Mobile and Ubiquitous Computing
TinyOS application example

Niki Trigoni
www.dcs.bbk.ac.uk/~niki
niki@dcs.bbk.ac.uk

Application

Consider an application with the following functionality:

- The gateway node sets a timer to fire every X msecs (say X=500msecs)
- When the timer is fired the gateway node increments a counter by one (resetting it to 0 when it becomes 100) and does two things:
  - It sends the counter value to the computer through the serial port.
  - It broadcasts a message with the counter value.
- Other nodes within one hop from the gateway receive the gateway's message (with the gateway's counter value) and toggle the leds to reflect the counter value (CV).
  - if CV < 30, toggle the red leds
  - if 30 <= CV < 60, toggle the yellow leds
  - if 60 <= CV <=100, toggle the green leds

Useful interfaces

- Leds
- Timer
- SendMsg
- ReceiveMsg

} **parameterised interfaces**

A parameterized interface allows a component to provide *multiple instances* of an interface that are parameterized by a runtime or compile-time value. For more details read: http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson2.html.

For example, the **TimerC** component provides 256 instances of the **Timer** interface, one for each **uint8_t** value:

… provides interface Timer[uint8_t id]; …

Useful components

- LedsC
  - **… provides interface Leds; …**
- TimerC
  - **… provides interface Timer [uint8_t id]; …**
- GenericComm
  - **… provides interface SendMsg [uint8_t id]; …**
  - **… provides interface ReceiveMsg [uint8_t id]; …**

Message type

- We must declare the type (and structure) of the messages that will be sent and received in the application.

- Create a file BroadcastCount.h

```
#ifndef    BROADCAST_COUNT_H
#define    BROADCAST_COUNT_H

enum {     AM_COUNT_MSG = 100  };

typedef struct Count_Msg
{
            uint16_t value;
} Count_Msg;      #endif
```

Configuration ‘BroadcastCountC’

```
includes BroadcastCount;

configuration BroadcastCountC {}

implementation {

    components Main, BroadcastCountM, LedsC, TimerC, GenericComm as Comm;

    Main.StdControl -> BroadcastCountM;
    Main.StdControl -> Comm.Control;
    BroadcastCountM.Leds -> LedsC;
    BroadcastCountM.ReceiveCountMsg -> Comm.ReceiveMsg [AM_COUNT_MSG];
    BroadcastCountM.SendCountMsg -> Comm.SendMsg [AM_COUNT_MSG];
    BroadcastCountM.CountTimer -> TimerC.Timer[unique("Timer")];

}
```

Include the .h file that describes the message structure

Radio handling

What is the meaning of the following code?

BroadcastCountM.SendCountMsg ->
Comm.SendMsg[AM_COUNT_MSG]

- The GenericComm component provides 256 different instances of the SendMsg interface, one of which is **SendMsg[AM_COUNT_MSG]**.
- Messages have handler IDs that reflect their type. The messages of this application have handler ID AM_COUNT_MSG.
- **BroadcastCountM** ***uses*** the interface **SendMsg** (with the alias **SendCountMsg**), which is ***provided*** by **GenericComm** (with the alias **Comm**).
- **GenericComm** (with the alias **Comm**) ***provides*** interface instance **SendMsg [AM_COUNT_MSG]**, which is used by **BroadcastCountM** to send messages of type AM_COUNT_MSG.

Module ‘BroadcastCountM’

```
includes BroadcastCount;
module BroadcastCountM {
    provides {
        interface StdControl;
    }

    uses {
        interface SendMsg as SendCountMsg;
        interface Leds;
        interface ReceiveMsg as ReceiveCountMsg;
        interface Timer as CountTimer;
    }
}
implementation {  …  }
```

‘BroadcastCountM’ local variables

```
includes BroadcastCount;
module BroadcastCountM {}
implementation {

    uint16_t value;          // value of the incoming counter message
    uint8_t serial=0;        // flag that shows whether a message was just sent to
                             // the serial port, or whether it was just broadcast
    TOS_Msg message;         // structure to store an outgoing or incoming message
    uint16_t counter = 1;    // value of the counter

    …
}
```

‘BroadcastCountM’ provides ‘StdControl’

```
implementation {        // implementation of BroadcastCountM
    command result_t StdControl.init() {
        call Leds.init();
        return SUCCESS;
    }

    command result_t StdControl.start() {
        //Gateway
        if (TOS_LOCAL_ADDRESS==0)
        {
                call CountTimer.start( TIMER_REPEAT, 500 );
                call Leds.redOn();
        }
        return SUCCESS;
    }
```

‘BroadcastCountM’ provides ‘StdControl’

```
// continued from previous page
…
command result_t StdControl.stop() {
        //Gateway
        if(TOS_LOCAL_ADDRESS==0)
        {
                call CountTimer.stop();
                call Leds.redOff();
        }
        return SUCCESS;
}

…
```

‘BroadcastCountM’ uses ‘ReceiveMsg as ReceiveCountMsg’

```
event TOS_MsgPtr ReceiveCountMsg.receive (TOS_MsgPtr receivedMessage) {
    // if the current node is not the gateway
    if (TOS_LOCAL_ADDRESS != 0)
    {
        Count_Msg * payload;
        payload = (Count_Msg *) receivedMessage->data;
        value = (uint16_t) payload->value;
        if (value<30) {
            call Leds.redToggle();
            call Leds.greenOff();
            call Leds.yellowOff();
        } else if (value>=30 && value<60)  { …}
        else {…}
    }
    return receivedMessage;
}
```

‘BroadcastCountM’ uses ‘Timer as CountTimer’

```
event result_t CountTimer.fired() {

    Count_Msg * payload;
    call Leds.greenOn();
    payload = (Count_Msg *) message.data;
    payload->value = counter;
    //BCAST for radio, UART for serial
    call SendCountMsg.send(TOS_UART_ADDR, sizeof(Count_Msg), &message);
    return SUCCESS;
}
```

‘BroadcastCountM’ uses ‘SendMsg as SendCountMsg’

```
event result_t SendCountMsg.sendDone(TOS_MsgPtr sentMessage, result_t result) {
    if (serial==0)
    // if the sendDone was signalled as a result of sending to the serial port
    {
        serial=1;
        call SendCountMsg.send(TOS_BCAST_ADDR, sizeof(Count_Msg), &message);
    }
    else // if the sendDone was signalled after broadcasting to neighbor nodes
    {
        serial=0;
        counter++;
        if(counter>100) counter=1;
        call Leds.greenOff();
    }
    return SUCCESS;
}
```

Java class ‘ListenCount.java’ that reads messages from the serial port

```
import net.tinyos.tools.*;
import java.io.*;
import net.tinyos.packet.*;
import net.tinyos.util.*;
import net.tinyos.message.*;

public class ListenCount {

    public static void main(String args[]) throws IOException {

        PacketSource reader = BuildSource.makePacketSource();
        if (reader == null) {
            System.err.println("Invalid packet source (check your MOTECOM environment variable)");
            System.exit(2);
        }
        // continued …
```

Java class ‘ListenCount.java’ that reads messages from the serial port

```
  // … continued from last page
  try {
        reader.open(PrintStreamMessenger.err);
        for (;;) {
          byte[] packet = reader.readPacket();
          double first = (double)unsignedByteToInt(packet[11]);
          double second = (double)unsignedByteToInt(packet[10]);
          double light = 256.0d*first+second;
          System.out.println(light);
          System.out.flush();
        }
      }
      catch (IOException e) {
          System.err.println("Error on " + reader.getName() + ": " + e);
      }
  }
  public static int unsignedByteToInt(byte b)
  {
     return (int) b & 0xFF;
  }
}
```

How to run the application

- Go to the directory where the code of the application is
- Connect a sensor node to the serial port
- Write ‘motelist’. This command should return which port the sensor node uses to connect to the computer (say this is COM7)
- Write ‘export MOTECOM=serial@COM7:tmote’
- MOTECOM is an environment variable that Java uses to know which port it should listen to
- ‘make tmote reinstall.0’ // to install code to the gateway node
- ‘make tmote reinstall.1’ // to install code to another node
- run the ListenCount program to listen to the serial port