FIST: a framework for flexible and low-cost wireless testbed for sensor networks

Cheng Guo, R. Venkatesha Prasad,
Jiang Jie He, Martin Jacobsson
and Ignas Niemegeers

Author: Please indicate who the corresponding author is.

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

Author: Please reduce abstract of no more than 100 words.

Abstract: Setting up a Wireless Sensor Network (WSN) for an experiment is both time consuming and effort intensive. Often, the time required to set up the experimental framework is even more than that of the experiment itself. Many existing fixed testbeds manage nodes in experimental network using gateways. The gateways are wired into a backbone network through cables. Moreover, testbeds are normally deployed in dedicated spaces and require a lot of support and maintenance. Changing the deployment place – especially if it is a crowded area – is difficult. Further, these testbeds also require extra devices than sensor motes, which could be expensive. We propose a framework for implementing a *F*lexible and low-cost w*I*reless for *S*ensor network *T*estbed (FIST). Our FIST platform consists of only sensor motes. Downloading the experimental code, reprogramming, testbed control, logging and collecting experimental results and synchronisation are all carried out by the sensor motes wirelessly without extra devices. Thus, the testbed can be easily and quickly deployed anywhere. We present our framework and also a case study using FIST.

Keywords: testbed; WSNs; wireless sensor networks; experiment; flexibility; low-cost.

Reference to this paper should be made as follows: Guo, C., Venkatesha Prasad, R., He, J.J., Jacobsson, M. and Niemegeers, I. (xxxx) 'FIST: a framework for flexible and low-cost wireless testbed for sensor networks', *Int. J. Ad Hoc and Ubiquitous Computing*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: AUTHOR PLEASE SUPPLY CAREER HISTORY OF NO MORE THAN 100 WORDS FOR EACH AUTHOR.

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 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 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

Copyright © 200x Inderscience Enterprises Ltd.

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 synchronised and nodes can discover and report their neighbouring information. Therefore, services like time synchronisation and neighbour discovery are also in demand.

Existing testbeds 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, experimental results are passed to the gateways then collected by a data server. In such a testbed, sensor motes only take care of experiments. The control and management are all carried out by the backbone, which consists of servers and gateways wired together. Although this kind of testbeds can reliably carry out experiments, they are not flexible because of the wired backbone. Their physical topology cannot be easily changed. Moving these testbeds from one place to another, especially to the places that are crowd, is difficult and time consuming. Moreover, additional supporting devices, such as gateways and servers, increase cost of a testbed.

In this paper, we propose a new 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 realise 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 fulfil 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.

We present the FIST framework that tries to harness the wireless links already available with the sensor motes to implement over-the-air command and control. We trace here our steps in implementing a testbed, which is different from the ones that are already in use. We provide an account of our design philosophy along with the segregation of different modules. We also present a simple sensor network experiment case study.

In the rest of this paper, Section 2 introduces the existing sensor network testbed in the literature and concludes their features. The design requirement and both hardware and software architecture of FIST are presented in Section 3. Then, Section 4 explains the details of the modules and services in FIST. A case study using FIST is presented in Section 5. Finally, Section 6 concludes the paper.

2 Related work

We first provide an exhaustive overview of various currently available solutions, techniques and implementations. Motelab (Werner-Allen et al., 2005) uses wired backbone to connect 30 MicaZ (http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf) sensor nodes to remote reprogram and monitor permanently powered sensor network nodes. The gateways are Crossbow MIB-600 (http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB600CA_Datasheet.pdf). A database is dedicated to log the experimental data. The testbed has a web interface for multi-users by which users can run experiments in two modes – either schedule a large number of jobs to run unattended or users can access the nodes in real time via the gateways. The gateways are interconnected using the Ethernet. Similar to Motelab, a testbed in CENs systems lab of UCLA (http://www.tinyos.net/ttx-02-2005/testbeds/ttx-testbed-panel-mlukac-v4.PPT) also uses the same structure but changes the sensor motes to Mica2s (http://www.xbow.com/Products/Product_pdf_files/Wireless pdf/MICA2 Datasheet.pdf). In these testbeds, tasks like reprogramming, data collection and command dissemination mentioned in Section 1 are carried out with the help of the backbone. Thus, the testbed cannot be deployed flexibly.

EmStar (Girod et al., 2007) is a software environment to simulate and emulate WSN applications.

It relies on micro-server platforms such as iPAQ (http: //welcome.hp.com/country/us/en/prodserv/handheld.html) or Crossbow Stargate (http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_NetBridge_Datasheet.pdf). It has a library for packet-level communication, tools for both simulations and emulations and services for networking, sensing and time synchronisation. The highlight of this testbed is that it provides 'real-code' simulation and emulation environment, which means that code developed for an application can directly run in the simulation and emulation modes. It provides services like neighbour discovery, time synchronisation and routing. Since command dissemination and data collection are handled by the advanced micro-severs, EmStar focuses on the software environment of the testbed tools and services mentioned

earlier. The sensor motes are used to transmit packets over air thus the wireless channel characteristics are taken into consideration in the emulation.

WSNTB (Sheu et al., 2008) is a testbed developed in the National Tsing Hua University in Taiwan. It consists of components such as a server, gateways and sensor motes. Sensor motes are connected to the gateways, which are linked to server. The logical topology in the testbed has three layers, service interface, testbed core and resource access. The testbed consists of 34 homemade Octopus sensor motes. These motes either have RS232 interface or USB interface. The gateways bridge sensor motes to the wired backbone by converting RS232 or USB interface to Ethernet interface. Local mode allows users to access hardware, i.e., sensor motes directly during experiments. Users first develop code on their own computer. When the access to the testbed is permitted, the users are given permissions to the desired number of motes as if they are connected via serial ports.

TWIST (Handziski et al., 2006) is also a testbed with several tiers. The outer most one is formed by TMote Sky motes (http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf). Then, the inner tiers are super-nodes connected to the motes by USB. Third and the last tier consist of servers such as databases and a control centre, which provide interfaces to users. Sensor motes are programmed and controlled through USB interface. They can be switched on/off during an experiment. The testbed also provides services such as synchronisation. Node ID is configured using the location of the motes. The difference between TWIST and MOTELAB is the usage of super-nodes, which reduces the number of gateways needed in the testbed. Several TMote Sky motes are connected to one super-node, which then connect to the Ethernet backbone. The connections between a super-node and its downstream motes are through USB cables. Command or exp-program is relayed from super-nodes to the motes through the USB cables. By using super-nodes, only a few nodes require gateways thus the cost of the testbed is reduced. Other testbeds, such as WUSTL (http://mobilab.wustl.edu/testbed), also use the same set-up as TWIST. However, the testbed is still not flexible since motes are connected by wires.

KonTest (Iwanicki et al., 2008) is a WSN testbed developed in the Vrij University in Amsterdam. The testbed consists of 60 TelosB (http://www.xbow.com/Products/productdetails.aspx?sid=252) and Tmote Sky sensor motes, 16 USB hubs and 6 PCs. The backbone of the testbed is a USB network. All the motes are connected to PCs by USB hubs. Thus, commands and exp-programs can be disseminated by USB interface. Meanwhile, motes are also powered through USBs.

Indriya (Doddavenkatappa et al., 2009) is another example of multi-tier WSN testbed. Similar to the TWIST testbed, it has TelosB sensor mote on the outmost tier. USB hubs are used to connect motes into clusters. The USB hubs are then connected to mini-PCs, which later are linked to a server by Ethernet. The server takes care of providing a user interface and logging experimental results. To extend the distance between motes, Indriya uses active-USB cables, which can extend the distance between a USB hub and a mote to 35 m. Currently, 127 motes are deployed on ceilings of three floors in an office building. Thus, a three-dimension network is formed.

Kansei (Arora et al., 2006) is a WSN testbed deployed at the Ohio State University. Unlike the above-mentioned testbeds, which are in office buildings, Kansei has its own dedicated space for wireless experiments. The major part of the testbed consists of 210 Extreme Scale motes (http://cast.cse.ohio-state.edu/exscal/), each of which is connected to an Extreme Scale Star-gate (http://cast.cse.ohio-state.edu/exscal/). The Stargates are then connected by Ethernet. Moreover, the Stargates are also equipped with WIFI interface for other wireless experiments. To collect sensor readings from environment, the testbed also has 50 Tmote Sky motes equipped with various sensors. Another feature of Kansei is that it has robots equipped with motors, Extreme Scale motes and Extreme Scale Stargates. Therefore, experiments with mobile nodes can be carried out. Kansei also has a visual user interface for experiment control.

TutorNet (http://enl.usc.edu/projects/tutornet/index.html) is a testbed similar to Indriya, which also connects Tmote Sky and MicaZ motes to gateways by USB cables. However, unlike Indriya, the gateways are not USB hubs and mini-PCs but Stargates. The Stargates are connected by WIFI. Tutornet does not have a visual user interface but is based on command lines.

Table 1 is a summary of hardware used in the testbeds introduced in Section 2. We can see that wired connections are used to connect to the backbone in one way or other in all these testbeds. Various additional devices are also required. The wires and extra devices reduce the flexibility of the testbed and increase the complexity as well as cost. Thus, our aim here is to design a framework that addresses the above-mentioned issues. However, we indeed take cues from these implementations and improve the usability of our testbed. Moreover, we want to design our testbed to be completely flexible and harness the capabilities of the wireless links.

Table 1 Hardware of four testbeds

Testbed	*Sensor node*	*Additional devices*
MoteLab	MICAz	EPRB, MIB600
CENsTestbed	MICA2	MIB600
EmStar	MICA	Compaq iPAQ 3760
WSNTB	Octopus I Octopus II	Single-board computers net4501 and net4801, RS232-to-Ethernet and USB-to-Ethernet converters
TWIST/ WUSTL	Tmote Sky	EyesIFX Linksys USB2HUB4 hub, Network Storage Link for USB2.0 (NSLU2) device

Table 1 Hardware of four testbeds (continued)

Testbed	*Sensor node*	*Additional devices*
KonTest	Tmote Sky and TelosB	USB hubs
Indriya	TelobsB	USB hubs, active USB cables and mini-PCs
Kansei	Extreme Scale motes, Tmote Sky	Extreme Scale Stargate, mobile robots
TutorNet	Tmote Sky and MicaZ	Stargate

3 Requirements and architectures of FIST

3.1 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, neighbour 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 start 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 take this procedure as an example to list the following functions to be incorporated in a testbed.

- We need to have a mechanism by which we can disseminate commands to a particular node, a group of nodes or the whole network.
- 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.
- We have to also disseminate exp-program and reprogram the motes with new programs when required.
- The experimental results have to be logged and reported to the user. If there are bugs or errors in user's program, then testbed has to log it and report them as well.
- Some services such as time synchronisation and neighbour discovery should also be provided.

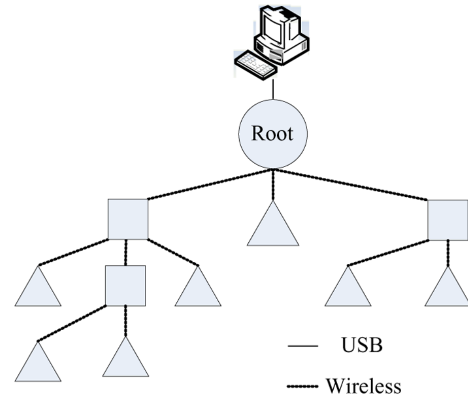
Except for the functional requirements, we also have some non-functional requirements of the proposed testbed. The testbed has to be,

- flexible to deploy and re-deploy in any environment
- reliable thus all the commands or programs to be disseminated into the motes have to be guaranteed to be deliver to the destination; the experimental results have to be successfully collected as well
- convenient, simple and user friendly; a user does not have to remember complex commands or configure the testbed prior to running an experiment.
- low cost so that with a fixed budget, we can then afford more nodes in the testbed.

3.2 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 Figure 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.

Figure 1 Hardware architecture (see online version for colours)



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 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 specialised 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.

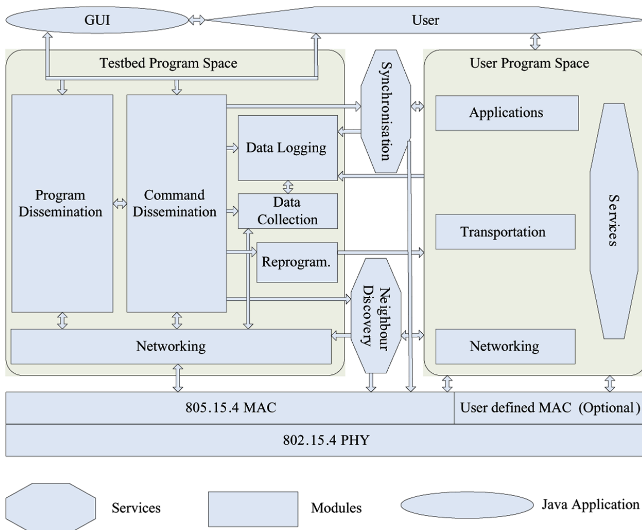
Currently, we choose Tmote Sky motes as our wireless sensor motes for its rich on-board modules, reliable quality

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 8 MHz Texas Instruments MSP430 micro-controller with 10 KB RAM and 48 KB 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 250 kbps at 2.4 GHz and the transmission range of about 50 m indoor and 125 m outdoor.

3.3 Software architecture of FIST

Figure 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 realising the functions of FIST is isolated from users' exp-program so that a user does not have to consider the testbed program when developing 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 synchronisation and neighbour 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.

Figure 2 Modules (see online version for colours)



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 Section 4.

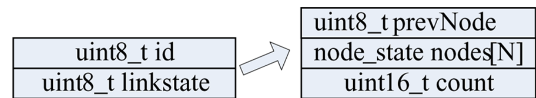
4 Modules and services in FIST

4.1 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 Figure 1, we organise 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 Figure 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 Figure 3(a). The variable 'id' in the structure keeps an 8-bit node id assigned uniquely. The 'linkstate' is used during the instruction dissemination, which will be introduced with dissemination mechanism later. Besides the list, a variable named 'prevNode' is used to save the id of its upstream node. With the node list and 'prevNode', a node can be linked to the network. The capacity of a *node_list* structure is pre-defined. Currently, the *node_list* is configured manually. After a user sets up the testbed, he/she has to configure the connectivity of each node based on the signal strength between each pair of nodes. Although it requires some extra efforts from the user, it increases the reliability of the routes. It is also useful since only the user knows well the deployed environment.

Figure 3 Data structure for configuring tree topology: (a) *node_state* structure and (b) *node_list* structure (see online version for colours)



Since we may have to reprogram the motes from time to time during an experiment, the connectivity configuration of each mote is saved in its permanent flash so that it can be reloaded when the node reboots. By this feature, we do not have to configure the same network every time. Once a node reboots, the module will read out the last configuration from the flash memory automatically. When the configuration changes, the old configuration saved in the flash will be overwritten. This feature provides convenience to the users.

4.2 Command dissemination

The format of command and ACK packets is shown in Figure 4. For different types of commands and ACKs, we define two sets of constants. All the commands have the prefix *CMD_* and the ACK have the prefix *ACK_*, e.g.,

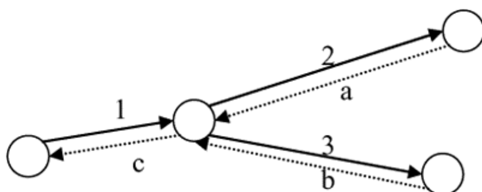
CMD_COLLECT is the command to start data collection while *ACK_COLLECTION* is to acknowledge the command. There are five fields in this format. 'cmd' is the type of commands or ACK. 'data' stores parameters associated with commands or ACK, e.g., command to add a node to the *node_list* has the node id as the parameter. 'error' is a 32-bit variable used with ACK. If an ACK is the feedback for disseminating program, it is used to store the *uidhash, which* is a 32-bit unique number of a program image. For other feedback, it keeps a 32-bit error flag for diagnosing the runtime situation of the framework. 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. If a message is for every node in the network, the 'des' field is set to 0xff.

Figure 4 Format of command packets (see online version for colours)

nx_uint16_t	cmd
nx_uint16_t	data
nx_uint16_t	error
nx_uint16_t	src
nx_uint16_t	des

Commands are broadcasted to downstream nodes in the *node_list*. Since each node only knows its upstream and downstream nodes, they always retransmit a command to all its downstream nodes if there are any. Before sending out commands to nodes in the list, the 'linkstate' of each node in the list are set to 'NOT_RESPONDED'. When an ACK is received, the corresponding nodeś 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. This procedure is executed hop by hop until the root node, which will acknowledge to the PC whether the command has reached all the intended nodes. An example is given in Figure 5 where the dissemination order is from 1 to 3, and the order for feedback is from *a* to *c*. If a downstream node fails to acknowledge before timeout, its upstream node will rebroadcast the message until the maximum retrial time, i.e., 10, is reached. 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 or the connection to that node is too weak. By this mechanism, we can fulfil the reliability requirement, since that in this scheme, a message reaches its destination unless it is physically unreachable, in which case an error is reported to the user.

Figure 5 An example of command dissemination and ACK



In general, there are four kinds of ACKs.

1 *For command dissemination*: The ACK confirms that commands are received by destination(s).

2 *For data collection*: The ACK confirms the start of collection.

3 *For reporting errors*: The ACK reports different kinds of errors to the user for diagnosis.

4 *For confirmation of received program*: The ACK reports which of the nodes have received the new program.

The second and the fourth type will be introduced in the corresponding modules.

4.3 Program dissemination

Program dissemination is also carried out through the network set up by the networking module. Since a program can be up to 48 KB 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 (Hui and Culler, 2004) implementation. In Deluge, a program is split into the so-called *pages* of fixed size, a node transmits one page at a time and the program can be assembled by the receiver after all the pages are collected. The program is stored in the permanent flash in a node with a unique address, with which the node can reprogram itself.

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. In case that one or several pages are not received, a NACK, with the ID of the missed pages, would be sent as a request. Upon receiving it, an upstream node has to retransmit only these pages. As soon as a complete program is received, an event will be triggered to send an ACK to report correct reception of the program. The message carried 32-bit program information. To prevent the message being lost along the route, we provide end-to-end acknowledgement for this special ACK. Before reception of the ACK confirmed by the root node, the sender periodically sends the ACK until maximum number of retry is reached. Although some redundant messages may be generated, we consider the redundant acknowledgements are necessary for ensuring high reliability. The same mechanism is also applied in data collection module, which will be introduced later. Besides the acknowledgement, which is an activity report of receiving a new program, a user can also check the program information of a node via the command dissemination module. Upon receiving the *check_prog* command, a node sends the same 32-bit program information back to the user, however, without the end-to-end acknowledgement since the user can issue the command again actively in case of an ACK failure.

Compared with the dissemination method used in Deluge (Hui and Culler, 2004), our method provides users the information whether and when a node has received a

new program. Besides, each node using Deluge has to periodically advertise its program information so that its neighbour can find out and request a missing program. However, these advertisements may interfere with experimental traffic in a testbed, which is highly undesirable. Certainly, our method also has a cost associated with. In Deluge, a node can be added to a network and automatically receives the newest program due to the advertisements. 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. Although extra steps have to be taken, we consider that the requirement of reliability has a higher priority. Besides, our method does not interfere with the experimental traffic.

4.4 Data logging

Since experimental results are valuable for analysing 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. As we introduced in Subsection 3.2, we have 1 MB flash memory on the TMote Sky motes. However, the actual size that is available for experimental results varies with the number of program images stored in the same memory. Normally, we should have up to 900 kB space. A user can define the structure of the results he/she wants to store. For instance, in Figure 6, the *user_data_t* structure contains a 16-bit integer for temperature value, a 32-bit integer for a timestamp and another 16-bit integer as a packet sequence number. The user is supposed to plan for the amount of data he or she wants to store. In case that the flash memory overflows, an error can be reported to the user and further results he wants to store in the memory would be dropped.

Figure 6 An example of user-defined data structure *user_data_t* (see online version for colours)

nx_uint16_t temp
nx_uint32_t time
nx_uint16_t count

4.5 Data collection

The data collection module also takes advantage of the networking module. The packet format in the data collection module is shown in Figure 7. 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 initialise 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 4.2. To avoid packet loss during the multi-hop transmission, the node repeats transmission of the same result to the PC at regular intervals until receiving a confirmation from the PC or a maximum number of retry is reached. After that, the node sends the next logged result. The sequence number is used to delete redundant data. The procedure continues until the entire log data have been read out and the node sends an ACK_CLDONE to the PC. So, the user can be notified that process is complete and all the data have been gathered.

Figure 7 An example of experimental result message (see online version for colours)

nx_uint16_t uid
nx_uint32_t source
nx_uint32_t dest
user_data_t content

4.6 Reprogramming

The function of this module is, as the name implies, to reprogram a node. The reprogramming event can be triggered in two ways as indicated in Figure 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. Sometime, a user may want to run an experiment on a new MAC layer, in which case our testbed software, which is built on the 802.15.4 MAC, may not work properly. Thus, we have to have a way by which we can reprogram the node back to our TPS program with 802.15.4 MAC to control and manage the testbed later (after the experiments). 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 reprogram module has an interface, which a user has to include in the exp-program. When the experiment starts, the timer has to be called to start at the same time through the interface. 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.

4.7 Time synchronisation and neighbour discovery services

Many time synchronisation protocols are proposed recently. We chose FTSP (Maróti et al., 2004) for our current implementation of FIST. The protocol relies on periodically exchanging synchronisation messages between nodes to

ensure the network-wide synchronisation. However, this traffic may interfere with the experimental traffic. Therefore, we added a few commands to control this service, namely *CMD_STOP_SYNC* and *CMD_ STARTSYNC*. Both of them have corresponding ACK messages. As the names imply, *CMD_STOPSYNC* and *CMD_STARTSYNC,* respectively, stop and start the timer, which fires periodically to send synchronisation messages.

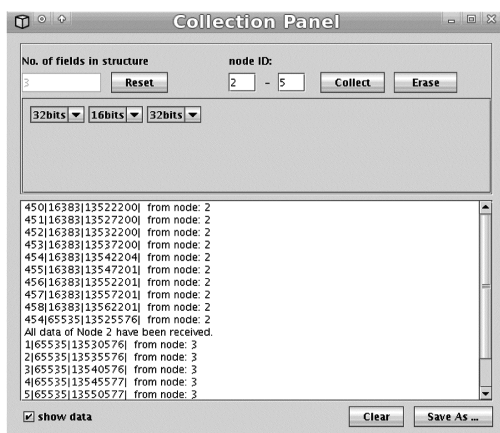
In the neighbour discovery service, the root node sends 10 flooding messages into FIST testbed. Each is identified with a unique id. Once a node receives a new flooding message, it will retransmit it once. Thus, every node in the network retransmits a unique message once. The RSSI value of each received message is stored in the flash memory and can be collected later by the data collection module. The neighbour information can help a user to optimise the network topology of the testbed so that each node has a good link to its upstream and downstream nodes.

4.8 GUI

To provide a user-friendly testbed, we developed GUI in Java to control FIST. There are three panels in the GUI: command, programming and collection. For the sake of brevity, we only show the collection panel in Figure 8. After configuring the panel with right parameters, user can gather logged data in batches and save them in a *text* file. The programming GUI is used to disseminate programs to the testbed and reprogram nodes. It also includes some functions such as checking program image information or deleting an image. The command GUI handles the operations such as sending commands, receiving ACKs and translating them. A small function called node manager is attached to keep the necessary information of all the nodes in the network. When there are many nodes in the network, to identify them, users can give descriptions to them such as their locations and their tasks.

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 fulfil the requirement of convenience and user-friendliness.

Figure 8 Collection panel



5 A case study

Because of the flexibility of our testbed, we need not to deploy it in a fixed fashion and place. We need not introduce the deployment procedure like in Werner-Allen et al. (2005), Girod et al. (2007), Sheu et al. (2008) and Handziski et al. (2006). Instead, we use a case study to illustrate how we carried out an experiment with our testbed using FIST framework. We intended to measure the room temperature of four offices in our university building. The deployment of the testbed is shown in Figure 9. Nodes are packaged in small boxes, which can be easily posted on the walls. Each node is programmed with a 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 48 KB programming flash. A complete version of the TPS program with all the modules and services takes as much as 40 KB. 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 2.

Figure 9 Deployment of nodes in the case study (see online version for colours)

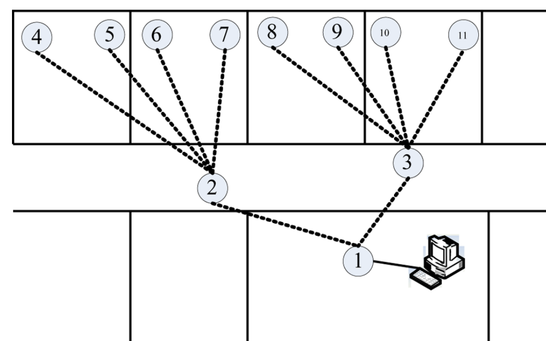


Table 2 Two program images

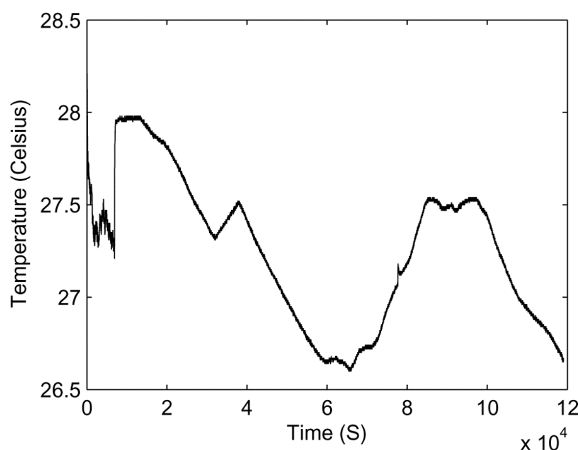
Image	*Modules and service*	*Functions*
Reprogram image	Program dissemination Command Dissemination Reprogramming	Disseminate program and reprogramming
Control image	Command dissemination Data logging Data collection Reprogramming Synchronisation	Run and control the experiment

The two programs can be switched from one to another by the reprogramming module. With the control image, we have 18 KB free space for the exp-program, which is big enough for many experiments. An extreme case is, if required, that users choose to load only data logging, reprogram and synchronisation modules and service with their exp-programs. It can further increase the free space. With the timer function in the reprogram module, they can switch back to the full version of the TPS program to perform data collection and program dissemination after the experiment.

After disseminating programs to the nodes with our GUI, we configured the topology by setting each node their upstream and downstream nodes. This can also be carried out in the command panel in the GUI suit. Then, we turn on the synchronisation since we want the temperature measurements to be timestamped. Since we do not have any wireless traffic during the experiment, we do not have to turn off the synchronisation. Otherwise, a user can do so as introduced in Subsection 4.7.

We let the experiment run for approximately one and half days. The temperature is collected every 5 s. All the measurement are tagged with an id and a timestamp then stored in the flash memory. Both temperature measurement and corresponding id are 16 bits and the timestamp is 32 bits. Therefore, each logged data is 8 bytes. We have 23870 logs stored at each node after the experiment. We put 8 data points in one packet to transport them back to the PC. Eight packets per sec were collected from a node. Thus, it costs us a bit more than 6 min to collect all the data from one node, which is a bit long but still acceptable. A user can also configure their program to compress the data before storing them. For instance, instead of saving the complete 32-bit timestamps, one can save the difference between two consecutive timestamps, which can be fit in 16 bits. The temperature measurement can also be compressed in the same way. Then, the collection time can be much reduced. Figure 10 shows the temperature variation collected from Node 3 in the period. Overall, the temperature did not change heavily in the period, thanks to the air-conditioning system in the building.

Figure 10 Temperature measured from environment



6 Discussions

WSN will be one of the important components of the future automated offices, homes and factories. It will also be an important component of support systems for humans. Thus, 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. To make the experimentation simple, we designed a testbed framework called FIST. This paper gave an account of our baby steps in planning, designing and implementing FIST. We also showed its capabilities and its enhancements over other similar systems. 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 also provided a case study using FIST. We plan to reduce the time taken for the code dissemination and data collection on FIST. We are also planning a thorough comparison with some of the available testbeds.

Compared with simulators such as NS2, GloMoSim or OPNET, FIST has several advantages on accuracy of experimental results. In a simulator, abstractions of physical and link layers have to be made. Since the complex nature of such modelling, there are always deviations in the performances of simulated system from the real implementations. However, the quantity of such deviation is a key question in choosing a simulator or a testbed for protocol evaluation. Moreover, since sensor devices in a WSN always have a poor processing ability and limited memory, simulators may not be able to take these into considerations since they all run on powerful computers with strong processors and theoretically unlimited memory. If we need to simulate these aspects too, the simulator becomes too complex. Consequently, a protocol performs well in simulators may not fit sensor motes since code size and complexity. Nevertheless, simulations are still a valuable tool when designing the protocol for WSNs. Similar to a product design cycle, a solution brought to the experimentation phase always shows some flaws, which were not visible in the designing phase. However, the repeating process of design and testing is too time and effort consuming to be carried out frequently for every change, especially for a WSN application. Even though FIST accelerates experimentation by providing high flexibility, it still requires more time than simulations. Moreover, a real experimentation is hard to be reproduced and thus a statistical result is hard to obtain. Some scenarios are hard or even cannot be configured in an experiment, i.e., a mobile scenario normally requires a lot of people to be involved; a random topology is not possible to be deployed in real environment. In these cases, we have to take advantage of simulation again to save our efforts. Finally, although FIST does not require any additional hardware than sensor motes, some investments have to be made. Contrarily, an open source simulator like NS2 does not cost anything more than a PC. A researcher is advised to balance the trade-off between convenience, cost, complexity and the accuracy of experimental results.

References

Arora, A., Ertin, E., Ramnath, R., Nesterenko, M. and Leal, W. (2006) 'Kansei: a high-fidelity sensing testbed', *IEEE Internet Computing*, Vol. 10, pp.35–47.

Doddavenkatappa, M., Chan, M.C. and A.A.L. (2009) *An Experience of Building Indriya*, National University of Singapore, December, Tech. Rep. [Online], Available: http://indriya.comp.nus.edu.sg/motelab/html/testbed.pdf AUTHOR PLEASE SUPPLY AUTHOR NAME FOR THE HIGHLIGHTED INITIALS.

Girod, L., Ramanathan, N., Elson, J., Stathopoulos, T., Lukac, M. and Estrin, D. (2007) 'Emstar: a software environment for developing and deploying heterogeneous sensor-actuator networks', *ACM Trans. Sen. Netw.*, Vol. 3, No. 3, p.13.

Handziski, V., Köpke, A., Willig, A. and Wolisz, A. (2006) 'Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks', *REAL-MAN '06: Proceedings of the 2nd International Workshop on Multi-Hop Ad Hoc Networks: From Theory to Reality*, ACM, New York, NY, USA, pp.63–70.

Hui, J.W. and Culler, D. (2004) 'The dynamic behavior of a data dissemination protocol for network programming at scale', *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ACM, New York, NY, USA, pp.81–94.

Iwanicki, K., Gaba, A. and van Steen, M. (2008) *KonTest: A Wireless Sensor Network Testbed at Vrije Universiteit Amsterdam*, Vrije Universiteit, August, Tech. Rep. IR-CS-045, Amsterdam, The Netherlands, Available at: http://www.few.vu.nl/~iwanicki/ [Online], Available: http://www.few.vu.nl/~iwanicki/publications/2008-08-TR-045/

Maróti, M., Kusy, B., Simon, G. and Lédeczi, A. (2004) 'The flooding time synchronization protocol', *SenSys '04: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ACM, New York, NY, USA, pp.39–49.

MICA2 datasheet (2008) Crossbow [Online], Available: http://www.xbow.com/Products/Product_pdf_files/Wireless _ pdf/MICA2_Datasheet.pdf

Sheu, J-P., Chang, C-J., Sun, C-Y. and Hu, W-K. (2008) 'Wsntb: a testbed for heterogeneous wireless sensor networks', *Ubi-Media Computing, 2008 First IEEE International Conference on*, Lanzhou, China, 31 August, Vol. 1, pp.338–343. AUTHOR PLEASE CHECK IF THE HIGHLIGHTED CHANGE MADE IS OK.

The wustl wireless sensor network testbed [Online], Available: http://mobilab.wustl.edu/testbed

Tmote sky datasheet (2006) Moteiv Corporation [Online], Available: http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf

Werner-Allen, G., Swieskowski, P. and Welsh, M. (2005) 'Motelab: a wireless sensor network testbed', *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, April, Los Angeles, California, USA, pp.483–488.

Websites

Extreme scale project [Online], Available: http://cast.cse.ohio-state.edu/exscal/

ipaq handhold computers datasheet, HP [Online], Available: http://welcome.hp.com/country/us/en/prodserv/handheld.html

Micaz datasheet, Crossbow [Online], Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf

Mid600 datasheet, Crossbow [Online], Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MIB600CA_Datasheet.pdf

Stargate datasheet, Crossbow [Online], Available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Stargate_NetBridge_Datasheet.pdf

Telosb sensor mote, Crossbow [Online], Available: http://www.xbow.com/Products/productdetails.aspx?sid=252

Tutornet: A tiered wireless sensor network testbed, Embedded Networks Laboratory, USC [Online], Available: http://enl.usc.edu/projects/tutornet/index.html

Wsn testbed, CENs Systems Lab, UCLA [Online], Available: http://www.tinyos.net/ttx-02-2005/testbeds/ttx-testbed-panel-mlukac-v4.PPT