REMOTE PROGRAMMING FOR HETEROGENEOUS SENSOR NETWORKS

by

VISHAL BHATIA

BCA(Hons), Devi Ahilya Vishwavidyalaya, India, 2006

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department Of Computing And Information Science
College Of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr Gurdip Singh

Abstract

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. It uses the 3-way handshake protocol consisting of 3 types of messages: advertise, request and data. The protocol is very useful but is restricted to homogeneous networks wherein all nodes must be programmed with the same code. This project is an attempt to modify the existing protocol to work for heterogeneous networks where different motes function differently and have to be programmed differently.

The project was developed using Java and nesC (a dialect of C) which supports component based programming. The nodes run an operating system called tinyOS which is specifically designed for sensor networks. The system was tested on a network of micaZ and TelosB motes.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dr Gurdip Singh for his kind guidance and support throughout the project. He always guided me with very efficient and productive ideas whenever I got stuck.

I would also like to thank Dr Daniel Andresen and Dr Mitchell Nielsen for being part of my committee.

CHAPTER 1 - Introduction

A wireless sensor network (WSN) is a network of widely distributed sensors which can be of different types. WSNs are increasingly becoming focused area of research because of their applicability in many different fields. A wireless sensor network can be used for common purposes such as monitoring temperature, pressure, or humidity of a particular area, be it a farm, a mill or a laboratory. For example, we have worked with researchers to use sensor networks to monitor temperatures in a flour mill in order to kill the insects and pesticides and not affect the quality of the food materials. Since it serves so many useful purposes, it has become important to come up with better way to maintain and manage sensor applications more efficiently.

This project aims at modifying an existing protocol for remote re-programming of motes in a sensor network by injecting images into motes or the sensors wirelessly thus resulting in great saving of time and resources. We will discuss this in a bit more detail in the following.

Suppose we have a large network of motes or sensors fixed in a field or a farm which is assigned some task such as sensing temperature for various portions of the field and sending it back to a mainframe system collecting data from all the motes and taking actions accordingly. Now, if we want all the motes to sense pressure instead of temperature after a certain point of time, then one doesn't expect to individually remove all the motes from the network and individually connect them to the main system to program them with a different application and fix them again at their respective positions. Keeping this problem in mind, a very affective protocol called Deluge was designed [http://www.cs.berkely.edu/~jwhui/deluge/documentation.html] and implemented that had the capability of wirelessly re programming the motes over a multi hop network without having to shift them from their current positions. We will now describe this protocol in detail.

1.1 –The Deluge protocol

Deluge is a data dissemination protocol for re-programming the motes wirelessly over a multi hop network. The basic functionality of Deluge relies on a push-pull based algorithm where every mote periodically spreads a message over the network. The motes advertise themselves periodically by giving information about the images they have with them. When a node receives this advertising message, it sends back its profile to the mote from which it receives the request. This mote, in turn checks whether it has that specific image or not. If it doesn't, then it sends a request message back to the sender to send those images and hence the whole network eventually gets any image that was with a single node in the beginning. In this way, the whole network gets programmed with the same image. This is the reason for Deluge being said to operate on a homogeneous network.

The protocol consists of 3 main types of messages:

A – DelugeAdvMsg.

When the motes start interacting, the first action they perform is to advertise their profile to all other motes. This message is the DelugeAdvMsg type which has the following structure:

```
typedef struct DelugeAdvMsg {
  uint16_t      sourceAddr;
  uint8_t       version;
  uint8_t       type;
  DelugeNodeDesc nodeDesc;
  DelugeImgDesc  imgDesc;
  uint8_t       numImages;
  uint8_t       reserved;
}
```

This message basically consists of the information about the image number, version number of the image, image description, and the type of image currently present on the mote and some meta-data along with it. When another mote receives this message, it sends an Adv message containing its profile information back to the mote it received the Adv msg from.

B – When the mote receives an Adv msg back from another mote, it compares its own profile with the profile of the sender and checks if it has an obsolete version of the image. If it does, then it sends back what is called a DelugeReqMsg message to the sender that has the following structure:

```
typedef struct DelugeReqMsg {
  uint16_t  dest;
  uint16_t  sourceAddr;
  imgvnum_t vNum;
  imgnum_t  imgNum;
  pgnum_t   pgNum;
  uint8_t   requestedPkts[DELUGE_PKT_BITVEC_SIZE];
}
```

This message contains the information of the requested packets of a particular image of a particular version number that the former mote is lacking.

C – After a mote receives a particular request from another mote, it sends the data requested by the requester in form of a DelgueDataMsg message which has the following structure:

```
typedef struct DelugeDataMsg {
  imgvnum_t vNum;
  imgnum_t  imgNum;
  pgnum_t   pgNum;
  uint8_t   pktNum;
  uint8_t   data[DELUGE_PKT_PAYLOAD_SIZE];
}
```

This message contains all the data requested by the requester and is sent only to the mote that requested it and is not broadcasted.

The following figure shows the interactions between motes that take place in Deluge:

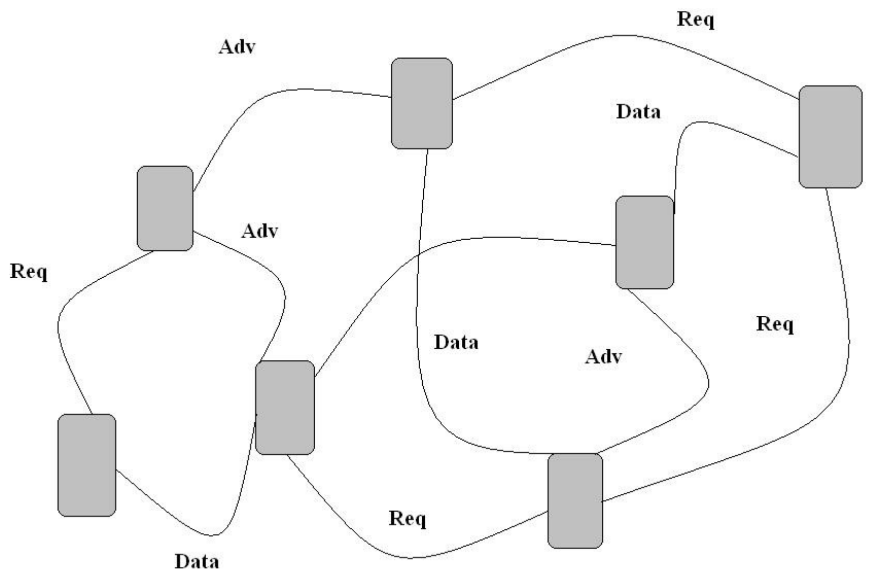


Figure 1.1 Interactions in Deluge

This whole process is repeated based on a timer which fires periodically. Eventually, there is no mote in the whole network that has the obsolete version of any image that was injected through the PC.

1.2 – The Deluge JavaToolchain

In order to inject messages into the network, Deluge provides a Java tool chain which is responsible for sending messages from the PC to the motes. These messages are forwarded by the Base Station. The base station is a mote fixed on MIB510 board running the TOSBase application that forwards messages from the UART to radio and from the radio to UART thus acting as a bridge between motes and the PC. The following figure shows MIB510 board with micaz mote that forms a part of a base station.

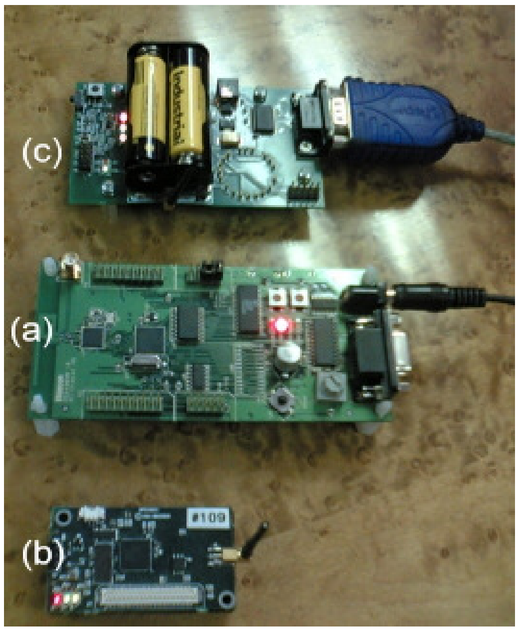


Figure 1.2 - MIB510 board with micaz mote

The toolchain consists of a set of various commands which include Inject, reboot, erase, reset, ping. They are explained in detail below:

1.2.1 - Ping:

The Ping command is given to check the current profile information on a mote. It retrieves the images stored on a particular mote at a point of time. It gives the information about the type of image, image number, page number, etc for each of the 4 slots present on the external flash of a mote. The format of the ping command is:

```
java net.tinyos.tools.Deluge --ping
```

The only condition for executing a ping command is that the mote should be in direct contact with the PC through a base station.

1.2.2 - Inject:

The inject command is used to inject a particular compiled image into the network. Before injecting the image, it is necessary to make or compile the application using the "make" command which compiles the image in form of binaries and stores it in the build/platform folder. The platform depends on the type of mote being used, in our case MicaZ or TelosB. The reason being that these images that need to be injected into the network are pretty large and cannot be injected directly as they are, that’s why they are stored in form of binaries and are further divided into pages which further break them into packets. The format of inject command is:

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.

1.2.3 - reset:

The reset command is used to reset an image in a particular slot. The format for the reset command is:

java net.tinyos.tools.Deluge –reset imgnum = <ingnum> where imgnum is the image number we want to reset.

1.2.4 - erase:

The erase command is given to erase a particular image on a mote so that it is ready to store another image in the specified slot. The format of the erase command is:

java net.tinyos.tools.Deluge –erase –imgnum=<imgnum>

1.2.5 - Dump:

The Dump command is used to extract the exact information of the image on a specific slot on a mote. It actually gives the tos_image.xml file for that particular application.

java net.tinyos.tools.Deluge –dump –imgnum=<imgnum> --outfile=<xml>

All these commands, when invoked, use their respective Java classes to send a message in one of the respective formats as discussed earlier to the base station where it is injected into the network, hence reprogramming the motes with the desired application.

1.3 -The need for a change

As we saw in the earlier section, Deluge is a pretty promising and reliable protocol that has been widely used for wirelessly re-programming the motes. However Deluge has its own merits and demerits that are discussed below.

Deluge provides a reliable way to inject images into the network based on the fact that motes are able to interact with each other and hence can eventually have the same image running on them even if the image is not directly injected to the mote (in other words, the mote is not in direct contact with the base station). The following figure shows the concept of Deluge:

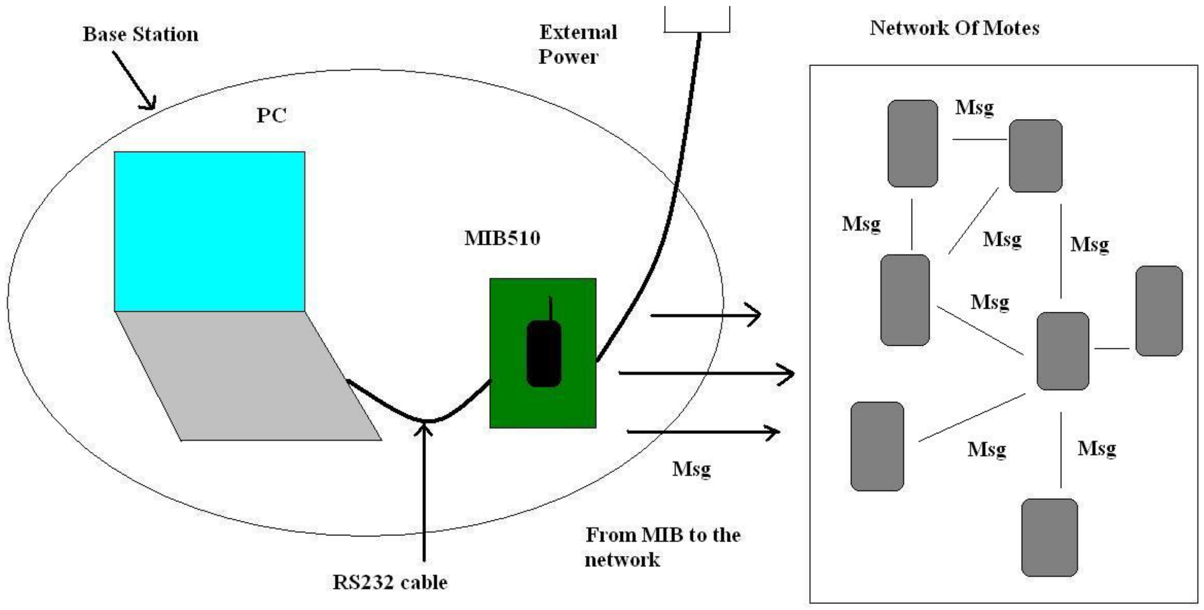


Figure 1.3 Message spreading in Deluge

The problem arises, however, when we wish different motes to run different images or applications simultaneously being a part of the network as well. This cannot be possible in the original Deluge model as the image is delivered to all motes in the network. This leads to a lot of wasted resource since all the 4 slots in each of the motes would be having the same images which means that at one time, there cannot be more than 4 images simultaneously existing in the network. This is a problem since we are not making complete utilization of the available memory. This project aims at designing and implementing a protocol where different images can be injected into the network for a particular mote which is also a part of the mesh of motes without affecting the applications running on the other motes that are not interested in getting the image. This paves way for the possibility of having a large heterogeneous network where different motes are used for different purposes of recording temperature, pressure, density etc as per the need.

The rest of the document is framed as follows.

Section 2.1 describes the initial approach taken to get Deluge working for heterogeneous networks, its drawback and the need to look for a better mechanism. Chapter 2 describes the approaches taken to achieve heterogeneity using Deluge and the changes made in the original protocol. Chapter 3 provides the implementation details and how to use the modified Deluge program to reprogram motes (including code snippets and commands needed). Chapter 4 provides details about testing the application. Chapter 5 concludes the document.

CHAPTER 2 - Design Considerations

This chapter describes the two design approaches taken to solve the problem and finalizing one of them that proved to be most efficient.

The very first step was to identify a mechanism to stop the motes from interacting with each other (using advertise messages).If they are allowed to interact and exchange data, they will eventually be having the same application. This was achieved by stopping the mote to send the Adv message on timer.fired event. In this way, the motes are just waiting for the messages from the base station and no inter-mote interaction takes place. After this step, the following 2 approaches were considered.

2.1 The Initial Approach

The first approach taken was to include an extra field called destAddr in the message structure of DelugeAdvMsg and to respond to messages according to the contents of the destAddr field. So, if a message is injected for a particular node id, then only that node will accept the message and all other nodes will forward the message, or broadcast the message until it reaches the destination. At the first go, it looked like a pretty simple and a straight forward approach to attain heterogeneity, but if we go into it in more detail, we find some interesting issues.

1 – There are 3 main types of messages that are a part of Deluge; these are DelugeAdvMsg, DelugeReqMsg and DelugeDataMsg. If we look at the concept of including a destAddr field just for DelugeAdvMsg, then it won't solve our purpose since all three kinds of messages must be parsed properly for correct injection based on destination address. The problem here is that we cannot include the same field in the other 2 message structures because they are already designed to have the maximum packet length that TinyOs packet can take (which is by default 28). Any change in the structure might affect the transmission of packets for Deluge. Hence, this approach doesn't seem to be that efficient and robust. One can think of using the same field from message type DelugeAdvMsg and access it in different files involved in the protocol. Although this doesn't seem feasible since the messages DelugeReqMsg and DelugeDataMsg are being accessed from a file

which itself acts as a component for Deluge Protocol. So it is possible to access the methods and variables of this file in DelugeM.nc but not vice versa.

2 - Another approach would be to increase the default size of Tinyos message while installing Tinyos (which is possible as per the Tinyos forum messages.) and before issuing the make command so that we can add the destAddr field in DelugeAdvReq as well as the DelugeAdvData message. This would well solve our purpose and would achieve what is desired, but let us take a look at it in a bit more detail.

Now, as the message enters the network, it goes through all the motes that can listen to it since it is a broadcast message and all the motes forward or broadcast it so that it can eventually reach the base station. The problem that is unseen here is the network traffic. As can be seen from Figure 2.1, the work of forwarding can be done by just id 1 or 2 alone and the rest of the motes don't even need to participate in the forwarding process.

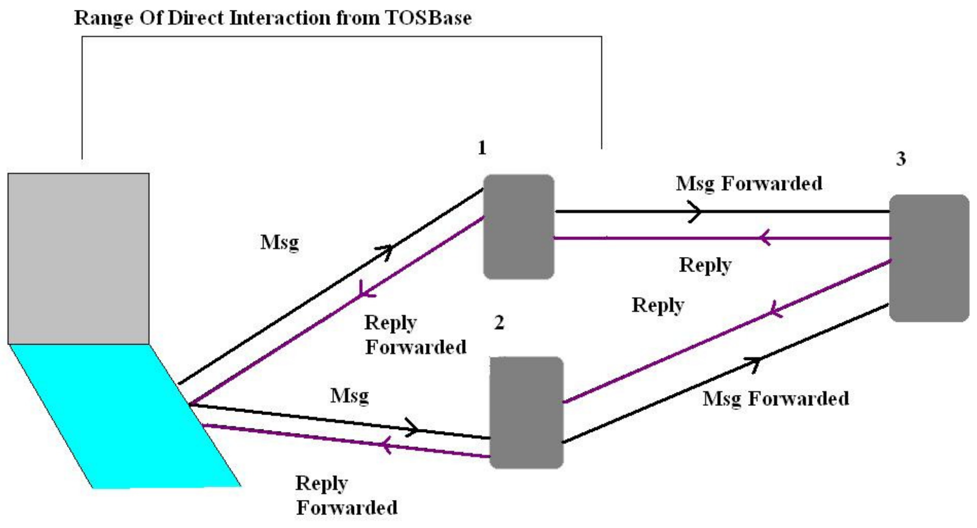


Figure 2.1 Initial approach taken to achieve heterogeneity

All motes taking part increases the network traffic manifolds as is pretty obvious from the above figure and there are great chances of the actual message being lost or not received because of a loop of messages that are formed in the network. This forces one think that if it is possible to come up with a way where we can assure that all the motes in the network are not taking part in forwarding the messages and only those motes that are on the shortest path from the base station to the destination participate and rest of the motes ignore the messages they receive. This approach reduces the network traffic manifolds as compared to the original approach and will be more robust. The approach being taken finally is explained in the next section.

2.2 The second approach

To ensure that only the motes interested should be the participating, we need to first come up with a way to set up a fixed path from the source to the destination so that messages travel only through that path. Also, it would be best to have that path as the shortest path so that the message can reach the destination in the minimum number of hops. The current approach is based onthis. Before even starting Deluge, a set path message is sent through the base station to the network of motes. The message contains a 'to' field which contains the destination address to which we want to set the path. The message is broadcasted so it traverses through the whole network. Each mote that receives the message checks whether the message is from the base station (the reason will be explained later), if yes, then it checks whether the 'to' field matches it's TOS_LOCAL_ADDRESS or not. If yes, then it knows that it is the destination and sends back a message to the base station. However, if it is not the destination, then it just forwards the message further so that it reaches the destination eventually. However, before sending the message further, it does an important thing: it stores the hop-count of the message it receives and also the 'id' address, i.e., the address from which it received the message increases the hopcount by one and attaches its own id to the 'id' field and sends the message further so that it can reach the destination. The purpose of doing this is explained further.

Now, when the message has reached the destination, the destination node also needs to respond to the base station saying that it has received the message. Otherwise the PC will have no idea as to whether the path has been set or not. Before sending the message, the destination mote waits for a certain period of time so that it receives all the messages it can and decide which one is coming from the nearest source and sets the hopcount accordingly. The hopcount comparison action takes place at every mote since each mote can listen to all the broadcasted messages by other motes and hence need to decide which ones are useful messages and which are not.

When the destination has received all the messages, it sends back a message and sets another global variable 'interested' to 2. However, this time, it does not broadcast the message. Rather it sends the message only to the mote from where it received the message that had the least hopcount. This mote receives the message and sets the ‘interested’ field to 1 and further sends the message to the mote from where it received the message with the least hopcount and so on till it finally reaches the PC. We now have established a path where only the destination mote has the variable ‘**interested**’ set to 2 and the other forwarders (motes involved in the path) have it set to1 and the rest of the motes have it set to 0. Now when we send any Deluge message to the network of motes, the messages can be parsed on the basis of the variable and the ones that are not in the path will simply ignore the messages.

The following figure shows the concept more clearly

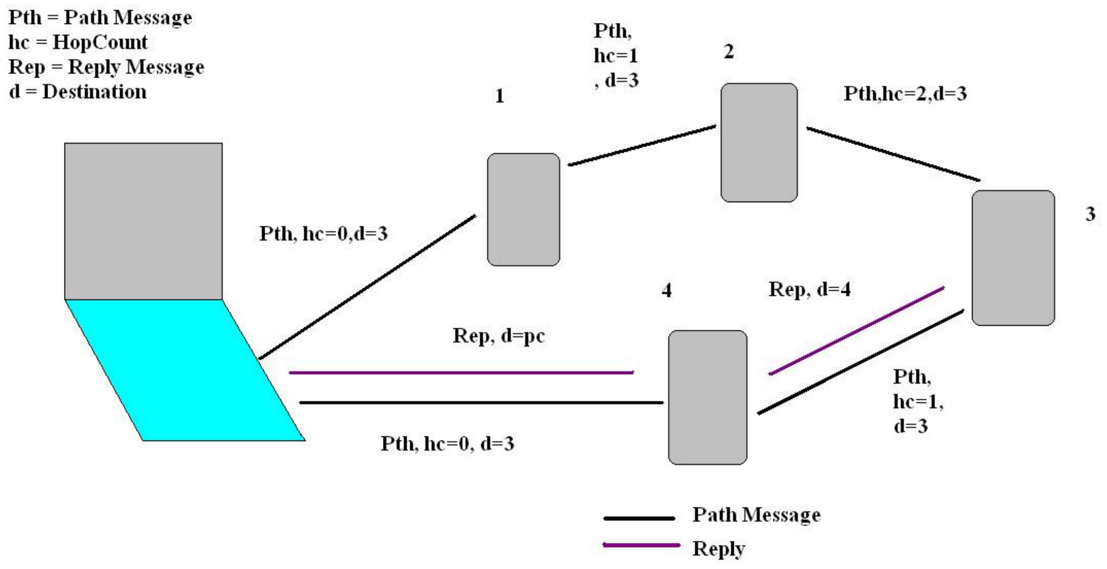


Figure 2.2 Correct approach to achieve heterogeneity

2.3 Changes made to Deluge

First of all, an extra message structure was added to the deluge messages file called the DelugePathMsg having the following structure:

```
typedef struct DelugePathMsg {
  uint8_t  from;
  uint8_t  hopcount;
  uint8_t  id;
  uint8_t  dest;
}
```
where:
  ‘*from’* contains the address of the source of the message.
  ‘*hopcount’* contains the hopcount of the message.
  ‘*id’* conatins the address of the source from where the message is coming.
  ‘*dest’* contains the destination address where the PC wants to send the message. The change is made in:
tinyos-1.x/tos/lib/Deluge/DelugeMsgs.h

One might think that “from” and “id” fields are essentially the same, which is true apart from the fact that “from” remains constant between the times it goes from the source to the destination, and changes at the destination and then again remains constant till it reaches the source. It is basically used by the mtoes to decide on what action to perform based on whether the message is from the base station or a mote. The “id” field is changed at every hop.
Now as the message classes are being generated by MIG, we need to make changes in the make file in
tinyos-1.x/tools/java/net/tinyos/deluge/

We added the following lines to the make file:
```
DelugePathMsg.java:
        $(MIG) -java-classname=$(PACKAGE).DelugePathMsg
-target=$(DELUGE_PLATFORM) $(INCLUDES) $(DELUGE_LIB)/DelugeMsgs.h
DelugePathMsg -o $@
```

Also, we need to add DelugePathMsgs.java to the variable MSGS.

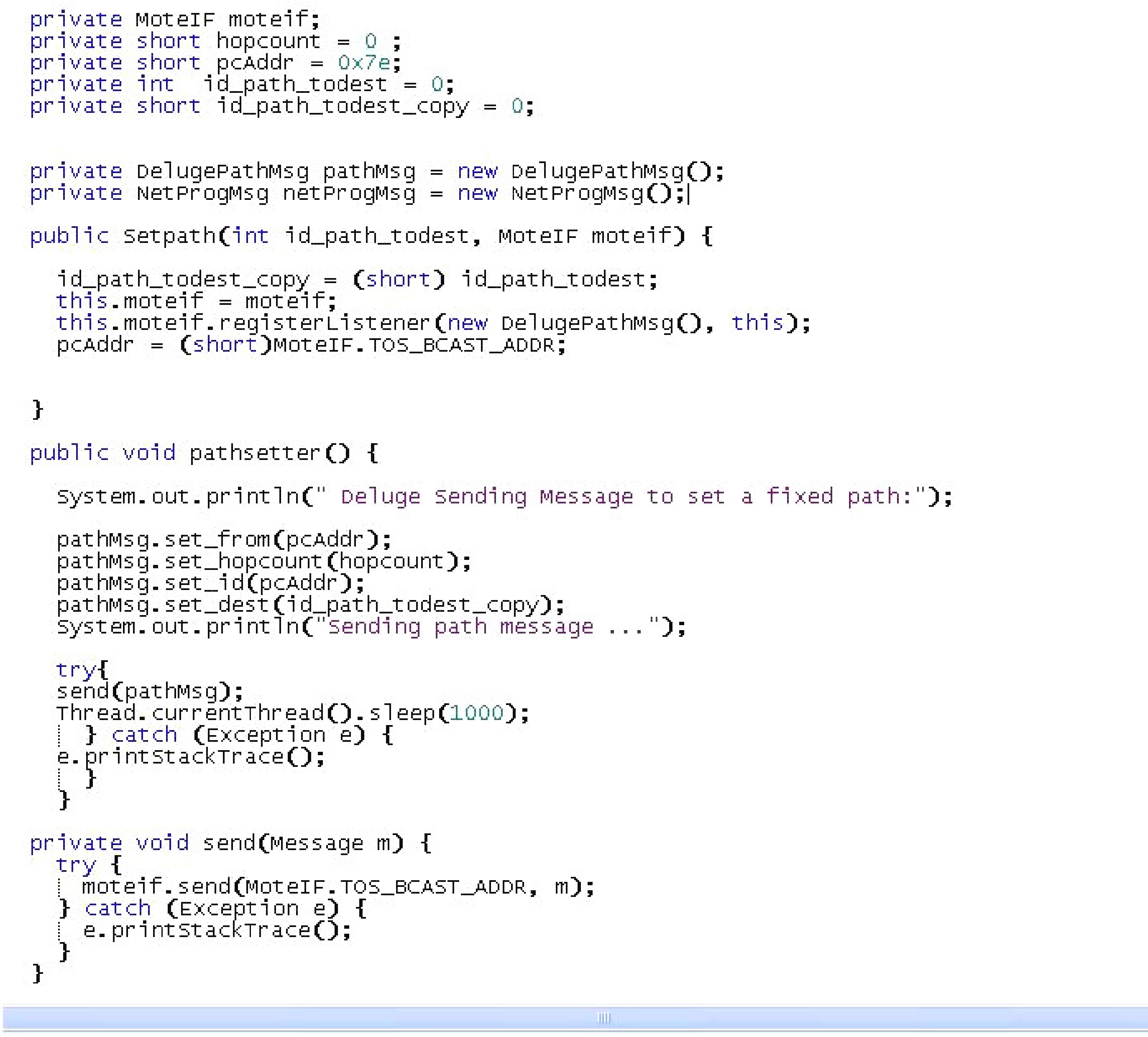
Next, the interaction of messages between the motes and the PC takes place on the basis of AM(Active Message) value of the message which is provided as an argument to the message. We need to add an AM type for DelugePathMsg so that it can interact with the motes. We provide AM value of 165 to the path message since AM values up till 164 are already being assigned to different message structures.
So, we need to add the following line to DelugeMsgs.h as an enum
AM_DELUGEPATHMSG = 165.

One can now go ahead and issue the “make” command in
tinyos-1.x/tools/java/net/tinyos/deluge.

Doing so will generate the required classes used by DelugePathMsg.
NOTE- THESE ARE AUTO GENERATED FILES BY THE MIG, SO DO NOT CHANGE OR TRY TO MODIFY IT.

Second, a Java program was written to send the setpath message from the base station to the motes. This Java file has to be a part of DelugeJava toolchain since it is based on this message and we need to set and access the global variables in the Deluge files as we discussed earlier.

The program simply sets the path message and implements a MessageListener class so that it can as well listen to the messages coming back from the motes.

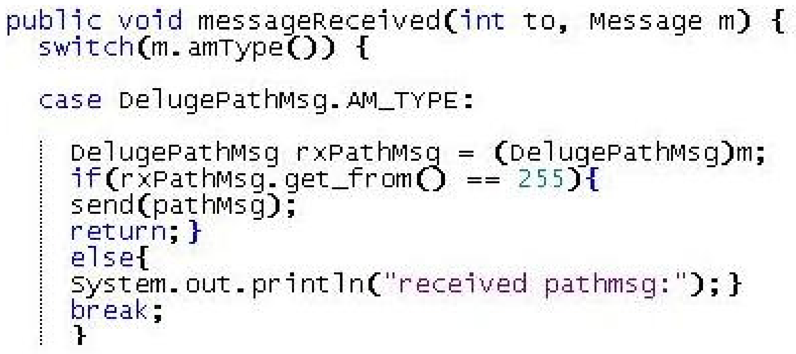


```
private MoteIF moteif;
private short hopcount = 0 ;
private short pcAddr = 0x7e;
private int  id_path_todest = 0;
private short id_path_todest_copy = 0;


private DelugePathMsg pathMsg = new DelugePathMsg();
private NetProgMsg netProgMsg = new NetProgMsg();

public Setpath(int id_path_todest, MoteIF moteif) {

  id_path_todest_copy = (short) id_path_todest;
  this.moteif = moteif;
  this.moteif.registerListener(new DelugePathMsg(), this);
  pcAddr = (short)MoteIF.TOS_BCAST_ADDR;


}

public void pathsetter() {

  System.out.println(" Deluge Sending Message to set a fixed path:");

  pathMsg.set_from(pcAddr);
  pathMsg.set_hopcount(hopcount);
  pathMsg.set_id(pcAddr);
  pathMsg.set_dest(id_path_todest_copy);
  System.out.println("Sending path message ...");

  try{
  send(pathMsg);
  Thread.currentThread().sleep(1000);
    } catch (Exception e) {
  e.printStackTrace();
    }
  }

private void send(Message m) {
  try {
    moteif.send(MoteIF.TOS_BCAST_ADDR, m);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

Figure 2.3 Code snippet for sending path message

The send method uses the standard moteif.send() method to send a message m to an address a as shown in the above figure.It has the following format moteif.send(a,m)

While setting the path, we will receive many messages other than the one we are expecting (the one sent by the destination) that we need to ignore since these are the messages that other motes are broadcasted while forwarding. Since these are broadcasted messages, the base station will also receive it, and forward it the UART but the program needs to ignore those messages.



```
public void messageReceived(int to, Message m) {
  switch(m.amType()) {

  case DelugePathMsg.AM_TYPE:

    DelugePathMsg rxPathMsg = (DelugePathMsg)m;
    if(rxPathMsg.get_from() == 255){
    send(pathMsg);
    return; }
    else{
    System.out.println("received pathmsg:"); }
    break;
    }
```

Figure 2.4 - Code Snippet for receiving path messages

If we do not receive a message back from the destination, then this implies that the path has not yet been set and we need to send the set path message again till we get this message on the PC.

The next change was made in the Deluge Protocol which is responsible for receiving the messages and parsing them and taking actions accordingly.

Before making any changes to the files DelugeM.n and DelugePageTransferM.nc which form the original Deluge protocol, we need to specify to Deluge that we would be using a receive event for DelugePathMsg which can be done by wiring it in the DelugePageTransferC.nc file as follows:

DelugePageTransferM.ReceivePathMsg = ReceivePathMsg;

We also need to mention in the DelugeC.nc file that we would be using AM 165 for DelugePathMsg so that it is consistent on both sides. This can be done by adding the following to DelugeC.nc:

  PageTransfer.ReceivePathMsg -> Comm.ReceiveMsg[AM_DELUGEPATHMSG]

The Comm interface is the one that actually takes part in the transferring messages from a mote to the base station.. Similar work has to be done for the SendMsg interface for the PathMsg message in the same files as follows:

DelugePagTransferC.nc:

  DelugePageTransferM.SendPathMsg = SendPathMsg;

DelugeC.nc

 PageTransfer.SendPathMsg -> Comm.SendMsg[AM_DELUGEPATHMSG];

Now we can use the send interface and the receive event in our files.

Coming back to the receive event:The receive event basically receives all the path messages it listens to.

TinyOS Messages have a fixed structure that is transmitted over the Radio which is as follows:

  uint16_t addr;

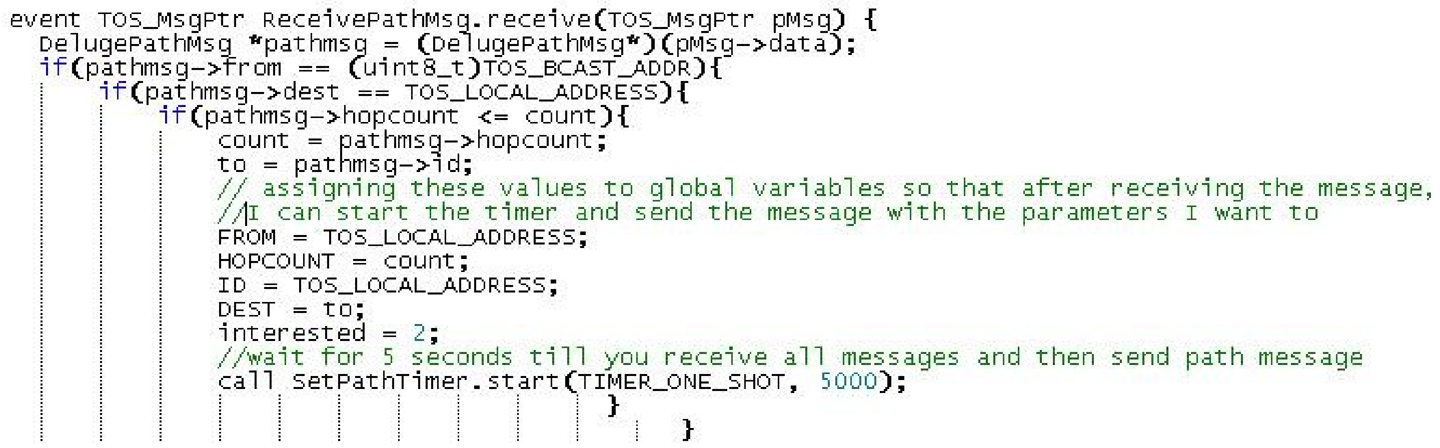
```
  uint8_t type;
  uint8_t group;
  uint8_t length;
  int8_t data[TOSH_DATA_LENGTH];     /* data that is sent through PC or through motes as
the case may be  */
  uint16_t crc;
```

The int8_t data[] is the actual part that is of interest to us since that contains the actual data that is being set by the PC.
So whenever we receive a message, the first step is to extract the data part from the msg and cast it to the proper message type. It is done as follows:

```
DelugePathMsg *pathmsg = (DelugePathMsg*)(pMsg->data);
```

Now we can access the separate fields of the message.

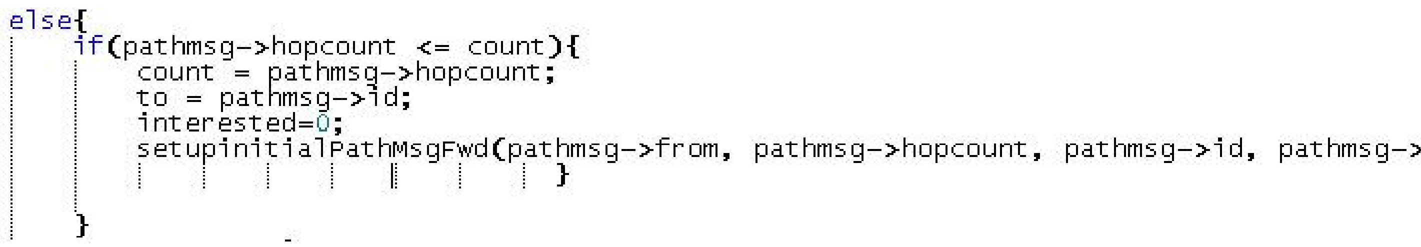
Whenever motes receive a path message, there can be 2 possibilities.
A – The message is from the base station:
If the message is from the base station and is intended for the mote receiving it, then it accepts the message, and checks whether the hopcount of the message received is lower as compared to any message that was received earlier. If yes then it starts a timer. The following snippet shows the above part



```
event TOS_MsgPtr ReceivePathMsg.receive(TOS_MsgPtr pMsg) {
  DelugePathMsg *pathmsg = (DelugePathMsg*)(pMsg->data);
  if(pathmsg->from == (uint8_t)TOS_BCAST_ADDR){
      if(pathmsg->dest == TOS_LOCAL_ADDRESS){
          if(pathmsg->hopcount <= count){
              count = pathmsg->hopcount;
              to = pathmsg->id;
              // assigning these values to global variables so that after receiving the message,
              //I can start the timer and send the message with the parameters I want to
              FROM = TOS_LOCAL_ADDRESS;
              HOPCOUNT = count;
              ID = TOS_LOCAL_ADDRESS;
              DEST = to;
              interested = 2;
              //wait for 5 seconds till you receive all messages and then send path message
              call SetPathTimer.start(TIMER_ONE_SHOT, 5000);
                                                  }
                                                        }
```

Figure 2.5 - receive event for path messages for the destination mote

If the message is from the base station but is not intended for the mote receiving it, then it just forwards it or broadcasts it further after setting the hopcount and to fields. The following snippet shows the above:



```
else{
    if(pathmsg->hopcount <= count){
        count = pathmsg->hopcount;
        to = pathmsg->id;
        interested=0;
        setupinitialPathMsgFwd(pathmsg->from, pathmsg->hopcount, pathmsg->id, pathmsg->
                                                }

    }
```

Figure 2.6 - receive event showing forwarding part for non destination motes

B – The message is from another mote (while coming back from the destination mote)

If the message is coming from another mote and the mote is receiving it, that implies that the mote will be a part of the path through which rest of the messages will be traveling, so it sets its “interested” variable to 1 and passes on the message to the id stores in its “to” field. The following snippet shows the above:

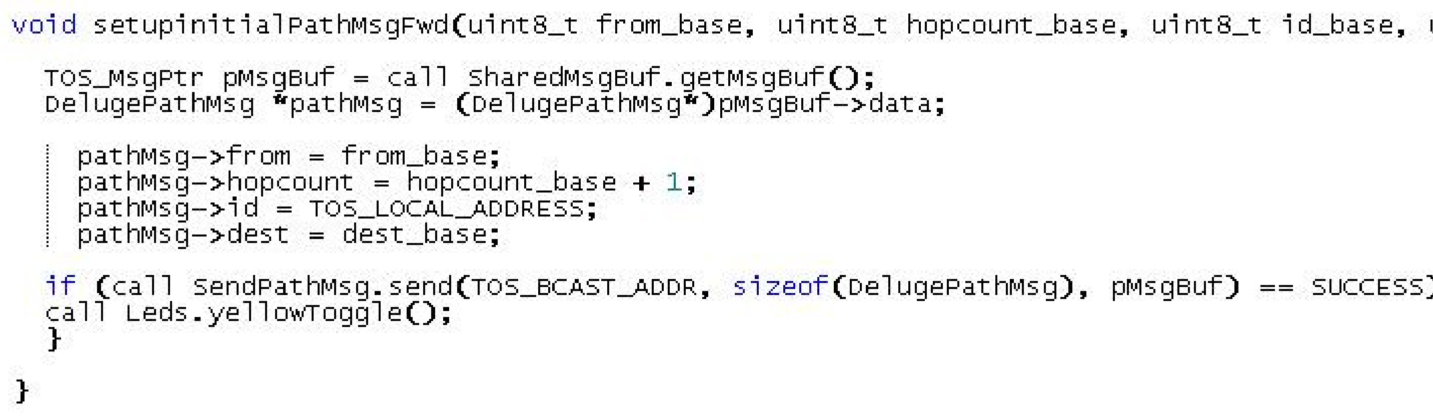
interested = 1;
setupfurtherPathMsg(to);

Next step is to consider message forwarding within the motes:

There are 2 types of messages:

A – One that the mote forwards (broadcasts)

When the mote forwards a message, it just sets the “hopcount” and the “to” fields and then makes a new message with its own id and increasing the hopcount by 1. The following snippet shows the following:

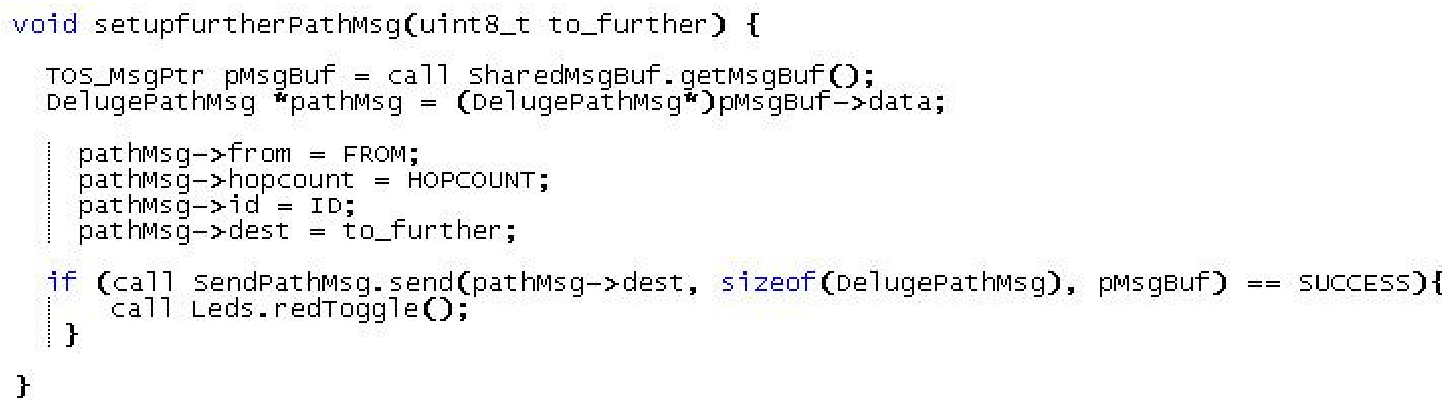


```
void setupinitialPathMsgFwd(uint8_t from_base, uint8_t hopcount_base, uint8_t id_base, 

   TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
   DelugePathMsg *pathMsg = (DelugePathMsg*)pMsgBuf->data;

     pathMsg->from = from_base;
     pathMsg->hopcount = hopcount_base + 1;
     pathMsg->id = TOS_LOCAL_ADDRESS;
     pathMsg->dest = dest_base;

   if (call SendPathMsg.send(TOS_BCAST_ADDR, sizeof(DelugePathMsg), pMsgBuf) == SUCCESS)
   call Leds.yellowToggle();
   }

}
```

Figure 2.7 Forwarding(broadcasting) part of the protocol

B – One that the mote forwards (not broadcasts)

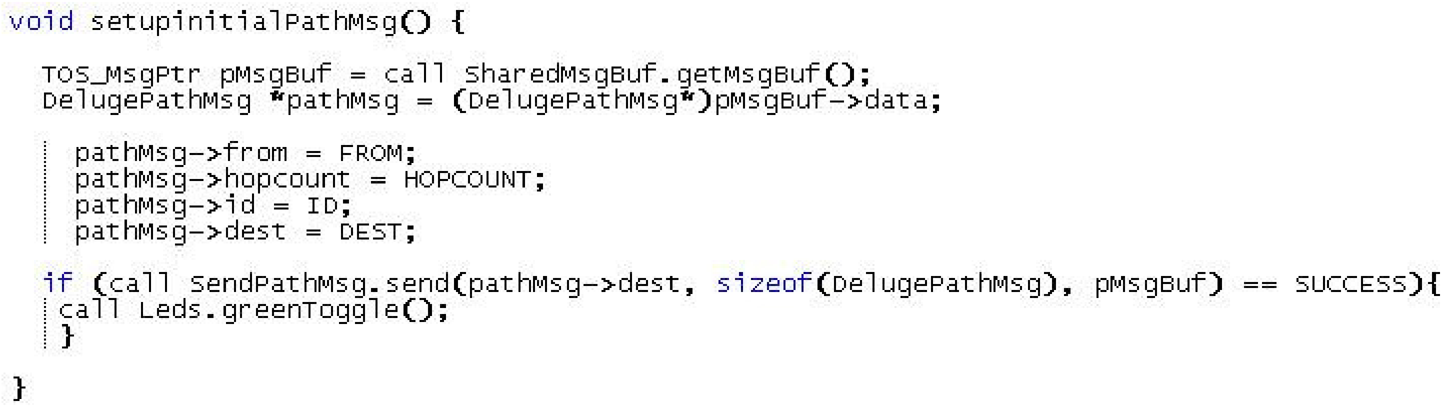
This is the message received only by the motes that will be a part of the path between source and destination. The following snippet shows the above.



```
void setupfurtherPathMsg(uint8_t to_further) {

  TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
  DelugePathMsg *pathMsg = (DelugePathMsg*)pMsgBuf->data;

    pathMsg->from = FROM;
    pathMsg->hopcount = HOPCOUNT;
    pathMsg->id = ID;
    pathMsg->dest = to_further;

  if (call SendPathMsg.send(pathMsg->dest, sizeof(DelugePathMsg), pMsgBuf) == SUCCESS){
       call Leds.redToggle();
   }

}
```

Figure 2.8 Forwarding (not broadcasting) part of the protocol

setupinitialPathMsg() is just for the purpose of sending the initial message from the destination mote since that has to be done outside the receive event since that only happens when the timer is fired.



```
void setupinitialPathMsg() {

  TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
  DelugePathMsg *pathMsg = (DelugePathMsg*)pMsgBuf->data;

    pathMsg->from = FROM;
    pathMsg->hopcount = HOPCOUNT;
    pathMsg->id = ID;
    pathMsg->dest = DEST;

  if (call SendPathMsg.send(pathMsg->dest, sizeof(DelugePathMsg), pMsgBuf) == SUCCESS){
   call Leds.greenToggle();
   }

}
```

Figure 2.9 Initial message send by the destination mote

This completes the process of setting the path from the source to destination. The rest of the protocol is as follows:

When receive any kind of Deluge messages from Base Station:

If the value of the “interested” is 1, forward it.

If the value is 2, accept it and take actions accordingly

If the value is 0, ignore the message.

For this purpose following code was added to the original deluge protocol:

DelugeAdvMsgFwd() in DelugeM.nc

SetupReqMsgFwd() in DelugePageTransferM.nc

SetupDataMsgFwd() in DelugePageTransferM.nc

We don’t need a separate send interface for Adv and Req message since it is already present in the protocol for these messages. There also exists a separate send interface for Data msg but still we need to have a different send interface for AdvFwd messages since the sendDone event of DataMsg actually starts a timer that does some actions after being fired, while with message forwarding, we do not want to do anything apart from simply forwarding the message so that it reaches the destination.

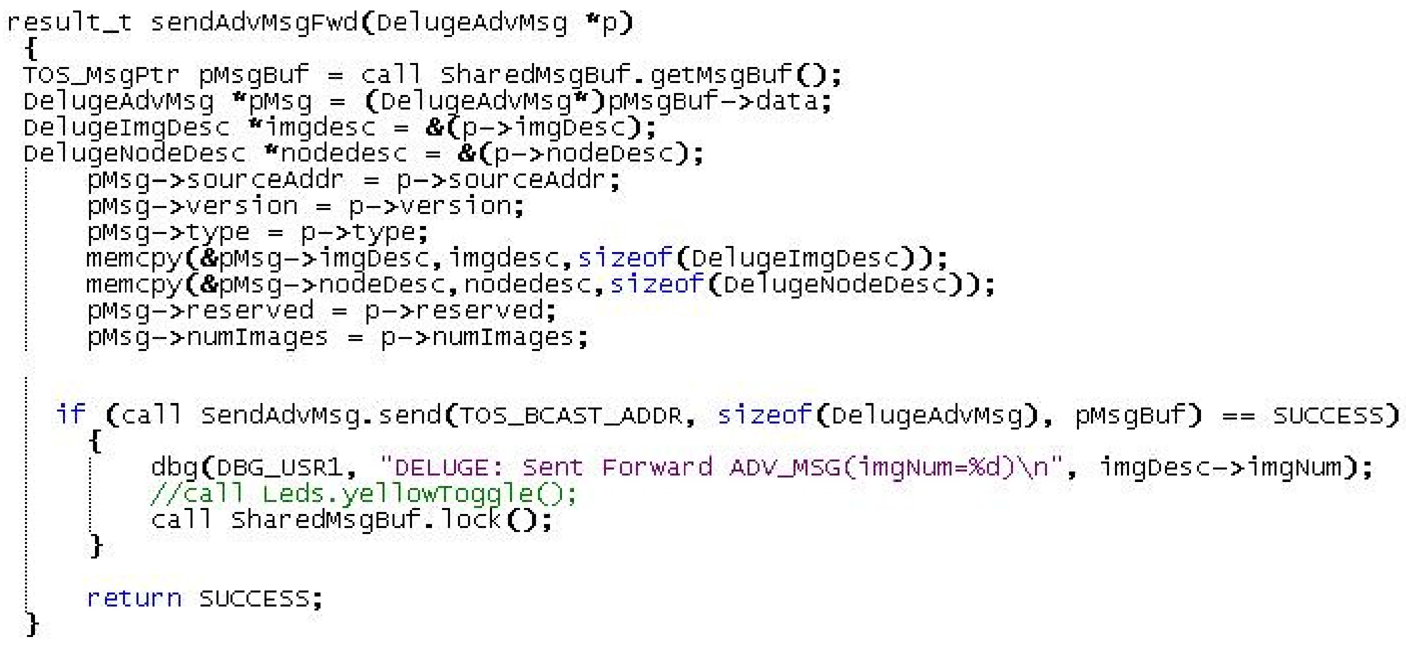
One very important point while writing the Forwarding functions is that whenever a function is being forwarded, it should be assured that exactly the same message is being forwarded as was received by the Base station, the reason being that there is some metadata associated with each and every packet that contains information about that packet. This metadata is used to decide what action needs to be taken after receiving the messages, so if this metadata is not forwarded correctly, there might not be any visible error in the protocol, but it won’t behave the way it should be and that can become pretty hard to debug

Let us discuss these 3 functions in detail:

DelugeAdvMsgFwd:

This method is for forwarding the Adv messages further so that they reach the destination.

The following snippet shows the SendAdvMsgFwd method:



```
result_t sendAdvMsgFwd(DelugeAdvMsg *p)
 {
 TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
 DelugeAdvMsg *pMsg = (DelugeAdvMsg*)pMsgBuf->data;
 DelugeImgDesc *imgdesc = &(p->imgDesc);
 DelugeNodeDesc *nodedesc = &(p->nodeDesc);
     pMsg->sourceAddr = p->sourceAddr;
     pMsg->version = p->version;
     pMsg->type = p->type;
     memcpy(&pMsg->imgDesc,imgdesc,sizeof(DelugeImgDesc));
     memcpy(&pMsg->nodeDesc,nodedesc,sizeof(DelugeNodeDesc));
     pMsg->reserved = p->reserved;
     pMsg->numImages = p->numImages;


   if (call SendAdvMsg.send(TOS_BCAST_ADDR, sizeof(DelugeAdvMsg), pMsgBuf) == SUCCESS)
     {
         dbg(DBG_USR1, "DELUGE: Sent Forward ADV_MSG(imgNum=%d)\n", imgDesc->imgNum);
         //call Leds.yellowToggle();
         call SharedMsgBuf.lock();
     }

     return SUCCESS;
 }
```

Figure 2.10 - Adv Message Forwarding

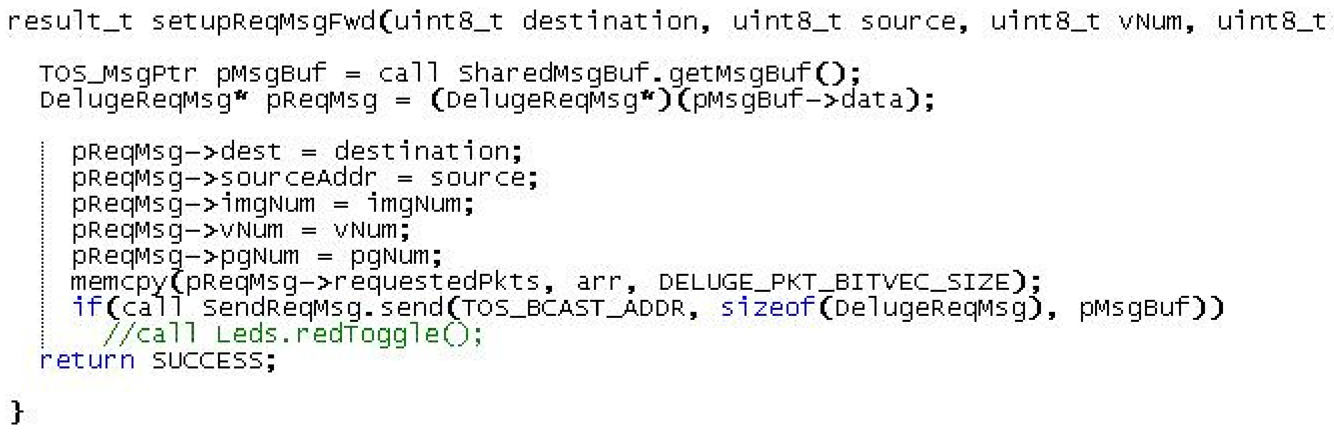
The function takes as arguments all the fields and pointers from the original message and prepares a new message with exact same structure and then forwards it.

SharedMsgBuf.getMsgBuf() is the command that is called to get the message buffer for preparing a new message and put the data in the data field of Tos_MsgPtr. It is important to make a copy of DelugeImgDesc* and DelugeNodeDesc* first and then use the memcpy function to copy the data from the copied version to the new message that is being prepared. The reason being that both the messages are using different buffers and if we try to use memcpy without making a copy of the pointer, it can go to a deadlock state where it is waiting to access the buffer but is never able to access the buffer.

SetupReqMsgFwd:

This method is for forwarding the Req messages further so that they reach the destination and also forward the request messages from motes to Base Station.

The following snippet shows the function body:



```
result_t setupReqMsgFwd(uint8_t destination, uint8_t source, uint8_t vNum, uint8_t

  TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
  DelugeReqMsg* pReqMsg = (DelugeReqMsg*)(pMsgBuf->data);

    pReqMsg->dest = destination;
    pReqMsg->sourceAddr = source;
    pReqMsg->imgNum = imgNum;
    pReqMsg->vNum = vNum;
    pReqMsg->pgNum = pgNum;
    memcpy(pReqMsg->requestedPkts, arr, DELUGE_PKT_BITVEC_SIZE);
    if(call SendReqMsg.send(TOS_BCAST_ADDR, sizeof(DelugeReqMsg), pMsgBuf))
      //call Leds.redToggle();
  return SUCCESS;

}
```

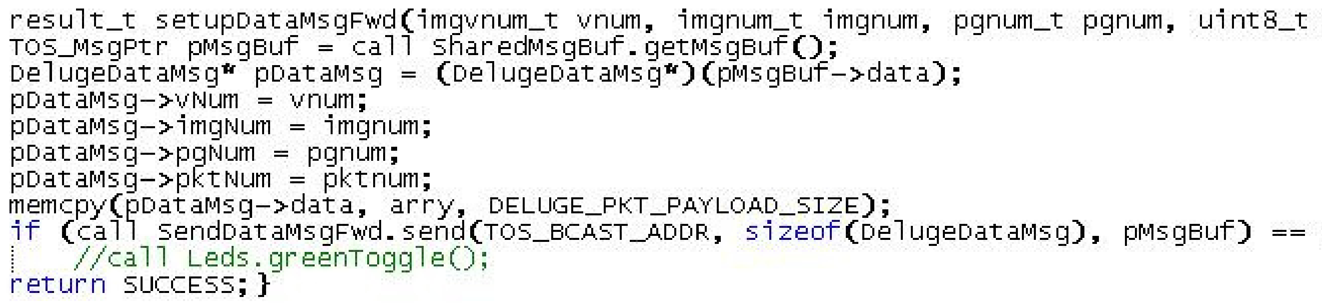
Figure 2.11 Req Message Forwarding

Similar to the DelugeAdvMsgFwd function, this method also takes as arguments all necessary fields and pointers from the original message and form a new message and pass it on.

SetupDataMsgFwd:

This method is for forwarding the data messages further so that they reach the destination and also forward the data messages from motes to Base Station.

The following snippet shows the function body:



```
result_t setupDataMsgFwd(imgvnum_t vnum, imgnum_t imgnum, pgnum_t pgnum, uint8_t
TOS_MsgPtr pMsgBuf = call SharedMsgBuf.getMsgBuf();
DelugeDataMsg* pDataMsg = (DelugeDataMsg*)(pMsgBuf->data);
pDataMsg->vNum = vnum;
pDataMsg->imgNum = imgnum;
pDataMsg->pgNum = pgnum;
pDataMsg->pktNum = pktnum;
memcpy(pDataMsg->data, arry, DELUGE_PKT_PAYLOAD_SIZE);
if (call SendDataMsgFwd.send(TOS_BCAST_ADDR, sizeof(DelugeDataMsg), pMsgBuf) ==
    //call Leds.greenToggle();
return SUCCESS;}
```

Figure 2.12 - Data Message Forwarding

Similar to the DelugeAdvMsgFwd function, this method also takes as arguments all necessary fields and pointers from the original message and form a new message and pass it on.

CHAPTER 3 - Implementation

The following steps must be followed to install tinyos.

- Install tinyos-1.11 from tinyos.net
- Upgrade to tinyos-1.15 by using the following rpm and issuing the following command.
- Replace the tinyos-1.x folder by the folder attached.

To run Deluge on any mote, do the following:
go to tinyos-1.x/apps/TestDeluge/FormatFlash and issue the following command:
Make micaz install mib510,/dev/ttySx
Where x = ComPort-1, ie if your base station is connected at COM4, the command would be using ttyS3.
NOTE: To check the comport being used
Right Click on My Computer->properties->Hardware->DeviceManager-> scroll down a bit and you can see the ports being used by your system.
Format Flash is a simple application to format the motes memory for each and every slot. It's a good practice to always run this application before starting Deluge.
After FormatFlash, go to DelugeBasic application in the same folder and issue to following command:
make micaz install.id mib510,/dev/ttySx
for telosb the command is
make telosb install.id bsl,x
id is the id with which you want to program the mote, x is explained before.
Deluge Basic is an application with minimum support for Deluge, its generally kept as Golden Image in slot 0 of each mote. If the motes run into an infinite loop or any other problem, just restart the mote 2-3 times continuously and they will reboot to the Golden Image hence avoiding the need to again program them for Deluge support. It is not necessary to have DelugeBasic as the Golden image, one can have any application as the Golden Image but it should be having the support for Deluge.

Now the motes are all set and one can disconnect them from the board and place them wherever required.

Now to form the base station, take a mote and program it with TOSBase application by going to tinyos1.x/apps/TOSBase by issuing the following command:

```
make micaz install.0 mib510,/dev/ttySx
```

Remember to assign the Id 0 to base station since it's a standard.

For telosb the command is

```
Make telosb install.0 bsl,x
```

To run Deluge on any application, we need to wire it with the current application. This can be done by going to the configuration file of the application and adding DelugeC to the list of components and then wiring it with the module like this

```
Main.StdControl -> DelugeC
```

We will take the example of Blink application to insert images into the network using Deluge.

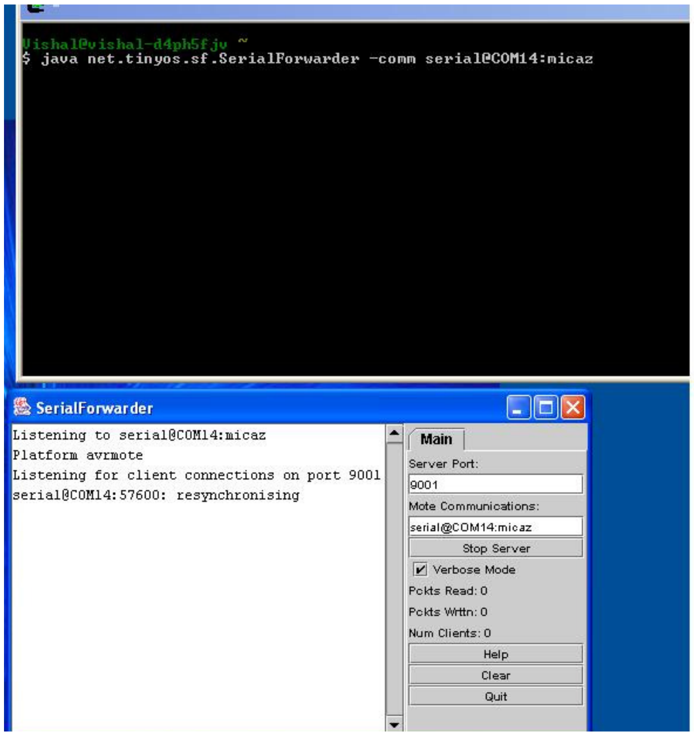
Go to tinyos-1.x/tools/java and give the following command

```
java net.tinyos.sf.SerialForwarder -comm serial@COMy:micaz
```

for telosb, the command is

```
java net.tinyos.sf.SerialForwarder -comm serial@COMy:telos
```

where y is the com port.. This opens the serial forwarder. Serial Forwarder is an application through which the port can read all the messages from the serial port and forwards them through an internet connection. If the serial forwarder opens successfully, you will see a window as shown in the following figure.



```
Vishal@vishal-d4ph5fjv ~
$ java net.tinyos.sf.SerialForwarder -comm serial@COM14:micaz
```

```
Listening to serial@COM14:micaz
Platform avrmote
Listening for client connections on port 9001
serial@COM14:57600: resynchronising
```

Server Port: 9001
Mote Communications: serial@COM14:micaz
[✓ Verbose Mode]
Pckts Read: 0
Pckts Wrttn: 0
Num Clients: 0

Figure 3.1 - serial Forwarder running

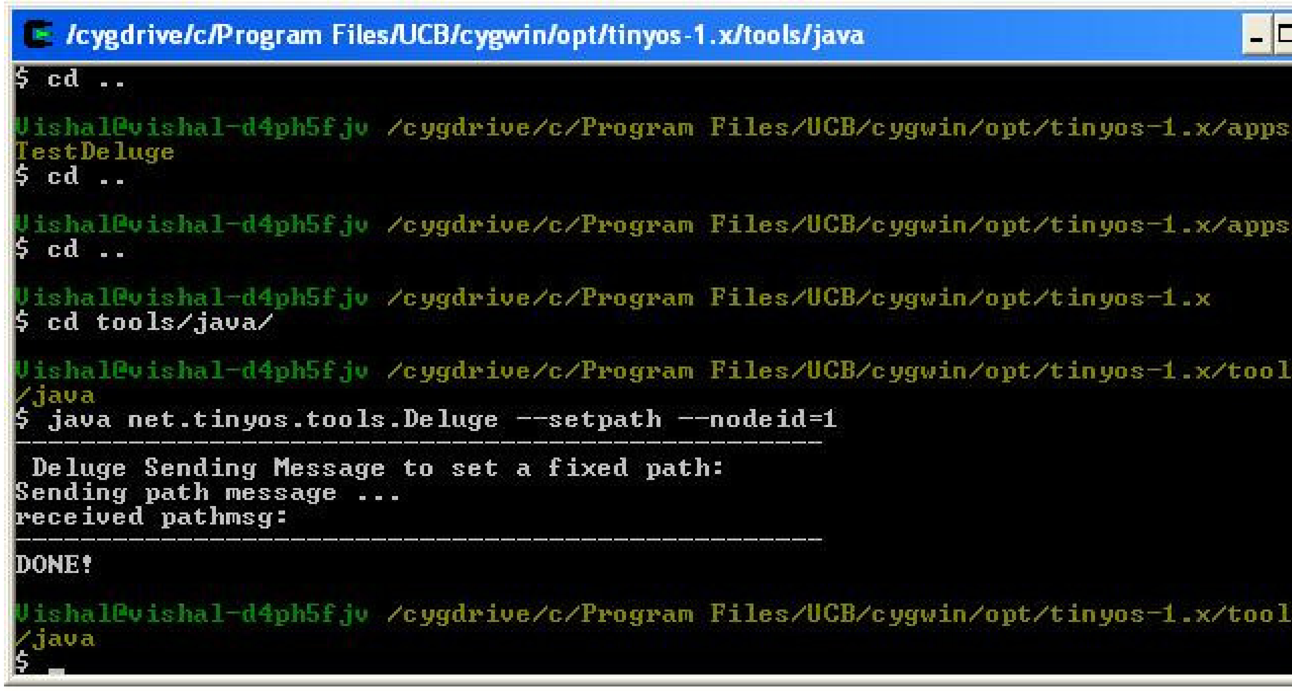
Go to tinyos-1.x/apps/Blink after changing its configuration file and issue the following command:

1 - make telosb

Now, in order to re-program an individual mote, we need to set the path to that mote. In order to do that, go to tinyos-1.x/tools/java and issue the following command:

Java net.tinyos.tools.Deluge --setpath --nodeid=m

where m is the node that we wish to program. If the path has been successfully set, you will receive a message confirming that the path has been set.


/cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/tools/java

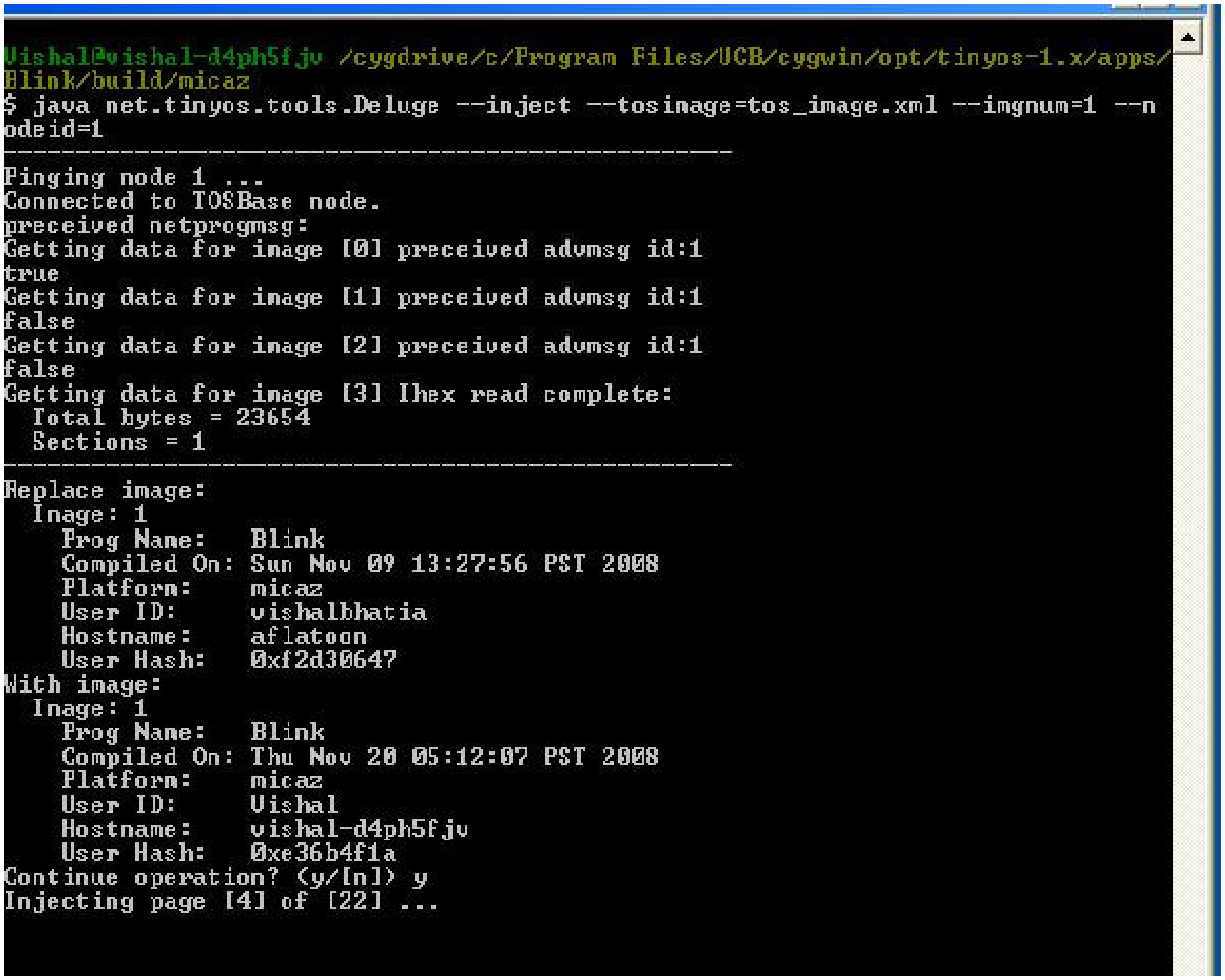
```
$ cd ..

Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/apps
TestDeluge
$ cd ..

Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/apps
$ cd ..

Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x
$ cd tools/java/

Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/tool
/java
$ java net.tinyos.tools.Deluge --setpath --nodeid=1
--------------------------------------------------
 Deluge Sending Message to set a fixed path:
Sending path message ...
received pathmsg:
--------------------------------------------------
DONE!

Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/tool
/java
$ _
```

Figure 3.2 - sending path message

NOTE: If the message is not received, that means that the path is not properly set, you need to again send the path setting command till the ack message is received.

Now go to tinyos-1.x/apps/Blink/build/micaz and issue the following command:

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



```
Vishal@vishal-d4ph5fjv /cygdrive/c/Program Files/UCB/cygwin/opt/tinyos-1.x/apps/
Blink/build/micaz
$ java net.tinyos.tools.Deluge --inject --tosimage=tos_image.xml --imgnum=1 --n
odeid=1
--------------------------------------------------
Pinging node 1 ...
Connected to TOSBase node.
preceived netprogmsg:
Getting data for image [0] preceived advmsg id:1
true
Getting data for image [1] preceived advmsg id:1
false
Getting data for image [2] preceived advmsg id:1
false
Getting data for image [3] Ihex read complete:
  Total bytes = 23654
  Sections = 1
--------------------------------------------------
Replace image:
  Image: 1
    Prog Name:   Blink
    Compiled On: Sun Nov 09 13:27:56 PST 2008
    Platform:    micaz
    User ID:     vishalbhatia
    Hostname:    aflatoon
    User Hash:   0xf2d30647
With image:
  Image: 1
    Prog Name:   Blink
    Compiled On: Thu Nov 20 05:12:07 PST 2008
    Platform:    micaz
    User ID:     Vishal
    Hostname:    vishal-d4ph5fjv
    User Hash:   0xe36b4f1a
Continue operation? (y/[n]) y
Injecting page [4] of [22] ...
```

Figure 3.3 - Injecting an image into the network

You can see the node m being programmed and the nodes in the path forwarding and the rest of the nodes just sitting idle.

REMEMBER: If we are just injecting or erasing the images into a particular mote, we do not need to send a path message again and again, but is we are sending a reboot message to that mote, the variables will get reset and we have to again send a path message to that mote to send other commands.

CHAPTER4: Testing and Performance Analysis

The application was tested for two different hardwares. The Broadcasting power of motes and the Base Station was reduced using CC2420Control interface for the purpose of testing.

1 – The first setup was using the Micaz hardware that has the following specifications:

ATMega 128L Processor
802.15.4 Radio Transceiver
2.4 GHz, IEEE 802.15.4 compliant
250 kbps, High Data Rate Radio

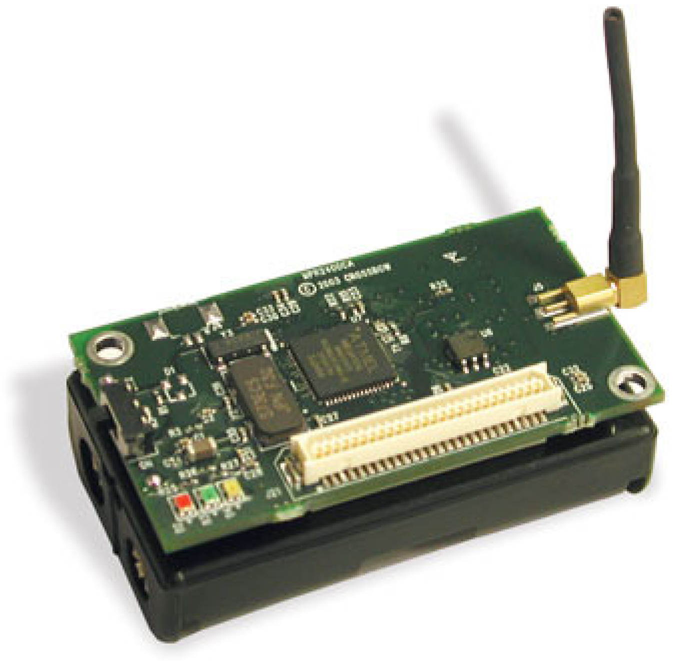


Figure 4.1 micaz motes

The application was tested with 4 different applications injected into the network on different motes based on the path settings:

The following table shows the observations:

Application	Size of the application(in terms of pages to be injected)	Time to inject from Original protocol	Time to inject from Modified protocol
Blink	22	48 seconds	51 seconds
CountToLedsAndRfm	21	45 seconds	50 seconds
SenseToLeds	22	40 seconds	42 seconds
Oscilloscope	23	59 seconds	65 seconds

Table 4.1 Observation table for micaz

The results are based on the injection for the motes that were not in direct contact with the Base Station and were programmed using the Forwarding protocol. The time of injection shown is taken by averaging the times taken to inject an image into a particular mote keeping it at different locations and also based on the number of motes involved in forwarding, ie the number of motes that were a part of the message path.

There can be a great variation in the results because of the battery power supplied to a micaz mote. Its advisable to keep the battery more than 2.5 V for micaz to work properly.

2 – The second setup was using the telosb hardware that has the following specifications:

250 kbps, high data rate radio.
TI MSP430 microcontroller with 10kB RAM

Figure 4.2 - telosb motes

The experiment was done with 3 different applications injected into the network on different motes based on the path settings. These applications were chosen because of the difference in the number of pages they contain since the testing is also based on the size of the message being injected. Since the test bed had a limited size, the application was tested by reducing the RF Power to 2 so that forwarding part can be tested reliably. Three types of scenarios were considered to check the performance under various conditions:
1 – Direct interaction of destination with Base Station – 1 hopcount
2 – Interaction through one forwarder – 2 hopcount

3 – Interaction through two forwarders – 3 hopcount

NOTE: The following table is based on the fact that the TOSBase has to be restarted again and again since the TOSBase has a queue length of 12 which gets filled up pretty quickly since there are a lot of messages received by TOSBase when we are injecting images into multiple motes at a time. This problem has been taken care of in the modified protocol by modifying TOSBase as has been discussed earlier.

The following figure shows the scenario for 1st case.

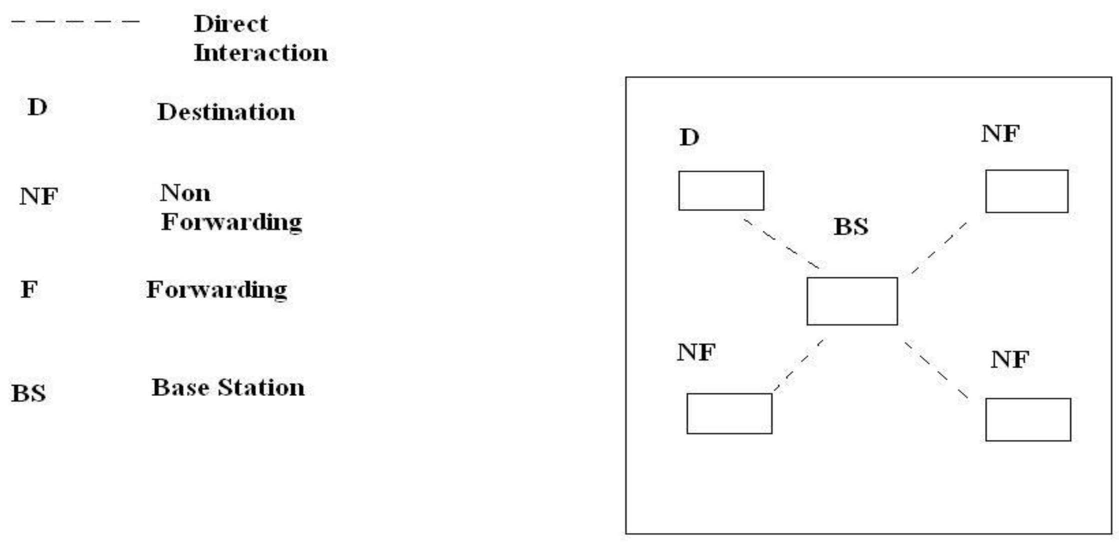


Figure 4.3 test case1

The following table shows the observations for 1st case **(HOPCOUNT = 1 - this means we are injecting images directly through TOSBase to destination)**

Application	Size of the application(in terms of pages to be injected)	Average Time to inject from Original protocol	Average Time to inject from Modified protocol with 0 hops in between
Blink	20	1 min 25 seconds	1 min 13 seconds
OscilloscopeRF	23	1 min 39 seconds	1 min 25 seconds
GroupCoord6	27	4 min 15 seconds	1 min 41 seconds

Table 4.2 - Observation table for telosb

These are the observations based on the direct interaction of the motes with the TOSBase.

Experiments with hopcount 1: There are 2 ways to give issue commands for injection or as a matter of fact any command in original deluge. One is where we specify the nodeid to be programmed specifically and another one where we do not specify any nodeid and both are generic messages for all the motes in the network. The difference is that when moteid is specified, the TOSBase first establish a connection with that mote and then starts injecting messages into the network. In this case if the destination is some hops away, it takes too long(TL)(beyond 10 minutes) to even establish a connection of TOSBase and destination. The reason being that in case where destination is far away, it periodically requests the intermediate mote to send the image but the intermediate mote doesn't even have the image since TOSBase has not started injecting the images since its waiting to establish a connection first. However whene the nodeid is not specified, the moment TOSBase starts injecting images, the first mote nearest to the Base Station will start receiving the images and sending the acknowledgments to the Base Station and then this mote can be used to fulfill the request of other motes.

However there is one drawback to the latter one. If we want to inject images to the very last mote but do not specify the nodeid, the first mote will receive all the messages and send all the acknowledgments to the base station indicating that the images have been properly injected into the network, which is true apart from the fact that the injection might not have been completed for the whole network. This scenario was tested and it was found that sometimes only partial motes have been programmed and the rest have been not. After that it may take any amount of time for the partially injected motes to interact with the ones that have full images and synchronize themselves based on what RFPower has been used.

The following figure shows the scenario for the 2nd case.

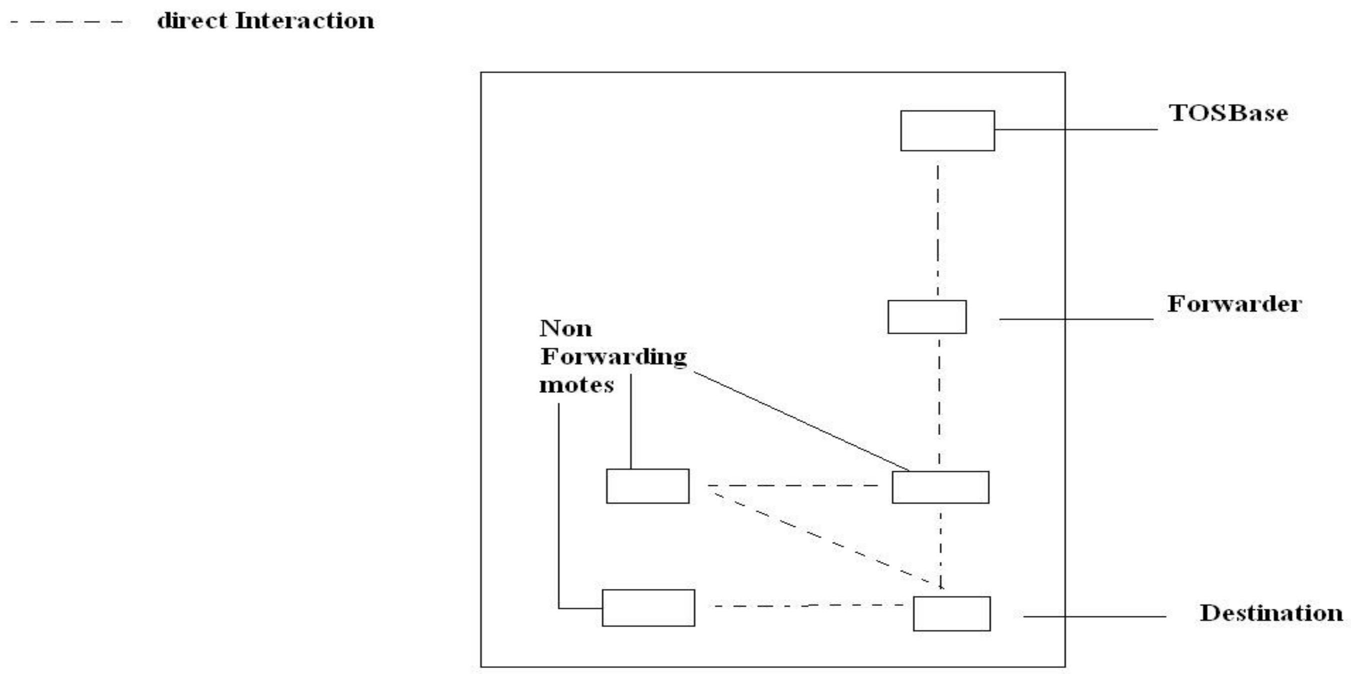


Figure 4.4 test case 2

The following table shows the observations for the 2nd case **(HOPCOUNT = 2- this means we are injecting images through 1 forwarder in between TOSBase and destination)**

Application	Size of the application(in terms of pages to be injected)	Time to inject from Original protocol	Time to inject from Modified protocol with 1 hop in between
Blink	20	TL(see comments for hopcount1 and 2)	2 min 24 seconds
OscilloscopeRF	23	TL(see comments for hopcount1 and 2)	2 min 27 seconds
GroupCoord6	27	TL(see comments for hopcount1 and 2)	3min 24 seconds

Table 4.3 - Observation table for telosb

The following figure shows the scenario for 3rd case

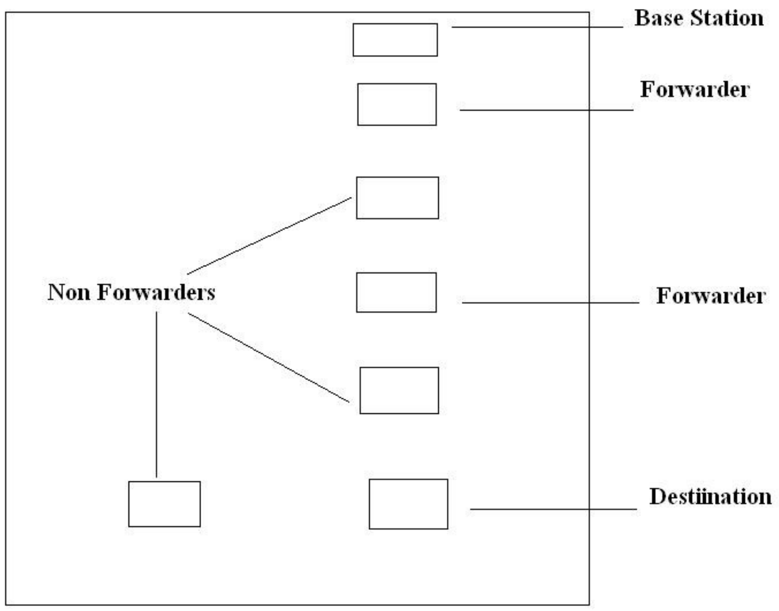


Figure 4.5 test case 3

The following table shows the observations for the 3rd case **(HOPCOUNT = 2- this means we are injecting images through 2 forwarders in between TOSBase and destination)**

Application	Size of the application(in terms of pages to be injected)	Time to inject from Original protocol	Time to inject from Modified protocol with 1 hop in between
Blink	20	TL(see comments for hopcount1 and 2)	2 min 25 seconds
OscilloscopeRF	23	TL(see comments for	3 min 31 seconds

		hopcount1 and 2)	
GroupCoord6	27	TL(see comments for hopcount1 and 2)	3 min 40 seconds

Table 4.4 - Observation table for telosb

As can be seen from the observation table, there can be a bit delay in the time taken to inject messages, that too in the cases where forwarding needs to take place, but on the other hand there is a great saving of the resources available to us in terms of mote's memory to store images. Now, each mote can have 4 different images in its slot as compared to the original protocol where the whole network can have maximum 4 applications at a time. Also, it increases the range through which a mote can be programmed drastically.

Apart from the time performance testing, various other kinds of tests were done to see if the protocol gets stuck at a certain point. In a network of 8 motes, Blink application was injected after setting the path to nodeid=8, after that Oscilloscope was being injected to nodeid=1 by setting the path to 1. nodeid 8 was then rebooted to Blink application and nodeid 1 was rebooted to OscillscopeRF application. It was seen that the both the motes ran different applications without interfering with each other and the rest of the motes were sitting idle and could be programmed with different applications. It was interesting to see that till we are injecting images and erasing or dumping images to a particular mote, we do not need to set the path again and again but if we inject a reboot command to a mote, the variables are all reset since the whole application reboots. In this case, we have to again set the path to do further interaction with the same mote.

So, if we are taking even about not a very large network, say having 10 motes, with this protocol, there can be 40 different applications existing in the network as compared to the original protocol that can support only 4.

CHAPTER 5: Future Enhancements

One of the future things to be looked at is the message failure detection. While injecting the path message, if we are not able to establish a path between the source and the destination the first time since the destination mote can crash at any point of time, it should be able to automatically take care of injecting another path message and so on till the connection is established. Right now, this part has to be done manually each and every time we are unable to establish a connection.

Secondly, there might be a situation when the mote was working while setting the path but after setting the path, it crashed at some point of time. In this case, it won't forward the images properly and we won't be able to detect where the things are going wrong. For that purpose, we have a minimum number of attempts it tries to inject images into the network. If it cannot inject complete image within those number of attempts, it goes into an idle state. In the modified protocol, there is no way to identify whether the destination mote or the forwarder crashed. This is one more issue that can be looked at in future.

CHAPTER6: Conclusion

We have modified the Deluge protocol to allow each mote to be individually programmed. It can be seen from the above experiments that the modified protocol works better and gives better results in terms of space and time both since now it is possible to have multiple images in the network (400 different images in a network of 100 motes) as compared to Deluge that only allows few images (4 images in a network of 100 motes)

Apart from the space, there is also a great saving in terms of time since we do not need to pluck the individual motes from their locations and program them again and again with different applications, this can be now achieved by using the modified protocol without plucking the motes from their respective locations, and that too with a greater time efficiency

References

[1] Tinyos concepts and Nesc Programming language

http://www.tinyos.net

[2] Tiynos and Nesc programming problems

http://mail.millennium.berkeley.edu/pipermail/tinyos-help

[3] Trickle

http://crewman.uta.edu/~pradip/papers/DCOSSPosterDraft.pdf

[4] XOTAP

http://www.willow.co.uk/MoteWorks_OEM_Edition.pdf

[5] Nabble

http://www.nabble.com/nesC-and-TinyOS-f6356.html