Navigating through SNSim’s network stack.

The SNSim’s network stack, which is illustrated in Figure 1, is a subset of the typical network stack found in wired networks and also in SWANS. Given the specifics of the wireless sensor network however, the Transport Layer, which was present in the original JiST-SWANS distribution, was not inherited in SNSim.

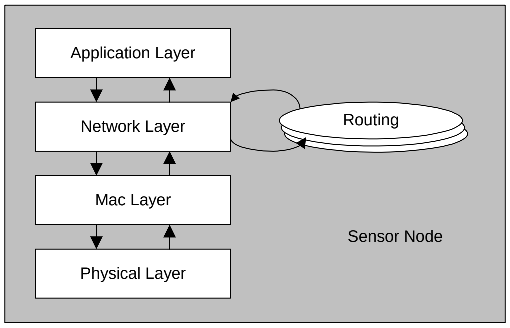

Figure 1. SNSim’s network stack

The Network Layer represents a switchboard between packets coming from the upper layers (Application Layer), lower layers (Mac Layers) and the Routing paradigms, and based on the destination address of the messages, it forwards the packets accordingly.

Figure 2 illustrates the overall message flow between the Application Layer, Mac Layer, Network Layer and Routing.

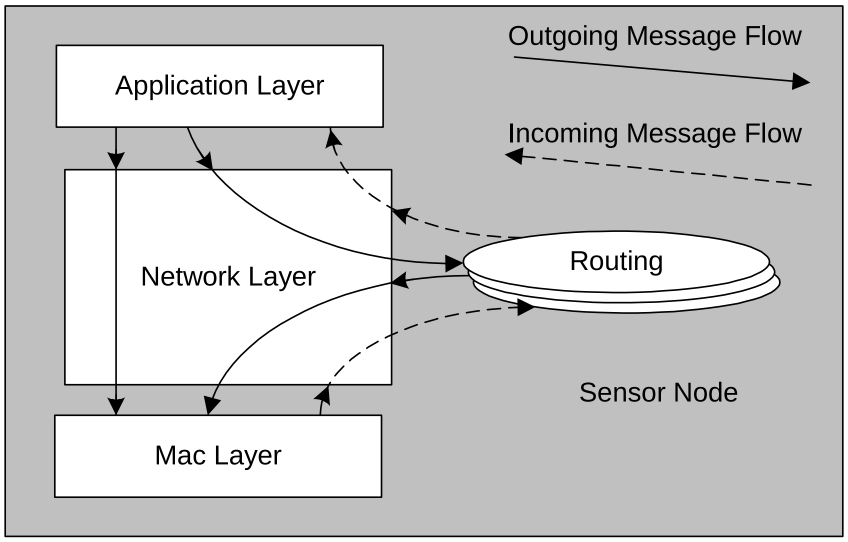

Figure 2. Overview of the message flow in the upper layers of the network stack

As it can be observed, most of the messages are flowing through the Routing element. Imagine that a user's application may need to send a data-packet to a distant node, located couple of hops away. The application may know the IP address of the destination node, but not the means of reaching it (the route). The Network Layer will let the Routing compute and retrieve the address of one (or more) of the neighboring (1-hop) nodes to which the message to be sent immediately in its route to the destination. The single exception to this rule applies to **outgoing Broadcast messages**, which are detected by the Network Layer and forwarded directly to the Mac Layer (there is no need for routing since broadcasting means transmitting to whoever can hear the message within communication range). But since routes may be built considering all types of messages, ALL the ***incoming*** messages however (both unicast and broadcast ones) are sent directly to the Routing algorithm, where they may be used either for updating the routing information, finding the next-hop node if the packet has not reached its final destination, or passed up, to the Application Layer, if and only if the message destination is the node itself.

Figure 3 gives a detailed description of the outgoing flow of messages and the associated methods that are being called within the corresponding .java files.

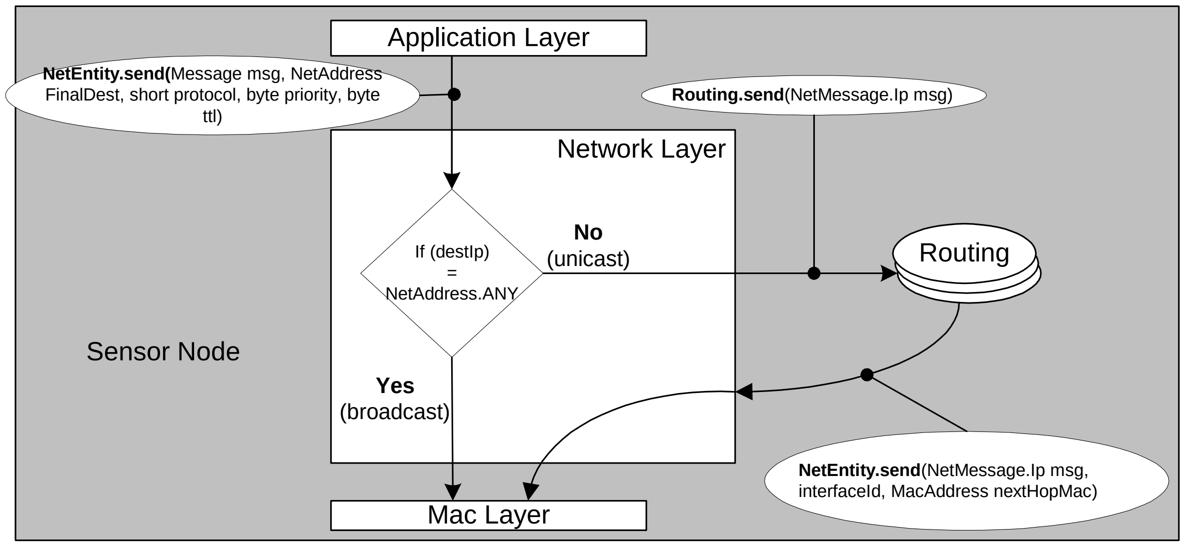
Application Layer
NetEntity.send(Message msg, NetAddress FinalDest, short protocol, byte priority, byte ttl)
Routing.send(NetMessage.Ip msg)
Network Layer
If (destIp) = NetAddress.ANY
No (unicast)
Routing
Sensor Node
Yes (broadcast)
NetEntity.send(NetMessage.Ip msg, interfaceId, MacAddress nextHopMac)
Mac Layer

Figure 3. Outgoing message flow in the upper network stack

In order to send a message from the Application Layer, the following parameters are needed:

Message – a user defined pojo containing the data to be transmitted (the payload)

NetAddress – the IP of the node that represents the FINAL destination of this packet

Protocol – if there are several routing protocols implemented, this is used to specify the routing protocol that is to be used for routing the packet

Priority – indicate if some packets have a higher priority than others

TTL – time to leave: in a congested network, indicate the amount of time a packet will be held before being dropped if continuous attempts of forwarding the data fail.

The application layer will call `NetEntity.send( … )` with the above parameters specified. For example, broadcasting a message can be specified as:

```
NetEntity.send(myMessage, NetAddress.ANY, Constants.MY_PROTOCOL, 1, (byte)100)
```

Or, unicasting to a known IP (net) address:

```
NetEntity.send(myMessage, destinationNetAddress, …. )
```

If the outgoing message represents a broadcast, it will be sent directly to the MAC layer, bypassing any routing primitives. However, if it is a unicast message, it will be passed to the Routing algorithm to be handled. In the latter case, the `NetEntity` will call the following method member of the Routing class:

```
Routing.send(NetMessage.Ip ipMsg)
```

The user must decide what to do with the message at the routing layer. Most likely, it needs to find the 1-hop neighbors' MAC address to forward the packet toward its final destination.

Note that the NetEntity will ship a wrapped version of the original message the application layer has sent. To obtain the content, use the `.getPayload()` method

```
MyMessage myMessage = ipMsg.getPayload()
```

Once the MAC of the next-hop neighbor has been retrieved, the routing layer can proceed sending the packet down the stack to the MAC layer. However, since it does not have access directly to the MAC Layer, it must rely on the following method of the Network Layer

```
netEntity.send(ipMsg, interfaceId, MacAddress);
```

Do note that the Network Layer will require the wrapped version of the payload, not the payload itself (that is, a `NetMessage.Ip` formatted message, which will contain also information about the original source of the message and its final destination). The user needs to use for the `interfaceId parameter the` default interface, which is indicated through `Constants.NET_INTERFACE_DEFAULT`. The `MacAddress` corresponds to the next-hop node the packet will be forwarded to.

If the IP address of the neighbor to forward the data-packet is known , you may use the `neighboursList` (`NodesList`) to retrieve the associated Mac address as follows:

```
neighboursList.get(nextHopDestIP).mac;
```

where the neighboursList is automatically preloaded by the hearbeat protocol that is executed in the first hour of simulation time.

WARNING! The following method may be also called from the Routing Layer

```
NetEntity.send(myMessage, NetAddress.ANY, Constants.MY_PROTOCOL, 1, (byte)100)
```

Since it is a broadcasting message, it will be send immediately to the Mac layer. However, if it wouldn't be a broadcasting message, the Network Layer will NOT send the packet to the MAC Layer. Instead, since the Network Layer is unaware of the origins of the call, it will handle the message back to the Routing Layer, just as if the Application Layer has called it, risking creating an infinite message-passing loop between the Routing and Network Layer.

Figure 4 represents a detailed illustration of the incoming flow of messages and the associated methods that are being called in the corresponding .java files.

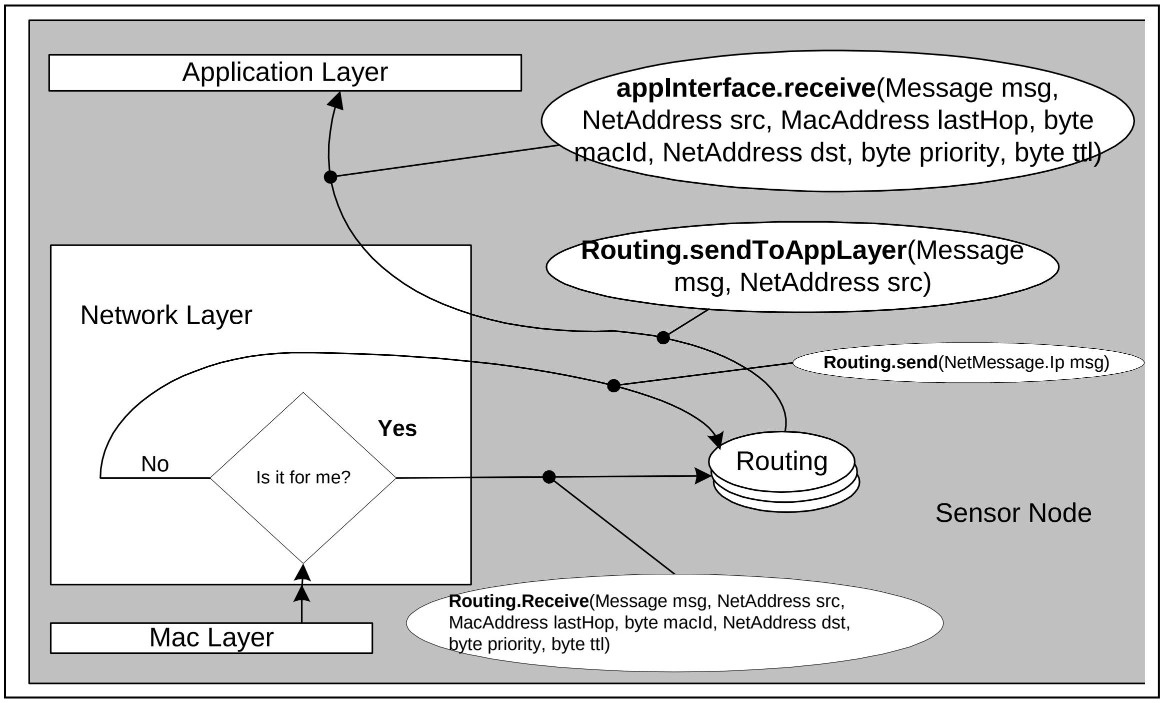
Application Layer
appInterface.receive(Message msg, NetAddress src, MacAddress lastHop, byte macId, NetAddress dst, byte priority, byte ttl)
Routing.sendToAppLayer(Message msg, NetAddress src)
Network Layer
Routing.send(NetMessage.Ip msg)
Yes
No
Is it for me?
Routing
Sensor Node
Routing.Receive(Message msg, NetAddress src, MacAddress lastHop, byte macId, NetAddress dst, byte priority, byte ttl)
Mac Layer

Figure 4. Incoming message flow in the upper network stack

The Network Layer, upon receiving a message from the Mac Layer, will check to see if the hosting node represents the final destination of the packet. If it is not, meaning that the current node is just a relay of the packet in its way to the final destination, the Network Layer will ask the routing protocol to "send" the packet again (aka, forward). If this node represents the final destination of the packet, it will be treated through the "receive" method of the Routing Layer, in which the packet must be treated, and, in most cases, sent to the application layer as well. Note that, the Network Layer will not forward any packet to the application layer by itself. Here is the "receive" method syntax the Network Layer will call the following method of the Routing class:

```
Routing.Receive(Message msg, NetAddress src, MacAddress lastHop, byte macId,
NetAddress dst, byte priority, byte ttl)
```

The “send()” method is the same one called when unicasting a packet from the Application Layer.

Indicating the content of the message, the IP of the original producer of the message, the last hop Mac address the message is coming from, the Mac interface it has been received through and the final destination of the packet. These may be used by the Routing layer to decide if the packet has reached its ***final*** destination.
The routing layer will decide whether the incoming data packet is to be forwarded again, and/or update its routing information if necessary. If the hosting node does not represent the final destination (represents just an intermediate node on the route) of the data packet, the Routing Layer must forward the message to the next node on the route following the indication for outgoing messages. Otherwise, it should pass the message to the Application Layer, which can be done directly by calling the following locally defined method:

```
Routing.sendToAppLayer(Message msg, NetAddress src)
```

In turn, the Application Layer will be notified by the incoming message by being called its local method, which follows:

```
appInterface.receive(Message msg, NetAddress src, MacAddress lastHop, byte
macId, NetAddress dst, byte priority, byte ttl)
```

APPENDIX

You may add the following lines of code to the Routing Layer implementation to ease the transmission of the messages to the lower stack layers:

If you have the unwrapped payload, then you can use:

```
public void sendToLinkLayer(Message msg, NetAddress originalSourceIP, NetAddress finalDestinationIP, NetAddress nextHopDestIP)
   {
      NetMessage.Ip ipMsg = new NetMessage.Ip(msg, originalSourceIP /* src IP addresss */, finalDestinationIP,
Constants.NET_PROTOCOL_INDEX_1, Constants.NET_PRIORITY_NORMAL, (byte)1);

      if (nextHopDestIP == NetAddress.ANY)
         netEntity.send(ipMsg, Constants.NET_INTERFACE_DEFAULT, MacAddress.ANY);
      else
      {
         NodeEntry nodeEntry = neighboursList.get(nextHopDestIP);
         MacAddress macAddress = nodeEntry.mac;
         netEntity.send(ipMsg, Constants.NET_INTERFACE_DEFAULT, macAddress);
      }
   }
```

or, if you have the wrapped payload as a NetMessage.Ip you can use:

```
   public void sendToLinkLayer(NetMessage.Ip ipMsg, NetAddress nextHopDestIP)
   {
      if (nextHopDestIP == NetAddress.ANY)
         netEntity.send(ipMsg, Constants.NET_INTERFACE_DEFAULT, MacAddress.ANY);
      else
      {
         NodeEntry nodeEntry = neighboursList.get(nextHopDestIP);
         MacAddress macAddress = nodeEntry.mac;
         netEntity.send(ipMsg, Constants.NET_INTERFACE_DEFAULT, macAddress);
      }
   }
```