x Family of Microcontrollers. http://www.ti.com/lsds/ti/microcontrollers_16-bit_32-bit/msp/ultra-low_power/msp430f1x/overview.page.

[7] E Akhmetshina, Pawel Gburzynski, and Frederick S Vizeacoumar. PicOS: A Tiny Operating System for Extremely Small Embedded Platforms. In *Embedded Systems and Applications*, pages 116–122, 2003.

[8] N.M. Boers, P. Gburzynski, I. Nikolaidis, and W. Olesinski. Supporting wireless application development via virtual execution. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 853–860, Oct 2008.

[9] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[10] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin. Emstar: An environment for developing wireless embedded systems software, 2003.

[11] Veselin Ganev, Dave Chodos, Ioanis Nikolaidis, and Eleni Stroulia. The smart condo: Integrating sensor networks and virtual worlds. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 49–54, New York, NY, USA, 2011. ACM.

[12] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[13] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. Springer US, 2009.

[14] David Kotz, Calvin Newport, Robert S. Gray, Jason Liu, Yougu Yuan, and Chip Elliott. Experimental Evaluation of Wireless Simulation Assumptions. In *Proceedings of the 7th ACM International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '04, pages 78–82, New York, NY, USA, 2004. ACM.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In Werner Weber, JanM. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[16] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 85–95, New York, NY, USA, 2002. ACM.

[17] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[18] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, Nov 2006.

[19] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: A simulation framework for sensor networks. In *Proceedings of the 3rd ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '00, pages 104–111, New York, NY, USA, 2000. ACM.

[20] L.F. Perrone and D.M. Nicol. A scalable simulator for TinyOS applications. In *Simulation Conference, 2002. Proceedings of the Winter*, volume 1, pages 679–687 vol.1, Dec 2002.

[21] Niels Reijers and Koen Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proceedings of the 2Nd ACM International Conference on Wireless Sensor Networks and Applications*, WSNA '03, pages 60–67, New York, NY, USA, 2003. ACM.

[22] Pascal Rickenbach and Roger Wattenhofer. Decoding Code on a Sensor Node. In *Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS '08, pages 400–414, Berlin, Heidelberg, 2008. Springer-Verlag.

[23] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct code execution: Revisiting library os architecture for reproducible network experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM.

[24] I. Vlasenko, M. Vosoughpour Yazdchi, V. Ganev, I. Nikolaidis, and E. Stroulia. The Smart-Condo™ infrastructure and experience. In *Evaluating AAL Systems Through Competitive Benchmarking*, volume 362 of *Communications in Computer and Information Science, S. Chessa and S. Knauth*, pages 63–82. Springer, 2013.

[25] Iuliia Vlasenko. Deployment Planning for Location Recognition in the Smart-Condo™: Simulation, Empirical Studies and Sensor Placement Optimization. In *Masters thesis, University of Alberta, Edmonton*, 2013.

[26] Qiang Wang, Yaoyao Zhu, and Liang Cheng. Reprogramming wireless sensor networks: challenges and approaches. *Network, IEEE*, 20(3):48–55, May 2006.

[27] Katie Woo, Veselin Ganev, Eleni Stroulia, Ioanis Nikolaidis, Lili Liu, and Robert Lederer. Sensors as an Evaluative Tool for Independent Living. In Vincent G. Duffy, editor, *Advances in Human Aspects of Healthcare*, page 612621. CRC Press, 2012.

[28] Meisam Vosoughpour Yazdchi. Indoor localization with passive sensors. In *Masters thesis, University of Alberta, Edmonton*, 2013.