Computer Networking, Fall 2021

Assignment 3: Routing Simulation

2021.11.20

1 Introduction

In this assignment, you will implement the distance vector routing protocol. You are asked to writing two programs: router and agent. A router program represents a router process. You may need to invoke multiple router programs. On the other hand, the agent program is a centralized control agent that triggers the route updates of the routers.

2 Protocol Description

2.1 Configuration files

We need to prepare two configuration files in advance before we start the router and agent programs. They are the router location file and the topology configuration file.

2.1.1 Router location file

The router location file maps a unique router ID to a router process. Each router is defined by a 3-tuple:

```
<router_IP,router_port,router_id>
```

Each router is assigned a unique identifier *router_id*. It runs on IP *router_IP*, and listens on port *router_port* for two pieces of information: (i) commands from the agent and (ii) messages from other routers. For example, the router location file can be represented as follows:

```
6
137.189.88.101,13001,1
```

```
137.189.88.101,13002,2
137.189.88.101,13003,3
137.189.88.102,13016,4
137.189.88.102,13007,5
137.189.88.102,13005,100
```

Both the routers and the agent need to parse the router location file. The first line denotes the number of routers in the network, while the following lines define the configuration of each router. Note that the router IDs may not be consecutive. For instance, it is possible to have a network composed of routers with IDs 1, 2, and 4 without the ID 3. For simplicity, we make the following assumptions:

- The router location file is error-free, in the sense that (i) there is no formatting error, (ii) the router ID in each row is unique, (iii) there is no conflict of router port numbers, and (iv) the IP addresses and port numbers are actually used by the corresponding router (the details of how we invoke a router are discussed later).
- Router IDs are all integers.
- The network has at most 10 routers.

2.1.2 Topology configuration file

The topology configuration file specifies the router topology. Here, the distance between two neighboring routers is bidirectional, i.e., the distance from router 1 to router 2 is *always* the same from the distance from router 2 to router 1 (either before update or after update). The topology configuration file is a text file composed of a number of topology tuples. The format of a topology tuple is shown below:

```
<RouterID1,RouterID2,weight>
```

Each term is an integer. The following example topology configuration file is for a network of four routers with the following link weights.

```
12
1,2,15
1,3,9
1,4,-1
2,1,15
2,3,4
```

```
2,4,8
3,1,9
3,2,4
3,4,7
4,1,-1
4,2,8
4,3,7
```

The first line of the file specifies the number of links we are going to define. If the weight is equal to -1, it means that the link does not exist (e.g., the links (1,4) and (4,1) are gone). Alternatively, if we do not specify the weight for a link, then it means that the link does not exist. Note that only the router programs need to read the topology configuration file. Furthermore, we make the following assumptions:

- The topology configuration file is error-free.
- If router A can reach router B, then router B must be able to connect to router A. We make sure the links (A,B) and (B,A) have the same link weights.
- All link weights are either positive or -1 (except if the links are disconnected). Also, the maximum link weight is 1000. Thus, if you find that the link weight is greater than 1000, you may assume that the link is disconnected.
- Router A and router B are neighbors of each other if link(A,B) and link(B,A) are well-defined and do exist.

2.2 The router program

Each router program will first read and parse the configuration files. After parsing the configuration files, the router program will listen on the *router_port*. It can also use the specified IP addresses or port numbers in the configuration files to set up connections to the neighboring routers.

Each router can receive commands from the agent. The agent can send commands, which can be classified as one of the following types: (i) link-weight updates, (ii) route updates, (iii) information maintained by a router. We will define the formats of the commands in later discussion.

In this assignment, the routers will not execute periodic updates. Also, they will not send distance vectors until being triggered by the agent. After being triggered by the agent, they will continue to transmit distance vectors to neighboring routers until the least-cost paths are determined (note that a router does not send DVs to another

router if they are not neighbors). If there is more than one path with the same cost, we pick the one that has the smaller router ID.

Note that you do not need to implement poisoned reverse. You would expect that the count-to-infinity problem could easily happen.

Note that routers must be ready to receive distance vectors from their neighbors. Each router must also count the number of distance vector packets it has received.

You may invoke each of the router programs as follows:

```
./router <router location file> <topology conf file> <router_id>
```

The router ID is the ID associated with the router process, and it correctly reflects the IP address and the port numbers.

2.3 The agent program

The agent program will send commands to routers. Table 1 defines the commands using examples. For the routing table of a router, it should show (i) the next hop and (ii) the path cost for each of the destinations (including itself) specified in router location file. If there is no path to a router, you don't need to display it. For example, the routing table of router 1 may look something like:

```
dest:  1, next:  1, cost:  0
dest:  2, next:  3, cost:  13
dest:  3, next:  3, cost:  9
dest:  4, next:  3, cost:  16
```

It is recommended that you format your routing table following this example. Note that there is only one space character after each colon and comma.

Table 1: Commands available for the agent.

Name	Example of commands	Description
`dv`	`dv`	All routers are triggered to propagate distance vectors to neighboring routers until the link costs become stable.
`update`	`update:1,6,4`	Change the link weight of link (1,6) to 4. If the link weight is -1, then it means that the link is deleted. If the link is absent before, then it means that the link is added with the update command.
`show`	`show:2`	Display the routing table of router 2.
`reset`	`reset:2`	Reset the number of distance vectors of router 2 to zero. This helps us easily debug your assignment.

Note that the commands might be sent to all routers, a single router, or a subset of routers. You may design the protocol of how to respond to each of the commands.

The commands are read from the console inputs. If the agent program receives anything that is not one of the above commands, it should display an error message.

You may invoke the agent program as follows:

```
./agent <router location file>
```

2.4 Protocol Messages

There are two main types of protocol messages: (i) messages shared between the agent and a router, and (ii) messages shared between the routers.

- **Messages between the agent and a router.** The agent will send a command request to an individual router. For the commands dv, update, and reset, the router do not need to reply anything; for the command show, the router replies a command response that contains the necessary information.

- **Messages between two routers.** The routers will exchange distance vectors. The distance vector of a router is the list of the estimated costs of the paths from the router to all other nodes in the network. Also, the maximum link weight is 1000. Thus, if you find that the link weight is greater than 1000, you may assume that the link is disconnected.

You only need to write a **single-threaded** router program to handle the above messages. Each router program listens on the *router_port* for the messages. When it receives a message, the router program makes the responses accordingly and waits for another message. In particular, if a router receives a `dv` command from the agent, it sends the distance vectors to its neighbors. If the router receives a distance vector from one of its neighbors, it updates its routing table and checks whether there is any cost change; if yes, then it sends distance vectors to its neighbors.

You may come up with your own format of the protocol messages for the command requests/responses as well as the distance vectors.

Make sure that the sender calls *htonl*() to convert the length into the network-byte order before you send out the message, and the receiver calls *ntohl*() to convert back the lexngth into the host-byte order.

3 Grading (100%)

We will use a certain number of test cases to verify your programs. One week before the deadline, we will publish all test cases and the score for each case.

For each test case, your score will be based on the correctness of the following functions:

- The `dv` command can trigger the distance vector routing protocol. (15%)
- The `show` command correctly shows the routing table. (10%)
- The distance vector routing protocol runs correctly for a static topology. (15%)
- The distance vector routing protocol runs correctly when the topology changes (60%)

4 Submission Guidelines

You must at least submit the following files, though you may submit additional files that are needed:

- *common.h*: the header file that includes all definitions shared by the router and the agent (e.g., message format)
- *agent.c*: the agent program
- *router.c*: the router program

- *Makefile*: a makefile to compile both agent and router programs

Your programs must be implemented in C/C++. They must be compilable on Unix/Linux machines.

The assignment is due on December 19 (Sunday), 23:59:59.

5 Remarks

- Our test cases ensure that each `update:a,b,w` command is followed by a `update:b,a,w` command. In other words, every update is mirror symmetric.
- Refer to section 2.3 to format the output of your `show` command. The routing table should be arranged in ascending order by router ids.
- We will check the output of your `agent`'s `show` command to determine if your program is correct, so your `router` is allowed to output debugging information, but you should comment out your `agent`'s debugging code before submitting.
- To accommodate automated tests, you need to add a statement at the beginning of the main function of your `agent` like this.

```
1  int main() {
2    setvbuf(stdout, NULL, _IONBF, 0);
3    /* Your Code. */
4  }
```

- You should name your code directory in a StudentID-WebLab3 format before you package and submit it.

6 AutoTest

We provide a Python script for automated testing of your code. You are suggested to use this script to test your own code before submitting.

6.1 Usage

Suppose `./` is the working directory for AutoTest. You should create two subdirectories, one named `Handin` and one named `Test`. Your code directory should be placed in the `Handin` directory. The test cases and our Python script should be placed in the `Test` directory.

```
.
|-- Handin
|   |-- 2100012345-WebLab3
|   |       |-- agent.c
|   |       |-- common.h
|   |       |-- Makefile
|   |       |-- router.c
|-- Test
    |-- TestCases
    |-- test.py
```

You can obtain the usage of the script by using the following command:

```
python3 test.py -h
```

To test `./Handin/2100012345-WebLab3/` on all test cases for 2 times with verbose output, run:

```
python3 test.py 2100012345-WebLab3 -v -i 2
```

To test `./Handin/2100012345-WebLab3/` on case 3 with a waiting rime of 0.5s per router per command, run

```
python3 test.py 2100012345-WebLab3 -c 3 -w 0.5
```

6.2 Script Workflow

- Each time it copies your code from `Handin/` to `Test/`. Also, it copies the test data from `Test/TestCases` into the copied code directory.
- A callback function on exiting is registered, which clean up all the resources and source files on demand.
- It then checks whether all the mandatory files exist in the folder. If so, it invokes `/usr/bin/make` to compile to get `agent` and `router`.
- On each test case, it first create one sub-process per router and agent. Then it sets up a polling thread which uses blocking I/O to read the `agent`'s output from the pipe and pushes the line into a thread-safe queue.

- On issuing a command, the main thread will sleep for a pre-configured amount of time and then uses non-blocking IO to consume all the items in the queue. If the issued command is a `show`, it matches the pattern with the standard answer.

6.3 Feedback

Our sctipt has a comprehensive error-handling framework. The most notorious errors are listed below:

- **Broken Pipe Error:** If the program try to reuse a just-freed socket, it is likely that such an error would occur. By default, the socket is not fully released until `TIME_WAIT` minutes after being `close()`.

  Solution: In lieu of calling `setsocketopt()` with `SO_REUSEADDR` for each socket you created, we modify `location.txt` each time so that the socket number never repeats. If such an error occurs, `test.py` will run the test case again with a larger address for the socket. If the error recurs for more than 8 times, `test.py` will blame your handin for this error.

- **Missing Output:** If `agent` outputs normally when being run manually in a linux terminal but fails to output anything with `test.py`, it is highly likely that the stdout of `agent` is buffered by default.

  Solution: Disable the buffer by simply adding a line `setvbuf(stdout, NULL, _IONBF, 0);` to the front of the `main()` function in `agent.c`.

- **Deadlock:** If the program deadlocks, your code can be the *only* culprit. When `test.py` was designed, the designer exhausted every detail ranging from the buffer of a pipe to the termination order to ensure that the autotest framework is robust.

- **Shutdown on Non-utf-8 String:** `test.py` only accepts utf-8 encoded string. **Not binary stream!** Please try to suppress the binary output (maybe debug info) of the your code.