UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Primitive Based Approach for Managing, Deploying and Monitoring In-building Wireless Sensor Networks

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Seemanta Dutta

Committee in charge:

Yuvraj Agarwal, Chair
Rajesh Gupta
Ryan Kastner

2012

The thesis of Seemanta Dutta is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California, San Diego

2012

DEDICATION

To Ma, Deuta and Jethi

EPIGRAPH

We are all connected; To each other, biologically. To the earth, chemically. To the rest of the universe atomically.

–Neil deGrasse Tyson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am thankful to Prof. Rajesh Gupta and Dr. Yuvraj Agarwal for giving me an opportunity to work in their lab and on their exciting projects. Dr. Yuvraj Agarwal has been a constant source of solid advice shaping the work that you now hold in your hands. But more than that, he has been a source of great inspiration throughout my work. No matter where I was stuck, he always had a different perspective to offer. I am grateful to Prof. Ryan Kastner for agreeing to be on my thesis committee. I would like to acknowledge Thomas Weng, whose insightful comments not only on software design but also in general has made it possible to achieve my goals. I would also like to thank Bharathan Balaji and Alex Bishop for their help regarding the hardware and in troubleshooting those extremely difficult to catch hardware bugs. They had a positive influence on my software design through their constant feedback and discussions. I would also like to extend my thanks to Sathyanarayan Kuppuswamy and Michael Wei for their advice whenever I needed a different perspective on things. Ryan Mast was also spectacularly helpful through his skillful use of Altium design software during the initial days of designing our hardware. Steve Hopper from CSE help and Virginia McIlwain from the CSE staff deserve my eternal gratitude for their alacrity in resolving my IT issues and in helping me with ordering parts and equipment that I needed for my work. I am also thankful to Zin Kyaw, Engineer from Texas Instruments, San Diego; Mihir Dani, Design Engineer, Anaren Microwave Inc.; Derek Smith, Tesla Controls for anwering my questions on a wide range of subjects ranging from Anaren/CC2530 hardware to ZStack/ZigBee. But most important of all, I would like to thank my friends, family and parents for standing by me and supporing me throughout my graduate education. Among family, a very special mention goes to my aunt whose encouragement, guidance and help was always available whenever I needed it.

VITA

2003	B. Tech. in Electronics and Communication Engineering *Honors*, Malaviya National Institute of Technology, Jaipur, Rajasthan, India
2011-2012	Graduate Teaching Assistant, University of California, San Diego
2012	Graduate Research Assistant, University of California, San Diego
2012	Master of Science in Computer Science, University of California, San Diego

PUBLICATIONS

Yuvraj Agarwal, Bharathan Balaji, Seemanta Dutta, Rajesk K. Gupta, Thomas Weng - "Duty-Cycling Buildings Aggressively: The Next Frontier in HVAC Control" In *Proceedings of the 10th Conference on Information Processing in Sensor Networks: Sensor Platforms, Tools and Design Methods (IPSN/SPOTS '11)*, April 2011.

T. Weng, B. Balaji, S. Dutta, R. Gupta, and Y. Agarwal - "Managing PlugLoads for Demand Response within Buildings" In *ACM Workshop on Embedded Sensing Systems For Energy-Efciency In Buildings, 2011.*

ABSTRACT OF THE THESIS

A Primitive Based Approach for Managing, Deploying and Monitoring In-building Wireless Sensor Networks

by

Seemanta Dutta

Master of Science in Computer Science

University of California, San Diego, 2012

Yuvraj Agarwal, Chair

Wireless sensor networks have become quite pervasive in the last few years. As their technology has matured, they have transformed from an academic research area to a viable means of solving practical engineering problems. This dynamic field has several vendors who can provide the necessary software and hardware infrastructure in order to get up and running quickly. While it is relatively easy to get up and running with a 'laboratory' setup, it is a completely different story when it comes to deploying a real world wireless sensor network. Any real world deployment, whether it is outdoors or indoors, has its own unique challenges as the scale of the deployment increases. We have observed these challenges, especially involving indoor deployments while working on several research projects in

our lab. Our previous projects required a large and distributed deployment of an indoor wirelesss sensor network. Those *ad hoc* deployments were done manually, making us realize the need for management primitives to do things more efficiently. The solution as proposed in this thesis, is a set of technology agnostic management primitives that help in overall management of wireless sensor networks. We used ZigBee as it is quite popular for smart building applications. Leveraging our management primitives we built features to address these challenges of administering a wireless sensor network in a building. We have used these primitives to deploy a small scale sensor network comprising of 100 nodes, which we then used to demonstrate their benefits as well as evaluate our deployment.

Chapter 1

Introduction

Recent years have seen Wireless Sensor Networks(WSNs) become pervasive in academic as well as industrial circles. They have proved themselves as worthy tools in solving numerous engineering problems like structural health monitoring[17], home and industrial automation[5, 19, 23], measuring building occupancy[13], wireless energy metering to name a few. All of these are aided not only by the continuous development of hardware and software platforms but also by the advent of low power networking protocols that specifically target this domain, *e.g* ZigBee[11], 6loWPAN[1], ZWave[12] etc.

Notwithstanding whichever combination of hardware, software and networking technology we choose, there are a set of universal problems that need to be solved in order to have a WSN deployment that is robust, scalable and is manageable. For example, commissioning strategy, security, over the air (OTA) software upgrades and load balancing are some concerns that immediately come to our mind when we move out of a 'lab' mindset to an actual real world deployment with hundreds or even thousands of nodes spread across a huge area. At this scale, a lab setup with a handful devices no longer serves as the standard and deployment problems no longer remain trivial.

Some of these problems might be specific to the underlying WSN technology being used while some of them are at a higher level and loosely coupled with the underlying WSN technology. Still some of these problems might have to do with what kind of WSN deployment we are referring to, *i.e.* outdoor or in-building.

Several papers have been published that deal with these problems at larger scales [15, 18]. MAC level protocols have also been an active area of research [20, 16, 21]. However, many of the above mentioned problems of deploying a large scale network are orthogonal to the advancements in the MAC or in the upper layers themselves. This has led us to believe in having a rich set of primitives on top of the WSN technology, independent of it that would allow us to tackle these problems. This primitive based approach is what we believe, makes our solution not only scalable, but also portable for the most part.

Our area of focus throughout has been in-building WSNs due to the nature of our work (Smart Buildings [14, 13]). We feel a good deployment solution should try to cover all types of problems as mentioned above. In fact, this is the next emerging area in WSNs which has several competing entities offering a wide range of tools and means to solve some of these deployment problems. However, the cost of these tools remains prohibitively high at this stage, even for residential installations, let alone commercial buildings which is our area of interest. For example, control4 home automation only has the 'home' version of their composer software available for download. The 'pro' version of composer is made available only to dealers and is not openly available [4]. Almost all of these solutions are closed and are not amenable to study in an open academic environment. Furthermore, while many of the prominent vendors have provided libraries to tackle some of these problems, they have not provided end-to-end solutions to these deployment related problems. It is this lack of solutions which has given rise to such closed third party solution providers.

In this thesis, we identify the specific problems that we encountered during our previous WSN installations for our smart buildings effort. We then suggest solutions to these problems via hardware and software based approaches. We have tried to attack these problems bottom up, realizing the hurdles we ourselves faced during our past in-building WSN deployments. The underlying insight of our work was to develop a set of rich primitives that can be used to address the wide range of deployment related challenges when it comes to scaling WSNs in buildings. At the same time however, we decoupled the 'networking' part of WSN deployment

from the 'sensing' part of it. We did this by creating a testbed of devices that were created for the sole purpose of having the ability to quickly and efficiently deploy a large testbed of devices at short notice. This testbed formed the substrate for all of our experiments and observations. While they lacked any sensing capabilities, they supported a number of networking primitives that allowed us to tweak and change network parameters on the fly and evaluate their effects.

The rest of the thesis is organized as follows: Chapter 2 provides a background of in-building WSNs, including our own efforts and familiarizes the reader with the various challenges we encountered during our work. It also gives an overview of the goals and requirements of our testbed. Chapter 3 then talks about our hardware design and the choices we made for our testbed. Chapter 4 follows with a detailed architectural description of the overall software at a high level. Chapter 5 describes in detail our primitive based approach in addressing the challenges as outlined before. Chapter 6 rises a level above and explains some key functionalities that we have built on top of the primitives from the previous chapter. Finally, we tackle the issue of network security in Chapter 7, which is then followed by a high level discussion of our backend design in Chapter 8. Chapter 10 concludes the thesis by presenting an evaluation of our primitives and the functionality we have built using them.

Chapter 2

Background

As part of the ongoing smart buildings effort in SYNERGY labs at the department of Computer Science and Engineering at the University of California, San Diego, we have frequently relied upon WSNs to realize various goals. These goals have ranged from sensing occupancy of offices, to saving HVAC energy[13] and finally to auditing energy consumption of plug loads[22]. In all of these endeavors, the WSN was critical piece of the project, upon which the ideas of our project were demonstrated, deployed and evaluated.

We began by evaluating various WSN technologies and focused on ZigBee since we felt it was a very popular and well supported technology by the industry as well by an active community of developers. Our initial WSN iterations involved utilizing the Texas Instruments(TI) ZigBee platform which consisted of a CC2530/CC2531 SoC coupled with a complete implementation of the ZigBee software protocol stack. The CC2530/CC2531[3] SoC is an 8051 core along with a 2.4 GHz radio. The ZigBee stack is based on the IEEE 802.15.4 MAC, the implementation of which was also included as part of the TI ZigBee Stack. The CC2531 has an added USB library that allowed it to natively communicate with any host device with an USB implementation. [10] has more details on ZigBee and the terminology when referring to this technology.

Our initial experience with deploying the sensor nodes was not without issues. First of all, deploying sensor nodes was a time consuming process which involved drilling holes in the walls and then hanging our sensing devices and in-

stalling the reed switch sensors on the side[13]. Furthermore, whenever we encountered software bugs, we would have to remove all sensor nodes, re-flash them with the updated firmware and then re-install them back to their original locations. Smart commissioning was virtually non-existent because the device came pre-programmed with various network parameters and it never changed throughout the duration of the deployment.

From the beginning we always advocated a multi-star network architecture, rather than a mesh network (which is one of the strengths of ZigBee) because a certain fraction of our devices had high data rates and we wanted to avoid the negative impact of multi-hop networks on data rates. Having a multi-star network fundamentally changes several key characteristics of the network. We now have a collection of independent networks rather than a single unified multi-hop network. This introduces additional complexity in terms of how a particular device figures out which network to choose to join to when presented with multiple alternatives as it is quite possible that in the vicinity of a sensor node, there are multiple networks 'in-range'. On the positive side, having a multi-star network eliminated the need of intermediate, possibly mains powered routing devices as now the battery powered end devices can directly communicate with the coordinators. It also allowed us to support higher data rates without losing too many packets.

It was during these deployments that we realized that a unified and coherent way of deploying a WSN was needed. 'Deployment' here not only refers to the physical installation of the device and getting it to work, but also refers to more broader issues like choosing the best network possible for a particular device, supplying it with run time parameters, over the air (OTA) software upgrades, dynamic load balancing between constituent networks, being able to change and monitor device settings from a central "dashboard" or control panel among others. We also recognized that few of these goals, for example, OTA software upgrades would depend on the hardware design, in mainly the available flash after the firmware was flashed, the kind of error rates in packets while an image was transferred so on and so forth. OTA upgrade would also depend on a custom bootloader that had to be flashed along with the application code that could load the OTA binary

from external or internal memory onto the active code memory.

Others issues like choosing the best network probably was a bit more far removed from the actual underlying protocol. For example, the metric influencing this decision could be just the RSSI, the loading factor (i.e. how many devices are currently joined out of the maximum possible devices a particular coordinator can support). ZigBee uses 16-bit PAN IDs for identifying networks uniquely and for some other technology this identifier could be 32 bit. All of these differences however, should not impact the way a decision is taken to decide one particular network over another in a multi-star network environment.

One of the key features of our testbed is to allow us to deal with networking issues independent of the actual sensing/actuating payload. We created a testbed of more than 100 nodes each of which could be deployed within minutes on wall plugs, thereby avoiding the long installation times of previous sensors. Overall, we figured that the basic network characteristics and operations could be divided into a set of primitives and we could form our policies and decisions on top of this veneer of primitives without worrying much about the underlying WSN technology. The idea of using a layered approach to abstract out lower layers and present the upper layers with an uniform set of primitives is not new and has been an important design strategy for systems. By using these primitives in a WSN, we have been able to get similar kind of benefits that a layered architecture provides. These primitives are the core of this thesis and we take a detailed stock of the various approaches and the design decisions that we undertook while implementing these primitives. Those will follow after we discuss the hardware and the software of our testbed in the following two chapters.

Chapter 3

Hardware design

As mentioned in the previous section, we started off our initial WSN deployments with the TI CC2530/CC2531 SoC based ZigBee platform. This platform came with TI's implementation of the ZigBee protocol stack called 'ZStack'. Over time, we became comfortable with this software stack and its various quirks. For the work described in this thesis, we therefore decided to stick with TI's ZStack. However, we took a slightly different approach this time and instead of the CC2530/CC2531 SoC reference design, we chose the AIR A2530X24A modules, manufactured by Anaren, a reputable Wireless manufacturing company. There were a couple of reasons that motivated this migration. First of all, these modules, though consisting internally of the same CC2530 SoC, were FCC tested by Anaren. This was not the case with the previous CC2530/CC2531 reference design. Since RF design and testing is not our core area of expertise we decided to favor the Anaren modules over the plain CC2530/CC2531 reference design. Second of all, the Anaren AIR A2530X24A modules came with an optional Power Amplifier/Low Noise Amplifier front end for improved range. We found this enhanced the range of our devices by as much as 3 times when compared to the older CC2530/CC2531 reference design. Thirdly, the Anaren module, though having a different footprint, had an almost identical pin out with the CC2530 SoC. This did not change our older design much, apart from making changes in the PCB layout to accommodate the new footprint.

On the flip side however, the Anaren module with the PA/LNA front end

had a higher power rating. This was however offset by using it as a USB powered ZBC. We used the Anaren module without the PA/LNA front end as our battery powered counterpart. Our testbed therefore had an equal mix of devices based on both the PA/LNA front end as well as those without it.

The other disadvantage with these new Anaren modules was that they increased the cost of our BOM somewhat significantly. While the total manufacturing total of our older design was about 10$ in quantities of 1000 or more, these Anaren modules alone were priced at around 18.74$ (14.41$ for the non PA/LNA version)in quantities less than 500. Nevertheless, we needed a module that was FCC compliant with good RF characteristics upon which we could depend on for a much larger wide scale 'Enterprise' deployment. Also recall our lack of expertise in RF design. Keeping these things in mind we made a conscious decision to bear with the increased cost of using the Anaren modules. Our tests also showed that the extra cost was worth well in terms of increased range and operating characteristics.

The other minor inconvenience was that since the Anaren modules did not have a variant based on the TI CC2531, we had to rely on an external USB to serial converter chip. Having the CC2531 in our previous design had the advantage that it would enumerate itself as a USB Communication class device on a Windows or Linux host. We therefore, had to use the FT232Q chip from Future Technologies in order to make up for the lack of a CC2531 based Anaren module. While the use of FTDI did increase our BOM cost by about 4$, its excellent support on both Linux and Windows was a major advantage. Moreover, not all devices in our testbed were designed to have this FTDI chip so the additional 4$ was only for about a fourth of the devices in our testbed.

We also spent a couple of weeks porting our earlier software for these Anaren modules. Originally, these Anaren modules came with firmware which could be interfaced using an SPI interface. This architecture was derived from TI's ZAPP/ZNP architecture where the Anaren module works in slave mode and the core application logic is hosted over the ZAPP which could be any device supporting an SPI bus. Our porting effort proved that the Anaren module was a viable alternative to the CC2530/CC2531 based designs that we were using earlier.

In summary, our hardware consisted of four main components:

- ZigBee end devices which were the core of our testbed. These devices actively transmitted test data and whose status was closely monitored through the primitives that we talk about later. These end devices came in two variants, as discussed above:
  - With the PA/LNA enabled A2530**E**24A
  - Without the PA/LNA front end A2530**R**24A

  Both variants were used in our testbed in equal proportions, though in reality the latter variant is suited more for battery operated sensors (like our occupancy sensors). The former was also used in our testbed and it was very well suited for always on coordinators and mains powered energy meters.
- ZigBee coordinators which acted as aggregators of information from these testbed devices.
- Small form factor PCs that acted as base stations which communicated with the coordinators and relayed the collected data to our backend infrastructure.
- Our backend consisting of server class hardware running a MySQL server instance where we stored our database tables and dynamic information about the network.

We designed our PCBs using Altium[2] and we had to do 3 revisions until we were able to resolve all hardware related glitches from the final revision. Once the design was finalized we gave it to a local boardhouse to get the design fabbed and assembled.

The following subsections will discuss the hardware in more detail.

3.1 Testbed hardware - End devices and Coordinators

Our end device hardware differed from the predecessor in two main ways:

- It was based on the Anaren A2530X24A ZigBee module.
- It also contained an external SPI flash chip that we needed for OTA flash upgrades.

Since we had two variants of the Anaren ZigBee module that we intended to use within our testbed and we also had to make use of an USB-to-serial chip for connecting our base stations, we ended up designing a common PCB and then populating it in 3 different ways for serving our purposes. Below we show with 3 block diagrams how these 3 designs were different from each other and how we utilized each of them different for our testbed.

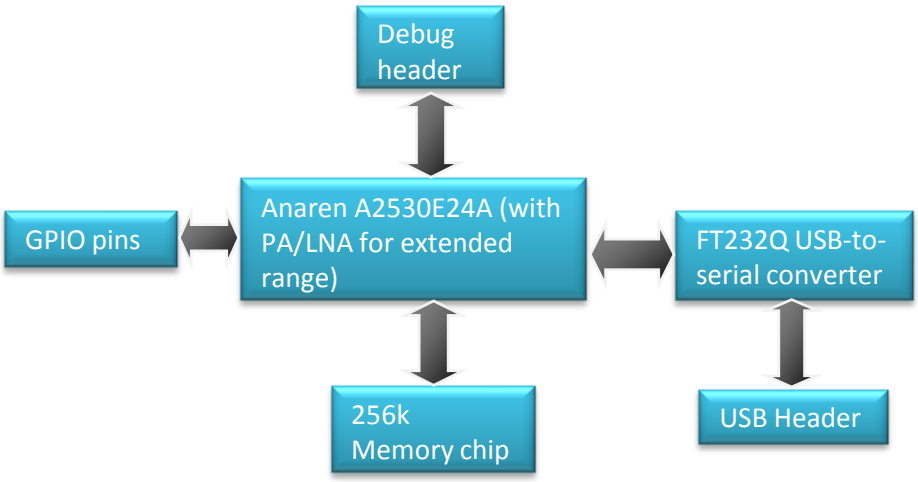


Figure 3.1: Type I design: With PA/LNA based Anaren module and FTDI chip

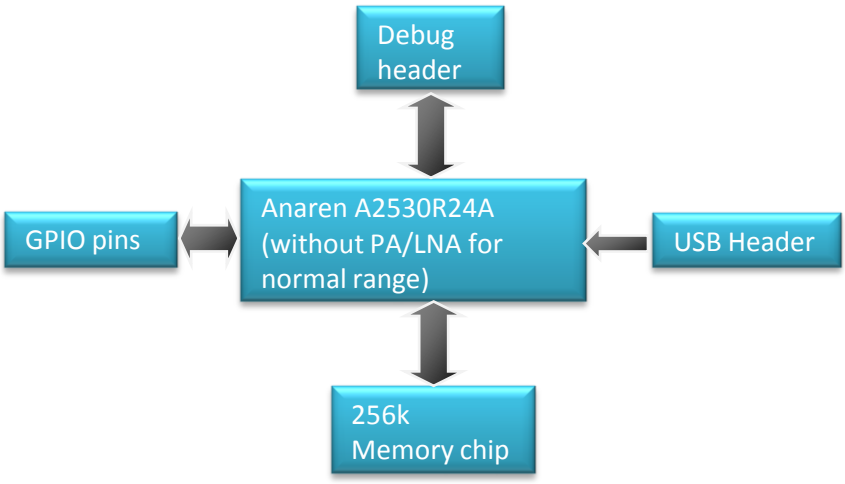


Figure 3.2: Type II design: Without PA/LNA based Anaren module and without the FTDI chip

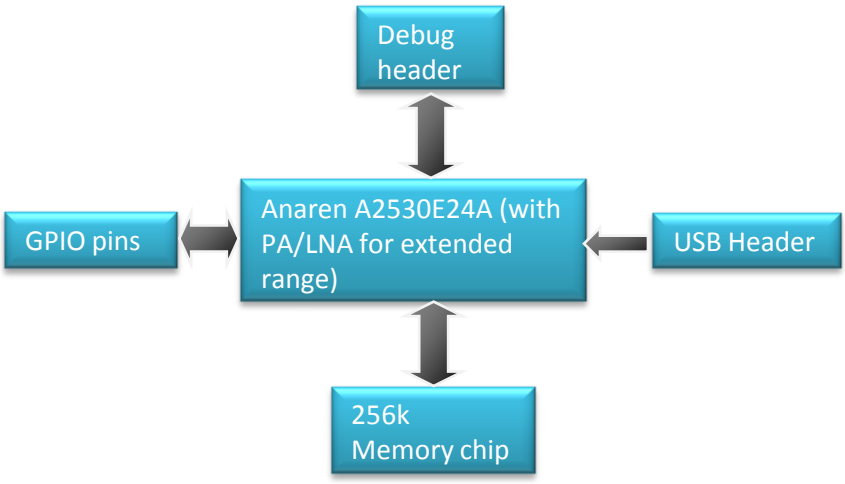


Figure 3.3: Type III design: With PA/LNA based Anaren module and without the FTDI chip

As you can see Type I design has a serial to USB converter along with an Anaren module that has the PA/LNA range extender front end. This design served very well as ZigBee Coordinators (ZBCs) which needed extended ranges and were powered via the USB of the base station PCs. Type II design lacks the

FTDI converter as well as the range extender front end. This design was meant to emulate actual devices which are battery powered (*e.g.* occupancy sensors) and which could not afford the power demands of a PA/LNA.

The third and final Type III design in addition to lacking an FTDI, had an Anaren module with the PA/LNA front end. This was meant to emulate actual devices which were mains powered (*e.g.* energy meters). Being mains powered, the higher power requirements of the PA/LNA was not important to this class of devices.

Our testbed therefore was a diverse mix of all the above types of devices. The table below shows each type of device along with the approximate number of each of them.

Table 3.1: Distribution of device types for our testbed

Device Type	**Number of devices**	**Remark**
Type I	˜25	Intended mainly as ZBCs
Type II	˜25	Intended to emulate battery powered ZEDs
Type III	˜50	Intended to emulate mains powered ZEDs

The ZBCs for our testbed were powered via the base station USB port. Both type II and type III ZEDs however were powered by using cheap and easily available wall plugs that have USB compatible socket at the other end. These wall plugs are sold from several online stores that are mainly intended to be used for charging smartphones and tablets. We repurposed them for our testbed. Having these wall plugs made deploying our testbed devices very easy and quick. All we needed was to find unused wall plugs to insert our testbed ZEDs.

Figure 3.4 show how the final testbed devices looked like after the entire process of designing, fabbing and assembly was complete. Closer inspection will reveal that type I has an FTDI chip, while the other two don't. Type I also lacks an onboard voltage regulator as it draws its power from the FTDI 3.3V output.

(a) Type I testbed device (b) Type I testbed device (c) Type III testbed device

Figure 3.4: Different types of testbed devices

Type II and Type III however need an external voltage regulator because it does not have an FTDI chip to draw power from.

Figure 3.5 shows a typical testbed device attached to a USB wallplug, ready to go into one of the AC mains wall plug points.

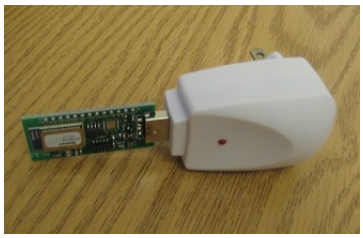


Figure 3.5: Typical testbed device with wall plug

3.2 Base Stations and backend server

This section discusses the rest of the testbed hardware components, i.e. the base stations that were responsible for interfacing with the ZBCs to collect WSN data and also the backend server that stored this data.

For base stations we tried using an Intel based PC with the mini-ATX form factor for the first time. This was a departure from our earlier choice of using plug computers. This move was motivated by the ease of managing a PC based architecture over the old ARM based plug computers. This was further evidenced by our visits to the various plugcomputer forums where we found a lot of developers and users struggling with solving seemingly simple problems like upgrading kernels or device drivers [7]. We realized that having a painless PC based solution for our

(a) Old ARM based BS (b) New PC based BS

Figure 3.6: Old ARM based Vs new PC based base station design

base stations would save us a lot of time which we could devote to solving our WSN problems that we initially set out to solve. Figure 3.6 show both types design for comparison.

For the server where we hosted our backend, we used a commodity server class machine with dual Intel Xeon cores and 16 GB of RAM. This machine was housed in the department server room to which we had access by virtue of being a part of SYNERGY Labs.

Chapter 4

Software Architecture and Design basics

This chapter deals with the overall software architecture and the system wide changes that we did to the stock ZStack implementation to come up with our primitive based approach. This chapter also deals with the architecture of base station scripts which were used to collate the WSN end device data and also to manage network specific parameters. The next chapter on the other hand, delves deeper into the specifics of these primitives. Thus, this chapter lays the groundwork necessary for understanding the primitives that we discuss in the next chapter.

Along the way, we also solved some miscellaneous issues that we found deserved attention because they impacted our overall design considerably. We discuss the solutions to these challenges in this chapter as well.

What follows now is a block diagram of where our software fits with respect to the overall software architecture of ZStack. Please note that for ease of understanding and reducing complexity, we have shown the ZDO, APS and NWK layers of ZigBee in the same level in this block diagram. However as per the ZigBee specs they exist in separate layers. The reader is referred to the ZigBee specs[10] for the exact layering semantics of the ZigBee networking protocol.

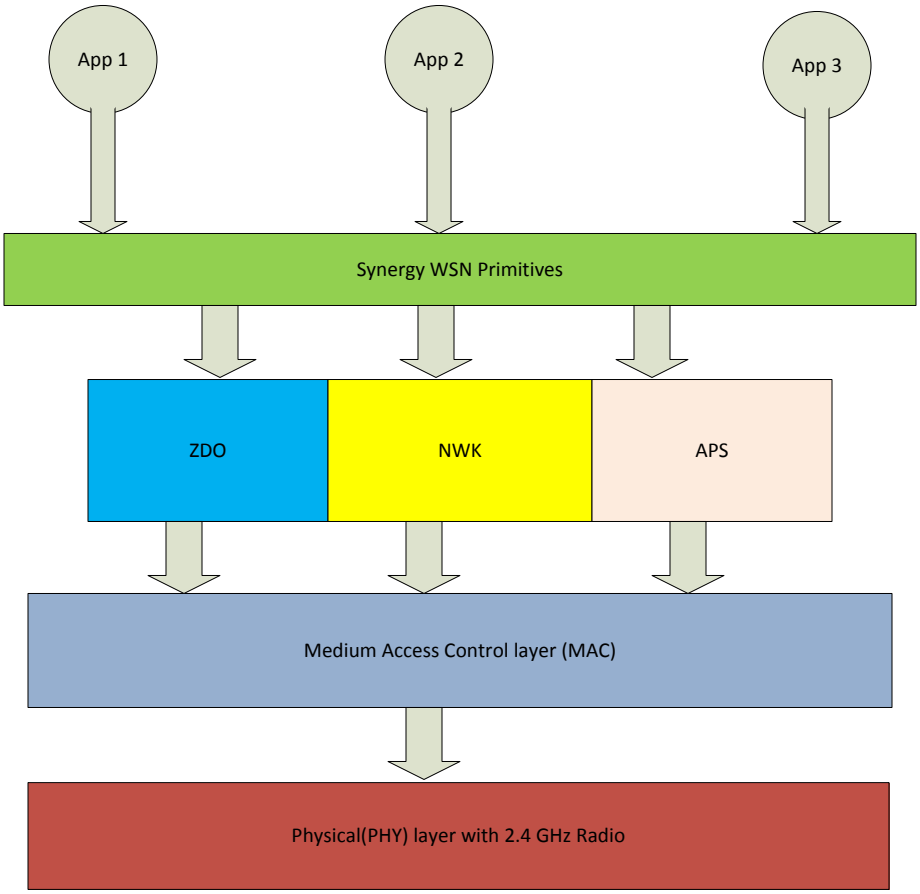


Figure 4.1: Simplified block diagram showing the relationship between our work and the ZigBee protocol

Our area of focus is indicated in green in this block diagram. Our primitives, which we have aptly named Synergy WSN primitives, sit between the applications and the lower layers of the ZigBee protocol stack. Some of the primitives are generic in nature while some others are specific to ZigBee. Regardless, the Synergy WSN Primitives provide a way for network engineers to abstract out the underlying layers to help them realize their goals such as commissioning, OTA upgrades and other key management tasks.

4.1 Monitor and Test (MT) API

Key to realizing Synergy WSN primitives was a subsystem called the TI Monitor and Test API. This subsystem came with TI's ZStack implementation. It was a convenient way to access several of ZStack's core APIs via the serial port. TI divided its MT subsystem into smaller modules and a dedicated operating system task was responsible for all MT related processing on the device. Each module including the entire MT subsystem could be independently turned ON and OFF via conditional compile flags. Some of the modules include system commands, utilities, ZDO functionality and so on.

MT APIs worked by exposing several key system and ZStack APIs through prearranged serial packets. By sending these prearranged packets to devices running ZStack one could have these devices execute certain APIs and then return their results. This formed our primary model of exposing Synergy WSN primitives to the outside world as well as to applications sitting on top. TI's implementation was also dependent on a Windows application front end that ran on a PC. Our changes to the MT API removes this limitation as our base station scripts were written to be MT-aware.

The sequence diagram in Figure 4.2 shows how by using a correctly formed serial packet (either via TI' PC tool or our base station scripts), one could control the behavior of any device running ZStack, compiled with the MT subsystem.

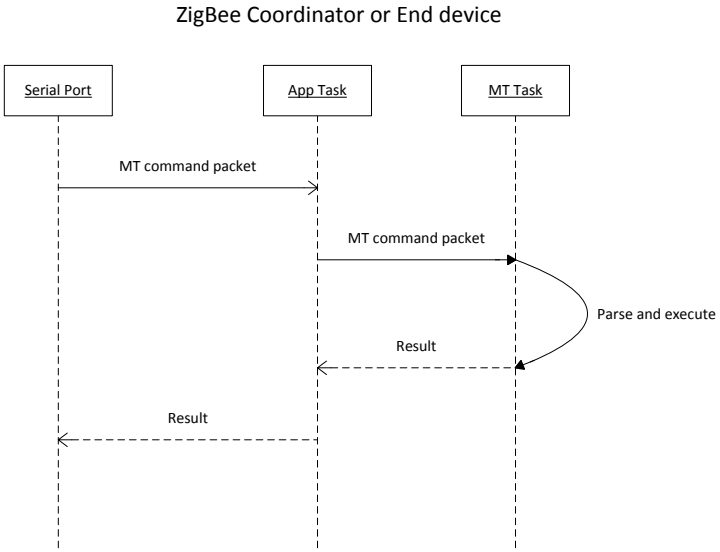


Figure 4.2: Message sequence chart showing typical in-device MT command execution on ZStack based devices

However, TI's MT implementation had three major limitations.

- It had to have exclusive control over the serial port. This conflicted with the primary functionality on our ZBC which was to access the serial port to send the network information to the base station scripts via the serial port.
- It accepted packets only through the serial port. It was not possible to send an MT command OTA and get its result back OTA, across multiple devices.
- It did not have a extensible design allowing addition of new primitives if desired.

We overcame all of these limitations by carefully rewriting parts of TI's original MT implementation. Doing so, we were not only able to leverage TI's rich set of pre-defined MT primitives, but also add our own primitives to the list as and when we felt necessary.

For the first problem, we encapsulated TI's MT packets within our legacy packet structure. Then we expanded the list of known packet types by adding "MT packet" as one of the possible serial packet types. A pre-existing field within the legacy packet structure indicated what type of packet was being transported.

All serial packets would be received first by application unlike TI's implementation where all packets were received by the MT task first. Thus, the MT task within ZStack would no longer be in control of the serial port. The application would disambiguate between different types of packets and in case of an MT packet it would send it to the MT task for parsing and execution. Once execution was complete, the MT task would not send the results back on its own, but it would first send the results to the application task. The application task would then send out the results via the serial port, with the correct message type flag set. This way sensor event packets could share the same serial transport as MT packets.

For the second problem, all we did was send the same packet that we used above, OTA to the target device. The target device in this case could either be the entire network via a network wide broadcast or it could be a single device via a unicast message. The receiving device application task would then figure out if it was an MT packet and if so, it would send it to its MT task for parsing and execution. The results would then be sent to the requesting device OTA. For the user requesting command execution via the serial port, it would not make any difference (except apart from a tiny speedup) whether the results were obtained from the device connected physically via the serial port or if they were obtained from a remote device OTA.

For the third problem, we ended up creating our own subsystem of MT commands, which we termed 'User Defined' MT commands. These were new primitives that were not present by default in TI's implementation. TI categorized all MT commands into different categories depending on their functionality. But rather than pollute the pre-existing TI categories with our user defined primitives, we decided to create our own category that could exist in parallel with TI's primitives. We called this new category UD or user defined MT commands (as opposed to system defined preexisting MT commands).

All of the above changes impacted the handling of all MT commands at a fundamental level because now an MT command could be just over the serial link or even OTA. Moreover, it could be independently one of the preexisting commands or an user defined command. The following steps describes the final algorithm for

handling all possible combinations of MT commands:

1. Receive potential MT commands via the serial port or OTA and after determining the message type, send a message to the MT task with the received MT packet as the payload.

2. The MT task determines which subsystem the command belongs to. In case it is a system MT command, it handles it within the MT task. If on the other hand it is a UD MT command, it trampolines it back to the application.

3. The application receives the trampolined MT packet which did not fall under the jurisdiction of the system MT commands. Thus, the application then parses the UD MT command and executes the necessary functionality.

4. The application then returns the results of the MT command either via the serial or OTA as the case may be.

A question the reader might have at this point is about the necessity of trampolining the MT packet at step 2 above. The UD MT commands could also be handled within the MT task without trampolining it to the application task. However, UD MT commands were mostly in the context of the application task and needed data structures that were specific to the application task. Thus, it made more sense to handle all UD MT commands in the context of the application task rather than have the MT task shadow the application task context variables.

The message sequence chart in figure 4.3 describes this handling of UD MT commands. But notice how the MT command packet trampolines from the MT task to the application task but does not cross device boundaries.

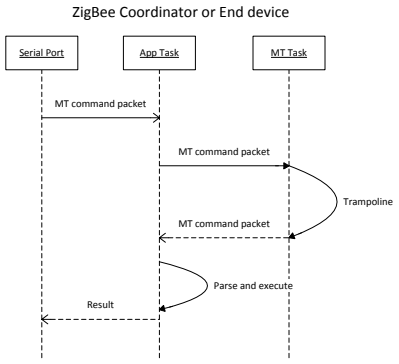


Figure 4.3: In-device UD MT commands execution

Contrast this with the message sequence chart in figure 4.4 which describes the exact same scenario as above, but with an UD MT packet crossing device boundaries in addition to task boundaries. This enables the command to be parsed and executed across the network. This is the basis of all Synergy WSN primitives which need remote execution.

Lastly, we are left with the case which is similar to the preceding scenario, but involves a system MT command executed across device boundaries, rather than a UD MT command executed across device boundaries. This is depicted in figure 4.5.

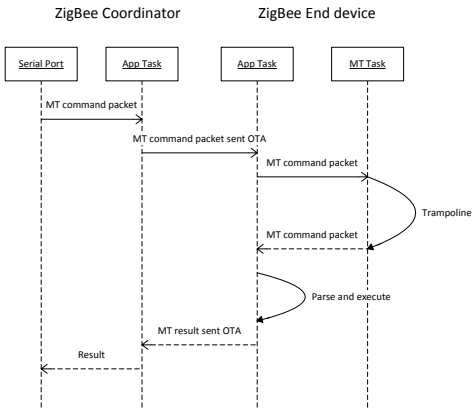


Figure 4.4: Across-device OTA UD MT commands execution

There is also a third type of MT command execution which is hybrid of serial and OTA executions. This type of execution is needed when the primitive is

complicated enough to warrant multiple subprimitives in order to achieve its functionality. In such a situation, a single MT command will give rise to several OTA MT commands being exchanged between devices. Finally, when all those subprimitives have been executed, the end result would be sent via a serial confirmation. This is depicted in the message sequence chart in figure 4.6.

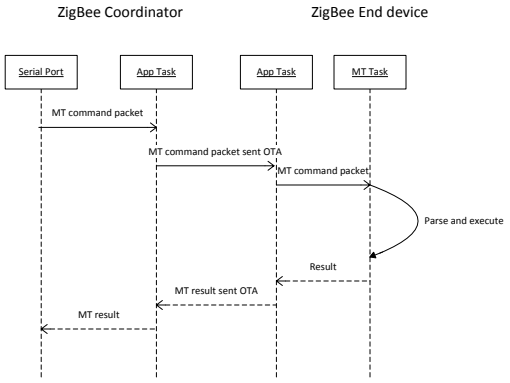


Figure 4.5: Serial to OTA System MT commands execution

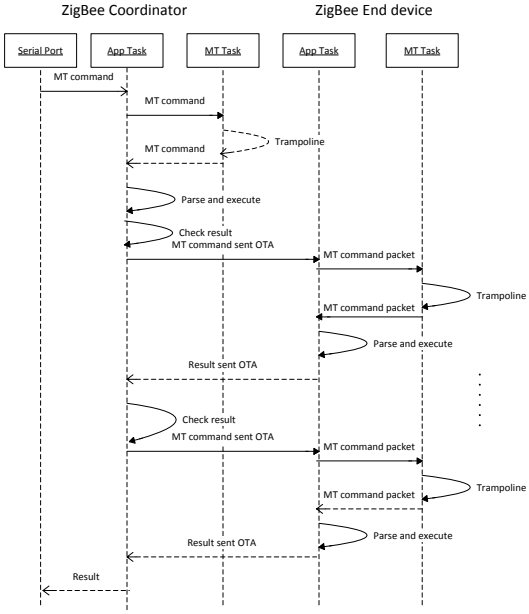


Figure 4.6: Serial to OTA Hybrid UD MT commands execution

4.2 Base Station scripts organization

The base station scripts were written in Python over a popular Linux distribution, Ubuntu 12.04 Precise Pangolin. These scripts gave a command line interface to the attached ZBC. The scripts were organized into multiple objects communicating via queues with each other. We used the python binding to the well known udev library in order to abstract out the script from the lower level USB device names *e.g.* */dev/ttyUSB0* or */dev/ttUSB1*. The command line interface was named "BSShell", *i.e.* a shell type of interface where the user could type out commands to the connected ZBC interactively. These commands would then get translated into corresponding primitives which would then be sent via the USB interface to the ZBC. The ZBC would execute the primitives and then return back the result back through the USB to the shell. Figure 4.7 shows its overall architecture.

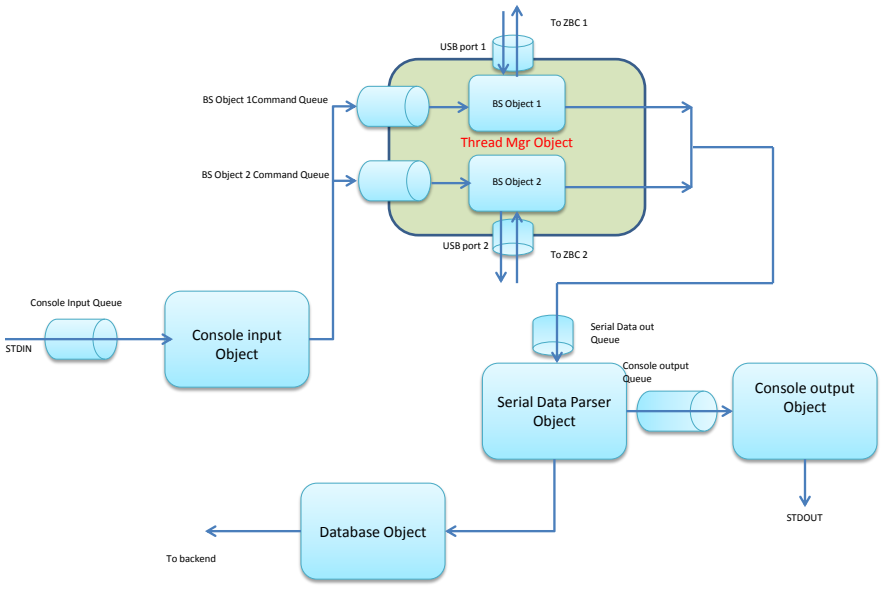


Figure 4.7: BSShell Software Architecture

The BSShell allowed us to quickly test new Synergy WSN primitives as well as existing MT primitives. For existing MT primitives we just had to write the corresponding Python front end and update the MT code within TI's implementation to work OTA as well as through serial. For UD MT primitives, we not only

wrote the primitives from scratch but also added the corresponding Python front end to the BSShell source code.

Figure 4.8 shows a screenshot from a typical interactive session.

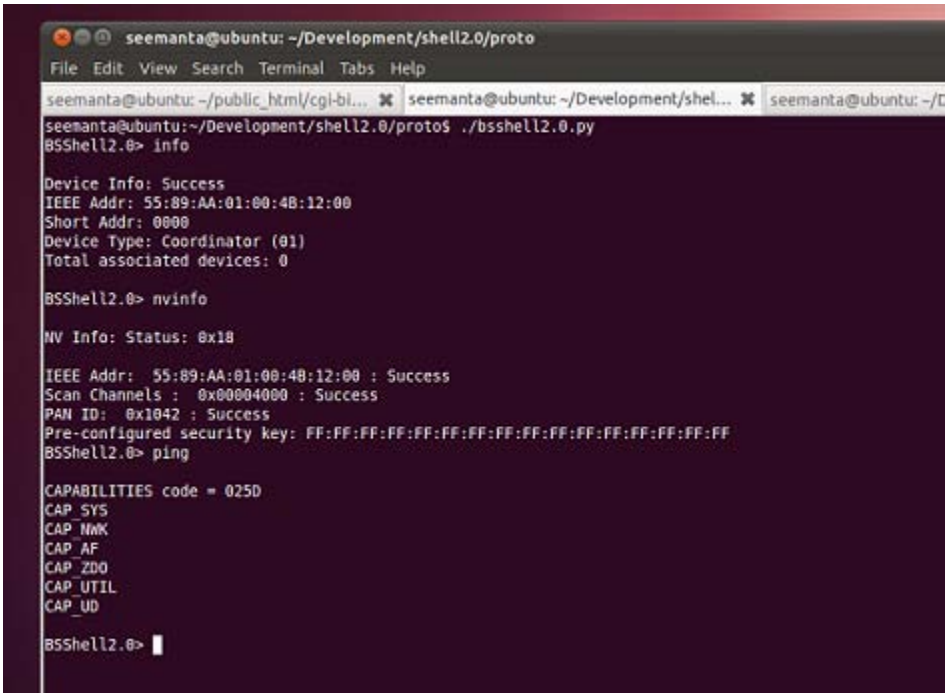


Figure 4.8: BSShell showing invocation of Synergy WSN primitives via the serial port

4.3 Miscellaneous software issues

This section discusses some miscellaneous issues that we encountered while developing Synergy WSN primitives. These issues are orthogonal to the MT architecture described above, but included in this chapter for the sake of completeness.

4.3.1 Sleep VS Active Power of Anaren based devices

ZigBee end devices are intended to be battery powered and therefore conserve power by waking up periodically, checking for any messages and then go back to sleep. This behavior is ingrained in the protocol itself and is controlled by

various runtime parameters which determine the rate of this polling to check for messages.

This behavior, however is defined only after an active connection is established by an end device with its parent coordinator/router. We discovered that in the absence of a coordinator, the protocol does not specify the behavior of the end device as to whether it should go to sleep or keep waking up periodically to check for parents in its vicinity. Without a proper strategy to handle the corner case when no ZigBee coordinator is available the end devices might end up draining their batteries quickly.

We therefore devised our own algorithm to offset this scenario. However, we realized that even with this new algorithm, we were not able to drive the power consumption during sleep periods. After visiting TI's developer forums, we found that unless the application indicates to the stack runtime that is a battery powered device, none of the power saving features would work. Normally, this indication is given to the stack runtime after the device has associated with a ZBC. But in this case, since we were scanning and trying to find one, the indication was never given during the scan stage. We therefore fixed this by calling the indication API right at task initialization even before we start scanning. This fix brought down the sleep current to about 380 uW.

The flowchart in figure 4.9 depicts this algorithm. As you can see, the device checks after a certain time interval if it has associated with a ZBC. If not, it goes to sleep for a second fixed time interval. After this time interval expires, it would wake up and stay awake for another interval of time. After this, it would check again if the device was able to associate. If yes, it would simply stop sleeping and if not, it would continue this process.

Typical interval values for scanning and sleeping chosen in our implementation were 15 seconds and 285 seconds, respectively, giving a total cycle time of 300 seconds, *i.e.* 5 minutes. All these time intervals are NV configurable and can be customized without re-compiling the ZED code.

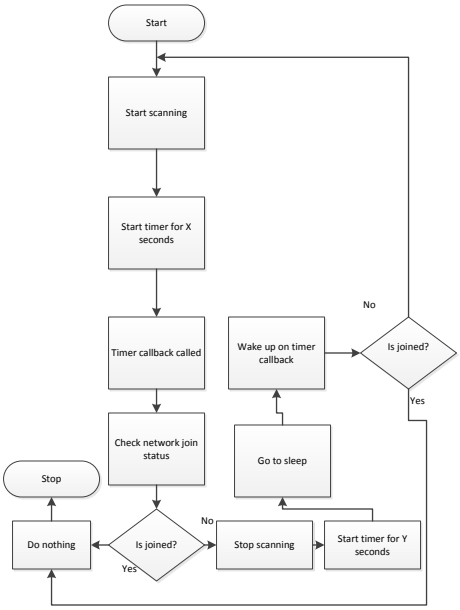


Figure 4.9: Sleep algorithm when no ZBCs are around. Intervals X and Y are NV configurable.

Chapter 5

Synergy WSN primitives

In this chapter we discuss the Synergy WSN primitives that we created in detail. From the block diagram in Figure 4.1, these primitives are represented by the green block in the last chapter.

Recall that we leveraged the existing MT infrastructure for our work. In doing so, we could keep approximately 170 MT primitives that were part of TI's implementation. We added our own user defined primitives on top of that and also did the necessary changes to make all primitives support OTA in addition to serial. This enhanced, superset of primitives containing our own subset of primitives was christened "Synergy WSN Primitives". Through the use of these primitives (for usage model please refer to Figures 4.2 through 4.6), we have implemented some key features of network management. The next chapter discusses these management features in detail, while the focus of this chapter are the primitives themselves.

In the subsequent sections, we begin by familiarizing the reader with the structure of an MT packet. We then list our own primitives and describe them briefly. We do not wish to duplicate the listing of MT primitives supported by TI because that is very well documented by TI in their documentation that comes with ZStack [8].

5.1 MT packet structure

Each MT packet is indicated by a special magic SOF (start of frame) byte. This is arbitrarily set to 0xFE. This was followed by the rest of the packet. The entire packet was then suffixed with an FCS or frame control sequence. The FCS was calculated by XOR'ing all the packet bytes, excluding the SOF byte. The FCS byte allowed the MT implementation to detect errors but there was no support to correct those errors. Our base station scripts would time out in case there was any checksum error or when the device under test failed to respond to our commands. The figure below shows this general structure.

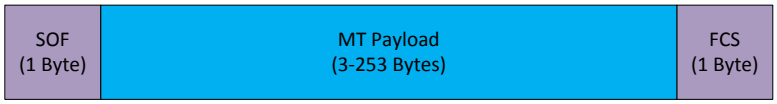


Figure 5.1: Overall structure of an MT packet

The MT Payload field itself could be divided into 4 further fields:

1. **Length**: This field indicated the length of the data field, excluding the SOF byte and the FCS. In cases of commands without any data, this would be set to 0.

2. **Command 0**: This field contained the MT subsystem to which the command belonged.

3. **Command 1**: This field contained the actual command that was to be executed.

4. **Data**: This was an optional field containing the request or response data.

The next figure shows the complete structure of the MT packet with all the above details.

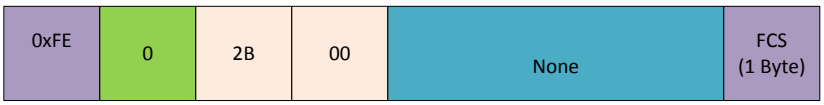


Figure 5.2: Detailed structure of an MT packet

The complete list of MT subsystems are as shown in the table below:

Table 5.1: MT subsystems and their values

Subsystem	**Subsystem Value**
Reserved	0x00
SYS interface	0x01
MAC interface	0x02
NWK interface	0x03
AF interface	0x04
ZDO interface	0x05
SAPI interface	0x06
UTIL interface	0x07
DEBUG interface	0x08
APP interface	0x09
OTA interface	0x0A
UD interface	0x0B

Out of these, the OTA interface was TI's own interface that was used for their OTA software upgrade solution which we did not use in our own implementation of OTA software upgrade. The UD interface was added by us to create a separate category for all higher level MT primitives.

5.2 Synergy WSN MT primitives

This section depicts all the primitives that we added under the UD category of MT commands. These together with the preexisting MT primitives form the basis of Synergy WSN primitives.

5.2.1 Get Link Quality (LQI)

This primitive can be used to get the average link quality information from a ZBC or from its connected ZEDs. The packet structure of this primitive is shown in Figure 5.3.

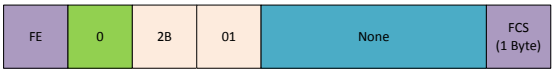
(a) Get LQI MT request

(b) Get LQI MT response

Figure 5.3: Packet structures for get LQI MT command

5.2.2 Get RSSI (Received Signal Strength Indicator)

This primitive can be used to get the average RSSI of a ZBC or a ZED. The packet structure of this primitive is shown in Figure 5.4.



(a) Get RSSI MT request

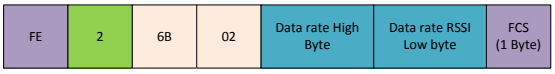
(b) Get RSSI MT response

Figure 5.4: Packet structures for get RSSI MT command

5.2.3 Get Data Rate

This primitive can be used to get the average data rate of a ZBC or a ZED. The packet structure of this primitive is shown in Figure 5.5.

(a) Get data rate primitive request



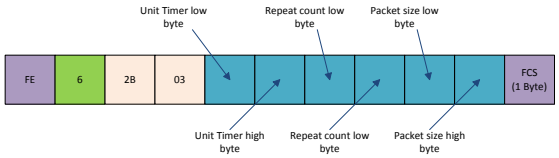
(b) Get data rate primitive response

Figure 5.5: Packet structures for get data rate primitive

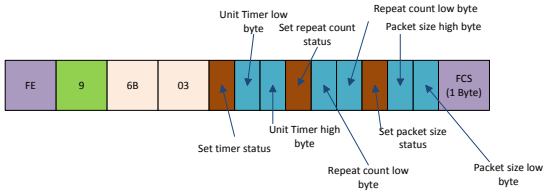
5.2.4 Set transmission parameters

This primitive can be used to set the transmission parameters of the device. This is typically used for testbed devices. Using this primitive one can change the transmission rate and the packet size of each packet. The device needs to be reset in order for the changes to take effect. The field definitions are shown in Figure 5.6 and are as follows:

- **Unit Timer**: Interval in milliseconds between each loop count
- **Repeat count**: Number of loops between each transmission. Product of unit timer and repeat count gives exact transmission interval. *i.e.* unit timer of 1000 and repeat count of 2 will give a delay of 2 seconds between each packet.
- **Packet Size**: Size of packet in each transmission.
- **Set unit timer status**: Status of setting the unit timer value in NV.
- **Set repeat count status**: Status of setting the repeat count in NV.
- **Set packet size status**: Status of setting the packet size in NV.



(a) Set transmission parameters primitive request



(b) Set transmission parameters primitive response

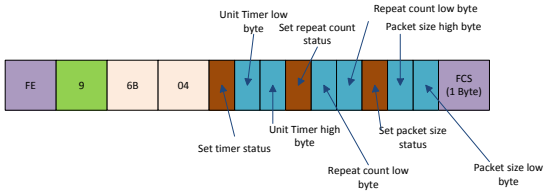
Figure 5.6: Packet structures for set transmission primitive

5.2.5 Get transmission parameters

This primitive can be used to get the current values of the transmission parameters *i.e.* unit Timer, repeat count and packet size. This is typically used for testbed devices. The field definitions are shown in Figure 5.7 and are as follows:

- **Unit Timer**: Interval in milliseconds between each loop count.
- **Repeat count**: Number of loops between each transmission. Product of unit timer and repeat count gives exact transmission interval. *i.e.* unit timer of 1000 and repeat count of 2 will give a delay of 2 seconds between each packet.
- **Packet Size**: Size of packet in each transmission.
- **Get unit timer status**: Status of getting the unit timer value in NV.
- **Get repeat count status**: Status of getting the repeat count in NV.
- **Get packet size status**: Status of getting the packet size in NV.

(a) Get transmission parameters primitive request



(b) Get transmission parameters primitive response

Figure 5.7: Packet structures for get transmission parameters primitive

5.2.6 Scan for active end devices

This primitive can be used to scan for active end devices in the vicinity of a ZBC. This returns only active, *i.e.* transmitting devices. Non-commissioned or device commissioned devices will not respond to this primitive. The ZBC should wait for a fixed time out to collect all responses from neighboring end devices before collating the results and sending it to the client. The fields of this primitive are shown in Figure 5.8 and are as follows:

- **Short Address high byte**: High byte of the short address of the device responding to this primitive.
- **Short Address high byte**: Low byte of the short address of the device responding to this primitive.

(a) Active scan primitive request

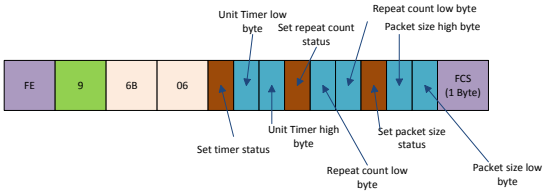
(b) Active scan primitive response

Figure 5.8: Packet structure for active scan primitive

5.2.7 Load default transmission parameters

Using this primitive, the device can be instructed to restore its default transmission parameters. Default transmission parameters are determined at compile time and cannot be changed via Synergy WSN primitives. Figure 5.9 shows the field structure of this primitive. The field definitions of this primitive is exactly same as Subsection 5.2.4. The device needs to be reset in order for the changes to take effect.

(a) Load default transmission parameters primitive request



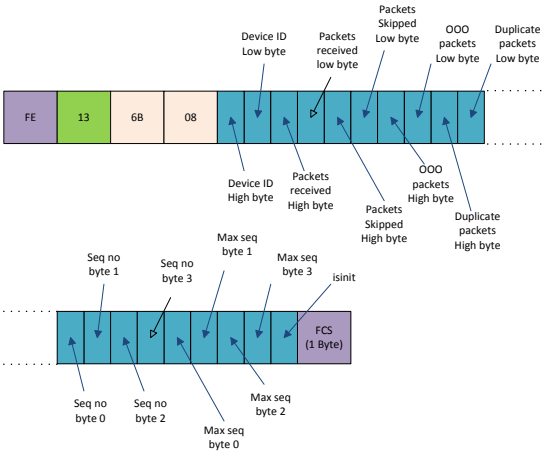
(b) Load default transmission parameters primitive response

Figure 5.9: Packet structure for Load default transmission parameters primitive

5.2.8 Get packet transmission statistics

This primitive gets the supported only on the ZBC and it can be used to get the packet transmission statistics from the ZBC.

(a) Get packet statistics primitive request



(b) Get packet statistics primitive response

Figure 5.10: Packet structure for get packet statistics primitive

This primitive is supported only on the ZBC, and is not supported OTA. Figure 5.10 shows its field structure and the field definitions of this primitive are as follows:

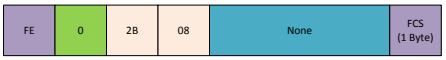
- **Device id high byte**: The high byte of the device id.
- **Device id low byte**: The low byte of the device id.
- **Packets received high byte**: The high byte of packet received.
- **Packets received low byte**: The low byte of packets received.
- **Packets skipped high byte**: The high byte of skipped packets.
- **Packets skipped low byte**: The low byte of skipped packets.
- **OOO packets high byte**: The high byte of out of order packets.
- **OOO packets low byte**: The low byte of out of order packets.
- **Duplicate packets high byte**: The high byte of duplicate packets.
- **Duplicate packets low byte**: The low byte of duplicate packets.
- **Sequence number byte 0**: Byte 0 of sequence number.
- **Sequence number byte 1**: Byte 1 of sequence number.
- **Sequence number byte 2**: Byte 2 of sequence number.
- **Sequence number byte 3**: Byte 3 of sequence number.
- **Max sequence number byte 0**: Byte 0 of maximum sequence number encountered so far.
- **Max sequence number byte 1**: Byte 1 of maximum sequence number encountered so far.
- **Max sequence number byte 2**: Byte 2 of maximum sequence number encountered so far.

- **Max sequence number byte 3**: Byte 3 of maximum sequence number encountered so far.
- **Is initialized**: Indicates the start of sequence numbers. Is False for the first packet, subsequent packets set it to True.

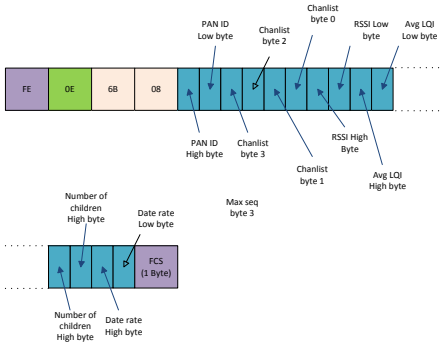
The client is responsible for using the data returned from this primitive to calculate packet statistics on the coordinator.

5.2.9 Get coordinator information

This primitive is used by an end device during phase II of the commissioning process,*i.e.* during the autonomous network acquisition phase of the commissioning process. This primitive does not have a Python binding yet because it is used transparently during phase II. However, a Python binding for this could be written if desired. When a ZBC receives this primitive, it responds with information necessary for the end device to decide if it wants to join its PAN. Figure 5.11 shows the field structure of this primitive.



(a) Get coordinator information primitive request



(b) Get coordinator information primitive response

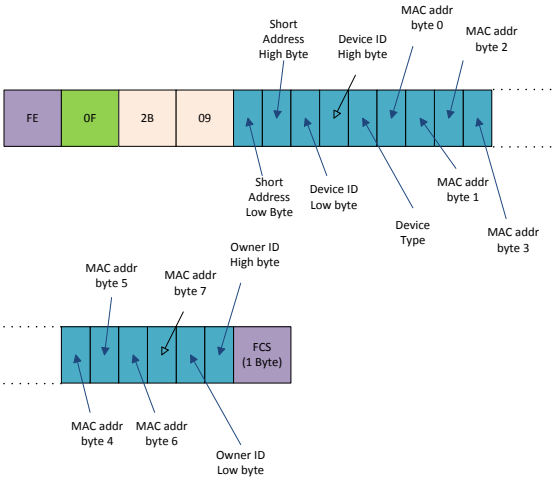
Figure 5.11: Packet structure for Get coordinator information primitive

5.2.10 Scan for non-commissioned devices

This primitive is used by the C-App during device commissioning. Upon receipt, only non-commissioned devices respond to this primitive. Figure 5.12

shows the field structure for this primitive.

(a) Non-commissioned devices scan primitive request

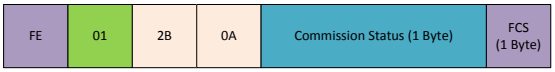


(b) Non-commissioned devices scan primitive response

Figure 5.12: Packet structure for non-commissioned devices scan primitive

5.2.11 Set commission status of device

This primitive can be used to change the commission status of a device. This needs a device reboot to take effect. Figure 5.13 shows the field structure for this primitive.



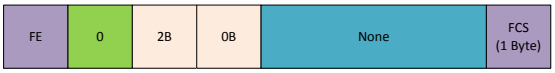
(a) Set device commission status primitive request

(b) Set device commission status primitive response

Figure 5.13: Packet structure for set device commission status primitive response

5.2.12 Get commission status of device

This primitive can be used to read the commission status of a device. Figure 5.14 shows the field structure for this primitive.



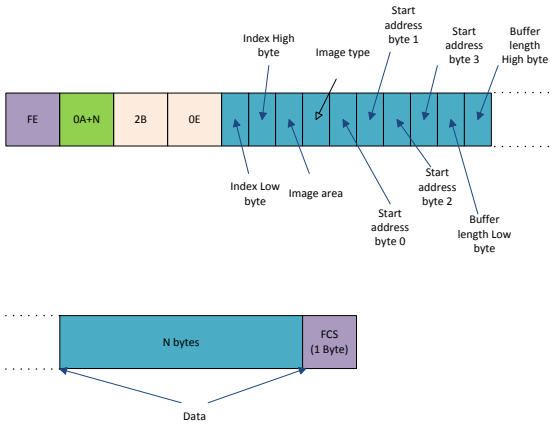
(a) Get device commission status primitive request

(b) Get device commission status primitive response

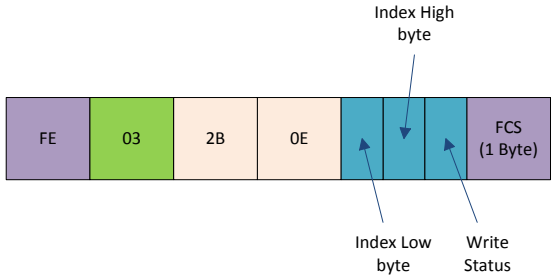
Figure 5.14: Packet structure for get device commission status

5.2.13 Write external NV memory

This primitive can be used to write an arbitrary buffer of data at an arbitrary location in the external NV memory of the device. **The buffer being written must be aligned at a 256-byte boundary for this primitive to work properly**. Figure 5.15 shows the field structure for this primitive.



(a) External NV memory write primitive request

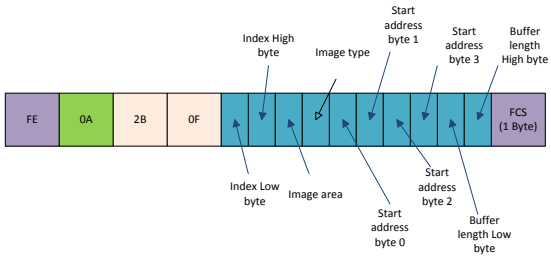


(b) External NV memory write primitive response

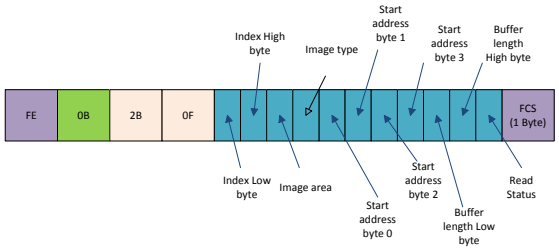
Figure 5.15: Packet structure for writing to external NV memory

5.2.14 Read external NV memory

This primitive can be used to read an arbitrary buffer from an arbitrary location from the external NV memory of the device. **The buffer being read must be aligned at a 256-byte boundary for this primitive to work correctly**. Figure 5.16 shows the field structure for this primitive.



(a) External NV memory read primitive request



(b) External NV memory read primitive response

Figure 5.16: Packet structure for reading external NV memory

5.2.15 Invalidate production code

This primitive invalidates the production code by writing zeros to the image checksum bytes. Invalidating the production code does not take effect until the next reboot of the device. Upon reboot, the bootloader will load the stored binary image from the external memory onto the internal NV flash of the Anaren module. If the device does not have the bootloader and this primitive is sent to the device, the device will return failure. If the bootloader is enabled in the device, the primitive will always return success. Figure 5.17 shows the field structure for this primitive.

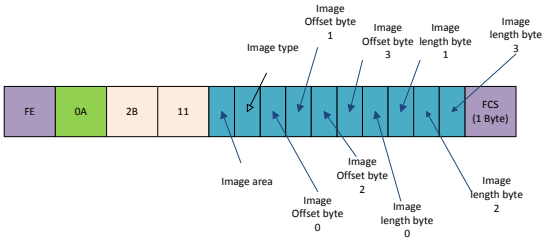
(a) Invalidate production code primitive request

(b) Invalidate production code primitive response

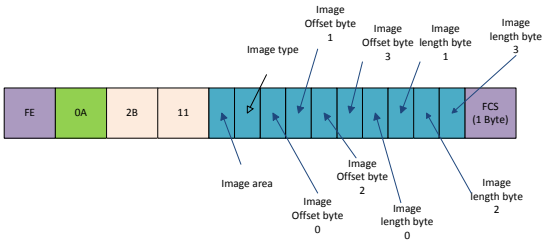
Figure 5.17: Packet structure for invalidating production code

5.2.16 Send external flash image

This primitive can be used to send a chunk of data from the external memory of the source device. It can be thereby used to send the entire memory contents of the external memory chip. After completion, the device sends a response back. Figure 5.18 shows the field structure for this primitive.



(a) Send external flash primitive request

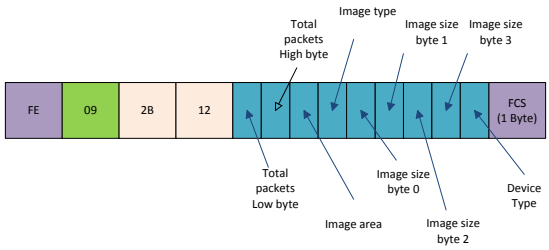


(b) Send external flash primitive response

Figure 5.18: Packet structure for sending external flash image

5.2.17 Send external NV memory information

This primitive is used as a sentinel primitive prior to beginning a full external NV memory transfer. This lets the end device know details about the impending transfer. This is sent via broadcast and therefore does not have any response message. Figure 5.19 shows the field structure for this primitive.

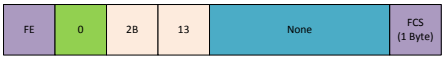


(a) Send external NV memory information primitive request

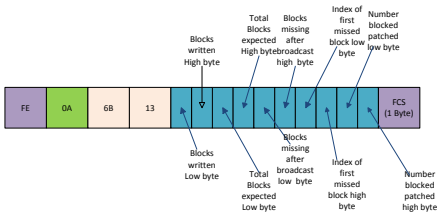
Figure 5.19: Packet structure for sending external NV memory information

5.2.18 Get external NV memory transfer statistics

This primitive is useful for knowing the external NV memory image transfer statistics. It can be used for debugging over the air memory transfers by knowing how many packets were lost and how many were actually written onto the external memory. Figure 5.20 shows the field structure for this primitive.



(a) Get external memory transfer statistics request

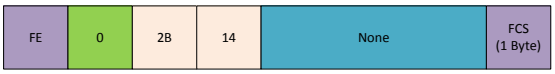


(b) Get external memory transfer statistics primitive response

Figure 5.20: Packet structure for get external NV memory transfer statistics

5.2.19 End external NV memory transfer

This primitive is used to indicate to the end devices that the ZBC has finished transferring all the image blocks. At this point the end devices can check if they are missing any blocks and start patching up the invalid regions in their received image. This primitive is sent via broadcast and there does not have any corresponding response. Figure 5.21 shows the field structure for this primitive.

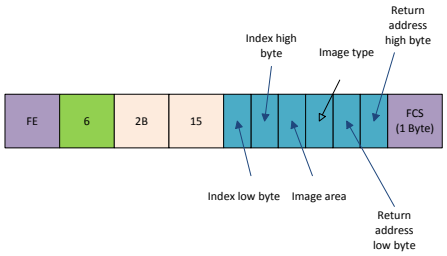


(a) End external NV memory transfer primitive request

Figure 5.21: Packet structure for ending external memory transfer

5.2.20 Request external NV memory block

This primitive is used by an end device to request for a missing memory block after the external NV memory transfer procedure has ended. Figure 5.22 shows the field structure for this primitive.



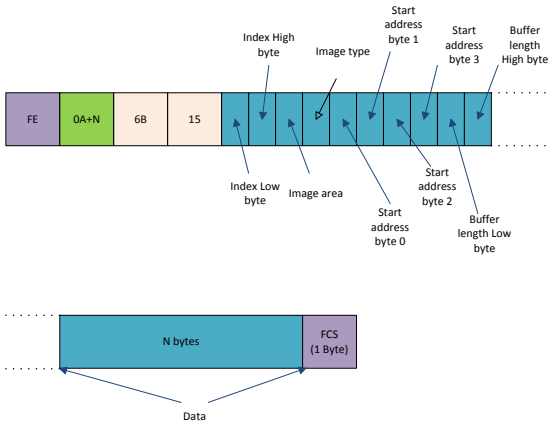
(a) Request external NV memory primitive request

Figure 5.22: Packet structure for requesting external NV memory block

5.2.21 Handle External NV memory block

This primitive is sent by the coordinator as a response to the 'Request external NV memory block' primitive. Upon receipt, the end device writes the

data into the external memory chip. Figure 5.23 shows the field structure for this primitive.



(a) Handle External NV memory block primitive

Figure 5.23: Packet structure for Handling external NV memory block

5.2.22 Send test command

This primitive when sent to the device makes the device perform some external visible action. This is useful for identification during device commissioning via the C-App. The default implementation lights an LED, but it can be modified to perform any task, even send another primitive in response to this command. The response structure contains a byte indicating the status of the test operation. Figure 5.24 shows the field structure for this primitive.

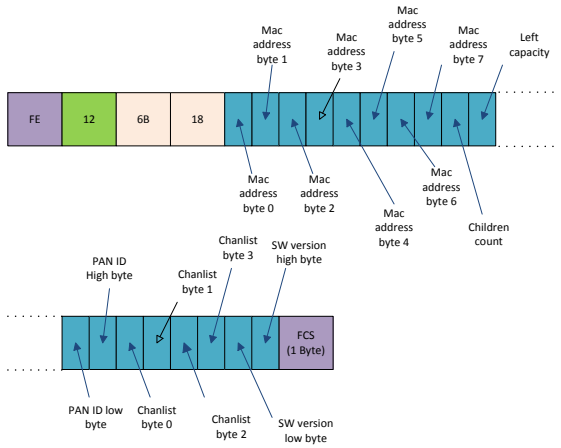
(a) Send test command primitive request

(b) Send test command primitive response

Figure 5.24: Packet structure for sending test command primitive

5.2.23 Send ZigBee coordinator information

This primitive is used by the ZigBee coordinator to send status information about itself. The base station scripts are responsible for parsing this primitive and updating the information in the back end. There is no associated request associated with this primitive. It is suggested that the ZigBee coordinator update the base station script with this primitive at a regular interval. Figure 5.25 shows the field structure for this primitive.

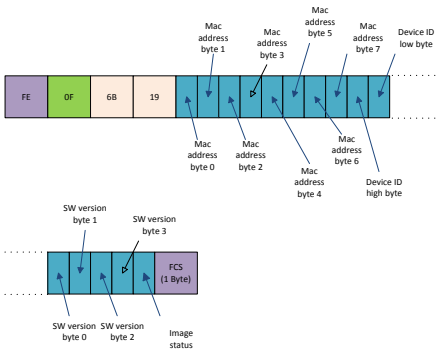


(a) Send ZigBee coordinator information primitive

Figure 5.25: Packet structure for sending ZigBee coordinator information

5.2.24 End device announce

This primitive is sent by an end device at least once in its lifetime. This is necessary to populate the back end with static information about the end device. It is suggested that an end device send this primitive once after initialization and then after any change of status. There is no associated request for this primitive. Figure 5.26 shows the field structure for this primitive.



(a) End device announce primitive request

Figure 5.26: Packet structure for end device announce

5.2.25 Disseminate encryption key

This primitive is used by a coordinator to disseminate its encryption key to all its connected end devices. Figure 5.27 shows the field structure for this primitive. This primitive is sent via a broadcast and therefore does not have any associated response structure.

(a) Disseminate encryption key primitive request

Figure 5.27: Packet structure for disseminating encryption key

5.2.26 Send encryption key

This is an internal primitive that is sent from a ZigBee Coordinator to an end device as a result of getting the 'Disseminate encryption key' primitive. This primitive is not currently made available via Python bindings. Figure 5.28 shows the field structure for this primitive.

(a) Send encryption key primitive request

Figure 5.28: Packet structure for sending encryption key

5.2.27 Clean NV on next reboot

This primitive makes the device to delete all its NV settings on the next reboot. It therefore requires a reboot to take effect. This primitive can be used to revert a device to its factory settings. Figure 5.29 shows the field structure for this primitive.

(a) Clean NV on next reboot primitive request

(b) Clean NV on next reboot primitive primitive response

Figure 5.29: Packet structure for clean NV on next reboot

5.3 Conclusion

We have described the list of primitives implemented in order to create the complete set for Synergy WSN primitives. Some of these primitives are not shown here because they are implemented as part of TI's stock implementation. They have, however been retro-fitted to conform to our idea of being able to send and receive MT commands OTA. This has not been done for all of the 160 odd TI primitives, but only for the primitives we felt were most needed for our work. A step by step method has been documented which details how a pure TI primitive can be retrofitted to behave as a Synergy WSN primitive by augmenting it so that

it can work over serial and OTA too.

The next chapter deals with our management features that we built using these primitives. We call them 'features' because they make use of multiple synergy WSN primitives in order to achieve their end goals.

Chapter 6

Synergy WSN management features

This chapter is the culmination of our efforts which detailed how we developed our set of primitives, termed Synergy WSN primitives. This is where we discuss various the problems that we discussed in Chapters 1 and 2, *i.e.* commissioning, OTA software upgrade *etc.*. We call the solution to these problems "features" rather than primitives because they are based on top of those preexisting primitives and are at a higher conceptual level than the primitives. This is keeping with the spirit of Synergy WSN primitives, whose sole existence is to help implement such WSN management features.

The following sections will discuss in detail how these features were implemented. They will make use of the primitives as discussed in the last chapter.

6.1 Commissioning a Synergy WSN device

Commissioning is the process by which a legitimate device can bootstrap itself to become a part of an already existing network. This is a challenge because now the devices no longer come pre-programmed with network and device parameters but have to get them at runtime via the commissioning process.

This is a area filled with possibilities and there are many companies operating in this space[6] which provide commissioning solutions. Even the ZigBee

protocol supports a commissioning cluster library (ZCL)[8] that was intended for this purpose. We decided not to depend on the ZigBee commissioning cluster and write our own Synergy WSN primitives based solution for a couple of reasons:

- The ZCL provided by TI was just an implementation that came without any working examples and we were hesitant to invest time on it without seeing some working examples first.
- Since our network topology consisted of a multi-star design with multiple networks operating simultaneously, we would have needed a solution with multiple instances of ZCL to manage the overall network. We were not sure how ZCL would have scaled in this regard.

Keeping the above in mind and also the fact our whole idea of Synergy WSN primitives was to enable such management features, we decided to go with creating our own custom commissioning solution. But before we go further in this chapter, we would like to explain a few key terms related to the commissioning process:

- **C-PAN** - Short for Commissioning PAN. The personal area network used just for device commissioning the node.
- **C-HOST** - Short for Commissioning host, a device, usually a laptop or a PDA that runs the C-APP which in turn communicates with a node programmed as the ZBC for the C-PAN via MT primitives.
- **C-App** - Short for Commissioning Application, the GUI/command line application running on the C-HOST that gives user control over the Device commissioning process.
- **O-PAN** - The Operating Personal Area Network, which is the eventual PAN on which the node will operate and attain its intended goal. There can be multiple O-PANs in the vicinity of a ZED, but the ZED will latch onto only one O-PAN at a time.

The entire commissioning process takes place over the C-PAN which was a dedicated PAN which was used only for the commissioning process. We used a laptop running Ubuntu 12.04 Precise Pangolin as our C-HOST. Later versions could be built on top of an Android Tablet or an iOS device so long as there was an USB interface that we could utilize on the host. The commissioning process was divided into two phases that were necessary for a device to transition from a brand new factory configuration into the a configuration that would allow it to legitimately connect with an O-PAN in its vicinity.

1. **Phase I**: This phase can be termed as Device commissioning where the device will be supplied with device specific parameters like, meter ID (in case of energy meters) or room ID (in case of occupancy sensors). This would be achieved via the C-APP running on the C-HOST connected to the C-PAN ZBC. The C-PAN, in conjunction with the C-APP running on the C-HOST could let the person in charge of the commissioning choose device parameters during node installation.

   This has the advantage that mass production of devices can be easily achieved because they will not come pre-programmed with their device specific parameters. This also lets us have flexibility to use the mass programmed nodes for any kind of sensor. In other words, the 'Device commissioning' sets up the 'Identity' of the device on the network. The network encryption key is also programmed during this period. Please check the Chapter 7 for more information about how Phase I device commissioning is carried out securely. We gave this phase an alternate name, **'Device commissioning'**.

2. **Phase** II: This is the phase where the device actually acquires the network parameters, *i.e.* the O-PAN characteristics and then decides on an O-PAN to join based on a predefined policy. In theory this phase could be eliminated and merged with phase I. However, doing so has the disadvantage that if at a later time if the O-PAN goes out of range or something happens to it, then one would have to come in with the C-App and C-Host to recommission the device. So a conscious decision was made to implement Phase II com-

pletely in-device. This increased the complexity of phase II, but it allowed us to not worry about corner cases of O-PANs going out of order. Just like device commissioning, we gave this phase an alternate name too, 'Network commissioning'.

The state chart in Figure 6.1 describes the relationship between various phases and commissioned states.

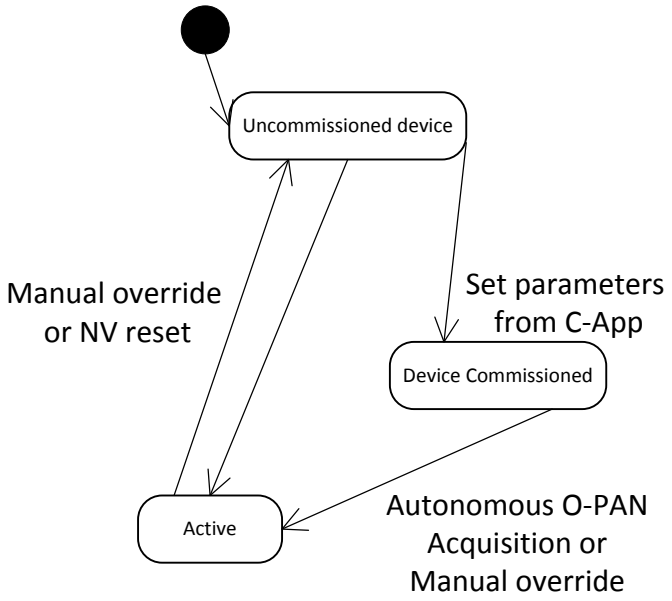


Figure 6.1: Commissioning states and the transitions between them

6.1.1 The Commissioning App (C-App)

The C-App was a GUI application that was written using Python and PyGtk, intended to run over a Linux machine. It used easily available Python bindings for the GTK+ graphical toolkit library in order to present the user with an easy to use interface to device commission a Synergy device. The GUI is just a front to the underlying Synergy WSN primitives which did the actual work of setting the commissioning parameters in the target device. The software for the ZBC used for commissioning was exactly identical to the O-PAN ZBCs except that these ZBCs operated at a dedicated PAN for commissioning, which we termed the C-PAN.

Figure 6.2 shows a schematic representation of this setup.

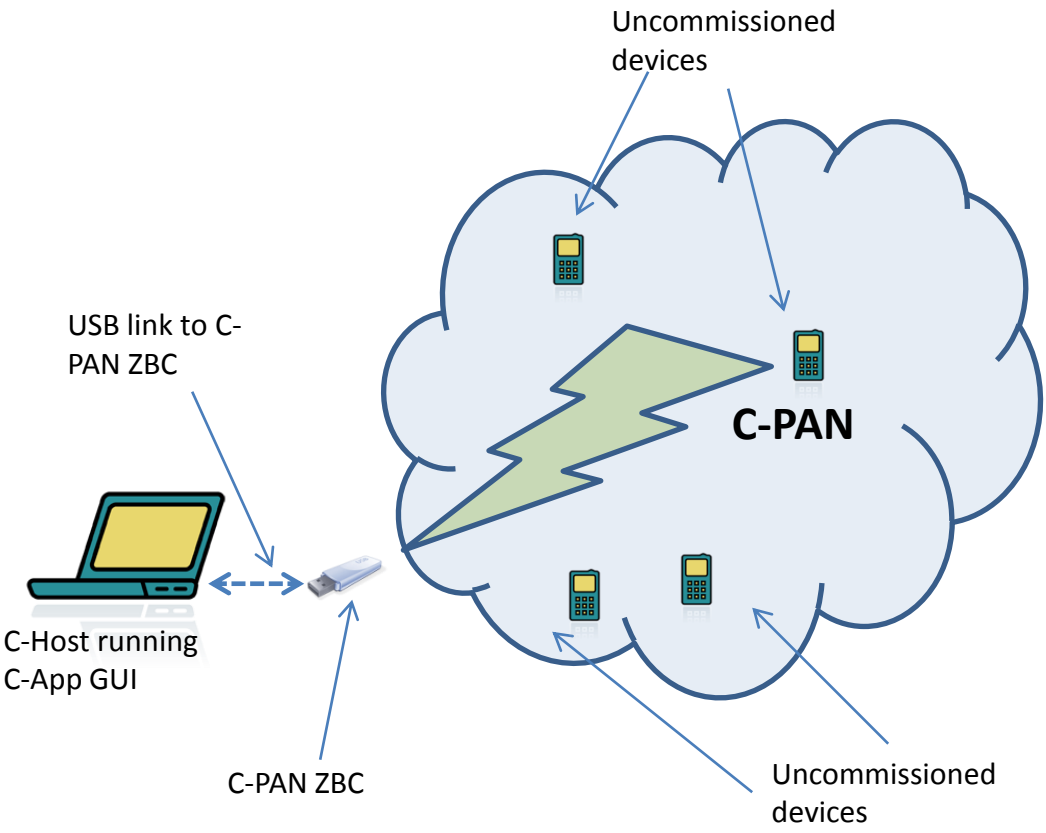


Figure 6.2: Setup for commissioning using the C-App

Note that the same commissioning could be achieved via base station scripts and sending the Synergy WSN primitives manually. A GUI however gives a much better user experience from typing commands and makes it ideal for use by non-technical personnel, *i.e.* building managers performing the actual installation of these devices in a building.

The below sequence diagram describes how the C-App was designed. It consisted of three main entities: the GUI entity, the C-App core entity and the ActivityThreads entity. This three layered design allowed the GUI part to be abstracted out from the inner implementation of ActivityThreads which dealt with sending and parsing raw Synergy WSN primitives. The ActivityThreads were designed a Python threads that were designed to perform just one functionality and block while doing so. The GUI would register callbacks with the C-App core

and the C-App core, in turn would register callbacks with the ActivityThreads implementation. This two level callback daisychaining allowed us to isolate the GUI from the implementation. If in future the GUI is replaced with Qt or some other toolkit, one just needs to rewrite the GUI entity without changing much of the remaining two.

The message sequence chart in Figure 6.3 depicts the interaction between these three entities.

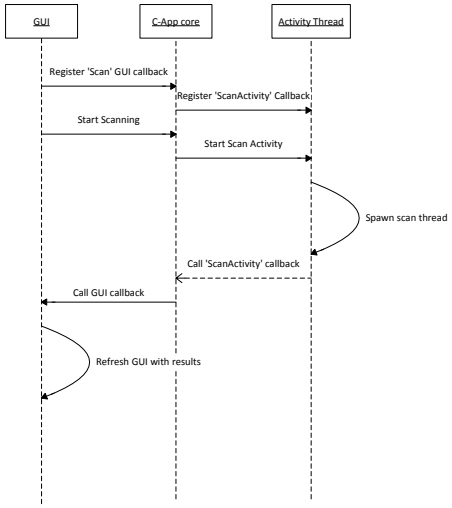


Figure 6.3: Message sequence chart showing the 'Scanning' use case in the C-App GUI

Commissioning is achieved by using the GUI to first scan the vicinity for non-commissioned devices and once some are found, the C-App can be used to send Synergy WSN primitives to these non-commissioned devices to transition them to active states. The following screenshots of the C-App will give the reader a feel for how this is done. We begin by first showing the reader how the GUI of the C-App looks like. Please note that the GUI we developed was for reference purposes only. Even though fully functional, we did find some usability issues like for example, the GUI had all the individual interfaces but lacked a wizard interface to help the commissioning personnel program the various parameters without remembering the steps from memory. This is something we intend to fix in the next revision.

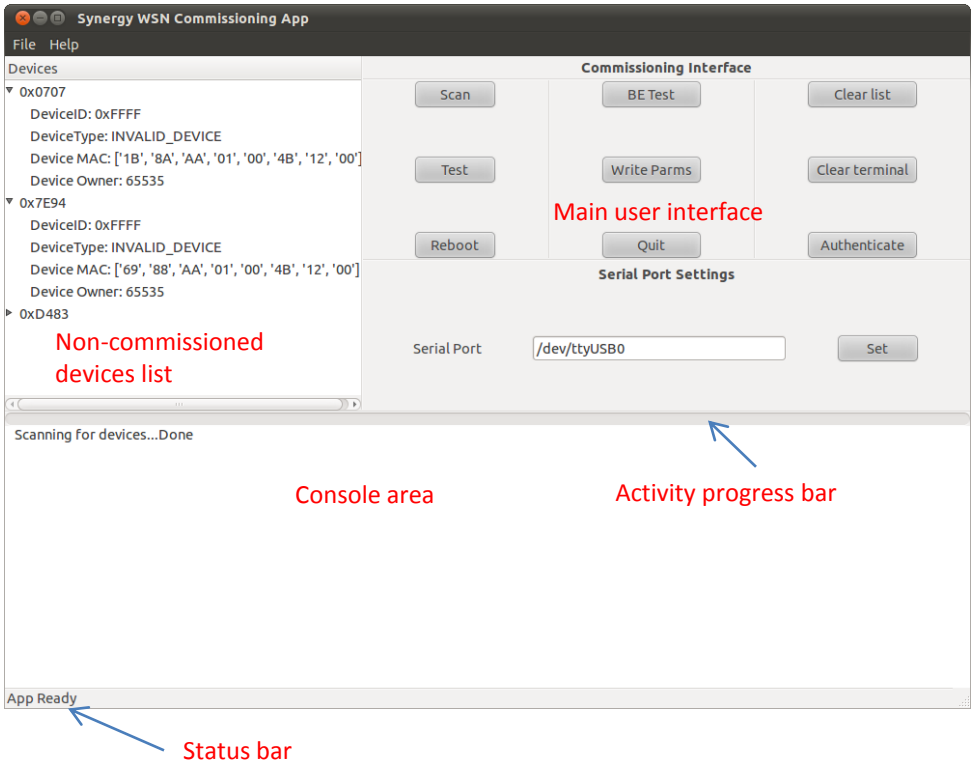


Figure 6.4: A labeled figure of the C-App GUI

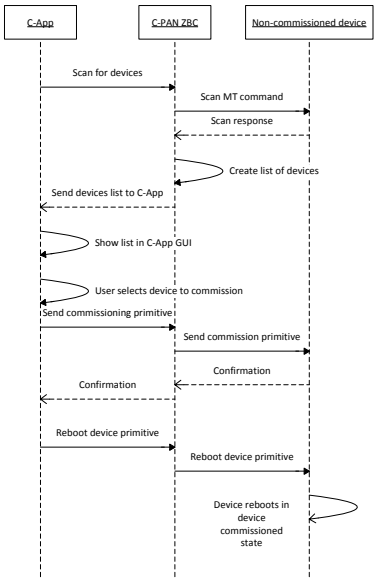


Figure 6.5: Device commissioning message sequence chart

As can be seen from Figure 6.4, the C-App GUI gives the user a means to scan for non-commissioned devices and then program such devices OTA with the device commissioning parameters. Figure 6.5 shows the message sequence chart detailing the device commissioning process.

6.1.2 The Autonomous Operating PAN (O-PAN) acquisition

This section deals with phase II, also known as the network commissioning. This phase takes place once the device has been commissioned with the device specific parameters via the C-App. The following flow-chart will make it easier to understand this process.

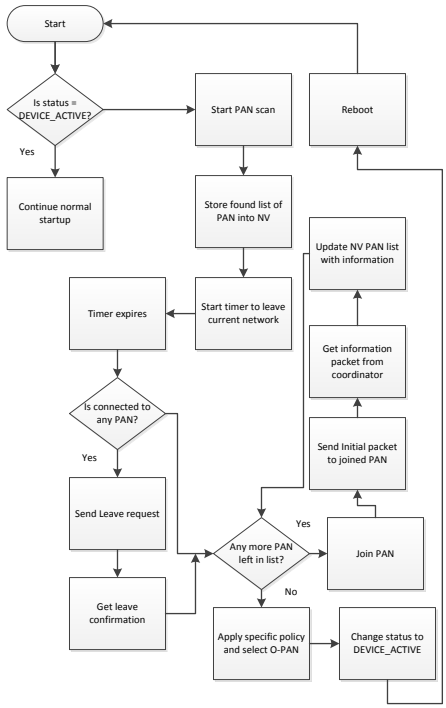


Figure 6.6: Autonomous O-PAN acquisition flowchart

Before phase II, we have successfully setup the identity of the device on the network. However, it has not figured out which O-PAN it will eventually latch onto as there could be more than one O-PAN in its vicinity. Once this phase is

complete, we have the device in active mode and it starts to work once it latches on to the correct O-PAN.

Note that, this phase could be integrated with the C-App by giving the installation personnel the capability to choose the O-PAN right after phase I was complete. However, this introduces a dependency on the physical C-App setup in case a O-PAN stops working. This is the reason we want this process to take place as autonomously as possible. The entire phase II is implemented on the device.

Also, worth noting is that the O-PAN selection policy is something that can be developed independently of the entire commissioning process. The scanning for O-PANs provides several key O-PAN characteristics which can be used to make a decision as to which O-PAN should be selected. For testing purposes however, we chose an algorithm that chose the O-PAN with the least amount of loading. Other possible policies can include taking the total data throughput in an O-PAN into account before making choosing that O-PAN, in conjunction with several other parameters like average RSSI, average LQI etc. (all of which are obtainable via Synergy WSN primitives).

6.2 Over the air (OTA) upgrade of Synergy WSN devices

Any big network deployment needs to have the ability to upgrade its software OTA (over-the-air) otherwise upgrading the software by manual flashing of tens or hundreds of nodes can quickly become annoying. Our previous network iterations lacked this feature and we realized from the start that having a dependable OTA solution was one of the key things to have. TI gives its own OTA solution but it seriously lacks in several key ways which prevent it from being used in a big deployment.

Some of the disadvantages that we found in TI's OTA solution are listed below:

1. TI's solution was not scalable to a big network with hundreds of end devices. The reason for this was that it required a Windows host connected to each

ZBC and a console application had to run on *that* host. There was no way for example, to run the console app on a host in the 2nd floor of a building to do an upgrade of a end device on the 4th floor connected to a different windows host.

2. For our base station hardware we used Linux hosts and moving to Windows host just for OTA upgrade was not a viable solution.

3. TI's solution used point to point unicast to upgrade a single end device. Assuming a single upgrade takes about 10 minutes, it would have taken 100 minutes just to upgrade a network with 10 end devices.

4. TI's solution was written for smart energy and home automation profile based devices and it had a lot of overhead involved in order to comply with those specifications. Our network, on the other hand, was designed independently of those specifications and therefore we needed an OTA solution customized for our network.

5. TI's solution was very rigid in the sense that it was tightly coupled with the OTA means of transporting the image binary. It would have needed considerable changes to accept serial and/or SPI as another means of image delivery.

Our OTA upgrade solution takes care of each of these problems:

1. **Scaling the OTA upgrade solution to a bigger network:** Our solution involved giving each base station the capability to upgrade the software on each of its connected end devices OTA. This meant that the base station side of the OTA implementation would reside on each base station as opposed to TI's solution whose graphical tool would need a Windows host to be run on these hosts. Being graphical also means that it could not be automated easily. Our solution was scriptable, which meant that we could start the OTA upgrade of an end device connected to base station A, without worrying in which building and which floor base station A was located.

2. **Having our solution work without any manual intervention:** Since our base station scripts were written using Python over Linux hosts, we integrated our OTA feature within those scripts, making use of the requisite Synergy WSN primitives. This meant that it was no longer a manual process to upgrade an end device, like in the case of TI's solution which used the Windows based graphical front end that had to operated by a human at each step.

3. **Broadcast VS unicast image delivery:** We used broadcast mode to deliver the new image to all connected end devices. This reduced the total time to transmit a new image by a factor of the size of the network. So it took almost the same amount of time to program a network with 2, 5 or 10 nodes. This was not trivial because end devices in ZigBee are not configured by default to be ON all the time. We used a strategy where we tweak the network parameters for these end devices temporarily for the duration of the OTA upgrade and then revert them back to their original values after the OTA upgrade was done.

4. **No dependency on smart energy or home automation profiles:** Our solution did not have any smart energy or home automation related overheads (checking certificates, hashing etc.) which made it faster than TI's solution.

5. **Having our solution work for non-OTA image delivery methods, *i.e.* serial and/or SPI:** Our solution decoupled the image delivery process with the image installation process. This gave us an opportunity to use serial as another means of deploying the new image to target devices. Once this image was delivered, the rest of the process which was to install this new image was completely independent of the delivery process. As of this writing only serial has been implemented as there were not enough hardware GPIO pins to open up an additional SPI bus on the Anaren module (one SPI port bus already in use to interface with the external memory chip).

6.2.1 Components of the Synergy OTA upgrade solution

External memory chip

An external memory chip was used to store the new image binary. We could have used the internal NV flash of the Anaren device, but it would have severely limited the total available flash for our software because half of the total code space would have to be then reserved for the new image binary. With a total of 256 kb flash on the Anaren module, this should have left us with just 128kb of usable NV flash. This number reduces further because the bootloader again takes 4 kb of flash. Our code at that time was approximately about 128 kb in size so this approach was ruled out from the start.

Synergy WSN Primitives for OTA image transfer

There were a set of synergy WSN primitives that were primarily written for transferring an image OTA. The main problem with transferring a 128 kb image OTA via broadcast was that some packets would be invariably lost during the transfer. The lost image blocks would be also different for each device. So what ends up happening after each transfer is that we are left with devices which have holes in different locations throughout their received image. Our experiments showed that these missing blocks were random throughout the entire image.

This is shown schematically in Figure 6.7.

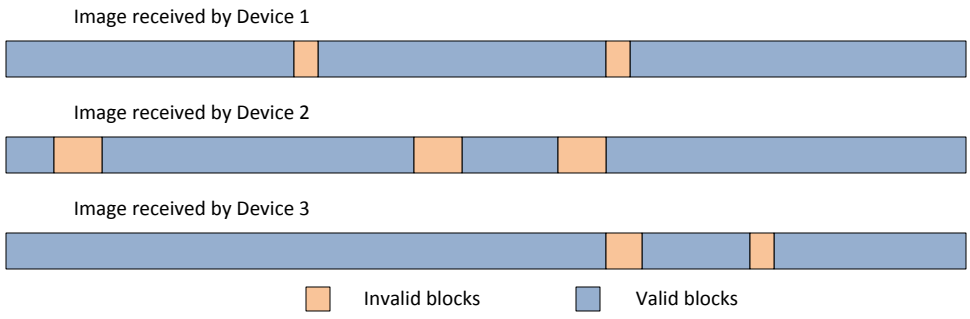


Figure 6.7: Image holes due to lost packets during an OTA image transfer

The solution to this problem was implemented in two steps:

1. Once the image was transferred, we implemented a one-to-one unicast mechanism using which each end device could ask the ZBC for the image data corresponding to the missing blocks. Only once this was completed would the end device be ready to switch over to the new image binary. This process was started and controlled by each end device because it would have been too much work for the ZBC to keep track which end device was missing which image blocks.

2. Since the end devices controlled filling these holes using the above mechanism, we first uploaded the image binary via serial to the ZBC. This enabled the ZBC to dole out missing image chunks to any end device from within anywhere in the raw image binary. The ZBC worked in slave mode in this scenario, fulfilling whatever request was sent by any of the end devices.

The message sequence chart in Figure 6.8 depicts this protocol.

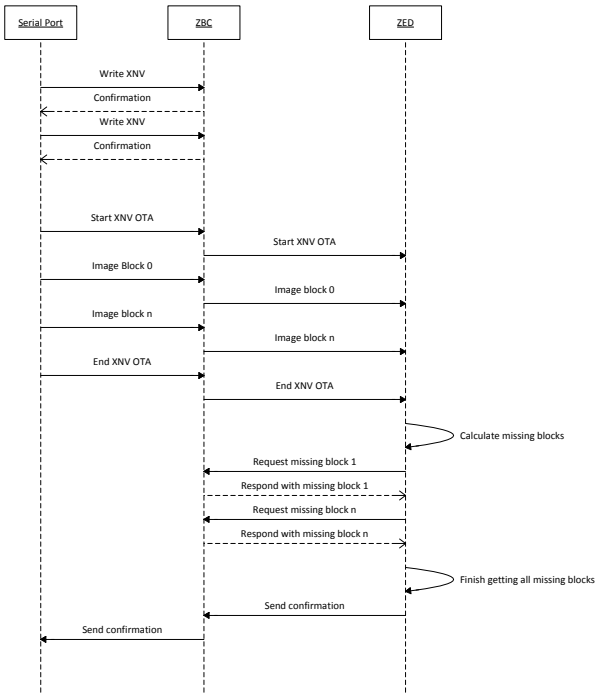


Figure 6.8: Hole plugging strategy during OTA image transfer

This protocol ensures that image integrity is always maintained at the end devices using a CRC scheme. This works even for corrupt packets because any corrupt packet will not pass the packet integrity checks at the end device side and will get dropped by the Synergy WSN primitives' parsing logic. All these dropped packets will therefore be treated by the end device as if the packet was never transmitted by the ZBC and they will be added to the list of holes in the image.

The Bootloader

Our OTA implementation required the use of a bootloader whose main responsibility was to check if the new image needed to be loaded onto the internal flash of the Anaren module and then go and load it. In case there was no need to load the new image, the bootloader would jump to the application code and forward all device interrupts to the application defined interrupt vectors from that point onwards.

Our bootloader was based upon TI's bootloader, except that we modified it to have the following differences:

1. We modified it to be able to detect whether it was to load the new image from the factory area or the production area. Our external NV was divided into two regions: production and factory. The factory region housed an image which could be used to restore the device to a known good state. The production region on the other hand, contained an image that was more recent and more bleeding edge.

2. Since the first change increased the size of the bootloader, it was no longer small enough fit into the 2kb dedicated to it in TI's design. We had to increase the total area allocated to the bootloader by modifying the appropriate linker scripts.

At device reset, the bootloader examines a specific GPIO which is connected to a push button switch and depending on the switch state it determines whether it needs to load the factory code or the production code. It then goes ahead and

reads data from the external NV chip and writes it onto the internal NV area of the Anaren module. Once this is complete, it simulates an application code reset by directly jumping to the address immediately after the bootloader region, which in this case happened to be 0x1000 as the bootloader space reserved was 0xFFF, *i.e.* 4kb. The following flow chart depicts the bootloader logic.

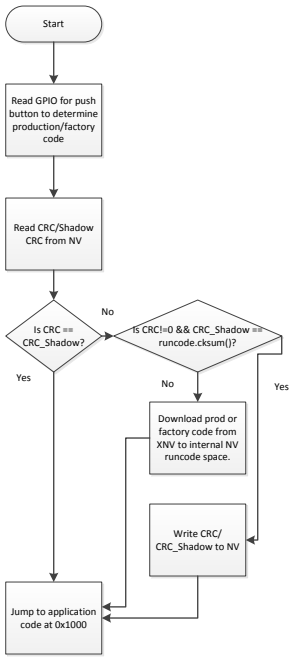


Figure 6.9: Synergy OTA bootloader logic

The bootloader uses a checksum to determine the integrity of the downloaded image. It stores the checksum in the "CRC_Shadow" variable so that it does not have to calculate this every time the device boots up as this is a time consuming process taking approximately a minute for an 128 Kb image.

Run code invalidation

The device uses the external NV memory as a staging area for the new image binary while the current image is still valid within the internal flash of the Anaren module. We need a mechanism to invalidate the current image within the internal flash so that the new image binary can be loaded from the external area. This is done by writing zeros to the current image checksum. A dedicated area is

present within all OTA enabled binary images that contains the image checksum and the image shadow checksum. By writing zeros to the image checksum we cause the bootloader to load the new image binary from the external memory on the next device reset.

OTA upgrade with encryption

Our initial OTA upgrade solution was done without enabling encryption. When we turned on encryption we found that our data download time almost doubled. This was mainly because encryption introduced extra packet overhead for each MAC frame that was transmitted. This made less space available for our payload. Since the image blocks had to end on 256 byte page boundaries on the external NV chip, we dropped from a 64 byte packet to a 32 byte packet. This effectively doubled the number of image packets we had to send and therefore caused the extra delay. We are investigating ways to increase the MAC frame size in order to be able to transmit the same sized packets with or without encryption.

Chapter 7

Security

7.1 Securing the ZigBee network

Security is always of utmost concern whenever talking about large scale deployments. ZigBee on its own supports several level of security[10] and even supports link based keys. For our purpose, we however decided to opt out of link based security as we felt having the same key throughout the network was adequate. To compromise such an arrangement it would imply physical access to the building, breaking into our backend server and having access to our firmware sources - all of which are decrease in likelihood, in that order. Future work could of course, enhance this model even further at the cost of increasing the inconvenience of users legitimately using the system, as with any security system. Nevertheless, the risks we faced were two fold:

- Since some of the end devices that we were planning to deploy were occupancy sensing nodes for offices, a malicious entity could in theory, gain access to the private occupancy data of its occupant(s). Many people would be uncomfortable with that notion.

- Another type of end devices that is being planned is an energy metering device that also has actuation capabilities. A malicious attacker in this case, could theoretically turn ON and OFF any electrical load that was connected to these energy metering devices.

Security was therefore never something we considered optional from the beginning. The advantage we had with ZigBee security was that it was designed to be orthogonal with everything else in the ZigBee protocol stack. We did our initial development without turning on security. It was only in later stages that we ended up turning it on. ZigBee supports full 128-bit encryption. The Anaren modules which are derived from the CC2530 have a separate hardware AES module that freed up the CPU from encryption/decryption. This did have some impact on the overall packet throughput nevertheless.

We realized that while it was relatively simple to get two or more ZigBee devices to communicate securely using ZigBee, it was another matter when it came to managing security over a large scale deployed WSN. Thinking of a large WSN deployment naturally brought up questions like:

- How do we perform key rollover when it is time for the network to switch to a new encryption key ?
- How do we ensure that before the key is rolled over, all the end devices have successfully updated their keys to the new ones ?
- How is commissioning affected with respect to encryption ?
- And not just ZigBee security, how do we secure the backend from external attacks ?

The first problem requires that we have a way of updating the encryption key dynamically within the running end devices. There are preexisting MT primitives for achieving this. However, they don't ensure that all end devices have been delivered the new key. In order to make sure that all devices have successfully received the key, it requires some additional work which our Synergy WSN primitives do.

This is shown in the sequence diagram in Figure 7.1. Notice that the key is broadcast at first. And then the ZBC queries each end device about the key status via a unicast. The new key is piggy backed in this query thereby giving a chance to each participating device to update its key in case it did not get it during the

initial broadcast. Once the ZBC knows that all end devices have updated their keys, it reboots itself. This causes the end devices to reboot themselves and when they come back up, the end devices along with the ZBC start using the new key.

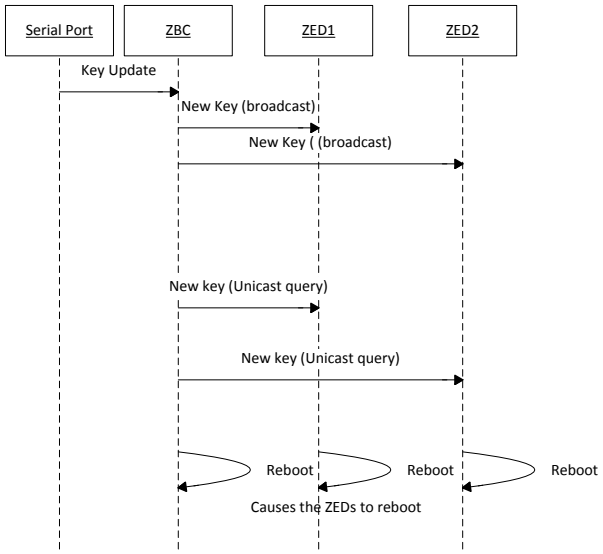


Figure 7.1: Key dissemination strategy used in our network

The network key is maintained in the backend and it can disseminated from there. The entire network with all its constituent multi-star networks use the same network key. The exact means of sending commands from the backend to the installed ZBC is a topic of the next chapter. The important thing is that once the key is pushed to the ZBCs, the ZBCs take care of disseminating it among their end devices. This key is also a part of the various commissioning parameters that are supplied to an end device during the commissioning process.

7.2 Securing the commissioning process

Just as securing the running ZigBee network is important, it is also important that the commissioning process is also secured from possible attacks. Otherwise, an attacker can sniff the key being sent out to a device being commissioned and compromise the whole network.

The way we prevent this is by programming each new non-commissioned

device with a key that is also pre-shared with the C-App. That way, the C-App and the device being commissioned can communicate with each other as soon as the device is out of the factory. During the commissioning process, the C-App authenticates with the backend by interactively getting the credentials from the user. Only users who are authorized to use the C-App will be able to authenticate against the backend. If they are able to do so, the C-App makes a backend query to get the currently valid key. This key is sent via the C-App to the device being commissioned through a Synergy WSN primitive. After rebooting, the device uses the latest key being used by all O-PANs and is hence successfully able to perform the network commissioning successfully. Thereafter, the device becomes a legitimate part of the network.

If an attacker gets hold of a non-commissioned device and the C-App, it will be able to get the two of them to communicate. But won't be able to sneak in the new device onto the actual network.

The only way an attacker will be successful is if he or she gets a hold of the pre-programmed key from an non-commissioned device by reverse engineering the flash memory to read the location where the key is stored. This is non-trivial and even if the attacker manages to do this, he or she then has to be in close vicinity of a C-App being used in order to use the extracted key to extract the current network key from OTA traffic. Thus both of the above requirements make it extremely hard for an attacker to gain access to the currently used network key.

There is of course the question of what if the attacker sniffs the backend authentication password when the commissioning personnel are entering the password in the C-App GUI. We address this problem by using an SSH tunnel between the C-App GUI and the backend so that the password is never sent in plain text over Wi-Fi or the wired network.

With these changes, we present an updated view of the process described by Figure 6.5 from Chapter 6.

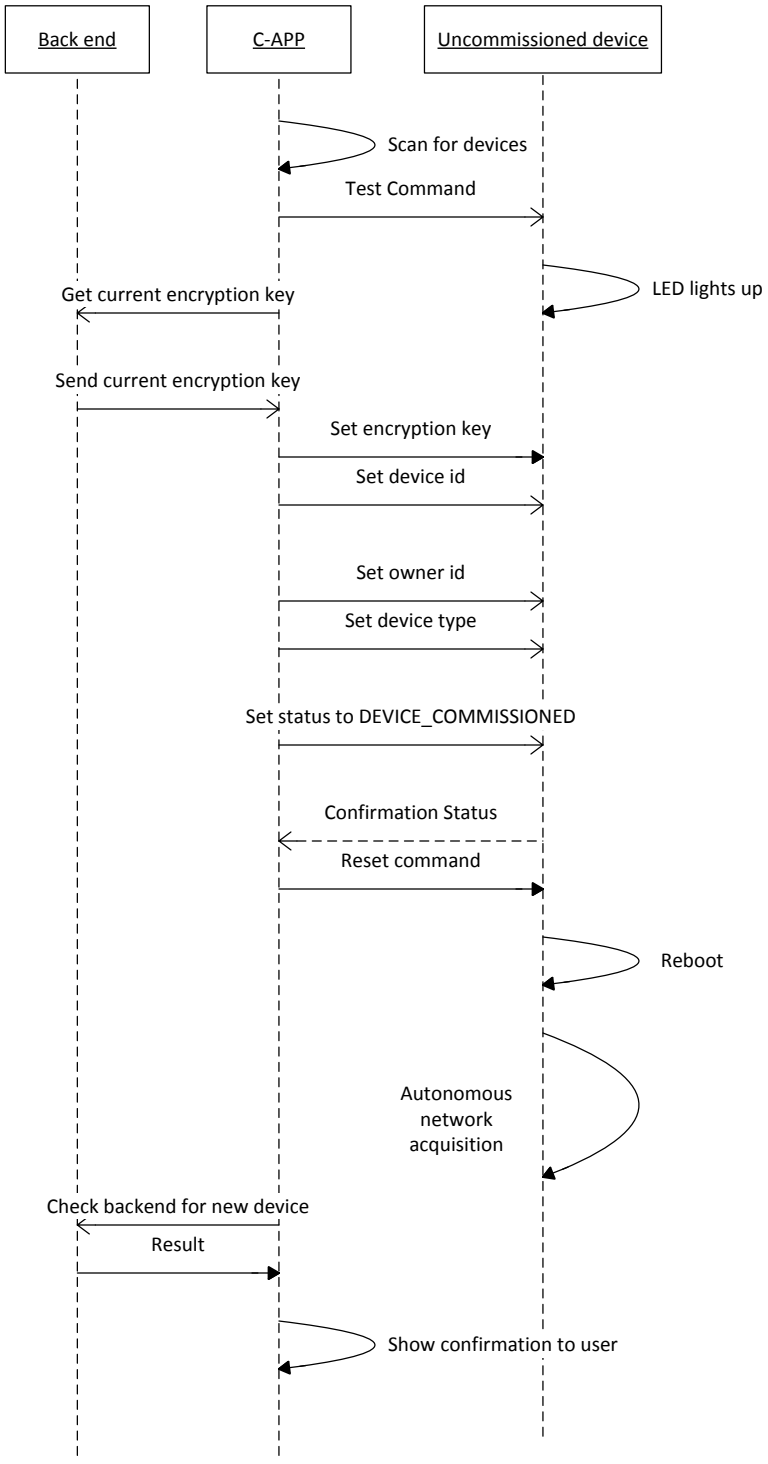


Figure 7.2: Device commissioning with encryption

7.3 Securing the base stations

The base stations that were used for our experiments were on the wired computer network of the Computer Science and Engineering department and their IP addresses had to be authorized by our IT department prior to installation. The network they were on was an internal IP network that was shielded from the outside world, so outside penetration is not possible. Hence, we did not feel the need to secure the wired link. However, we recently realized that securing these links via SSH tunneling would be a good idea nevertheless. This is planned to be implemented in the next iteration of our network.

Chapter 8

Backend design

8.1 Database organization

As mentioned in chapter 3 we used MySQL as our backend database to store the running network statistics. We divided our data into separate tables for better management. In all, our database had about seven different tables which are listed below:

- **Device Info**: This table contains the most recent reading from all the end devices. Device ID is the primary key for this table.
- **Device software status**: This table contains the software version information for each of the end devices. The primary key for this table is the device mac.
- **Device heartbeat**: This table contains the periodic heartbeat messages from end devices that can be used to detect if it has fallen-off the network.
- **Base station info**: This table contains specific information about ZBCs/base stations like number of children, PAN ID etc. The primary key for this table is the base station IP address.
- **Device statistics**: This table contains statistical information about packets processed by each ZBC/base station. Device ID is the primary key for this table.

- **Base station children map**: This table contains all the short addresses connected to any particular ZBC/base station. There is no primary key in this table.

8.2 Remote command execution

One of the key features that we need is to visualize the network status through a web based interface or a standalone application. This would allow us to represent the network graphically over a browser canvas with icons for ZBCs and ZEDs. The user can right click any particular ZBC/ZED and execute any arbitrary Synergy WSN primitive on that device. This would suit not only interactive applications, but even batch maintenance of the network could be done using such a mechanism. For example, one can schedule key rollover every Sunday at midnight, or a network wide software upgrade could be done with the help of a single button click.

We propose the use of a web server gateway interface on each base station, similar to CGI or WSGI for this. Native scripts could be executed on each Linux base station for each Synergy WSN primitive being exposed. A test implementation was done using CGI or Common Gateway interface for demonstration purposes. The next revision of the Synergy WSN will contain a more complete implementation.

The idea of remote command execution on the ZBCs/ZEDs is depicted graphically below:

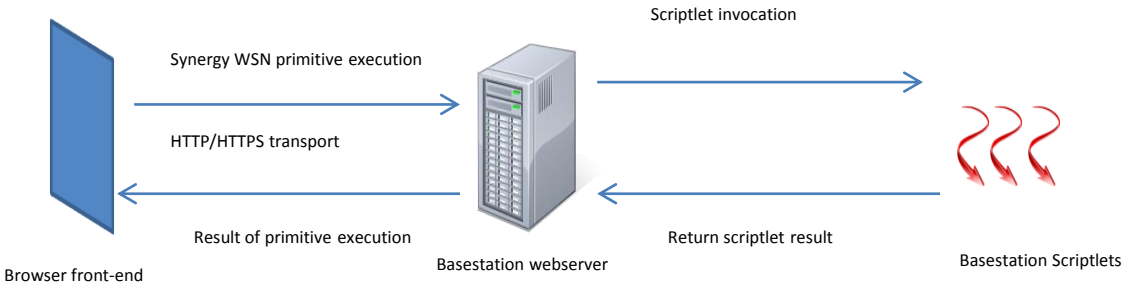


Figure 8.1: Remote Synergy WSN primitives execution mechanism

For the above scheme to work, we split up the base station scripts into tiny single threaded programs that we called 'scriptlets'(not to be confused with Java Server Pages Scriptlets). These scriptlets are tiny scripts that do only one thing and do it well, in the spirit of the Unix philosophy. Using a web based interface, at the press of a button say, a particular scriptlet could be executed within the context of the browser. The results could then be returned via CGI or WSGI to the browser. The browser may then render the output for the user to see. We did a small proof of concept implementation for this and were convinced that this was the right way to go forward with the problem visualizing the entire Synergy WSN from a central vantage point. The screenshot in Figure 8.2 demonstrates a proof of concept of this proposed implementation.

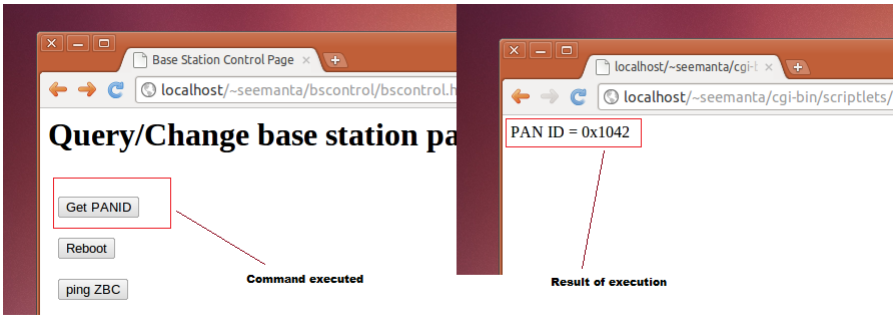


Figure 8.2: Prototype for web based Synergy WSN primitives execution

Chapter 9

Evaluation

9.1 Network setup and deployment

For evaluating our work as seen in the previous chapters we used a deployment of more than 90 testbed devices in the second floor of the Computer Science and Engineering building. While deploying these testbed devices we tried to emulate real sensing devices, *i.e.* occupancy sensors and energy meters. This was done by deploying them under desks and tables in addition to hallways and offices, just as the actual sensors would have been deployed.

Our test deployment consisted of 5 different star networks with approximately 20 devices each. 3 of these networks were intentionally co-located in the same area because we wanted to test the impact of having different networks close by. The other 2 networks were spread across the second floor.

For measuring packet losses, we initially implemented the accounting within the BS Shell infrastructure. However, we soon realized that the Python based approach to do the accounting had performance issues. It was not able to keep up with arriving packets at higher transmission rates. We therefore, decided to do all the accounting within the ZBC itself. This proved to be the correct decision because we saw a sharp decrease in values reported by the in-device implementation than the older Python based implementation at same transmission data rates. Changing transmission parameters was also easy because we created Synergy WSN primitives for this very purpose as shown in Section 5.2. All tests were performed

with encryption turned ON as in any real deployment. The following sections will describe the results of our experiments.

9.2 Power measurements

For power measurements, we used a USB based National Instruments Data Acquisition System, the NI-DAQ 6218[9] for accurate measurement.

9.2.1 Sleep VS Scan power

Recall from Section 4.3 that battery powered ZigBee End devices can potentially drain their batteries if no ZigBee Coordinator is available nearby. To overcome this problem we implemented a scheme to periodically wake up and go to sleep if no ZigBee Coordinators are detected. We measured the power for this scenario and Figure 9.1 compares sleep power VS scan power.

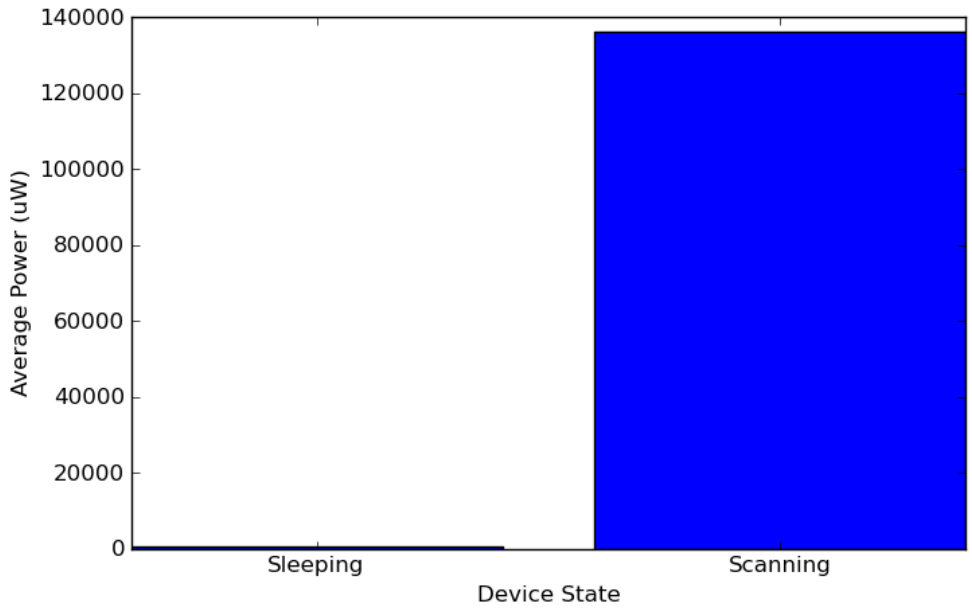


Figure 9.1: Sleep VS Scanning power of a battery powered testbed device. Sleep power is approximately 380uW while scan power is approximately 136 mW.

The reason for a high power consumption during scanning is because the radio is always ON during the scanning stage.

9.2.2 Power profile of single transmit

In order to measure the transmit power with finer granularity, we measured the power consumption before, during and after a single packet of length 10 bytes was transmitted from a ZigBee end device. Figure 9.2 shows this measurement.

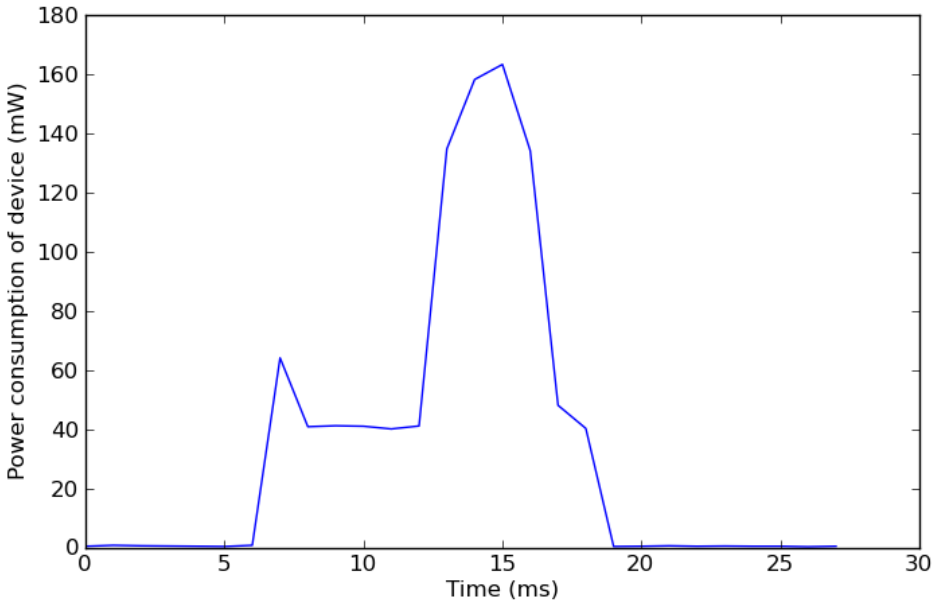
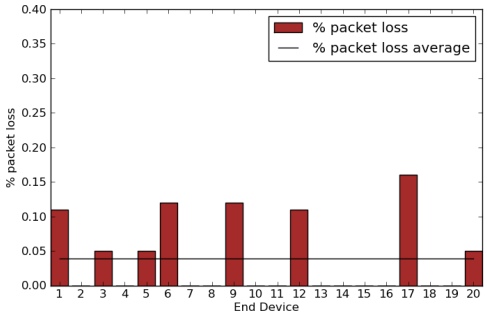


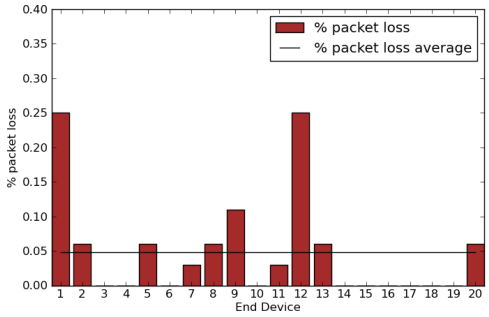
Figure 9.2: Power profile of a single transmit of 10-byte packet length.

9.3 Network evaluation

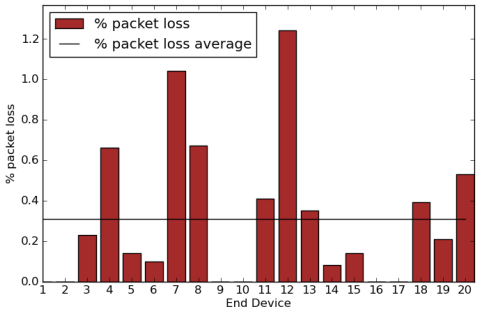
Our testbed devices were programmed to transmit a test data buffer of a certain size after every fixed time interval. Both the buffer size and time interval were NV configurable. The default rate was fixed at a packet size of 10 bytes, transmitting at the rate of once per second. Tests were performed to study the effect of changing the transmission rate while keeping the packet size fixed and vice versa.



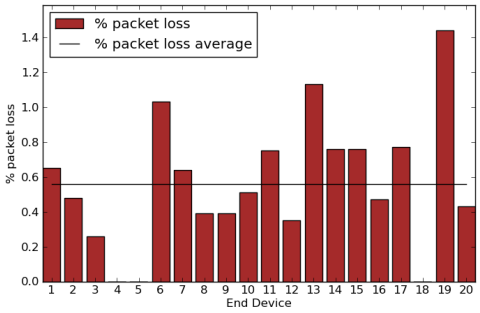
(a) Packet loss with 1/s transmit rate



(b) Packet loss with 2/s transmit rate



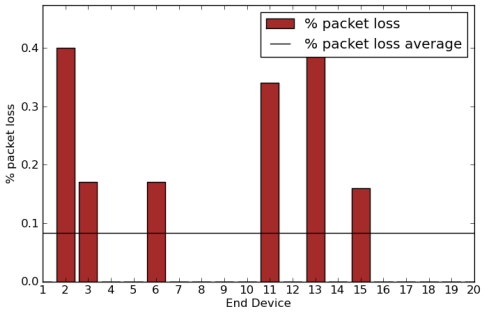
(c) Packet loss with 4/s transmit rate



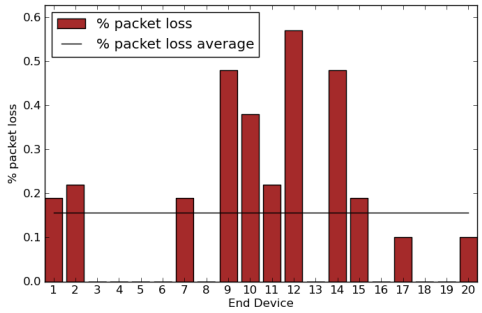
(d) Packet loss with 8/s transmit rate

Figure 9.3: Packet loss statistics for transmit rates of 1/s, 2/s, 3/s and 4/s with packet size of 10 bytes. Notice how the packet losses increase with increased data rate.

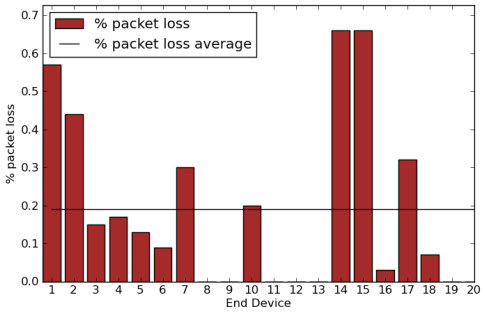
Figure 9.3 shows the packet loss statistics when the transmission rate is varied from once per second to eight times per second with a payload size of 10 bytes. As expected, the packet losses increase as we increase the throughput of the testbed device. The same test is repeated with a different payload size of 20 bytes and the results are shown in Figure 9.4. However, as can be observed from the figures, the packet loss is lower than 1



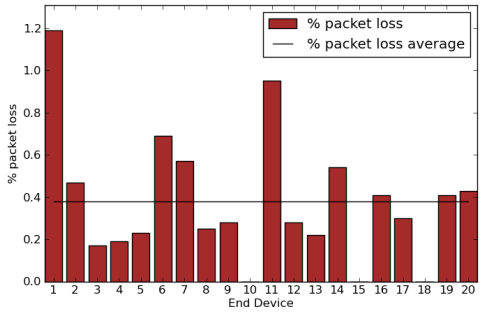
(a) Packet loss with 1/s transmit rate



(b) Packet loss with 2/s transmit rate



(c) Packet loss with 4/s transmit rate



(d) Packet loss with 8/s transmit rate

Figure 9.4: Packet loss statistics for transmit rates of 1/s, 2/s, 3/s and 4/s with a packet size of 20 bytes. As observed, the packet loss is below 1% for all of the cases.

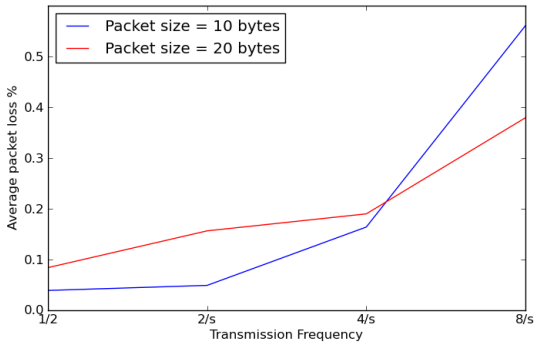


Figure 9.5: Packet loss for different packet sizes.

In Figure 9.5 we plot the average packet losses for both packet sizes. As can be seen, increasing either the packet size or the transmission frequency increases the

number of lost packets. However, even for a transmission frequency of 8 packets/s, the losses are well within 1% for a 20 byte packet.

The payload size in all the above tests excluded the packet overhead which was 16 bytes per packet and the tests were done with a PAN with 20 transmitting end devices in it.

9.4 Over the air upgrade evaluation

As mentioned in Section 6.2.1, we use broadcast messages to disseminate blocks of the image binary. This broadcast is not without lost packets. Each transfer results in some lost packets. These lost packets are recovered in the next phase which is the hole plugging strategy as discussed in Section 6.2.1. The graph in Figure 9.6 quantifies these losses for a 12 node network. We used full 128-bit encryption for this test, employing a 32-byte packet for transferring a test binary blob of 128Kb. This resulted in a total of 4096 image blocks being transferred from the ZigBee Coordinator(ZBC) to each of the end devices. The transfer was repeated 10 times and the packet losses at each iteration is plotted in the graph.

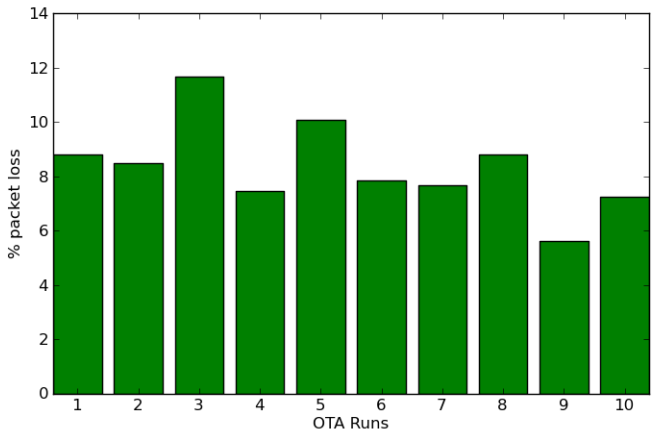


Figure 9.6: Packet loss statistics during a 12 node OTA transfer with 4096 blocks of 32 bytes each.

These lost packets turn up as invalid image blocks and are subsequently filled in the next stage as described in Section 6.2.1

Chapter 10

Conclusion

We started our work by identifying the goals and features of any large indoor WSN deployment. These goals include commissioning, OTA software upgrade, general maintenance to name a few. We then justified the need of an abstraction layer within a large and complex WSN to provide units of functionality to higher layers. We showed how this abstraction layer of primitives could be agnostic to the underlying WSN technology, though parts of it could be tightly coupled with the underlying WSN technology.

We also provided a justification of how having such a layer of primitives is helpful to implement more complex features by making use of these basic primitives. This led us to propose a layer of abstraction which we named 'Synergy WSN primitives'. We developed a number of these primitives to show that they can be a viable approach to managing the complex problems of deploying real world WSNs. We described our software and hardware design to support these Synergy WSN primitives and how we leveraged TI's Monitor and Test API for our work by enhancing it beyond its original scope. Thus, Synergy WSN primitives was the union of preexisting MT primitives and the set of primitives that we developed for our work.

We also documented our primitives and their functionality. This was followed by a discussion of commissioning and over the air software upgrade of a WSN which utilized these Synergy WSN primitives. We evaluated our system over a deployment of more than 90 testbed devices in the second floor of the Com-

puter Science and Engineering building. The results of our evaluation were critical in demonstrating the usefulness of our system. We were able to show that our network could use these management features securely and with low packet loss rates. It is hoped that future WSN deployments undertaken by SYNERGY labs will make use of the rich set of primitives developed as part of this thesis.

Bibliography

[1] 6lowpan. http://datatracker.ietf.org/wg/6lowpan/charter/.

[2] Altium designer. http://www.altium.com/.

[3] Cc2530 user guide. http://www.ti.com/lit/ug/swru191c/swru191c.pdf.

[4] Control4 composer pro. http://control4.com/residential/products/software/.

[5] Control4 home automation system. http://control4.com/.

[6] Daintree commissioning white paper. http://backup.daintree.net/downloads/whitepapers/ZigBee-commissioning-tools.pdf.

[7] Guruplug developer and user forum. http://www.plugcomputer.org/plugforum/index.php.

[8] Installer for zstack - the texas instruments zigbee stack. http://www.ti.com/tool/z-stack.

[9] National instruments ni usb-6218 data acquisition module. http://sine.ni.com/nips/cds/view/p/lang/en/nid/203484.

[10] The zigbee protocol specifications. http://www.zigbee.org/Specifications.aspx.

[11] The zigbee wireless protocol. http://zigbee.org/.

[12] The zwave wireles protocol. http://www.z-wave.com/modules/ZwaveStart/.

[13] Y. Agarwal, B. Balaji, S. Dutta, R.K. Gupta, and T. Weng. Duty-cycling buildings aggressively: The next frontier in hvac control. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 246–257. IEEE, 2011.

[14] Y. Agarwal, T. Weng, and R.K. Gupta. The energy dashboard: improving the visibility of energy consumption at a campus-wide scale. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, pages 55–60. ACM, 2009.

[15] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 43–56. ACM, 2008.

[16] M. Buettner, G.V. Yee, E. Anderson, and R. Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 307–320. ACM, 2006.

[17] M. Ceriotti, L. Mottola, G.P. Picco, A.L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon. Monitoring heritage buildings with wireless sensor networks: The torre aquila deployment. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 277–288. IEEE Computer Society, 2009.

[18] S. Dawson-Haggerty, S. Lanzisera, J. Taneja, R. Brown, and D. Culler. @ scale: Insights from a large, long-lived appliance energy wsn. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 37–48. ACM, 2012.

[19] D. Egan. The emergence of zigbee in building automation and industrial control. *Computing & Control Engineering Journal*, 16(2):14–19, 2005.

[20] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369. IEEE, 2005.

[21] X. Shi and G. Stromberg. Syncwuf: An ultra low-power mac protocol for wireless sensor networks. *Mobile Computing, IEEE Transactions on*, 6(1):115–125, 2007.

[22] T. Weng, B. Balaji, S. Dutta, R. Gupta, and Y. Agarwal. Managing plug-loads for demand response within buildings. In *ACM Workshop on Embedded Sensing Systems For Energy-Efficiency In Buildings*, 2011.

[23] L. Zheng. Zigbee wireless sensor network in industrial applications. In *SICE-ICASE, 2006. International Joint Conference*, pages 1067–1070. IEEE, 2006.