International Journal of Interdisciplinary Innovative Research & Development (IJIIRD)
ISSN: 2456-236X
Vol. 01 Special Issue 02 | 2017

Sensor Operated Vice

Neha Bhagwat, Swati Mandurkar .Mandar Deshmukh
Department Of Computer Engg
Manav School Of Engineering and Technology, Akola

Abstract

A wireless sensor network (WSN) is a wireless network consisting of spatially distributed autonomous devices us-ing sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations. Wireless sensor networks gather data from places where it is difficult for humans to reach and once they are deployed, they work on their own and serve the data for which they are deployed. When the environment changes, sensor network should change too. This paper is an attempt to de-vise an efficient, robust and stable solution for the problem of remote reprogramming of wireless sensor networks and trying to address some of the problems associated with attempts taken by other researchers such as Network Reprogramming, Sensor reconfiguration and Supporting Tools.

Key Words: Wireless Sensor Network, Wireless Sensor, Network

1. Introduction

A wireless sensor network (WSN) is a wireless network consisting of spatially distributed au-tonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations.[Romer (2004)][Haenselmann (2006)] . Originally developed as a military application for battlefield surveil-lance , wireless sensor network has been an area of active research with many civilian application covering areas such as environment and habitat monitoring, traffic control, vehicle and vessel mon-itoring, fire detection, object tracking, smart building, home automation, etc are but few examples [Hadim (2006)][LEWIS (2004)][Mainwaring et al. (2002)]

Wireless sensor networks gather data from places where it is difficult for humans to reach and once they are deployed, they work on their own and serve the data for which they are deployed. When the environment changes, sensor network should change too. For an example , it is meaningless, if the sensor network is collecting data of rainfall in the months of January-March in India. However, the same network could be utilized to gather temperature data for the same period. Or at least we should stop retrieving data of rainfall. And also, the aggregation function ought to be changed from ”Send the data continuously”, to ”Send the data if it rains”. Since bug fixes and regular code updates are common to any software development life cycle as one goes through a number of analysis-design-implementation-testing iterations, there is also a need to reconfigure the nodes so that they can keep generating relevant information for us.

It is not feasible to collect each and every sensor node which is deployed and reconfigure it for our needs. Hence a set of protocols, applications and operating system support are needed to reconfigure wireless sensor networks remotely. The ability to add new functionality or replace an existing functionality with a new one in order to change the sensor behavior totally, without having to physically reach each individual node, is an important service even at the limited scale at which current sensor networks are deployed. TinyOS supports single-hop over-the-air reprogramming, but the need to reconfigure or reprogram sensors in a multihop network will become particularly critical as sensor a network grows and moves toward larger deployment sizes. Hence, this paper reports an attempt to develop suitable protocols and techniques to achieve reconfigurability of sensor networks with minimal human intervention. As such , the problem can be defined as follows:

2. Background

The problems specified above have been studied by many researchers in various ways. In this sec-tion we survey those works which directly address the concerned topics. We also discuss various problems and shortcomings associated with the the current approached.

2.1. Network Reprogramming

TinyOS 1.0 [Jeong (2003)] already supports Network Reprogramming for the Mica-2 motes. But the support is for single-hop reprogramming only. It uses a NACK based broadcasting protocol for code dissemination. The Base station breaks the code to be transmitted into small units known as capsules. It then transmits these code capsules to all nodes within its broadcast range. After the entire code image has been transmitted, the base station polls each node for missing capsules. Nodes scan the received code in EEPROM to find gaps. They then reply with NACKs if gaps are found. The base station unicasts the missing capsules to particular node.

TinyOS 2.0 also supports In-Network Reprogramming of wireless sensor networks. This sup-port is available mainly for telos and micaZ platforms. It implements the Deluge algorithm [Hui (2004)] for code dissemination and for remote reprogramming of wireless sensor networks. Multi-hop reprogramming is supported with the use of Deluge. We, in our literature, analyze Deluge for the problems faced by it when it is scaled to highly dense networks and try to solve these problems with our approach.

A completely different mechanism for implementation of reprogramming has been used in [Dunkels et al. (2006)]. It uses runtime dynamic linking for reprogramming wireless sensor net-works. This approach uses standard ELF object file format of Contiki [Dunkels (2004)] operating systems for sensor networks since it supports loadable modules. They have ported Java virtual ma-chine from lejOS [Dunkels (2004)] to the Contiki operating system. So, only native code is sent to the destination machine which is then linked and loaded dynamically by the Contiki operating sys-tem. Barring the overhead of running a virtual machine and linking code dynamically, this approach further optimizes transmission cost since it reduces the size of code to be transmitted.

2.2. Code Dissemination

MHOP (MultiHop Over-The-Air Programming) is used in [Stathopoulos (2003)]. This approach uses a Ripple dissemination protocol, unicast retransmission policy and Sliding Window for segment management. In this protocol, nodes transfer the data in a neighborhood-by-neighborhood basis. In essence this implies a single-hop mechanism that can be recursively extended to multi-hop. At each neighborhood, only a small subset (preferably, only one) of the nodes is the ‘source’ while the rest are the receivers. Here, when a node is having a code update, it will advertise its code and those nodes which are interested in updating the code, will subscribe to that node.

The Problem with this approach is that it cannot handle delayed subscriptions. In case of delayed subscriptions, if no other advertisement reaches the node, it will not be able to update the code.

Deluge [Hui (2004)] proposes a trickle [Levis et al. (2004)] based epidemic protocol [16] for reliable multihop code dissemination. Trickle is a protocol for maintaining code updates for WSN. Here, nodes stay up-to-date by periodically broadcasting a code summary to their neighbors. Deluge builds directly off Trickle, adding support for the dissemination of large data objects with a three-phase (advertise-request-data) handshaking protocol.

2.3. Sensor Reconfiguration

TinyOS 2.x documentation specifies sensor configuration . Configuration data is stored on non-volatile storage space i.e. EEPROM of the mote. The sensor node is supposed to read it and take appropriate action. This configuration data possesses the following characteristics:

They are conservative in size between a few tens and a couple of hundred bytes. Their values may be non-uniform across nodes. Sometimes, their values are unknown prior to deployment in the field Their values can be hardware-specific, rather than being tied to the software running on a node.

2.4. Dynamic Loading of Existing Code (on EPROM of mote)

The literature available rarely addresses the mentioned issue directly. But algorithms specified in [2, 3, 4] are useful for downloading remote code. This mechanism can be slightly modified by removing the code download part and replacing it with receiving instructions to load existing code and also receive metadata capsules containing information about location of the code to be loaded and parameters to be passed to it before loading. Then load the specified code into the memory with those parameters.
TinyOS 2.x documentation discusses storage management, in which it specifies the sensor con-figuration and provides interfaces (APIs in normal computer science terms) to access non-volatile storage of motes. It also talks about the boot loader which is an application which runs on every mote boot and copies code from specified EEPROM location to program memory of the mote, if instructed to do so. Then it jumps to the program memory where the code of the mote application to be started is stored. This boot loader is quite different from the boot loader program provided with TinyOS 1.x. The latter was a special application which runs in kernel space of the mote, which has special privileges to access program memory of the mote. The boot loader of TinyOS2.x is imple-mented under /opt/tinyos-2.x/tos/lib/tosboot and is described in detail in section 5, Design & Implementation.

2.5. Data Dissemination Algorithm

Dissemination is also used to float sensor information throughout the network. Adaptive protocols for Information Dissemination are specified in [Rabiner (1999)]. In this paper, the author has presented a family of adaptive protocols called SPIN (Sensor Protocol for Information via Negotiation.) that efficiently disseminates information among sensors in an energy deficient wireless sensor network. According to this protocol, the sensor nodes which use this protocol name their data using high level data descriptors called meta-data. They use meta-data negotiations to eliminate the transmission of redundant data throughout the network. In addition, SPIN nodes can base their communication deci-sions both upon knowledge of resources that are available to them. This allows sensors to efficiently distribute data given a limited energy supply.
Also, the authors of [Rabiner (1999)] analyze traditional approaches like classic flooding. They encounter following deficiencies with these approaches.

(1) Implosion: A node always broadcasts the data to its neighbors even if they already have a copy of same data.
(2) Overlap: Since multiple sensors cover overlapping geographical data, they gather overlapping piece of sensor data many a times.
(3) Resource Blindness: The nodes do not modify their activities based on the amount of energy available to them at a given point of time. Hence they degrade exponentially with energy avail-able with them.

In order to tackle these issues with classic flooding, [Rabiner (1999)] introduced two innovations:

(1) Negotiation: To overcome the problem of implosion and overlap SPIN nodes negotiate with each other before transmitting the data. This ensures that only useful data is being transferred.

2.6. Security and Code Authentication

The authors of [Deng (2006)] touch perhaps the most important factor, security. To avoid reprogram-ming false or viral code images, each sensor node needs to efficiently authenticate its received code image before using it and propagating it. Public key schemes based on elliptic curve cryptography are feasible in WSNs, yet are still very expensive in terms of memory and CPU consumption. In this paper, the author proposes a hybrid mechanism that combines the speedy verification of hash schemes with the strong authenticity of public key

schemes. A hash tree is computed from packe-tized code and its root is signed by the public key of the base station. Each sensor node can quickly authenticate the data packet as soon as it is received. They also show by simulation that the proposed secure reprogramming scheme adds only a modest amount of overhead to a conventional non-secure reprogramming scheme, Deluge, and is therefore feasible and practical in a WSN.

2.7. Writing Reprogrammable Code

TinyOS documentation specifies how single hop reprogramming facility is done in TinyOS. Network reprogramming consists of mote modules and a Java program on a PC host. On the mote side, the XnpM module handles most of the functions like program download and query. The main application needs to be wired to the XnpC module. On the PC side, the Xnp Java program sends program code and commands through radio. Messages of reversed message ID (47) are transferred between mote and PC.
TinyOS documentation also specifies how to write code for Dynamic Reprogramming of sensor networks. It describe a 3-step reprogramming: Download Phase, Query Phase and Reprogram Phase. A Brief description of each phase is given below:

(1) Download Phase: Network reprogramming starts with the Xnp Java program telling the start of download:

(a) Start of download: Network reprogramming starts with the download start message: Xnp Java program sends a request and XnpM relays this request to the main module.

(b) Download: After sending start of download message a couple of times, Xnp Java program sends each line of program as a capsule. The XnpM module on the mote side receives this capsule and stores in EEPROM.

(2) Query Phase Once Xnp java program finishes sending program capsules, it sends download terminate message to notify the end of download. Then, the mote searches for missing capsules in its EEPROM and asks the retransmission of it to PC side. This is done in the following steps:

(a) The Java program asks motes for missing capsules.
(b) Each mote scans its EEPROM and requests the retransmission of the next missing capsule.
(c) In response, the Java program sends the missing capsule.

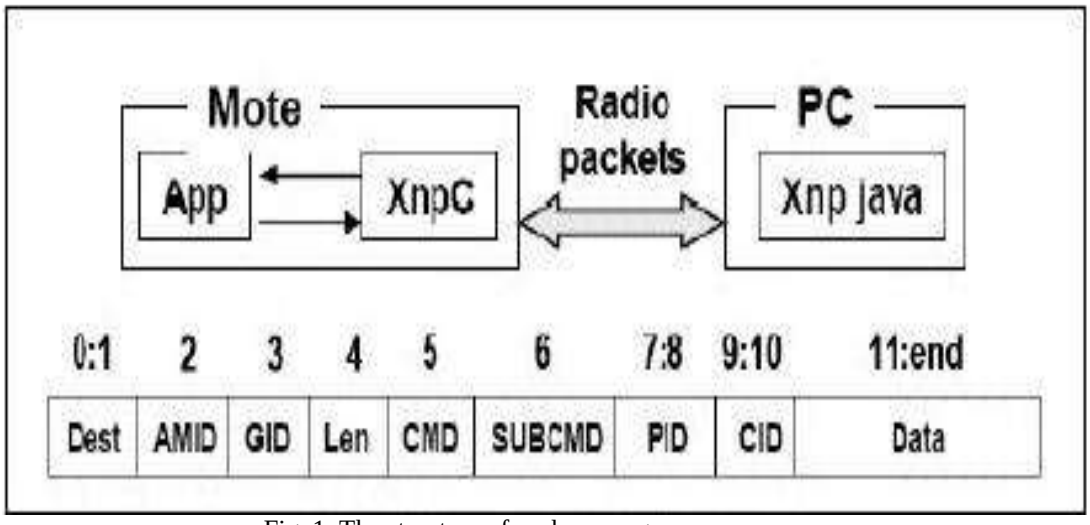

Fig. 1. The structure of each message

(d) Other motes can also fill the hole as well as the requestor i.e. they can also send the missing capsule to the requestor.

(3) Reprogram Phase: In the reprogram phase, the downloaded code is transferred to the program memory and the mote starts the new program. The reprogram phase works in the following steps:

(a) First, the Java program sends a reprogram request.
(b) the XnpM module transfers control to the boot loader.
(c) The boot loader copies the code in EEPROM to program memory and Reboots the system.

3. Algorithm

In this section we develop a new algorithm for code dissemination and in the next section we ana-lyze the algorithm. We name the new algorithm as ”Tree Based Algorithm for Code Dissemination for Dynamic Reprogramming of Wireless Sensor Networks”. This algorithm avoids the problems created by Deluge and MHOP and produces a new approach by establishing source centric fixed topology, obeying strict rules in order to meet reliability requirements. In essence, the algorithm first establishes the topology of the network dynamically and then performs the code dissemination using the information of the topology.

3.1. Goals

(1) Multihop mechanism for code update.
(2) Disseminate code with high reliability.
(3) Propagate code update in the whole network.
(4) Minimize latency.
(5) Maximize data rate.
(6) Minimize redundancy.
(7) Fault tolerant.
(8) Missing Packet recovery.
(9) Try to remove limitations faced by Deluge and MHOP.

3.2. Assumptions

The tree based algorithm makes certain assumptions:

(1) There is only one sink i.e. only a single base station is transmitting code updates.
(2) All nodes are having equal transmission and receiving range and they transfer with equal power.
(3) All nodes are given a unique id and each node knows its own id.
(4) It is a uniformly spread homogeneous network.
(5) Sink is having omni-directional antenna.

3.3. Actual Algorithm

The algorithm takes the system through three stages:

(1) Topology Establishment.
(2) Code Dissemination.
(3) Reprogram Phase.

3.3.1. Topology Establishment

The first stage, Topology Establishment, establishes a tree topology for the entire network. Thus, each node is aware of its parent. The sink / base station is the root of the tree. The advantage of having a definite topology is that one can leverage the extra information about the topology to build efficient algorithms for code dissemination. The topology establishment is done using the following steps.

(1) PC Side Java interface instructs the network to initiate the reprogramming process. It does so by intruding Topology Establishment Object into the network through the base station connected to it through Universal Asynchronous Receive/Transmit (UART) interface.
(2) The sink/ base station initiates the reprogramming process. It broadcasts a special object known as Topology Establishment Object, containing code version information, code size and other meta-code information.
(3) The nodes, as soon as they receive the object, broadcast it, sending the reply back to sink.
(4) Also they will save the meta-code information while forwarding the packets.
(5) Whenever a node receives an object / a packet, it checks for code version information / packet identifier. If it has already received the packet, it will not forward the same, since it has already been forwarded by it. Though, it should forward the packet if it has not yet forwarded it.

3.3.2. Code Dissemination

The second stage of the algorithm is code dissemination. In essence, it uses the knowledge of the topology, gained in the first stage, to cascade the code from the root of the tree to the leaves. However, we have to first contend with a problem which relates to the convergence of the topology establish-ment stage. We first discuss the problem and then carry on the discussion with the algorithm for code dissemination.

(1) Problem of identification of convergence: While doing topology establishment, we come across a basic problem viz. - how the sink will know that the whole network is reached and topology has been established and now it should start disseminating the code. The best option is to ignore the problem, since knowing this does not add up to the algorithm, neither would ignoring this affect the reliability of the algorithm. In short, as soon as the sink receives replies for first packet sent, it should start the dissemination process.
(2) So sink, as it receives the replies for first packet sent, starts code dissemination process.
(3) So sink starts broadcasting the actual code packets.
(4) When an internal node receives a code packet, it keeps a copy for itself and forwards the same. While forwarding, it uses serial unicast (a substitute for multicast in wireless sensor networks) or multicast (if an efficient multicast mechanism other than mobicast i.e. spatial multicast) is devised.
(5) Also these nodes, upon receiving the packet, reply to their parent as an acknowledgment.
(6) When all code packets are transmitted, the base station transmits Code Download Complete object through the network.
(7) When a node receives this object, it executes failure model of the algorithm, in case of missing / incomplete downloads.
(8) If there are no failure cases, the mote executes the reprogram phase of the algorithm.

3.3.3. Reprogram Phase

This is the final stage of the total process. Each mote, after receiving the complete set of packets, will now install it. The following steps are used for this purpose.

(1) When a node stores a copy of the code for itself, it does so in steps. First it extracts the code from the received packet.
(2) Then it buffers it to optimize read / write to EEPROM operations.
(3) Also the code which is received by a node is in SREC format which is quite different from what can be understood by the TinyOS boot loader.
(4) So it is a design decision whether to construct the required code format at PC side or at mote side. We do it on the mote side.
(5) It also checks the request type (whether to reconfigure or reprogram).
(6) In case of reprogram, the node converts incoming packets to the required format and stores them in the EEPROM. It then configures the boot loader with the required parameters and reboots the mote.
(7) In case of reconfigure, the mote just overwrites the existing configuration. There is no need to reboot the mote in this case.

3.4. Failure Model

The main advantage of this protocol is its fault tolerance, i.e. its detection of failed nodes and reestab-lishment of the topology. The failure model considers the following points of failure:

(1) A node fails before it replies to the Topology Establishment Object.
(2) A node fails after it has replied to Topology Establishment Object.
(3) The parent of a node fails after it has received incomplete code image.

3.4.1. A node fails before it replies to Topology Establishment Object

When a node fails during topology establishment phase, before it replies to Topology Establishment Object (TEO), it is simply out of picture. This happens because it will not reply to the TEO and parent has no way to find out its existence. The nodes which are accessible through the failed node will be accessible through some other node in a densely populated wireless sensor network. The protocol will ensure a seamless operation unaware of the node failure.

3.4.2. A node fails before it has replied to the Topology Establishment Object

The exact scenario here is, the node has already replied to the TEO. It is all set and ready to get updated. Now, all of a sudden, it fails. In this case, the parent will not be able to receive acknowl-edgments for code update / reprogram packets. In this case, the parent broadcasts a special ”Node Failure Notification Object” (NFNO) which is flooded through the rest of the network. The NFNO contains all the information which was there in the TEO. In addition to that, it also contains informa-tion of the failed node. When a node receives NFNO, it checks the whole information and matches it with its own bookkeeping information. If the failed node information matches with its parent, it updates the current sender as its new parent.

3.4.3. Parent of a node fails after it has received incomplete code image

The node here receives incomplete code image and then its parent suffers failure. In this case the node timer times out. The timer value is set by node based on code size and other runtime parameters. The

parameters can be node density, node receive queue size etc. In this case, the node generates the Node Failure Notification Object (NFNO), with failure type=parent. This information also contains current code information along with incomplete code information and information about missing data.
If a node receives this NFNO, it extracts the information embedded within it and if it is able to fulfill the desires of the sender, it immediately replies to the sender saying so. In this case it does not forward the packet and avoids unnecessary flooding of the packets through the network. This is required because the same NFNO

message may be received by several nodes in the neighborhood and only one of them should become the parent. Thus, the node whose parent has failed (i.e. the node that had sent the NFNO) will receive several responses from other nodes in its neighborhood. It will choose that node as its new parent from which it received the first response.

3.5. Illustration

3.5.1. Topology Establishment

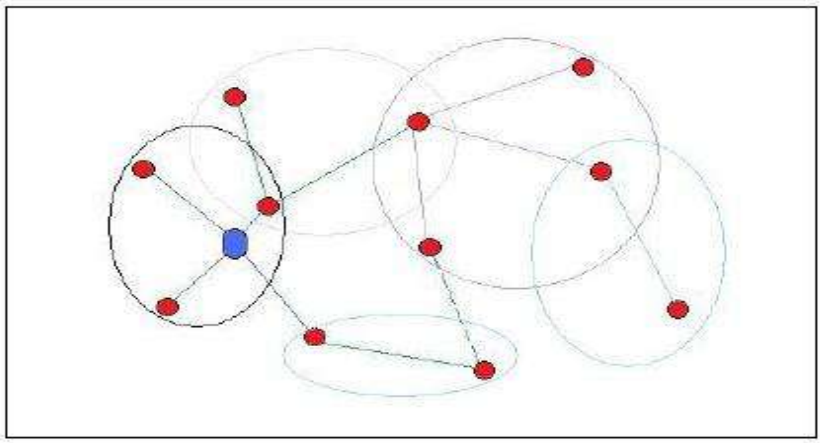

Fig. 2. A random topology with tree based topology established

We now try to illustrate the above mentioned algorithm with a random topology (Fig. 2), which is the general nature of wireless sensor networks. Here, the blue ellipse is sink / base station with sensor nodes surrounded by it as red circles. As per the algorithm, the sink initiates the process and broadcasts the TEO. It is captured by all the nodes in its range and forwarded further. And so the protocol runs. In case where a node is reachable through more than one node (dotted link), the node receives the first packet and replies to its sender. The packet which is received later is simply ignored. Thus, each node knows the id of its parent as well as those of its children. It is easy to infer that the total message complexity for the topology establishment process is O(N) where N is the number of nodes. This is so because each node broadcasts the message once.

4. Algorithm analysis

4.1. Simulation Results

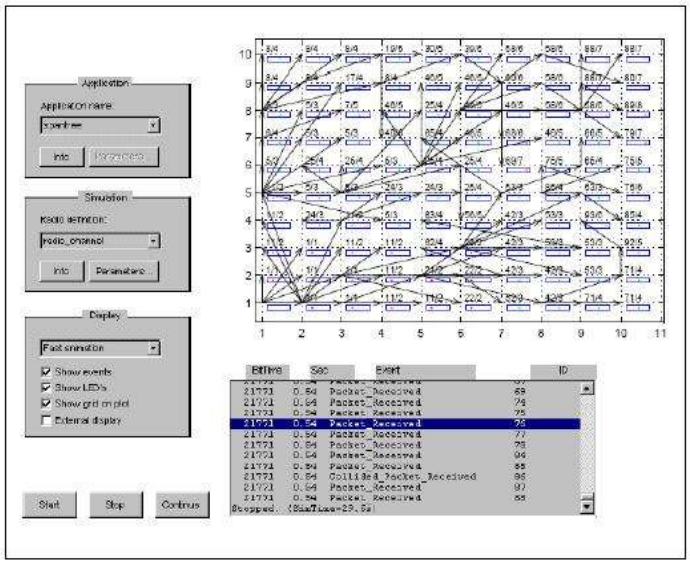

Fig. 3. Prowler Screens hot

We tried to simulate Topology Establishment part of the algorithm with Prowler, a probabilistic simulator for wireless sensor networks. We assumed simple grid topology and checked the hop count given by it for that topology. The screens hot below depicts one run of the system. It can be seen in the simulation, that number of hops required to cover whole network is 8. For this topology, we will try to calculate the hop count by the result we have derived. Total area of the network: 100 units. Maximum area covered by transmission range of a node: 16 units (rectangular area considered for simplicity). Hence number of hops = 100=16 = 6:25. Since the number of hops is an integer, so we require a minimum of 7 hops which is close to our practical result, that is 8 hops.

5. Design and Implementation

The design and implementation of the Remote Reprogramming of wireless sensor networks using the code dissemination algorithm described in the previous section is presented in this section. We use the mathematical model designed by us and the parameters derived by us during our algorithm analysis to optimize the performance.
The point to be noted here is that we are building the reprogramming system from scratch. That is we are not building the system on top of some existing remote reprogramming framework like XNP which is available with TinyOS 1.x. Also, currently we support TinyOS 2.x with TelosB platform only.

5.1. Optimization Techniques

The following are some optimization techniques used in our implementation to avoid some typical problems faced in wireless sensor networks:

(1) We use the value of time period for which the base station should wait after it transmits TEO. To further optimize the performance, we use this value for further successive packet transfers as well. Also internal nodes are provided this value through the TEO. Thus, they too use it as an interval between two transmissions.
(2) Also, internal nodes use the values provided through TEO only for few initial transmissions. They can alter the timer dynamically depending on the replies provided by the subsequent nodes.

For example, let the initial value of the timer provided by TEO is X. However, the internal node may receive replies in :25 X time, and then it stays idle for :75 X time, without receiving any reply. In this situation it can alter the new timer value as,

$$X = 2\ 0{:}25\ X{:}$$

(2) We also realized during our experiments that we do not need to acknowledge the received TEO at all. As we forward the packets, they reach back to the previous sender (call it parent) as well. In order to make the parent treat it as a reply, we introduced the PARENT ID field in each packet.

Each node should check if P AREN T ID! = M Y ID to accept a packet. Else it should reject the packet.

5.2. Operations

The message types used to implement remote reprogramming using tree based algorithm for code dissemination are given in the following table, together with a brief description of each and some important commands are briefly described. These messages / commands are required to implement the algorithm that has been described in the previous sections. Also, some of them are required for the implementation of various optimization techniques employed in the present work or to recover from various types of failure.

Table 1. Message Types Used in the Implementation

Command	Value	Description
NRP EST TOPO	1	Command used to start network reprogramming.
NRP COD DWN	2	Download an SREC packet.
NRP NFN INC	3	Node failure notification-incomplete code download
NRP NFN PAR	4	Node failure notification-parent failed.
NRP NFN WOK	5	Node failure notification-just now woke up
NRP DWN FIN	6	Code download complete.
NRP STATUS REQ	7	Request download status.
NRP STATUS REPLY	8	Reply to status request.
NRP REBOOT	10	Reboot the mote.

(1) NRP EST TOPO: This is our topology establishment object. The other message parameters which are supposed to accompany this are discussed further. Message with this type is used by PC side interface to start the whole reprogramming process.
(2) NRP NFN INC: This is a ”Node Failure Notification” packet. This is transmitted by a node if it receives some packets of SREC code and then the ”Reprogram Status Timer” times out. The ”Reprogram Status Timer” is the timer used by a node to wait for the next code packet, after which it should execute failure action in order to promise reliability. The value of the timer is selected by the value which is provided to the node through NRP EST TOPO message. Also, optimization techniques which we discussed above can also be applied to get the best runtime value for .
(3) NRP NFN WOK: When a node fails for some reason and wakes up, it sends this packet with currently available code version, in order to receive updated one.

The format of the messages used in the present work is discussed next. These are based on the interface provided by the TinyOS 2.x. Effort is made to use these as optimally as possible in order to reduce message complexity.

5.3. Message Format

We use Active Message Packet interface which is provided by TinyOS 2.x. We exploit ”Payload” part of Packet structure through AMPacket interface in order to implement our algorithm. Also, since none of the interfaces or structures provides a mechanism to implement sequence number which is required for detecting missing packets. Hence we implement it too. Following is the message format, with actual values which were used during the implementation.

(1) Initiation message, used to start network reprogramming by PC side Java interface. It contains Parent ID to piggy-bag the acknowledgment as we discussed in the optimization techniques.

Table 2. Message Format 1

0	1-2	3-4	5-6	7-28
Commend				
NRP EST TOPO	Parent ID	Code Version	Code Length (in terms of No. of packets	0

(2) This message contains actual SREC code, which every node should extract and keep a copy for itself and forward, if the algorithm permits that node to do so. Buffering is required since the node should convert SREC code to the format which is acceptable to boot loader.

Table 3. Message Format 2

0	1-2	3-4	5-28
Command			
NRP_COD_DWN	Sequence No.	Code Version	Actual SREC Code

(3) Failure notification when incomplete code is received and timed out.

Table 4. Message Format 3

0	1-2	3-4	5-28
Command			
NRP_NFN_INC	Missing Sequence No.	Code Version	0

(4) Failure notification when parent fails. More specific version of previous message. Used for de-bugging purpose.

Table 5. Message Format 4

0	1-2	3-4	5-6	7-28
Commend				
NRP_NFN_PAR	Missing Sequence No.	Code Version	Parent ID	0

(5) This message is transmitted by a node when it wakes up. It does so in order to stay updated with latest code.

Table 6. Message Format 5

0	1-2	3-4	5-6	7-28
Commend				
NRP_NFN_WOK	0	Code Version	Parent ID	0

(6) This message is transmitted by PC side interface when it has got no more code packets to trans-mit. Same is forwarded by base station.

Table 7. Message Format 6

0	1-2	3-4	5-28
Command			
NRP_DWNS_FIN_	0	Code Version	0

(7) Request download status of current code. The replies can be, download completed, downloading, running with specified code etc.

Table 8. Message Format 7

0	1-2	3-4	5-28
Command			
NRP_STATUS_REQ_	0	Code Version	0

(8) Reboot the mote if SREC code for specified code version is completely acquired and placed on EEPROM in proper format, which can be very well understood by boot loader.

Table 9. Message Format 8

0	1-2	3-4	5-28
Command			
NRP REBOOT	0	Code Version	0

5.4. Implementation

We implement our algorithm from scratch, instead of building it on top of existing reprogramming modules like XNP or sharing some libraries with multihop support like Deluge. The implementa-tion comprises three different parts, viz. 1. PC Side Interface; 2. Base Station; and 3. Mote Side reprogramming module.

5.4.1. PC Side Interface

NRPInterface implements PC side interface, in order to communicate between mote and PC. It provides user, a command line interface with a specific set of commands, which enable user to interact with the network and reprogram it remotely.

Mote Interface Generator: While implementing PC side interface, which is written in Java, we face basic problem of compatibility of data structures. It is not easy to understand what a mote sends to the PC through UART. So we must be able to retrieve the information from raw data which is sent by mote. TinyOS has very few tools to retrieve this information.

The data types, data structures which are used by the application or supported by TinyOS, must be understood by Java. The interface must be able to read the raw bytes and parse them into a given packet format. The TinyOS toolchain makes this process easier by providing tools for au-tomatically generating message objects from packet descriptions. Rather than parse packet for-mats manually, we can use the MIG (Message Interface Generator) tool to build a Java, Python, or C interface for the message structure. Given a sequence of bytes, the MIG-generated code will automatically parse each of the fields in the packet. It provides a set of standard accessors and mutators for printing out received packets or generating new ones [Dunkels (2004)].

The MIG tool takes three basic arguments which are:

Programming language to generate code for (Java, Python or C)

The file in which to find the structure, which is used by mote application The name of the structure.

The tool also accepts standard gcc compiler options as well, such as -I for includes and -D for defines. The NRPInterface application, for example, uses MIG so that it can easily create and parse the packets over the serial port. A class is created by MIG corresponding to each structure which is used by mote application. Object of these classes are used by NRPInterface class to float specific commands through the network, as these objects contain fields which we specify in message format, which are understood by our mote application.

5.4.2. Base Station

This is the software which resides on the base station of a mote. Its function is to receive packets from the PC through UART interface and forward them to the network by broadcasting them over radio interface and vice-versa. However, since there can be thousands of motes which may forward to the base station, there might by problems at base station. Hence we have implemented proper buffering for transmission as well as receiving.

Interfaces in base station application:

(1) AMSend:

AMSend stands for Active Message Send interface. Since it is very common to have mul-tiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. The term ”AM type” refers to the field used for multiplexing. As Send provides basic address-free message sending interface, AMSend pro-vides the same with Active Message specifications. The main difference between AMSend and Send is that AMSend takes a destination AM address in its Send command.

(2) Receive: This is a basic message receiving interface. Since TinyOS is an event based sys-tem, this interface provides an event for receiving messages. The application is supposed to capture this event, which then gives received message pointer, pointer to payload part of the message and length of the message.

(3) Packet: Packet interface provides basic access methods for message t abstract data type. It provides commands for clearing the contents of a message, getting its payload length and getting a pointer to its payload area.

(4) AMPacket: Similar to Packet, AMPacket provides the basic AM accessors for the message t abstract data type. This interface provides commands for getting a node's AM address, an AM packet's destinatiōn and an AM packet's type. Commands are also provided for setting an AM packet's destination and type and checking whether the destination is the local node.

Components Used:

(1) ActiveMessageC: This component is used as Radio and wired with AMSend, Receive, Packet and AMpacket interfaces which are used as radio communication interfaces.

(2) SerialActiveMessageC: This component is used as Serial and wired with AMSend, Receive, Packet and AMpacket interfaces which are used as serial communication interfaces.

Other interfaces like Boot, SplitControl, Leds are also used which are not significant as far as implementation of our algorithm is concerned.

5.4.3. Mote Side reprogramming module

NRPLib interface in the codebase provides reprogramming libraries. The whole reprogramming process is implemented as a state machine. Also, appropriate timeouts and buffers are provided for optimized performance. Mote Side Reprogramming module consists of 2 main parts: a) Code dissemination and b) Storage and Reprogramming.

Code Dissemination: Synchronization is a major problem in event based platforms like TinyOS. Considering these failures, like the event can occur at any point of time and there is no control over the packet inter arrival time, we must store the packets as they arrive even before we process them. Buffering is one of the techniques by which this can be achieved. Lower layers of TinyOS provide poor buffering support. In order to tackle the issue, we have implemented a higher layer buffering system. Whenever a packet receive event occurs, it is captured by the application and is buffered for further processing. A circular queue is implemented for this purpose. This is the process buffer. After process buffer is appended with new packet, if radio send task is not going on, it is posted and radio status is set as busy. In this radio send task, the packet is checked on various grounds according to the algorithm. Checks are performed for duplicate packets, version validity, sender validity, sequence numbers etc.

If the received packets are found valid, then

They are forwarded over the radio and then added to EEPROM - write buffer, Write task is submitted.

Storage and reprogramming:

5.5. Implementation of Failure Model

In order to cater packet loss, we implemented sequence numbering of packets into the system. The lost packet is retrieved in two stages: 1) Normal Packet Recovery and 2) Delayed Packet Recovery.

5.5.1. Normal Packet Recovery

This packet recovery action is taken as soon as loss is detected. A piggy-bagged negative acknowledgment is sent, in order to retrieve the packet. If the neighboring nodes can immediately fulfill the request, they do so. The node keeps on sending this negative acknowledgment till the packet is retrieved, for every incoming packet. While doing so, it puts all the out of sequence packets into a buffer. Once this buffer is filled, it makes an entry of missing packet and writes all the buffered packets on EEPROM of the mote. While doing so, it leaves required space for that missing packet. Packet losses due to collisions are quickly recovered due to this technique. Most of the other algo-rithms have a dedicated query phase, which leads to unnecessary buffer loss and packet recovery is taken to base station or pc, which in turn reduces the efficiency of recovery system. Due to this throughput of the system goes down.

5.5.2. Delayed Packet Recovery

(1) This inherits the packet recovery process from other algorithms, with the only difference that it uses pull mechanism instead of push. That is after the image transfer completes, nodes scan its missing packet queue and then generates a NRP NFN INC which is specially designed for such recoveries. During this missing phase, all the packets are recovered and written on EEPROM. NRP REBOOT
(2) NRP DWN FIN

Please refer previous section for details of these commands.

6. Results

While developing the system, we used the hardware which was available to us. It consisted of 3 TelosB motes and a bunch of batteries. As we were developing the system, we were testing it against this hardware.
The most important and critical part in the implementation was multihop data transfer. The code image on PC must reach the motes which reside at a distance of more than or equal to 2 hops. As we have seen, we devised an algorithm for this. By designing specific packet formats, and the sequence in which they should be sent and received, we came up with a protocol, with which we could achieve our main goal. So while testing the dissemination part, we used the following test cases:

(1) Single Byte Test This was an elementary test case. If we can pass single byte over multichip, we can pass many similar packets like that one. So we first tried to establish the communication by sending a single byte over the network. If this fails, no case hence forth will work.
(2) Array size=max packet payload size. Now we try to send a fully stuffed packet. The default maximum payload length for tinyOS 1. X is 29 bytes. In case of improper timers, more is the probability of packet drops when we send maximum allowed data. Since for better throughput, we will opt for this extreme case only, testing for this case is important.
(3) Number of bytes=size of buffer. We must also test if the buffering support which we provide in our implementation is enough or not. In case of insufficient buffers and inappropriate timeouts, we might get higher packet losses. Thus, this test is a type of stress test that checks whether te buffering mechanism used in our implementation is adequate.
(4) File size=1kb Now we try to send bulk of data over the network. We test this for scalability of our firmware in terms of data.
(5) File Size=128kb (Maximum Program size.) We now test our system for boundary value of data size. We must be able to transfer 128 kb of data and save it on EEPROM.

With the above test cases; following was the test scenario: We were provided 3 Crossbow TelosB motes. They have USB connectivity. We made one as base station and then tested the mote side software with different placements of nodes. We consider the following scenarios:

(1) The two motes at a distance of single hop from base station. With this we tested how efficiently, the algorithm can filter out duplicate packets so that least of them are generated, and they die as soon as generated.

(2) One mote was at a distance of one hop and the other was kept at a distance of 2 hops. With this we checked multi-hop dissemination ability of the system.

As far as correctness of the algorithm is concerned, it was found that in first four cases, the data was transferred with 99% reliability. In final case, out of 20 tries, we got 2 failures, which brought down the reliability to 90%. The images residing on mote was transferred back to PC using a small Java interface to check correctness of data which was transferred. Out of successful transfers, the code image was found correct in all the cases. Moreover, in all the cases the failures were detected and the correct codes were acquired. The above results clearly demonstrated that our implementation was working as per the goals. In particular, the failure model adopted in our work was adequate to address the failures in real systems. However, if more motes had been available to us then we could have performed more tests, particularly those related to the scalability of the system.

7. Conclusion and Future Work

In this paper, we examined existing algorithms for multichip network reprogramming of wireless sensor networks and some flaws were observed with many. We hence developed an algorithm which rectifies some of the deficiencies that exist in some of the review algorithms like Deluge, MHOP. We then mathematically modeled the algorithm and derived some timer values, which were used while implementing the algorithms. This led to an efficient algorithm for multichip network reprogramming of wireless sensor networks. We then made an attempt to implement this algorithm on TinyOS plat-form. Since we were implementing the system from scratch, i.e. we were even implementing single hop network reprogramming system, instead of building it on top of existing reprogramming libraries like XNP. Due to the large scope of the implementation part, we set a finite goal of implementing part of the system and we succeeded. The following modules of the system were implemented:

(1) PC Side Interface in Java.
(2) Base station software.
(3) Reprogramming module, which successfully and reliably transfers the code image over multihop network.

The work progress slowed down here due to unavailability of proper documentation about Boot Loader of TinyOS which is required for implementation of further part. Hence, as soon as the proper documentation of TinyOS boot loader is available, further part of the algorithm can be implemented. This part contains:

(1) Transforming the received image to boot loader understandable format.
(2) Configuration of boot loader.
(3) Reboot the mote, in order to run new program.

We also consider the remaining part of our problem statement as a direction for future work, in order to make this system to a better system for remote reprogramming than the existing one viz. Deluge.
After the system is fully implemented, a comparative analysis of existing systems versus this one is needed. This would lead to discovering possible logical flaws in the algorithm, and hence help us to improve it on various fronts. The system could also have an extensive platform support, which currently is limited to Telosb motes only.

8. References

[1] Deng, J. Dang, R. Mishra,S. (2006). Efficiently Authenticating Code Images in Dynamically Reprogrammed Wireless Sensor Networks, Proceedings of the 4th annual IEEE international conference on Pervasive omputing and Communications Workshops, 2006.

[2] Demers, A. et al.(1987). Epidemic algorithms for replicated database maintenance. In Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, pages 1-12. ACM Press, 1987.
[3] Dunkels, A. Gronvall, B. and Voigt,T. (2004) Contiki - A lightweight and flexible operating system for tiny networked sensors. In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, November 2004.
[4] Dunkels, A. , et al. (2006). Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks, Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, November 2006.
[5] Hadim, S. , Mohamed, N. (2006). Middleware Challenges and Approaches for Wireless Sensor Networks. IEEE Distributed Systems Online. 7 (3): 1. doi:10.1109/MDSO.2006.19. art. no. 0603-o3001.
[6] Haenselmann, T. (2006). Sensornetworks. GFDL Wireless Sensor Network textbook. Retrieved on 2006-08-29. Hui, J. W., David Culler, D. (2004). The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale, Proceedings of the 2nd international conference on Embedded networked sensor systems, Baltimore, MD, USA, 2004
[7] Jan, C. et al.(2008). Typhoon: A Reliable Data Dissemination Protocol for Wireless Sensor Networks, 5th European Conference on Wireless Sensor Networks, EWSN 2008.
[8] Jeong, J., Kim, S. and Broad, A. (2003). Network Reprogramming, TinyOS documentation, Aug. 2003
[9] Levis, P. et al. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. Technical report, University of California at Berkeley, 2004.
[10] Lewis, F.L. Wireless Sensor Networks, in Smart Environments: Technologies, Protocols, and Application , ed.D.J. Cook and S.K. Das, John Wiley, New York, 2004.
[11] Mainwaring, A. et al.(2002) . Wireless Sensor Networks for Habitat Monitoring. WSNA'02, September 28, 2002, Atlanta, Georgia, USA.Ni, S.Y. ,et al. (1999). The broadcast storm problem in a mobile ad hoc network. In Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking, ACM Press, 1999.
[12] Rabiner, W., Kulik, J. and Balakrishnan, H. (1999). Adaptive Protocols for Information Dissemination in Wire-less Sensor Networks, Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking, Seattle, Washington, United State, 1999.
[13] Reijers, N., Langendoen, K.(2003) Efficient Code Distribution in Wireless Sensor Networks, Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, San Diego, CA, USA, 2003.
[14] Romer, K., Mattern, F.(2004). The Design Space of Wireless Sensor Network. IEEE Wireless Communications, 11 (6): 54-61.
[15] Stathopoulos, T., Heidemann, J., Estrin, D. (2003). A Remote Code Update Mechanism for Wireless Sensor Networks, Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November, 2003. University of California, Berkeley. Tinyos. http://www.tinyos.net/.