Programming Assignment 2: JRIP

Last updated: 04/23/2018

In this project, you will be implementing two key networking algorithms, namely reliability and routing, simulating a simple network. You will also be asked to interoperate with the network infrastructure set up by the course team.
The routing part will implement a text-formatted version of the standard intra-domain routing protocol RIP which we call JSON-RIP or JRIP, using the same Bellman-Ford algorithm, but running over a vaguely TCP-like transport protocol implementing go-back-N over UDP instead of "plain" UDP.
You will implement the protocol in steps, building a router and a traceroute tool. **You can use either Python or Java and you can run the program on either a single host (with multiple ports) or on several hosts, such as your Google Cloud account.** (Using Google Cloud is recommended.)

Due date: April 20, noon EDT for Step 1; April 30, 2018 at 5 pm EDT for the complete assignment

Changes since 04/16 version

This 04/23 version separates the routing and traceroute function into two executables, to simplify implementation and testing. This led to the addition of the Origin parameter in the JSON payload. It also encodes the link weights into the command line arguments, to avoid having to parse a file.

Step 1: Connectivity and Loss Emulation

In the first step, you will implement the ability to transmit packets, via UDP, to neighbors and randomly "lose" these packets. Before sending each packet, you draw a random number and decide whether to actually transmit the packet. For example, if the loss probability specified via the **-l** (ell) parameter is 0.1, the packet is transmitted with a probability of 90%. You can use the random number generator, such as `random.randint(0,1000)` in Python, to return an integer between 0 and some large integer like 1,000, and then "lose" the packet if the returned random number is below the loss fraction, e.g., 100 for 10% loss. Random number generators return uniformly distributed integers between the range endpoints configured.
The receive port number is specified with the **-p** argument. The log entries are written to standard output; you can use the Unix "tee" command to capture the log to a file.
The other arguments specify destination hosts, port numbers and link weights, separated by colons. The host is provided either as a numeric IPv4 address (e.g., 128.59.16.1) or domain name (compute01 or compute01.cs.columbia.edu). The link weight is a positive integer less than 256. You should be able to specify an unlimited number of hosts. For example:

```
jrip -l 0.1 -p 7999 compute01.cs.columbia.edu:8000:10 \
  compute02.cs.columbia.edu:8001:7
```

To simplify testing of your implementation, please use the executable name `jrip`; use the `#!/usr/bin/env python` approach to make your Python script executable[1].
For testing purposes, send a simple identifying text string containing the local host name to the other hosts every ten seconds and display the output with a timestamp, the local host and the source, as in (as seen on host compute03)

```
Thu Mar 29 22:46:41 UTC 2018 compute03 128.59.11.31:8080 compute01
Thu Mar 29 22:47:49 UTC 2018 compute03 128.59.11.32:8080 compute02
```

Step 2: Reliability

In this step, you will implement a basic bidirectional GBN reliability mechanism, without session establishment and without flow or congestion control, but with TCP-like cumulative acknowledgements.
Each packet contains a packet sequence number (unsigned integer of at least 32 bit) and the ACK number, indicating the next sequence number expected. Sequence numbers start at zero. You do not have to worry about sequence number wrap-around. Your transport layer supports two upper layers: the routing protocol itself (protocol identifier “JRIP”) and a path tracing protocol (“TRACE”).
Assume timeoutvalue = 500ms, and window size = 5.
***Note**: ACKs will not contain any data, only seq #. See packet format below.

The **JSON packet format** is shown below:

```
/* Description
SEQ >=0 : sequence number
ACK >=0 : next sequence number expected
SEQ = -1 : no data
*/

{
  p
  "SEQ" : 1,
  "ACK" : 0,
  "Data" :
  {
    "Type" : "JRIP",
```

[1] see https://stackoverflow.com/questions/304883/what-do-i-use-on-linux-to-make-a-python-program-executable

```
    "RIPTable": [
      {
        "Dest" : "192.168.1.10:5001",
        "Next" : "192.168.1.11:5000",
        "Cost" : 5
      },
      {
        "Dest" : "192.168.1.13:4000",
        "Next" : "192.168.1.12:5002",
        "Cost" : 2
      },
      {
        "Dest" : "192.168.1.15:5003",
        "Next" : "192.168.1.11:4999",
        "Cost" : 6
      }
    ]
  }
}


{
  "uni" : "im2496",
  "SEQ" : 1,
  "ACK" : 0,
  "Data" :
  {
    "Type" : "TRACE",

    "Destination" : "192.168.1.11:5000",
    "Origin": "192.168.1.8:3000",

    "TRACE": [
      "192.168.1.51:5001",                ← this is the first hop
      "192.168.1.6:4000",
      "192.168.1.81:3000",
      "192.168.1.77:5002"
    ]
  }
}

/*ACK PACKET:
Note that you may piggyback data with an ack if you see fit however
this is not required. However note that if there is data, you should
ack this piggybacked packet.
The below example assumes that it has received the first SEQ. For
simplicity in this assignment the SEQ number does not have to be the
```

```
number of bytes but just the nth packet that has been sent. It is also
not piggybacking data and so we set SEQ = -1
*/
{
  "uni" : "im2496",
  "SEQ" : -1,
  "ACK" : 1,
  "Data" : {}
}
```

Step 3: Routing

The routing algorithm is Bellman-Ford, with interfaces and destinations indicated by IPv4 addresses. You can use any of the topologies in the textbook to test your routing algorithm, e.g., Chapter 5/P3 (see pg. 429).
The link weights are specified initially on the command line, as described in Step 1 above.
We will be providing code that will automatically change some link weights, so that you can know that your RIP protocol works dynamically.
Note that you should be sending out JRIP packets **every 10 seconds**.
Be sure to log any changes in your routing table to standard output, as in

```
Thu Mar 29 22:46:41 UTC 2018 192.168.1.51:5001 7
```

This indicates that the cost to the 196.168.1.51:5001 node is 7.

Step 4: Path Tracing

This step implements a separate traceroute-like function, call `jtraceroute`. It sends a TRACE packet to the designated router, using the same GBN reliability mechanism as before. (You should be able to re-use almost all of your code from the `jrip` function.) For example,

```
jtraceroute -p 4321 192.168.1.51:5001
```

would send a TRACE packet to the router at 192.168.1.51, port 5001 and listens for responses on port 4321.
Each `jrip` router simply takes the incoming packet, checks if it is meant for the local address and forwards it towards the destination if not, adding its own address and port number to the bottom of the TRACE list. If the trace packet has reached its destination, it returns the packet to the origin router, i.e., the address and port contained in the “Origin” field. This should reflect the correct trace path to the destination. Your `jtraceroute` tool waits for the response and then prints out the TRACE elements to stdout, e.g.,

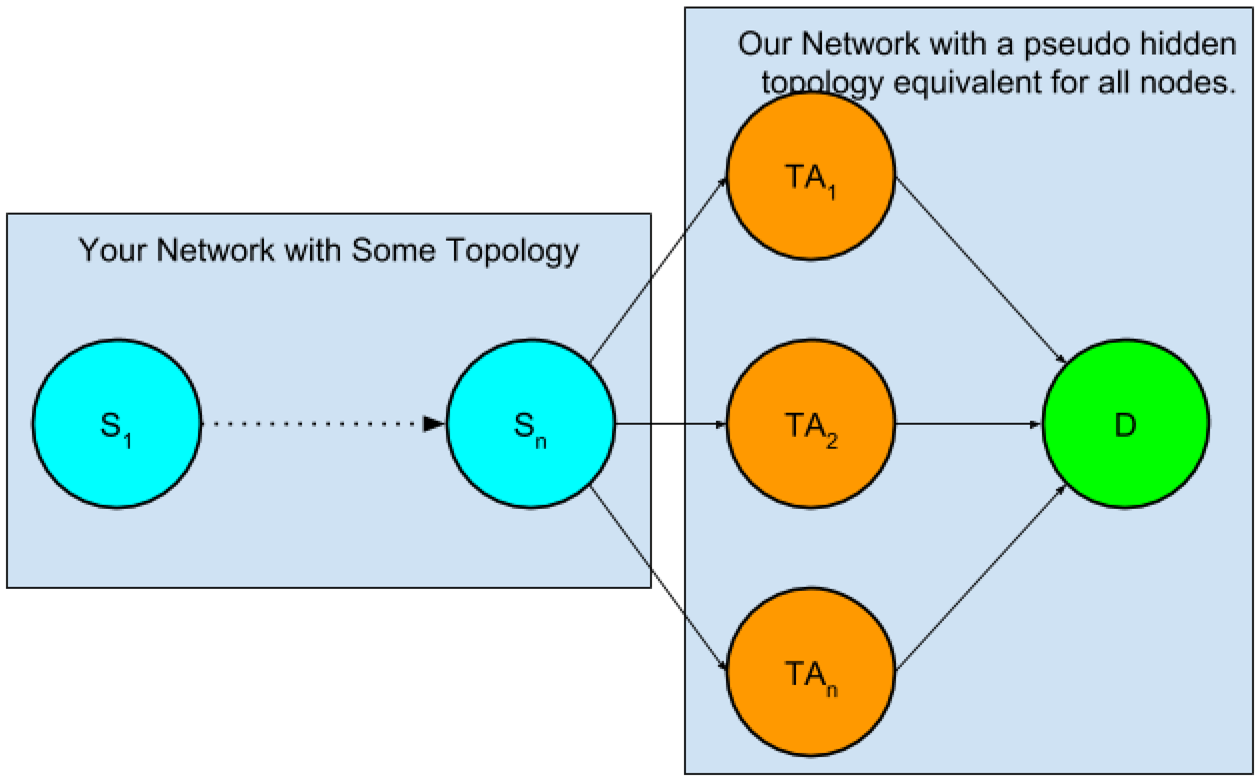
```
192.168.1.51:5001
192.168.1.6:4000
```

```
192.168.1.81:3000
192.168.1.77:5002
```

The “Origin” field is added by the first router, i.e., a router that receives a TRACE packet without any TRACE fields, based on the socket information, and ignored by all but the destination router. (This makes it easier to discover the right IP address where to send responses, even if your host running `jtraceroute` has multiple interfaces.) For Python, you can use `socket.getpeername()`.
Your `jtraceroute` tool only sends one traceroute message (and any retransmissions needed) and then exits after having received the response.

Step 5: Interoperability

To test interoperability, we will set up a set of nodes that you can connect to route your packets and send trace packets to. For example, we will later provide specific nodes that you may connect to.


Our Network with a pseudo hidden topology equivalent for all nodes.
Your Network with Some Topology
S_1
S_n
TA_1
TA_2
TA_n
D