Reprogramming in Heterogeneous Sensor Networks

Vishal Bhatia

Civic Plus, Inc
Manhattan, KS, 66503, USA

Gurdip Singh

Dept of Computing and Information Sciences
Kansas State University
Manhattan, KS, 66503, USA

Abstract— **With advances in sensor networking technology, new applications involving remote and real-time data collection are becoming popular and becoming increasingly deployed. For applications that are deployed in remote locations or deployed at a large scale, over-the-air remote programming may be needed to update the application code executing on the sensor nodes. Deluge is a protocol that provides the capability of remotely re-programming nodes in a wireless sensor networks. This protocol accomplishes reprogramming by injecting messages containing the code image into the network, which are then installed by the sensor nodes. While this protocol is very useful, it is applicable only to homogeneous networks wherein all nodes in the network must be programmed with the same code. This paper proposes a protocol that allows remote reprogramming of only a specific node in the network with a new code image. This allows different nodes to be programmed with different images, which is required in heterogeneous applications. The protocol has been implemented using Java and nesC on a network of sensor nodes. We have conducted extensive experimentation to evaluate the effectiveness and performance of the proposed protocol. We present results to show that our protocol is able to reprogram specific nodes in less time as compared to the original Deluge protocol, and is able to simultaneously deploy multiple applications on different subset of nodes.**

Keywords— Sensor networks, pervasive systems, Remote data collection, communication protocols

I. INTRODUCTION

Advances in communication and computing technologies are enabling deeply embedded, networked systems of sensors that can collect real-time data from a number of different, remote sources. A number of companies such as Crossbow, Intel, and Dust Networks manufacture wireless platforms that can be used for sensing and communication. Crossbow Inc., for example (now Memsic, Inc.[1]), provides wireless modules (motes) for several platform families which include Mica2, MicaZ, TelosB, and IRIS [1]. Each family may have one or more platform boards, each operating at a different frequency range. Figure 1 shows the typically structure of sensor network with motes communicate via wireless links using Zigbee protocol. With wireless sensor networks becoming more prevalent, network of motes are being deployed in remote environments such as monitoring forest fires, movement of vehicles in a battlefield, and environmental monitoring in buildings [2, 4, 6, 8].

The software infrastructure for motes provided by Crossbow allows a mote can be programmed by first connecting the mote directly to a base station (computer) via a serial or a USB port. The complied program image can be downloaded from the computer on the mote. For example, to deploy an application on the sensor network in Figure 1, each mote must be directly first connected to the PC and programmed. However, sensors systems that have already been deployed pose a new challenge. As these systems may have to remain deployed for a period of time, there are situations where the network nodes may have to be re-programmed after deployment. Reprogramming may be needed for several reasons such as software updates or deployment of new algorithms. For such already deployed networks, the re-programming of nodes may have to be accomplished remotely over the air, and in some cases, this might be the only alternative. For example, for a senor network deployed in a forest, directly connecting each mote to the base station may not be an option [3, 5]. Deluge is a protocol that has been developed for such remote re-programming of nodes [7, 9]. This protocol, for instance, allows the computer in Figure 1 to

Figure 1: Architecture of a sensor network

inject messages (containing complied code) into a network of motes without having the motes directly connected to it. The messages are propagated over wireless links to all motes, and after its propagation, all nodes reboot with the new image. While this protocol is very effective and is being used widely, it is applicable only to homogeneous networks wherein all nodes must be programmed with the same code. However, there are applications where sensor nodes in a network may have different functionality and may have to be programmed with different code images.

This paper describes the design and development of a protocol that allows re-programming in heterogeneous networks where different sensor nodes function differently and have to be

programmed with different code images. The proposed protocol is modular in nature in that it uses Deluge as a module, and superimposes control via layering to obtain the desired behavior. This allows our protocol to be compatible with future versions of Deluge. We have designed this protocol and implemented it using combination of JAVA and nesC, an extension of C programming language [1, 10, 11]. JAVA has been used to program the module for the PC whereas the nesC was used to program the sensor nodes (motes). The motes use TinyOS, an event-driven operating system specifically designed for resource-constrained sensor network nodes. The system was tested on a network of micaZ and TelosB motes. Our results show that the proposed protocol is very effective in selectively reprogramming nodes in the network as compared to the situation where the original Deluge protocol.

This paper is organized as follows. The next section describes the Deluge protocol. In Section III, we describe the proposed protocol in detail. Section IV presents our performance results. Finally, we conclude in Section V.

II.   DELUGE PROTOCOL

Deluge is a protocol used for remote re-programming of nodes in a wireless sensor networks by injecting messages into a network of motes without having the motes directly connected to the PC. The basic functionality of Deluge relies on a push-pull based algorithm where every mote periodically spreads a message containing code images over the network. In the following, we describe the basic operation of this protocol.

Deluge uses the 3-way handshake protocol consisting of 3 types of messages: DelugeAdvMsg, DelugeReqMsg and DelugeDataMsg. The protocol starts with the PC injecting a new image into the mote connected to the PC. Note that motes communicate via the Zigbee protocol whereas PCs typically do not have this capability. Hence, at least one mote must be connected to a PC. The motes advertise themselves periodically by giving information about the images they have with them. For this, they advertise their profile to the neighboring motes within the communication range using the message DelugeAdvMsg. This message consists of the following information: image number, version number of the image, image description, the type of image currently present on the mote and some meta-data along with it. When another mote, say M1, receives this message from another mote M2, it sends a DelugeAdvMsg message containing its profile information back to M2. In addition, M1 compares its own profile with the profile of M2 and checks whether it has an obsolete version of the image. If it does, then it sends back what is called a DelugeReqMsg message to the sender. This message contains all the data requested by the requester and is sent only to the mote that requested it and is not broadcast. The node receiving the request message then sends the pages of the code image via DelugeDataMsg messages.

The Deluge package provides a JAVA tool chain with a number of commands for the PC to interact with the mote connected to it. For example, by using the ping command:

java net.tinyos.tools.Deluge –ping

the PC can detect the version number of the image on the node which is directly connected to it. The inject command:

java net.tinyos.tools.Deluge --inject --tosimage=<file> --imgnum=<imgnum>

where tosimage is the image file to be injected and imgnum is the image number to be injected into the network, allows the PC to inject messages containing pages of the code image into the network. Deluge also provides other commands such as reset, erase and Dump.

Deluge provides a reliable mechanism to inject images into the network and has been widely distributed and used. However, it is designed to propagate the same code image to all nodes in the network. Its applicability is restricted, however, when the application is heterogeneous; that is, different nodes in the network have different functionality and have to be programmed with different images. A similar situation arises when the application needs to be executed on a portion of the network. In Deluge, however, the same image is delivered to all motes in the network. It is possible to incorporate different functionalities via Deluge, but it is wasteful in resources. For example, one can incorporate different functionalities as part of the same image, and have different nodes execute different portions of the code image based on their functionality. This, however, requires that the common image (containing all of the functionalities) be delivered to all nodes. This increases the size of the code image, and nodes will contain significant amount of code that they may never execute. Each mote has 4 slots to store images. It is also possible to store different images in each of the slot corresponding to different functionalities. However, in this case again, with Deluge, all images will get delivered to all nodes, and maximum of four different images can be propagated. This paper aims at address this problem by designing and implementing a protocol where different images can be injected into the network targeted for specific motes. This is done in a manner where motes that do not need to be re-programmed are not impacted. This paves way for the possibility of having a large heterogeneous network where different motes are used for different purposes, or different portions of the network can execute different applications.

III.   PROPOSED PROTOCOL

There were two approaches that we investigated in coming up with the proposed solution. The first one involved making changes to the Deluge protocol itself. The main idea in this approach was to identify nodes where re-programming is not required and selectively curtail the processing of messages on these nodes. However, this required making changes to the Deluge message structure to include the address of the target mote to be programmed, and making changes the code for processing of the messages. Since Deluge may be evolving independently, we decided against this approach, as it would not allow easy integration with future versions of Deluge. Rather, we adopted a modular approach where we used Deluge as a stand-alone module and superimposed control via layering to achieve our goal.

The proposed protocol operates in the two phases. During the first phase, we create a path from the source node (the mote connected to the PC) to the target node – the node to be re-programmed. Our goal is for this to be the shortest path so that messages travel with the minimum number of hops to the target (destination) node. During the second phase, we start the Deluge protocol. However, we use a superimposition technique whereby the execution of Deluge is restricted only to the path identified in the first phase.

In the following, we explain the two phases of our protocol in more detail. In the first phase, an Explore message is sent by the base station to its neighbors. This message contains a 'Destination' field that contains id of the destination mote that has to be re-programmed. As the location of the mote is unknown in the beginning, the message has to be broadcasted throughout the network. The Explore message also has an additional hop-count field, which is initially set to 0. Each mote also maintain two variables: parent and min-count, where parent is the id of the mote from which the Explore message with the least hop-count so far has been received, and min-count is that hop-count value. We will now explain the actions which are carried out when a mode M1 receive an Explore message from another mote M2. Mote M1 first checks whether the message is from the base station. If so, then it compares the hop-count in the message with min-count. If the hop-count in
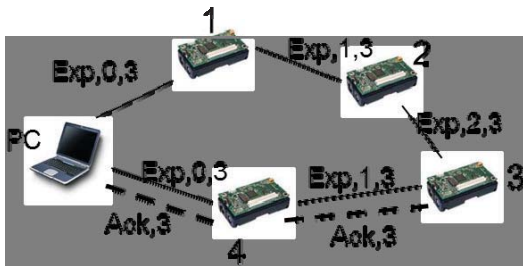

Figure 2: Illustration of the algorithm

the message is smaller, then the parent and min-count variables are updated. Subsequently, M1 checks whether the 'Destination' field matches its own address. If yes, then M2 knows that it is the destination and will send back an ACK message to the base station. However, if it is not the destination and it has not already forwarded the message, then it forwards the message to all of its neighbors with the hop-count increased by one. For example, in Figure 2, the PC sends an Explore (abbreviated as Exp) message with hop count of 0 and destination as 3. When this message is received by mote 1, it is forwarded to its neighbor 2 with hop count as 1.

Now, when the Explore message reaches the destination, the destination node needs to respond to the base station with an ACK message. However, before sending this message, the destination mote waits for a certain period of time to allow messages along different paths to arrive. Normally, messages along the shortest path would arrive first; however, due to network delays, this may not be the case. Rather than developing an extensive (and more expensive) shortest path algorithm, we simply allow the destination node to wait for a timeout period. During this period, it keeps track of the node from which the message with the least hop-count has been

received. After the timeout period, the ACK message is sent by the destination to its parent node. In addition, it also sets another variable participant_type to Destination. For example, in Figure 2, the destination node 3 may receive the Explore message from node 2 with hop-count 2 first. However, when it receives the Explore message from 4, it see that that the message has a lower hop-count which results in node 3 designating node 4 as its parent. Thus, the ACK message will be sent by 3 to 4. When a mote receives the ACK message, it sets the variable participant_type to Forwarder, and propagates the message to its parent. Hence, the ACK message will traverse a path along which the messages with least hop-count were received. Finally, when the message reaches the base station, it sets the participant_type to Source.

At the beginning of the second phase, we already have a path established such that all nodes along this path have the participant_type set to either Source, Forwarder or Destination, whereas all other nodes has been variable undefined (default value). For example, in Figure 2, we will have node 4 as Forwarder, node 3 as Destination and the PC as Source. Nodes 1 and 2 will have the participant_type variable undefined. To start the second phase, the Source node initiates the Deluge protocol. We have written a nesC component that acts as a filter (lower layer) for Deluge. When a message meant for the Deluge protocol is received from the network, it is first processed by this filter component. If the participant_type is undefined for a node, then messages received for the Deluge protocol are dropped by the filter component. As a result, we now have the execution of the Deluge protocol restricted to the just the direct path from the base station to the destination. This avoids the propagation of the code image to other nodes in the network. A common concern when restricting the execution of an existing protocol is the possibility of introducing deadlocks. In this case, we have shown that the restricted execution of Deluge does not cause any deadlocks. For example, in Figure 2, only nodes 3 and 4, along with the PC, will be involved in execution of Deluge. All other nodes will remain passive. In the original Deluge protocol, all nodes in the network are involved by default. Hence, in situations where the network is very large but only one nearby node has to be programmed, Deluge will involve all nodes, which is avoided by our approach.

IV. PERFORMANCE EVALUATION

We implemented the protocol using JAVA and nesC, which is an extension of C programming language designed for programming motes. The program on the PC whose main function is to interact with the mote connected directly to it has been written in JAVA. The protocol was tested using TelosB motes that have the following specifications:

- 250 kbps, high data rate radio.

- TI MSP430 microcontroller with 10kB RAM

The experiment was conducted with 3 different applications from the TinyOS software distribution injected into the network. These applications were chosen because of the difference in the number of pages they contain. This allowed us to have variation in testing based on the size of the messages being injected. As we wanted to test the system with the lab

environment, we reduced the transmission range by setting the RF Power to 2. Three types of scenarios were considered to check the performance for the applications outline above:

Scenario 1: A network with 4 motes, all connected directly to the Base Station resulting in hop-count of 1

Scenario2: A network with 8 eight motes in which the destination mote is at hop count 2 (resulting in one forwarding node)

Scenario3: A network with 8 eight motes in which the destination mote is at hop count 3 (resulting in two forwarding nodes)

Tables 1, 2 and 3 give the results for the three scenarios respectively. We have configured the system to test the performance of the original protocol as well as the proposed protocol. For this, we have two versions of the inject command, one in which we specify the id of the specific mote to be re-programmed and the other in which we do not specify an id. The difference is that when mote id is specified, our version of the protocol with two phases is started; otherwise, the original

Deluge protocol is used. We set a timeout period of 10 minutes so that if the reprogramming took longer than this time period, we have stopped it. Since our goal is mainly to compare the performances of the original and the proposed protocol (and not scalability studies), this early stopping was sufficient to illustrate our performance comparisons.

In Scenario 1, there are four nodes and all nodes are connected directly to the base station. As can be seen in Table 1, we do not see much difference between the performances of the two protocols. However, for the last case of the application with more number of pages, the original protocol takes longer. Note that in the original protocol, all nodes have to be programmed whereas in the proposal protocol, only the single targeted node is reprogramed.

In Scenario 2, we had eight motes in the network and the target mote to be re-programmed is two hops away from the base station. As can be seen, with the original protocol, it takes more than 10 minutes for each of the applications. However, since a single mote gets re-programmed in the proposed protocol, we are able to program the node in less than 3

Table 1: Performance for Scenario 1

Table 2: Performance for Scenario 2

Table 3: Performance for Scenario 3

minutes for each of the application. In our case, the execution of the protocol is just restricted to the three nodes (base station, forwarder and the destination). A similar observation is made in Scenario 3 as shown in Table 3. In this case, the destination node is three hops away. Hence, it takes proportionally longer to reprogram as compared to Scenario 2. However, we observed that the original algorithm still took more than 10 minutes. Note that if more nodes are added to the network while keeping the distance of the destination remains the same, the original protocol will take even longer to execute whereas the execution time of the proposed protocol will remain the same.

In addition to the time required to reprogram, the proposed protocol also utilizes the available memory more effectively. Each mote has four slots into which different images can be stored. With the original Deluge protocol, there can be a maximum of four application images deployed in the network. However, with the proposed protocol, more than four applications can be deployed in cases where different applications involve different subset of nodes in the network.

Apart from performance testing, we have conducted extensive testing to evaluate the correctness and advantages of the protocol. For example, one such test case was the following. In a network of 8 motes, we injected the Blink application with the target node with id 8. At the same time, we also started injection of the Oscilloscope application with target node as 1. This was done to determine whether more than two nodes can be programmed concurrently with different applications, which is not possible in the original Deluge protocol. To check the correctness of the protocol, we rebooted nodes 1 and 8 to check that the Blink and OscillscopeRF applications were deployed properly which we found was the case. It is also important to note that if we are injecting a fresh image to the same mote, we do not need to set the path again.

V. CONLCUSION

In this paper, we have presented a protocol that can be used to reprogram specific nodes in a sensor network without impacting other nodes in the system. The protocol is appropriate for sensor network in which heterogeneous applications have to be deployed – those in which different nodes have different functionality requiring different code images. This protocol uses the original Deluge protocol as a stand-alone module and is compatible with any future variations of Deluge. We have shown via extensive experimentation that the proposed protocol can reprogram specific nodes using less time and resources as compared to the original protocol which reprograms all nodes in the systems by default.

REFERENCES

[1] MEMSIC web site. http://www.memsic.com.

[2] I. Akyildiz, T. Melodia, and K. Chowdhury. A survey on wireless multimedia sensor networks. Computer Networks (Elsevier) Journal, 51(4), 2007

[3] Q. Wang, Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: Challenges and approaches. IEEE Network Magazine, 20(3):48–55, May-June 2006.

[4] Warren, Steve, Luke Nagl, Scott Schoenig, Balakumar Krishnamurthi, Tammi Epp, Howard Erickson, David Poole, Mark Spire, and Daniel Andresen. "Veterinary Telemedicine: Wearable and Wireless Systems for Cattle Health Assessment," 10th Annual Meeting of the American Telemedicine Association, Colorado Convention Center, Denver, CO, April 17–20, 2005. Poster presentation. Abstract published in Telemedicine and e-Health, Vol. 11, No. 2, April 2005, pp. 264-265.

[5] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In 26th IEEE Real-Time Systems Symposium, 2005.

[6] G.Singh, S. Pujar and S. Das, Rate-based Data Propagation in Sensor Networks, IEEE Wireless Communication and Networking Conference, March 2004.

[7] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In SenSys'04, Baltimore, Maryland, USA, Nov. 2004.

[8] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In Proceedings of ICDCS, 2003.

[9] A. Chlipala, J. Hui, and G. Tolle. Deluge: Data dissemination for network reprogramming at scale. Class Project, http://www.cs.berkeley.edu/~jwhui/research/deluge/cs262/cs262a-report.pdf, Fall 2003.

[10] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).

[11] TinyOS website http://www.tinyos.net