TUTORIAL FOR UW NETWORK CODING USING OPNET AND MATLAB

CHRIS LYDICK

1. Walkthrough

The user should have downloaded the *.zip* or *.tar.gz* file from the UWEE FUNLAB website (http://www.ee.washington.edu/research/funlab), unzip, and follow the *README.txt* file's instructions for where to place which files. Then execute OPNET, and follow along...

Within the OPNET environment, open the *networkcoding* project and select the *test_scenario_one* scenario. Figure 1 demonstrates what that window should look like. What we see is a configuration of network-coding enabled routers, and work station end nodes with a single UDP traffic demand. The network coding enabled routers function as regular routers, and intercept and do special processing to specific packets (identified by the bulk-size, which we can specify in the traffic demand).

To demonstrate what these network coding-enabled routers do and how to configure them, one must first double click one of the router icons, which displays the following window:

From here, we see the four interfaces (named *eth_rx_...* and *eth_tx_...*), which submit packets to a queue (named *mac_x_x*), and then to their specific ARP processor (named *ARPx*). From there, packets identified as UDP packets from the network coding traffic demand (ie, the packets we are processing for network coding) follow the dark blue arrow to either a sink processor or to a nc-proc processor.

Because of the inherent necessity of network coding to operate on unidirectional links, each interface is designated as a *SEND* or *RECEIVE* interface (for the purposes of this tutorial). For interfaces which will only *SEND* network coded packets (remember, links are unidirectional only in terms of the network coded packets), received network-coded packets will be assumed in error, and should be discarded. They are done so by forwarding the packet to the *sink* processor. Conversely, the *RECEIVE* interfaces forward the network-coded packets to the *nc-proc* processor.

The *nc-proc* processor then simply copies the packet to all of its output streams (again, the dark blue arrows going from the *nc-proc* processor to other processors). The *nc-proc* processor has been hard-coded (see future work) to require a constant number of output interfaces (for other topologies we can make that number change, but it has to remain constant for all routers in the network), thus the need to send some of the streams to the *sink*.

The steps for configuring a network-coding aware router are:

(1) First begin by removing all dark blue arrows from the router model (See Figure 2).

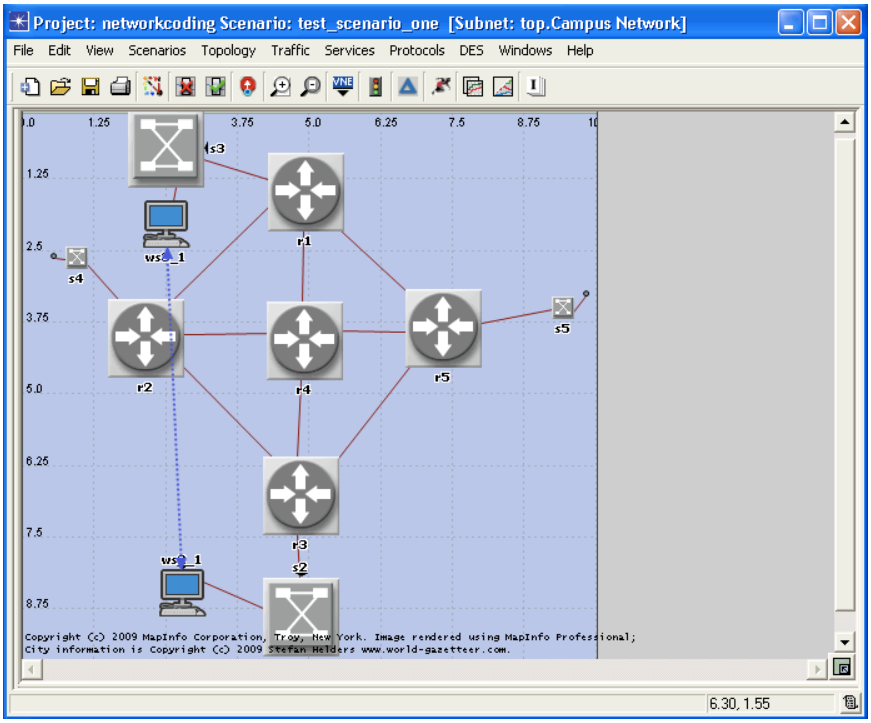

FIGURE 1. OPNET Project Window: networkcoding project.

(2) Identify which interfaces are *SEND* or *RECEIVE* interfaces (briefly described above).
(3) For *SEND* interfaces:
    (a) Make a single dark blue stream path from the ARP processor of that interface to the sink processor. This is to make sure that all received packets on that interface are immediately dropped.
    (b) Make another dark blue stream path from the *nc-proc* processor to the ARP processor of that interface. This guarantees that copied packets from other interfaces make it to this interface.
(4) For *RECEIVE* interfaces:
    (a) Make a single dark blue stream path from the ARP processor of that interface to the *nc-proc* processor. This guarantees that packets received on this interface get copied to the other interfaces.
    (b) Make a single dark blue stream from the *nc-proc* processor to the sink processor. This will have to be done once for each *RECEIVE* interface.

To continue configuring the router, one must now setup the timeouts (or delays) for the routers to obtain and combine network coded packets. But first, let us dig into the actual methodology for these timeouts...

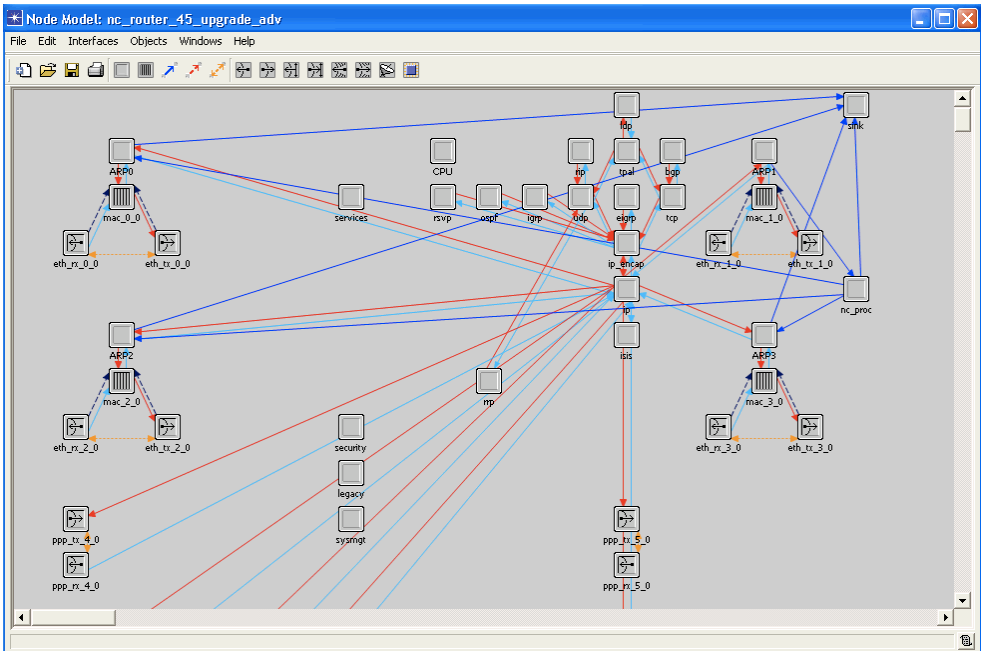

FIGURE 2. OPNET Network Coding Router Node Model Window. Interface 0, 2, and 3 are *SEND* interfaces, while Interface 1 is a *RECEIVE* interface.

At the ARP level, a set of functions was needed to obtain the network-coded packets, enqueue them and at some time later, combine them and send them along the *SEND* interface(s). In some routers, a timeout is unnecessary, as the network-coded packet sometimes only needs to be copied and is not dependent on other packets within the network; in other words, the packet can be immediately forwarded from a *RECEIVE* interface, combined with the *SEND*'s *nc_value*, and forwarded to the next router.

There are also two other adjustable parameters at the *SEND*'s ARP process. These two parameters are quite critical, but only need to be adjusted if necessary. Their defaults will enalbe correct function. The first parameter which is the *Bitsize of NC Value* referrs to just that – the size of the code that we can use within the finite fields calculation. By default these reside within a byte boundary (ie, 8 bits), but it can be changed to anything smaller or larger. The second adjustable parameter adjusts how the *SEND*'s output *nc_value* is calculated. By default (which is a value of 0), the value is calculated randomly, but the user can specify specific values to load the network at if desired. That is done through this parameter.

Figure 3 shows the path of the network-coded packets traversing through the network for the scenario in Figure 1. Using the same naming conventions from Figure 1, we see *R1* (outputting with the three red arrows) does not need to wait to send packets to its neighbors. This is because it only receives packets from a single interface. A value of zero

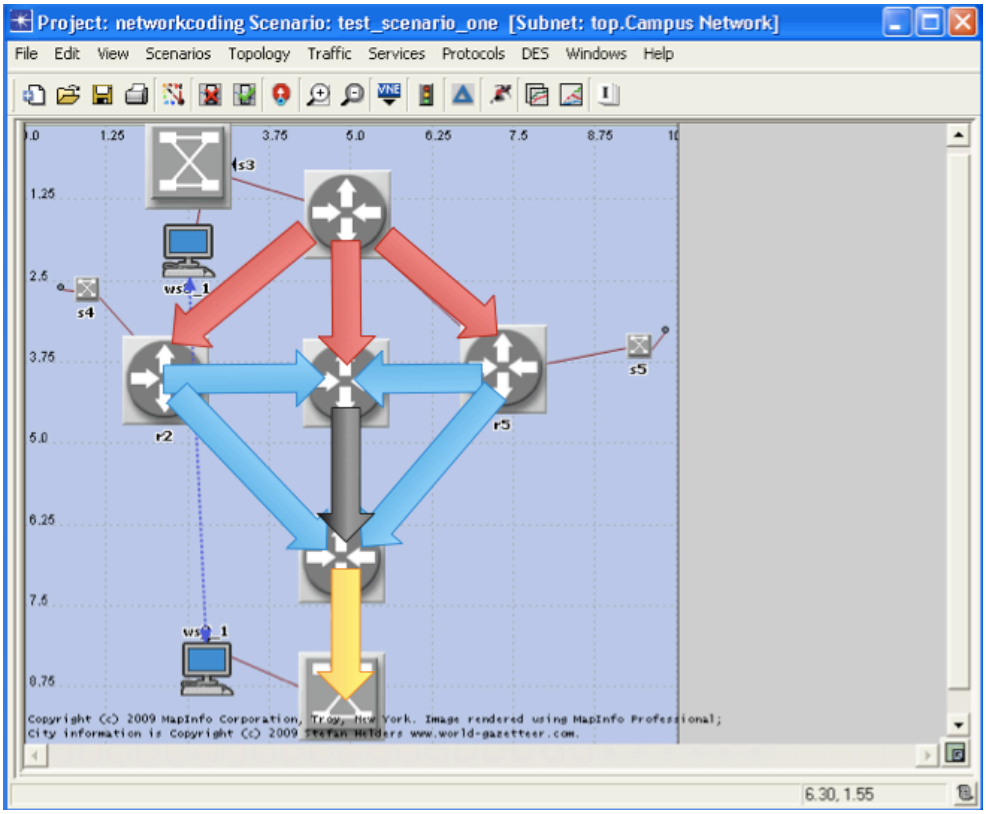

FIGURE 3. Demonstration of Timing and Packet Flow for the networkcoding Project

can be assigned to the timeout value for *R1*. With *R2* and *R5*, we see similarly, that these routers do not need to wait to forward the packets from the *RECEIVE* interface(s) to the *SEND* interface(s). It only has one input, therefore it can promptly send the packets, being also given a value of zero to the timeout value.

We see a different scenario for the case of *R4*, as it should receive the packet from the blue paths at the same time, but will receive the packet from the red path at an earlier time. Because of this, we must institute some kind of delay to enqueue the packets so that they all will be combined and sent to *R3*.

R3 is in the same boat, it receives packets at different intervals, so a nonzero timeout value must also be assigned. Another way to think is to observe that the red arrows occur at time step 1, blue at time step 2, black at time step 3, and yellow at time step 4

This timeout can be adjusted by right clicking the ARP processor for each interface and navigating to find the *Timeout Value*. The default is set to zero.

The user may also adjust this value to institute an infinite delay – which is necessary to evaluate how the network coding would respond to link failures. This can be done by

adjusting the delay parameter to be something very high, such as 100 seconds. Basically that amounts to having some particular interface enqueueing the network-coded packets for 100 seconds. Alternatively, one can just direct the dark blue output stream that normally goes from the *nc-proc* to the ARP interface to go from the *nc-proc* to the sink.

Delving deeper into the ARP processor, the code has been adapted to identify the network coded packets, and immediately direct the packets to a static stream (identified as stream 2) to the *nc-proc* processor. The ARP processor also receives network coded packets from the *nc-proc*.

Upon receipt of a network-coded packet from the *nc-proc* processor, the *SEND* ARP processor enqueues the packet and makes sure a timeout has been established (using the timeout value described above). Once said timeout occurs, the ARP processor dequeues all packets, and combines them using a Matlab interface to the gf() function for finite fields. It also uses a randomly (though it can also be statically set) value to multiply the combined packets value with that of the interfaces value. This value is referred to as the *nc-value*The packet is then set out of the interface.

For debugging and evaluation purposes, the programmer needs only to then identify the module ID of the ARP module (printed to the console at the beginning of the simulation) and that of the interfaces random value to manually verify the results. Figure 4 shows the entire network, all output interface *nc-values*, and the values of the network coded-packets as they traverse through the network. The rectangular boxes with the numbers are the *nc-value*s that are multiplied by each combined outgoing packet.

In Figure 5, we see the console output that shows when a user executes the simulation (given the above *nc-values* at each output interface). The output may seem cryptic at first, which is why understanding the output is important. It takes the form:

```
(Current simulation time | Process ID) At <NC Value at the output interface> = Value of
          packet after combination and multiplication of nc value at output.
```

So, here we see at time 163.5 (seconds), the value at processor ID 4980, which refers to node 5 (see Figure 4), with output *nc-value* of 102, gives a correct final product *nc-value* of 62. By causing failures on individual (or multiple) links, we see this value change, as the combination of *nc-values* differs.

2. Future Work

(1) Global function to hand out *nc_values* for each node, as opposed to relying on the random number generator to issue unique values. New seeds must currently be used if a simulation ends up generating multiple outputs for the same *nc_value*.
(2) Make the *nc-proc* Processor capable of sensing the number of flows it has going to other processors within the router model, reducing the need for hard-coding a set number of links to the *sink* processor.
(3) The memory management utilized within this work, may not be well suited for large simulations.

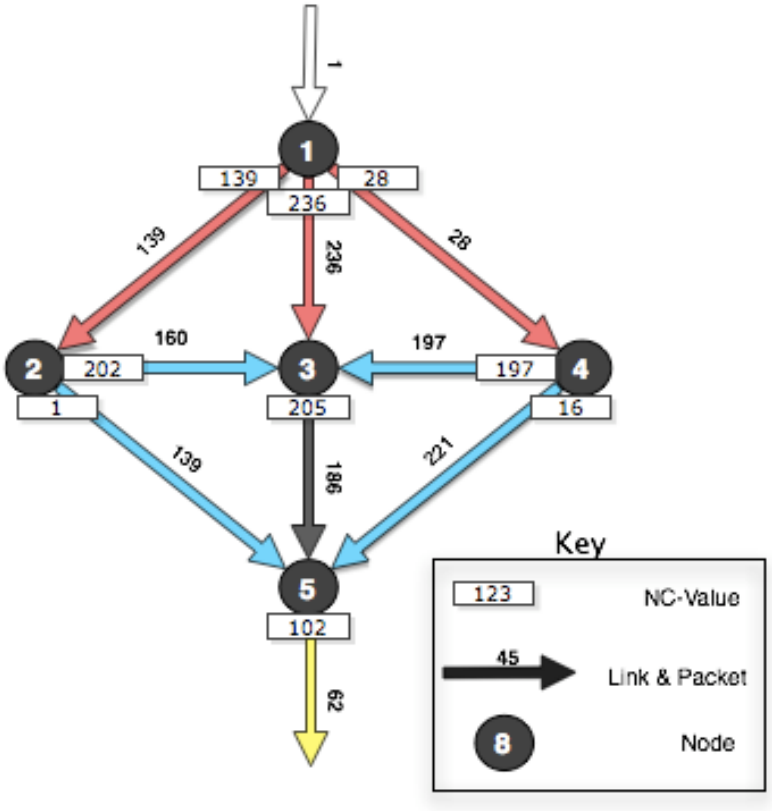

FIGURE 4. NC-Values and Packet Values for a Simulation without Failures

(4) While the implementation of Matlab was essential, if one could generate some functions to calculate Finite Fields without the use of Matlab, that may reduce some complexity and wait-time during execution of the simulation.

There are also some limitations to cover in terms of the implementation of this code, and they are as follows:

(1) Network sizes are definitely an issue, as this implementation was initially created to work on smaller networks and handle small loads of UDP data between source and destionation.
(2) Currently, this implementation only codes a value within the UDP header, and not the data itself. This was done primarily because it was very simple to add "hidden" data within the UDP frame, which is invisible to the end-user and to the simulation statistics (ie, the addition of this data does not make the packet appear to be any larger to the rest of the simulation environment.

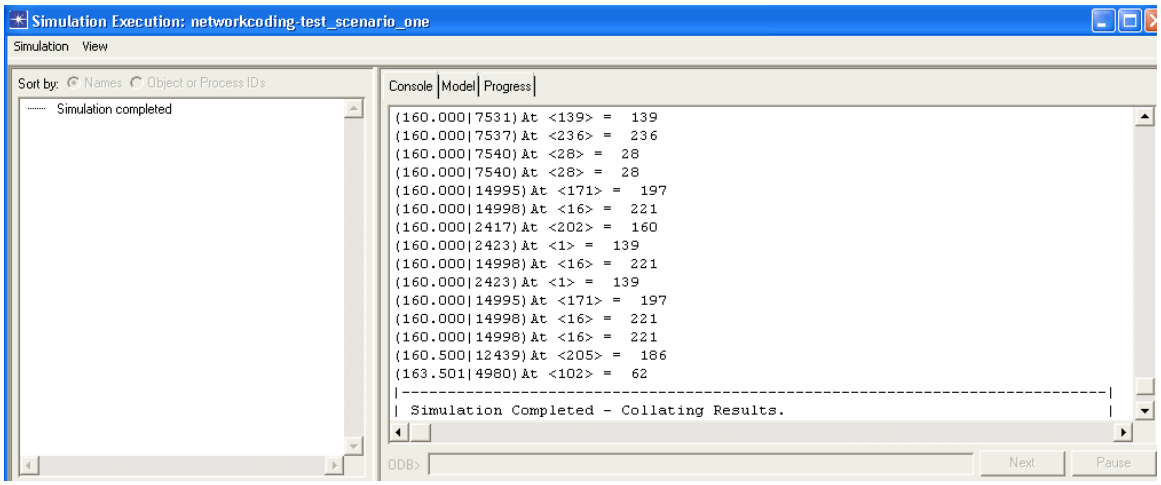

FIGURE 5. Simulation Console showing Successful Network-Coded Packets.

(3) While a UDP traffic demand used within this simulation environment, it would be simple to switch to TCP traffic demands. Because the ACKs are of different size than the data packets, routers would only intercept and process the one-way data flows while letting the ACK flows return in the reverse direction.
(4) It could be helpful to "identify" network-coded packets in another way than by data size. If chosen correctly, the issue of accidentally processing a non-network coded packet as such can be mitigated. If the size to identify these packets is chosen poorly, it will definitely increase the problem.
(5) As the simulation environment stands, once the network-coding topology (ie, the unidirectional topology that the network-coded packets run along) must remain static through the simulation.
(6) The discovery algorithm that would normally occur is not currently implemented.