Developer Tutorial for Aqua-Sim

Underwater Sensor Network Lab

University of Connecticut

http://uwsn.engr.uconn.edu/

Contents

How to implement MAC protocol in Aqua-Sim

1 Introduction

In this chapter, we will implement a simple protocol, namely BroadcastMac, to show how to implement MAC protocol in Aqua-sim. In BroadcastMac, nodes just broadcast the packet but do not care what the next hop is. They also do not care whether it has broadcasted the receiving packet before, because routing protocol is responsible for determining that whether it is a duplicated packet. Additionally, BroadcastMac supports back-off mechanism and sleep mode.

2 Implementation

For this protocol, two files, 'broadcastmac.h' and 'broadcastmac.cc' [1], are created in underwatersensor/uw_mac/. 'broadcastmac.h' is a header file where BroadcastMac and some necessary handlers are declared. In 'broadcastmac.cc', BroadcastMac and the handlers are implemented.

2.1 Declaration

Now, we get started. In 'broadcastmac.h', we define a new class called BroadcastMac containing some functions and attributes to implement its functionalities.

underwatersensor/uw_mac/broadcastmac.h

```
1:  #ifndef ns_broadcastmac_h
2:  #define ns_broadcastmac_h
3:
4:  #include "underwatermac.h"
5:
6:  #define BACKOFF    0.1 // the maximum time period for backoff
7:  #define MAXIMUMCOUNTER 4 // the maximum number of backoff
8:  #define CALLBACK_DELAY 0.0001 // the interval between two consecutive sendings
9:
10: class BroadcastMac;
11:
12: class StatusHandler: public Handler{
13: public:
14:      StatusHandler(BroadcastMac*);
15:      void handle(Event*);
16: private:
17:      BroadcastMac* mac_;
```

[1] You can find these two files in 'underwatersensor/uw_mac/' in aqua-sim package.

```
18:  };
19:
20:  class CallbackHandler: public Handler{
21:  public:
22:      CallbackHandler(BroadcastMac*);
23:      void handle(Event*);
24:  private:
25:      BroadcastMac* mac_;
26:  };
27:  class BackoffHandler: public Handler{
28:  public:
29:  BackoffHandler(BroadcastMac*);
30:      void handle(Event*);
31:      void clear();
32:  private:
33:      int counter;
34:      BroadcastMac* mac_;
35:  };
36:
37:  class BroadcastMac: public UnderwaterMac {
38:  public:
39:      BroadcastMac();
40:      int    command(int argc, const char*const* argv);
41:      int packetheader_size_;                  //# of bytes in the header
42:      Event backoff_event;
44:      Event status_event;
45:      Event callback_event;
46:      StatusHandler status_handler;
47:      BackoffHandler backoff_handler;
48:      CallbackHandler callback_handler;
49:      virtual void RecvProcess(Packet*);    // to process the incoming packet
50:      void StatusProcess(Event*);
51:      void CallbackProcess(Event*);
52:      void DropPacket(Packet*);
53:      virtual void TxProcess(Packet*);       // to process the outgoing packet
54:  protected:
55:      //friends
56:      friend class StatusHandler;
57:      friend class BackoffHandler;
58:  };
59:
60:  #endif
```

In line 4, underwatermac.h includes class UnderwaterMac, the base class of BroadcastMac. In line 6-8, two constants for back-off and one constant for consecutive sending are defined. Then, from line 12 to 35, three handlers, which assist BroadcastMac to implement the back-off, are

declared. We will introduce the details later. BroadcastMac is defined within the rest lines.

As you see, BroadcastMac derives from UnderwaterMac, which has integrated much functionality, such as turn on/off the power and binding a variable to the node it attaches. Thus, BroadcastMac can be implemented easily based on the powerful base class.

2.2 Constructor

underwatersensor/uw_mac/broadcastmac.cc

```
59:  BroadcastMac::BroadcastMac():
       UnderwaterMac(),status_handler(this),backoff_handler(this),callback_handler(this)
60:  {
61:        bind("packetheader_size_",&packetheader_size_);
62:  }
```

In line 70, we initialize some variables in the class. In line 72, we bind *packetheader_size_* to a *tcl* variable so that users can set *packetheader_size_* as any value they want in tcl script. Obviously, it is much more convenient than setting new value in the source code and recompiling the whole project. By the way, *packetheader_size_* refers to the size of MAC protocol header. With this variable, we can simulate the overhead from UnderwaterMac protocol header effectively.

2.3 Interfaces to tcl command, upper layer and lower layer

In any subclass of UnderwaterMac, we should overload *command*(), *RecvProcess*() and *TxProcess*() so as to implement the interfaces to *tcl* command, upper layer and lower layer protocols. Note that we do not need to overload *recv*() any longer because UnderwaterMac has split its functionality into *RecvProcess*() and *TxProcess*(). In *RecvProcess*(*Packet* pkt*), we can process the incoming packets (the parameter of function). And in *TxProcess*(), we can process the outgoing packets (the parameter of function). In *command*(), we can process any command passed to class UnderwaterMac.

command()

```
194:  int
195:  BroadcastMac::command(int argc, const char*const* argv)
196:  {
197:      if(argc == 3) {
198:          TclObject *obj;
199:          if (strcmp(argv[1], "node_on") == 0) {
200:              Node* n1=(Node*) TclObject::lookup(argv[2]);
201:              if (!n1)     return    TCL_ERROR;
202:              node_ =n1;
203:              return TCL_OK;
204:          }
205:      }
206:
207:      return UnderwaterMac::command(argc, argv);
208:  }
```

The same as ns2, we process commands passed to this class in *command*(). Note that all commands for MAC protocol will be first passed to this class, so we must return the *command*() of base class like that in line 207 so as to pass the unprocessed commands to base class. And then, base class or its base class can process corresponding commands.

Line 209-215 show how to process command "node_on". The codes bind *node_* to the node object to which the instance of the BroadcastMac protocol attaches.

RecvProcess()

```
70:   void
71:   BroadcastMac::RecvProcess(Packet* pkt){
...         /*some code*/
78:         if (cmh->error())
79:         {
...               /* deal with the error and then return */
84:         }
85:
86:         if(dst==MAC_BROADCAST){
87:               uptarget_->recv(pkt, this);
88:               return;
89:         }
90:
91:         if(dst==index_){
92:               uptarget_->recv(pkt, this);
93:               return;
94:         }
95:         printf("underwaterbroadcastmac: this is neither broadcast nor my packet, just drop
it\n");
96:         Packet::free(pkt);
97:         return;
98:   }
```

In *RecvProcess*(), we first drop the packet if error is detected in line 78-84. Then, we process the packet separately according to the destination address in MAC header. When the destination address is the broadcast address or equals to the address of this node[2], the packet will be passed to routing layer using *uptarget_->recv*(*pkt*, *this*). Otherwise, the packet will be released. Definitely, you can deal with packet here if you want to use some overhearing information.

TxProcess()

```
118:  void
119:  BroadcastMac::TxProcess(Packet* pkt){
...         /*some code*/
121:        cmh->size()+=(packetheader_size_*8);
```

[2] To emphasize that broadcast address is an important special case, we process the two situations separately although they are same. By the way, *index_* can be used to refer to the address of this node.

In line 127, the size of MAC header is added. As explained above, this can simulate the overhead incurred by MAC header.

```
...         /*somde code*/
131:        Scheduler& s=Scheduler::instance();
132:        switch( n->TransmissionStatus() )
133:        {
134:              case SLEEP:
135:                    Poweron();
136:              case IDLE:
137:                    n->SetTransmissionStatus(SEND);
138:                    cmh->next_hop()=MAC_BROADCAST;
139:                    cmh->direction()=hdr_cmn::DOWN;
140:                    cmh->addr_type()=NS_AF_ILINK;
141:                    sendDown(pkt);
142:                    backoff_handler.clear();
143:                    s.schedule(&status_handler,&status_event,txtime);
144:                    return;
145:              case RECV:
146:                    {
147:                          double backoff=Random::uniform()*BACKOFF;
148:                          s.schedule(&backoff_handler,(Event*) pkt,backoff);
149:                    }
150:                    return;
151:              case SEND:
152:                    Packet::free(pkt);
153:                    return;
154:              default:
155:                    /*
156:                     * all cases have been processed above, so simply return
157:                     */
158:                    return;
159:        }
160:
161:  }
```

Because BroadcastMac supports SLEEP mode, we should deal with four states: SLEEP, IDLE, RECV and SEND.

For state IDLE, we should first set the state as SEND, and then change some variables in common header, a virtual header assisting simulation. After completing transmission[3], we set the node state and update the status of *Queue module* via code in line 143 so as to permit *Queue module* to send following packet to MAC layer.

[3] We can use sendDown(Packet*) to send packet to lower layer and use sendUp(Packet*) to send packet to upper layer. These two functions are implemented in UnderwaterMac

For state SLEEP, we turn on the power and then deal with it as IDLE.

Because the node cannot receive and send packet simultaneously, the sending packet will be resent after a random back-off time when the state is RECV.

State SEND will never happen here. We process the packet here just to improve the program robustness.

2.4 Handlers

As mentioned above, BroadcastMac supports SLEEP mode and back-off mechanism via three Handler classes: StatusHandler, CallbackHandler, and BackoffHandler. When we design a Hanlder, we must overload *handle*(). We can call a Handler in the following way:

```
Scheduler& s=Scheduler::instance();
s.schedule(&status_handler,&status_event,txtime);
```

Then *status_hanlder.handle*() will be called after *txtime* seconds. Actually, before sending out the packet, we must calculate *txtime_* in the common header. Usually, we can utilize the function provided by UnderwaterMac to do that.

```
cmh->txtime() = getTxTime(cmh->size());
```

In *StatusHandler*::*handle*(), *BroadcastMac*::*StatusProcess*() is called to set the node state and update the status of *Queue module*. In fact, it updates the status of Queue module by calling *CallbackHanlder*. Finally, *CallbackHanlder::handle*() calls *callback_->handle*(*callback_event*), where callback_ is the handler passed from upper layer[4]. By the way, here are two important things to which we should pay attention. **First, we should call *callback_->handle*(*callback_event*) whenever a packet from upper layer is sent out. If not, MAC layer cannot receive any packet from upper layer. In addition, if the MAC protocol sends out a packet constructed in MAC layer, we should make sure that *callback_->handle*(*callback_event*) must NOT be called. Otherwise, a segment default will occur.**

2.5 Binding BroadcastMac to corresponding Tcl object

We can bind BroadcastMac to Tcl using following code.

underwatersensor/uw_mac/broadcastmac.cc

```
50:  static class BroadcastMacClass : public TclClass {
51:  public:
52:      BroadcastMacClass():TclClass("Mac/UnderwaterMac/BroadcastMac") {}
53:      TclObject* create(int, const char*const*) {
54:          return (new BroadcastMac());
55:      }
56:  }class_broadcastmac;
```

[4] callback_ is assigned value in UnderwaterMac::recv()

Note that the string "Mac/UnderwaterMac/BroadcastMac" used to initialize TclClass will be used to set the values binding to Tcl or to get commands from Tcl. For example, we can set packetheader_size_ as follows.

```
Mac/UnderwaterMac/BroadcastMac set packetheader_size_ 20
```

Then, BroadcastMac::packetheader_size_ will be 20.

3 Needed changes

3.1 Changes in C++ code

If we define a new packet header, we should register it in enum packet_t{} and the constructor of p_info class in common/packet.h. Then we should add some code into 'trace/cum-trace.h' and 'trace/cum-trace.cc' to support tracing. Because BroadcastMac does not introduce new packet header, we do not need to do such work. If your work includes a new packet header, you can refer to "Implementing a New Manet Unicast Routing Protocol in NS2"[1] to find details about above changes. Also, you can find ways to use timer from the document.

3.2 Changes in Tcl code

We also need to register the new protocol via the following code in tcl/lib/ns-packet.tcl.

```
1:    foreach prot{
...         /*protocol names*/
163:        TMAC
164:        /*add your protocol type here*/
165:
166:        NV
167: }   {
168:        Add-packet-header $prot
169: }
```

default value for the variables which bind to Tcl in tcl/lib/ns-default.tcl

```
Mac/UnderwaterMac/BroadcastMac set packetheader_size_ 10
```

Here, we set packetheader_size_ as 10. In other words, if we do not set another value for packetheader_size_, BroadcastMac::packetheader_size_ will be 10.

Makefile

Now, we finish all work and only need to compile the project. To do so, we should update Makefile as follows.

```
149:   OBJ_CC =\
150:   tools/random.o tools/rng.o tools/ranvar.o common/misc.o common/timer-handler.o \
...       #objects files \
218:   underwatersensor/uw_mac/underwatermac.o   underwatersensor/uw_mac/broadcastmac.o \
...       #objects files \
311:   $(OBJ_STL)
```

Then, we can compile the project using command make.

```
[ns-2.30]$make
```

If you add new packet header into common/packet.h, you should 'touch' common/packet.cc to trigger the compiler to recompile the file before typing make.

```
[ns-2.30]$touch common/packet.cc
[ns-2.30]$make
```

How to implement Routing protocol in Aqua-Sim

Please refer to Ref. [1] to implement your routing protocol in Aqua-Sim. In the new routing protocol, probably you want to disable the ARP request. If so, please see 'how to avoid ARP' in the FAQ part.

FAQ

MAC protocol development related FAQ

- How to get the MAC address of the receiver?

The link layer has already set it in the hdr_mac. It can be accessed as the following example.

```
//the type of p is Packet*
hdr_mac* mach = hdr_mac::access(p);
nsaddr_t mac_recver = mach->macDA();
```

Actually, in class LL, both source and destination MAC addresses are filled.

- Invalid command name "something rated to your protocol"

Check the following entries one by one until you solve it.

1. Have you combined the new added protocol with the tcl variable? If not yet, do it as shown in section 2.5
2. Have you recompile the ns project after revising the source code? Sometimes, you need to "make clean" and then "make".
3. Check if the bash environment variable $PATH contains a path to your AquaSim?

4. Check if the link files in AquaSim/ns-allinone-2.30/bin link to the files in your working directories. If not delete them, and run "./install". This case usually happens when you copy a installed package to a new folder.
5. If it still cannot be solved, try to get help from some forums.

Routing Protocol development related FAQ

- How to Avoid ARP?

Before the routing layer send down packet, make sure the hdr_cmn::uw_flag() is true. Otherwise, the LL will do arp, which introduce intolerable delay.

Reference

[1] http://masimum.inf.um.es/nsrt-howto/pdf/nsrt-howto.pdf