Appendix B

Matlab and TinyOS: getting them to work together

B.1 Introduction

In this chapter we will go through the different aspects which are needed to be taken into account when interconnecting Matlab and TinyOS, as well as the way in which they have to be used. We will show why Matlab is a suitable tool for interacting with WSNs.

Matlab is a scripting language in which large applications can be written and is interpreted, which means that it is slower than other languages like C or Java. However, being an interpreted scripting language is also part of what makes Matlab an appealing way to interact with a sensor network: the user can interact with the network by calling commands on the Matlab command line. In contrast, once a java application is started, it can only be controlled through a GUI.

It is possible to connect Matlab with the motes to receive and inject information from and to the network. The system architecture will consist of a sensor (or several) connected to the computer through the USB port, acting as a data-gathering node, and many others sensing nodes. The sink node will behave as base station receiving all the packets addressed to its network address. From Matlab we will be able to connect to this mote, listening to the desired packets in the network and filtering the rest of them.

We will also have full access to the message structure having the possibility of accessing/editing the packets'fields in a java-oriented syntax. It will be also possible to send packets to any sensor in the network, provoking reactions (if programmed beforehand) which will make our network reactive.

Last but not least, Matlab is a complete programming development environment having numerous tools for helping the developer in their work. Thus, Matlab appears as an ideal environment for an easy and fast development of a broad range applications.

B.2 Setting up the Matlab environment to use it with TinyOS and Java

Step 1 The Matlab directory structure provided by TinyOS was meant to mirror that of the tinyos-1.x directory. Each of the directories is meant to serve the following purposes:

APPS holds Matlab functions that were built for a certain tinyOS applications, e.g. oscilloscopeRF.

CONTRIB contains subdirectories to mirror tinyos-x.x/contrib for matlab applications.

LIB tools that correspond to tinyOS components in tos/lib.

TOOLS Matlab functions or apps that are generally useful but do not relate specifically to one app (e.g. 'listen.m').

UTIL functions (not apps) that may be shared among several other Matlab apps, eg. message processing utilities.

Add to your matlab directory in: *your_path_to_tinyos-1.x\tools\matlab* those directories above which are not present with the distribution.

Step 2 Create the file ***startup.m*** in the folder *your_path_to_Matlab\toolbox\local*. Edit the file with the following code:

```
flag=0;
global TOSDIR
TOSDIR='your_path_to_UCB\cygwin\opt\tinyos-1.x\tos';
addpath your_path_to_tinyos-1.x\tools\matlab;
addpath your_path_to_tinyos-1.x\tools\matlab\comm;
```

```
addpath your_path_to_tinyos-1.x\tools\matlab\apps;
addpath your_path_to_tinyos-1.x\tools\matlab\lib;
addpath your_path_to_tinyos-1.x\tools\matlab\util;
addpath your_path_to_tinyos-1.x\tools\matlab\tools;
addpath your_path_to_tinyos-1.x\tools\matlab\contrib;


defineTOSEnvironment;
```

Basically, what we do in this Matlab script is to set up the Matlab path and call the script ***defineTOSEnvironment.m***, which in turn, will initialize the Matlab comm (communications)stack. By giving the script the name ***startup.m*** and placing it in the aforementioned directory, we ensure the file is executed when Matlab starts up.

Step 3 Edit the file ***defineTOSEnvironment.m*** that you can find in *your_path_to_tinyos-1.x\tools\matlab* so that it looks like:

```
global DEBUG
DEBUG = 0;


global COMM
COMM.TOS_BCAST_ADDR = 65535;
COMM.TOS_UART_ADDR = 126;
COMM.GROUP_ID = hex2dec('7D');


defineComm;
```

Step 4 Copy the file ***comm.jar*** located in the following directory:
your_path_to_UCB\jdk1.4.1_02\j2sdk1.4.1_02\jre\lib\ext to
your_path_to_tinyos-1.x\tools\java

Step 5 Open the Matlab file ***classpath.txt*** and add the following pathes:

```
your_path_to_tinyos-1.x\tools\java
your_path_to_tinyos-1.x\tools\java\comm.jar
your_path_to_tinyos-1.x\tools\java\net\tinyos\message
```

Step 6 Copy the files ***win32com.dll*** and ***getenv.dll*** located in
your_path_to_UCB\jdk1.4.1_02\j2sdk1.4.1_02\jre\bin
and the folder *your_path_to_tinyos-1.x\tools\java\jni* in the Matlab folder
your_path_to_Matlab\sys\java\jre\win32\jre1.4.2\bin

Also copy the file ***javax.comm.properties*** located in the folder
your_path_to_UCB\jdk1.4.1_02\lib to the destination folder:
your_path_to_Matlab\sys\java\jre\win32\jre1.4.2\lib

Step 7 Edit the file ***cygwin.bat*** to include in the system classpath the following pathes:

your_path_to_Matlab\java\jar\jmi.jar
your_path_to_UCB\cygwin\opt\tinyos-1.x\tools\java\comm.jar
your_path_to_UCB\cygwin\opt\tinyos-1.x\tools\java

Step 8 The `net.tinyos.matlab.MatlabControl` class is needed to call Matlab commands from Java. Previously to compilation is mandatory to fix a bug in the ***MatlabControl.java*** file, by using the provided patch file.

Once it has been fixed, proceed to compile the folder *your_path_to_tinyos-1.x\tools\java\net\tinyos\matlab* by typing `make matlab`.

Please, note that you should have already included the jar file ***jmi.jar*** in your CLASSPATH environment variable (otherwise it returns the error: `package comm.mathworks.jmi does not exist`)

Step 9 Before you are able to use the Matlab functions ***(connect, receive, send and stopReceiving)*** provided along the TinyOS distribution, you will have to go through several typographical errors:

1. Edit the file ***receive.m*** as below:
   - Delete line 52
     (`moteIFs = [COM.sourceMoteIF{TF}];`) to place instead:

```
if isempty(TF)
    TF=0;
else
    TF=1;
end
if TF==0
    moteIFs=[];
else
    moteIFs = [COM.sourceMoteIF{TF}];
end
```

   - In line 63 take the transpose away.
2. Edit the script ***stopReceiving.m*** as follows:
   - Substitute the code lines:
     `COMM.globalFunction=COMM.globalFunction{~TF};`

```
COMM.globalMessageType=COMM.globalMessageType{~TF};
COMM.globalMessageName=COMM.globalMessageName{~TF};
```

by

```
COMM.globalFunction={COMM.globalFunction{~TF}};
COMM.globalMessageType={COMM.globalMessageType{~TF}};
COMM.globalMessageName={COMM.globalMessageName{~TF}};
```

- And the code lines:

```
for i=1:length(varagin)
    receive(functionName, message, varargin{i})
```

by

```
for i=1:length(varargin)
    stopReceiving(functionName, message, varargin{i})
```

Step 10 To finish with, it should be noticed that there still exist some bugs in the Java code used by the Matlab scripts that will make necessary to check the code carefully for every application.

B.3 Using the TinyOS java tool chain from Matlab

It is often easier to use an existing Java tool with TinyOS than to rewrite it in Matlab. Thus, we can use Matlab to launch the Serial Forwarder, the Oscilloscope application seen in the TinyOS tutorial [29], etc.

If we want to start the Oscilloscope application from Matlab, we should enter the following command:

```
net.tinyos.oscope.oscilloscope.main('125')
```

You should see the Java GUI open and connect to your serial forwarder.

When using Java from Matlab the syntax remains basically the same, except that there is no ”new” operator and functions with no arguments do not terminate with empty parenthesis ”( )”.

Every time a value is returned or passed to a Java method, a conversion of types takes place automatically (according to the information provided in

the Matlab help). Most of the arguments are passed by value except for the Java objects which are passed by reference.

Using Java classes from Matlab reveals two common bugs that should be fixed before you obtain a successful execution. While both of these bugs do not appear as such when running the java program from the default Java environment (i.e. starting a Java application from the command line), they do it when the program is called from Matlab. Thus, you will find them in many Java classes, including those in the TinyOS Java toolset.

A. Command Line Arguments In this section you will learn with a very simple example how to pass arguments from the Matlab command line to a Java method.

Imagine you want to run the Serial Forwarder to listen to packets arriving at port COM7. If you are calling the program from your default shell you should type the following command:

```
java net.tinyos.sf.SerialForwarder -comm serial@COM7:telos
```

To run the same program from the Matlab command line you should write the command:

```
net.tinyos.sf.SerialForwarder.main({'-comm','serial@COM7:telos'})
```

From the previous example we can deduce that the shell automatically packages up the command line arguments into string arrays before passing them to the main function of the class being called. In Matlab this has to be explicitly done by directly passing the arguments as string arrays.

In case we do not want to pass any argument, we should send a null array, which is done as:

```
net.tinyos.sf.SerialForwarder.main({ })
```

We get a null pointer exception! This is because the static main function in the main class of the SerialForwarder uses the string before checking if it is null. Since sending no command-line arguments from the shell does not result in a null string being passed, this normally does not cause an error. However, this should be fixed if this class were to be used from Matlab.

B. Virtual Machines You will have already realized that when we run a program in Matlab we do not use the command *java*. This is due to the fact that in Matlab we instantiate the objects within the same JVM (Java Virtual Machine) in which the Matlab session is running. This is only important for the `java.lang.System` class, which directly refers to the JVM you are running in; `java.lang.System.exit()` will kill your JVM, and therefore all the classes and your Matlab session! You will see this if you close the SerialForwarder window, because this causes a call to `System.exit()`. Hence, `System.exit()` should never be called.

B.4 Using Matlab with TinyOS

B.4.1 Preparing the message

Prior to connecting the computer to the network we need to build the messages to which we want to listen to. To illustrate this we will go through a simple example and we will make use of the available tools we have.

Let us imagine we have a network in which the base station is polling the nodes one by one and asking them to send back an application-specific data.

We need to construct two kinds of messages, one carrying the request for data to the sensors (*SimpleCmdMsg*) and another one carrying the data sent back by the sensors (*DataMsg*).

We can build two header files each one with the message structure we have devised, for example, if we list the code corresponding to the file ***SimpleCmdMsg.h***:

```
enum {
     AM_SIMPLECMDMSG = 18
};

typedef struct SimpleCmdMsg {
     uint16_t dst;
     uint16_t source;
     uint16_t seqno;
```

```
    uint8_t type;
    uint8_t focusaddr;
    uint8_t newrate;
    uint8_t hop_count;
    uint8_t bitsT;
    uint8_t bitsH;
    uint8_t bitsLtsr;
    uint8_t bitsLpar;
} SimpleCmdMsg;
```

Once we have the messages in the header files, we can use the MIG tool (see [29] for further information) to automatically generate Java classes which take care of the cumbersome details of packing and unpacking fields in the message's byte format. Using MIG saves you from parsing message formats in your Java application.

Once we have the output from MIG: *SimpleCmdMsg.java* and *DataMsg.java* we can proceed to compile them, obtaining the respective *.class* files.

Now we can easily instantiate these objects in Matlab.

B.4.2 Connecting Matlab to the network

This subsection does not intend to be a detailed guide of how to use Matlab with TinyOS because it would be a redundant work over that present in [29]. We will just try to provide a basic understanding of the overall working by continuing with the example we started in section B.4.1.

The first step is to connect your Matlab session to your network (namely to your base station). If you are working with Tmotes this can be done as:

```
connect('serial@COM7:telos');
```

Where, same as before, we are assuming that your base station is identified by the serial port `COM7` (you can use the command `motelist` in your cygwin environment to check what devices are connected to your computer).

Once you have done this, you can instantiate the MIG message, which is a Java class that is a subclass of `net.tinyos.message.Message`. In Matlab you can instantiate Java objects on the command line as follows:

```
dataMsg=net.tinyos.report.DataMsg
```

Now you are prepared to start receiving packets:

```
receive('handleMsg',dataMsg)
```

This command specifies the Matlab function ***handleMsg*** as the one in charge for handling the received messages, and the *DataMsg* messages as the ones to which the base station is listening to. Any other arriving packet will be discarded.

If we want to send a packet to a node:

```
send(3,simpleCmdMsg)
```

Where *simpleCmdMsg* is an instance of the class SimpleCmdMsg sent to the node with network address 3.

At this point, the *DataMsg* objects should be printed to your screen every time a message of this type is received. To stop this behaviour you can use the `stopReceiving` command to deregister your Matlab function as a message handler:

```
stopReceiving('handleMsg',dataMsg)
```

Finally, you can disconnect yourself from the sensor you are connected to with the command:

```
disconnect('serial@COM7:telos')
```