Modeling PicoRadio in Metropolis

Pete Perlegos

Abstract

The basic goal of this project is to implement the PicoRadio protocol using the Metropolis meta-model. I have used the application that consists of a set of sensors (S), controllers (C), and actuators (A). The interaction between S and C has two scenarios. The first is the Pull scenario in which controllers request data to a set of sensors upon requests from an external user. The second is Push in which sensors send data to controllers periodically. The interaction between C and A has only one scenario in which the controllers issue commands to actuators. At the application layer level, each pair of interacting S, C, and A are connected by a reliable medium of one-hop links. This link is refined at the network layer into a multi-hop connection, where any node can be used as an intermediate hop in a path from source to destination. This multi-hop connection is essential to maximize reuse of resources and reduce power consumption. I have implemented this protocol using Metropolis to specify both the behavior and structure. This includes specifying the behavior of each block and the connections between them at each successive layer. I conclude by explaining the benefits of the Metropolis meta-model.

1. Introduction

Establishing formal design methodologies is needed to manage complex design tasks required in modern system designs.

Levels of abstraction must be defined to formally represent systems being designed. This calls for a design environment in which: systems can be unambiguously represented throughout the abstraction levels, the design problems can be mathematically formulated, and tools can be incorporated to solve some of the problems automatically.

The core of the infrastructure is a meta model of computation, which allows one to model various communication and computation semantics in a uniform way. By defining different communication primitives and different ways of resolving concurrency, the user specifies different models of computation (MOCs).

At a high level of abstraction, the designer may want to use a MOC that is convenient to describe the functionality, but does not reflect any physical resources (Figure 1) [7]. At a lower level of abstraction, a MOC that reflects a chosen implementation platform (or architecture) may be more appropriate (Figure 1).

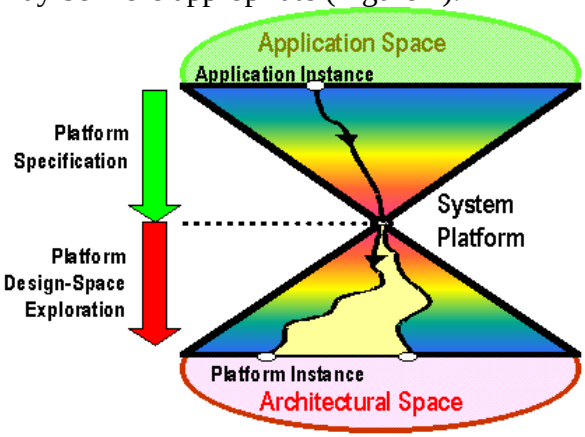

Figure 1: System Platform.

Metropolis provides a unified framework that allows a system to be represented at both of these abstraction levels, as well as in any combination of the two that arises as parts of the design are refined and implemented.

2. PicoRadio Protocol

I will now go over the PicoRadio protocol for a simple application. The application consists of a set of sensors (S), controllers (C), and actuators (A) (Figure 2). The connection is first described at the application layer and is then refined down to the network layer. It can be then refined down to subsequent layers.

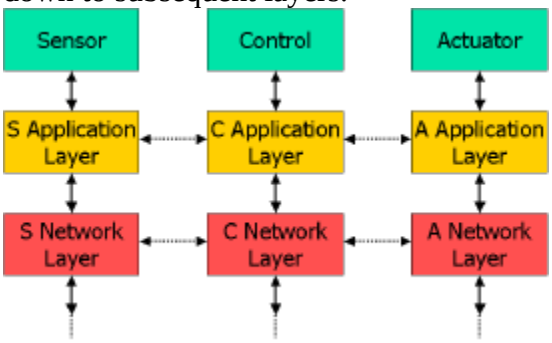

Figure 2: Protocol Stack.

2.1 PicoRadio Application Layer

The interaction between S and C has two scenarios (Figure 3) [9]. The first is the Pull scenario in which controllers request data to a set of sensors upon requests from an external user. The second is Push in which sensors send data to controllers periodically. The interaction between C and A has only one scenario in which the controllers issue commands to actuators.

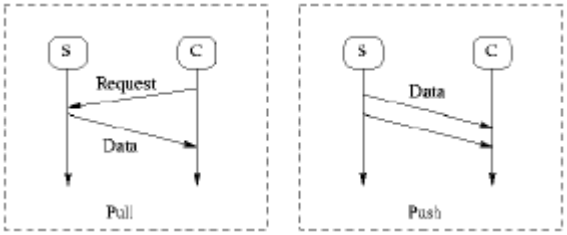

Figure 3: The application layer between the sensors (S) and controllers (C) showing the Pull and Push interactions

At the application layer level, each pair of interacting S, C, and A are connected by a reliable medium of one-hop links (Figure 4) [9].

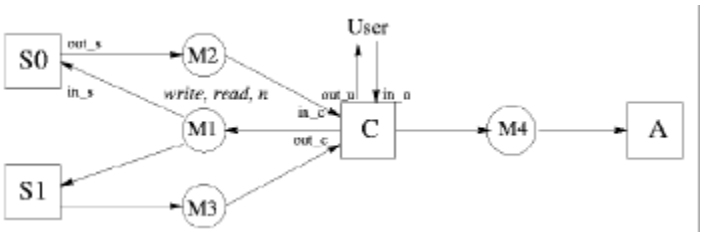

Figure 4: M1 connects a controller and the sensors to which the controller may request data. The controller writes the message requests into the buffer and sensors read the buffer when there is an unread message. Write is non-blocking because it can overwrite equivalent requests. Read is destructive only when all the sensors have read the buffer.

2.2 PicoRadio Network Layer

This link is refined at the network layer into a multi-hop connection, where any node can be used as an intermediate hop in a path from source to destination (Figure 5) [9]. This multi-hop connection is essential to maximize reuse of resources and reduce power consumption.

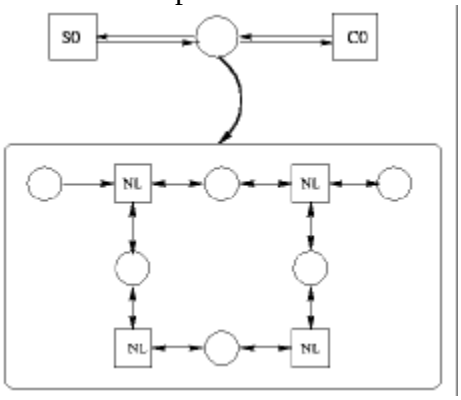

Figure 5: Network layer.

Sending a bit of information through free space directly from node A to node B incurs an energy cost, which is a function of the distance d between the nodes. The energy cost is proportional to d^y (y is generally between 2 to 4 indoors) [6]. Given this greater than linear relationship between energy and distance, using several short hops to send a bit may be more energy efficient than using one large hop (Figure 6).

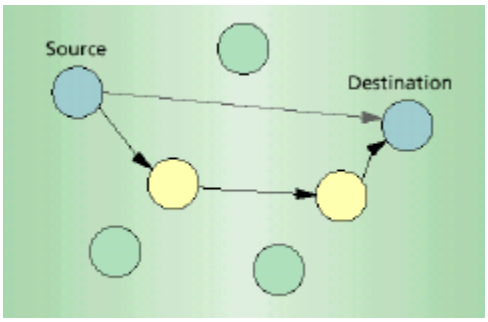

Figure 6: Multi-hop network with several short hops instead of one long hop.

3. Motivation

The motivation of this project was to evaluate a design in Metropolis in various capacities such as the design methodology, whether it is easier to design modules, whether it is less error prone making for easier debugging, and how well one can do many types of analysis.

3.1 Standard Development

Standard development allows you to design conceptually, but it is not described that way (Figure 7).

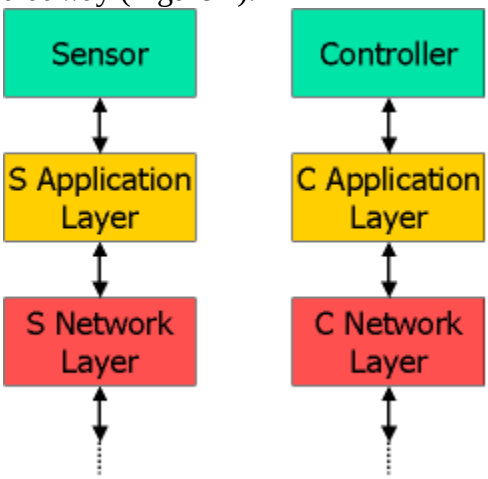

Figure 7: Standard development.

The actual interface between the layers is made as shown in Figure 7. This makes it very difficult to design the functionality that you want at each layer. In this design, it is also difficult to verify if each layer works well. This makes it strenuous to design a system in such a way.

3.2 Metropolis Development

Metropolis allows you to describe each layer well and the functionality that you want at each layer (Figure 8).

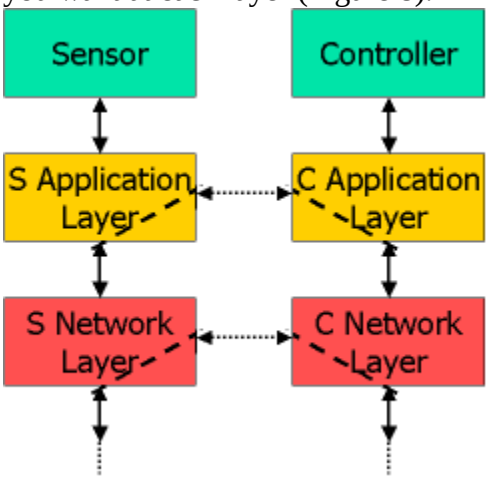

Figure 8: Metropolis development.

The Metropolis design methodology allows you describe the functionality that you want at each layer and then test each layer. If you want to refine a given layer, such as the connection between the application layers, you must maintain that interface in the connection from the application layer to the network layer. In this type of a development, if you want to change the implementation at a given layer, it is much easier to do so and test it.

3.3 Can Do Many Types of Analysis

As you can imagine, the Metropolis development allows us to do many types of analysis. For example, one idea that was discussed was to replace the single unit buffers in the application layer with a larger buffer and explore the repercussions, such as the need for blocking write and blocking read (Figure 9). Metropolis allows us to more easily test such ideas.

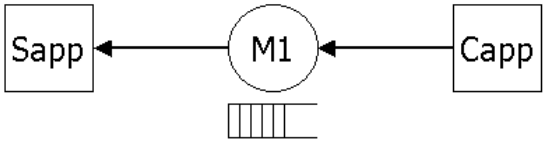

Figure 9: Analysis example.

4. Development

I will now give an overview of the actual development of the PicoRadio protocol in Metropolis.

4.1 Specification

I will first go over some aspects of the coding in Metropolis beginning with the Controller as an example.

```
process Controller {
  port CReader in_c2;
  port CReader in_c3;
  port CWriter out_c2s;
  port AWriter out_c2a;
  port UReader in_u;
  port UWriter out_u;
  parameter int MyID;
  parameter int MyLocationX;
  parameter int MyLocationY;
  parameter int MyLocationZ;

  controller(int id, int x, int y, int z) {
    MyID = id; MyLocationX = x; MyLocationY = y;
MyLocationZ= z;
  }
  void thread() {
    PacketUReq Ureq;
    PacketSensedData SensedValue;

    while (true) {
      await {
        (in_u.n()>0; ; ) { // user request
          Ureq = in_u.read();
          out_c2s.write(MyID, Ureq.dest)
        }
        (in_c2.n()>0; ; ) { // data from the first sensor
          SensedValue = in_c2.read();
          out_u.write(SensedValue);  // return data to user
          if (controlAlgorithm(SensedValue))
            out_c2a.write(command);  // command actuator
        }
        (in_c3.n()>0; ; ) { // data from the second sensor
          SensedValue = in_c3.read();
          out_u.write(SensedValue);  // return data to user
            if (controlAlgorithm(SensedValue))
              out_c2a.write(command);  // command actuator
        }
      }
    }
  }
}
```

A description of the various components involved with this example follows. The first items declared are the ports with which the controller communicates with the outside world. But these are not simply declared as ports but as the specific types of ports that they are, such as CReader which reads from the sensors, CWriter which writes to the sensors, AWriter which writes to the actuator, and UReader and UWriter which communicate with the user. Following the ports are various parameters, which are assigned in the initializing function. The behavior of the controller is given in the main function called “thread”. The controller awaits either a request from the user or for data to arrive from the sensor. The controller knows that something has arrived by checking the n function of a media, which I will go over next. When something arrives, the controller takes the appropriate action (request data from sensors upon request from the user or decide whether to send a command to the actuators upon receipt of data from the sensors).

I will now go over the media, specifically medium M1 and how it interacts with the controller and the sensors. The code for medium M1 is shown below.

```
public medium M1 implements SReader, CWriter {
PacketReq p;
int BufferSize, n[];
parameter int rsize;
  M1(int rn){
    int i;
    BufferSize=1; n=new int[rn]; rsize=rn;
    for(i=0; i<n.length; i++) n[i]=0;
  }
  eval int n(int SensorID) {
    await { (true; this.CWriter; this.SReader) return n[SensorID]; }
  }
  update void write(int ControllerID, LocRange dest) {
    int i;
    await {
      (true; this.SReader; this.CWriter) {
        p.src=ControllerID; p.dest=dest;
        for(i=0; i<n.length; i++) n[i]=1;
      }
    }
  }
  update PacketReq read(int SensorID) {
    await {
      (n[SensorID]>0; this.CWriter; this.SReader) { n[SensorID]=0;
return p;
      }
    }
  }
  constraint {
    ltl G(connectionnum(this, CWriter)==1);
    loc val.A(beg(any, this.n),i).SensorID<rsize;
    loc val.A(beg(any, this.read),i).SensorID<rsize;
  }
}
```

The first line of the code indicates that medium M1 implements SReader and CWriter. This corresponds to the connections(ports) of the controller and sensor that the medium is connected to. The n function, which is called by a process, such as the controller, which is connected to the medium, returns whether there is an item in the medium. There are also the write and read functions which the controller and sensors call on the medium. The this.CWriter and this.SReader statements in the await statements prevent a n, write, or read function from being called while another action is being taken on the medium. This is done because if you tried to read while a write was in progress you would get junk. At the end of the medium specification there is a set of constraints, which in this case specify that that only one CWriter can be attached to the medium and the range of sensors that can be attached.

4.2 Refinement

To achieve a refinement, there are a series of steps that one must carry out. My specific refinement is very simple (Figure 10).

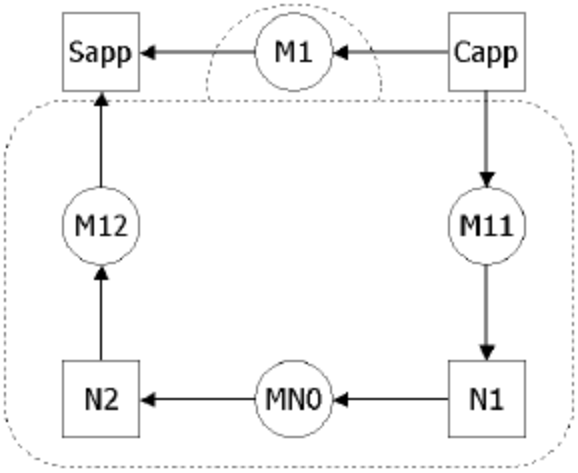

Figure 10: Refinement example.

The example that I will discuss is the refinement of M1 into a netlist at the network layer. As can be seen from Figure 10, there must exist an interface in the form of media between the application and network layers. These media (M11, M12) must maintain the same interface that exists in the application layer. Specifically, the interface between Capp and M1 must be maintained between Capp and M11. Similarly, the interface between Sapp and M1 must be maintained between Sapp and M12.

```
// create objects for Application Layer
netlist ApplicationLayer {
  ApplicationLayer() {
    Controller c1 = new Controller(1, 0, 0, 0);
    Sensor s1 = new Sensor(2, 1, 1, 0, 60);
    M1 m1_1 = new M1(1);

    connect(c1, out_c2s, m1_1);
    connect(s1, in_s, m1_1);
  }
}

// create netlist and objects for refinement
netlist NetworkLayer {
  NetworkLayer() {
    NetworkNode n1 = new NetworkNode(1, 0, 0, 0, 10);
    NetworkNode n2 = new NetworkNode(2, 1, 1, 0, 10);
    MN0 mn0_1 = new MN0();
    M11 m11_1 = new M11(1);
    M12 m12_1 = new M12(1);

    connect(n1, out_n, mn0_1);
    connect(n2, in_n, mn0_1);
    connect(n1, in_n, m11_1);
    connect(n2, out_n, m12_1);
  }
}

//refine M1 into netlist1;
refine(M1, NetworkLayer);
redirectconnect(Controller, M1, M11);
redirectconnect(Sensor, M1, M12);
```

After all of the components have been defined, the first step is to create the objects in the application layer and then connect them. At this point you have a functioning application layer.

To refine M1 into the network layer, there are three steps that must be taken. The first step is to create all the objects for the refinement. The second step is to connect all of the objects appropriately, as shown in Figure 10 for my example. Third, you must refine M1 into the network layer netlist and redirect the connections for each item connected to M1 from M1 to the appropriate interface medium.

5. Simulation

Simulating Metropolis involves the use of the Metropolis simulator, which is still under development. The Metropolis simulator currently handles processes and communication media, but does not handle netlists (so refinement will be a problem).

I attempted to do a simulation [10], but I received numerous errors that I tried to sort out with Rong's help. I got to the point where I attempted to checkout the source files. But when I enter the command "cvs checkout metro" I get the following error:
perlegos@ic.eecs.berkeley.edu's password:
Sorry, you don't have read/write access to the history file
/projects/hwsw/hwsw/common/src/CVSROOT/history
Permission denied
It seems that I do not have permission to access the files. At this stage, we have been trying to figure out what was done in the account setup by the person who set it up for me.

6. Conclusion

Metropolis seems to offer strong advantages. The design methodology allows for more conceptual development; you start by describing the functionality. Simple refinement makes the design methodology feasible. The Metropolis design methodology also allows us to do many types of analysis.

Acknowledgements:
I would like to thank the Marco Sgroi and Rong Chen for their assistance and review.

References:

[1] Metropolis: Design Environment for Heterogeneous Systems. http://www.gigascale.org/metropolis/

[2] PicoRadio. http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/.

[3] J. Rabaey, J. Ammer, J. L. da Silva Jr., D. Patel. PicoRadio: Ad-hoc Wireless Networking of Ubiquitous Low-Energy Sensor/Monitor Nodes, April 2000.

[4] J. L. da Silva Jr., M. Sgroi, F. De Bernardinis, S. F. Li, A. Sangiovanni-Vincentelli, J. Rabaey. Wireless Protocols Design: Challenges and Opportunities, May 2000.

[5] M. Sgroi, J. L. da Silva Jr., F. De Bernardinis, F. Burghardt, A. Sangiovanni-Vincentelli, J. Rabaey. Designing Wireless Protocols: Methodology and Applications, June 2000.

[6] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva Jr., Danny Patel, Shad Roundry. Pico Radio Supports Ad Hoc Ultra-Low Power Wireless Networking, July 2000.

[7] Kurt Keutzer, Sharad Malik, A Richard Newton, Jan M. Rabaey, A. Snagiovanni-Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design.

[8] J. L. da Silva Jr., J. Shamberger, M. J. Ammer, C. Gua, S. Li, R. Shah, T. Tuan, M. Sheets, J. M Rabaey, B. Nikolic, A. Sangiovanni-Vincentelli, P. Wright. Design Methodology for PicoRadio Networks, March 2001.

[9] Marco Sgroi. Modeling PicoRadio in Metropolis.

[10] Claudio Passerone. The Metropolis Java Simulator.