Handbook of Dynamic System Modeling, chapter 35,
Paul Fishwick (edt.), CRC/Taylor and Francis Publ., 2007, pp. 35-1 -- 35-16

Sensor Network Component Based Simulator

Boleslaw K. Szymanski and Gilbert Gang Chen
Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY 12180
{szymansk,cheng3}@cs.rpi.edu

The Need for a New Sensor Network Simulator

The emergence of wireless sensor networks brought many open issues to network designers. Traditionally, the three main techniques for analyzing the performance of wired and wireless networks are *analytical methods*, *computer simulation*, *and physical measurement*. However, because of many constraints imposed on sensor networks, such as energy limitation, decentralized collaboration, fault tolerance, algorithms for sensor networks tend to be quite complex and usually defy analytical methods that have been proved to be fairly effective for traditional networks. Furthermore, few sensor networks have come into existence, for there are still many unsolved research problems, so measurement is virtually impossible. It appears that simulation is the only feasible approach to the quantitative analysis of sensor networks.

Why a New Simulator

A good simulator possesses two essential features. First, it must support reusable models. A model written for one simulation should be able to be effortlessly embedded into other simulations that require the same kind of a model. Second, the model should be easy to be built from scratch. Interestingly, we observe that most existing simulators do not possess these two features simultaneously. Most commercial simulators provide a reusable model library, often coming with a friendly graphical user interface, but adding new models to the library is always a painful task. On the other hand, most freely available simulators follow a bottom-up approach; writing models from scratch is straightforward, but the reusability is severely limited.

ns2 (ns2, 1990), perhaps the most widely used network simulator, has been extended to include some basic facilities to simulate sensor networks. However, one of the problems of ns2 is its object-oriented design that introduces much unnecessary interdependency between modules. Such interdependency sometimes makes the addition of new protocol models extremely difficult, only mastered by those who have intimate familiarity with the simulator. Being difficult to extend is not a major problem for simulators targeted at traditional networks, for there the set of popular protocols is relatively small. For example, Ethernet is widely used for wired LAN, IEEE 802.11 for wireless LAN, TCP for reliable transmission over unreliable media. For sensor networks, however, the situation is quite different. There are no such dominant protocols or algorithms and there will unlikely be any, because a sensor network is often tailored for a particular application with specific features, and it is unlikely that a single algorithm can always be the optimal one under various circumstances.

Many other publicly available network simulators, such as JavaSim (Javasim, 2004), SSFNet (ssfnet, 2000), Glomosim (Glomosim, 2004) and its descendant Qualnet (Qualnet, 2004), attempted to address problems that were left unsolved by ns2. Among them, JavaSim developers realized the drawback of object-oriented design and tried to attack this problem by building a component-oriented architecture. However, they chose Java as the simulation language, inevitably sacrificing the efficiency of the simulation. Moreover, C++ with Standard Template Library (Musser and Saini 1996) can easily achieve high efficiency while maintaining a high level of code reuse, which matched our design goal better than Java. SSFNet and Glomosim designers were more concerned about parallel simulation, with the latter more focused on wireless networks. They are not superior to ns2 in terms of design and extensibility.

Features of SENSE

SENSE is designed to be an efficient and powerful sensor network simulator that is also easy of use. We identify three most critical factors as:

- Extensibility: The enabling force behind the fully extensibility network simulation architecture is our progress on component-based simulation. We introduced a ***component-port model*** that frees simulation models from interdependency usually found in an object-oriented architecture, and then proposed a ***simulation component classification*** that naturally solves the problem of handling simulated time. The component-port model makes simulation models extensible: a new component can replace an old one if they have compatible interfaces, and inheritance is not required. The simulation component classification makes simulation engines extensible: advanced users have the freedom to develop new simulation engines that meet their needs.

- Reusability: The removal of interdependency between models also promotes reusability. A component developed for one simulation can be used in another if it satisfies the latter's requirements on the interface and semantics. There is another level of reusability made possible by the extensive use of C++ template: a component is usually declared as a template class so that it can handle different type of data.

- Scalability: Unlike many parallel network simulators, especially SSFNet and Glomosim, parallelization is provided as an option to the users of SENSE. This reflects our belief that completely automated parallelization of sequential discrete event models, however tempting it may seem, is impossible, just as automated parallelization of sequential programs. Even if it is possible, it is doomed to be inefficient. Therefore, parallelizable models require extra effort than sequential models, but a good portion of users are not interested in parallel simulation at all. In SENSE, a parallel simulation engine can only execute components of compatible components. If a user is content with the default sequential simulation engine, then every component in the model repository can be reused.

Currently Available Components and Simulation Engines:

- Battery Model: Linear Battery, Discharge Rate Dependent and/or Relaxation Battery

- Application Layer : Random Neighbor; Constant Bit Rate

- Network Layer: Simple Flooding; a simplified verion of ADOV without route repairing, a simplified version of DSR without route repairing

- MAC Layer: NullMAC; IEEE 802.11 with DCF

- Physical Layer: Duplex Transceiver; Wireless Channel
- Simulation Engine: CostSimEng (sequential)

Component Simulation Toolkit

COST (Component-Oriented Simulation Toolkit) is a general-purpose discrete event simulator (Chen and Szymanski, 2001). The main design purpose of COST is to maximize the reusability of simulation models without losing efficiency. To achieve this goal, COST adopts a component-based simulation worldview based on a component-port model. A simulation is built by configuring and connecting a number of components, either off-the-shelf or fully customized. Components interact with each other only via input and output ports, thus the development of a component becomes completely independent of others. The component-port model of COST makes it easy to construct simulation components from scratch. Implemented in C++, COST also features a wide use of templates to facilitate language-level reuse.

COST is a library of several classes that facilitates the development of discrete event simulation using CompC++, a component-oriented extension to C++. It differs from many other tools in the simulation worldview it adopts. There are primarily two worldviews that are widely used in the discrete event simulation community: Event Scheduling and Process Interaction. Both have their strengths and weaknesses. The Event Scheduling is much more efficient, but it is hard to program. Process Interaction technique requires less programming effort. However, it is difficult to implement using imperative programming languages and many implementations based on special simulation languages are not efficient.

COST adopts a component-oriented worldview, which is a variation of the Event Scheduling worldview. Using this technique, a discrete event simulation is viewed as a collection of *components* that interact with each other by exchanging messages through communication *ports*. Besides *components*, the simulation contains a *simulation engine* that is responsible for synchronizing *components*. An event-oriented view is adopted to model individual *components*, i.e., the *component* has one or more event handlers each of which performs corresponding actions upon the arrival of a certain type events. Events are divided into two categories. Synchronous events are the messages arriving at the input *ports*, which is sent by its neighboring *components*. Asynchronous events are associated with *timers*, a special kind of ports lying between the *components* and the *simulation engine*. *Components* receive and schedule asynchronous events through *timers*.

COST takes advantage of component-oriented features that are only available in CompC++.

Motivation: From Object to Component

Convenient and powerful as object-oriented programming is, it has its limits. One of these is that it often imposes unnecessary inter-object dependence on the deployment of objects that prevents objects from being reusable. As a small example, in the following diagram, an object *A* calls a method, *g()*, of another object *B*.

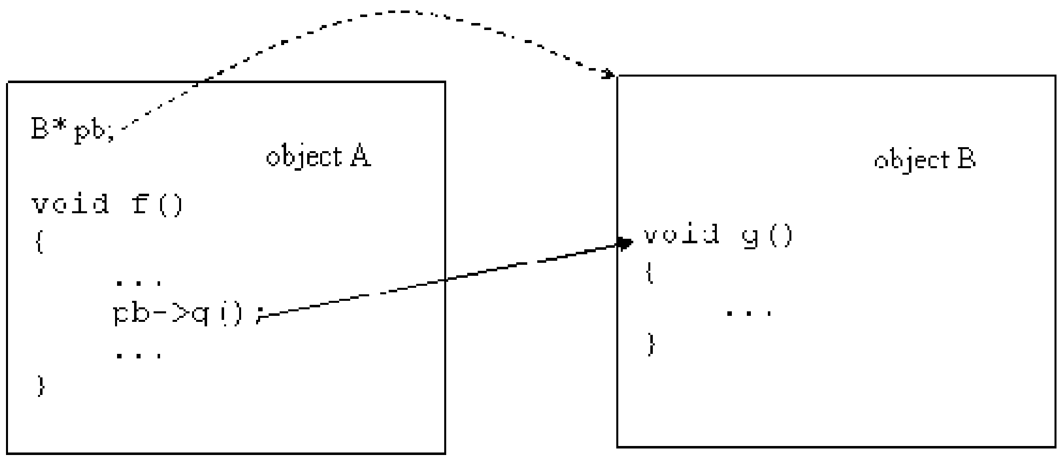


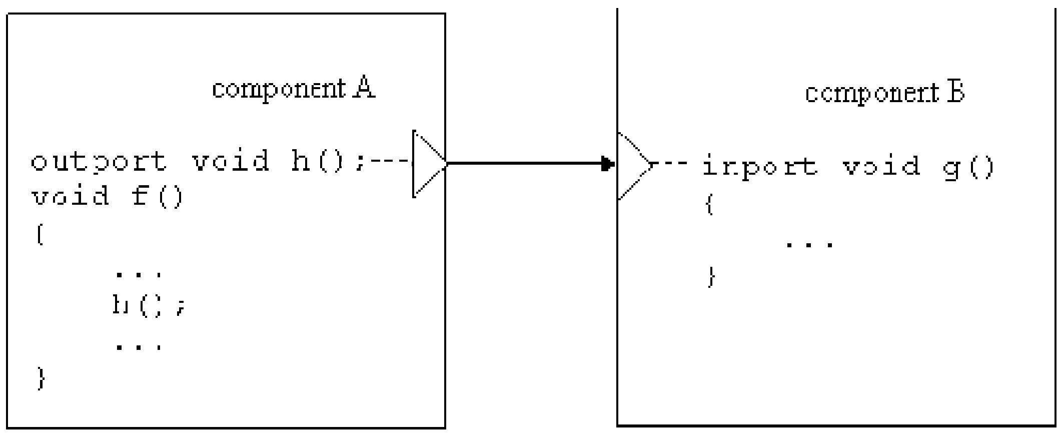
Figure 1. Object Dependencies in Object-Oriented Languages

Object *A* must keep a pointer (or a reference) to object *B* in order to make such a method call. Let us assume that these two objects have been set up correctly in one program. The difficulty arises when *A* is to be *reused* in another program. Obviously, in such a case, either *B* must also be moved over to the new program, or there is another object that is derived from *B* available for *A* to access . If it is the former, things will become worse if some of B's methods are dependent on another object C. As a result, any object that A depends upon, either directly or indirectly, must be available in the new program.

Yet we only need, in the above example, a method that provides the same functionality as *B.g()* does. We should not be concerned with which object can provide such a functionality, whether it be *B* or a different, completely unrelated object *D*. In object-oriented programming, once you make an inter-object method call, you not only introduce explicit inter-object dependence manifested by the method call (represented by the solid arrow in the above diagram), but also implicit dependence that is hard to trace and maintain (represented by the dashed arrow).

The solution is to introduce ***inports*** and ***outports***. The use of inports and outports has been introduced in DEVS formalism (Zeigler et al., 1997; DEVS, 2004), however, in our approach their role expanded to become, next to component themselves, and integral part of composing simulations. An inport defines what functionality an object (now it is becoming a component) provides, and an outport defines what functionality it needs. From the diagram below, it is apparent that the implicit dependence has been removed. Another benefit of having inports and outports is that any interaction a component may have with other components can be deduced only from its interface (which is composed of declarations of inports and outports). In contrast, for an object the same information is clear only after scanning through the entire source code.

The central idea of CompC++ is to add inports and outports to objects to make them look and function like components. The extension to the standard C++ language is minimal: only 4 new keywords (***component***, ***inport***, ***outport***, and ***connect***) and 4 new syntactic rules are needed. The addition of these, nevertheless, opens up a whole new programming paradigm, which is referred to as *component-oriented programming*.



```
outport void h();
void f()
{
    ...
    h();
    ...
}
```

```
inport void g()
{
    ...
}
```

Figure 2. Component Dependencies in Component-Oriented Simulation

Implementation of COST

The first issue of implementing the aforementioned simulation component model is the choice of the implementing language. Discrete event simulators can be roughly divided into two groups: those based on a special simulation language, such as GPSS and SIMSCRIPT (Law and Kelton, 1982), and those based on a general programming language, such as SIMPACK (Fishwick, 1992) and SIMKIT (Gomes et al., 1995) Simulation languages contain abundant semantics designed for simulation, but requires a steep learning curve. General programming languages are more familiar to programmers, but lack the essential simulation constructs.

We chose C++ as the implementation language for two reasons. First, general programming languages always have good compiler support, and thus their execution speed is generally faster after optimization. Second, language-level reusability is a factor as important as component-level reusability, and C++ is one of the few languages that support code reuse well. With STL (Austern, 1999; Musser and Saini, 1996), C++ programs can easily achieve high efficiency while maintaining a high level of code reuse, which matches our design goal.

However, with C++ we ran into a problem. As mentioned in last section, input ports are equivalent to functions, so it is natural to define them as member functions of the component. But how can we represent output ports? C++ language standard requires that the address of an object must be provided when the member function is being called. This conflicts with the requirement that component development should be completely independent. The classical solution for such a problem is a functor, which is the generalization of the function pointer.

Functor

A functor, or a function object, is an object "that can be called in the same way that a function is" (Austern, 1999; Musser and Saini, 1996). A functor class overloads the operator *()* so that it appears as a function pointer. For instance, the following is declaration of a functor class that takes one function argument.

```
template <class T> class functor {
public:
 typedef funct_t bool (*f)(T );
 functor (funct_t _f): f(_f) {}
 virtual bool operator (T t) { return f(t); }
private:
 funct_t f;
};
```

The class *functor* is a helper class that wraps a function pointer of type *funct_t*. Upon invocation, it calls the actual function pointer and returns the result. The syntax of using the functor is exactly the same as that of a function pointer.

The same idea can be applied to member functions as well. In C++, a member function of a class always takes an implicit parameter *this*, which is a pointer to the object upon which the member function will be invoked. As a result, two member functions that belong to different classes but take the same explicit parameters are treated as functions of different types. In the component level, however, they should be viewed as interchangeable. A *mem_functor* declared below can hide the class type as well as the implicit parameter *this*.

```
template <class C, class T>
class mem_functor : public functor {
public:
 typedef funct_t
  bool (C::*f)(T);
 mem_functor (C* _c, funct_t _f)
  : c(_c), f(_f) {}
 virtual bool operator(T t){return c->f(t);}
private:
 C* c;
 funct_t f;
};
```

With these two classes, *functor* and *mem_functor,* it is now straightforward to implement input and output ports. An input port could be simply an instantiation of the *mem_functor* class. Since an output port does not know the component(s) to which it will be connected, it could be represented as a pointer to a *functor*. When connecting an input port to an output port implemented in this way, the address of the *mem_functor* object corresponding to the input port is assigned to the functor pointer corresponding to the output port, because the class *mem_functor* is derived from the *functor* class. When the output port is invoked, the operator *()* of the *mem_functor* class is called, because it is declared as virtual.

Inport and Outport Class

The method of implementing input and output ports directly on top of two functor classes should work well, but there are some practical considerations. For instance, a port should have a name for the purpose of the debugging and a port must be set up properly before it can be used in order to initialize its member variables. Moreover, one to multiple connections would make topology generation more convenient. It is easy to connect an input port to multiple output ports by passing its address to each of them, but when connecting an output port to multiple input ports, the output port must store the addresses of all connected input ports. Those reasons are the main motivation for building the *inport* and *outport* class on top of functor classes.

The *outport* class is declared to be a class with a template parameter that is the type of the events that can be handled by the output port. The function *Setup()* gives the port a string name. The function *Write()* is invoked by the component to output a message. *ConnectTo()* connects an input port to the output port.

```
template <class T>
class outport {
public:
 void Setup(typeii* c, const char* name);
 bool Write(T t);
 void ConnectTo(inport& port);
private:
 std::vector<functor<T>*> inports;
};
```

Similarly, the *inport* class takes one template parameter that is the type of the function argument. It must be bound to a member function of a component, therefore the type of the component is passed as the template parameter of the member template function *Setup()*, as shown below.

```
template <class T>
class inport {
public:
 template <class C>
  void Setup( typeii* c,
   mem_functor<C,T>::funct_t _f,
   const char* name);
 bool Write(T t) { return (*f)(t); };
private:
 functor<T>* f;
};
```

Since the type of the member function bound to the input port must be passed to the *Setup()* function, we need to find a way to construct this type from two template parameters, C and T. Fortunately, this type is declared publicly in the class *mem_functor<C,T>* as *funct_t.*

Simulation Time and Port Index

Until now, functors in COST take only one function argument, which is the message exchanged between components. However, two more arguments are necessary. First, all the components in COST are time-dependent components, so messages should be timestamped. Hence, an extra argument is needed to denote the simulation time at which the message is generated. Another extra argument is for arrays of input ports, which are convenient if a number of input ports are of the same type. All elements in an input port array share the member function bound to them. Therefore, it is necessary to have an extra argument to distinguish between them by their indices.

The index of an input port that is an element of an array is always zero. The resulting *functor* class could be like (other classes must be modified accordingly):

```
template <class T> class functor {
public:
 typedef funct_t bool (*f)(T,double,int );
 functor (funct_t _f): f(_f) {}
 virtual bool operator (T t, double time) {
  return f(t,time,index); }
private:
 funct_t f;
 int index;
};
```

Timer

The timer class requires two different functor classes, *t_functor* and *mt_functor*, because a time event has empty content, so the binding function of a timer only takes the timestamp argument and the index argument. A *timer* object is actually an array of timers, each of which is identified with a unique integer number, as in the input port arrays. The timer class has two methods: *Set()* to schedule an event and *Cancel()* to cancel an event.

```
class timer {
public:
 void Setup( typeii*,
  mt_functor<C>::funct_t, const char* name);
 void Set(double time, int index=0);
 void Cancel(int index=0);
private:
 t_functor * f;
};
```

So far, we have described techniques that we adopted to implement the component-port model in C++. It should be noted that all these implementation details are transparent to users. Users do not need to have advanced knowledge of C++ templates in order to write simulations in COST.

Wireless Sensor Network Simulation

Building a wireless sensor network simulation in SENSE consists of the following steps:

- Designing a sensor node component
- Constructing a sensor network derived from *CostSimEng*
- Configuring the system and running the simulation

Here, we assume that all components needed by a sensor node component are available from the component repository. If this is not the case, the user must develop new components, as described at the COST website (http://www.cs.rpi.edu/~cheng3/cost). We should also mention that the first step of designing a sensor node component is not always necessary, if a standard sensor node is to be used.

This first line of this source file demands that *HeapQueue* must be used as the priority queue for event management. For wireless network simulation, because of the inherent drawback of *CalendarQueue*, and also because of the particular channel component being used, *HeapQueue* is often faster.

```
#define queue_t HeapQueue
```

This header file is absolutely required

```
#include "../../common/sense.h"
```

The following header files are necessary only if the corresponding components are needed by the sensor node component

```
#include "../../app/cbr.h"
#include "../../mob/immobile.h"
#include "../../net/flooding.h"
#include "../../net/aodvi.h"
#include "../../net/dsri.h"
#include "../../mac/null_mac.h"
#include "../../mac/mac_80211.h"
#include "../../phy/transceiver.h"
#include "../../phy/simple_channel.h"
#include "../../energy/battery.h"
#include "../../energy/power.h"
#include "../../util/fifo_ack.h"
```

#cxxdef is similar to #define, except it is only recognized by the CompC++ compiler. The following two lines state that the flooding component will be used for the network layer. These two macros can also be overridden by command line macros definitions (whose format is '-D='.

```
#cxxdef net_component Flooding
#cxxdef net_struct Flooding_Struct
```

For layer XXX, XXX_Struct is the accompanying class that defines data structures and types used in that layer. The reason we need a separate class for this purpose is that each XXX is a component, and that due to the particular way in which the CompC++ compiler was implemented data structures and types defined inside any component is not accessible from outside. Therefore, for each layer XXX, we must define all those data structures and types in XXX_Struct, and then derive component XXX from XXX_Struct.

The following three lines state:

- The type of packets in the application layer is *CBR_Struct::packet_t*
- The network layer passes application layer packets by reference (which may be faster than by pointer, for *CBR_Struct::packet_t* is small, so *app_packet_t* becomes the template parameter of *net_struct*; the type of packets in the network layer is then *net_packet_t*.
- Now that *net_packet_t* is more than a dozen bytes long, so it is better to pass it by pointer, so *net_packet_t** instead of *net_packet_t* becomes the template parameter of the *MAC80211_Struct*; the type of packets in the mac layer is then *mac_packet_t*. Physical layers also use *mac_packet_t*, so there is no need to define more packet types.

```
typedef CBR_Struct::packet_t app_packet_t;
typedef net_struct<app_packet_t>::packet_t net_packet_t;
typedef MAC80211_Struct<net_packet_t*>::packet_t mac_packet_t;
```

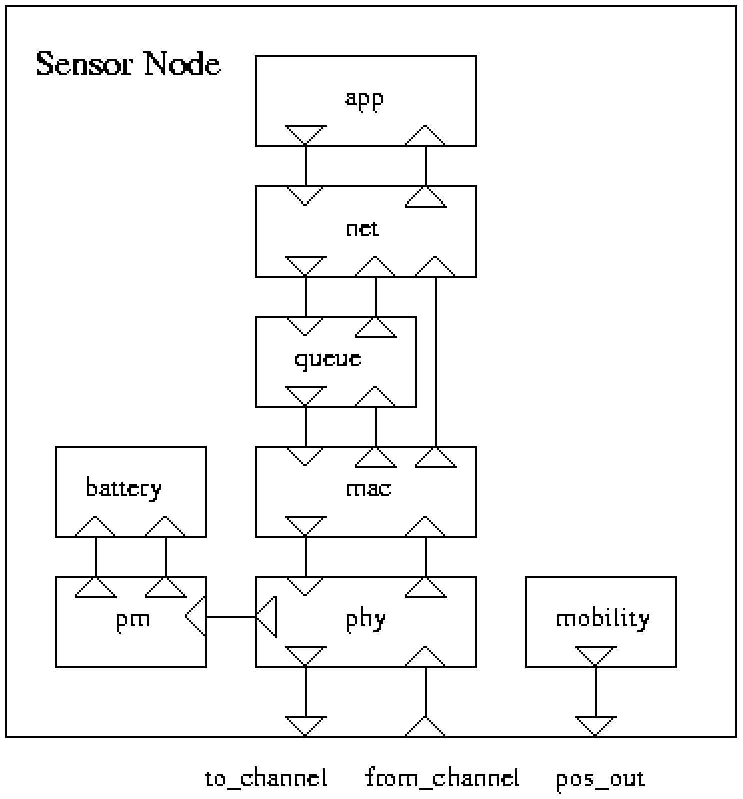


Figure 3. Sensor Node Components in SENSE

Now we can begin to define the sensor node component. First we instantiate every subcomponent used by the node component. We need to determine the template parameter type for each subcomponent, usually starting from the application layer. Normally the application layer component does not have any template parameter.

The picture above shows the internal structure of a sensor node.

```
component SensorNode : public TypeII
{
public:

    CBR app;
    net_component <app_packet_t> net;
    MAC80211 <net_packet_t*> mac;
    // A transceiver that can transmit and receive at the same time (of course
    // a collision would occur in such cases)
    DuplexTransceiver < mac_packet_t > phy;
    // Linear battery
    SimpleBattery battery;
    // PowerManagers manage the battery
    PowerManager pm;
    // sensor nodes are immobile
    Immobile mob;
    // the queue used between network and mac
    FIFOACK3<net_packet_t*,ether_addr_t,unsigned int> queue;

    double MaxX, MaxY;  // coordinate boundaries
    ether_addr_t MyEtherAddr; // the ethernet address of this node
    int ID; // the identifier

    virtual ~SensorNode();
    void Start();
    void Stop();
    void Setup();
```

The following lines define one inport and two outports to be connected to the channel components.

```
            outport void to_channel_packet(mac_packet_t* packet, double power, int id);
            inport void from_channel (mac_packet_t* packet, double power);
            outport void to_channel_pos(coordinate_t& pos, int id);
};

SensorNode::~SensorNode()
```

```
{
}

void SensorNode::Start()
{
}

void SensorNode::Stop()
{
}
```

This function must be called before running the simulation.

```
void SensorNode::Setup()
{
```

At the beginning the amount of energy in each battery is 1,000,000 Joules.

```
   battery.InitialEnergy=1e6;
```

Each subcomponent must als know the ethernet address of the sensor node it resides. Remember the application layer is a CBR component, which would stop at FinishTime to give the whole network an opportunity to clean up any packets in transit. Assiging *false* to *app.DumpPackets* means that if COST_DEBUG is defined, *app* still won't print out anything.

```
   app.MyEtherAddr=MyEtherAddr;
   app.FinishTime=StopTime()*0.9;
   app.DumpPackets=false;
```

Set the coordinate of the sensor node. Must also give *ID* to *mob* since *ID* was used to identify the index of the sensor node when the position info is sent to the channel component.

```
   mob.InitX=Random(MaxX);
   mob.InitY=Random(MaxY);
   mob.ID=ID;
```

When a net component is about to retransmit a packet that it received, it cannot do so because otherwise all nodes that received the packet may attempt to retransmit the packet immediately, inevitably resulting in a collision. *ForwardDelay* gives the maximum delay time a needed-to-be-retransmit packet may incur. The actual delay is randomly chosen between [0,*ForwardDelay*].

```
   net.MyEtherAddr=MyEtherAddr;
   net.ForwardDelay=0.1;
```

```
net.DumpPackets=true;
```

If *Promiscuity* is ture, then the mac component will forward every packet even if it not destined to this sensor node, to the network layer. And we want to debug the mac layer, so we set *mac.DumpPackets* to true.

```
mac.MyEtherAddr=MyEtherAddr;
mac.Promiscuity=true;
mac.DumpPackets=true;
```

The PowerManager takes care of power consumption at different states. The following lines state the power consumption is 1.6W at transmission state, 1.2 at receive state, and 1.115 at idle state.

```
        pm.TXPower=1.6;
        pm.RXPower=1.2;
        pm.IdlePower=1.15;
```

phy.TxPower is the transmission power of the antenna. *phy.RXThresh* is the lower bound on the receive power of any packet that can be successfuly received. *phy.CSThresh* is the lower bound on tye receive power of any packet that can be detected. *phy* also needs to know the id because it needs to communicate with the channel component.

```
phy.TXPower=0.0280;
phy.TXGain=1.0;
phy.RXGain=1.0;
phy.Frequency=9.14e8;
phy.RXThresh=3.652e-10;
phy.CSThresh=1.559e-11;
phy.ID=ID;
```

Now we can establish the connections between components. The connections will become much clearer if we look at the diagram.

```
connect app.to_transport, net.from_transport;
connect net.to_transport, app.from_transport;

connect net.to_mac, queue.in;
connect queue.out, mac.from_network;
connect mac.to_network_ack, queue.next;
connect queue.ack, net.from_mac_ack;
connect mac.to_network_data, net.from_mac_data ;
```

These three lines are commented out. They are used when the net component is directly connected to the mac component without going through the queue.

```
   //connect mac.to_network_data, net.from_mac_data ;
   //connect mac.to_network_ack, net.from_mac_ack;
   //connect net.to_mac, mac.from_network;

   connect mac.to_phy, phy.from_mac;
   connect phy.to_mac, mac.from_phy;

   connect phy.to_power_switch, pm.switch_state;
   connect pm.to_battery_query, battery.query_in;
   connect pm.to_battery_power, battery.power_in;
```

These three connect statements are different. All above ones are between an outport of a subcomponent and an outport of another subcomponent, while these three are between a port of the sensor node and a port of a subcomponent. We can view these connections as mapping from the ports of subcomponents to its own ports, i.e., to expose the ports of internal components. Also remember the connect statement is so designed that it can take only two ports, and than packets always flow through from the first port to the second port, so when connection two inports, the inport of the subcomponent must be placed in the second place.

```
   connect phy.to_channel, to_channel_packet;
   connect mob.pos_out, to_channel_pos;
   connect from_channel, phy.from_channel;
}
```

Once we have the sensor node component ready, we can start to build the entire simulation, which is named *RoutingSim*. It must be derived from the simulation engine class *CostSimEng*. This is the structure of the network.

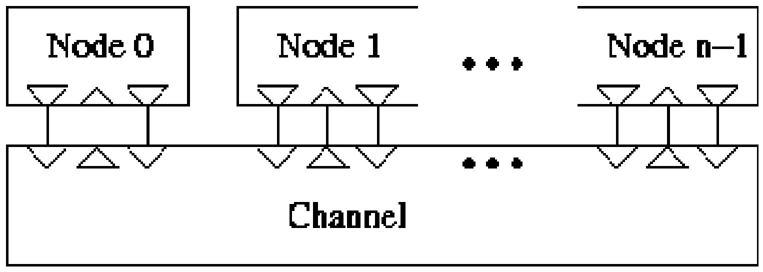


Figure 4. Sensor Network Components in Sense

```
component RoutingSim : public CostSimEng
{
public:
    void Start();
    void Stop();
```

These are simulation parameters. We don't want configurators of the simulation to access the parameters of those inter-components.

```
    double MaxX, MaxY;
    int NumNodes;
    int NumSourceNodes;
    int NumConnections;
    int PacketSize;
    double Interval;
```

Here we declare sense nodes as an array of *SensorNode*, and a channel component.

```
    SensorNode[]  nodes;
    SimpleChannel < mac_packet_t > channel;

    void Setup();
};

void RoutingSim :: Start()
{

}
```

After the simulation is stopped, we will collect some statistics.

```
void RoutingSim :: Stop()
{
    int i,sent,recv;
    double delay;
    for(sent=recv=i=0,delay=0.0;i<NumNodes;i++)
    {
                    sent+=nodes[i].app.SentPackets;
                    recv+=nodes[i].app.RecvPackets;
                    delay+=nodes[i].app.TotalDelay;
    }
    printf("APP -- packets sent: %d, received: %d, success rate: %.3f, delay: %.3f\n",
                    sent,recv,(double)recv/sent,delay/recv);
   for(sent=recv=i=0;i<NumNodes;i++)
   {
                    sent+=nodes[i].net.SentPackets;
                    recv+=nodes[i].net.RecvPackets;
   }
   printf("NET -- packets sent: %d, received: %d\n",sent,recv);
   for(sent=recv=i=0;i<NumNodes;i++)
   {
                    sent+=nodes[i].mac.SentPackets;
                    recv+=nodes[i].mac.RecvPackets;
   }
   printf("MAC -- packets sent: %d, received: %d\n",sent,recv);
}
```

The simulation has a *Setup()* function which must be called before the simulation can be run. The reason we don't do this in the constructor is that we must assign values to its parameters after the simulation component has been instantiated. The *Setup()* function, which you can rename to anything you like, first maps component parameters to corresponding simulation parameters (for instance, assign the value of the simulation parameter *interval* to the component parameter *source.interval*). It then connects pairs of inport and outports.

```
void RoutingSim :: Setup()
{
   int i,j;
```

The size of the sensor node array must be set using *SetSize()* before the array can ever be used.

```
   nodes.SetSize(NumNodes);
   for(i=0;i<NumNodes;i++)
   {
                    nodes[i].MaxX=MaxX;
                    nodes[i].MaxY=MaxY;
                    nodes[i].MyEtherAddr=i;
                    nodes[i].ID=i;
             nodes[i].Setup(); // don't forget to call this function for each sensor node
   }
```

The channel component needs to know the total number of sensor nodes. It also needs to know the value of *CSThresh* since it won't sent packets to nodes that cann't detect them. *RXThresh* is also needed to produce the same receive power in those nodes that can just correctly recieve packets when using different propagation models.

In this example *FreeSpace* is used.

```
   channel.NumNodes=NumNodes;
   channel.DumpPackets=false;
   channel.CSThresh=nodes[0].phy.CSThresh;
   channel.RXThresh=nodes[0].phy.RXThresh;
   channel.PropagationModel=channel.FreeSpace;
```

The channel component also has a *Setup()* function which is to set the size of its outport array.

```
   channel.Setup();

   for(i=0;i<NumNodes;i++)
   {
                   connect nodes[i].to_channel_packet,channel.from_phy;
                   connect nodes[i].to_channel_pos,channel.pos_in;
                   connect channel.to_phy[i],nodes[i].from_channel ;
   }
```

This is to create communication pairs.

```

   int src,dst;
   for(i=0;i<NumSourceNodes;i++)
   {
                   for(j=0;j<NumConnections;j++)
                   {
                   do
                   {
                                   src=Random(NumNodes);
                                   dst=Random(NumNodes);
                   }while(src==dst);
                   nodes[src].app.Connections.push_back(

       make_triple(ether_addr_t(dst),Random(PacketSize)+PacketSize/2,
                                   Random(Interval)+Interval/2));
                   }
   }
}
```

Running the Simulation

To run the simulation, first we need to create a simulation object from the simulation component class. Several default simulation parameters must be determined. *StopTime* denotes the ending time of the simulation. *Seed* is the initial seed of the random number generator used by the simulation.

To compile the program, the following commands can be used:

```
../../bin/cxx sim_routing.cc
```

```
g++ -Wall -o sim_routing sim_routing.cxx
```

The following command line will start the simulation run:

```
sim_routing [StopTime] [NumNodes] [MaxX] [NumSourceNodes] [PacketSize] [Interval]
```

```
int main(int argc, char* argv[])
{
   RoutingSim sim;

   sim.StopTime = 1000;
   sim.Seed = 1234;

   sim.MaxX = 2000;
   sim.MaxY = 2000;

   sim.NumNodes = 110;
   sim.NumConnections = 2;
   sim.PacketSize = 2000;
   sim.Interval = 100.0;

   if(argc >= 2) sim.StopTime = atof(argv[1]);
   if(argc >= 3) sim.NumNodes = atoi(argv[2]);
   sim.NumSourceNodes = sim.NumNodes / 10;
   if(argc >= 4) sim.MaxX = sim.MaxY = atof(argv[3]);
   if(argc >= 5) sim.NumSourceNodes = atoi(argv[4]);
   if(argc >= 6) sim.PacketSize = atoi(argv[5]);
   if(argc >= 7) sim.Interval = atof(argv[6]);

   printf("StopTime: %.0f, Number of Nodes: %d, Terrain: %.0f by %.0f\n",
               sim.StopTime, sim.NumNodes, sim.MaxX, sim.MaxY);
   printf("Number of Sources: %d, Packet Size: %d, Interval: %f\n",
               sim.NumSourceNodes, sim.PacketSize, sim.Interval);

   sim.Setup();
   sim.Run();

   return 0;
}
```

Conclusions

The example given in the previous section has been extended to simulate two innovative protocols for sensor networks: ESCORT (Branch et al., 2005) and SSR (Chen et al., 2005). In the first case, the focus was on identifying groups of sensor nodes that can share communication duties to save energy. The second protocol tested efficiency of a Self-Selecting Routing in which each hop decides its successor on the fly, using Lecture Hall Algorithm for self-selection (Chen et al., 2006). Both systems used large sensor networks (several thousand nodes in each case) and many traffic scenarios requiring multiple runs for each combination of parameters. In all cases, sensor simulator performed reliably and efficiently. Moreover, introducing additional features in the simulation, or trying different variants of the implemented protocols required either small modifications in existing or introduction of new components, greatly simplifying maintenance of the code.

More generally, COST has been used for other network simulations, like queuing networks, computer networks and PCS simulations. These examples can be found at <http://www.cs.rpi.edu/~cheng3/cost>. COST is targeted at the simulation modelers who have beginning or intermediate knowledge of the C++ language. Once they understand the basic component-port model and its support classes, it is fairly easy for them to write models with COST, and, more importantly, to take the component-based approach to model the system to be simulated. Once a component repository with a wide range of models is developed, the modeler will be able to construct a simulation just by connecting components obtained from the component.

COST is a discrete event simulator written in C++ that embodies a component-oriented modeling style. At the heart of COST is a component-port model, which is distinguished from many developed component models by the notion of output ports. Our simulation component classification allows us to extend such a component-port model to make it well suited for discrete event simulation by introducing the implicit timestamp mechanism and timers.

The most distinct feature of COST is the component reusability. Components developed for one simulation can be effortlessly reused in other simulations. With an extensive set of library components, writing simulation in COST could be as simple as dragging a few components from the library and connecting them, as some commercial simulators do. The extra advantage of COST is that building components from scratch is simple.

The only inefficiency of COST simulations comes from the message exchange between components, which may involve several layers of function calls and a few virtual function table lookups. However, this is rather the deficiency of the C++ language, not of the underlying component-port model, because theoretically such overhead can be eliminated during the configuration phase. Had we had a truly component-oriented language, COST would have achieved perfect efficiency.

REFERENCES

Austern, M. H., 1999. *Generic Programming and the STL*. Addison-Wesley.

Branch, J., Chen, G., and Szymanski, B.K., 2005. ESCORT: Energy-efficient Sensor Network Communal Routing Topology Using Signal Quality Metrics. *Proc. Int. Conf. on Networking - ICN 2005*, pp. 438-448. LNCS, Springer-Verlag, Volume 3420.

Chen, G.G., Branch, J. W., and Szymanski, B.K., 2005. Self-selective Routing for Wireless Ad hoc Networks. *Proc. IEEE Int. Conf. Wireless and Mobile Computing, Networking and Communications WiMob 2*005, Vol. 3, pp. 57-64.

Chen, G.G., Branch, J. W., and Szymanski, B.K., 2006. A Self-selection Technique for Flooding and Routing in Wireless Ad-hoc Networks. *Journal of Network and Systems Management*.

Chen, G., and Szymanski, B. K., 2001. Component-Based Simulation. *Proc. 2001 European Simulation Multi-Conference*, pp. 68-75. SCS Press.

DEVS: http://acims.arizona.edu

Fishwick, P. A., 1992. SIMPACK: Getting Started with Simulation Programming in C and C++. *Proc. 1992 Winter Simulation Conference*, ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson, pp. 154-162.

GloMoSim: http://pcl.cs.ucla.edu/projects/glomosim/

Gomes, F., Franks, S., Unger, B., Xiao, Z., Cleary, J., and Covington, A., 1995. SIMKIT: A High Performance Logical Process Simulation Class Library in C++. *Proc. 1995 Winter Simulation Conference*, ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, pp. 706-713.

Javasim: http://www.j-sim.org/

Law, A. M., and Kelton, W. D., 1982. *Simulation Modeling and Analysis*. McGraw-Hill.

Musser, D. R., and Saini, A., 1996. *STL Tutorial and Reference Guide.* Addison-Wesley.

ns2: http://www.isi.edu/nsnam/ns/

Qualnet: http://www.scalable-networks.com/

Zeigler, B. P., Kim, D. And Praehofer, H., 1997. DEVS formalism as a framework for advanced distributed simulation. *Proc 1st Int. Workshop on Distributed Interactive Simulation and Real Time Applications.*