CSEE 4119: Computer Networks, Spring 2014

Programming Assignment 2: Reliable transmission and routing

Due Thursday, May 8th 11:55 pm

P. Prakash, W. An, P. Nirantar, R. Yu, X. Zhu (TAs), A. Chaintreau (instructor)

Academic Honesty Policy

You are permitted and encouraged to help each other through Piazza's web board. This only means that you can discuss and understand concepts learnt in class. However, you may NOT share source code or hardcopies of source code. Refrain from sharing any material that could cause your source code to APPEAR TO BE similar to another student's source code enrolled in this or previous years. Refrain from getting any code off the internet. Cheating will be dealt with severely. Cheaters will be penalized. Source code should be yours and yours only. Do not cheat.

---

1. Introduction

In this programming assignment, you will implement the Bellman Ford routing algorithm over a distributed system. The algorithm will operate using a set of distributed client processes. Clients are identified by an <IP address, Port> tuple. Also, each client maintains a routing table which records the weight of the path between itself and the other clients in the network.

Each client process should read the following from its configuration file for the initial setup.

- Information about self – localport, timeout
- Information about neighbors – IP address, port and link costs (weights)

Each client maintains a distance vector and exchanges distance vector information with its neighbors using ROUTE UPDATE messages.

On a high-level, you will be doing the following.

- In the first part of the assignment, you will focus on building the routing table. Your program should support the some commands for changing the network configuration dynamically. These commands are explained in detaill under Section 2c.

- In the second part, you will implement a P2P like file transfer mechanism. We ask you to first split an image file into two chunks randomly and then send them from any two nodes to a destination node using the routing table you built in the first part. In the destination node, you will combine the two chunks to get the original file and print out the two paths.

---

2. Specifications

a. You should write your program in one of the following languages
   - **C/C++**
   - **Java**
   - **Python**

b. You should write a client routing program.
   - Each client process gets as input the following.
     - the set of neighbors
     - the link costs to these neighbors
     - a timeout value
   - Each client has a read-only UDP socket on which it listens for incoming distance vector messages. This port number is known to its neighbors.
   - Each client maintains a distance vector which is a list of <destination, cost> tuples (one tuple per client). Here, the cost refers to the current estimate of the total link cost of the shortest path to the other client. Clients exchange distance vector information using a ROUTE UPDATE message, i.e., each client uses this message to send a copy of its current distance vector to its neighbors. Each client uses a set of write only sockets to send these distance vectors to its neighbors. The clients wait on their sockets until their distance vector changes or until TIMEOUT seconds pass, whichever occurs sooner, and then transmit their distance vectors to all neighbors.

c. Each client also provides a command line interface to the user. Five types of commands are required to be supported.
   - **LINKDOWN** {ip_address port } – This allows the user to destroy an existing link, i.e., change the link cost to infinity for the mentioned neighbor.

- **LINKUP** {ip_address port weight}. This allows the user to restore a link to the mentioned neighbor with the given weight after it was destroyed by a LINKDOWN.
- **SHOWRT** – This allows the user to view the current routing table of the client. In other words, this command should print, for each client in the network, the cost and neighbor (next hop) in the path to reach that client.
- **CLOSE** – With this command the client process should close/shutdown. Link failures is also assumed when a client does not receive a ROUTE UPDATE message from a neighbor (i.e., hasn't 'heard' from a neighbor) for 3*TIMEOUT seconds. This happens when the neighbor client crashes or if the user isuues the CLOSE command on it. When this happens, the link cost should be set to infinity and the client should stop sending ROUTE UPDATE messages to that neighbor. The link is assumed to be dead until the client comes up and a ROUTE UPDATE message is received from it again.
- **TRANSFER** {destination_ip_address port} – This is essentially the second part of the assignment. With this command, you should print out the next node (i.e., next hop) in the path to the destination mentioned and start the file transfer. Note that the destination may not be this client's neighbour. It can be assumed that the destination is in the network and that this command will be called only after all the Distance Vectors have converged. Please refer to Section 3 for more details on this.

d. Client Interface

A client process receives command-line arguments as follows:

```
%> ./bfclient config-file
```

where:

- `bfclient` is the name of the local process (your executable).
- `config-file` is a text file. Each client has its own config file that is formatted as follows:

```
localport timeout file_chunk_to_transfer file_sequence_number
ipaddress1:port1 weight1
[ipaddress2:port2 weight2]
[...]
```

where:

- `localport` (first line) is the local port number that the process should listen to (this is used to distinguish multiple clients on the same machine).
- `timeout` (first line) specifies the inter-transmission time of ROUTE UPDATE messages in steady state. It is specified in ***seconds***.
- The first line of the file may also have the arguments - `file_chunk_to_transfer` and `file_sequence_number` which is used for file transmissions. Please note that these arguments are only present in the config files that belong to source nodes. See section 3 for details.
- The remaining part of the file consists of lines of triples (at least one) indicating incident links to neighbors. Each triple consists of the following
  - `ipaddress`: the IP address of the neighbor (in dotted decimal notation)
  - `port`: the port number on which the neighbor is listening
  - `weight`: a real number indicating the cost of the link

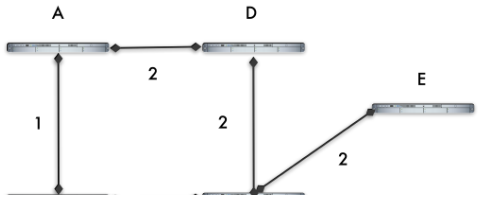
An example of a configuration file of a client listening on port 12345 with timeout 30 seconds, having neighbours 192.168.2.2:54321 and 192.168.2.10:53241 with weights 10.5 and 15 respectively and transfering file.jpg which is the second chunk is as follows.

```
12345 30 file.jpg 2
92.168.2.2:54321 10.5
192.168.2.10:53241 15
```

e. Poison Reverse

You should implement Poison Reverse in your routing algorithm. This means that if client A has to go through client B to get access to client C, in the distance vector A sends to B, the cost AC is infinity.

Here is a simple example of the routing algorithm:



The IP and port tuples are:

128.59.196.2:20000 (A)

128.59.196.3:20001 (B)

128.59.196.4:20002 (C)

128.59.15.48:20003 (D)

128.59.196.3:20004 (E)

For node A, when the routing table converges, SHOWRT command should print the following.

Destination = 128.59.196.3:20001, Cost = 1.0, Link = (128.59.196.2:20000)

Destination = 128.59.196.4:20002, Cost = 4.0, Link = (128.59.15.48:20003)

Destination = 128.59.15.48:20003, Cost = 2.0, Link = (128.59.196.4:20000)

Destination = 128.59.196.3:20004, Cost = 6.0, Link = (128.59.15.48:20003)

---

3. Specifications for the File Transmission part

The assignment has some aspects of distributed P2P file transfer. Here, the objective is to send data from one client to another using the routing table you built before. To make it interesting, we ask you to randomly split an image file into two chunks and send each split chunk of the file from any two nodes (called sources hereafter) in the topology to a destination node in the topology using the routing table.

File splitting can be simply done by yourself with **split -b** command, or you can use file chunk1 and chunk2 from our wiki site. Concatenating the two chunks by yourself by calling **cat chunk1 chunk2 > output** (giveing you the original file). Our example is a .jpg image of Internet map.

The goals here are as follows.

- Upon receiving each chunk of the file, the destination prints out the path traversed by that chunk of the file in the network, the timestamp and the size of the chunk.

- Upon receiving both the chunks of the file, the destination combines these and forms the original file that you split before. Write it to a file named **output**.

Here are the assumptions you can make:

- This part will be tested only after the routing table has converged and the network is stable.
- A route from each source to the destination exists.
- Each source is aware of which sequence number it sends.
- All chunks are smaller than 2MB
- The maximum number of hops is 20
- The node without a chunk to send can simply ignore the filename and sequence number in its config file.

To test this, following command will be issued from the source nodes:

```
%>TRANSFER <DESTINATION_IP> <DESTINATION_PORT_NUMBER>
```

You will need to specify the chunk name and sequence number in the config file as explained earlier. For extra credit, you can make this generic (i.e. more than two chunks) and add some innovative ideas. Make sure you document this in your README.

---

4. Example

Consider a client with the following config file (config.txt):

```
20000 3 file.jpg 2
128.59.196.2:20000 4.1
128.59.196.2:20001 5.2
128.59.196.4:20000 3
```

An example of the user interface is as follows:

%> ./bfclient config.txt

%>SHOWRT

```
<Current Time>Distance vector list is:
Destination = 128.59.196.2:20000, Cost = 4.1, Link = (128.59.196.2:20000)
Destination = 128.59.196.2:20001, Cost = 5.2, Link = (128.59.196.2:20001)
Destination = 128.59.196.4:20000, Cost = 3.0, Link = (128.59.196.4:20000)
```

(After sometime, new nodes may be added as they are discovered)

%>SHOWRT

```
<Current Time> Distance vector list is:
Destination = 128.59.196.2:20000, Cost = 4.1, Link = (128.59.196.2:20000)
Destination = 128.59.196.2:20001, Cost = 4.8, Link = (128.59.196.2:20000)
Destination = 128.59.196.4:20000, Cost = 3.0, Link = (128.59.196.4:20000)
Destination = 128.59.15.48:20000, Cost = 7.0, Link = (128.59.196.4:20000)
Destination = 128.59.15.41:20000, Cost = 5.2, Link = (128.59.196.2:20000)
```

Break a link

%>LINKDOWN 128.59.196.2 20000

The link cost to this neighbor should be set to infinity. The distance vector needs to be updated and a LINK DOWN message has to be sent to the neighbors. On receipt of a LINK DOWN message, a client should set the link cost to the sender as infinity. The two nodes are not neighbors till the link is restored (by a LINK UP message) and stop exchanging ROUTE UPDATE messages.

%>LINKUP 128.59.196.2:20000 5

The link cost should now be restored to the original value. The distance vector needs to be updated and a LINK UP message is sent to the neighbor. On receipt of a LINK UP message, a client should restore the link cost to the sender to the mentioned weight. The two nodes are now neighbors again and should resume exchanging ROUTE UPDATE messages.

%>TRANSFER 128.59.15.48 20000

file.jpg chunk 2 transferred to next hop 128.59.196.4: 20000

Upon receiving this command, the client should transfer the file (file.jpg chunk 2) to the next hop 128.59.196.4. 128.59.196.4 should be told the destination 128.59.15.48:20000 as well as the chunk sequence number via your own protocol. The next hop in turn transfers the received file chunk to its next hop and so on till the destination receives the chunk.

4. Requirements and Tips

For the routing part:

- The clients should not do busy waiting while listening on its socket.
- The clients should be able to adjust to dynamic networks – i.e. new nodes joining and existing nodes leaving the network.
- When a new node enters the network, its neighbors should also add it to their list of neighbors when they receive the first ROUTE UPDATE from it. They should use their distance from it as the link cost (picked from the first ROUTE UPDATE message they receive from the new neighbor).
- When the CLOSE command is issued, it is like simulating link failure of all the links to that client since the process exits and its neighbors don't receive ROUTE UPDATE messages for more than 3*TIMEOUT. (It is the same as hitting CTR-C).
- You do nott need to worry about the counting to infinity problem that is not addressed by Poison Reverse.
- Use of the select() is highly recommended.
- The protocol messages should be compact. Avoid containing redundant or unnecessary information.
- You should design your own protocol for the inter-client communications. You may use HTTP-like text protocol encoding or a proprietary designed one. Project submission should include a description of the designed protocol, including syntax and semantics.
- System time is obtained using gettimeofday and it should be truncated before it is display in order to improve readability.
- Design a good timer mechanism. Since you will be using only one timer (implicit in the select system call), you have to manage a queue of all timeout values that should occur in the future (two values for each neighbor; one for the time by which the client should send a ROUTE UPDATE and one for when the client expects a ROUTE UPDATE before concluding that the node is dead).
- Test your program with non-trivial topologies. Test your program behavior in a dynamic environment in which new clients join the system. Use the LINK UP/DOWN commands to test the convergence of your bellman-ford implementation.
- You should build your own protocol to transfer file chunks in the network. Describe it briefly in README.

5. Deliverables:

Submission will be done on courseworks. Please post a single **<UNI>_<Language>.zip** (Ex. zz1111_java.zip) file to the Programming Assignment 2 folder. The file should include the following:

- **README.txt**: This file should contain the following.
    a. A brief description of your code
    b. Details on development environment
    c. Instructions on how to run your code
    d. Sample commands to invoke your code
    e. Description of an additional functionalities and how they should be executed/tested.
- **Makefile**: The make file is used to build your application. Please try to keep this as simple as possible. If you don't know how to write a makefile, read this quick tutorial http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/.
- **Source code** and **related files**. (Make sure your code is commented!)
- **Sample config.txt file**

Grading rubric

Functionality	Max Points awarded / deducted
Basic client programs with following functionality as per the specifications: - Initial convergence of routing tables (prior to dynamic addition/removal of nodes using LINKDOWN and LINKUP) - Correct implementation of SHOWRT	45
Correct implementation of LINKDOWN	5
Correct implementation of LINKUP	5
Correct implementation of CLOSE	5
Correct implementation of poison reverse	10
Correct implementation of TRANSFER - Printing relevant details about path and chunk - Recreating original file on receiving all chunks	10+10
Well documented README, makefile (if required)	5
Code quality	5
Extra features: The feature should be useful and grading will be done on its innovation, utility and amount of effort.	Max 10 points per extra feature. On TA discretion.
Total max extra feature points	20

NOTE:

- You will lose 10 points if you do not follow the format of the config file passed as an argument while initializing the client.