

Octopus :: Developer Documentation

Summary

I. Global Structure.....	2
II. Communication Protocol.....	3
III. The GUI.....	4
1. Makefile.....	4
2. Scout.java.....	5
3. Mote.java and MoteDatabase.java.....	5
4. Util.java.....	6
5. ConsolePanel.java.....	6
6. MapPanel.java.....	6
7. ChartPanel.java & ReadingRecordList.java.....	7
8. RequestPanel.java and MsgSender.java.....	7
9. LegendPanel.java & Logger.java.....	8
IV. The Embedded application.....	8
1. OctopusApp.nc.....	8
2. OctopusC.nc.....	8
3. Octopus.h.....	9
4. OctopusConfig.h.....	9
V. How-to Section.....	10
1. How to add a stack in the message collected.....	10
2. How to add a new protocol to Octopus.....	12
3. How-to change the number of sensors per mote.....	13

I. Global Structure

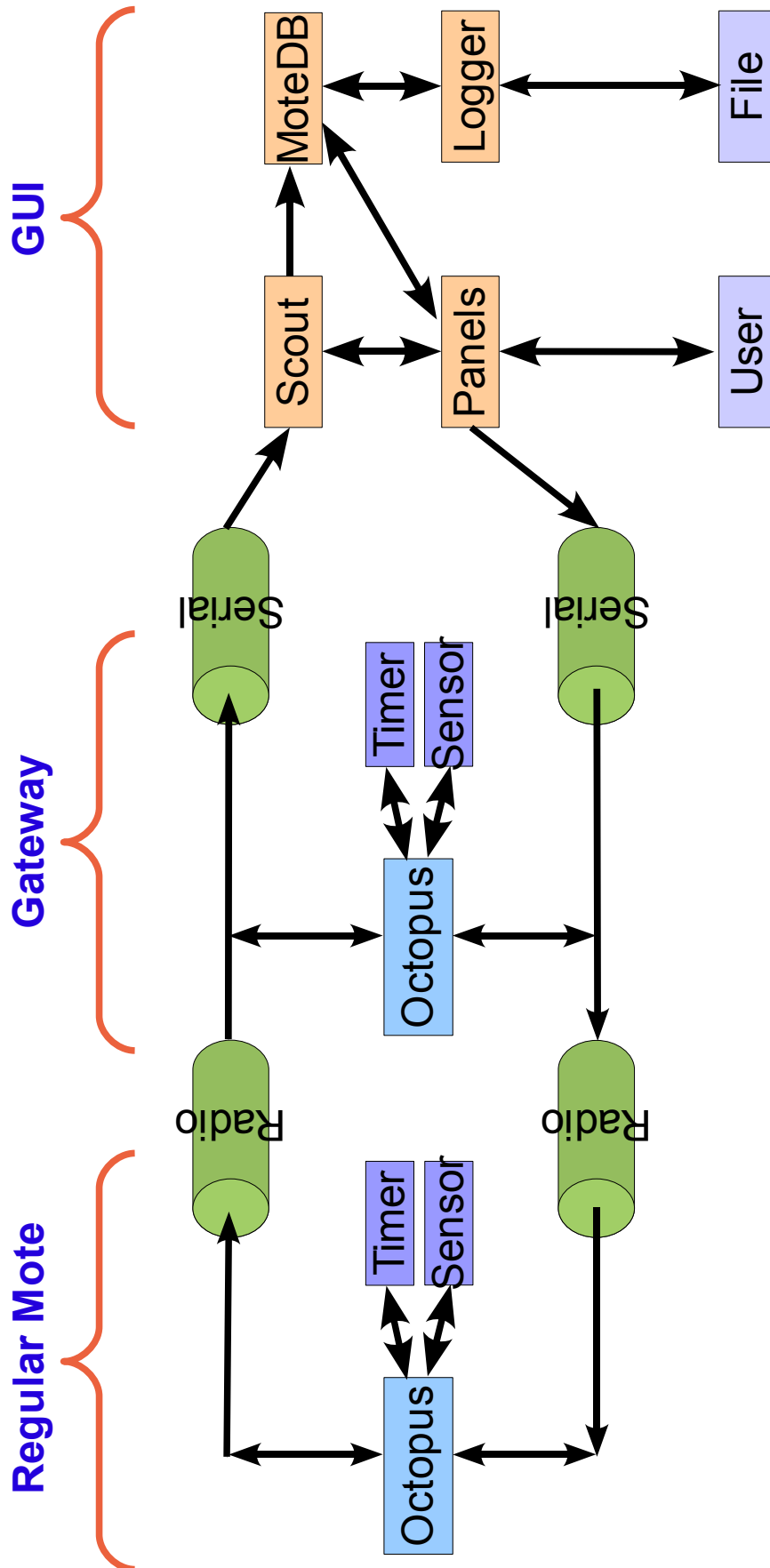


Illustration 1: Global structure of Octopus

As shown on the illustration above, three components appear : the GUI connected to the gateway through a serial connection and the gateway connected to the regular motes through a radio connection.

The gateway and the regular motes are built the same way, with Octopus using the radio/serial connection to communicate and the timer/sensor to process and send data.

The GUI is composed of many blocks. Scout is a thread that is just listening the serial port and processing the messages received. Then the database of motes can be updated. This database is used by every block that needs to access to the status of a mote. The block logger for example is used to store the data collected in a file for future treatment and uses this database, before writing data in a file. Finally many panels are used to display the date for the user, and also to get input from the user, through the mouse events. Those panels are also used to send requests to the serial port.

II. Communication Protocol

Two messages are designed for the communication between the GUI and the gateway and between the gateway and the regular motes. Those protocols are located in an upper layer of TinyOS 2.0, so the same structure is used for the serial and the radio connection. Both messages are defined in "motes/Octopus.h"

The first message is designed for the communication from the GUI to the gateway, and from the gateway to the regular motes. It is a message for the request of the user.

The message is composed of the targetId (16 bits) because the message will be broadcasted, the request (8 bits) which is a constant defined in Octopus.h and the parameters (16 bits).

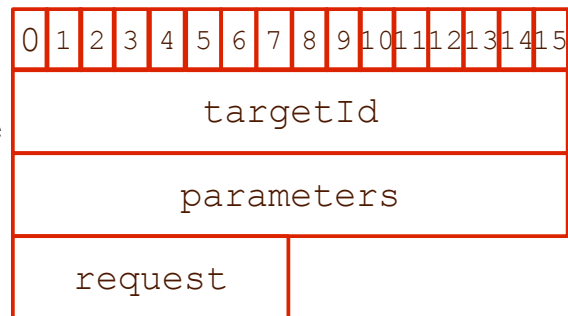
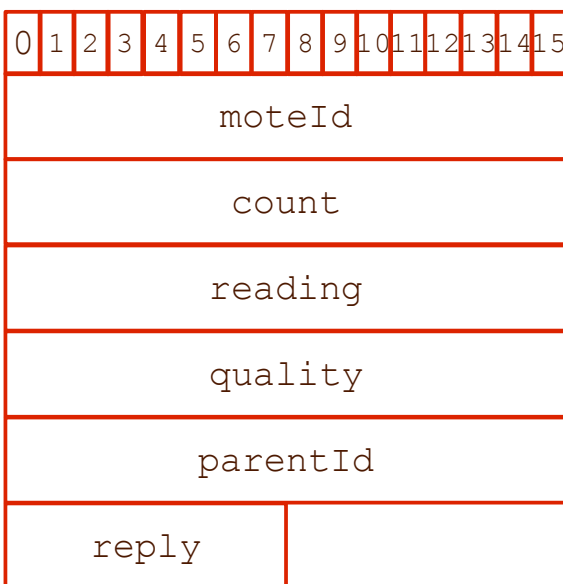


Illustration 2: Structure of OctopusSentMsg



The second message is designed for the communication from the regular mote to the gateway, and from the gateway to the GUI. It is a message for sending back the reading of sensors, and for the replies to requests.

The message is composed of the moteId of the emitter of the message, because the message can be forwarded by many motes before arrive to the gateway. The count is a kind of sequence number, incremented at each reading of the sensors. The reading is the value read from the sensors, or the value of the reply if the reply field is not empty. The quality and the parentId are related to the connection of the mote which has emitted the packet, and its parent. Finally the reply field is used to say if the packet is a reply or just a reading, and the type of reply the packet is.

Illustration 3: Structure of OctopusCollectedMsg

So we finally have the structure below for the communication protocol :

Type (8 bits)	Parameters (16 bits)	Reply (8 bits)	Reading (16 bits)
SET_MODE_AUTO_REQUEST	none	none	none
SET_MODE_QUERY_REQUEST	none	none	none
SET_PERIOD_REQUEST	period (16 bits)	none	none
SET_THRESHOLD_REQUEST	threshold (16 bits)	none	none
GET_STATUS_REQUEST	none	mode (1 bit)	battery (16 bits)
GET_PERIOD_REQUEST	none	none	period
GET_THRESHOLD_REQUEST	none	none	threshold
GET_READING_REQUEST	none	none	reading
SLEEP_REQUEST	none	none	none
WAKE_UP_REQUEST	none	none	none
GET_SLEEP_DUTY_CYCLE_REQUEST	none	none	sleepDutyCycle
GET_AWAKE_DUTY_CYCLE_REQUEST	none	none	awakeDutyCycle
SET_SLEEP_DUTY_CYCLE_REQUEST	sleepDutyCycle	none	none
SET_AWAKE_DUTY_CYCLE_REQUEST	awakeDutyCycle	none	none

Illustration 4: Structure of the communication protocol

III. The GUI

1. Makefile

The Makefile is just used to let the user launch the command "make" in a console, then the instructions of compiling and linking are done through this file.

Some useful constants are firstly declared :

```
GEN=Constants.java OctopusSentMsg.java OctopusCollectedMsg.java

OCTOPUS_H=../nodes/Octopus.h
OCTOPUS_CONFIG_H=../nodes/OctopusConfig.h
```

Next the standart syntax "name_of_the_target: file_and_options" is used. The target "all" is the target built by invoking either "make all" or just "make".

The java files "Octopus*Msg.java" are created to let an access to the structure of messages created in NesC language, through the tool "mig". The documentation of mig, available with "man mig" can be useful if a modification is needed.

The tool "ncc" is used to translate the constants defined in NesC language to constants defined in Java language.

```

all: octopus.jar

octopus.jar: OctopusGui.class
    jar cf $@ *.class

OctopusCollectedMsg.java: $(OCTOPUS_H)
    mig -target=null -java-classname=OctopusCollectedMsg java $(OCTOPUS_H)
octopus_collected_msg -o $@

OctopusSentMsg.java: $(OCTOPUS_H)
    mig -target=null -java-classname=OctopusSentMsg java $(OCTOPUS_H)
octopus_sent_msg -o $@

Constants.java: $(OCTOPUS_H)
    ncg -target=null -java-classname=Constants java $(OCTOPUS_H) Octopus.h
$(OCTOPUS_CONFIG_H) OctopusConfig.h -o $@

OctopusGui.class: $(wildcard *.java) $(GEN)
    javac *.java

clean:
    rm -f *.class $(GEN)

veryclean: clean
    rm octopus.jar

```

2. Scout.java

This class implements two interfaces, *Runnable* and *MessageListener*. It means that this class is a thread and this class can also be registered as a messageListener.

The main method is *run()* that is called when the thread is started. The gateway is registered as the source of messages, and then an infinite loop is started. The thread will then sleep and regularly ask for repaint of some panels.

The other method *messageReceived()* is called when a message is received on the serial port. Then the message is processed. If the mote which has emitted the message is ever in the database, the database is updated, else a new entry is added to the database. If the message is a reply to a previous request, the database is updated with this reply.

In order to avoid any concurrency issue, a mutex is used to get writing access to the database.

3. Mote.java and MoteDatabase.java

MoteDatabase is a class that uses a linked list to store instances of *Mote* objects. *MoteDatabase* owns methods to add, remove or get a *Mote*. If major modifications are made on the mote, or the database, the mutex should be asked to avoid any concurrency issue. *getMutex()* and *releaseMutex()* are the two methods to use. *getMutex()* returns true if the mutex is acquired and false else, without waiting.

Mote is an object standing for a virtual mote with all the characteristics related to the mote (*moteId*, *reading*, *count* etc) but also the characteristics related to its parent (*parentId*, *quality*, *lastTimeSeen*).

4. Util.java

This class is used only to store global constants that can be useful for the developer wishing to do some minor modifications of Octopus.

5. ConsolePanel.java

This class is used to warn the user of special events happening, like when a mote is added to the database, or updated, or when a message is received or sent.

To add a message, the *append()* method should be used. This method takes in parameter the message and the type of message. The message is then added with a header to the consolePanel.

6. MapPanel.java

This class is used to display a map of the network. The main method is *paint()* that is called by Java each time the frame needs to be repainted, for example when Octopus is minimized and then maximized. Firstly a white background is painted with a black border, and then some tests are done to see what must be displayed. The string objects **Legend* are used to define the kind of display that is done. Many methods are called, depending of the legend used.

drawCircleMote() is used to draw a mote with a circle filled of blue if the mote is a regular mote, and filled of red if the mote is the gateway. If the mote is selected (if the user has clicked on the mote) then a black border is also drawn.

drawMoteText() is used to draw some text near from a mote. This method parses the text displayed in many lines, using the special character "\n" as a separator.

drawParentRoute() is used to draw the route between a mote and its parent. The quality is represented by the color of the route, and the *lastTimeSeen* is represented by the transparency of the route.

drawParentRouteLabel() is used to draw some information about the route between a mote and its parent. This label appears in blue.

When a mote is created, its location (x and y) is chosen randomly. In order to translate these values for the screen, the methods *toVirtualX()* and *toVirtualY()* can be used. To get real values from the screen values, the methods *toRealX()* and *toRealY()* can be used.

In order to get the mouse events from the user and move the motes, the Java libraries are used. A *MouseListener* and a *mouseMotionListener* are then used. They will call the functions *mouseClicked()*, *mouseReleased()*, *mousePressed()*, and *mouseDragged()*. When the mouse is clicked, a loop is run through to check if the mouse is clicked over a mote. If so, the boolean *moteClicked* is set to true and the mote is added or removed of the selected mote list, depending of its former state. If the *mouseDragged()* method is called, it means the mouse has been clicked and the user is moving the mouse, keeping the button pressed. Then we get the new values of the mouse pointer and set these values to the mote before ask a repaint of the panel.

7. *ChartPanel.java* & *ReadingRecordList.java*

This class is used to display the reading value of the motes. A *Date* object birth is stored to keep track of the oldest *ReadingRecordList*.

ReadingRecordList is a class extending the *LinkedList* class. An object is instanced for each mote that is displayed on the chart. This class stores *ReadingRecord* objects. Each one of these objects is composed of time and reading. The time for every one of those classes is recorded in seconds.

In *ChartPanel*, the main method is *paint()* that is called by Java. This method shouldn't be called so much because all the panel is repainted, and all the records are processed and drawn.

So the method *paintUpdate()* can be used to run through the list of records, and display only the last one by calling *drawEndLine()*. *drawLine()* and *drawEndLine()* are basically the same functions but the first one process all the record, whereas the other just draws the last values.

To translate the values from the motes (time and sensor values) to values of the screen, the methods *timeToVirtualX()* and *toVirtualY()* should be used.

In order to compress the chart when the time is big, the method *getCurrentTime()* must be used to get the biggest time between the moment when a first mote was added to the *chartPanel* and the last reading received for each mote. This value in seconds is used to display the chart with three different scales: 100 sec, 20 minutes and 2 hours.

Finally to add or update a mote to the chart, the methods *addMote()*, *addRecord()* and *deleteMote()* should be used.

8. *RequestPanel.java* and *MsgSender.java*

This panel is used to send requests from the user to the network. The method *actionPerformed()* is called when a button is pressed. Because the network is big, we assume that the request sent is well received and processed. So each time a request is sent, the corresponding mote is modified. For example if the request is "set samplingPeriod=2000ms for moteID=4", then the mote whose the ID is 4 in the database will also get its samplingPeriod=2000.

If the broadcast checkbox is checked, all the motes of the database are updated, and a message with broadcast adress (0xFF) is sent.

To send many messages, the class *MsgSender* should be used. This class is a thread, that process a queue of messages. To add a message, you should use the method *add()* that takes in parameter the targetId, the request, the parameters, and the message to send to the consolePanel.

This thread sends a message every *SENDER_WAIT_TIME* ms. So in the case of the dissemination protocol, in a medium or big network, the time to disseminate a request and be sure this request is processed is important, and if two requests are sent in a small amount of time, the second one will wipe out the first one. So this value *SENDER_WAIT_TIME* should be set according with the protocol used.

The other main method of *RequestPanel* is *stateChanged()* that is called when a slider is moved by the user. When the user has released the slider, the value is translated according to the slider, and then the request is sent to the *MsgSender* class.

When one or many motes are selected by the user, the method *displayMoteState()* is used to set the good state for each button (enabled or disabled) and eventually change the labels. There is also a linked list used to store the list of motes selected by the user. The methods *selectMote()*,

unselectMote(), *unselectMote()* and *getSelectedMotesListIterator()* should be used to manage this list.

9. LegendPanel.java & Logger.java

This panel lets the user choose the way the map is drawn, and if the data must be recorded in a data file. So the panel is basically getting user events and sending them to the *mapPanel* or to the *Logger*.

The *Logger* is a class that automatically records data in csv format in a file. This class uses a library csv (cf javacsv2.0.zip). To log a file the *startLogging()* method should be called in first, next the *addRecord()* method should be called with in parameter the mote used. Some flags are available to specify the fields that will be recorded. This flags are set or unset by the *LegendPanel*.

IV.The Embedded application

The embedded application is built based on four files. Two files of configuration, one file for the wiring and one for the application itself.

1. OctopusApp.nc

This file implements the wiring between the components of *Octopus*. The main component is *OctopusC.nc* and the other ones are for example *LedsC* or *TimerMilliC*.

Some tests are done through *#if defined(PLATFORM_*)* to check that the correct component is wired for the radio. The choice between the protocols used is also done through these tests. A new protocol should try to implement the maximum of features similar to the Collection and Dissemination protocols, even if the interfaces or useless. Else, some modifications will be needed to get Octopus working.

2. OctopusC.nc

This file is the core of *Octopus*. The interfaces are firstly defined, and next the methods. The first method called is *Boot.booted()*. This method is *Octopus* called when Tinyos 2.0 has finished its boot process. Then the radio is started and the variables are initialized to their default value. When the radio start process is done, *RadioControl.startDone()* is called and if the moteId is 0, then the method *RootControl.setRoot()* is called.

The method *processRequest()* is used to process a request received by the gateway or a regular mote. The request is only processed if the targetId is the one of the mote, or the broadcast adress (0xFF).

The method *SerialReceive.receive()* is called when a message is received by the gateway through its serial port. Then the message is sent to the network using the protocol available. Finally the message is processed.

The tasks *collectSendTask()* and *serialSendTask()* are used to send a packet to the parent of the

current mote, or to the Gui for the gateway. Some flags are used to get a status of each channel with the methods *CollectSend.sendDone()* and *SerialSend.sendDone()*.

The method *fillPacket()* is used to fill the general fields of a packet (quality and parentId).

The method *RequestValue.changed()* for the *Dissemination* protocol or *BroadcastReceive.receive()* for another one protocol, is used to notice *Octopus* that a request has been received and must be processed.

Timer.fired() is the event from the Timer and is used to start a reading if the mote is not sleeping neither in query mode. To start a reading the method *Read.read()* is used and we get the result with *Read.readDone()*. Then the packet is built and sent with the *collectSendTask()* or *serialSendTask()*.

CollectReceive.receive() is only used by the gateway which receive collected messages. Those messages are forwarded to the serial port.

Snoop.receive() is used when a mote has to forward messages intended for the gateway.

3. Octopus.h

In this file are located the constants needed for the requests, the mode etc. Because those constants can be used in *OctopusConfig.h*, they should be defined before and next the file *OctopusConfig.h* should be included. Finally the structures of the messages are defined.

4. OctopusConfig.h

In this file are located the constants that the user should get access to. See the comments in the file for more detail about the constants.

V.How-to Section

1. How to add a stack in the message collected

In a big network, the tracing can lead to an expensive use of the radio and the battery !

To debug efficiently the way a message is collected, it is possible to change the message and add a stack instead of the both fields *parentId* and *quality*.

The first step is to define a structure in *Octopus.h* :

```
typedef nx_struct watched_route_ {
    nx_am_addr_t parentId; /* mote Id of parent */
    nx_uint16_t quality; /* quality of the route */
} watched_route_t;
```

Next this structure can be used to replace the fields *parentId* and *quality*, a new field appears also, *nbHops*, indicating the number of hops. If the mote is connected directly to the gateway this value is one, else it's one plus the number of motes to reach the gateway.

```
typedef nx_struct octopus_collected_msg {
    nx_am_addr_t moteId; /* Mote id of sending mote. */
    nx_uint16_t count; /* sequence number */
    nx_uint16_t reading; /* sensor value */
    nx_uint8_t nbHops; /* number of hops traced */
    watched_route_t watchedRoute[MAX_TRACED_HOPS]; /* stack of hops */
    nx_uint8_t reply;
} octopus_collected_msg_t;
```

Next the embedded code and the Java code both need to be updated. The embedded application must add its *parentId* and *quality* to the stack, and update the *nbHops* field in the OctopusC.nc file. So two functions must be modified, *fillPacket* and *Snoop.receive* which is called when a message is forwarded is used.

The function *fillPacket* initializes a packet :

```
void fillPacket() {
    uint16_t tmp;
#ifdef COLLECTION_PROTOCOL
    call CollectInfo.getEtx(&tmp);
#elif defined(DUMMY_COLLECT_PROTOCOL)
    call CollectInfo.getQuality(&tmp);
#endif
    localCollectedMsg.nbHops = 1;
    localCollectedMsg.watchedRoute[0].quality = tmp;
    call CollectInfo.getParent(&tmp);
    localCollectedMsg.watchedRoute[0].parentId = tmp;
}
```

and the function *Snoop.receive* updates this packet :

```
event message_t* Snoop.receive(message_t* msg, void* payload, uint8_t len) {
```

```

uint16_t tmp;
// we get a pointer on the payload
octopus_collected_msg_t *fwdCollectedMsg = payload;
if (len == sizeof(*fwdCollectedMsg)) {
    if(fwdCollectedMsg->nbHops == 0)
        reportProblem();// bad initialization of the packet
    else if(fwdCollectedMsg->nbHops < MAX_TRACED_HOPS){
        // stack not full
#ifdef COLLECTION_PROTOCOL
        call CollectInfo.getEtx(&tmp);
#elif defined(DUMMY_COLLECT_PROTOCOL)
        call CollectInfo.getQuality(&tmp);
#endif
    fwdCollectedMsg->watchedRoute[fwdCollectedMsg-
>nbHops].quality = tmp;
    call CollectInfo.getParent(&tmp);
    fwdCollectedMsg->watchedRoute[fwdCollectedMsg-
>nbHops].parentId = tmp;
    fwdCollectedMsg->nbHops++;
    }
}
return msg;
}

```

Finally the java application just needs to read the packet differently. The thread *Scout* is still in charge of this work, with the function *messageReceived*. When a packet is received, the bottom of the stack is read to fill the fields of the mote that has emitted the packet :

```

localMote.setParentId(collectedMsg.getElement_watchedRoute_parentId(0));
localMote.setQuality(collectedMsg.getElement_watchedRoute_quality(0));

```

And next, a loop is done to read the stack, and update each mote concerned :

```

for (int i=1; i<collectedMsg.get_nbHops(); i++) {
    int parentId = collectedMsg.getElement_watchedRoute_parentId(i-1);
    if (parentId==0)
        break;
    Mote parentMote = moteDatabase.getMote(parentId);
    date = new Date();
    parentMote.setLastTimeSeen(date.getTime());
    parentMote.setParentId(
        collectedMsg.getElement_watchedRoute_parentId(i));
    parentMote.setQuality(
        collectedMsg.getElement_watchedRoute_quality(i));
}

```

2. How to add a new protocol to Octopus

Two kind of protocols can be added to *Octopus*, broadcast protocol or collection protocols. Because Octopus uses extensively the interfaces of the *Dissemination* and *Collection* protocols, some minor changes could be necessary to get *Octopus* working with another protocol. The best way would be to define firstly the differences between this new protocol and *Dissemination/Collection* protocol to see if new interfaces or needed, if the name or the type of some interfaces change, and if some interfaces or useless. Then by using the structure :

```
#if defined(DISSEMINATION_PROTOCOL)
    // ...
#elif defined(DUMMY_BROADCAST_PROTOCOL)
    // //
#else
#error "A protocol needs to be selected to broadcast data"
#endif
```

a new protocol can be added. If some parts are similar to one of the supported protocol, then this syntax can be used :

```
#if defined(DISSEMINATION_PROTOCOL) || defined(SIMILAR_PROTOCOL)
```

To see how are working both fully supported protocols, check :

<http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html> (Collection)

<http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html> (Collection Tree Protocol)

<http://www.tinyos.net/tinyos-2.x/doc/html/tep118.html> (Dissemination)

3. How-to change the number of sensors per mote

If many sensors are available on a mote, then the best way would be to change firstly the protocol to replace the reading field by an array of readings.

In the embedded application, when the timer is fired, the first one of the sensors could be read, and next the second one. So when the stack of sensors is fully read, the message can be sent.

Another simpler idea would be to keep the same communication protocol and just add in the reply field some constants, instead of *NO_REPLY* that is used when a reading is sent. So the constants could be *READING_SENSOR1* or *READING_LIGHT* etc.

Finally the Gui would need to be updated. In the Scout.java file, where the received packet will be processed, the constants should be changed according to above. The request panel should also be changed to get many buttons for each sensor, or maybe a Jchooser to gain some space.