Programming Assignment 2: Routing in a Virtual Network

Due: November 7th, 2017 23:59:59pm

This assignment is worth **11%** of your total score in this course. In this assignment, you will build a virtual network and implement routing for this network. Each node in this network is a process, and links in this network are emulated by UDP sockets. You can use **C or C++** to implement this assignment.

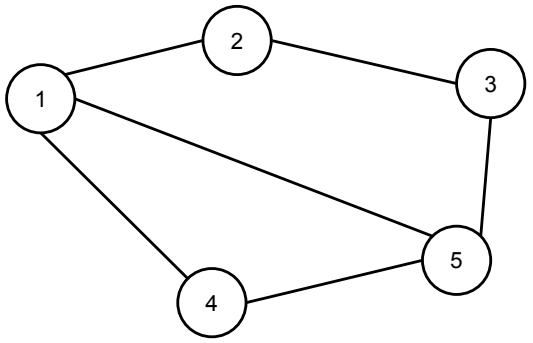
This is a **group assignment**. Each group should have **exactly two** students. A group should have either both undergraduate students or both graduate students to allow each of our two TAs to focus on grading only graduate or undergraduate students.

Though strongly discouraged, you can choose to work by yourself on this assignment. No extra credit will be given for working individually.

Once you have formed a group, at least one member in the group **must** send the instructor an email listing the names and email addresses of both members of your group by **October 27th, 2017** by the end of day. If we do not receive your email by this deadline, we will record that you will be working individually. No groups are allowed to be formed after the October 27th deadline.

1 Virtual Network Topology

In your virtual network, each node/host is a process running on a machine. Each process is identified by the hostname of the machine it runs on and a port number on the host. We then use a configuration file to specify the virtual network topology. The following figure shows an example network topology.



We can create a configuration file as follows:

//Node ID	hostname	control port	data port	neighbor 1	neighbor 2	neighbor 3 ...
1	remote00.cs.binghamton.edu	5000	5001	2	4	5
2	remote00.cs.binghamton.edu	5002	5003	1	3	
3	remote01.cs.binghamton.edu	5000	5001	2	5	
4	remote02.cs.binghamton.edu	5002	5003	1	5	
5	remote03.cs.binghamton.edu	5004	5005	1	3	4

Each line of the configuration files provides information about the node's ID, which is a unique integer, hostname, port numbers, and ID of its directly connected neighbors. Each node runs on two ports. One port is the

control port, used for exchanging routing information (control packets) with directly connected neighbors. The other port is the data port, used for data packet sending, receiving, and forwarding.

At virtual network start time, each node will read this configuration file to determine the hostnames and port numbers of its directly connected neighbors. For example, node 1 will learn its directly connected neighbors are nodes 2, 4, and 5. Nodes cannot use any information about nodes other than their direct neighbors. For example, from the configuration above, node 1 should not learn anything about node 3.

2 Distance Vector Routing

Nodes on this virtual network use distance vector routing for determining the shortest path to other nodes in the network.

Each node periodically sends its distance vector to all its directly connected neighbors. For example, node 1 will send its distance vector to nodes 2, 4, and 5. Each distance vector entry contains `(Destination, Next hop, Distance)` - to send a packet to destination `Destination`, the packet should first be forwarded to next hop `Next hop`, and the total number of hops to the destination is `Distance`.

Each node also listens for incoming distance vectors sent by its neighbors. Whenever a new distance vector is received, the node will follow the algorithm we discussed in class (lecture 8, slides 10 and 11) and update its **routing table** accordingly.

Routing messages, i.e., distance vectors, should be exchanged using the node's control port (e.g., node 1's control port is port 5000). Within each node, you can use a dedicated "control thread" to handle the sending and receiving of all routing messages through a `select()` call. (We have used the select() call in assignment 1).

Routing messages will be transmitted between nodes using UDP. To create a UDP socket, pass `SOCK_DGRAM` as the second parameter to the socket() call. However, since UDP is connectionless, a UDP socket does not need to `listen()` or `accept()` a connection.

To receive UDP messages, we use the `recvfrom()` call:

```
ssize_t recvfrom (int socket, void *buffer, size_t size, int flags,
                  struct sockaddr *addr, socklen_t *length-ptr)
```

According to the libc manual[1]:

"The `recvfrom` function reads one packet from the socket `socket` into the buffer `buffer`. The `size` argument specifies the maximum number of bytes to be read.

If the packet is longer than `size` bytes, then you get the first `size` bytes of the packet and the rest of the packet is lost. There's no way to read the rest of the packet. Thus, when you use a packet protocol, you must always know how long a packet to expect.

The `addr` and `length-ptr` arguments are used to return the address where the packet came from. "

The receiver can determine the sender of the UDP packet by inspecting the `addr` field. For this assignment, you should use the `addr` field to determine which source node sent a received distance vector.

To send UDP messages, the `sendto` call is used.

```
ssize_t sendto (int socket, const void *buffer, size_t size, int flags,
                struct sockaddr *addr, socklen_t length)
```

Accordng to the libc manual[2]:

"The `sendto` function transmits the data in the `buffer` through the socket `socket` to the destination address specified by the `addr` and `length` arguments. The `size` argument specifies the number of bytes to be transmitted."

[1] `https://www.gnu.org/software/libc/manual/html_node/Receiving-Datagrams.html`
[2] `https://www.gnu.org/software/libc/manual/html_node/Sending-Datagrams.html`

3 Data Packet Forwarding

Each node should use a separate thread, the "data thread", for sending, receiving, and forwarding data packets. Such "data traffic" goes through the data port, e.g., node 1's data port is port 5001.

You are free to design your data packet format. But your packet should contain at least the following information:

```
8-bit Source node id
8-bit Destination node id
8-bit Packet id
8-bit TTL
Data
```

`Source node id` and `Destination node id` are the sender and receiver's node ids. `Packet id` should be generated by the sender in monotonically increasing order. `TTL` should be set to 15 initially by the sender. Every time a packet is forwarded by an intermediate node, the `TTL` should be decremented by 1. A packet should be destroyed if its `TTL` is 0 (e.g., not forwarded). `TTL` is used for preventing routing loops in the virtual network. `Data` should be the packet's payload. Try to limit `Data` size to be within 1 KB.

The data thread should print debug information (e.g., the packet header) whenever it generates a new packet, receives a packet, drops a packet or forwards a packet.

For testing purposes, when a data thread receives a data packet, it checks if its id is the destination node id of this packet. If so, the node should display this packet's information as well as the complete forwarding path of this packet on the console. This complete forwarding path can be recorded by having each intermediate node add its own node id in the packet's data field. If the packet's destination node id is not the id of the current node, the data thread should forward the packet to appropriate next hop node in the routing table and print debug information on the console.

Because both the data thread and the control thread read/write the routing table, accesses of the routing table should be protected by an exclusive lock.

4 A Control and Testing Client Program

You will have to show that your routing and data forwarding operates according to the specification described in the previous sections.

To do so, you will also write a client program to control nodes' network links and to instruct nodes to send new data packets. This client will accept command line inputs and send corresponding control commands to each node via their "control port" individually. These control commands include:

1. ask a node to generate a new data packet and send it to a destination node. For example, if the command line argument says "`generate-packet 1 2`", then your control client should send a message to node 1 to create a packet destined to node 2. When node 1 receives this message, it will create a data packet and send it to node 2, potentially via other intermediate nodes.

2. add a new link. For example, if the command line argument says "`create-link 1 2`", then your control client should send messages to node 1 and node 2, respectively, indicating a new link has been established. These nodes will then exchange their distance vectors and update their routing tables correspondingly.

3. remove an existing link. For example, if the command line argument says "`remove-link 1 2`", then your control client should send messages to node 1 and node 2, respectively, indicating a link has been removed. These nodes will immediately update their routing tables and send them to their neighbors.

Because the "control thread" has to handle these different types of messages as well as distance vector messages, you need to define these message formats and include a special "type" field in the packet header to allow the receiver node to understand how to interpret these client control packets.

Once the network topology has been changed, e.g., via create-link or remove-link, you should show that your virtual network sends distance vector messages appropriately, nodes' routing tables are changed correspondingly, and packets follow new paths to their destinations. For simplicity, you can assume that the set of nodes in the virtual network remain unchanged.

5 Demonstration

After the submission deadline, every group will sign up for a demonstration time slot with the TA. Both group members have to be present during the demonstration. You will show the capabilities of the network nodes that your group has implemented, such as routing, data packet forwarding, and how your virtual network adapts to network topology changes.

6 Other Helpful Information

Each node will create at least two threads: the control thread and the data thread. You will use the **pthread**[3] library for creating and managing these threads.

If you are not familiar the **pthread** library, you can read the following link to learn about how to use it: `http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html`

7 How to submit

To submit the assignment, you should first create a directory containing your BU email IDs. For example, if a group's two students' email IDs are `jdoe@binghamton.edu` and `jsmith@binghamton.edu`, you should create a directory called `jdoe-jsmith-p2`.

You should put the following files into this directory:

1. Your source code.
2. A `Makefile` to compile your source code into executables.
3. (Optional) A `Readme` file if there is anything you want the TA to be aware of when grading.
4. A `Task` file, listing the tasks both group members worked on in this assignment.
5. Two `STATEMENT` files, signed by each group member individually, containing the following statement:

   "I have done this assignment completely on my own and in collaboration with my partner. I have not copied my portion of the assignment, nor have I given the project solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of **0** for the involved assignment and my grade will be reduced by one level (e.g., from A to A- or from B+ to B) for my first offense, and that I will receive a grade of **"F" for the course** for any additional offense of any kind."

Compress the directory (e.g., `tar czvf jdoe-jsmith-p2.tgz jdoe-jsmith-p2`) and submit the tar-ball (in `tar.gz` or `tgz` format) to myCourses. You should name your submission: `jdoe-jsmith-p2.tar.gz` or `jdoe-jsmith-p2.tgz`. Failure to use the right file name will result in a 5% point deduction.

Your assignment will be graded on the CS Department computers `remote.cs.binghamton.edu`. It is your responsibility to make sure that your code compiles and runs correctly on these remoteXX computers.

[3] `http://man7.org/linux/man-pages/man7/pthreads.7.html`

Your assignment must be your original work. We will use MOSS[4] to detect plagiarism in programming assignments.

[4] https://theory.stanford.edu/~aiken/moss/