CIS-553/TCOM-512: Networked Systems
Spring 2013
Project 1 Code Documentation

In this assignment, we will be using the emerging ns-3 discrete network simulator (www.nsnam.org) to teach core principles of network routing protocol design and implementation. Your assignment is to extend ns-3 to support efficient routing using link-state and distance-vector protocols. Your code MUST work on speclab cluster.

The goal of this documentation is to provide you and your group with a starting point for understanding portions of the ns-3 code relevant to this project. An important goal of this project is to provide you the opportunity to read and understand a sizable piece of software and extend it. Hence, we have deliberately not included all the details about the ns-3 code, particularly on specific APIs.

Please be aware that no amount of documentation can replace actual reading and running of the code itself. So rather than spend hours digesting this document without looking at the code, make sure you treat this document as a reference guide while you run the simulator and step through the control flow of various interacting software modules. To get into the habit of working as a team, we encourage you to spend a day or two to get your entire group together to try to understand the code as a team, and help each other out.

All questions regarding the ns-3 code must be posted on Piazza. Do not directly email the ns-3 mailing list (which the TAs are regularly monitoring).

1.0 Getting Started

The TAs developed a version of ns-3.9 release for this assignment. Instead of downloading from the ns3 website, please download the following version instead from Piazza:
ns3proj1.tar.gz

Add this code into your group's svn repository for version control. Refer to the notes from our subversion lecture for commands that will add the code into your group's repository. Once the code is added, other team members can check out the code for development.

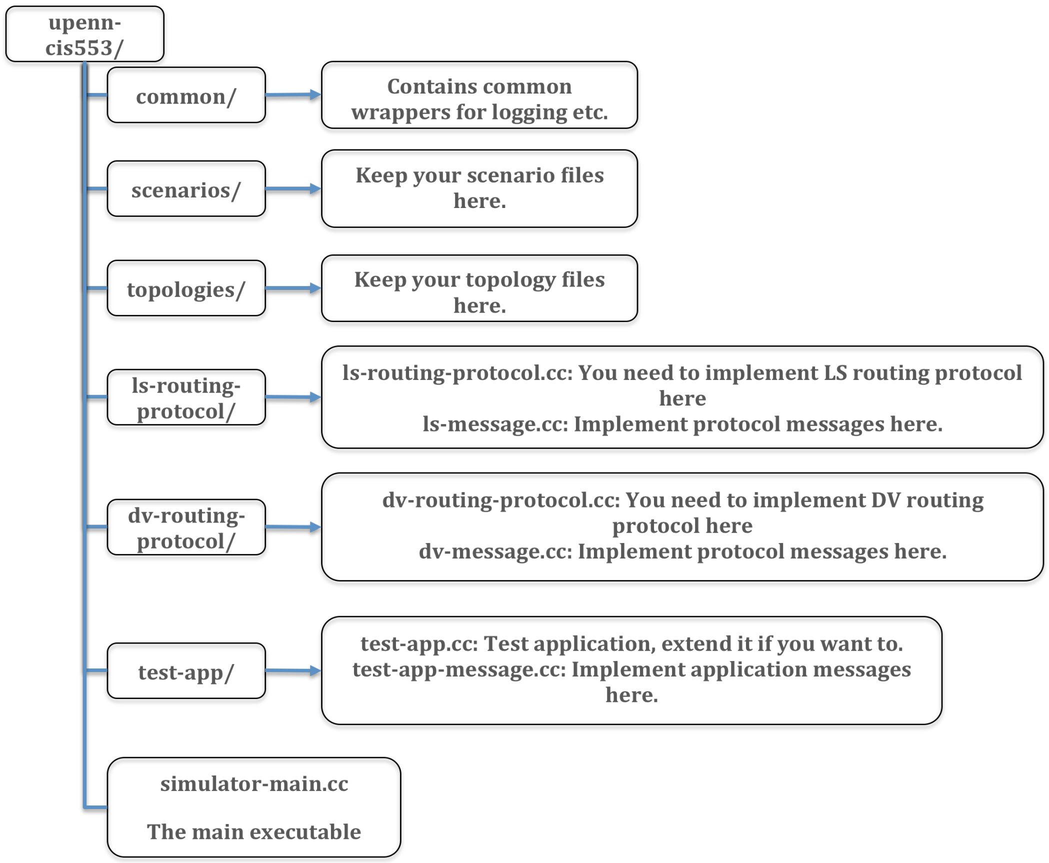
2.0 Code Structure and Compilation

The entire ns-3 code has hundreds of thousands of lines of code. For the purpose of this project, we have isolated all the code that you need to make/learn within the upenn-cis553 folder. If you are running low on disk quota on the seas machines, you can choose to only check the upenn-cis553 folder into the repository. There are extra credits that may require changing files outside this directory, but do not even think about them until you get the regular credits working!!

If you are interested to learn about how our code interacts with the rest of ns-3, you should feel free to investigate further. There are a few useful data structures of ns-3 that you will need to learn along the way.

2.1 Description of files

The upenn-cis553 folder contains the following files:


upenn-cis553/
common/
Contains common wrappers for logging etc.
scenarios/
Keep your scenario files here.
topologies/
Keep your topology files here.
ls-routing-protocol/
ls-routing-protocol.cc: You need to implement LS routing protocol here
ls-message.cc: Implement protocol messages here.
dv-routing-protocol/
dv-routing-protocol.cc: You need to implement DV routing protocol here
dv-message.cc: Implement protocol messages here.
test-app/
test-app.cc: Test application, extend it if you want to.
test-app-message.cc: Implement application messages here.
simulator-main.cc
The main executable

One of the important things to learn from this project is learning how to read and understand APIs within the code, before filling in the actual implementation. We will not provide details of APIs here. Understanding these APIs is part of your project. To help you understand each function, we have added some comments (function headers).

The files you have to modify are:

- **ns-3/upenn-cis553/ls-routing-protocol/ls-routing-protocol.cc:** This file contains all the event handles that you need to implement for handling incoming/outgoing route messages.
    - For each incoming message, you have to implement the logic to handle the message, update local routing tables, and send out outgoing messages.
    - Periodic/triggered link-state updates should be transmitted and handed within this class.
    - Once the routing table is computed, handles for forwarding messages should be implemented here, i.e. given an incoming message, forward the message along the computed next hop to the destination.
- **ns-3/upenn-cis553/ls-routing-protocol/ls-message.cc.** This file implements all the packet formats used in the above file. To implement your link-state protocol, feel free to extend this file to add new packet formats, for instance, "hello" packets for neighbor discovery, and "lsa" for link-state advertisements.
- **ns-3/upenn-cis553/dv-routing-protocol/dv-routing-protocol.cc**
- **ns-3/upenn-cis553/dv-routing-protocol/dv-message.cc**

Distance-vector has a similar set of files in the dv-routing-protocol directory. The functionality of route computation and forwarding has to be implemented as well. We omit discussing this directory since it is similar to that of link-state.

The test application that we provide:

- **ns-3/upenn-cis553/test-app/test-app.cc:** This is an example application that we have provided to test your link-state or distance vector implementation. This application periodically generates a small packet to be transmitted from any source to any destination nodes.Feel free to modify test-app.cc to include your tests. Note that our actual tests during the demonstration may be more stringent than our test-app.cc, which is mostly a sanity check for your implementation. While debugging your code, it is advisable to modify our test-app.cc to do something simpler, for instance, sending just one message from a fixed source to destination.

The inputs to your implementation include:

- **ns-3/upenn-cis553/topologies:** see section 4 on description of input topology. This is essentially the input network to the ns-3 simulation.
- **ns-3/upenn-cis553/scenarios:** see section 5 on description of the scenario file. This is a step-by-step scenario file that you can customize to start/stop a network, bring up/down links, output network state, etc.
- **ns-3/upenn-cis553/results**: see section 6 on how to enable and use auto-checker. It is essentially the sample output for DUMP NEIGHBOR, DUMP ROUTES, and TRAFFIC TRACE. It is an optional argument and we will only use it during demo to speed up testing. Make sure you can pass auto-checker on the sample example before our demo.

Main simulation code:

- **ns-3/upenn-cis553/simulator-main.cc:**This contains the main driver program for your simulation. It takes as input the topology and scenario file, and executes the scenario. In the process, commands from the scenario file are sent to ls-routing-protocol.cc and dv-routing-protocol.cc, for instance, to generate a table dump, etc. Basically, our simulator-main handles basic commands related to link/node topology changes, and redirect routing-related commands to other modules. For instance, routing related commands are sent to ls-routing-protocol.cc and dv-routing-protocol.cc, while test-case specific commands are sent to test-app.cc.This file also generatesoptional outputs traces for animation. **DO NOT MODIFY simulator-main.cc.**

2.2 Compilation

To compile the code, run "./waf" from the main ns-3 directory. Refer to the TA's first lecture on ns-3 for details. **Make sure you set up LD_LIBRARY_PATH correctly**.

You should feel free to add new helper source files, for instance, creating your own classes for route entries, neighbor entries, etc. Whenever you add a new file, you need to modify the wscriptfile within the subdirectory where your new file resides. Failure to do this means that the new file may be excluded from the build.For faster compilation, use "./waf –j 4" to enable parallel compilation on speclabs. Please do not be greedy and increase 4 to a larger number, since that will slow down the speclabs machines and may not speed up your compilation by much!

2.3. Running and interacting with the simulator

Now that you have successfully compiled your code, you can run the simulator as follows:

```
./build/debug/upenn-cis553/simulator-main --inet-topo=<topology-file> --scenario=<scenario-file> --routing=<LS/DV/ANY/NS3> --anim-file=<animation-dump> --result-check=<result-file>
```

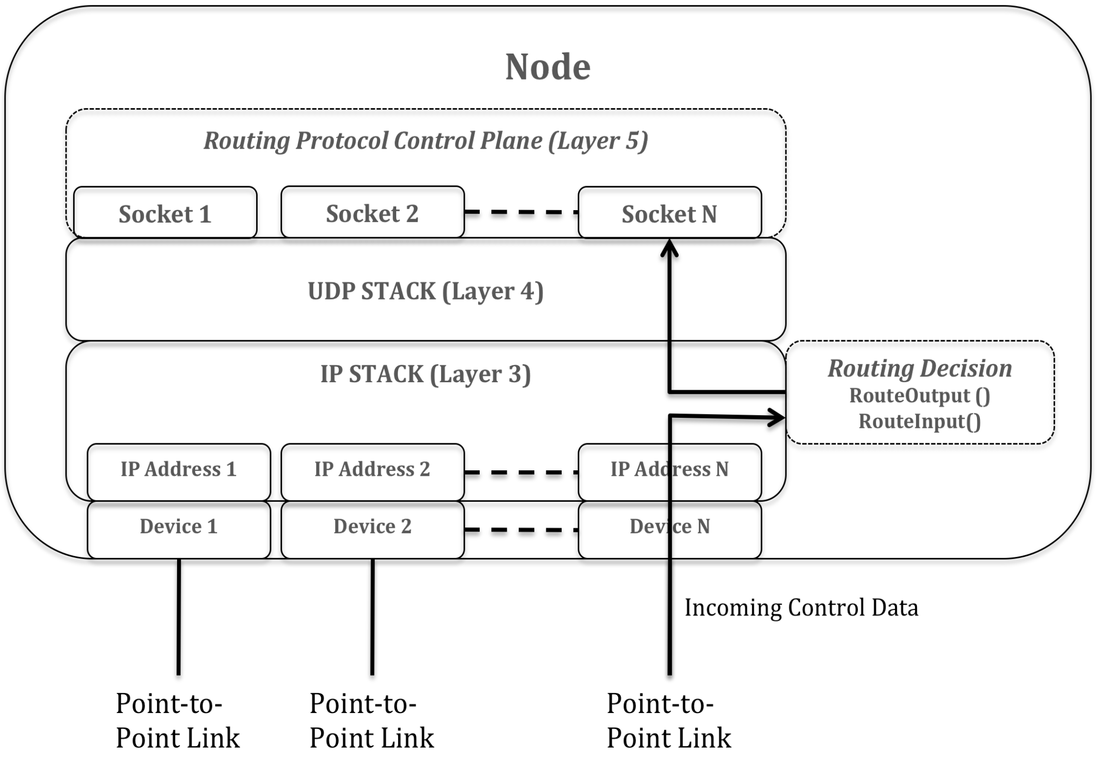
The compiled binaries are located in the build directory. Go to the main ns-3 directory, and run the command above. To understand the commands above, run
"./build/debug/upenn-cis553/simulator-main --help"

The anim-file flag is optional, so feel free to leave it out. The animation software (http://www.nsnam.org/wiki/index.php/NetAnim) cannot run on speclabs without installing extra libraries. However, it is a useful debugging tool at your disposal, and you are free to compile and run the animation software on your own linux desktop where you can compile the animation code yourself. Since we have made the anim-file flag optional, you should omit this flag if you do not intend to use the animation software.If you are using this flag, do note that the file may grow quickly as your simulation runs.

The result-check flag is also optional. You don't need it until you have finished your implementation. But do test your code by auto-checker on the sample input provided before the demo day.

3.0 Overall Architecture

For most part, you do not need to read any additional files beyond those in ns-3/upenn-cis553. However for debugging purposes, it is important to understand the interactions of your code with other parts of ns-3. We provide a high-level overview here.



The figure above shows layers 2-5 from the perspective of a single ns-3 node. At the link layer, each node has multiple IP addresses and interfaces/device (See Section 4 for details). At the IP stack (layer 3), IP packets are forwarded as follows:

- RouteInput() receives a message from one of the devices. Your implementation has to determine the next hop to be taken for this message and forward the message to the appropriate device or the local host (if the destination is itself).
- RouteOutput() takes messages that originated from the local node, and then performs a similar next-hop forwarding or to the local host (if the destination is itself).

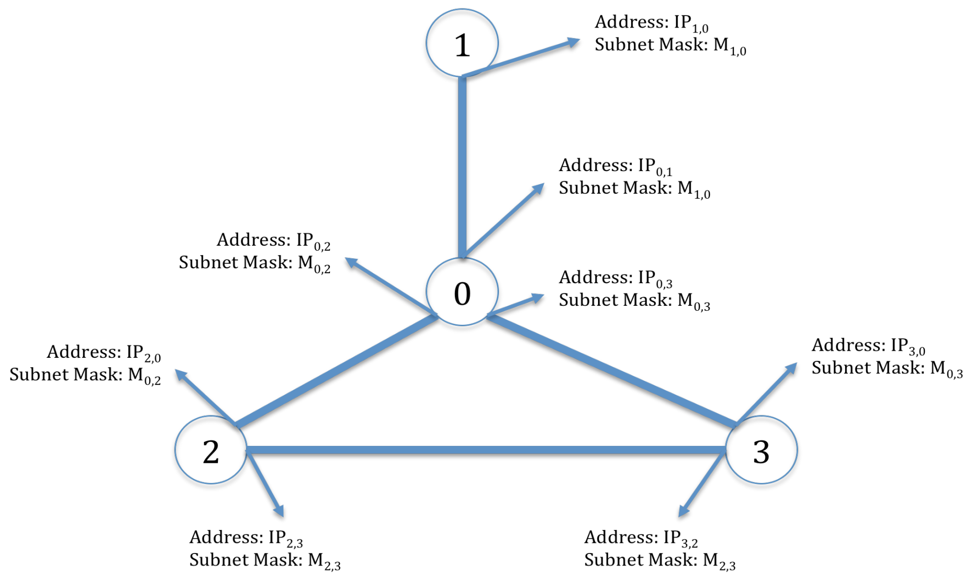
To understand how RouteInput() and RouteOutput() used, consult the OLSR code in src/routing/olsr. You do not need to implement multicast for this project.

Note that all control messages used in link-state and distance-vector are sent via UDP protocol in layer 5 to immediate neighbors. This is *counter-intuitive at first*, since these protocols are implemented in layer 3, while UDP is a layer 4 service. However, this is a common practice to utilize the UDP protocol to bootstrap the protocol itself, where UDP is initially used for communicating with direct neighbors within the same subnet for exchanging control messages. However, once your routing protocol works, you should be able to write an application that uses UDP sockets to send packets to a destination node that is multiple hops away.

4.0 Input Topology

Your simulation has to take in an initial input topology. Our topology uses the Inet topology format (http://topology.eecs.umich.edu/inet/). Before describing the specifics of the topology, we first describe conceptually what the topology looks like.

4.1 Point-to-point topology



The above figure shows an example point-to-point topology. Each node has a set of neighbors, one for each of its interfaces/device. The figure above contains also subnets and masks, which you need not worry about for this project. However, we included them in case you are interested to learn more.

In the topology file, we refer to nodes by node numbers, e.g. 0,1,2… However, once the topology is initialized, our simulator-main assigns to each node multiple IP addresses. The reason why each node requires multiple IP addresses is that it participates in multiple subnets (which you need not worry about). Our simulator-main will select one of the IP addresses as the unique identifier of the node*(hint: look for the m_mainAddress variable in the code).* This identifier is what the node should advertise to its neighbors for computing routes. For instance, if node 0 has IP addresses $IP_{0,1}$, $IP_{0,2}$, and $IP_{0,3}$, $IP_{0,1}$ is selected as the unique identifier. We have also provided APIs that will map from the Inet node number to the corresponding unique IP address (and vice versa). However, we have included this discussion here for your own understanding should you be curious to learn how addressing works in a pt-to-pt network.

4.2 Generating the Inet Topology

To generate the topology, you have to run the topology generator located inside ns-3/inet-3.0. To run the topology, use the following command: ./inet –n <Num>
The command will create a point-to-point topology with Num nodes.

For the actual demonstration, the TAs will be using our own manually generated topology, in addition to running the topology generator. Feel free to also manually generate your own topology. To learn about the Inet topology format, refer to the above website.

To see an example topology, refer to ns-3/upenn-cis553/topologies

5.0 Scenario File

After the simulator has started the network, the scenario file is used to generate network events, such as link failures, node additions/failures, sending messages, dump commands to display network state, etc.
We have provided an example scenario file at ns-3/upenn-cis553/scenarios/test.sce. This scenario file contains most of the commands indicated below. Your goal is to read this scenario file and understand what it is doing. For your own testing purposes, you probably should write your own (simpler) scenarios initially, and feel free to add your own commands (for example, commands to dump link-state updates and network statistics).

Most of the common commands have the following format: <Node Number><Module Name><Command><Arguments>:

- <Node Number> refers to a node in the simulator (from 0,1,2,…).
- <Module Name> refers to the protocol/module, e.g. LS, DV, or the application e.g. APP (test-app.cc). Our simulator-main.cc will direct the command based on the module name to the appropriate code that implements the command handles. (ls-routing-protocol.cc, dv-routing-protocol.cc, or test-app.cc).
- <Command> is the command to be sent to the node, e.g. VERBOSE is to turn on debugging messages, and PING is to send ping messages to another node.
- <Arguments> is any additional arguments required specific to the command.

For instance, the command "1 LS VERBOSE TRAFFIC ON" will result in node 1 turning on all the traffic traces for the link-state protocol. "* LS VERBOSE TRAFFIC ON" will turn on the link-state traces for all nodes.

Command	**Modules**	**Remarks**	**Example**
VERBOSE<TYPE><ON/OFF>	LS, DV, APP	Turn on debugging messages. <TYPE> 1. TRAFFIC: Use this when data is sent/received. 2. ERROR: Use this when error messages are to be printed. 3. DEBUG: Use this to print debug logs. 4. STATUS: Use this to print status messages. 5. ALL: Use this to switch all traces at once. Note: ERROR and STATUS verbose is ON by default	*1 LS VERBOSE TRAFFIC ON* Switches on TRAFFIC traces for node 1.
PING<NODE><MESSAGE>	LS, DV, APP	Send PING to a node. Note that for LS and DV modules, PING can only be sent to immediate neighbors. We have added this functionality as an example for students to implement their neighbor discovery for milestone 1. On the other hand, the APP module PING is multi-hop, which means the PING message ought to be forwarded to the destination node using the routing tables computed by LS or DV.	*1 LS PING 2 hi!* Node 1 sends PING request to node 2 with a message: "hi!" You may also use "*" wildcard with APP module: *1 APP PING * "hello!"* Node 1 sends PING to all the nodes in the topology.

TRAFFIC <START/STOP>	APP	Start sending traffic towards a node. Traffic is nothing but PING packets sent every 2 milliseconds.	*1 APP TRAFFIC START ** Node 1 starts sending traffic to all the nodes in the topology.
NODELINKS <UP/DOWN><NODE NUMBER>	---	Bring up/down all links of a node.	*NODELINKS DOWN 1* Brings down all links on node 1.
LINK <UP/DOWN><NODE-A><NODE-B>	---	Brings up/down all links between NODE A and NODE B	*LINK DOWN 1 8* Bring down link(s) between node 1 and 8.
LINK <UP/DOWN><LINK NUMBER>	---	Bring up/down a specific link as defined in topology file.	*LINK DOWN 6* Bring down 7th link defined in topology file.
TIME <MILLISECONDS>	---	Advance scenario file time before next command.	*TIME 100* Advance time by 100 milliseconds.
TIME	---	Display the current simulator time	
QUIT	---	Quit Simulator	

Note that the commands you need to implement for our actual demo are as follows:

Command	**Remarks**	**Example**
DUMP ROUTES	Print out routing table	*1 LS DUMP ROUTES*
DUMP NEIGHBORS	Print out neighbors	*1 LS DUMP NEIGHBORS*

There is a similar set of commands for DV (replace "LS" with "DV").

5.1 Interactive Scenario Mode

In addition to using the scenario file, you can also enter the scenario commands interactively via the keyboard. While the simulation is running, there is a command prompt that you can use to enter the commands described above.

5.2 Real-time Simulation (Wall-clock mode)

In the normal ns-3 operation, the simulation jumps from one event to the next, even if the events are several seconds apart. You can also run simulator-main in the wall-clock mode if it helps in your debugging. In this mode, the simulator would schedule events with reference to real clock. Thus, if any two events were scheduled 10 seconds apart, it would actually feel like 10 seconds when they are executed. This mode can be enabled by passing "--real-time=yes" flag to the simulator-main.

6. Auto-checker Mode

At some point before your actual demonstration, we would be providing you with a sample topology/scenario and a result-check file. The auto-checker mode can be invoked by providing the given topology/scenario files, together with the result-check file by using the option "--result-check=<filename>" to the simulator-main. Our auto-checker would then compare your implementation's output with our reference implementation's output (provided in result-check file). To make this comparison work, you have to call two "hook" methods in your code. If you look into *DumpNeighbors()* and *DumpRoutingTable()* functions, you will find commented function calls to *checkNeighborTableEntry(...)* and *checkRouteTableEntry(...)* respectively. After you get your implementation running, uncomment these function calls and pass on the required arguments (declarations are in *upenn/test-result.h*). Once these hooks are properly setup, the auto-checker would pinpoint the inconsistencies between your implementation and the reference implementation, if any.

Please note that having your implementation match our result-check file is the bare minimum that you should aim to achieve. It does not ensure that your implementation would pass all the advanced tests on the demonstration day. We are providing you this file so that you can be sure that your implementation is on track.