Project 3
Overlay Routing & Distributed Transaction Processing

CMSC 417 Fall 2016

Updated: October 19 2016

1 Deadline

This project will comprise multiple parts. Each part will have a separate due date.

2 Introduction

In this project, you will apply what you have learned throughout the semester to build an overlay routing system that uses Link State routing to pass messages between arbitrary nodes.

Overlay routing is routing that happens at the **application layer**. The application layer processes on a given node, connect to application layer processes on other nodes to build routing tables and pass messages.

This project is intended to be "open ended" in the sense that we leave many of design decisions for you to make. We want to see that you possess the ability to not only design and implement a system, but to defend your design decisions. As such, your grade will be based not only on your implementation but on a writeup that you will turn in at the end of the project. Do not underestimate the difficulty of this project.

Environment

You will run this project in the CORE network environment. We will provide network configurations for you to use. The basis of this project is that you have several groups of nodes which are directly connected. When you load the network configuration, you should notice (and test this), nodes that are directly connected to each can ping each other. However, nodes that are not directly connected to each other cannot ping each other. Why is this? The short answer is that they don't know how.

Background

Let's discuss some network routing background (really this should be just a refresher; and will be presented at a high-level). When you connect your computer to your home network, what really happens? The router that you get from your ISP (really, it's a NAT Box), connects to the next hop router.

EXAMPLE: Open up a command line and run a traceroute to www.cs.umd.edu. On windows you can use *tracert www.cs.umd.edu* and on Unix (Linux or Mac) it should be *traceroute www.cs.umd.edu*

In the example, you'll see all of the intermediate routers that the packets you sent from your machine to my server took. The first entry is the router (or access point) you are connected to. If you're on a home or residential network, chances are the first hop is just a consumer grade NAT Box (it doesn't really know how to route to much). Basically, all that router knows how to handle is: if traffic is for the local network, route it to that node, otherwise, just send it to the ISP and let the ISP handle it. Those are the basics of routing.

Going back to the question in the CORE environment, why can't nodes that aren't directly connected ping each other? The longer answer is that there's not a routing protocol running on the nodes to tell them what to do with the traffic. Nodes basically know the identity of the node on the other end of a connection, but nothing beyond that.

What you will be doing

Your task is to implement an application layer protocol using the fact that nodes know how to communicate with their direct neighbors. Using this simple fact, you will enable communication between arbitrary nodes. Your code will run at the application layer. Instances applications running on different nodes will be able to communicate with instances running on other nodes. Put another way, instances of your application will be able to communicate across nodes.

What you will NOT be doing

You are **NOT** writing a generalized routing protocol to facilitate arbitrary data transfer. That is to say, even after your project is finished, arbitrary nodes will not be able to ping each other using the Unix command "ping." However, they will be able to send "ping messages" to each other through your application.

3 Team Project

This project will be done in groups of 2. A group of 1 student may work with a group of 2 students. However, each group is responsible for making his or her own submissions. All students must work on all aspects of the project. **ALL STUDENTS ARE RESPONSIBLE FOR MAINTAINING AND SUBMITTING THEIR OWN CODEBASES**

4 Requirements

1. You must implement this project in Ruby. In this past, this project has been done in C, but the reality is that just getting the C code to work correctly consumes a huge amount of time. The goal of this project is to help you design protocols and design complete distributed system, not to debug C code.

2. You may **NOT** use any specialized libraries. (By this, we mean that we don't want you to pull an entire implementation of Dijkstra's algorithm from the Internet)

3. You **MUST** use CORE for this project.
4. You may **NOT** use RPC (Remote Procedure Calls) for any part of this project.
5. Nodes must communication using only network connections; i.e. they may not communicate via config files, etc.
6. Each node should keep it's own internal clock; at the beginning of execution, each node may make a single call to Ruby's *Time* library. Each node should store the time in a local variable, and keep the variable up to date. You may use a timer to regularly update the variable, but you may NOT get the time from the OS again.

5 Time Table

1. Part 0: Routing Table: Nov 8, 2015
2. Part 1: Routing Core: Nov 15, 2015
3. Part 2: Messages 1: Nov 29, 2015
4. Part 3: Messages 2: Dec 6th, 2015
5. Part 4: Write up Due last day of class

Parts

Part 0: Routing Table

The first step of this project is to build the routing tables. In this part, you will enable your nodes to connect to each other and exchange edge information You will need to make a data structure to hold information about routing tables. At this point, nodes should only know information about their "next hop" neighbors.

For Part 0, you will need to have the following methods implemented (see below for a description)

- EDGEB
- DUMPTABLE
- SHUTDOWN

Part 1: Routing Core

In this part, you will enable nodes to send and receive link state updates from neighbors to obtain a full picture of the network Nodes will receive updates from the console and build edges accordingly. Nodes should run an algorithm such as Dijkstra (or Bellman-Ford if you're feeling creative).

For Part 1, you will need to have the following methods implemented (see below for a description)

- EDGED
- EDGEU
- STATUS

Part 2: Messages 1

In this part, you will enable nodes to send and receive messages. While many of these messages resemble protocols that exist in practice today, you are free to design them as you see fit. Your grade will partly be based on how well you design these protocols. In your final report, you will be asked to justify all of the decisions that you made, so make sure that you can not only recognize what you are doing, but why.

For Part 2, you will need to have the following methods implemented (see below for a description)

- SENDMSG
- PING
- TRACEROUTE
- FTP

Part 3: Messages 2

In this part, you will enable nodes to communicate via circuits. Circuit switching is different than packing switching in that all parts of the message follow the same circuit. **NOTE: CAREFULLY PLAN TO SUPPORT THIS PROTOCOL AS FAILURE TO DO SO MAY RESULT IN HAVING TO RE-DESIGN A SUBSTANTIAL PORTION OF YOUR CODE** For Part 3: you will need to have the following methods implemented (see below for a description)

- CIRCUIT

Configuration

Message Formats

API

Your application should ultimately act as a server to the outside world. That is, it should listen for connections from other nodes and handle them accordingly. End nodes need to be able to accept messages from the end user (as well as deliver messages to the user) You should use STDIN and STDOUT for this.

Console Operation

commands will be given in the form of: *command [args]* Here are some examples of messages:

- SENDMSG N1 ”Hello World” : *send the message “Hello World” to the process running on N1*
- PING N2 3 5: *send 3 ping control messages to the process running on N2, separated by 5 seconds*

Commands: Descriptions

The following section describes the input, output and formatting of the commands

DUMPTABLE

Format: DUMPTABLE [FILENAME]

When a node receives a DUMPTABLE command from the console, it should write its current view of the routing table as a CSV (with NO headers) to the file specified by FILENAME. Each row of the table should be formatted as follows: *src,dst,nextHop,distance*

SHUTDOWN

Format: SHUTDOWN

This should cleanly shutdown the node and flush all pending write buffers (stdout, files, stderr). The node should exit with status 0

STATUS

Format: STATUS

The node should print out the following status information, formatted accordingly:
Name: ¡nodename¿ Port: ¡port the node is listening on¿ Neighbors: ¡lexicographically sorted list of neighbors, separated with commas and no spaces¿

EDGEB

- **Format**: [EDGEB] [SRCIP] [DSTIP] [DST]
- **Description**: This method creates a **symmetric** edge between the node on which the command is run, and the node specified by DST. By symmetric, we mean that the node specified by DST should have the reverse edge in its routing table. The cost of the edge should be initialized to 1. SRCIP and DSTIP are given to facilitate the initial connection between the nodes. This will enable your edges to be build without the need for address resolution (as is the case in NRL's CORE).

HINTS

- It is usually simplest to keep a TCP connection open between nodes to simulate a "connection"

EDGEU

- **Format**: [EDGEU] [DST] [COST]
- **Description**: This method updates the cost of the link from the current node to the neighbor node specified by DST.

HINTS

- COST will be a valid 32 bit integer (the cost between any two nodes can be represented by an integer)

- This method is only defined when DST is a next hop neighbor of the current node.
- This method should produce no output if any of the fields are invalid or missing

EDGED

- **Format**: [EDGEU] [DST]
- **Description**: This method destroys the edge from the source node to the dst node (i.e. removes all state information). It does NOT need to cause any changes on the DST node. Simply, it should be run from each node.

HINTS

- Will be run once from each node

SNDMSG

- **Format**: SNDMSG [DST] [MSG]
- **Description**: This method will deliver the string MSG to the process running on DST.
- **Output (Source) Success**: <nothing>
- **Output (Source) Failure** : SENDMSG ERROR: HOST UNREACHABLE
  1. The source should print this message if it is unable to deliver the entire message to the DST.
- **Output (Destination Success)**: SENDMSG: [SRC] −− > [MSG]
- **Output (Destination Failure)**: <nothing>

PING

- **Format**: PING [DST] [NUMPINGS] [DELAY]
- **Description**: This method will send NUMPINGS ping messages to the DST. There should be a delay of DELAY seconds between pings.
- **Output (Source) Success**: [SEQ ID] [TARGET NODE] [ROUND TRIP TIME]
  1. SEQ ID should start at 0 and increase by one every round
  2. Subsequent calls to ping should start at 0
  3. Use the ping timeout option found in the config file
  4. If a response to a lower number ping is received first, you may print it immediately.
  5. For an example open a terminal window and type: ping www.cs.umd.edu
- **Output (Source) Failure** : PING ERROR: HOST UNREACHABLE

1. A failure is considered to have occurred if a response to a ping is not heard within the ping timeout interval.
2. The failure of one ping message should not affect other the other NUMPINGS-1 messages.

- **Output (Destination Success**: <nothing>
- **Output (Destination Failure**: <nothing>

TRACEROUTE

- **Format**: TRACEROUTE [DST]
- **Description**: This method will perform traceroute from the SRC to the DST
- **Output (Source) Success**: [HOPCOUNT] [HOSTID] [TIME TO NODE]
  1. There should be one line of output for each node in the path from SRC to DST
  2. The SRC node should be included in the output
  3. The SRC node is considered HOPCOUNT of 0
  4. The SRC node *must* print the list sorted by HOPCOUNT; do not print lines out of order
- **Output (Source) Failure** : [TIMEOUT] ON [HOPCOUNT]
  1. A timeout is considered to have occured if a node does not reply within the ping timeout interval specified in the config file.
- **Output (Destination Success)**: <nothing>
- **Output (Destination Failure)**: <nothing>

FTP

- **Format**: FTP [DST] [FILE] [FPATH]
- **Description**: This method will transfer the file specified by the FILE argument to the node specified by the destination argument. The DST node should store the file (with the same file name) to the location specified by FPATH (FPATH is a directory, do NOT include the trailing slash)
- **Output (Source) Success**: FTP [FILE] $--$ > [DST] in [TIME] at [SPEED]
  1. TIME is the amount of time (in seconds) since the file transfer began
  2. SPEED is calculated by dividing the SIZE (in bytes) of the FILE by TIME and rounding down to the nearest integer (i.e. take floor of SIZE/TIME)
- **Output (Source) Failure** : FTP ERROR: [FILE] $--$ > [DST] INTERRUPTED AFTER [BYTECOUNT]
  1. BYTECOUNT is the number of bytes that were successfully transferred.

2. This message should be displayed for any type of error relating to the FTP
3. Neither the SRC nor the DST should attempt to recover from FTP errors
4. If an error occurs, both SRC and DST should clean up internal state.

- **Output (Destination Success)**: FTP: [SRC] $--$ > [FPATH]/[FILE]
- **Output (Destination Failure)**: FTP ERROR: [SRC] $--$ > [FPATH]/[FILE]

CIRCUITB

- **Format**: CIRCUITB [CIRCUITID] [DST] [CIRCUIT]
- **Description**: This method will build a named circuit from the current node, to the DST node along to the comma separated list of CIRCUIT (Note: this list is a list of node names, with no spaces, separated by commas). This circuit may contain loops. CIRCUITID is a string given by the user to uniquely identify the circuit. For simplicity, assume only circuit names are unique in the network. For additional simplicity, assume that the n+1^th node in the circuit is a direct neighbor of the n^th node. Do not include the source or destination in the CIRCUIT string.
- **Output (Source) Success**: CIRCUITB [CIRCUITID] $--$ > [DST] over [HOPS]
  1. CIRCUITID is the circuit ID
  2. HOPS is the number of intermediate nodes (i.e. not counting source or destination)
- **Output (Source) Failure** : CIRCUIT ERROR: [SRC] $-/-$ > [DST] FAILED AT [FNODE]
  1. SRC is the node from which the circuit originates
  2. FNODE is the first node in the circuit that cannot be reached
- **Output (Destination Success)**: CIRCUIT [SRC]/[CIRCUITID] $--$ > [DST] over [HOPS]
- **Output (Destination Failure)**: <nothing>

This method should fail if any intermediate node already has information for a circuit with a given circuitid. Simply, only one circuit can go through a node with a given circuit ID. The circuit must be destroyed before the name can be reused.

CIRCUITM

- **Format**: CIRCUITM [CIRCUITID] [MSG]
- **Description**: This method will send a MSG over the circuit. All message types are supported over this cicuit (PING, SNDMSG, etc), except for CIRCUITB, CIRCUITM, CIRCUITD.
- **Output**:
  1. The output for success and failure is the same as for the previous messages, except that it should begin with CIRCUIT [CIRCUITID]/
     EXAMPLE: CIRCUIT [CIRCUITID]/FTP [FILE] $--$ > [DST] in [TIME] at [SPEED]

CIRCUITD

- **Format**: CIRCUITD [CIRCUITID]
- **Description**: This method will destroy the circuit specified by CIRCUITID and remove all information from intermediate nodes.
- **Output (Source) Success**: CIRCUITD [CIRCUITID] $--->$ [DST] over [HOPS]
  1. CIRCUITID is the circuit ID
  2. HOPS is the number of intermediate nodes (i.e. not counting source or destination) from which the circuit was removed
- **Output (Source) Failure** : CIRCUIT ERROR: [SRC] $-/->$ [DST] FAILED AT [FNODE]
  1. SRC is the node from which the circuit originates
  2. FNODE is the first node in the circuit that cannot be destroyed
- **Output (Destination Success)**: <nothing>
- **Output (Destination Failure)**: <nothing>

This method should fail if any intermediate node already has information for a circuit with a given circuitid. Simply, only one circuit can go through a node with a given circuit ID. The circuit must be destroyed before the name can be reused.

Design

Your will need to design control messages to help carry traffic. **Note**: even though these are application layer messages, they are analogous to IP Layer packets. You will need to decide which fields to include in your control messages. This should be clearly specified in your writeup; think carefully about what fields to include. Using the correct fields will make this project substantially easier.

Getting Started

Controller

In order to facilitate testing, we have given you a file controller.rb. The controller reads in a node configuration file, and starts a node application for each node specified. The controller then allows the user to interact with the nodes.

For simplicity, the controller runs all nodes on the same machine. This is OK for testing, but your code ultimately needs to run in CORE, on different nodes.

node.rb

To get you started, we have given you some skeleton code in node.rb. You are free to use this as a basis for starting your program, or you can start over entirely. However you decide to proceed, you should look at this code to obtain an idea of how the code runs.

The First Steps

These steps should enable you to get a skeleton version of the nodes working.

Running a Node

1. The first step is to run a node; the starter code for the node is contained in node.rb. You can run this code by tying ruby node.rb [HOSTNAME] [PORTNO] [NODESFILE] [CONFIGFILE]. HOSTNAME is the assigned name for the node; it is analogous to a DNS record. PORTNO is the port that the node should listen for incoming connections (if you are running all nodes on the same machine, each node must use a different port number. why?) NODESFILE is the file specifying which port each other node in the simulation is running on, and CONFIGFILE is the list of options for the node. For simplicity, start the first node
   ruby node.rb n1 10241 nodes config
   You now have a single instance of the node running on your local machine.

2. You should now be able to interact with the node using your console input; you can give the node commands such as SENDMSG n1 Hello, but the node will just give a message that the method is not yet implemented.

3. You can run other nodes in the same way. After you have implemented the first part of the project, you should be able to have two nodes start, connect to each other, and exchange edge information.

Running the Controller

To simplify testing, we have given you a controller to automatically start and interact with several instances of the nodes on the same machine.

1. The controller is started with by running ruby controller.rb [NODES] [CONFIGFILE]

2. The controller will start an instance of each node in the NODES file, pass the appropriate NODES file name and CONFIGFILE to each of the nodes.

3. Additionally, the controller will maintain a pipe to STDIN on all of the nodes

4. Finally, the controller will redirect the output of the nodes to a text file console_[nodename]

Controller Commands

To give commands to the controller type CONTROLLER and then the command
At the time of this writing, the controller supports the following commands

- SHUTDOWN - tells the controller to issue the shutdown command to all of the nodes. This will hang if the nodes do not implement shutdown.

- SLEEP [SECONDS] - tells the controller to sleep for SECONDS seconds. This is useful if you want to issue commands to the controller via i/o redirection. Example *cat commands.txt — ruby controller.rb nodes config*

Node Commands

To give commands to a node through the controller, type NODE [nodename] and then the command

Example Commands

The following is an example of a simulation using the controller. Hint: it is the first public test

```
NODE n1 EDGEB 127.0.0.1 127.0.0.1 n2
NODE n1 DUMPTABLE ./t1_n1_dumptable.txt
NODE n2 DUMPTABLE ./t1_n2_dumptable.txt
CONTROLLER SLEEP 1
CONTROLLER SHUTDOWN
```

Files

Config File

- Each line in the config file should be formatted as *option=value*, where option corresponds to one of the options below, and value is the assigned value.
- Each option, value pair should be on a separate line
- Do not include headers or any other data

Config File Options

Your config file must support the following options. In order for your code to work with our grading scripts, please ensure that you name the options exactly as they appear below:

- **updateInterval**: Specifies (in seconds) how often routing updates should occur
- **maxPayload**: Specifies (in bytes) the maximum payload size for a message.
- **pingTimeout**: Specifies (in seconds) how long a node should wait for a reply before considering the packet to be timed out. This timeout should be used for all message types.

FAQ

The following section addresses questions I have frequently been asked in the previous years of teaching this project.

Do I have to support fragmentation?
Yes, your implementation MUST support fragmentation

Should I Use Multithreading?
Yes, but think carefully about where to use it. It can quickly make your task much more difficult than it needs to be.

Does ordering output matter?
The order of the output does not matter, but you cannot interleave chunks of a message. For exmaple, it's fine if your pings come back out of order, but you can't have a ping message printed out betweeen two chunks of a single SENDMSG

Can I block on the console?
No, once a command is given, the console should be ready for more input

Should I print errors to STDERR?
No, print all console messages to STDOUT

Technical Specifications

The following are a collection of technical nuances particular to this version of the project

- You will need to explicitly refer to ruby's STDIN and STDOUT and STDERR when printing and reading. For example: STDIN.gets() STDOUT.puts
- Node names will not contain spaces
- The node should bind to all interfaces