Using Application Motes with Cooja

Kerry Veenstra

May 22, 2021

Contents

1 Introduction

Cooja[1] lets one simulate "application level" motes whose operation is described by pure Java code. Using an Application mote avoids memory limitations of ordinary motes that run emulated MSP430 binaries. In this document I explain how to create an Application mote from the AbstractApplicationMote base class.

2 Create a Mote Subclass

Create a subclass of AbstractApplicationMote. For this example our subclass is called AppMoteMobile, and we create it in a file `AppMoteMobile.java`.

Listing 1: AppMoteMobile subclass

```java
public class AppMoteMobile extends AbstractApplicationMote {
  /* constants and variables */
  /* constructors */
  /* execute() method */
}
```

The rest of Section 2 describes other parts of the subclass.

2.1 Constants and Variables

2.1.1 Required Object References

The subclass requires simulation and radio object references. These references will be initialized elsewhere.

Listing 2: Required object references

```java
  private Simulation simulation = null;
  private ApplicationRadio radio = null;
```

2.1.2 Optional Object References

For our example, we also provide references to a random-number generator and LEDs. As with the required object references, these references will be initialized elsewhere.

Listing 3: Optional object references

```java
private Random random = null;
private ApplicationLED = null;
```

2.1.3 Local Information

Create variables to store local information. In our example the mote stores location information that it has received from another mote, but any other information could be stored. Note that the value of each variable is initialized.

Listing 4: Local information

```java
private boolean receivedNeighborPosition = false;
private double neighborX = 0.0;
private double neighborY = 0.0;
private double neighborZ = 0.0;
```

2.1.4 Events

Events are scheduled through MoteTimeEvent objects. An event can reschedule itself (making it periodic) or it can merely fire once and then wait for other code to reschedule it. Each MoteTimeEvent object has an `execute` method which is called when the event fires.

Periodic Events Create a series of periodic events with a MoteTimeEvent object whose `execute` method ends by rescheduling itself. Here's an example of an event that moves the mote every second. You also must include additional code in the class's `execute` method which initiates the event series. See Section 2.2.

Listing 5: Example periodic event

```java
private MoteTimeEvent moveEvent = new MoteTimeEvent(this, 0) {
  public void execute(long t) {
    /* Add code here to move the mote. */

    /* Reschedule this event 1 second later. */
    simulation.scheduleEvent(
      moveEvent,
      simulation.getSimulationTime() + 1000 * Simulation.MILLISECOND
    );
  }
}
```

While this example shows an unvarying event rate of one second, the code for the event can vary the delay to the next event in any way that it desires.

Scheduled Events The MoteTimeEvent objects for non-periodic events still must be created, even though they are scheduled by other code. For example, here is a `transmitEvent` object which is scheduled whenever the mote needs to transmit. The event is scheduled in the *mote's* `execute` method (not the MoteTimeEvent's `execute` method). See Section 2.2.

Listing 6: Example scheduled event

```
private MoteTimeEvent transmitEvent = new MoteTimeEvent(this, 0) {
  public void execute(long t) {
    /* Transmit Rime location radio packet.
     * sentPacket is called when transmission finishes.
     * Format is like
     *
     *    RIME_CHANNEL1, RIME_CHANNEL2, 'A', 'B', 'C', '\0'
     *
     * where 'A', 'B', 'C' is the packet payload.
     */

    Position pos = getInterfaces().getPosition();

    double x = pos.getXCoordinate();
    double y = pos.getYCoordinate();
    double z = pos.getZCoordinate();

    byte[] packetBytes = payloadOfPosition(x, y, z);
    RadioPacket locationPacket = new COOJARadioPacket(packetBytes);
    radio.startTransmittingPacket(
        locationPacket, 1*Simulation.MILLISECOND);

    /* Turn on red LED. */
    leds.setLED(ApplicationLED.LEDS_RED);
  }
}
```

2.2 `execute` method

The mote has its own `execute` method which is called whenever the mote wakes up. The first time the `execute` method is called, it initializes all of the mote's object references and starts any periodic events. In this example, a null value of the radio reference causes initialization. After initialization, radio will no longer be null.

Listing 7: Mote's execute method

```
public void execute(long time) {
  /* First call? */
  if (radio == null) {
    /* Initialize references. */
    simulation = getSimulation();
    random = simulation.getRandomGenerator();
    radio = (ApplicationRadio) getInterfaces().getRadio();
    leds = (ApplicationLED) getInterfaces().getLED();

    /* Start periodic events. */
    simulation.scheduleEvent(
      moveEvent,
      simulation.getSimulationTime() +
        1000 * Simulation.MILLISECOND
    );
  }

  /* Schedule an event: Transmit a packet after 2-4 seconds. */
  simulation.scheduleEvent(
    transmitEvent,
    simulation.getSimulationTime() +
      (2000 + random.nextInt(2000))*Simulation.MILLISECOND
  );
}
```

2.3 Constructors

You must provide a two-argument constructor which invokes the two-argument superclass constructor. If the subclass needs any initialization code, it goes after the line that invokes the superclass's constructor.

Listing 8: Constructor

```
  public AppMoteMobile(MoteType moteType, Simulation simulation) {
    super(moteType, simulation);
    /* initialization code goes here */
  }
```

3 Mote-to-Mote Communication

For mote-to-mote communication you will provide a packet-payload class and define a pair of methods for transmitting and receiving packets.

3.1 Packet Payloads

Packet payloads are text. Create a payload class that handles the conversion between the text payload and a set of Java data variables. For example, here is class MobileMotePacketPayload. It assumes that there already is a class PacketFormatException. The rest of this section describes the class's contents.

Listing 9: Packet payload class

```
package org.contikios.cooja;

import java.util.Arrays;
import org.contikios.cooja.PacketFormatException;

public class MobileMotePacketPayload {
  /* Required Constants */
  /* Variables for packet fields */
  /* Constructors */
  /* Field Accessors */
}
```

RIME packets have a two-byte prefix. Define their values.

Listing 10: Example packet payload class: required constants

```java
/* Required Constants */
private static final byte RIME_CHANNEL1 = -128;
private static final byte RIME_CHANNEL2 = 0;
```

Each of the packet fields is represented by data variable.

Listing 11: Example packet payload class: variables for packet fields

```java
/* Variables for packet fields */
private int id;
private double x;
private double y;
private double z;
```

3.1.1 Encoding a Packet for Transmission

Create a constructor that accepts values for each of the packet payload's fields. Also provide a corresponding `getBytes` accessor method that returns a `byte` array of the packet payload that represents the fields' values.

Listing 12: Example packet payload class: encoding a packet

```java
/* Constructor */
public MobileMotePacketPayload(
  int id, double x, double y, double z) {

  this.id = id;
  this.x = x;
  this.y = y;
  this.z = z;
}

/* Payload accessor */
public byte[] getBytes() {
  String data =
    "" + this.id +   /* initial "" makes + string concatenation */
    "," + this.x +
    "," + this.y +
    "," + this.z;

  byte[] dataBytes = data.getBytes;
  byte[] payload = new byte[2 + dataBytes.length + 1];
  payload[0] = -128;
  payload[1] = 0;
```

```
    System.arraycopy(dataBytes, 0, payload, 2, dataBytes.length));
    payload[payload.length - 1] = 0;
    return payload;
  }
```

3.1.2 Decoding a Received Packet

Create a constructor that accepts an existing packet payload as a `byte` array. Also provide a set of accessor methods, one for each of the payload's fields.

In this example, the constructor parses the payload's `byte` array through manual coding (rather than using a table-driven parsing algorithm). Any format error is reported through a `PacketFormatException`.

In this example, the accessor methods are `getID`, `getX`, `getY`, and `getZ`.

Listing 13: Example packet payload class: decoding a packet (part 1 of 3)

```
  /* Constructor */
  public MobileMotePacketPayload(byte[] b)
    throws PacketFormatException {

    /* Packet format:                   */
    /*                                  */
    /* b[0] == -128                     */
    /* b[1] == 0                        */
    /* b[2 .. b.length - 2] == payload */
    /* b[b.length - 1] == 0x00          */

    try {
      /* Check the packet format. */
      if (b.length < 3) {
        throw new PacketFormatException("too short");
      }
      if (b[0] != -128) {
        throw new PacketFormatException("wrong RIME_CHANNEL1");
      }
      if (b[1] != 0) {
        throw new PacketFormatException("wrong RIME_CHANNEL2");
      }
      if (b[b.length - 1] != 0) {
        throw new PacketFormatException("missing 0x00 end byte");
      }
      ...
```

At this point we know that `b` starts with a valid RIME prefix and ends with `0x00` suffix. The remainder of the constructor extracts comma-separated fields from the packet payload.

Listing 14: Example packet payload: decoding a packet (part 2 of 3)

```
      ...
      /* Payload starts at b[2] and ends with b[b.length - 2]. */
      String s = new String(Arrays.copyOfRange(b, 2, b.length - 1));

      /* Find the field separators. */
      int comma1 = s.indexOf((byte) ',');/Pa
      int comma2 = s.indexOf((byte) ',', comma1 + 1);
      int comma3 = s.indexOf((byte) ',', comma2 + 1);
      int comma4 = s.indexOf((byte) ',', comma3 + 1);
      if (comma1 == -1 || comma2 == -1 || comma3 == -1 ||
          comma4 != -1) {    /* use != to check for too many fields */
        throw new PacketFormatException(
          "wrong number of fields in '" +
          s +
          "'");
      }

      /* Extract the data fields. */
      this.id = Integer.parseInt(s.substring(0, comma1));
      this.x = Double.parseDouble(s.substring(comma1 + 1, comma2));
      this.y = Double.parseDouble(s.substring(comma2 + 1, comma2));
      this.z = Double.parseDouble(s.substring(comma3 + 1));
    } catch (NumberFormatException e) {
        throw new PacketFormatException(
          "bad number format in '" +
          new String(Arrays.copyOfRange(b, 2, b.length)) +
          "'");
    }
  }
```

Provide an accessor method for each field.

Listing 15: Example packet payload: decoding a packet (part 3 of 3)

```
  /* Field Accessors */
  public int getID() {
    return this.id;
  }

  public double getX() {
    return this.x;
  }

  public double getY() {
    return this.y;
  }

  public double getZ() {
    return this.z;
  }
```

3.2 Mote Position

Your code can get its XYZ position through a Position object. This use already appears in the listing for `transmitEvent` in the **Scheduled Events** paragraph of Section 2.1.4.

Listing 16: Getting mote's position

```
      Position pos = getInterfaces().getPosition();

      double x = pos.getXCoordinate();
      double y = pos.getYCoordinate();
      double z = pos.getZCoordinate();
```

Your mote can set its own coordinates using this code. Be aware that for motes that reside in 3D, the computation of `newZ` must be done properly (keeping the mote on the ground, over the ground, or wherever it's supposed to be).

Listing 17: Setting mote's position

```
      Position pos = getInterfaces().getPosition();

      double newX = ...;
      double newY = ...;
      double newZ = ...;

      pos.setCoordinates(newX, newY, newZ);
```

3.3 Packet Transmission/Reception Methods

Provide a `sendPacket` method and a `receivedPacket` method. Examples are below.

Listing 18: sentPacket method

```java
public void sentPacket(RadioPacket p) {
  byte[] packetData = p.getPacketData();
  requestImmediateWakeup(); /* triggers execute method */
}
```

Listing 19: receivedPacket method

```java
public void receivedPacket(RadioPacket p) {
  byte[] packetData = p.getPacketData();

  try {
    MobilePacketPayload payload =
      new MobilePacketPayload(packetData);

    /* store packet data */
    neighborX = payload.getX();
    /* etc. */
  } catch (PacketFormatException e){
    log("received packet error");
    return;
  }
}
```

References

[1] F. Österlind. A sensor network simulator for the contiki os. Technical Report T2006:05, Swedish Institute of Computer Science, 2006.