Network Protocols Emulation Program Manual

1. Go-Back-N Protocol

Program name: gbnnode.class

Function

There are two nodes. The sender sends packets to the receiver through the UDP protocol. They use the Go-Back-N (GBN) protocol on top of UDP on both nodes to guarantee that all packets can be successfully delivered to the higher layers in the correct order. To emulate an unreliable channel, the receiver or the sender needs to drop an incoming data packet or an ACK, respectively, with a certain probability.

Features

●Data packet structure

SeqNum + “space” + Data

SeqNum: sequence number of the packet. Type:integer

Data: one character of the keyboard input. Type:char

The ACK packet is similar to the data packet except that it only has the header and does not have any data in it.

●Buffer

Each node has a sending buffer. All data packets (not ACK) will be put into the buffer before sending, and removed from the buffer once the corresponding ACK is received. In this program, the buffer is implemented as an array and the size of it is set to 100 to guarantee no sequence number conflict. If the buffer is full, the sending function simply waits until more space is available.

●Window

The window moves along the sending buffer. The window can move back to the beginning of the array after reaching the end of the buffer. Packets in the window will be sent out immediately. The size for the window will be passed in as an argument when starting the program.

●Timer

There is only one timer for GBN protocol. It starts after the first packet in the window is sent out, and stops when the ACK for the first packet in the window is received. After the window moves, if the first packet of the new window has already been sent out, the timer will simply restart; otherwise it should stop and wait for the first packet. The timeout for the timer should be 500ms and all the packets in the window should be resent after timeout.

●**Command**

$ java gbnnode <self-port> <peer-port> <window-size> [-d <value-of-n> j -p <value-of-p>]

The user should only specify either -d or -p. The square bracket and the vertical line mean to choose between the two options.

-d means the GBN node will drop packets (data or ACK) in a deterministic way (for every n packets).

-p means the GBN node will drop packets with a probability of p.

node> send <message>

message is a string which the sender will send to the <peer-port>

●**Status Message**

Sender

[<timestamp>] packet<packet-num> <packet-content> sent

[<timestamp>] ACK<packet-num> received, window moves to <packet-num>

[<timestamp>] ACK<packet-num> discarded

[<timestamp>] packet<packet-num> timeout

Receiver

[<timestamp>] packet<packet-num> <packet-content> received

[<timestamp>] packet<packet-num> <packet-content> discarded

[<timestamp>] ACK<packet-num> sent, expecting packet<packet-num>

Test Result:

Sender	Receiver
java gbnnode 1111 2222 5 -p 0.1	*java gbnnode 2222 1111 5 -p 0.1*
node>send abcdefgh [1355362831981] packet0 a sent	 node>[1355362831982] packet0 a received

[1355362831983] ACK0 discarded [1355362831997] packet1 b sent [1355362832015] packet2 c sent [1355362832016] ACK0 received, window moves to 1 [1355362832025] packet3 d sent [1355362832041] packet4 e sent [1355362832052] packet5 f sent [1355362832070] packet6 g sent [1355362832084] packet7 h sent [1355362832097] ACK0 received, window moves to 1 [1355362832097] ACK0 received, window moves to 1 [1355362832098] ACK0 received, window moves to 1 [1355362832098] ACK0 received, window moves to 1 [1355362832098] ACK0 received, window moves to 1 [1355362832597] packet1 timeout [1355362832598] packet1 b sent [1355362832600] ACK1 received, window moves to 2 [1355362832617] packet2 c sent [1355362832630] packet3 d sent [1355362832640] packet4 e sent [1355362832659] packet5 f sent [1355362832672] packet6 g sent [1355362832685] packet7 h sent [1355362832698] ACK2 received, window moves to 3 [1355362832699] ACK3 received, window moves to 4 [1355362832699] ACK4 received, window moves to 5 [1355362832700] ACK5 received, window moves to 6 [1355362832701] ACK6 received, window moves to 7 [1355362832701] ACK7 discarded [1355362833201] packet7 timeout [1355362833201] packet7 h sent [1355362833202] ACK7 received, window moves to 8	[1355362831983] ACK0 sent, expecting packet1 [1355362831998] packet1 b discarded [1355362832015] packet2 c received [1355362832015] ACK0 sent, expecting packet1 [1355362832025] packet3 d received [1355362832026] ACK0 sent, expecting packet1 [1355362832041] packet4 e received [1355362832041] ACK0 sent, expecting packet1 [1355362832052] packet5 f received [1355362832052] ACK0 sent, expecting packet1 [1355362832070] packet6 g received [1355362832070] ACK0 sent, expecting packet1 [1355362832083] packet7 h received [1355362832083] ACK0 sent, expecting packet1 [1355362832599] packet1 b received [1355362832599] ACK1 sent, expecting packet2 [1355362832617] packet2 c received [1355362832617] ACK2 sent, expecting packet3 [1355362832630] packet3 d received [1355362832630] ACK3 sent, expecting packet4 [1355362832640] packet4 e received [1355362832640] ACK4 sent, expecting packet5 [1355362832659] packet5 f received [1355362832659] ACK5 sent, expecting packet6 [1355362832672] packet6 g received [1355362832672] ACK6 sent, expecting packet7 [1355362832685] packet7 h received [1355362832685] ACK7 sent, expecting packet8 [1355362833202] packet7 h received [1355362833202] ACK7 sent, expecting packet8

Data structure

class gbnnode	Main function to start PacketListener thread and keep listening to the keyboard input.
class PacketListener	Thread for listening incoming port.
class SendingBuffer	A sending buffer of a node. It contains all methods of sending packets such as MakePacket(), Send(), Resend(), MoveWindow(), Ack().
class TimeoutResend	A timer for oldest unacked packet, timeout is 500ms.

Brief algorithm

Sender: Main() of gbnnode keeps listening for user keyboard input. If it gets the input, it will push packets into a buffer by MakePacket(). The buffer will send the packet whenever it has data. After sending packets within the window size, it will wait for ack. If an ack is received, the window will move to a proper position and the buffer sends new packets; otherwise, after 500ms, TimeoutResend thread will be called and the packet will be resent automatically. Of course, ack will be dropped according to the command.

Receiver: No buffer but a ack number is stored in it. If packet sequence num is the same as ack num, then it will send an ack and increase ack number. Otherwise, ack number will not increase. Packets are dropped according to the command.

2. Distance-Vector Routing Algorithm

Program name: dvnode.class

Function

A simple version of a routing protocol in a static network. A program which builds its routing table based on the distances (i.e., edge weights) to other nodes in the network. The Bellman-Ford algorithm should be used to build and update the routing tables. The UDP protocol should be used to exchange the routing table information among the nodes in the network.

Feature

●Network Model

We assume that that all the nodes run on the same machine and they all have the same IP address. Each node can be identified uniquely by a (UDP listening) port number, which is specified by the user. The port numbers must be > 1024. The maximum number of nodes to support is 16. The links among the nodes in the network and the distances (non-negative integer) between two directly connected nodes are specified by the user upon the activation of the program and stay static throughout the session. The link distance is the same for either direction.

●Bellman-Ford algorithm

The node uses Bellman-Ford algorithm to calculate the routing table in itself. The key idea of Bellman-Ford algorithm is “relax the edge”, which is shown below:

```
for each edge uv in edges: // uv is the edge from u to v
    u := uv.source
```

```
        v := uv.destination
        if u.distance + uv.weight < v.distance:
            v.distance := u.distance + uv.weight
            v.predecessor := u
```

Every node will find the minimum distance to any reachable node by this algorithm.

●**Command**

$ dvnode <local-port> <neighbor1-port> <loss-rate-1> <neighbor2-port> <loss-rate-2> ... [last]

<*local-port*> The UDP listening port number (1024-65534) of the node.

<*neighbor#-port*> The UDP listening port number (1024-65534) of one of the neighboring nodes.

<*loss-rate-#*> This will be used as the link distance to the <neighbor#-port>.It is between 0-1 and represents the probability of a packet being dropped on that link. For this section of the assignment you do not have to implement dropping of packets.Keep listing the pair of <*neighbor-port*> and <*loss-rate*> for all your neighboring nodes.

last Optional. Indication of the last node information of the network. Upon the input of the

command with this argument, the routing message exchanges among the nodes should kick in.

●**Status Messages**

1. Routing table (every time after a message is received, and for the initial routing table):

```
[<timestamp>] Node <port-xxxx> Routing Table

- (<distance>) -> Node <port-yyyy>

- (<distance>) -> Node <port-zzzz> ; Next hop -> Node <port-yyyy>

- (<distance>) -> Node <port-vvvv>

- (<distance>) -> Node <port-wwww> ; Next hop -> Node <port-vvvv>
```

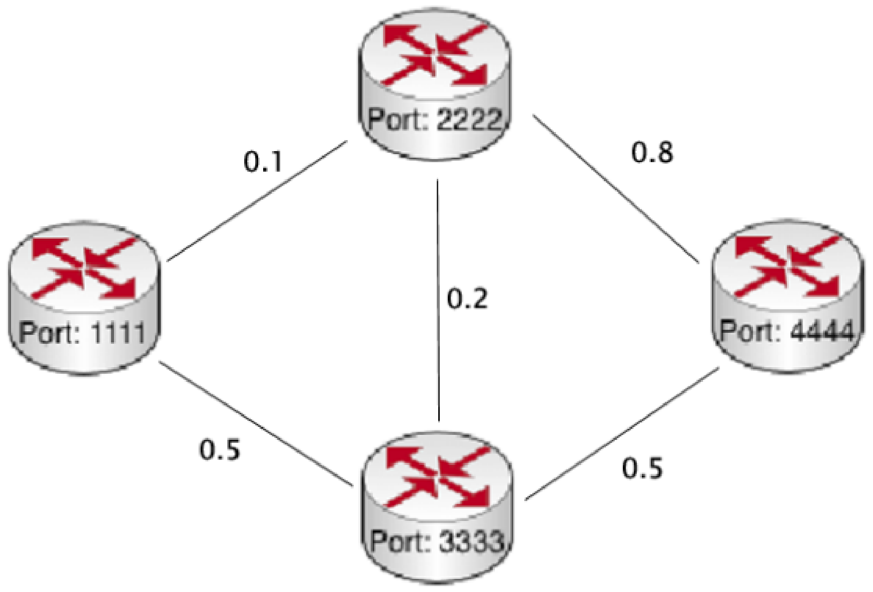
2. Routing message sent:

```
[<timestamp>] Message sent from Node <port-xxxx> to Node <port-vvvv>
```

3. Routing message received:

[<timestamp>] Message received at Node <port-vvvv> from Node <port-xxxx>

Test result:



Node 1111	Node 2222	Node 3333	Node 4444
java dvnode 1111 2222 .1 3333 .5	java dvnode 2222 1111 .1 3333 .2 4444 .8	java dvnode 3333 1111 .5 2222 .2 4444 .5	java dvnode 4444 2222 .8 3333 .5 last
[1355370237129] Node 1111 Routing Table **- (0.1) -> Node 2222** **- (0.5) -> Node 3333** [1355370243599] Message received at Node 1111 from Node 2222 [1355370243602] Node 1111 Routing Table - (0.1) -> Node 2222 - (0.3) -> Node 3333 ; Next hop -> Node 2222 - (0.9) -> Node 4444 ; Next hop -> Node 2222 [1355370243603]	**[1355370239593] Node 2222 Routing Table** **- (0.1) -> Node 1111** **- (0.2) -> Node 3333** **- (0.8) -> Node 4444** [1355370243597] Message received at Node 2222 from Node 4444 [1355370243599] Message sent from Node 2222 to Node 1111 [1355370243599] Message sent from Node 2222 to Node 3333 [1355370243601] Message sent from	**[1355370241762] Node 3333 Routing Table** **- (0.5) -> Node 1111** **- (0.2) -> Node 2222** **- (0.5) -> Node 4444** [1355370243597] Message received at Node 3333 from Node 4444 [1355370243599] Message sent from Node 3333 to Node 1111 [1355370243599] Message sent from Node 3333 to Node 2222 [1355370243599] Message sent from	**[1355370243587] Node 4444 Routing Table** **- (0.8) -> Node 2222** **- (0.5) -> Node 3333** [1355370243596] Message sent from Node 4444 to Node 2222 [1355370243596] Message sent from Node 4444 to Node 3333 [1355370243600] Message received at Node 4444 from Node 3333 [1355370243602] Node 4444 Routing Table

Message sent from Node 1111 to Node 2222 [1355370243603] Message sent from Node 1111 to Node 3333 [1355370243604] Message received at Node 1111 from Node 3333 [1355370243606] Message received at Node 1111 from Node 3333 [1355370243608] Message received at Node 1111 from Node 2222 **[1355370243608] Node 1111 Routing Table** **- (0.1) -> Node 2222** **- (0.3) -> Node 3333 ; Next hop -> Node 2222** **- (0.8) -> Node 4444 ;** Next hop -> Node 2222 [1355370243609] Message sent from Node 1111 to Node 2222 [1355370243609] Message sent from Node 1111 to Node 3333	Node 2222 to Node 4444 [1355370243604] Message received at Node 2222 from Node 3333 **[1355370243606] Node 2222 Routing Table** **- (0.1) -> Node 1111** **- (0.2) -> Node 3333** **- (0.7) -> Node 4444 ; Next hop -> Node 3333** [1355370243607] Message sent from Node 2222 to Node 1111 [1355370243607] Message sent from Node 2222 to Node 3333 [1355370243608] Message sent from Node 2222 to Node 4444 [1355370243608] Message received at Node 2222 from Node 3333 [1355370243609] Message received at Node 2222 from Node 4444 [1355370243610] Message received at Node 2222 from Node 1111 [1355370243610] Message received at Node 2222 from Node 4444 [1355370243611] Message received at Node 2222 from Node 4444	Node 3333 to Node 4444 [1355370243600] Message received at Node 3333 from Node 2222 **[1355370243600] Node 3333 Routing Table** **- (0.3) -> Node 1111 ; Next hop -> Node 2222** **- (0.2) -> Node 2222** **- (0.5) -> Node 4444** [1355370243601] Message sent from Node 3333 to Node 1111 [1355370243601] Message sent from Node 3333 to Node 2222 [1355370243601] Message sent from Node 3333 to Node 4444 [1355370243603] Message received at Node 3333 from Node 4444 [1355370243604] Message received at Node 3333 from Node 1111 [1355370243607] Message received at Node 3333 from Node 4444 [1355370243609] Message received at Node 3333 from Node 2222 [1355370243610] Message received at Node 3333 from Node 4444	- (0.7) -> Node 2222 ; Next hop -> Node 3333 - (0.5) -> Node 3333 - (1.0) -> Node 1111 ; Next hop -> Node 3333 [1355370243603] Message sent from Node 4444 to Node 2222 [1355370243603] Message sent from Node 4444 to Node 3333 [1355370243603] Message received at Node 4444 from Node 2222 [1355370243604] Node 4444 Routing Table - (0.7) -> Node 2222 ; Next hop -> Node 3333 - (0.5) -> Node 3333 - (0.9) -> Node 1111 ; Next hop -> Node 2222 [1355370243605] Message sent from Node 4444 to Node 2222 [1355370243605] Message sent from Node 4444 to Node 3333 [1355370243605] Message received at Node 4444 from Node 3333 **[1355370243606] Node 4444 Routing Table** **- (0.7) -> Node 2222 ; Next hop -> Node**

	[1355370243611] Message received at Node 2222 from Node 1111	[1355370243610] Message received at Node 3333 from Node 1111	**3333** **- (0.5) -> Node 3333** **- (0.8) -> Node 1111 ; Next hop -> Node 3333** [1355370243607] Message sent from Node 4444 to Node 2222 [1355370243607] Message sent from Node 4444 to Node 3333 [1355370243608] Message received at Node 4444 from Node 2222

Data structure

class dvnode	It contain main() to keep listening incoming routing table packets. If it's the last node, it will broadcast its routing table once.
class DVUpdate	This thread is to deal with every single routing table packets. It contains run() to implement Bellman-Ford algorithm and also BroadcastTable() and PrintRouteTable()
class RouteTable	It has attributes like DestPort, Dist, NextHop to maintain the routing table for the node.
class TimeStamp	Its attributes contains NeigPort and Time. It's used to check whether the received routing table is up-to-date.
class Neig	Its attributes contains NeigPort and Dist. Link weights are stored in it.

Brief algorithm

The main function will create a DVUpdate thread every time it receives a new routing table packet. After that, new thread will calculate by Bellman-Ford algorithm and update routing table. Meanwhile the main thread keeps waiting for next routing table packet.

For DVUpdate thread, it will compare the timestamp in the packet with the one in its TimeStamp table. If it's an old packet, the thread simply ignores it and return. Otherwise, the new timestamp will be set.

3. Combination

Program name: cnnode.class

Function

The combination of above two protocols. The GBN protocol can create reliable links, and the DV algorithms should determine the shortest paths over these GBN links. The distance of each link will be the packet loss rate on that link calculated by the GBN protocol.

Feature

●Loss rate calculation

Window size is always 5.

The data packet will only be dropped in a probabilistic way.

ACK will never be dropped.

Timeout is still 500ms.

The loss rate of each link is calculated by the following equation:

Link cost = (Total number of dropped packets/Total number of sent packets) and initial 0;

●Probe packets

To obtain the loss rate quickly, probe packets (data packets with any data) should be sent continuously (in one direction) on all links. Because each link has two nodes, the user has to specify which node is the probe sender, and which node is the probe receiver. Also, the sender should only start to send probe packets when all nodes in the network have become ready. As described in Section 3, there will be a last node with the word "last" as the argument. Once that node is started, the DV update messages should be sent to all that last node's neighbors, triggering the neighbors to send out DV messages as well. Therefore, the first time a node receives a DV update message, it should know that the network has become ready, and it can start sending probe packets.

●Routing information exchange

The distances in the routing table should be rounded to the second decimal place.

The updates of routing table should only be sent out 1) for every 5 seconds (if there is any change in the rounded distances based on the newly calculated loss rate), or 2) when a DV update is received from a neighbor that changes the local routing table.

The updates should be sent to all the neighbors, including all probe senders and receivers.

The probe receivers will only get the calculated distance of the corresponding links through the updates sent by the probe senders.

●Command

cnnode <local-port> receive <neighbor1-port> <loss-rate-1> <neighbor2-port> <loss-rate-2> ... <neighborM-port> <loss-rate-M> send <neighbor(M+1)-port> <neighbor(M+2)-port> ... <neighborN-port> [last]

<local-port> The UDP listening port number (1024-65534) of the node.

receive The current node will be the probe receiver for the following neighbors.

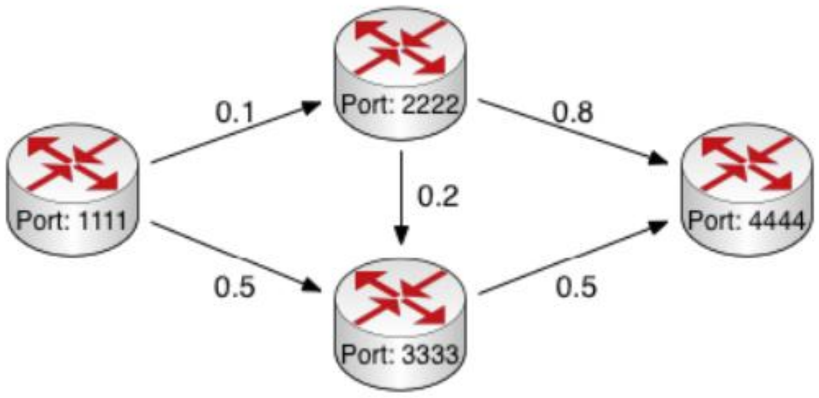
<neighbor#-port> The UDP listening port number (1024-65534) of one of the neighboring nodes. If you are using a different sending port number, you have to add the listening port number to your own packet header.

<loss-rate-#> The probability to drop the probe packets. Keep listing the pair of <sender-port> and <loss-rate> for all your neighboring nodes.

send The current node will be the probe sender for the following neighbors. Keep listing the pair of <neighbor-port> and <loss-rate> for all your neighboring nodes.

last Optional. Indication of the last node information of the network. Upon the input of the command with this argument, the routing message exchanges among the nodes should kick in.

Test Result



There are too many status messages, and I will show some important ones below.

Node name	Node 1111	Node 2222
Command	cnnode 1111 receive send 2222 3333	cnnode 2222 receive 1111 .1 send 3333 4444
Initial routing table	[1355366370464] Node 1111 Routing Table - (0.0) -> Node 2222 - (0.0) -> Node 3333	[1355366390485] Node 2222 Routing Table - (0.0) -> Node 1111 - (0.0) -> Node 3333 - (0.0) -> Node 4444
Loss rate after several seconds	[1355366488416] Link to Node 2222: 578 packets sent, 160 packets lost, loss rate 0.28 [1355366488417] Link to Node 3333: 434 packets sent, 221 packets lost, loss rate 0.51	[1355366493349] Link to Node 1111: 616 packets sent, 177 packets lost, loss rate 0.29 [1355366493349] Link to Node 3333: 589 packets sent, 169 packets lost, loss rate 0.29 [1355366493349] Link to Node 4444: 463 packets sent, 349 packets lost, loss rate 0.75
Routing table after several seconds	[1355366488822] Node 1111 Routing Table - (0.3) -> Node 2222 - (0.5) -> Node 3333 - (1.0) -> Node 4444 ; Next hop -> Node 3333	[1355366493733] Node 2222 Routing Table - (0.3) -> Node 1111 - (0.3) -> Node 3333 - (0.8) -> Node 4444

Node name	Node 3333	Node 4444
Command	cnnode 3333 receive 1111 .5 2222 .2 send 4444	cnnode 4444 receive 2222 .8 3333 0.5 send last
Initial routing	[1355366398674] Node 3333 Routing	[1355366447948] Node 4444

table	Table - (0.0) -> Node 1111 - (0.0) -> Node 2222 - (0.0) -> Node 4444	Routing Table - (0.0) -> Node 2222 - (0.0) -> Node 3333
Loss rate after several seconds	[1355366488242] Link to Node 1111: 429 packets sent, 220 packets lost, loss rate 0.51 [1355366488242] Link to Node 2222: 515 packets sent, 140 packets lost, loss rate 0.27 [1355366488243] Link to Node 4444: 438 packets sent, 215 packets lost, loss rate 0.49	[1355366493164] Link to Node 2222: 452 packets sent, 339 packets lost, loss rate 0.75 [1355366493164] Link to Node 3333: 481 packets sent, 239 packets lost, loss rate 0.50
Routing table after several seconds	[1355366488644] Node 3333 Routing Table - (0.5) -> Node 1111 - (0.3) -> Node 2222 - (0.5) -> Node 4444	[1355366493559] Node 4444 Routing Table - (0.8) -> Node 2222 - (0.5) -> Node 3333 - (1.0) -> Node 1111 ; Next hop -> Node 3333

Data structure

class cnnode	Main function to keep listening the incoming packets.
class DVUpdate	Thread to deal with three kinds of incoming packets(routing table, probe, probe ack).
class SendingBuffer	A sending buffer of a link. It contains all methods of sending packets such as MakePacket(), Send(), Resend(), MoveWindow(), Ack().
class RouteTable	It has attributes like DestPort, Dist, NextHop to maintain the routing table for the node.
class TimeStamp	Its attributes contains NeigPort and Time. It’s used to check whether the received routing table is up-to-date.
class Neig	Its attributes contains NeigPort and Dist. Link weights are stored in it.
class ReceiveNeig	Its attributes contains NeigPort, ValueOfN, AckNum. It’s used to store receive list.
class SendNeig	Its attributes contains NeigPort, BufferNum. It’s used to store send list.
class TimeoutResend	A timer for oldest unacked packet, timeout is 500ms.
class UpdateLossRate	Timer to update link weight every second.
class UpdateRoutePeriod	Timer to update routing table every 5 second

Brief algorithm

The main algorithm is the same as two protocols above. Some parts are modified.

Because of great amount of incoming packets, the main function just receives the packets and throws it to a new thread without doing anything with packets.

In the part of dealing with routing table packets, every time a routing table from node x is received. The first thing is to update local routing table entries whose next-hop is x, because the link weight between it should be different from last calculation. Then Bellman-Ford algorithm is used.

In the part of sending UDP probe packets, total number of sent packets and received packets are contained in the data. So every node can calculate the loss rate individually.

Because of link weight in NeigList changes very fast, so this program uses a FormerNeigList to maintain link weight relatively steady. It's synchronized with NeigList every second. Every value in the calculation is fetched in the FomerNeigList to avoid inconsistency.

Problem

I still have some problems with the combination part. I found it heavily depends on the performance of the PC and not very steady. The results of it running in my PC and running in ILAB7 are somehow different.

I spend a lot of time on debugging it and try several methods to improve. One thread is tried but incoming packets are lost heavily because they arrive very quickly. Then I used main function to spread the packet and pass the contents to the sub-thread. In this implementation, I found the loss rate of link keeps increasing and never converges. Also, the CPU usage becomes 100%. After debugging, I found some acks are lost in processing. Finally, I don't let main function do anything but give the incoming packets to a new thread. The performance improves greatly, CPU usage keeps under 50% and loss rate of link will converge.

The remaining problems now are small loss rate and delay of packets. As shown above, the loss rate between Node 1111 and Node 2222 is set as 0.1, but after program runs it will converge very slowly. And if the loss rate is set a little bigger such as 0.3, it will converge fast. Another problem I found in debugging process is that the packet will delay somehow, so the node sometimes cannot receive the update routing table messages over five seconds. As a result the routing table sometimes is not very up-to-date. I think the reason may be that too many threads are running at the same point, so the management of threads is quite unpredictable.