Technische Universiteit Delft
Wireless and Mobile Communication (WMC) Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Design and Implementation of a Wireless Sensor Network Testbed

Jiangjie He

Committee Members:

Mentor	Ir. Cheng Guo
	Dr. Martin Jacobsson
Supervisor	Prof. Ignas Niemegeers
Member	Dr. Stefan Dulman

Date: July, 2009

Abstract

Wireless Sensor Network (WSN) is becoming a more and more popular research area since its emergence. Many WSN researches rely on practical experiments in real wireless networks. However, to setup and conduct such an experiment is always cumbersome and time-consuming. So in this thesis, we design and implement a WSN testbed to solve the problem and help user focus on their researches. Our testbed provides necessary modules and service, such as command dissemination and feedback module, data logging and collection module, network programming module and time synchronization service. Different from other existing testbeds, our testbed removes wire connections and gateways, which are used for transmitting commands, feedbacks and users' programs, between nodes so that it provides high flexibility in deployment and reduces costs. This testbed can be widely applied in different environments without extra devices but experimental motes.

We have deployed our testbed on the 19th floor of EWI building at TU Delft and verified its functionality and performance with tests. The testbed meets our design goals that it can be used for most WSN experiments. We hope this testbed can make a contribution to WSN research in the future.

Keywords: Testbed, wireless sensor network, IEEE802.15.4

Acknowledgments

First, I would like to thank my mentor Cheng Guo for his patient guide and suggestion on my thesis project, especially for his big help on my thesis writing. And I want to thank Martin Jacobsson for his patient explanation of related knowledge and comments. I appreciate Prof. Ignas Niemegeers and Dr. Fernando A. Kuipers for their kindness and patience when I was looking for my thesis project.

I am thankful to all the friends who were working together with me in the lab. They make the research life enjoyable.

And I also want to say thanks to Egbert W. Bol who gave me an impressive interview before I came to TU Delft.

At last, I would like to thank my parents who give me confidence and courage to face the difficulties in the life.

Contents

Chapter 1. Introduction

Wireless Sensor Networks (WSN) has been widely applied to numerous fields such as battlefield surveillance, environmental monitoring, intelligent building and etc since its emergence. At present, there are thousands of research developments taking place in this field, including new protocols, energy saving mechanism and performance improvements. Most of these researches have to be verified by real-world experiments under practical environment. However, setting up and conducting such an experiment usually costs lots of time and effort. In order to solve this problem, building up a WSN testbed to handle complicated things of setup and make user to focus on their researches should be a necessary work.

1.1 Wireless Sensor Network

WSN is a network that uses wireless connected sensor devices to monitor and communicate specific conditions. It gathers concerned information such as voltage, pressure, motion, sound and etc in different locations, especially in some restricted place that people cannot be competent, e.g. in nuclear power plant.

A WSN can be formed by multi sensor nodes with different topologies such as star, tree and mesh structures. Different multi-hop routing protocols are applied in these WSN to broaden the communication range. Take a typical WSN for instance which is showed in Figure 1.1, the Basestation node connected with PC plays a role of gateway between WSN and PC while other nodes communicate with each other by wireless. The nodes in the WSN can receive and send information over multi hops with suitable protocols. In order to support interconnection among the different WSNs or between WSN and other kinds of networks, some standards have been proposed with different scope.

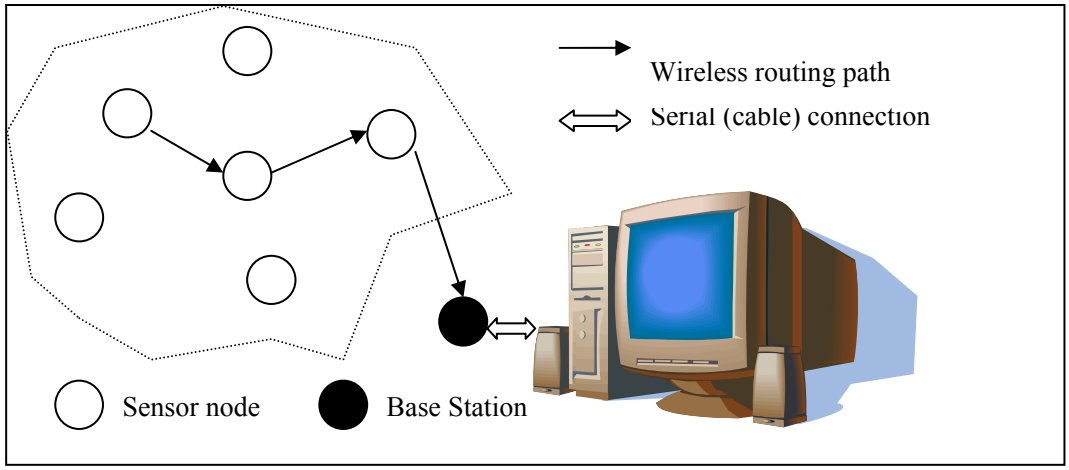


Figure 1.1. A typical structure for WSN

Among others, IEEE standard 802.15.4 is the major standard of WSN, which specifies the physical and MAC layer. IEEE802.15.4 focuses on low cost, low speed ubiquitous communication between devices. With little underlying infrastructure, communication among nearby devices can be very low. In addition, IEEE802.15.4 also has other important features, such as real-time suitability, collision avoidance, power management and etc.

ZigBee, which develops their protocols based on IEEE802.15.4, is an example of successful commercial alliances which has already tag their trademark in some consumer devices. ZigBee supplements IEEE802.15.4 with additional specification on higher layers which also targeted at RF applications that require a low data rate and long battery life.

In order to interconnect wireless network with traditional IP network, 6lowpan is proposed to provide internet connectivity for low cost wireless devices, mapping IPv6 on IEEE802.15.4 which includes packet sizes adaptation, address resolution and power management for different devices.

As we see above, low energy consumption is one of the most important characteristics of WSN. Due to the varying locations of WSN, batteries are usually the power supply. The network must to be alive for a long period in most situations. So hardware and software of WSN should be designed with low energy consumption in mind.

Along with the development of semi-conductor circuit, not only energy cost but also the size of device is becoming smaller, which broadens the fields of application for wireless sensor devices. With these achievements and progress of techniques, WSN is becoming a more and more popular research topic in different area.

1.2 Related issues for WSN applications

For a real-world WSN application, every sensor node in the wireless network must be able to be reached and act as expected steadily. As we know, many issues still exist in practice. Generally speaking, several important subjects should be considered before implementation:

- Multi-hop routing strategy, which guarantees every node in the network can be communicated with.
- Sensor data management, consisting of data storage and collection.
- Time synchronization of network, which synchronize the time of all the nodes in the network.
- Energy consumption. For example, developer should take into account the battery life when using battery as power supply.

Although there are several existing protocols or implementations for these subjects, for different cases, these solutions have to be discussed in detail to meet the specific requirement of the applications, some of which even have to be rewritten or tailored to be compatible with other modules.

In addition, from the angle of embedded systems, other issues such as code size, RAM usage, portability and reliability have to be taken into account. A stable system with few bugs is the key to run programs safely for a long period.

So developing a real-world application for WSN is always complicated and time-consuming. Even making experiments for verifying protocols or theoretical results of WSN is not easy to conduct. Though there are some simulators that can be utilized to analyze network behavior, the simulation-based results still have a distance from reality.

1.3 Motivation of this thesis

Over the past few years, researchers usually setup the WSN environment by themselves before doing their experiments. As we talked above, besides their research experiments, they have to think of many trivia which are always time consuming, for example, the way to collect data from sensors. Usually, these configurations and programs can not be reused and have to be abandoned after experiments.

In addition, to develop an application for wireless sensors always need to reprogram the sensor nodes iteratively with the updated program. So as traditional way, user has to collect sensor nodes from their deployed places, reprogram them manually and deploy them again, all of which are tedious and time consuming.

So it’s necessary to build up a WSN testbed to lighten the burden for researchers.

Nowadays, there are already some WSN testbeds developed for solving the problems we discussed above, to which we will give an introduction in next chapter. However, these testbeds always lead to a high expense because of their extra devices. And some of them quite rely on underlying infrastructure such as Ethernet.

In order to improve the testbeds with a new scope, we would like to design and implement a WSN testbed with a high flexibility and cost-performance ratio, which also contains the necessary components for WSN experiments.

1.4 Thesis Objective

According to our motivation, our objective is to implement such a high flexibility WSN testbed that can help user to conduct experiments easily.

With reference to the existing testbeds, we would like to show another idea to build up a WSN testbed in this thesis, putting the emphasis on the flexibility and cost. However, reducing the cost of a testbed always affect its robustness. So there is a tradeoff between cost and reliability. What we want to achieve is designing a high flexibility and low-cost WSN testbed with an acceptable reliability which may contribute to the research of WSN.

In our plan, we are going to achieve our objective by following steps:

- Determine basic functions for the testbed according to our objective.
- Review other existing solutions, and discuss their advantages the disadvantages, then give out our solution with the improvement on their deficiencies such as cost and mobility.
- Implement our solution with selected devices and programming language. Make the required components to be compatible with each other.
- Make GUI tools for testbed at PC-end to provide a friendly interface to users.
- Testing and debugging with sufficient test cases, verify the whole system with its characteristics, including functionality and stability.

As to our expectation, our testbed can be reprogrammed remotely with a wireless approach. The testbed controlled by PC-end with different commands can collect data from all sensor nodes easily and report failure or abnormity to users in time. And a service of time synchronization is also provided to users.

1.5 Terminology

Some special vocabulary we used in this thesis is introduced below.

- Basestation: node with extra functions that play a role of gateway between

PC-end and wireless sensor nodes. It receives instructions from PC via serial link and sends them to sensor nodes over wireless connection; it also deals with feedback from sensor nodes and sends them back to PC via serial link.

- Previous node: the node which is in charge of current sensor node. A sensor node receives instructions from its previous node and also feedback information to PC through its previous node.
- Node list: a list structure exists in every node that maintains information of sensor nodes. It is used for multicasting instructions.

1.6 Thesis Outline

The rest of the thesis is organized as follows:

In Chapter 2, an introduction to existing WSN testbeds is given with a discussion on their good approaches and also their shortages.

In Chapter 3, we proposes our solution in comparison with the testbeds which are introduced in Chapter 2, including the description on both hardware and software parts.

Chapter 4, implementation details of our testbed are given. Improvements are proposed to solve the problems which are found during the implementation procedure. In addition, test cases are made to verify the whole system with consideration of reliability.

Chapter 5 draws the conclusions of this thesis and proposes some future work.

Chapter 2. Existing solutions for WSN testbed

Real experiments play an important role in the research of WSN. Compared with simulations, experiments give a better view of how a protocol or an algorithm performs in real life. Since some parameters such as resource usage, link quality fluctuation and energy consumption can not be simulated with all details. But unlike simulations, the implementation and debugging of WSN applications require frequent reprogramming of nodes with new code. Therefore, in a fixed testbed, it is not practical that a user frequently goes to the location of dozens of nodes, connects them and reprograms them through wire. So reprogramming sensor nodes remotely is a necessary feature for a good testbed. Moreover, a developer must take into account the total expense of testbed, making a fitting testbed in his budget.

So far, some WSN testbeds have been designed for different kinds of purposes. These testbeds use different device, software and have differet system architectures. They can be applied in different application areas. Users can utilize these testbeds to setup WSN experiments quickly. In this chapter, I will describe two testbeds, which are designed with motivations close to ours introduced in the last chapter, in details.

2.1 MoteLab

MoteLab[1], a Ethernet-based sensor network testbed, developed by Harvard University. It consists 30 MicaZ[2] motes connected by Ethernet and a central server. User can access the central server via web-based interface to upload experimental code or download experimental results. This testbed has been deployed in Harvard for research and teaching.

2.1.1 Elements of MoteLab

MoteLab consists a fixed array of wireless sensor network nodes. MicaZ is chosen as sensor node which radio is IEEE802.15.4 compliant. The nodes are connected by Ethernet. MoteLab uses two kinds of Ethernet interface boards: EPRB[3] and Crossbow MIB-600[4]. Both of the boards provide interface to reprogramming and data logging through Ethernet.

The central server is responsible for data storage, reprogramming nodes and scheduling tasks for nodes. They use MySQL database to store useful information for testbed operations such as log data from sensor nodes, user account information with access priority, schedule information and other related running states. It also provides a Web interface for user which is represented as PHP-generated pages. All the

operations such as job creation, scheduling, and data collection can be issued through this interface by users from remote terminals. The whole system is shown in Figure 2.1.

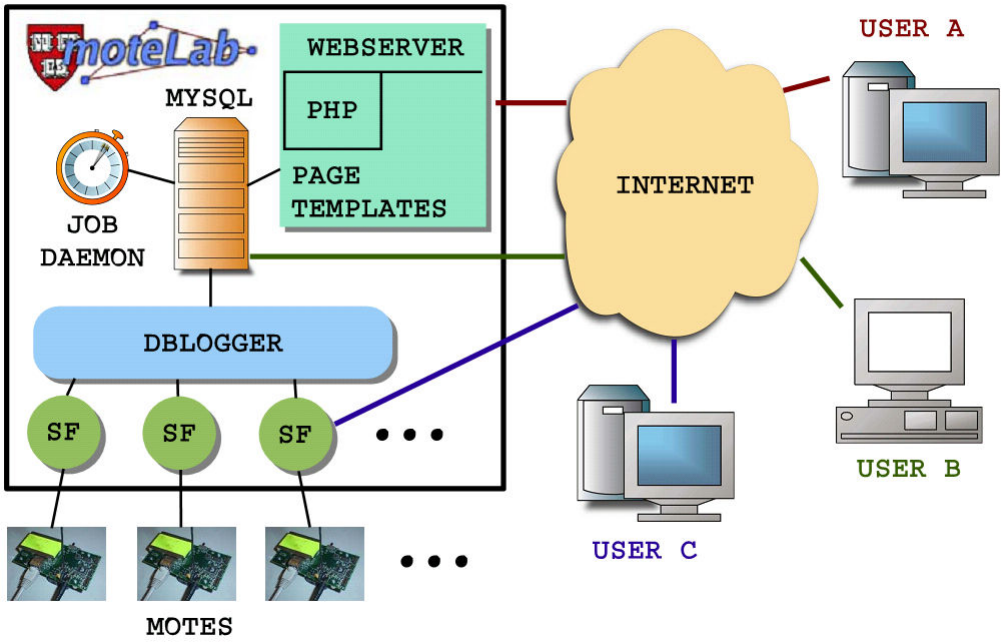


Figure 2.1 MoteLab System Architecture[1], SF is abbr. of SerialForwarder which is a tool of TinyOS[5]. It acts as a TCP multiplexer with which user can directly access serial port of each node via TCP connection.

Since all of the devices are linked by Ethernet, the testbed needs a suit of tools to manage the whole system. The central server operates in Linux with Apache, MySQL and PHP on which several software runs. As we see in Figure 2.1, there are three key components:

- DBLogger: a java programme connected with each sensor node. It translates data message from sensors and insert useful segments into database with node id, received time and global sequence number.
- Job Daemon: a Perl script that handles experiments initialization and disposal such as reprogramming nodes, starting or terminating related components(e.g. DBLogger), and providing a formatted log data for user to download.
- Web interface: the pages are generated by PHP dynamically. When user login with his own account, he can maintain his account information, create/modify job, upload his experiment image to server and download logging data.

Beside these, MoteLab has an extra feature of power measurement, which can help user to monitor energy consumption of WSN jobs.

Depending on these components, MoteLab performs well as a mature system and

allows remote users to create a WSN experiment conveniently.

2.1.2 Summary of MoteLab

In short, MoteLab testbed is an Ethernet-based solution to manage images distribution, data logging and collection. All controls and data transmissions are undertaken by Ethernet. Sensor nodes in this testbed only have to run users' programs, transmit or receive packets with their wireless transceiver and submit the experimental data to the central server. The central server takes over most of the works for the whole system, serving multi users and managing sensor nodes with all sorts of software.

2.2 EmStar

EmStar[6] is a software framework for WSN experiments. It provides sufficient tools for simulation and also includes an array of real sensors deployed on the ceiling of the CENS systems laboratory. An impressive feature of EmStar is its indefinite boundary between simulation and real life experiment, with which an experimental program running with simulated models can also run well with real sensors.

2.2.1 Elements of EmStar

Unlike MoteLab that uses real devices for experiments, EmStar is developed as a PC-based software framework that emphasizes its outstanding capability of simulation.

EmStar provides three different modes for WSN experiments: pure simulation, real life experiment with deployed devices, and a hybrid mode that combines simulation with real wireless communication and sensors. Because of our motivation, simulation mode is out the scope of the thesis.

For the real life experiment part, there is an array of 54 Crossbow Mica motes on the ceiling of lab, which are all mains powered and have a serial-port connection to a central simulation machine. Similar to the real-world mode, the hybrid mode also utilize real wireless links. The advantage of the hybrid mode is that a user can use a rich set of debugging and visualization tools, which improves the system visibility from the experiment mode.

In order to work in different modes, two main tools called EmSim and EmCee are provided:

- EmSim is a pure simulation environment. In the simulations, many virtual nodes are run in parallel, interacting with simulated modules and radio channel.
- EmCee is a modified version of EmSim that provides an interface to use real

radio transceivers to communicate. It's easy to switch the mode by typing EmSim or EmCee when user starts a program. Because a same program can run well with both tools without any change.

2.2.2 Summary of EmStar

EmStar is a simulation-driven testbed that provides plenty tools for simulation and debugging. With its simulation mode user can test and debug code to check its logic and consistency while real life experiment can be realized by running the communication part in the real wireless links between the nodes in the array on the ceiling. Each mode covers the shortage of the other, so user has multi-options to write and debug WSN applications.

2.3 Other testbeds

In Section 2.1 and 2.2, we introduced two solutions for WSN testbed, which inspires us on designing our testbed. Beside these two solutions, there are still dozens of solutions in use. Some of them are listed below.

WSNTB[7], similar to MoteLab, also manages reprogramming and data logging through Ethernet and provides remote control to user by web-base interface. An extra feature proposed by the authors is its "local mode" which allows users to run their programs locally with testbed resource. The testbed includes 34 sensor nodes, three gateways and three servers while software framework provides many functions such as compiler, online editor, data logger, working status monitor, job queue system, and etc.

In [8], TU Berlin proposed another WSN testbed TWIST which also uses Ethernet to connect WSN nodes. It supports flat and hierarchical sensor networks with basic services such as node configuration, network programming, application data gathering, active power control, and etc. This testbed consists of 102 Tmote Sky nodes and 102 eyesIFX nodes which are deployed in three floors in a building of TU Berlin. Users can use the system through a web interface.

2.4 Discussion

The goal of our solution is to implement a testbed prototype with reliability, low cost and compact software framework. We want to acquire some good characteristics from existing designs. As we see, solutions to WSN testbed vary with developers' purposes, from software to hardware. In this subsection, we will summarize and discuss characteristic of the testbeds introduced.

As we see in Table 2.1, all of these four testbeds need additional devices such as Ethernet interface boards, standard/micro server, and link converters. When we check the purpose of these devices, we find that most of these devices are deployed for

connecting the sensor nodes by Ethernet connection. Nearly all of the solutions use Ethernet connection to transmit new programs, log data and control message. So far it seems that these backbone devices are requisite for remote reprogramming.

These extra devices can be divided into two categories: One is peripheral board which usually acts as interface of reprogramming, disseminating control messages and collecting experimental results between sensor nodes and central servers. E.g., MIB-600 provides Ethernet connectivity to IRIS/MICAz/MICA2 family of motes. It offers two separate TCP ports: one for in-system programming and one for routine data communication like serial login, telnet login and security verification. The other category is micro-server which plays a role of super node in network, linked with sensor nodes via wired cable. E.g., Compaq iPAQ 3760, a handheld PDA having a strong ARM processor, 64MB RAM and 32MB FLASH. It can handle more works than the connected sensor node with its large process capability and memory size, which reduce the work load for main server.

With support of these devices, the testbeds gain extra benefit: utilizing fixed Ethernet as network backbone to transmit data, many mature protocols for Ethernet can be applied so that developers can save much time on protocols design and validation. What's more, as hardware backup, these devices which have low possibility to crash can detect and recover abnormal nodes when some fatal failure occurred on these nodes' system. So apparently the reliability of these solutions is higher than the others without these devices.

However, devices in both categories introduce extra cost on hardware purchase. Especially for some customized peripheral board, i.e., each MIB-600 costs $300. If a WSN testbed contains 30 sensor nodes and uses MIB-600 for each node, it means user have to pay 300*30=9000 dollars extra.

Item / Testbed	Sensor node	Additional devices
MoteLab	MICAz	EPRB[3], MIB-600[4]
EmStar	MICA	Compaq iPAQ 3760
WSNTB	Octopus I Octopus II	Single-board computers net4501&net4801, RS232-to-Ethernet & USB-to-Ethernet convertors
TWIST	Tmote Sky EyesIFX	Linksys USB2HUB4 hub, Network Storage Link for USB2.0 (NSLU2) device

Table 2.1. Hardware of four testbeds

In addition to extra devices, these solutions also rely on Ethernet connection. It is not

a big problem when we set up the testbeds indoor, e.g. in the modern buildings, the fixed network infrastructure can be utilized to form a LAN. However, when we want to deploy one of those testbeds outdoors, the difficulty emerges that we have to take additional workload such as connecting devices by cable and arranging spaces for housing, to constitute our local Ethernet. So in other words, mobility is one of the challenges for these solutions.

With advances in WSN technology, nowadays some sensor boards are competent for super nodes with their rich on-board hardware such as strong processor, large memory and programming space, and plenty peripheral components. It can fulfill the requirements of routine communication and management. So it's possible that some functions can be migrated from standard/micro server to these boards, which means some of those extra devices are not necessary anymore. The challenge we confront is the reliability of the WSN systems: without these extra devices, the whole WSN should still run steadily as expected.

As a result, we are not satisfied with these existing testbeds since no one fulfills the requirement of our objective which is aiming at design of a testbed with high flexibility and cost-performance ratio. Nevertheless, we appreciate some good ideas of these solutions which may benefit our own design proposed in next chapter.

Chapter 3. Our Solution for WSN testbed

We have discussed several solutions for WSN testbed in the last chapter. According to our objectives, these solutions cannot meet all of our requirements such as good flexibility and low cost. So in this chapter, we would like to propose our own solution with high flexibility and a reasonable cost.

We explain the main idea of our solution in this chapter. The content is organized as follows: in Section 3.1, we describe our solution in three aspects: hardware and architecture, operating system and software framework. In Section 3.2, we discuss the solution with respect to our objective and compare it with other solutions.

3.1 Overview of our solution

With reference to other solutions, we decide to design a new solution for our testbed in order to fully meet our objectives. To explain the characteristics of our testbed, I will give an introduction in different aspects in the following subsections.

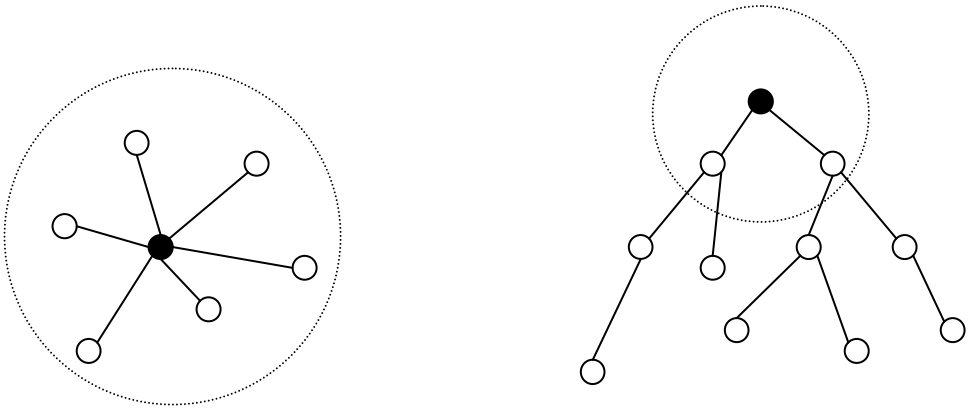
3.1.1 Hardware and Architecture

We choose Tmote Sky[9] as our wireless sensor motes due to its rich on-board modules, reliable quality and acceptable price. The mote complies with IEEE802.15.4 standard. It provides USB interface through which a mote can be programmed. The mote equips an 8MHz Texas Instruments MSP430 microcontroller with 10KB RAM and 48KB Programming Flash. Optional accessories include integrated Temperature, Humidity and Light sensors. The transceiver has a data rate of 250kbps at 2.4GHz and the transmission range is 50m indoor and 125m outdoor.

Figure 3.1. Our wireless sensor node: Tmote Sky.

In order to increase the flexibility of the testbed and reduce the cost of whole system, we plan to build up our network backbone by wireless connection. Thanks to the good capability of Tmote sky board, no extra device is necessary for our testbed except a PC as the root server.

The PC server connected with the basestation node runs our software to manage the whole WSN. User can perform all the operations through the user interface, such as topology configuration, distribution of new programs, management of the experiment and data collection.



(a) star-based topology (b) tree-based topology

Figure 3.2. possible topologies of the system

The topology of our WSN testbed can be configured as tree or star like Figure 3.2 shows. Physically, we deploy our sensor nodes in the different rooms on the 19th floor of EWI building at TU Delft. Tree-based topology is the better option since all of the nodes can be reached over one or several hops. However, we can change the logic topology of the network by our software which will be described in later subsections.

3.1.2 Operating system

An operating system is a good choice to be added for managing different tasks for embedded systems. In our design, nodes have to handle all different kinds of operations, such as linking high-level APIs to low-level hardware interface, scheduling tasks with different priorities and allocating memory. So we would like to introduce an operating system in our development.

TinyOS[5] is an open-source operating system designed for wireless embedded sensor networks. It is a popular system for wireless network which supports several wireless platforms such as micaZ, Tmote Sky, IntelMote2 and Eyes. Dozens of groups and organizations use TinyOS as the software platform to develop WSN protocols and contribute to the platform actively. Many WSN testbeds use TinyOS as their operating systems, such as MoteLab, which we introduced in the last chapter. Since wireless embedded device always be used to handle specified works, its programs on these devices are well-defined and customized. Therefore, TinyOS decides the complete call graph at compile time, avoiding dynamic allocation or reloading components in runtime, which optimizes code efficiency and decreases complexity of compiling.

TinyOS adopts event-driven style and components-based programming method. The programming language for TinyOS is nesC, a kind of C dialects with extra features, which has a different coding style and approaches. However, it is not difficult for users to adapt themselves to nesC if he has already had sufficient knowledge of C or Java.

In our project, I select newest version TinyOS 2.1 as the operating system for nodes, which incorporates new features and useful modules, such as FTSP synchronization module. We use this module to carry out synchronization in our software framework.

3.1.3 Software framework

In our solution, the testbed consists of three main modules and one service:

- command dissemination and feedback,
- Remote reprogramming,
- Data logging and collection,
- Time synchronization.

The Tmote Sky motes are logically divided into two categories by our software framework: the basestation and the sensor nodes. The basestation acts as a gateway between PC and other sensor nodes while sensor nodes run users' programs and log sensor data for research.

Communication between the basestation and other nodes is wireless. So a node

cannot be reached if it is out of the transmission range of the basestation. In order to ensure every node in our testbed can be reached we need to organize the network into a topology to support multi-hop transmission. Spinning tree topology is a nice option for our solution, because it is straightforward for disseminating commands by flooding and collecting feedbacks with little overhead. Meanwhile, it is good for network extension.

Other testbeds using Ethernet as their backbones do not have to consider the networking functions like routing and flooding because the underlying Ethernet protocols can handle these. However, we have to make a wireless flooding solution for our testbed.

Thanks to tree topology, routing is easy to be carried out by maintaining a configurable node list in every node which provides the data stream path for every node. Our software provides an interface for configuring the node list remotely. So every node can be either configured as a “Branch” or a “Leaf” of the tree as showed in Figure 3.3. When user floods an instruction to every node, the basestation or branch node will only distribute the instruction to the nodes which are in its nodes list. And when a node receives an instruction, it returns an acknowledgement message to its upstream node. By this acknowledgement, a node can confirm the link with its neighboring nodes. If a node doesn’t response to an instruction after many retries, its upstream node will report the failure to the basestation, then user can diagnose the problem with the failure report. Feedback is a mechanism to report internal error of system or network situation to user that makes the whole system to be diagnosable. The transmission path for feedback is the reverse of flooding path. That’s the basic idea of our flooding and feedback module.

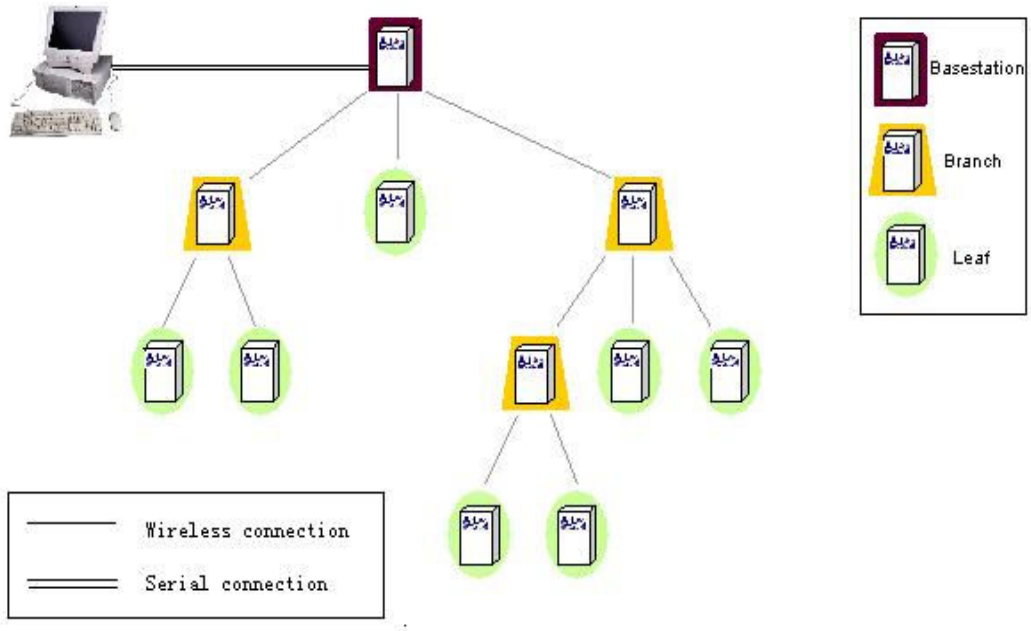


Figure 3.3. Hierarchy architecture of our testbed. Theoretically, the number of intermediate Branch is unlimited. In our deployment, two or three Branches are enough to cover all the nodes of our testbed.

Though the manual configuration looks a little cumbersome, it gives much flexibility for changing the wireless network topology. There are three advantages: first it is easy to implement with a little footprint, secondly it is clear to draw the exact topology of whole WSN by user's configuration. Thirdly it is easy to trace the feedbacks of abnormality through the well-defined paths. However, to implement a routing protocol will be a possible future work thus the routes between nodes can be automatically configured.

Deluge[10] is an existing solution for remote reprogramming, which is released with TinyOS 2.x. Deluge provides a reliable way to disseminate program images then nodes can be reboot to these images. It uses an epidemic way to propagate the program image. Nodes broadcast advertisement periodically. Each advertisement contains a version number and a vector which indicates the complete pages of version the advertiser has received. When a node notices that it needs to update some parts of its old program from an advertisement, it will send a request to the advertiser to ask for the needed packets. After completing a page, the node starts to broadcast an advertisement to other nodes too. This mechanism helps all the nodes in the network update to the same new version of program. It's suitable to be our reprogramming module as it also support multi-hops networks. However, the way of propagation decides that the basestation can not know whether every node has received the program image completely. So our software tries to complement Deluge by adding confirmation function. And the propagation of deluge may affect our experiments, because nodes keep broadcasting advertisement periodically even all expected nodes has received the new version. So we need to add an interface to stop it before running our experiments.

We add a wrapper on Deluge, which contains management interface. By this interface, user can be notified as soon as a node has completely received a new image. So it's easy for user to find which nodes have not received the new image and check if there is something wrong. In addition, user can turn on or off the propagation function of Deluge by the interface. By this extra feature, reprogramming different nodes with different program in the network becomes possible. For example, if user wants to reprogram node A with version 1 and node B with version 2, the approach is: first turn off the propagation function of node B and start deluge to send version 1 to node A. When it is done, turn off the propagation function of node A and turn on for node B, then send version 2 to node B. Finally, turn off the propagation function for both. Now, node A is running version 1 and node B is running version 2.

Without extra devices, the experimental data have to be temporally stored on the wireless sensor boards before being collected. There is a 1 MB external flash on Tmote Sky which can be used as the storage space. TinyOS provides storage interfaces to read and write the external flash. So operation of logging data can be done with modification of original interfaces.

Beside storage, gathering data from nodes have to be implemented. There are some protocols implemented for data collection. But most of them designed for all-purpose have a large code size. So we decide to give our light protocol to collect data

according to the architecture of our testbed. It's a simple yet useful way: when a sensor node receives the collection command by command collection module, the collection procedure starts. A piece of data is read out in the user-defined structure and stamped with a sequence number to be sent. The message is transmitted to the basestation via the reverse path of flooding hop by hop, then the basestation sends it to the PC via serial link. Due to the packet loss in wireless link, the basestation should acknowledge the messages which it has received. The node that sent the message should repeat until its last message is acknowledged. The drawback of this retransmission mechanism is that redundant messages may be received by the basestation. The sequence number in every message can identify duplicated packets. Theoretically, all the log data should be collected by PC without loss.

Time synchronization is a service we provide with the testbed. The requirements, such as calculating delay or time synchronization, in some experiments drive us to decide that this module is a must-have. There are many wireless synchronization protocols proposed for WSN, such as Reference Broadcast Synchronization (RBS) [11], Timing-sync Protocol for Sensor Network(TPSN) [12] and Flooding Time Synchronization Protocol (FTSP) [13]. We choose the FTSP module as our synchronization module because it has already implemented in the TinyOS package and shows promising performance. The principle of FTSP is utilizing a single radio message which is time-stamped at both the sender and the receiver to synchronize the time of sender and receiver. FTSP also supports synchronization across multi hops. So the whole network can be synchronized to a global time which is the local clock of the node with the smallest ID in the network. So in our testbed, if we want to synchronize all the nodes to the basestation, we assign the smallest node ID to the basestation. After repeating tests on this module, we have decided to include it which suits our testbed well.

The software framework for sensor board is made up of these modules and service. Because of the limited programming space of Tmote Sky which is 48KB, removing unnecessary modules for some experiments can save the programming space for user's program. In order to use the programming space efficiently, modules of collection and time synchronization can be included or excluded by user's configuration at compile time.

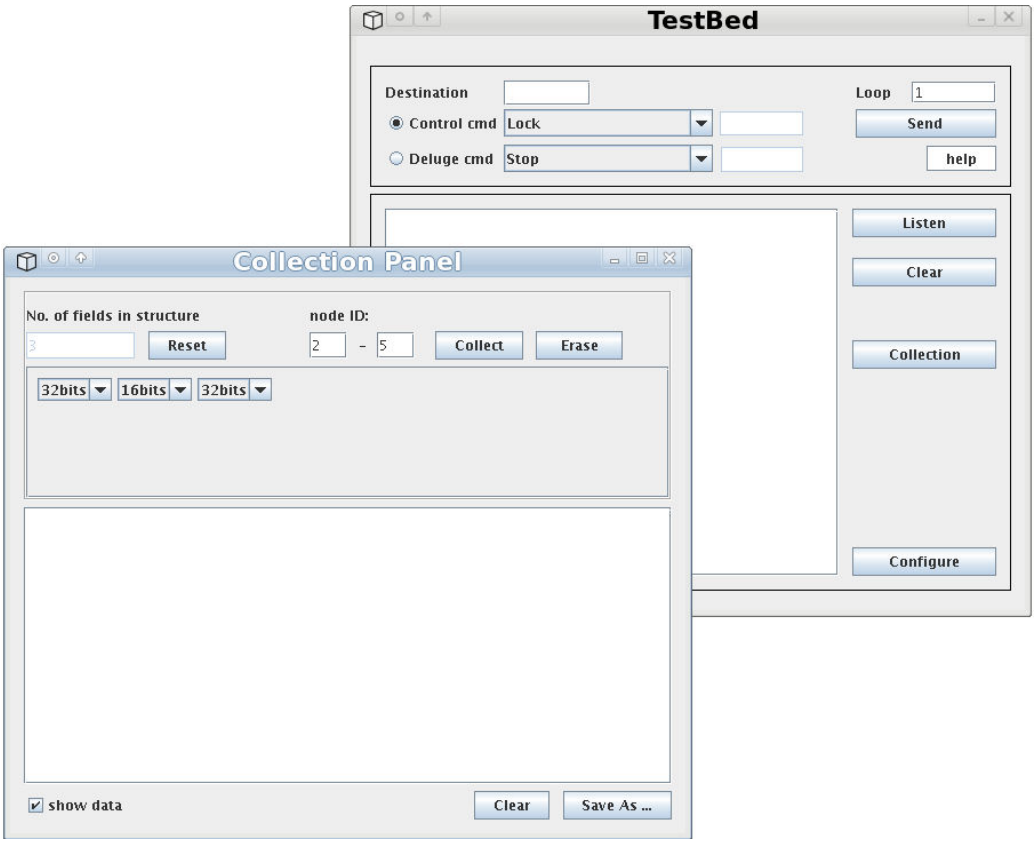


Figure 3.4. Our GUI software on PC-end is written in Java with NetBeans IDE, including many tools for operating our testbed, such as instruction sender, feedback receiver and translator, data collection and conversion tool.

In addition to the software framework for sensor board, for the PC, we have designed a suit of GUI tools written in Java for users' convenience, as shown in Figure 3.4. With these tools, a user can perform all the operations by clicking a button instead of typing commands in a console. And the control tool can translate the feedback information to be straightforward, explaining received binary segments for users with clear words on screen. Furthermore, a tool for collecting data can gather data from different nodes in batch and save the data as a formatted text file.

3.2 Discussion of our solution

As we discussed in the last subsection in Chapter 2, those solutions have their advantages and disadvantages. Our solution is to improve the deficiencies of these solutions, which do not meet our objectives. Comparing with those solutions, our solution has the following differences.

Firstly, our testbed do not depend on the Ethernet equipments. The Tmote sky motes act as both experimental devices and experiment management devices. Any PC or handheld equipments which have Linux OS and Java Runtime Environment can be used as the root server of our testbed. So the expense for hardware devices is reduced, which meets one of the requirements of our objective. In addition, the application

field is broadened since we do not depend on Ethernet any more. So we can deploy the testbed in different topologies in most places where we are going to run experiments with great flexibility. Moreover, when we want to add more sensor nodes into our testbed, we can easily power up the nodes and allocate them anywhere within the radio coverage of our testbed.

Secondly, without support of extra devices and Ethernet, our software framework based on wireless communication is much different from other solutions. The testbeds we introduced before, such as MoteLab, let the sensor nodes run experimental programs only, the remaining works can be handled by extra devices with their software. The wireless transceiver on a sensor node is used for experiments only and the generated data are submitted to a center server immediately via extra devices. However, in our solution, the sensor nodes are not only used for running experimental program, but also have to deal with networking things like flooding instruction, sending and receiving program binaries. And the experimental data are stored on the local flash and sent to the server when received the certain instruction. As we see, the difference is obvious, but we cannot say they are the improvements for Ethernet-based solutions. Our software framework is just designed for the WSN testbed without Ethernet backbone. What we want to show is, with no Ethernet support, the testbed can be still competent for conducting different WSN experiments, and of course, the expense of our testbed is quite lower than other Ethernet-based testbeds.

So far as we see, the solution we present above can fully meet our requirements. However, with the reduced cost and devices, the testbed also has some drawbacks.

Due to the fact that our software framework is compiled with user's experimental program, so user has less programming space for experimental codes since the software framework has taken up some space. That's one of the reasons that we make the software framework to be module-customizable. The external flash for storing experimental data also has the similar problem. Because the experimental data are saved in the local flash with size of 1MB, it may be overflow when a program generating a large quantity of data runs for a long time.

As we mentioned above, user's program runs with the software framework, so allocation of the shared resource is a problem we have to figure out. E.g., both the software framework and user's program use the same wireless transceiver of the node, so we cannot send the experimental data to root server as soon as it is generated, or it may affect the running program of users.

Removing the backbone hardware also affect the reliability of the testbed. E.g., the micro server of some testbeds also plays a role of backup system. When the system of a sensor node cashed with fatal failure, the micro server which is connected by cable can detect the abnormality and recover it as soon as possible. So from this angle, our solution is not as reliable as those which are equipped extra devices.

Nevertheless, our solution gives a new idea for implementing a WSN testbed with

low cost and high flexibility. We have implemented the first version of the testbed. Details of our implementation will be given in the next chapter.

Chapter 4. Implementation of the Software Framework

In the previous chapter, we proposed our solution of WSN testbed which contains both hardware and software components, focuses on flexibility and cost. We choose Tmote Sky as our wireless sensor board and TinyOS as our operating system for sensor board. The software framework which contains several modules was also introduced briefly.

However, due to some real-world restriction, our software framework needs some slight modifications in practice. In this chapter, I will present implementation details of the modules.

The content of this chapter is organized as follows: the first four sections introduce the implementation of the three main modules and the service: command dissemination and feedback, remote reprogramming, data logging and collection, and time synchronization service respectively. In Section 5, optimizations are introduced with related modification works for our original design. Then we give further illustrations of the frame in the form of two tutorials in Section 6. Performance of our testbed is measured in different test cases in last section.

4.1 Command Dissemination and Feedback

This module not only sends instructions from base station to every other node, but also ensures those nodes have received the instructions and reports failure/error information when nodes do not act as expected.

4.1.1 Dissemination

Before introducing implementation details for this module, we explain some terminologies. The logic topology of our testbed is a tree. As we see in Figure 3.2, we defined the basestation as the root of a network. Node A, which relays messages to node B, is called the branch of node B. Node B is the leaf of node A. Please note that node B can also be a branch if it has leaves.

The routing strategy is straightforward in this module. Messages are disseminated from the root (basestation) to branches and then to their leaves. So first of all we have to organize nodes to form a tree topology. Every node in the network must know its branch and its leaves if it has any.

To make it easy to implement and control, every node maintains a node list shown in

Figure 4.1(b) which keeps the information about its leaves nodes. The variable "count" in a node_list structure keeps the number of leaves. The structure named node_state is made up of a unique id and a linkstate variable as shown in Figure 4.1(a). The variable "id" in the structure keeps an 8-bit node id assigned uniquely by TinyOS. The "linkstate" is used during the instruction disseminations which I will introduce with dissemination mechanism later.

The capacity of a node_list is defined by pre-defined parameter NODES_CAPACITY. The node list will initialize an array of nodes_state with a pre-defined length. Since the TinyOS doesn't support dynamic allocation of memory, we use pre-defined capacity of node list here. User can change the capacity of node list by modifying the NODES_CAPACITY in a header file named "MyDeluge.h".

Beside the list, a variable named "prevNode" is used to save the id of previous node. With the node list and prevNode, a node can be linked to the network. The "bootKeeper" is a 16-bit non-volatile flag that keeps useful states for reboot.

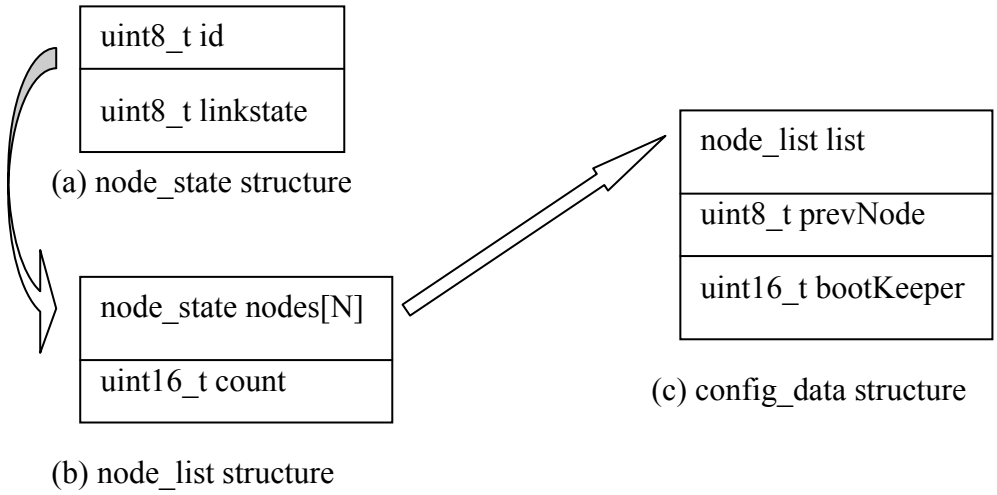


Figure 4.1. Node list and Configuration structure, uint8_t is unsigned 8-bit integer and uint16_t is unsigned 16-bit integer.

In order to define all different kinds of instructions, we have set a command structure as shown in Figure 4.2, all control instructions as well as feedback messages use the same structure. And for different types of commands and feedbacks, we also defined two arrays of constants, the prefix CMD_ signifies that this constant is used for a command type while the prefix ACK_ means the constant is for a feedback. e.g., CMD_COLLECT is the instruction to start data collection while ACK_COLLECTION is the feedback to acknowledge the former instruction. In command structure, there are 5 fields. "cmd" is the variable to give the type of instructions or feedbacks. "data" is used to store parameters. Since some instructions and feedbacks need parameters, e.g. command to add a node to node list needs a parameter to indicate the node id to be added. "uidhash" is a 32-bit variable which is

used for feedbacks in two ways. For image feedback which feedback image information to user, it's used to store the uidhash which is a 32-bit unique number for program image; for other feedbacks, it keeps a 32-bit error flag for diagnosing the running situation of system. The flag is a global variable of our software framework that each bit stands for one kind of error, so it is possible to report 32 kinds of error. Last two fields "src" and "des" keep the original source id and the destination id of a message. If a message is for every node in the network, the "des" field is set to 0xff.

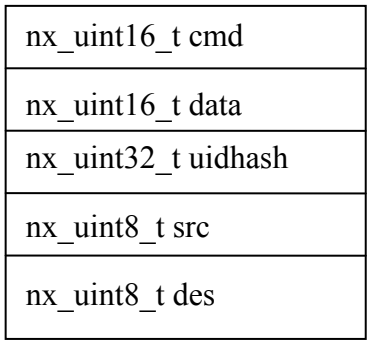


Figure 4.2. Command structure, the nx_ prefix is specific to the nesC language and signifies that the uint8_t, uint16_t, uint32_t are external types

There are two types of dissemination: first is one-to-one mode that commands are transferred from basestation to destination node. And the other is one-to-all mode that commands are distributed to every node in the network. For each node, the field "des" of command is used to distinguish between the two modes. In the first mode, a node only executes received command if the "des" field is same as its own id, otherwise the node retransmits the command to its leaves. If the "des" field of command is 0xff, the node not only retransmits the command to its leaves but also executes the command as well. However, each node keeps the previous and next nodes that it can connect, which means that the basestation do not know the actual path to the destination node. So, both modes deliver commands to every node in the network, the difference is that in the latter one a node needs to check every node's feedback to ensure all of the nodes have received the command while in the former one, it only has to know if the destination node acknowledged. Details will be given in subsection 4.1.3.

4.1.2 Feedback

Feedback is a mechanism to confirm if a message is delivered successfully, gather useful information from nodes and diagnose potential network problems for user. In this module, we apply feedback mechanism in several places such as command acknowledgement, image disseminations acknowledgement, and error reporting. The feedback can be divided into two types: passive feedback and active feedback. As the name implies, the passive feedback is activated when it has been requested, e.g. feedback of checking version function, the node which has received the request sends back a feedback with version information. And for active feedback, the feedbacks

will be returned automatically when a specific event happens without being requested, e.g. sending a feedback with image information will be triggered after a node has completely received a new image.

For the convenience of description, we divide feedbacks into four categories by their functions:

- For command dissemination. The feedbacks confirm the commands are received by destination nodes.
- For data collection. The feedback confirms the start of collection and returns initialization information for PC end.
- For error reporting. The feedbacks report different kinds of errors to user for diagnosis.
- For confirmation of received image. The feedbacks report which nodes have received a new image.

I am going to explain the first one for this module and I will give the details of remains in other sections for other modules.

4.1.3 Protocol of Dissemination and Feedback

I take the one-to-all mode dissemination for an example. The dissemination is carried out in following steps: user send out a command in which “des” field is set as 0xff. The basestation receives the command via serial link and check if its node list contains any nodes. If so, the command is sent to these nodes one by one in a unicast way. The reason of using unicast is that broadcast will arouse problems such as broadcast storm and feedback collision, and multicast has not been implemented yet in TinyOS. Compared with broadcast, unicast avoids all receiver to acknowledge at the same time thus decreases the possibility of feedback collision. The drawback is it takes more time and energy to send command to multi nodes. As we mentioned above, each node state contains a field of “linkstate”. Before starting a dissemination, this flag for each node is initialized to “NO_RESPOND” state. When one of the leaves has received the command, it returns an acknowledgement feedback to the basestation. After receiving the acknowledgement, the basestation sets the “linkstate” of corresponding node as “ACKed”. Every time the basestation receives an acknowledgement, it switch the corresponding “linkstate” and checks if the “linkstate” of all the nodes have been set to “ACKed”. If so, the basestation send a feedback to PC to tell the user that every node has received the command. Since the sending and receiving events are asynchronous, there is a timeout timer start as soon as the command has been sent to all the leaves, which is used to check if all the leaves have received the command before timeout. When the timer expires and there are still some nodes haven’t acknowledged the command, the basestation report the nodes id of which did not acknowledge to PC. In late modification, we add in retry mechanism

before reporting to ensure whether the nodes cannot receive the command or just missed the command due to collision.

In practice, there may be two or three hops between the basestation and outermost leaves. Then the protocol is a little more complex than above. The outermost leaves acknowledge the command which is received from branches. Branches and the basestation, when receive a command, they follow the steps as above. The slight change is that they return an acknowledgement to previous node only after all of its leaves have received the command and acknowledged. If there are still some nodes haven't respond to the command after timeout, the basestation or branch will send the report to user too. As we see in Figure 4.3, the dissemination order is from 1 to 3, and the order for feedback is from a to c.

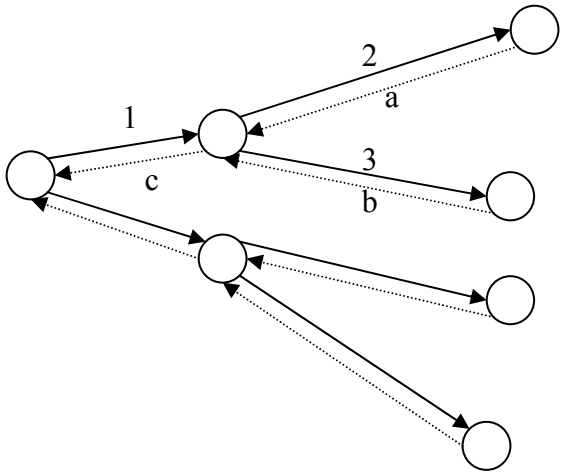


Figure 4.3. Dissemination and feedback steps are shown with order.

Thanks to the feedback mechanism, dissemination of command becomes reliable and easy to track. User can know whether there are failed nodes which do not respond to commands, and by feedback information, user can also know which nodes are out of order and the route to these nodes, both of which can help user find and fix the problems.

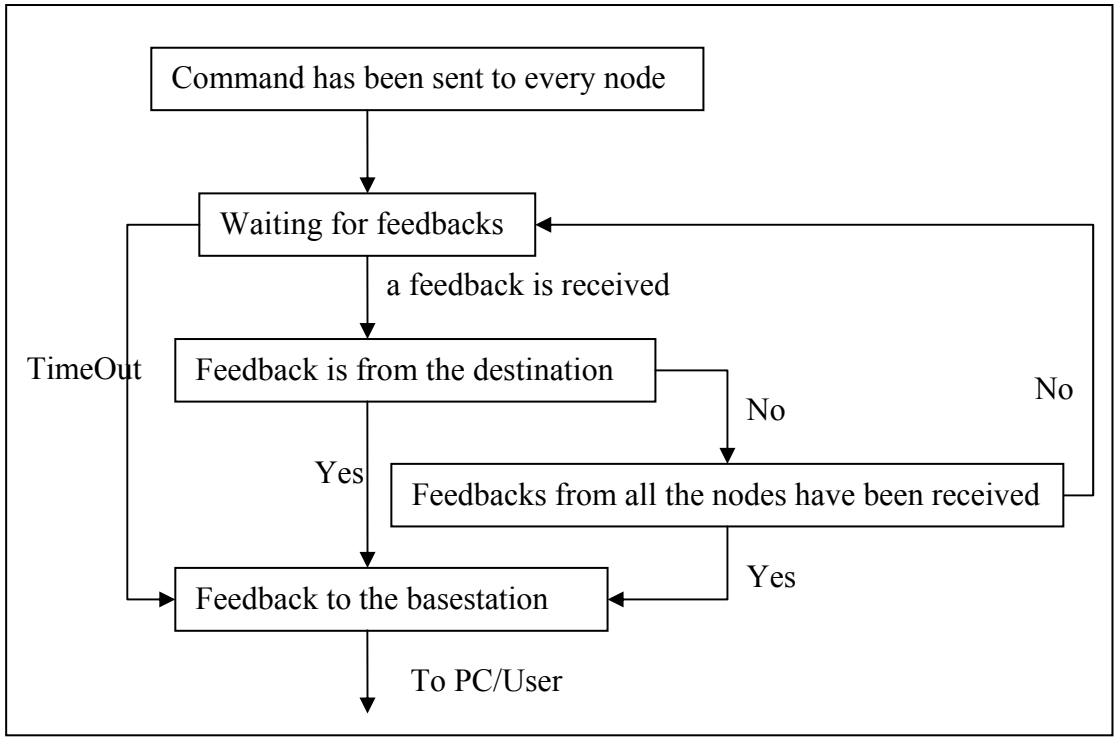


Figure 4.4. Flowchart for the feedback of one-to-one mode.

In the one-to-one mode, the dissemination is the same, but the feedback is simplified. During the dissemination, if the basestation or branches have received the acknowledgement from destination node, it will stop waiting and checking the states of node list, the feedback will be directly sent to basestation to confirm that the destination node has received the command. And if the destination node is not in the network, the PC will receive a feedback informing that every node in the network has received the command, which means that the destination node is not in this network because every node in the network has received the command but no feedback is from the destination. However, if the routing path to destination is broken by one or several nodes, the feedback which indicates the link broken will be returned to user.

This is the structure of dissemination and feedback module, a tree-based, manual configuration and reliable implementation. By this module, all the nodes build up a wireless network that each node in the network can be reached and controlled by commands. This module is the groundwork that provides a basic command sending and receiving service to other modules. Other modules and service, such as data collection, utilize it for signaling, confirmation and identification. I will explain in the next sections along with the description of these modules.

4.1.4 Configuration reuse

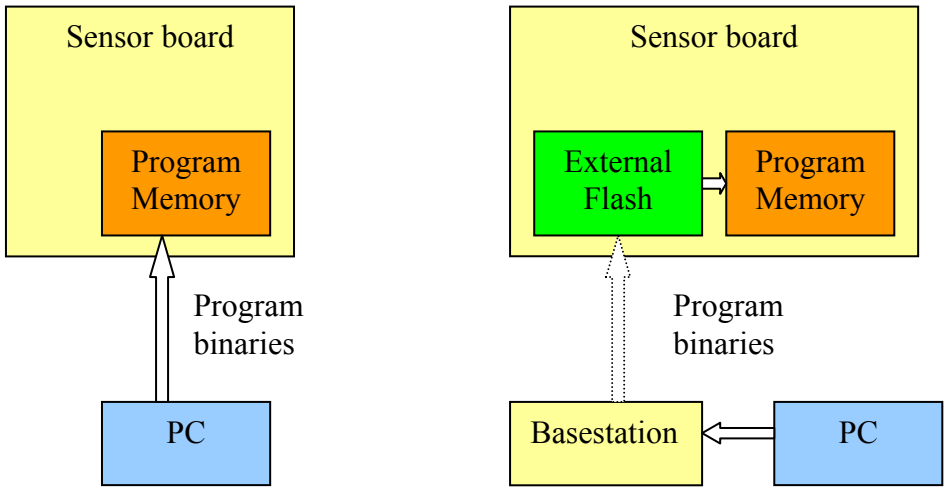
Since switching from one program image to anther and rebooting are normal during an experiment, the configuration for a node including information of node list and previous node are saved in the permanent flash that can be reloaded next time the node reboots. By this feature, we do not have to configure the same network every

time.

To conduct these operations, ConfigStorage which is an existing interface for storing data of small size is included in our module. Our structure of configuration is showed in Figure 4.1(c), the field bootKeeper is a permanent flag that it is reserved for future extensions. We defined the size of volume in volumes-stm25p.xml file for configuration. Every time a node reboots, the module will mount the flash volume automatically and read out the last configuration. And when the configuration changes, it will be used to overwrite the old one in the flash immediately.

4.2 Network Programming

Network programming is much different from a normal programming, in which we wire the sensor board with PC via USB and reprogram it with a new program image. But for network programming, we have to consider the way to disseminate program to multi nodes via wireless links and reprogram them remotely. We can see the difference in Figure 4.5.



(a) Direct programming (b) Network programming

Figure 4.5. Direct programming and Network programming approaches

As we introduced in Chapter 3, we decide to use deluge module as our program dissemination and reprogramming module. The version of deluge we use is 2.0 which is released with TinyOS 2.x.

4.2.1 Introduction of Deluge

Deluge is a reliable dissemination protocol for large amounts of data. It can be used to propagate program images from one node to multi-nodes which are also running deluge protocol.

For a new image, Deluge assigns it with a greater version number in order to inform

other node which runs old version to update. Deluge split an image into pages of fixed-size. A node can only send or receive a single page at one time

A “deluge” process contains following steps. Firstly, nodes broadcast advertisements periodically which contain a version number and a vector indicating the pages that the advertiser has. When a node realizes from the advertisement that it need to update with some new pages, it will wait for some time to check which node provides the needed pages by listening advertisements. And then the node will send request to the chosen node, indicating the pages it need. After received the request, a node will prepare the requested data and broadcast them. After completely receiving the last packet of a page, the node will broadcast an advertisement of the page before requesting next needed page. In multi-hop network, this will increase the speed of dissemination because the nodes in different part of network can communicate in parallel.

The implementation of Deluge is in nesC language and also includes a reprogramming function. When a node has received an image of a new version completely and be requested to reprogram itself, it will call the reprogram interface to reboot.

TinyOS provides an interface called NetProg for network programming. This interface has two commands, one is used to reboot only, and the other is reprogram and then reboot. The parameter for latter one is the physical address of image in external flash.

So, Deluge 2.0 with TinyOS should be a sufficient solution to the network programming module to our testbed. However, Delugerelies on periodic advertisements to keep nodes informed of their neighbor’s states. As we discussed before, this feature may interfere users’ experimental program. So we introduce an interface to Deluge to provide extra functions to stop advertisement while still guarantee every node can receive an image.

4.2.2 A new interface for Deluge

To adapt Deluge to our network programming module, we still need some additional functions to meet our requirements. Therefore, we design a new interface called DelugeTools for Deluge.

The DelugeTools interface contains two commands and one event: the functions of two commands, lock and unlock, are to stop/restart the propagation function and the advertisement broadcast in Deluge, and the event “objectReceived” is triggered when a node has received a new image completely by Deluge protocol. To use this interface, user should register a handler for dealing with this event.

The implementation of this interface is added to the main components of Deluge module which are DelugeC and DelugeP. For command “lock”, a flag which

indicates whether the propagation function is locked will be set. So a node which called the lock command can not receive new packets from Deluge and also will not broadcast advertisements. Command “unlock” is the opposite operation of “lock” that it reopens the propagation function again. In DelugeP, there is an event handler for receiveDone event of underlying interface ObjectTransfer. If a node has received the whole program image and last received command asked for reprogramming, this handler will call the command of NetProg interface to reprogram the node. So we add a new task in this handler before reprogramming in order to trigger objectReceived event for DelugeTools interface.

The DelugeTools interface is used in our software framework with the help from the command dissemination and feedback module. For the commands “lock” and “unlock” of DelugeTools interface, there are two commands reserved by command dissemination module, by which user can turn off/on the propagation function of Deluge for all the nodes by sending a single command.

The DelugeTools also improve the Deluge with a new feature that user can know which nodes have received a new program. It’s an important feature that keeps user to be informed with the nodes that completely receive the program. The implementation is carried out by the event handler of DelugeTools. In our software framework, when this event ObjectReceived triggered, the event handler will generate a feedback message with a 32-bit uidhash of program image and send it to the basestation. Considering the fact that a feedback may be lost in the retransmission from the sender to the basestation, the feedback sender should know whether the feedback has been received by the basestation. In our framework, there is a timer for iteratively sending the feedback to the basestation until the feedback has been confirmed by a confirmation message from the basestation. Although some redundant message may be generated, we consider the redundant acknowledgements are necessary for keeping a high reliability.

Now Deluge plus DelugeTools interface is a complete solution to network programming for our testbed.

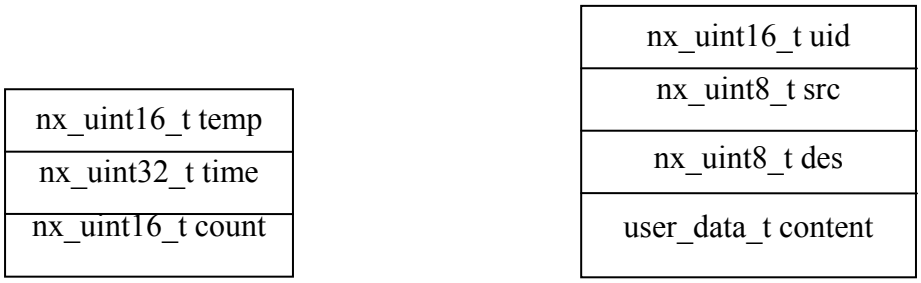
4.3 Data Logging and Gathering

In most of WSN experiments, experimental data will be generated during the process of experiments, which are valuable for analyzing the behavior of WSN afterwards. So data logging is a necessary function in a WSN testbed especially in our testbed. . Since our testbed uses wireless communication to disseminate commands, program binaries and etc. The wireless transceiver of a sensor board is shared by our software framework and user’s program. During the execution of user’s program, the wireless transceiver is occupied by user’s program. So log data has to be saved in remote sensors’ flashes, which will be collected after the end of user’s programs. Moreover, To help user to gather experimental data is also a required function. Thus in our design, this module contains two functions: data logging and data collection.

TinyOS provides several modules and interfaces for permanent storage of data from which we choose LogStorageC as our underlying component for data logging which provides sufficient functions. We can call the functions and set event handlers via two interfaces LogRead and LogWrite which should be linked to the component LogStorageC.

In addition to storage, we design the collection part of the module. In TinyOS, there are also some collection modules that we can use. However, these implementations are designed for different purpose that not suit to the architecture of our testbed. Moreover they usually have a large code size. So we decide to write a light implementation for this function.

To provide basic log and collection operations to user, we write an interface called DataStorage. It contains commands for logging and collection, and also some events for feedback and error checking. The implementation is carried out by DataStorageP and DataStorageC components. The operations for data storage extend the functions of LogStorageC, e.g. DataStorage.append() is implemented by calling LogWrite.append(). We extend it by adding error reporting mechanism for these commands which may return input/output errors. When an error occurs, event failureOccured is triggered with a variable as its parameter which indicates the type of error. If a developer wants to deal with some of these errors, he can assign corresponding feedbacks for them and collect them by the module of feedback. When a command of this module has been executed successfully, event workDone is triggered with a parameter indicating the type of command.



(a) example of user_data_t structure (b) dataToSend structure

Figure 4.6. data structures for collection

In the collection part, we take advantage of the routing strategy from the feedback module. However, the data structure which is showed by Figure 4.6 are different from the one in the module of command dissemination & feedback. The structure dataToSend is the message container for transmission, including a 16-bit sequence number, source and destination id, and actual log data. The size of the structure varies with the user_data_t structure, which is used to store experimental data, is defined at compile-time. E.g. in Figure 4.6 (a), the user_data_t structure contains a 16-bit integer for temperature value, a 32-bit integer for system time and a 16-bit counter.

The sequence number in dataToSend structure is used to eliminate packet redundancy.

Because in real-life transmission, packet loss is possible to happen, the retry mechanism is applied to solve packet loss, however, it may cause the problem of redundancy. User can utilize the sequence number to judge if the packet has already been received and discard it if it's redundant.

There are two commands and two feedbacks for this module: commands CMD_COLLECT and CMD_ERASE, and feedbacks ACK_COLLECTION and ACK_CLDONE. CMD_ERASE is used to erase all the data in the flash of a sensor board. When a node receives this command, it will call DataStorage.erase() to erase all the data.

When a node received a command CMD_COLLECT, it will first return a ACK_COLLECTION feedback of which the "data" field is the size of user_data_t in bytes. When PC has received this feedback, it will initialize a message instance with the size and register a new listener for this kind of message. When PC is ready for collection, then the node will call DataStorage.collect() to start the collection. During the collection by the module, the node will iteratively read out a piece of data, insert into a dataToSend message and send it to PC. Each node follows the route of feedback to transfer logging data. However, in order to avoid packet loss during the multi-hop transmission, a node will repeat the same message of data to PC at regular intervals until receiving a confirmation from PC. The sequence number for eliminating redundant data increases in every message. So when PC received a message of which the sequence number is smaller than the last message, this message will be considered as redundancy and discarded.

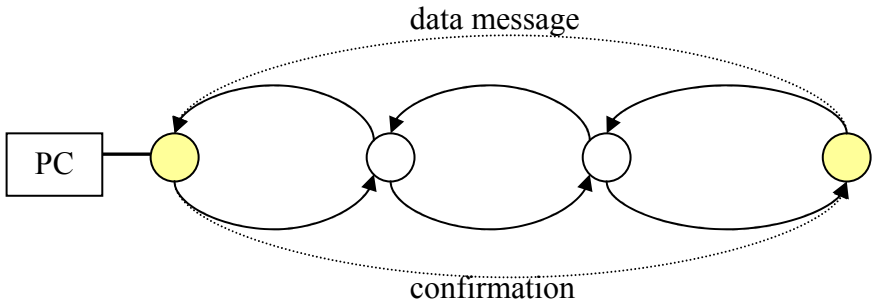


Figure 4.7. Collection scheme for data uploading and confirmation downward. The dotted line points out the source and final destination of messages and the real line is the actual route for messages.

When the module tries to read the next data but finds the entire log data have been read out and sent to user, the event allDataSent will be triggered. Accordingly, the event handler in our software framework will generate a feedback of ACK_CLDONE and send it back to PC. So user can be notified that collection process is done and all the data have been gathered.

4.4 Time Synchronization

FTSP module is released with TinyOS 2.1. It is used as an additional service for our testbed. The FTSP module consists of two main components TimeSyncC and TimeSyncP, following the protocol introduced in [13]. The global time of all the nodes in the network is synchronized to the node with minimal node id. So in our testbed, we can set node id of the basestation to the smallest one, then other nodes synchronize their time to the basestation.

Some interfaces are provided to user: GlobalTime, TimeSyncInfo and TimeSyncMode. GlobalTime contains common functions, by which user can get the current global and local time, or make conversion between global time and local time. TimeSyncInfo has more functions that let user see more details of time synchronization, as it provides functions to get the dynamic parameters, like the latest sequence id, current skew between the local and global time, and current root id. This interface is usually used for testing the synchronization module. TimeSyncMode is a interface to change the mode of FTSP module. The FTSP protocol relies on flooding the synchronization messages to every node. The FTSP module iteratively sends the message at regular intervals by default. However, there are two modes for the module: one is TS_TIMER_MODE; it's the default mode that a timer is running for sending the synchronization message automatically. The other is TS_USER_MODE, with this mode, user have to send synchronization messages manually. The interface TimeSyncMode just gives a function setMode() to change the mode. And also a function send() for sending synchronization message manually during the TS_USER_MODE mode. These interfaces are linked to our software.

For this module, the command dissemination & feedback module assigns two commands and one feedback: TEST_SYNC, CMD_STOPSYNC and ACK_SYNC. CMD_STOPSYNC is a command to stop the timer which fires periodically to send synchronization message. The purpose is similar as stopping the propagation function of Deluge which avoids interferences to users' program. TEST_SYNC and ACK_SYNC are used for testing the performance of synchronization module. The command TEST_SYNC asks the basestation to broadcast a message. Nodes which have received this message will obtain the local arrival time from the message, convert it to a global time and send the global time back in a feedback of ACK_SYNC type. Since the message is broadcasted, theoretically every node in range should receive the message at the same time. Therefore, if they are all synchronized, the global time returned should also be the same. From the feedbacks, we can see how precise the network is synchronized.

4.5 Key Optimizations

Three modules and one service constitute a complete software framework in our testbed. In order to achieve a better flexibility and be more user-friendly, we made some improvements to our original design.

4.5.1 Modularization

For embedded systems, resources are usually limited thus user should be able customize their program to fit the available resources. Restrictions on programming memory and RAM also require the customizability in our software framework.

Among the modules and service, collection module and the time synchronization service are not must-have for some experiments. So in our software framework, we set macros of pre-processor for these two modules. User can set whether to include the collection module and time synchronization service in a header file called "TB_Config.h" at compile-time. The definitions are as below:

```
#define TB_COLLECTION
#define TB_SYNC
#define TB_DEBUG
```

TB_COLLECTION is the directive for data collection module. TB_SYNC is for the service of time synchronization and TB_DEBUG is for functions of debugging. If these directives are commented out, respective module or service will not be compiled. This enhancement can save a lot of programming memory and RAM for user. In our test, without these module and service, user can save about 11KB programming memory and 500Bytes RAM.

4.5.2 Division solution

This enhancement is also about the limitation on memory. The Tmote Sky has only 48KB programming flash. A complete version of the software framework takes much programming space. The remaining space for users' program is limited, which is problem we would like to tackle.

We find that the network programming module is bulky in the software. This module is only used to disseminate new programs and reprogram nodes remotely. Therefore when user's program runs, it is not necessary to keep it in the program memory. So we propose a solution which splits the software into two different versions: A Deluge version which includes Deluge module, and a light version without Deluge module. The Deluge version is mainly used to disseminate program and reprogram nodes. And when the process of network programming is done, user's program starts to run with the light version of software framework. The light version has an interface that it can be switched to the Deluge version when user needs to disseminate a new program. For Deluge module, a golden image is saved in the external flash which can be modified only via physical connection. It will be loaded when a buggy application works abnormally, which brings the error node into a recoverable state. We utilized the external flash to store the Deluge version software framework as the golden image, which gives more programming space for users' program.

Function / Version	Deluge	Disseminate&Feedback	Collection TimeSync
Deluge	√	√	×
Light	×	√	√

Table 4.1. Components for different versions.

In order to implement the switch between the Deluge version and the light one, we add some extra interfaces and functions, especially for the light version. User compiles his program with light version and inserts it to the basestation, and then utilizes Deluge module of Deluge version to disseminate the program image to other nodes and reprogram them. All the nodes then begin to run user's program, supported by the light version of the software framework. When user has a new program to test, the command CMD_TODELUGE can be disseminated to all the nodes and ask them to switch to the Deluge version. As we introduced before, the reprogramming function is included in the NetProg interface that need physical address of image as its parameter. However, to get a physical address of a volume in the flash is one of the functions in Deluge which is not included in this case. So we write a component StorageMapP and its configuration StorageMapC to provide the interface StorageMap which has a function of getting physical address of a Deluge volume. The implementation is borrowed from component BlockStorageManagerP and its configuration BlockStorageManagerC which are included in Deluge module. As we plan to place the Deluge version in the golden image position, we need to get the physical address of golden image volume by calling StorageMap.getPhysicalAddress, and then switch to the Deluge version by calling NetProg.programImageAndReboot with the address.

Deluge module has signal commands such as disseminate, reprogram and stop disseminate as we introduced in the first subsection. If some intermediate nodes run light version that they cannot retransmit Deluge commands to next hops. For example with Figure 4.8, user wants to reprogram node c, but c is out of range of a, and b runs light version that do not have functions to retransmit the command for a. So c fails to be reprogrammed. To solve this problem, in light version, our command dissemination&feedback module takes over these signal commands for Deluge.

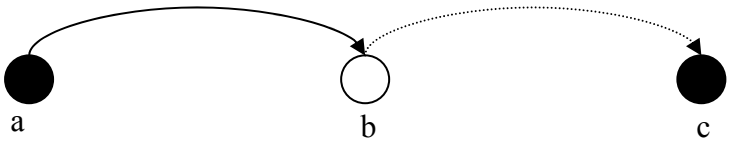


Figure 4.8. Deluge commands dissemination, the black nodes are running Deluge version and the white node is running light version.

In order to provide additional information for diagnosis, our command dissemination & feedback module reserves CMD_CHECKVERSION and ACK_VERSION for verifying versions of nodes. To get the version information of a node, user can send

the command CMD_CHECKVERSION to destination node, a feedback of ACK_VERSION will return in the field “data” which indicates which version is running on the node. Here we define 0 as Deluge version and 1 as light version. In addition, this command and feedback can be used to check the internal errors of a node too. Because the field “uidhash” of the feedback is used to save the 32-bit error flag which records the internal errors occurred since the node boots. Each bit of this flag stands for a kind of error so that it can be used to monitor 32 different kinds of errors.

With this improved solution, user can save about 10KB programming space. However, the tradeoff is extra operations for switching between two versions that increases the consumption of power.

4.5.3 GUI Tools

In order to provide a user-friendly interface, we write a suit of GUI software in Java for management of our WSN testbed. All the operations can be conducted in this GUI, including command dissemination, feedback receiving, data collection and error diagnosis. The suit of software is developed by NetBeans IDE and supported by TinyOS SDK.

Figure 4.9 shows the batch reprogramming GUI, which can be used to reprogramming dozens of local nodes in batches. It also includes some functions from Deluge: user can check image information in a specified Deluge slot and clear an image. User can also make program images and insert them to the different slots of Deluge volumes.

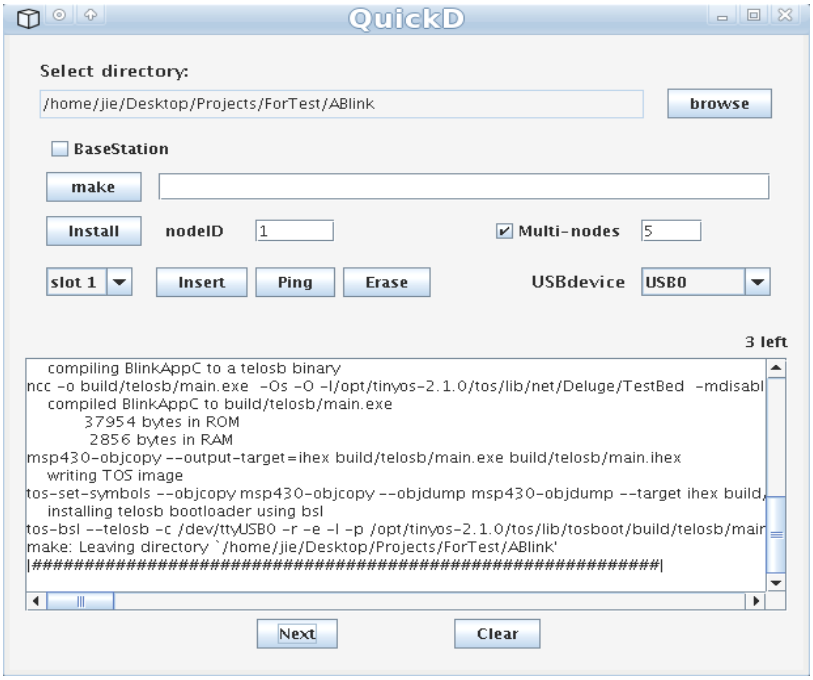


Figure 4.9. Batch reprogramming GUI with additional functions for Deluge volumes

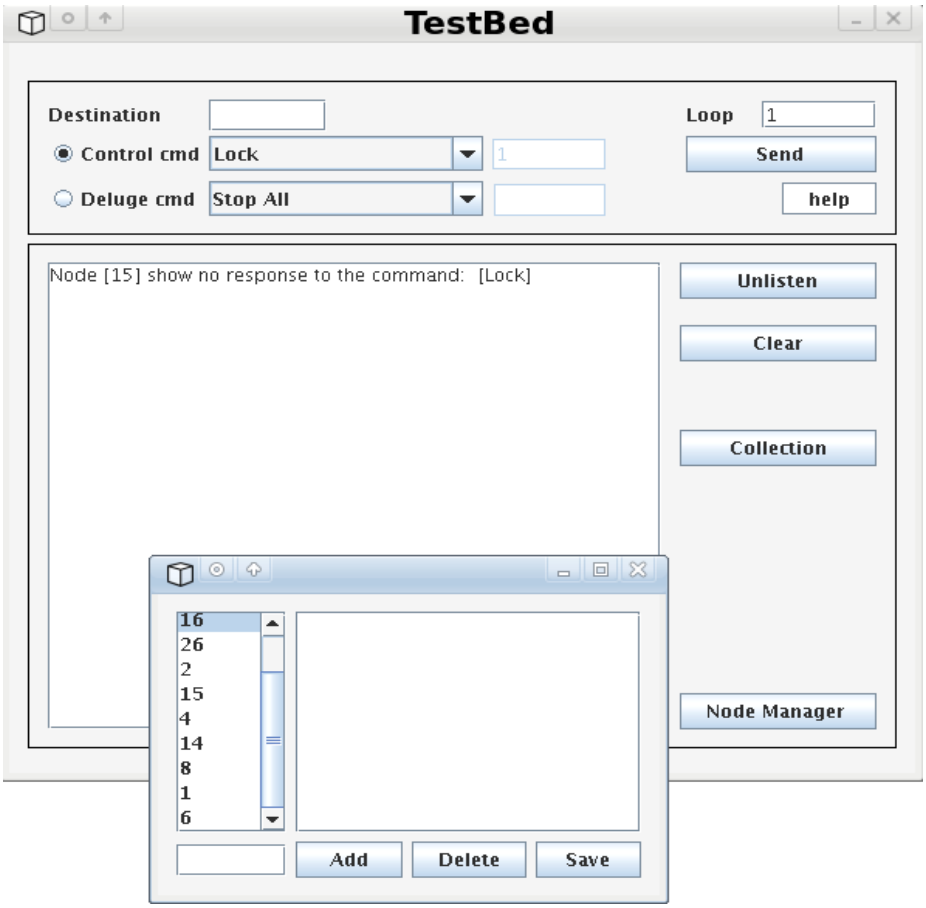


Figure 4.10. Main GUI for our testbed

The main GUI controller is showed in Figure 4.10. It can handle the operations like sending commands, receiving feedbacks and translating them. User can choose an option of command in the list and press "Send" button to send. User can also give instructions to Deluge module, e.g. reprogramming the basestation. A small function called node manager is attached to keep the information of all the nodes in the network. When there are dozens of nodes in the network, in order to identify different nodes in the future, user can give descriptions such as their locations along with the nodes by this function. And this function also provides id of all the nodes to command for checking version, if a user wants to check the version of all the nodes, he/she can obtain all the version information by polling all the nodes saved in the node manager. A typical result is showed in the Figure 4.11.

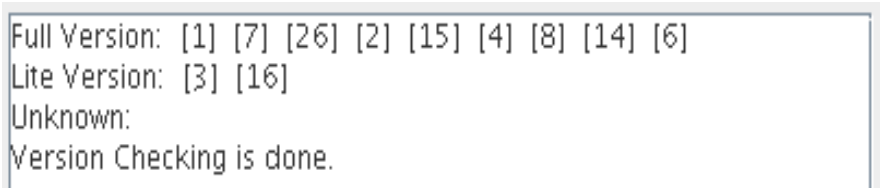


Figure 4.11. An example result for checking versions of all the nodes

For collection of log data, a graphic collection panel is also provided. As we see in Figure 4.12, after configuring the panel with right parameters, user can gather log

data in batches easily by this tool, and at the end of collection, these data can be saved in a txt file.

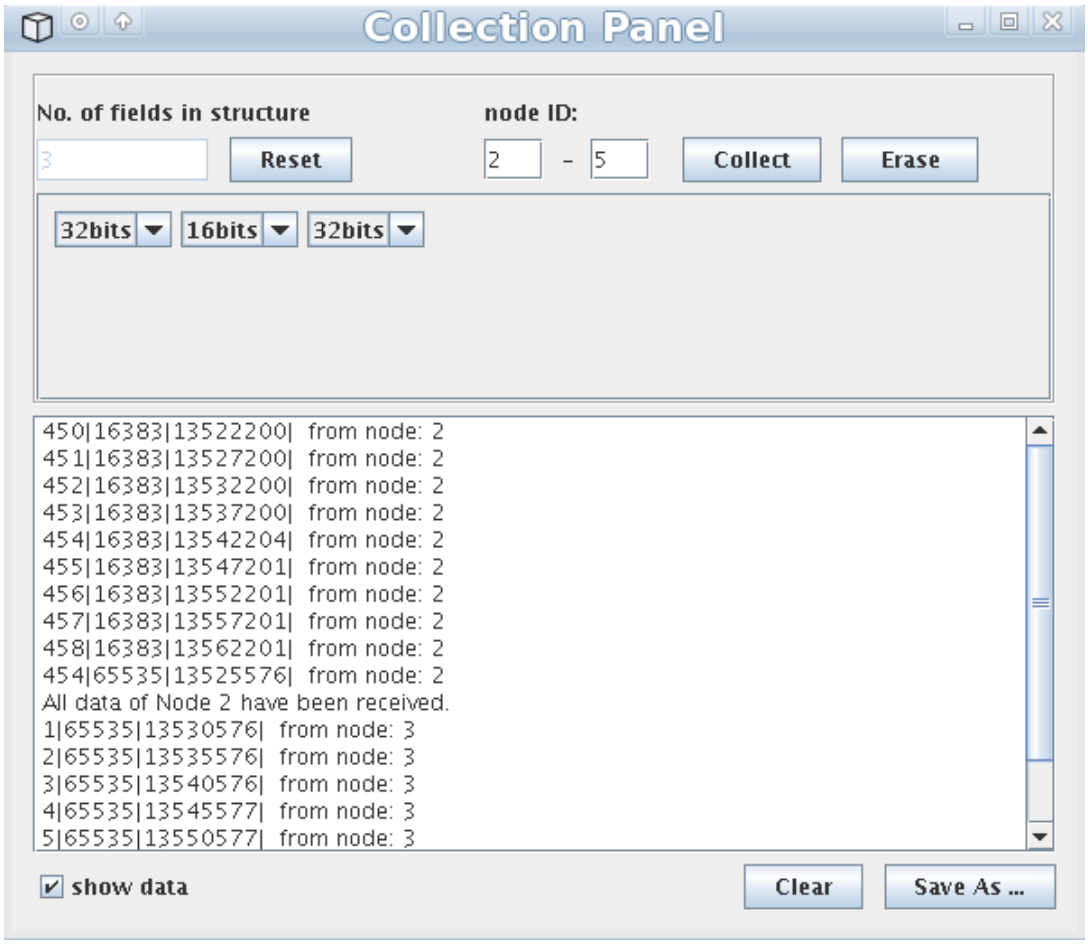


Figure 4.12. Data collection panel

With these GUI tools, a user doesn't have to type different commands to control the testbed, and doesn't have to look up our instruction document to know the meanings of different feedbacks. It saves much time for user and also provides a convenient way for experiment.

4.6 Tutorial

In order to give some instructions to users and developers for using our testbed and extending it in the future, we would like to present two brief tutorials for users and developers respectively.

4.6.1 Quick tutorial for user

Before using our testbed, user has to install TinyOS 2.1 with its toolchain first, some of which are the basis of our GUI software. The details of installation can be found on the official website of TinyOS (http://www.tinyos.net). Java Runtime Environment is

also necessary to support our GUI tools. Related software can be downloaded from the website of Sun Microsystems (http://java.sun.com).

When setting up the nodes at the first time, a user load our golden image which contains the Deluge version of the software framework.

Enter the folder of our golden image. First initialize the node with Deluge version that we can insert volume later.

```
% CFLAGS=-DDELUGE_BASESTATION make tmote install,0 bsl,/dev/ttyUSB0
```

Then make the golden image for different kinds of node,

For the basestation:

```
% CFLAGS=-DDELUGE_BASESTATION make tmote
```

For other sensor nodes:

```
% make tmote
```

Now insert the image into the slot of golden image:

```
% tos-deluge tmote -i 0 build/telosb/tos_image.xml
```

User can also use our GUI tools to insert the golden image, which is easier than the method above.

After installing Deluge version on each node with unique id, the sensor nodes can be deployed. The distance between hops should be in the range of communication according to the practical environment.

To combine with light version of our software framework, a user must add our component into his/her program with a statement:

```
Components MyDelugeLiteC;
```

Users can decide whether to add data collection module and time synchronization service. Comment relevant lines in “TB_Config.h” file can exclude the collection module and time synchronization service at compile-time. If a user’s program includes data collection module, he/she should define the log structure in the program or its header file with the following form:

```
#define USER_DATA_T

typedef nx_struct user_data {

  nx_uint8_t field1;

  …

} user_data_t;
```

Our software framework has reserved a command append() for users' program. If they want to save experimental results in permanent flash, they have to link the command append() to MyDelugeLiteC and call it with a log as a parameter.

If time synchronization service is included, users should define the rate of synchronization message in Makefile file. The default rate is 10 times per second. He/She can define the rate (times in seconds) by:

```
PFLAGS += -DTIMESYNC_RATE=1
```

In addition, the memory space for Deluge volumes, configuration and log data should be defined too. The users' programs should contain a volumes-XXX.xml file which defines the allocations for these permanent data. Take our Tmote(stm25p) for example, a volumes-stm25p.xml should be configured as follows:

```
<volume_table>
  <volume name="GOLDENIMAGE" size="65536" base="983040" />
  <volume name="DELUGE1" size="65536"/>
  <volume name="DELUGE2" size="65536"/>
  <volume name="DELUGE3" size="65536"/>
  <volume name="CONFIG" size="131072"/>
  <volume name="LOGTEST" size="262144"/>
</volume_table>
```

Compiling user's program for non-basestation nodes, insert it to a volume slot of the basestation. And compile the program again for the basestation, with a parameter CFLAGS = -DDELUGE_BASESTATION. Then insert it to another slot among the three available. Here for example, we use slot 1 to store the image for non-basestation nodes and slot 2 to store the image for the basestation. All these work can be carried out with our GUI tools.

A user has to configure the logic topology for the whole network firstly. Command "Add Node to list" and "Clear Node List" can be used to add or remove leaves. The logic topology should be adjusted according to the physical deployment in order to ensure each node in the network can be reached.

To start dissemination of user's program, we need to open our main GUI controller. Choose the "Disseminate" in Deluge commands set and fill in 1 as its parameter because we are inserted the image in slot 1. Waiting until each node in the network has received the image, user can be notified by the feedback from each node. Then send "Reprogram Remote" command to request all the nodes except the basestation to reprogram with new program they have received. User can use the command "Check Version" to verify if all the nodes have reprogrammed. For the basestation, user has to send "Reprogram Local" with parameter 2 to reprogram with its special version for basestation that we have inserted in slot 2 before.

Before starting the experimental program, user may have to turn off some functions in our software framework, e.g. stop time synchronization by command "Stop

Synchronization".

At the end of experiment, user can gather the experimental results from nodes by collection graphic panel. Please note that that the structure of data must be set the same as the one defined in his program.

If a user wants to start a new program, he/she has to send "CMD_TODELUGE" to switch all the nodes to the Deluge version again and then repeat the same steps as above.

4.6.2 Instructions for developer

There are some useful tips for developer to extend or modify the software framework.

In the TinyOS 802.15.4 frame, AM type is a single byte field which indicates which active message type the payload contains. In our software framework, there are several AM types occupied by different modules. We list these occupied AM types below:

In command dissemination&feedback module, the AM type from the basestation to PC is defined as 7, and to wireless nodes is defined as 9. For data collection module, the AM type from the basestation to PC is defined as 10 and to wireless nodes is defined as 8. 0x53 and 0x54 are reserved for Deluge module.

In header file "MyDeluge.h", there are several pre-defined constants. RETRY_TIMES is the maximum retry times for sending a message to a node. IMG_REP_RETRY_TIMES is the maximum retry times for feedback an image report to previous node. BUFFER_SIZE is the buffer length for our feedback buffer. In order to avoid feedback packets loss, we write a simple component called SimpleBufferC which contains basic operations of a buffer and has two modes of FILO and FIFO. When a branch has many leaves, the BUFFER_SIZE should be set large as it may receive many feedbacks in a short time. However, large buffer size will cost more RAM. So this constant should be modified according to the real applications.

Error flags which are used for diagnosing the problems in the system, we have defined 9 different kinds of errors in MyDeluge.h file. Developer can still define 23 other kinds of error in this enumeration.

In data logging and collection module, we have introduced an event called failureOccurred, which can be used to handle different errors. This event is still kept for extension that can be used to improve the reliability of this module.

4.7 Performance analysis

To test our testbed in a practical environment, we configure several test cases to check its performance of different part.

Before performing the tests, we setup the WSN network which consists of one basestation and ten Tmote sensors which are deployed in the different rooms on the 19th floor of EWI building at TU Delft. The deployment is illustrated in Figure 4.13. Node 1 is the basestation which is connected with PC. Node 2 and 3 are deployed in the corridor, and other eight nodes are deployed in four rooms, each room has two nodes. The logic topology is configured as Figure 4.13 showed. Initially, all of these nodes are programmed to Deluge version of software framework.

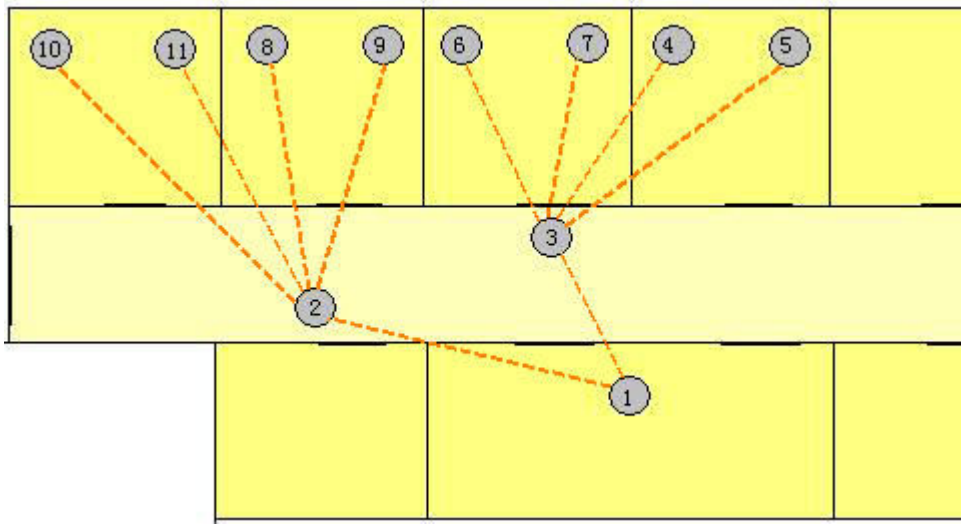


Figure 4.13. Deployment of our wireless sensor nodes in different rooms of 19th floor, the dashed lines indicate the logic topology for this WSN testbed. Node 1 is the basestation.

4.7.1 Dissemination of commands

The first test case is to verify the functionality of dissemination and feedback module. We want to check the performance of this module under normal conditions. Every node in our network was turned on and running normally. Firstly we sent a command to a single node for certain times to test the module in one-to-one mode; and then we sent the command to every node for certain times to test the module in one-to-all mode. Here, for every step, we sent the same command for 100 times. If a feedback shows the destination node has received the last command in one-to-one mode or every node has received the last command in one-to-all mode, we counted it as a feedback of “success”, or it is a feedback of “fail”. The results are in Table 4.2.

Results / Tests	No. of sent commands	No. of received feedbacks (redundancy)	No. of feedbacks with success info.	No. of feedbacks with fail info.
One-to-one	100	116(16)	92	8
One-to-all	100	93(4)	79	10

Table 4.2. Results for dissemination and feedback

As we see the results in Table 4.2, due to the retry mechanism, the redundancy of feedback exists that user may receive several acknowledgements for a command. For instance, in one-to-one mode, the number of feedbacks (116) is larger than the times we sent the command (100). However, for each command there is a corresponding feedback returned at least. Because wireless communication is not reliable that sometimes the channel may be jammed due to interference, a command or its acknowledgement feedback may be lost in the transmission. The one-to-one mode has a 92% success ratio for sending the packet to the right node. For one-to-all mode, the results are not so perfect to show the reliability of the module. Although we sent a command for 100 times, we got 89 feedbacks with 4 redundant feedbacks and actually 15 feedbacks were lost in the paths. The success ratio for delivering a command to all the nodes is about 79%. Number of fail time is 10 which is similar to one-to-one mode.

4.7.2 Detection of malfunctioning nodes

The second test case is created to test the function of detecting malfunctioning nodes. In order to simulate an error node in the network, we turn off a node intentionally before the tests. Then we send a command to the node which is turned off to test the detection in one-to-one mode, and then send it to all the nodes to test the detection in one-to-all mode. For column of “Correct report” in Table 4.3, if a feedback reports the malfunctioning node correctly, we count it as a correct report.

Results / Tests	No. of sent commands	No. of received feedbacks (redundancy)	No. of feedbacks with fail info.	Correct reports
One-to-one	100	110(10)	100	95
One-to-all	100	96(4)	92	83

Table 4.3. Results for detection of malfunctioning nodes

From the results in Table 4.3, in one-to-one mode, the ratio for reporting malfunctioning node correctly is 95% while it becomes 83% in one-to-all mode. However, the inaccuracy makes sense, as we realized from the first test case, failing to deliver a packet is possible to occur that feedback mechanism may report the fail

before it finds the malfunctioning node.

4.7.3 Network Programming

The third test case is for network programming module. We want to check the relation between image size and transmission time. So we have created several images with different sizes and disseminate them one by one for certain times. In this test, we have recorded the intervals from the start of dissemination to all the nodes have received the image completely. The average time of dissemination in our scenario is showed in the Figure 4.14. The result also relies on the deployment of the nodes. With a dense network, the time for dissemination will decrease a bit. For example, when we placed all the nodes in the same room, the dissemination time for a 38.5KB image is about 72 seconds while in Figure 4.14 it needs 105 seconds.

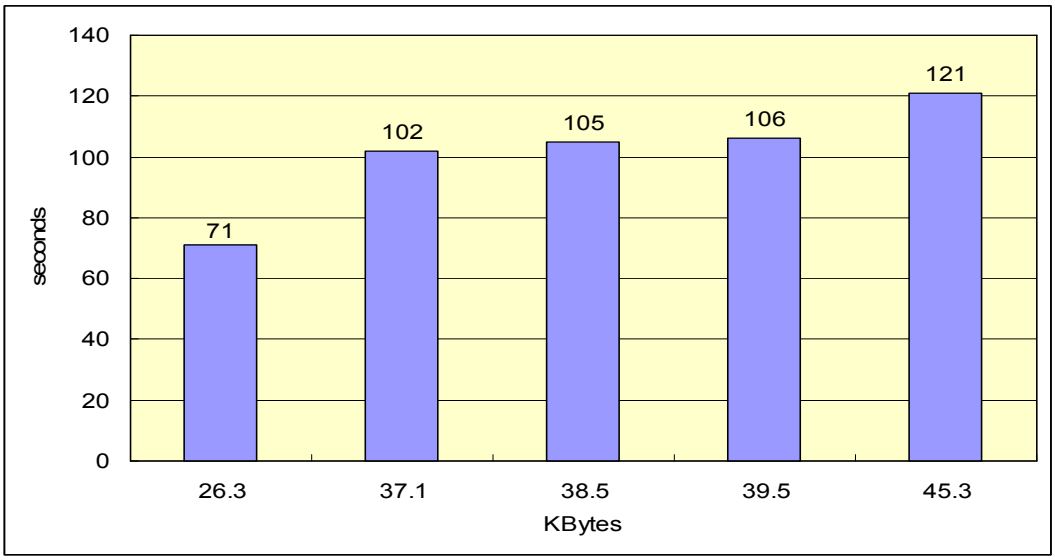


Figure 4.14. Dissemination time for images with different size

4.7.4 Data Collection

The fourth test case is for data logging and collection module. We made a test program which records environment temperatures every five seconds by the sensor on the mote. The size of each log is 10 bytes which includes a 16-bit sequence number, a 32-bit value for temperature and a 32-bit value for global time. For this program, we set 256KB as the space for log data in “volumes-stm25p.xml” file. We run the program for three days and then checked the system and its collection module. When we checked the error flag by feedback, we found that the space for logging data was full and the error flag recorded this event correctly. After collecting all the log data from a node, we found that these data were logged consecutively which completely displays the change of temperatures during these days. So this module is stable for long term running. However, by this test, we can also find some shortages. Firstly the external flash for log data is limited which may overflows due to the large amount of data. The size of whole external flash is 1 MB. However, the external flash is used for

other storages too, such as storage of configuration and Deluge volumes. As default, the space for Deluge volumes costs about 64*3=192 KB. So the space for log data is much less than 1 MB. Secondly the time for collection is a little longer than we expected. For the collection in our test, the total number of log data is 23807 and each structure of log data contains 12 bytes. The time for receiving all the data is about 50 minutes that there are 8 log data received per second on average. However, in this test, we defined that each message contained only one log data. In order to reduce the time for collection, we can insert five log data into a single message which will reduce the collection time to 10 minutes.

4.7.5 Time Synchronization

The last test case is for time synchronization service. We have reserved several commands and feedbacks for debugging. As we explained the principle in Section 4.4, we can receive several feedbacks after sending the command “TEST_SYNC” to the basestation. Each feedback contains a 32-bit field which keeps global time of message arrival. Theoretically, these 32-bit values should be the same. The difference among these 32-bit values can be considered as inaccuracy of time synchronization service. After repeating the command for 10*100 times, we found the maximum difference is 3 milliseconds and the average difference is about 0.42 milliseconds. Considering some actual error in the testing process, the error less than 1 millisecond is acceptable. The time synchronization service can be competent for most applications with normal synchronization requirements.

By these small-scale tests, we have verified the functionality and performance elementarily for our WSN testbed which shows acceptable results. However, on the other hand, we also realized that there is still much space for improvements on the reliability of the whole system.

Chapter 5. Conclusions and Future Work

5.1 Conclusions

In this thesis, we have introduced a new solution for WSN testbed. We have studied some existing solutions for WSN testbed and discussed their advantages and restriction. According to our objective, we hardly find a suitable solution to meet our requirements so that we would like to propose our own solution. With reference to these solutions, our testbed takes advantage of wireless communications as our backbone network instead of Ethernet or USB cables, for the sake of high flexibility and low cost. Our testbed can be widely deployed at different places for experiments without the support of Ethernet and extra devices. The whole testbed consists of Tmote Sky sensor boards and our software framework which provides necessary modules and service like command dissemination, data logging and collection and time synchronization.

We presented the details of our implementation. For our software framework, we implemented a tree-based protocol for command dissemination and feedback, and a protocol for data collection. Writing the codes from scratch for these two modules requires a good amount of knowledge of TinyOS as well as the non-standard programming language nesC that is used to program the Tmote. And for development, unlike other embedded systems, we don't have online debug system for Tmote board, which means programming and debugging rely on simulations or programming real devices with every modification. So it is really troublesome and time-consuming. The main body of the software framework contains about 1100 lines-of-codes (LoC) and 330 LoC for data collection module. Beside the codes for each module, different testing programs have been written along with the modules to verify the functions one by one. It was a useful way to trace and fix problems in the process of development. The total amount for all test programs is close to 1000 LoC. And in order to save time and energy, some existing modules have been adopted from TinyOS 2.1 as parts of our software framework, such as Deluge 2.0 network programming module and FTSP synchronization protocol. It was also not an easy work of importing these modules to our software framework. The existing code was hard to understand, because of its complexity and lack of documentation. And before applying these modules in our software framework, compatibility testing and some modifications or extra functions had to be carried out. For example, in order to optimize programming space for Deluge module, the software framework needed to be refined and additional functions for both split versions had to be implemented. To make our software easy-to-use, a suit of GUI software has been designed for managing the testbed which includes thousands of LoC. Writing Java program on PC is much easier than writing

embedded codes for Tmote. Some codes of graphic components are generated automatically by integrated development environment (IDE) software and programs can be tested with rich debugging tools, such as break point, memory watcher. In order to make sure everything works fine, we conducted a small scale testing for the whole system. After being verified by several test cases, our testbed was proven that it meets our design goals. And we hope our testbed can be widely applied for WSN research.

5.2 Future Work

Considering the requirements of reliability and user-friendly, we have realized that in order to be a mature system, the original edition of our WSN testbed still needs many improvements and optimizations. Especially for reliability of our testbed, some extra mechanisms should be applied to solve the potential problems which may cause some system failures. There are some suggestions for the future work:

1. Fixing bugs and optimizing codes. There may be still some bugs we didn't find in our small scale testing. So working on debugging is still an important work in the future. And for the limited programming space, some optimizations for our software framework are still necessary to be carried out to reduce the code size even further.

2. A protocol for configuring network topology automatically. So far user still has to manually configure the node list for every node. When topology changes, user has to configure them again. So it would be a handy feature if the configuration is automatic.

3. Data compression for collection process. As we tested, the time for gathering a mass of data from each node is longer than a user may expect. So compression of log data is possible before sending them back, which may increase the effective rate and reduce the time for transmission.

4. On-off solution for low-power applications. For some low-power oriented experiments, the radio is duty cycled on and off in order to save the energy. So to ensure all the node can receive commands when their radios are on, an on-off solution should be carried out for our software framework.

5. Improving GUI software to make it more user-friendly. Some useful functions can be added. E.g. Scheduler to handle multi tasks for user. For a complex experiment, several tasks should be performed continuously along with the entire experiment. The software should be able to help users manage all of these tasks automatically. In addition, multi-user interface for remote access could be a plus.

Reference

[1] G. Werner-Allen, P. Swieskowski, and M. Welsh, “MoteLab: A Wireless Sensor Network Testbed”, Proceedings of Information Processing in Sensor Networks, pp 483-488, 2005.

[2] Crossbow Technology Inc., “the MICAz Module,” http://www.xbow.com

[3] P. Buonodanna, “EPRB: Ethernet PRogramming Board,” http://berkeley.intel-research.net/pbuonado/EPRB/

[4] Crossbow Technology Inc., “MIB-600 Ethernet Interface Board,” http://www.xbow.com/

[5] Open-source operating system for wireless embedded sensor networks, http://www.tinyos.net

[6] L.Girod, J.Elson, A.Cerpa, T.Stathopoulos, N.Ramanathan, and D.Estrin, “EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks”, Proceedings of USENIX Annual Technical Conference, June 2004.

[7] Jang-Ping Sheu, Chia-Jen Chang, Chung-Yueh Sun, and Wei-KaiHu, “WSNTB: A Testbed for Heterogeneous Wireless Sensor Networks”, Proceedings of first IEEE International Conference on Ubi-Media Computing 2008, pp 338-343, 2008.

[8] V.Handziski, A.K¨opke, A.Willig and A.Wolisz, “TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks”, Proceedings of International Symposium on Mobile Ad Hoc Networking and Computing, pp 63-70, 2006.

[9] Moteiv Corporation, “Tmote Sky: Low Power Wireless Sensor Module,” http://www.moteiv.com/

[10] Jonathan W. Hui and David Culler, “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” Proceedings of the 2nd international conference on Embedded Sensor Systems (SenSys’04), Baltimore, Maryland, Nov. 2004

[11] Elson J. E., Girod L. and Estrin D, “Fine-Grained Network Time Synchronization using Reference Broadcasts,” The Fifth Symposium on Operating Systems Design and Implementation (OSDI), pp 147-163, Dec. 2002

[12] Ganeriwal S., Kumar R., and Srivastava M. B, “Timing-Sync Protocol for Sensor Networks,” Proceedings of the first international conference on Embedded Networked Sensor Systems (SenSys’03), Los Angeles, California, pp 138-149, Nov. 2003

[13] Miklos, Branislav Kusy, Gyula Simon, and Akos Ledeczi, “The Flooding Time Synchronization Protocol,” Proceedings of the 2nd international conference on Embedded Sensor Systems (SenSys’04), Baltimore, Maryland, pp 39-49, Nov. 2004