FIST: A Framework for Flexible and Low-Cost Wireless Testbed for Sensor Networks

Cheng Guo, R. Venkatesha Prasad, JiangJie He, and Martin Jacobsson

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology, Mekelweg 4, 2600 GA Delft, The Netherlands
{c.guo, r.r.venkateshaprasad,j-he2, m.e.jakobsson}@tudelft.nl

Abstract. Setting up a wireless sensor network for an experiment is both time and effort intensive. Sometimes, the time required to set up the experimental framework is even more than that of the experiment itself. Many existing fixed testbeds manage nodes using gateways and require a lot of support and maintenance. Changing the deployment place – especially if it is a crowded area – is difficult. We propose a framework for implementing a Flexible and low-cost wIreless for Sensor network Testbed (FIST). Downloading the experimental code, reprogramming, testbed control, logging and collecting experimental results and synchronization are all carried out by the sensor motes wirelessly. Thus the testbed can be easily and quickly deployed anywhere. We present our framework and also a case study using FIST.

Keywords: Testbed, Wireless Sensor Networks, Experiment, Flexibility, Low-cost.

1 Introduction

In wireless research, the simulation and analytical work outnumber real-world experiments. At the same time this is absolutely crucial for understanding the characteristics and issues regarding Wireless Sensor Networks (WSNs) in real environments. In fact at some point the algorithms and ideas need to be verified practically. It is important to enable researchers to conduct the experiments while developing the ideas. The main reason for the lack of experimental research is the difficulty in conducting experiments. Setting up an experimental platform and conducting an experiment are cumbersome and time consuming. To fill this gap and making it easier to do more real wireless network experiments instead of solely relying on simulations, we developed a testbed. While designing the testbed, we developed a framework that makes the testbed flexible and easy to deploy. This paper is an account of our understanding, experiences and accomplishments of designing a framework for a sensor network testbed.

We first list all the requirements: a testbed should support implementation of protocols, algorithms, easy and changeable deployment and testing. It should have interfaces for researchers and developers to conveniently develop their code, load and test. Later they can deploy the network in an environment inline with

D. Ślęzak et al. (Eds.): FGCN/ACN 2009, CCIS 56, pp. 216–225, 2009.

the application scenarios of the protocol. The users should be provided the functions to freely start, stop or reset the experiment. Further the experimental results should be collected so that an analysis can be performed. Frequently, a user also has to update the experimental code due to bugs, errors or simple enhancements to the already available code. On a WSN, all these requirements will indeed make it difficult for the experimentalists to run their code and get the results. In summary, a testbed should provide functions such as experimental program (denoted as *exp-program* in the rest of the paper for convenience) dissemination, reprogramming, command dissemination, data logging and data collection. Sometimes, a user may also require the experimental network to be synchronized and nodes can discover and report their neighboring information. Therefore services like time synchronization and neighbor discovery are also in demand.

Existing testbeds [1, 2, 3] normally require a fixed support system to control individual nodes/devices, here sensor motes, through gateways. These gateways have interfaces such as USB or Ethernet through which they are wired into a network. A user issues new experimental code or commands to sensor motes by first sending them to the gateways which then relay them to the sensor motes. Similarly, data is collected by server. In such a testbed, sensor motes only take care of experiments. The control and management are all carried out by the backbone consisting of wired servers and gateways. Although this can reliably carry out experiments, they are not flexible since their physical topology can not be easily changed. Moving these testbeds from one place to another, especially to the places which are crowded, is difficult and time consuming. Moreover, additional supporting devices increase cost of a testbed.

In this paper, we propose a new Flexible and low-cost wIreless Sensor network Testbed (FIST), which removes the wired backbone but uses wireless connection itself to perform testbed management and control. It means that all the functions and services are all carried out by sensor motes and through wireless links between them. Since wires are removed, it can be quickly deployed at any place and does not cause obstacles to people. However, the removal of wires and gateways requires the testbed control and maintenance traffic not to interfere with the experimental traffic since they share the same medium. Furthermore, a smart software should be designed and implemented in sensor motes so that it can realize all the functions with a reasonable amount of resources. We show that our testbed framework can meet these requirements by well designed software and hardware architecture and fine-tuned implementation. We give the users the freedom to choose functions that are required in their specific experimental environments and load only necessary modules. Therefore memory – which is one of the limited resources in motes – can be saved for loading the experimental code. FIST can fulfill most of the requirements of usual sensor network experiments unless an experiment generates a large amount of results, or desperately requires two separate medium for experimental traffic and control traffic, such as results have to be collected during an experiment or experiments have to

run under user's instructions. The design requirement and both hardware and software architectures of FIST are presented in the next section.

2 Requirements and Architectures of FIST

2.1 Requirements of a WSN Testbed

Before we describe our design of FIST, we first list all the requirements to aid us in ahead. We have to review the process of doing a WSN experiment since the ultimate use of FIST framework is for doing WSN experiments. Usually, for any WSN we start with programming the nodes then deploy them in the field. Once deployed, neighbor discovery needs to be done then the command to start an experiment is issued or implicitly assumed. During the experimentation, usually one will find some bugs. Then the experiment has to be stopped. After collecting the error codes, the exp-program needs to be debugged. Later motes have to be reprogrammed with newer exp-program; then test procedures starts again. If no bug is found, then the necessary data has to be stored somewhere or transmitted back to a data server. In case that data is stored locally, we have to collect it from each node in the network. We list here the functions to be incorporated in a testbed: (a) We need to have a mechanism by which we can disseminate commands to a particular or any number of nodes. (b) To guarantee a successful dissemination we have to build routes between the control station and each node in the testbed. The route must be reliable. (c) We have to disseminate exp-program and reprogram to the motes. (d) The experimental results have to be logged and reported to the user. (e) Some services such as time synchronization and neighbor discovery should also be provided.

We also have some non-functional requirements of the proposed testbed. The testbed has to be: (1) flexible to deploy and re-deploy in any environment; (2) reliable; (3) convenient, simple and user friendly; and (4) low cost.

2.2 Hardware Architecture of FIST

As introduced in Section 1, we want to get rid of the wired backbone from the testbed and take advantage of wireless connections to manage and control experiments. Thus we propose a simple hardware architecture for FIST. An example of such architecture is as shown in Fig. 1. A root mote is connected to a root server by USB and other motes are connected wirelessly to the root mote. We have a tree topology in the network since it is easy to configure, control and maintain yet proven to be relatively reliable network topology and widely used in many existing networks. We can see that the motes act as both experimental devices and experiment management devices. Any PC or handhold equipments which have Linux OS and Java Runtime Environment can be used as the root server of our testbed.

We can see that there are no extra devices required in FIST and more importantly we do not have wires. This feature allows us to deploy this testbed quickly

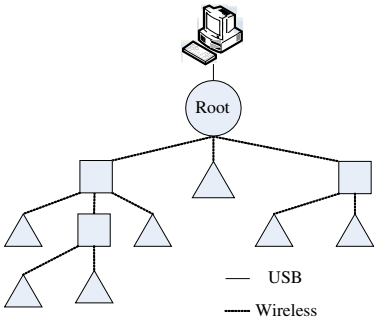

Fig. 1. Hardware Architecture

at any place especially in the environments where wires can be a hassle, such as canteen, crowded markets or museums. The deployment neither requires any specialized personnel to drill or mount the motes on ceilings nor needs dedicated place. Moreover, when we want to add more sensor nodes into our testbed, we can easily power up the motes and allocate them anywhere within the radio coverage of our testbed. Thus the architecture meets the non-functional requirement of flexibility.

Currently we choose Tmote Sky motes as our wireless sensor motes for its rich on-board modules, reliable quality and acceptable price. The transceiver of 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 micro-controller with 10KB RAM and 48KB Programming Flash. Besides, it also has a 1 MB flash memory. A user can choose to have integrated temperature, humidity and light sensors on board. The transceiver has a data rate of 250kbps at 2.4GHz and the transmission range of about 50m indoor and 125m outdoor.

2.3 Software Architecture of FIST

Fig. 2 shows the software architecture of FIST. We divided the whole software into two spaces. One is the Testbed Program Space (TPS) and the other is the User Experimental Program Space (UEPS). Code developed for realizing the functions of FIST is isolated from users' exp-program so that a user does not have to consider the testbed program when develops exp-program. A few interfaces are provided to link the two spaces so that the testbed program can control the exp-program, such as start, stop or reboot. Services, such as time synchronization and neighbor discovery, are shared between the two spaces. A user control the functions in TPS by either console commands or a GUI interface so that the non-functional requirement of being user-friendly can be fulfilled. The testbed program uses the standard 802.15.4 MAC and physical layer. Inside the user's space, one can implement and test protocols from MAC to application layer or services.

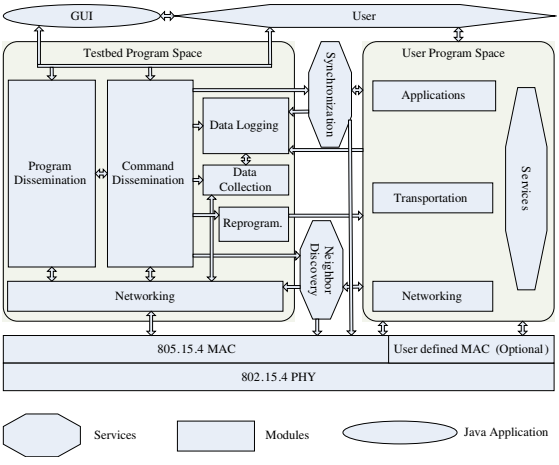

Fig. 2. Modules

There are several necessary modules in the TPS, *viz*, networking, program dissemination, data logging, data collection and reprogram. Among them the command dissemination module is directly controlled by user. The rest of the modules are controlled indirectly via the command dissemination module. We introduce these modules in the next section.

3 Modules and Services in FIST

3.1 Networking

We start the introduction of modules and services from the network module since others depend on it. The networking module in the TPS is designed to construct and maintain the routes between the root node and all the nodes in the testbed. As shown in Fig. 1, we organize the nodes in the testbed in a tree topology. 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 every node in the network must know its parent and its leaves if there are any.

Every node maintains a node list shown in Fig. 3(b) which keeps the information about its upstream and downstream nodes. The variable "count" in a *node_list* structure keeps track the number of downstream nodes. The structure named *node_state* is made up of a unique id and a linkstate variable as shown in Fig. 3(a). Since we may have to reprogram the motes from time to time during an experiment, the connectivity configuration of each mote are saved in its permanent flash.

3.2 Command Dissemination

The format of command and ACK packets is shown in Fig. 4. For different types of commands and ACKs, we define two sets of constants. All the commands have

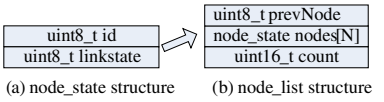

Fig. 3. Data structure for configuring tree topology

the prefix *CMD_* and the ACK have the prefix *ACK_*, e.g., *CMD_COLLECT* is the command to start data collection while *ACK_COLLECTION* is to acknowledge the command. There are five fields in this format. "cmd" is the type of commands or ACK. "data" stores parameters associated with commands or ACK. "error" is a 32-bit variable used with ACK. 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 errors. Last two fields "src" and "des" keep the source and the destination id of a message and des = 0xff for broadcast. Commands are broadcasted to downstream nodes in the *node_list*. Before sending out commands to nodes in the list, the "linkstate" of each node in the list are set to "NOT_RESPONDED" with an ACK state is set as "ACKed". If the destination node/nodes specified in the command message is set as ACKed, the node has finished the command dissemination and sends an ACK to its upstream node. Nodes rebroadcast the message until the maximum retrial count, i.e. 10. Then the ID of the non-ACKed node is reported to the upstream node until the root node and in turn the user knows which node is out of order to support reliability. In general, there are four kinds of ACKs. (a) *For command dissemination*: the ACK confirms that commands are received by destination(s). (b) *For data collection*: the ACK confirms the start of collection. (c) *For reporting errors*: the ACK reports different kinds of errors to the user for diagnosis. (d) *For confirmation of received program*: the ACK reports which of the nodes have received the new program.

3.3 Program Dissemination

Program dissemination is also carried out through the network set up by the networking module. Since an program can be up to 48KB large, which is the size of the program memory in the Tmote Sky, we have to split a program into several pieces. Here we take the split mechanism as it is in the well-known Deluge [4] implementation. Dissemination of program is same as the one of command dissemination except that the receiver only acknowledges the receiving of an entire program instead of page by page to avoid interference with the

nx_uint16_t	cmd
nx_uint16_t	data
nx_uint16_t	error
nx_uint16_t	src
nx_uint16_t	des

Fig. 4. Format of command packets

experiment. Compared to the dissemination method used in Deluge [4], our method provides users the information whether and when a node has received a new program. In our method, adding a node has to be done by manually configuring the upstream and downstream node id in the networking module then a program dissemination process has to be carried out.

3.4 Data Logging

Since experimental results are valuable for analyzing the performance of a test program later, data logging is a necessary function in a WSN testbed. The wireless transceivers in FIST are shared by users and testbed program, thus during the execution of a user's exp-program, the wireless transceiver should be fully occupied by user's program to avoid interference. Consequently, experimental results have to be saved in the flash memory of nodes, that will be collected after the completion of the experiment. Normally, we should have up to 900 kB space. A user can define the structure of the results he wants to store. The *user_data_t* structure contains a 16 bits integer for temperature value, a 32-bit integer for a timestamp and another 16-bit integer as a packet sequence number.

3.5 Data Collection

The data collection module also takes advantage of the networking module. The packet format in the data collection module is shown in Fig. 5. The packet contains a 16-bit sequence number, a source and a destination id and logged experimental result introduced in the last subsection. The size of the structure varies with the *user_data_t* structure. The sequence number is used to distinguish redundant packets. Two commands and two ACKs are dedicated for controlling this module, namely, *CMD_COLLECT*, *CMD_ERASE*, *ACK_COLLECTION* and *ACK_CLDONE*. *CMD_ERASE* is used to erase all the data in the flash of a sensor mote. When a node receives a command *CMD_COLLECT*, it will first return an *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 type of messages. During data collection, the node will read out a logged result and send it via the ACK route introduced in Subsection 3.2. The procedure continues until the entire log data have been read out and the node sends an ACK_CLDONE to the PC.

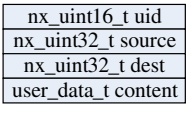

Fig. 5. An example of experimental result message

3.6 Reprogramming

The reprogramming event can be triggered in two ways as indicated in Fig. 2. A user can ask a node to reprogram via the command dissemination module or the event can be triggered using a timer. Using the first method, we can reprogram the node with a new program received from the program dissemination module. The second, we provide a timer in the testbed by which a user can set a node to reprogram from exp-program to TPS program after the timer expires. The expiration of the timer will trigger the reprogramming no matter the state of the experiment. Thus the user is supposed to carefully set the timer.

3.7 Time Synchronization and Neighbor Discovery Services

Many time synchronization protocols are proposed recently. We chose FTSP [5] for our current implementation of FIST. The protocol relies on periodically exchanging synchronization messages between nodes to ensure the network wide synchronization. We added a few commands to control this service, namely, *CMD_STOPSYNC* and *CMD_STARTSYNC*. Both of them have corresponding ACK messages. In the neighbor discovery service, the root node sends ten flooding messages into FIST testbed. Each is identified with a unique id. Once a node receives a new flooding message, it will retransmit it once. Thus every node in the network retransmits a unique message once. With these GUI tools, a user need not have to remember and type different commands to control the testbed, and does not have to look up our instruction document to know the meanings of different feedbacks. It saves much time for the user and fulfill the requirement of convenience and user-friendliness.

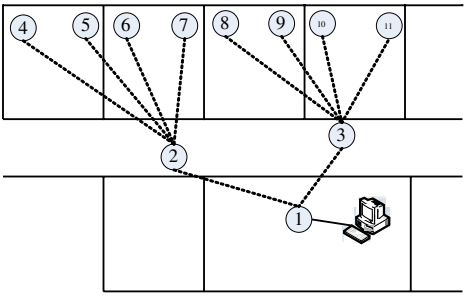

Fig. 6. Deployment of nodes in the case study

4 A Case Study

Due to the flexibility of our testbed we need not to deploy it in a fixed fashion and place. We intended to measure the room temperature of four offices in our university building. The deployment of the testbed is shown in Fig. 6. Nodes are packaged in small boxes which can be easily posted on the walls. Each node is

Table 1. Two program images

Image	Modules and Service	Functions
Reprogram Image	Program Dissemination Command Dissemination Reprogramming	Disseminate program and reprogramming
Control Image	Command Dissemination Data logging Data Collection Reprogramming Synchronization	Run and control the experiment

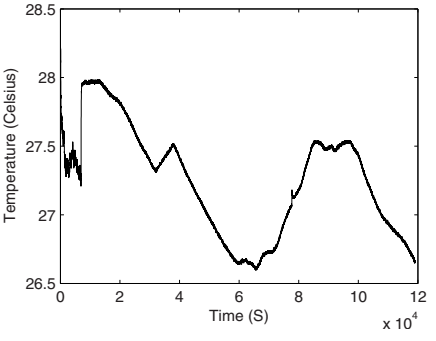

Fig. 7. Temperature measured from environment

programmed with an unique id as shown in the figure. Motes can be powered by batteries or mains. In the latter case, off-the-shelf USB hubs can replenish power from sockets.

The Tmote Sky mote has only 48KB programming flash. A complete version of the TPS program with all the modules and services takes as much as 40KB. The remaining space for exp-program is very limited. Thanks to the modular design of the testbed software, we configured two program images, which are listed in Table 1.

We let the experiment run for approximately one and half days. The temperature is collected every five seconds. All the measurement are tagged with an id and a timestamp then stored in the flash memory. The results are shown in Fig. 7.

5 Conclusions

When more and more applications are to be supported, the sensor networks need to be studied and programs need to be developed on the testbed. We designed a testbed framework called FIST and an account of our baby steps in planning, designing and implementing FIST is given. The main advantage is that it is flexible and transparent to deployment scenarios, location and the

topology. Moreover, it fits our requirements of a testbed yet keeps a low-cost. We plan to reduce the time taken for the code dissemination and data collection on FIST. We are also planning a thorough comparisons with some of the available testbeds.

References

1. Girod, L., Ramanathan, N., Elson, J., Stathopoulos, T., Lukac, M., Estrin, D.: Emstar: A software environment for developing and deploying heterogeneous sensor-actuator networks. ACM Trans. Sen. Netw. 3(3), 13 (2007)
2. Sheu, J.-P., Chang, C.-J., Sun, C.-Y., Hu, W.-K.: Wsntb: A testbed for heterogeneous wireless sensor networks. In: First IEEE International Conference on Ubi-Media Computing, Lanzhou, China, July 31- August 1, pp. 338–343 (2008)
3. Handziski, V., Köpke, A., Willig, A., Wolisz, A.: Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In: REALMAN 2006: Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality, pp. 63–70. ACM Press, New York (2006)
4. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: SenSys 2004: Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 81–94. ACM Press, New York (2004)
5. Maróti, M., Kusy, B., Simon, G., Lédeczi, A.: The fooding time synchroniza- tion protocol. In: SenSys 2004: Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 39–49. ACM Press, New York (2004)