CSEE W4119 Programming Assignment II

Due: May 6 2015

Prof. Augustin Chaintreau, Pooja Shah, Olalekan Afuye, Yelin Hong, Jen-Chieh Huang, Nivvedan S, Chenze Zhao

---

Description

In this assignment you will implement a version of the distributed Bellman-Ford algorithm. The algorithm will operate using a set of distributed host processes. The hosts perform the distributed distance computation and support a command line user interface, e.g. each host allows the user to edit links to the neighbors and view the routing table. Hosts may be distributed across different machines and more than one host can be on the same machine.

There are several parts to this assignment -
Part 1 - In this part, you will work on an implementation of the Bellman-Ford algorithm i.e. maintaining a correct routing table in the presence of link cost changes / failures / recoveries. You will also implement poison reverse as a solution to the count-to-infinity problem seen in the Bellman-Ford algorithm.

Part 2 - In the second part, you will implement a file transfer mechanism using your protocol from Part 1. We ask you to transmit an image file from any node to a destination node using the routing table you built in Part 1. The file will be sent from one neighbor to the next along the route from the sender to the destination.

Part 3: (Bonus) In the bonus part, you will simulate recovery from network failures. We will provide you a simulator which is a configurable proxy server that can simulate network failures such as packet corruptions and packet losses. Any packet that needs to be routed is sent via this proxy server. Your network nodes need to exhibit the correct behavior in handling these network failures. You need to implement ACKs to handle packet losses, and checksums to detect errors during transmission, and retransmit packets in case of packet losses / corruptions.

If you would like to implement any other feature for the bonus part instead of the suggested, you need to confirm it with one of the TAs in person or through private message on Piazza by the 30th of April. You also need to clearly explain what you have accomplished in your report.

Technical Specifications

You will basically implement a host program. Multiple host instances will interact with one another in a distributed setup.

Part 1: Host Program Requirements -

- Hosts are identified by an <IP address, Port> tuple.
- Each host process gets as input the set of neighbors, the link costs and a timeout value. The input set of neighbors is specified in a config file (e.g. client0.txt) personalized for each host. The format of the config file is specified below:

```
localport timeout
ipaddress1:port1 weight1
ipaddress2:port2 weight2
...
```

  Where:
  - ❖ `localport` is the udp port number on which client0 will listen for route updates.
  - ❖ `timeout` is the time in seconds between route updates. If route updates are not received after 3*timeout elapses, the node is assumed dead and the link cost is set to infinity
  - ❖ Every other line after the first line specifies immediate neighbors and associated link costs.
    - ➢ `ipaddress1:port1` specifies an immediate neighbor to client0
    - ➢ `weight1` specifies the link cost from `client0` to `ipaddress1:port1`
    - ➢ Note that since all link costs are symmetrical, all neighbors specified will also have corresponding entries in their config files. For example, `ipaddress1:port1` will also have an entry in its config file specified as

      `ipaddress0:localport weight1`

      where `ipaddress0` is the ip address of client0.

  An example config file for a host listening on port 12345 with timeout 30 seconds and neighbors 192.168.2.2:54321 and 192.168.2.10:53241 with weights 10.5 and 15 respectively is given below:

```
12345 30
192.168.2.2:54321 10.5
192.168.2.10:53241 15
```

- Each host has a UDP socket on which it listens for incoming messages and its neighbors know the number of that port.
- Each host maintains a distance vector, that is, a list of <destination, cost> tuples, one tuple per host, where cost is the current estimate of the total link cost on the shortest path to the other host.
- Hosts exchange distance vector information using a ROUTE UPDATE message, i.e., each host uses this message to send a copy of its current distance vector to its neighbors.
- Poison reverse functionality should be implemented. In other words, if B is connected to C through A, in B's distance vector published to A, the cost to C should be infinity.
- The hosts wait on their sockets until their distance vector changes or until TIMEOUT seconds pass, whichever arrives sooner, and then transmit their distance vectors to all neighbors.

Part 2: File Transfer

In this part, you are going to implement a file transfer protocol. The basic idea is to send a file from a source node S to a destination node D over the routing protocol designed in Part1 i.e. the file transmission should follow the SAME PATH from S to D as can be seen in the routing table.

Your protocol should also be based on UDP and uses the same socket in Part 1. In this part, you can assume that transmission channels are reliable, i.e. there is no packet loss, corruption, etc.

In order to route packets in the network, you have to design your own network, transport and application layer protocols. **For security reasons, each node is only allowed to send packets to its immediate neighbors. In other words, you have to forward the packets hop by hop according to your own routing table. Transferring the file directly to the destination address is not allowed.**

Another thing you should take into consideration is the fragmentation. Since the file might be too large to fit one UDP packet (and notice that your own headers are also part of the packet), you have to set an appropriate MSS (maximum segment size) and fragment the file into multiple segments. We will not test the case that multiple hosts transfer files to a specific host simultaneously, but you can implement it if you like.

When the a node receives a packet, you are required to print the addresses of the source and destination nodes of that packet. The final destination has to indicate a successful transmission after it receives the last segment of the file.

Part 3: Reliable File Transfer in the presence of losses and distortion (Bonus)

In the bonus part, you will extend the file transfer feature implemented above to work correctly in the more realistic network setting that there are packet losses and packet corruptions.

Packet Losses
We will be using UDP packets for the file transfer. You need to implement a reliability layer on top of this so that all packets sent are properly acknowledged. You need not implement a fully functional TCP layer on top of UDP - just the acknowledgements. You are not allowed to directly use TCP packets for this.

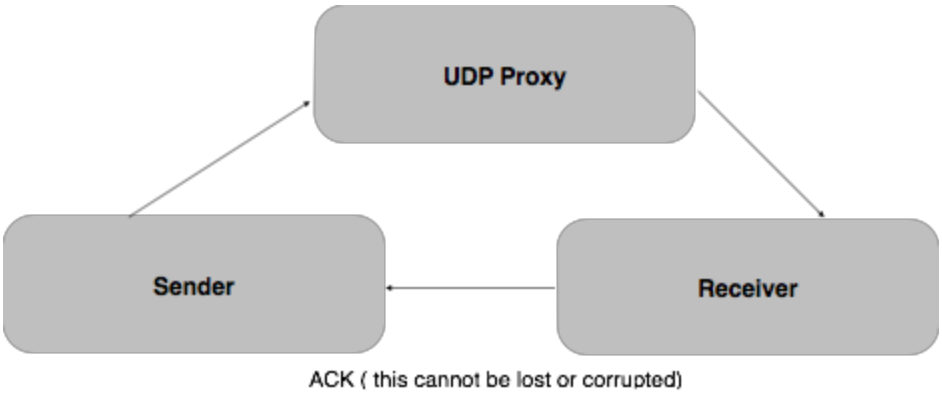
Packet Corruption
You need to implement a checksum feature to check for packet corruptions at the recipient side. The IP layer already includes a checksum - you need to implement one for the transport layer.

Since these packet errors are almost impossible when you're developing in local, we will provide you with a proxy that can simulate these network errors. An instance of the proxy needs to be run for each recipient.

Emulator Details

To test your implementation you will use a link emulator provided by us (see proxy.zip under Programming Assignment 2; contains a README that guides usage). Data is exchanged via UDP, i.e., you will be running your TCP-like ACK implementation "on top of" UDP. In the real world, a network can drop, corrupt, reorder, duplicate and delay packets. To mimic this behavior you will use the link emulator. Specifically, you will invoke the emulator so that it lies in the sender-to-receiver path, whereas the acknowledgements (i.e., the packets on the receiver-to-sender path) will be sent directly from the receiver to the sender, i.e., you can assume that packets arrive without a loss on this path.

A simple diagram to show the interactions is given below -



You can run your sender, receiver and link emulator either on one, two or three machines. The link emulator acts like a "proxy", i.e., the sender sends packets towards the emulator and the emulator should forward the packets to the receiver. The receiver process should send its acknowledgements directly to the data sender.

Commands

Each host provides a command line interface to the user.
The following commands should be supported. The syntax and semantics of the commands are listed below.

Command Name and Syntax	Command behavior
LINKDOWN {ip_address port}	This allows the user to destroy an existing link, i.e., change the link cost to infinity to the mentioned neighbor
LINKUP {ip_address port }	This allows the user to restore the link to the mentioned neighbor to the original value after it was destroyed by a LINKDOWN
CHANGECOST {ip_address port cost}	This allows the user to change the cost of the link to the mentioned neighbor to the new cost.If the destination link is currently down, this command should be ignored.
SHOWRT	This allows the user to view the current routing table of the host. It should indicate for every other host in the network, the cost and

	neighbor used to reach that host.
CLOSE	With this command the host process should close/shutdown. This command is essentially the same as force closing the host. Neighboring hosts should detect this link failure after 3*TIMEOUT seconds. When a link failure is detected, the link cost should be set to infinity and the host should stop sending ROUTE UPDATE messages to that neighbor. The link is assumed to be down until a ROUTE UPDATE message is received again from that neighbor
TRANSFER {filename destination_ip port}	This is 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 host'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. At each node on the path from the sender to the destination, you should print the source, destination and next hop (of the packet) when a packet is received. When the destination has received all of the segments for a file, it should indicate a successful transmission. You can ignore the case where the filename already exists in the recipients directory.
ADDPROXY {proxy_ip proxy_port neighbor_ip neighbor_port}	This command is used in the third (bonus) part of the assignment. With this command, you are introducing a proxy between 2 hosts, i.e. the host where the command is executed and the neighbor mentioned in the command. The proxy ip address/port is also provided. If a proxy is added between 2 hosts, all direct communication from the host to this neighbor(unidirectional) has to be routed via the proxy. You also need to ensure that the proxy is added between the host and its neighbor. If the ip/port details don't match any neighbor, the proxy shouldn't be added.
REMOVEPROXY { neighbor_ip neighbor_port}	This command is used in the third (bonus) part of the assignment. With this command you can remove an existing proxy between the host and its neighbor. If such a proxy doesn't exist, the command should be ignored. After executing this command, all direct communication between the host and its neighbor shouldn't be routed via the proxy.

Sample Run

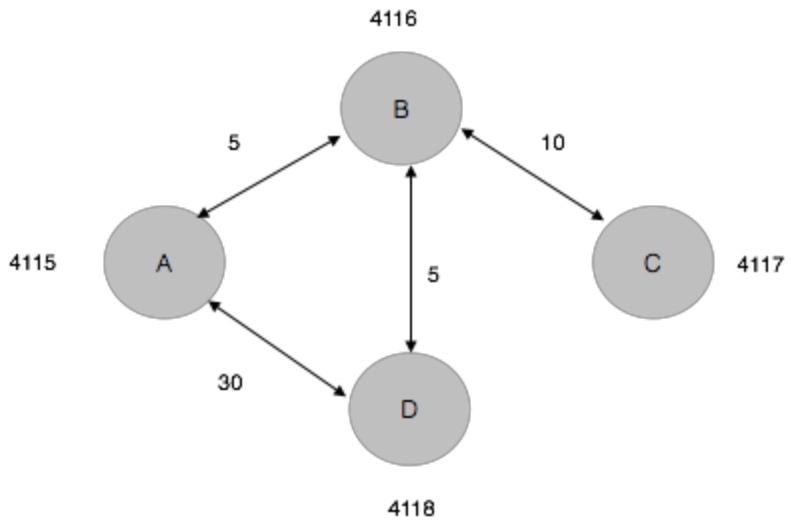
A host process receives command-line arguments as follows:

```
%> ./bfclient client0.txt
```

where

- bfclient is the name of your executable.
- `client0.txt` specifies the node configuration for the host as specified in the host program requirements above.

Consider the topology portrayed in the figure below and a client0 represented by node



A. Its config file would be specified as:

4115 30
128.59.196.2:4116 5
128.59.196.2:4118 30

and saved as client0.txt.

This host would be invoked as

```
%>./bfclient client0.txt
```

Show current route

```
%>SHOWRT
```

<Current Time>Distance vector list is:

Destination = 128.59.196.2:4116, Cost = 5.0, Link = (128.59.196.2:4116)
Destination = 128.59.196.2:4118, Cost = 30.0, Link = (128.59.196.2:4118)

After some time, new nodes might be added as they are discovered
%>SHOWRT
<Current Time> Distance vector list is:
Destination = 128.59.196.2:4116, Cost = 5.0, Link = (128.59.196.2:4116)
Destination = 128.59.196.2:4118, Cost = 10.0, Link = (128.59.196.2:4116)
Destination = 128.59.196.2:4117, Cost = 15.0, Link = (128.59.196.2:4116)

Break a link
%>LINKDOWN 128.59.196.2 4116
The link cost to this neighbor should be set to infinity (The distance vector needs to be updated) and a LINKDOWN message is sent to the neighbor. On receipt of a LINK DOWN message, a host should set the link cost to the sender as infinity. The two nodes are not neighbors till the link is restored (by a LINKUP message) and stop exchanging ROUTE UPDATE messages.

Recover a link
%>LINKUP 128.59.196.2 4116
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 host should restore the link cost to the sender to the original value specified in the config file. The two nodes are now neighbors again and should resume exchanging ROUTE UPDATE messages.

Send a file
%>TRANSFER myFile.jpg 128.59.196.2 4118
Next hop = 128.59.196.2:4116
File sent successfully

When a node on the path receives a packet, it should print the source and destination of the packet. For example, if node 128.59.196.2:4116 receives a packet from 128.59.196.2:4115 to 128.59.196.2:4118, it should print:
Packet received
Source = 128.59.196.2:4115
Destination = 128.59.196.2:4118
Next hop = 128.59.196.2:4118

When the destination successfully receives the whole file, it should also indicate a successful transmission:
Packet received
Source = 128.59.196.2:4115
Destination = 128.59.196.2:4118
File received successfully

Note: The above example has all the nodes on the same machine, but we will test using nodes on different machines.

Development Environment Specifications

Development Environment

You can choose between Java, Python and C/C++ to write your program. Whichever you choose, your program **must compile and run on the CLIC machines**. For your reference, the versions of the compilers/interpreters on the CLIC machines include:

- Java - 1.7.0_75
- Python 2.7.3 / Python 3.2.3
- C/C++: gcc/g++ 4.6.3

Please note the change in Python and Java versions from the previous assignment.

NOTE - If your code doesn't compile, we will call you to have a look at it and fix it. However, it will result in a deduction of 20% of your total points.

Deliverables

Your program will be submitted via courseworks. Please submit a zip file using the format **<UNI>_<Programming Language>.zip** (e.g. cn1111_python.zip) to the Programming Assignment 1 folder. Make sure you include all of the following files in your zip file:

- README.txt: Your readme file should include but not limited to the following parts
    - A general description of your programming design and data structure
    - Explanation of your source code (make sure your code is well commented and readable or 5-10 points will be deducted. To avoid such an undesirable scenario, tools like astyle[1] may be helpful.)
    - Detailed instructions on how to run/compile your source code
    - Sample commands to run your program
    - A short introduction of your additional features and sample test cases (**Notice: discuss your design of additional features to one of TAs during office hour or post a private message on Piazza before you actually implement them, some features may not be considered worthy of the bonus points**)

[1] http://astyle.sourceforge.net/

- Makefile: if you are not familiar with makefile, you can read through the tutorial: http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ This article is also helpful: http://www.devin.com/cruft/javamakefile.html
- Source code and other files you think are important

Grading Scheme

Functionality	Points
Part 1 - Distributed Bellman Ford algorithm	**80 points**
Creation of a stable initial distributed routing environment. This should work across a minimum of 5 hosts, with multiple hosts on the same machine as well as spread across other machines. SHOWRT will be used to validate this state. All tests should succeed for full credit.	**30**
Correct implementation of the LINKDOWN command. This should update link costs. Routing table should be updated if required. SHOWRT should display the correct state after the change. This will be tested by running LINKDOWN on 1 or multiple hosts. All tests should succeed for full credit.	**10**
Correct implementation of the LINKUP command. This should update link costs. Routing table should be updated if required. SHOWRT should display the correct state after the change.This will be tested by running LINKUP on 1 or multiple hosts. All tests should succeed for full credit.	**10**
Correct implementation of the CLOSE command. The host's neighbors should update link cost to infinity after 3*timeout. Routing table should be updated if required. SHOWRT should display the correct state after the change.This will be tested by running CLOSE on 1 or multiple hosts. All tests should succeed for full credit.	**10**
Correct implementation of CHANGECOST command. This should update the cost to and from the specified neighbor on both the host and the specified neighbor.	**10**

Correct implementation of poison reverse to avoid the count to infinity problem.	**10**
Part 2 - File Transfer (details required).	**20 points**
Successful transmission of the file.	**10**
Correct routing path from source to destination.	**10**
Bonus - Reliable File Transfer in the presence of losses / corruption of packets	**20 points**
Reliable File Transfer in the presence of loss	**10**
Reliable File Transfer in the presence of corruption	**10**

Note:

1. If we find detect any plagiarism in your assignment, you will get 0 points in this assignment. In addition, you will be referred to the Dean of Students without any exception.

Appendix A: Guidelines and Tips

The hosts should not do busy waiting while listening on its socket.

- The hosts 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 host since the process exits and its neighbors don't receive ROUTE UPDATE messages for more than 3*timeout. (It is the same as hitting CTRL-C).
- The protocol messages should be compact and avoid containing redundant or unnecessary information.
- You should design your own protocol for the inter-host communications. You may use HTTP-like text protocol encoding or a proprietary designed one.
- 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 host should send a ROUTE UPDATE and one for when the host 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 hosts join the system. Use the LINK UP/DOWN commands to test the convergence of your bellman-ford implementation.

Appendix B: If You Don't Have a Clue -- A Reference Design

In this assignment, you will need to implement a routing algorithm, which is the core of the network layer. This section will show you some important ideas about how to start dealing with the task.

First of all, since the project will be built upon the existing stack, it is intuitive to select UDP to simulate the link layer because of the unreliable nature of the underlying medium. Thus, the first thing you may want to do is to define some messages related to the link status, such as LINK UP, LINK DOWN and LINK_COST_CHANGE.

Then, what to be conveyed using the link layer is to be defined. The possible data to run upon the link layer are routing information, and the user data. To distinguish between the different data formats, a proper packet format should be designed. If the received data are routing information, the routing protocol in your network layer should handle that; otherwise, the payload should be passed to the higher layer. For your network layer packet format, you may want to put the source address, the destination address, and the protocol id (what kind of data in the payload) in the header.

Note that layering may not be necessary, and indeed you can do this without any layering. However, it’s going to be messy. If you can clearly define the responsibilities and the data formats of each layer, developing new features and debugging will be much easier. The complexity of the whole project will also be reduced significantly.

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 hard copies 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.
