2/2/17

Creating Your Own Protocol in Aqua-Sim NG

1. MAC and Routing Protocol Creation:

In some instances, editing and revising currently supported protocols in Aqua-Sim NG will satisfy a user's requirements. While this will limit the necessity of this document for some users, it can still provide assitance in either understanding previously written protocols or editing certain parts correctly. In many causes though, a user or developer will need to fully recreate their own protocol to either bypass the potential overhead of learning and editing a previous protocol or to specialize their own code. To accomplish this it is strongly recommended that users study this document to gain knowledge of Aqua-Sim NG's structure and proper usage.

1.1 Packet Handling for Routing Layer

Since Aqua-Sim NG is currently using standard NS-3 application models we must properly handle new packets generated from the application layer. To do this we must distinguish between newly received packets, from the app layer, or preexisting packets. This is typically handled in the *Recv()* function since this is the entry point for packets on the routing layer. The *Recv()* function must be overloaded by newly created protocols. To avoid any out of bounds faults, it is critical for a newly created packet, from the application layer, to receive the appropriate header fields. More about this can be seen in section 1.3.

Aqua-Sim NG's routing layer does not currently support *SendUp()* function seen in the base class. Typically this could send the given packet to upper application layers, but for this simulator we have a simply counter for sink related stats. This of course can be adjusted if a developer prefers. The *SendDown()* function is a base class inherited function that should be used when passing packets between layers. Currently this function uses some basic header adjustments to help avoid any unexpected consequences due to incorrect header fields. Additionally, this function will schedule the passing of the given packet after the delay, in seconds.

The routing layer base class also offers typical functionality and net device access. For functionality, a user can verify information about the current packet by calling any of the boolean functions. For net device access, such as checking the current node's address, you can use the *m_device* variable, which is set up during net device creation.

1.2 Packet Handling for MAC Layer:

The MAC layer handles packets in a similar fashion to the routing layer, in which the base class offers proper packet passing and basic layer functions. Through MAC layer's base class we can expect to receive packets through the *RecvProcess()* function and *TxProcess()* function. In which the *RecvProcess* is used to process incoming packets and *TxProcess* to process outgoing packets. Both of these functions must be overloaded by the newly created protocol. For passing packets from the MAC layer you use the *SendUp()* function to pass packets to the routing layer and *SendDown()* to pass to the phy layer. To access the net device we can use the *m_device* variable, set up during device creation.

There are also addition features which can be taken advantage of such as *GetTxTime()* and *NotifyRx()/NotifyTx()*. These functions add additional commonly used features and tracer support. Additionally, we offer device control through MAC's base class, such as *PowerOn()* and *PowerOff()*.

MAC layer also offers busy terminal problem simulation. The busy terminal problem is meant to not allow packets to send while the modem is receiving or otherwise busy. This is important even if the current modem is overhearing a packet that is not destined for itself. To better simulate this problem we created a queue on the MAC's base class which will hold all packets, which are ready to send, if the phy layer is currently busy. This is completed during the *SendDown()* function, where we check the transmission status of the current node. If the device is not busy, this function will change the transmission status to the SEND mode and pass the packet to the physical layer to be sent out. Furthermore, this function will automatically set the transmission status to IDLE once the transmission is complete. If a user wishes to specify what status will be set once the transmission finishes, they can simply pass the transmission status as an additional parameter when calling the *SendDown()* function.

1.3 Header Modularity

Proper header management is crucial for accurate simulation and packet handling. To complete this we layer specific modularity among packet headers. With the exception of *AquaSimHeader* which is a broad reaching header with generic fields that can be widely used. Furthermore, since NS-3 simulates packets as a single buffer, we must ensure proper header handling in order to avoid out of bound buffer access and incorrect header assignments. A generic view of Aqua-Sim NG's header ordering is as follows:

AquaSimPacketStamp → AquaSimHeader → MAC header(s) → Routing header(s)

AquaSimPacketStamp is only used on the physical layer and channel, and therefore, can be ignored during upper layer protocol creation. Additionally, some layers may use multiple headers, such as slotted-FAMA (*aqua-sim-mac-sfama*) which uses *MacHeader* and *SFamaHeader* to manage its protocol. Since this is managed modularly, this should not affect other layers of the protocol stack. Furthermore, it is important to add and remove headers in the correct order throughout the created protocol. For example, if we want to change certain fields in a slotted-FAMA packet, we will have to first remove the AquaSimHeader before we can remove the slotted-FAMA header.

```
AquaSimHeader ash;
SFamaHeader sfamaH;
packet->RemoveHeader(ash);         //Remove AquaSimHeader first.
packet->RemoveHeader(sfamaH);      //Followed by removing SFamaHeader.
…                                  //Change fields within sfamaH (and ash if necessary)
packet->AddHeader(sfamaH);         //Adding headers back to packet in reverse order (of removal).
packet->AddHeader(ash);
```

A common error related to header misuse is incorrect header access through serialize and deserializing. This typically occurs when a user removes or adds headers and can either crash the simulation or cause the packet headers to be incorrect. The assert error may look like this: "*assert failed. cond="m_current >= m_dataStart && m_current < m_dataEnd", msg="You have attempted to read beyond the bounds of the available buffer space.*" As specified, this is related to trying to remove a header from a packet which has not been added yet. In this case, by properly using a debugging tool and/or NS-3's logging API, you should be able to find where this assert error is occurring. Additionally, you can use the print feature on a packet to show the current packet's details. This may look like "packet->Print(std::cout);" which will print out the packet headers and the overall packet's size to standard output. The printing feature can be very helpful when debugging your protocol and assuring that all headers are properly handled with no incorrect fields.

1.4 Logging:

Simulation logging is an important feature that assists in quicker and easier debugging and checking function call graphs. Furthermore, it allows for easier variable print outs during simulator runtime, providing a more transparent environment. To accomplish this we use the NS-3's logging API to set up and denote what should be logged. We can first set up logging by including the log header and defining the component within NS-3's log API:

```
#include "ns3/log.h"
…
NS_LOG_COMPONENT_DEFINE("AquaSimMac");
```

From here we can use the logging system to log certain variables or conditions at certain intervals of our protocol. An example of this is the logging the execution of a given function. This can be done by using the function log as follows:

```
NS_LOG_FUNCTION(this);
```

If logging is enabled for this protocol, this function log will output the current function, such as "*AquaSimBroadcastMac::Recv(0xc29500)*". Addition examples of logging can be found in Aqua-Sim NG code and in NS-3 documentation.

1.5 Smart Pointers:

One of the major drawbacks of previous Aqua-Sim iterations was memory management. This problem was mainly due to the simulation scale and lack of support within NS2. Therefore, correctly handling memory allocation and deallocation was directly placed on the developer. Thankfully, in NS-3, we see the implementation of smart pointers. Functions that inherit from the Object class are capable of using these smart pointers. The main benefit we receive from this transition occurs from the internal component of this smart pointer template. In which all assigned smart pointers are unreferenced when they leave their current scope, therefore, removing many unwanted memory leakage bugs. The unreference process is managed by the smart pointer class through a counter and *Unref()* function (which should not be called by the user or developer). Smart pointers are denoted as follows:

```
Ptr<pointer_type> NameOfPointer;
```

Such as:

```
Ptr<AquaSimMac> m_mac;
```

Typically, we only need to reassign a smart pointer to NULL in the case that it is a class defined pointer. An example of this can be seen in the *AquaSimNetDevice* class, where we define the variable *m_mac* as a *AquaSimMac* pointer. An important side note, is that while this variable is of the base class for MAC, it is defined as a child class which is set from the example script or helper script when the device is created. So in order to ensure proper clean up of memory management for this *m_mac* variable, we must reassign it to NULL once the simulation is complete. Note that we do not need to delete this pointer since it is using the smart pointer template which will take care of this work us. Since we are inheriting from the class Object, we can also take advantage of its disposal system. We do this by including a *DoDispose()* function within *AquaSimNetDevice*. Therefore, when the simulation is being destroyed, the *DoDispose()* function will be called, allowing for us to assign NULL to all class defined smart pointers. Additionally, it may be beneficial to call parent classes (excluding Object class), to ensure inherited smart pointers are properly managed as well. Other examples of proper smart pointer handling can also be found in other source files in Aqua-Sim NG, such as *aqua-sim-mac-aloha.cc* and *aqua-sim-phy-cmn.cc*.

1.6 Object Inherited Components:

Another heavily used component that is inherited by the Object class is TypeId. To continue our modularity view for Aqua-Sim NG, we must use the *GetTypeId()* function to declare what NS-3 name will be defined for the current class. For example references we will take code snippets from *AquaSimPhyCmn*.

```
static TypeId tid = TypeId(“ns3::AquaSimPhyCmn”);
```

This allows for users to easily assign the name of the class they plan to us from an example or helper script. For example, in the *aqua-sim-helper.cc* file we directly assign *AquaSimPhyCmn* as the physical layer which will be used in our simulator:

```
 m_phy.SetTypeId("ns3::AquaSimPhyCmn")
```

Next, in *GetTypeId()*, we must assign any unique identifiers for this given type. In the case of *AquaSimPhyCmn* we set the parent and the constructor, which will be used when using this specific class.

```
.SetParent<AquaSimPhy>()
.AddConstructor<AquaSimPhyCmn>()
```

Note that there are no semi-colons used in this portion of this function. This is because we are fully populating the *tid* TypeId variable and using the API given names as distinguishers. And the last varying component of *TypeId()* function is the attribute area. These are used to add attributes to a given type id, which can be set from an example or helper script. An example of this can be seen in *aqua-sim-helper.cc*:

```
m_phy.Set("CPThresh", DoubleValue(10));
m_phy.Set("CSThresh", DoubleValue(0));
```

Here we are manually setting certain variables, prior to device creation, by using their attribute string seen within the *TypeId()* function.

```
.AddAttribute("CPThresh", "Capture Threshold (db), default is 10.0 set as 10.",
    DoubleValue (10),
    MakeDoubleAccessor(&AquaSimPhyCmn::m_CPThresh),
    MakeDoubleChecker<double> ())
```

```
 .AddAttribute("CSThresh", "Carrier sense threshold (W), default is 1.559e-11 set as 0.",
     DoubleValue(0),
     MakeDoubleAccessor(&AquaSimPhyCmn::m_CSThresh),
     MakeDoubleChecker<double>())
```

While there are many different variations of attribute assignments, here we are using double variable types. More on this can be found on NS-3's documentation for TypeId.

1.7 Event Scheduling:

To assist in handling events in Aqua-Sim NG, we use a combination of NS-3's *Timer* API and *Simulator* API. The most typically call for scheduling consisting of the delay until execution, the memory pointer, the object, and any necessary arguments (may also have no arguments). One example of this can be seen in the *SendDown()* function of *AquaSimRouting*:

```
Simulator::Schedule(delay, &AquaSimRouting::SendPacket, this, p);
```

Here we are scheduling an event for this (the current node's routing layer) to execute the *SendPacket()* function after a given delay, while passing the packet *p*. Additionally, we can use a timer to assist in scheduling an event. This can allow for more control over checking if a timer is still running and canceling it if needed. An example of this can be seen in *aqua-sim-mac-uwan*.

```
m_startTimer.SetFunction(&AquaSimUwan_StartTimer::expire,&m_startTimer);
m_startTimer.Schedule(Seconds(0.001));
```

The variable *m_startTimer* is an inherited *Timer* class, which is used to decide when a given function should be called. In this case we are calling the *expire()* function of the object *m_startTimer*. Additionally, we are setting this event to occur after a set period of time. In some cases, it may be necessary to cancel a timer before it executes. To accomplish this, we can use *Timer*'s API to first check if a timer is running, and then cancel the timer which will stop the *SetFunction* from executing.

```
if( m_sleepTimer.IsRunning() )
        m_sleepTimer.Cancel();
```

1.8 Coding Style

Since Aqua-Sim NG is meant to work with NS-3, it is important for us to follow their coding styles for easier readability and consistency. Therefore, it is highly encouraged to read over the coding techniques mentioned on NS-3's site: ns3 coding style <https://www.nsnam.org/developers/contributing-code/coding-style/>.

2. Header Protocol Creation:

In the case that a user or developer wishes to create a specialized header for their protocol, they may have to create their own header class. This section is targeted at doing this or at the very least helping a user to better understand of how headers function.

1.1 Header Creation

Newly created headers must inherit functions from the *Header* class, in order to ensure proper functionality in NS-3. To do this we must include the following functions:

```
  virtual TypeId GetInstanceTypeId(void) const;
  virtual void Print(std::ostream &os) const;
  virtual void Serialize(Buffer::Iterator start) const;
  virtual uint32_t Deserialize(Buffer::Iterator start);
  virtual uint32_t GetSerializedSize(void) const;
```

The *GetInstanceTypeId()* is rather straight forward and returns the TypeId of the given class. The *Print()* function is meant to add all header variables to the provided output stream, which is used when calling the *Print()* function of a packet. *Serialize()* and *Deserialize()* are used when adding or removing headers to a packet, respectively. Since a packet in NS-3 is a single buffer, we use the *Buffer* API to assist in this process of either writing or reading. To ensure consistency, it is important that we serialize and deserialize all header variables, in the given buffer, in the same order. And finally, we see the *GetSerializedSize()* which denotes the expected size of the current header, in bytes.

Examples of header classes in Aqua-Sim NG can be found in the source file *aqua-sim-header.cc*.

1.2 Field Management

Beyond the standard functions of a header, we must also include get and set functions. This should be relatively straight forward, consisting of either assigning a parameter to a class variable, or returning the value of a class variable.

Since *AquaSimAddress* is a predefined type, it is important to change this value to a integer before writing to a buffer. This may look something like the following:

```
i.WriteU8(m_sAddr.GetAsInt());
```

In this example *i* is the buffer iterator and *m_sAddr* is the *AquaSimAddress* we are trying to write. Furthermore, when trying to read from this buffer we want to use a typecast back to *AquaSimAddress*. This may look like the following:

```
m_sAddr = (AquaSimAddress) i.ReadU8();
```

To avoid any out of bounds segment faults it is important to properly set up your *GetSerializedSize()* dependent on the header variables' sizes. For instance, in the previous example, *m_sAddr* is 8 bits in size or 1 byte. Therefore, our *GetSerializedSize()* should return 1, assuming this is the only variable in this header.

Contact Information:

Questions, comments, issues, or anything else can be directed to:

Robert Martin
Robert.Martin@engr.uconn.edu

or posted directly under aqua-sim-ng issues on GitHub.