Summer 2018 - CSEE W4119 Computer Networks
Programmi t 2 - Network Protocols Emulation

f. Gil Zussman

/4/2018, 23:59pm

1 Introduction

In this assignment, you will emulate the operation of a link layer and network layer protocol in a small computer network. The program you write should behave like a single node in the network. You will start several nodes (instances of your program) so that they can send packets to each other as if there are links between them. Running as an independent node, your program should implement a simple version of Go-Back-N Protocol (Link Layer) and the Distance-Vector Routing Algorithm (Network Layer), in order to provide reliable transmissions and efficient routing.

Each section of the assignment will walk you through a different portion on the way to this end goal. Section 2 will have you implement the Go-Back-N protocol whereas Section 3 will have you implement a Distance Vector Routing Algorithm. By the time you get to Section 4, which is to combine these two algorithms, you should have the vast majority of the hard work done and can focus on these two algorithms working in tandem. The assignment is split into these sections not only to make it easier, but also to give you the most opportunity for partial credits. Accomplishing only Sections 2 and 3 will get you 80 of the 100 points available. Please follow the instructions step by step, and submit the programs for whichever sections you attempt/accomplish.

This assignment is meant to be done on your own, do not collaborate with others as this is cheating! Please start early, and read the entire homework, and set up the specified programming environment before you start implementing!

2 Go-Back-N Protocol (40pt)

2.1 Description

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

2.2 Protocol

- Packet
  For simplicity, each character is regarded as one data packet, which means that the data in each packet should have the max length of only 1 byte. This section involves two kinds of packets, data packets and ACK packets. Your own packet header (not the UDP header) is needed to maintain the order of data packets. There will be no requirements on the format of these headers, so you should come up with your own design. For example, you can add a 32-bit sequence number ahead of the data you want to send. (See Figure 1).

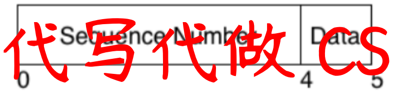

Figure 1: GBN Packet Header Example

The ACK packet i[illegible]t except that it only has the header and does not have any data in it.

- Buffer
  Each node should [illegible] All data packets (not ACK) should be put into the buffer before sending, and remo[illegible] the corresponding ACK is received. The sending buffer should be long enough to avoid the conflict of packet numbers given the window size below. If the buffer is full, the sending function simply waits until more space is available.

- Window
  The *window* moves along the sending buffer. If you implement the buffer as an array, the window should wrap back around to the beginning of the array after reaching the end. As in GBN, packets in the window should be sent out immediately. The size for the window will be passed in as an argument when starting your 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 should simply restart, otherwise it should stop and wait for the first packet to be sent out.
  The timeout for the timer should be 500ms and all the packets in the window should be resent after timeout.

2.3 Emulation

In this section, you should create a program that emulates a GBN node using the specifications above. Two GBN nodes will be running to send packets to each other over the UDP protocol. For emulation purposes, the receiver will deliberately discard data packets (as if the packets are lost and not received) and the sender will deliberately discard ACKs before actually handling them. There are two ways to control the drop of the packets - deterministic and probabilistic. The deterministic way is to drop every nth packet, and the probabilistic way is to drop packets randomly with certain probability p (for sanity check, probability of 0 here means no packets are lost, and probability of 1 means all packets are lost). Which method will be used and the value of n or p will be passed in when starting the program.

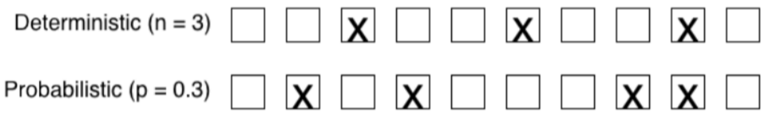

Figure 2: Packet Loss Example

2.4 Command Syntax

- Starting Programs
  The GBN node should be started with any of the following commands:

C	`$ ./gbnnode <self-port> <peer-port> <window-size> [-d <value-of-n> | -p <value-of-p>]`
Java	`$ java gbnnode <self-port> <peer-port> <window-size> [-d <value-of-n> | -p <value-of-p>]`
Python	`$ python gbnnode.py <self-port> <peer-port> <window-size> [-d <value-of-n> | -p <value-of-p>]`

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

`-d` means the GBN [illegible] data or ACK) in a deterministic way (for every `n` packets), and `-p` means the GBN n[illegible] a probability of `p`.

(To make it clear, [illegible] the program. The arguments should be passed in at the launch of the program, not a[illegible]ed to run). The square brackets with a bar in the middle indicate an "either or" com[illegible]

- Sending Messages[illegible]
  After the GBN no[illegible] prompted for commands. There is only one command allowed:
  `node> send <message>`
  `message` is a character string which the sender will send to the `<peer-port>`

2.5 Status Messages

The following messages should be displayed on each node for the situations that have been described above:
Sender side:

```
[<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 side:

```
[<timestamp>] packet<packet-num> <packet-content> received
[<timestamp>] packet<packet-num> <packet-content> discarded
[<timestamp>] ACK<packet-num> sent, expecting packet<packet-num>
```

Note that `<timestamp>` should be the current time accurate to at least the millisecond. The following functions can be used to retrieve time information (you can use other functions as well):

C	`gettimeofday()`
Java	`Calendar.getInstance().getTime()`
Python	`time.time()`

2.6 Loss Rate Calculation

At the end of the transmission, have both the sender and receiver print out the ratio of total packet failures to total packets received, to see how it relates to the failure rate specified on the command line:

```
[Summary] <# dropped>/<# total> packets discarded, loss rate = <value>%
```

To test the loss rate emulation, set up a sender and receiver node where:

- No ACKS fail (sender drop rate = 0)
- Packets dropped probabilistically (receiver drops with probability p)
- 1000 packets are sent (message of 1000 characters)
- window size is 5

At the end of the transmission, the receiver's summary message should present a loss ratio near the input probability p.

2.7 Cases

Your program should deal with the following cases:

Normal packet sending	Window moves and ACK replied	(10pt)
An ACK is lost	Following ACK could also move the window	(10pt)
A packet is dropped	Timer timeout and packets resent	(5pt)
Duplicate packets	[illegible]ered and the correct ACK is replied	(5pt)
Duplicate ACK's	[illegible]ove	(5pt)
Loss rate calculation	[illegible]test converges to input probability	(5pt)

2.8 Example

Sender side:

```
$ ./gbnnode 1111 2222 5
node> send abcdefgh
[1353035731.011] packet0 a sent
[1353035731.031] packet1 b sent
[1353035731.049] packet2 c sent
[1353035731.062] ACK0 received, window moves to 1
[1353035731.078] packet3 d sent
[1353035731.082] ACK1 received, window moves to 2
[1353035731.087] packet4 e sent
[1353035731.090] ACK1 received, window moves to 2
[1353035731.103] packet5 f sent
[1353035731.106] ACK1 received, window moves to 2
[1353035731.118] packet6 g sent
[1353035731.124] ACK1 received, window moves to 2
[1353035731.136] ACK1 received, window moves to 2
[1353035731.549] packet2 timeout
[1353035731.551] packet2 c sent
[1353035731.562] packet3 d sent
[1353035731.573] ACK2 received, window moves to 3
[1353035731.575] packet4 e sent
[1353035731.593] ACK3 discarded
[Summary] 1/8 packets discarded, loss rate = 0.125%
```

Receiver side:

```
$ ./gbnnode 2222 1111 5 -p 0.1
[1353035731.033] packet0 a received
[1353035731.042] ACK0 sent, expecting packet1
[1353035731.053] packet1 b received
[1353035731.059] ACK1 sent, expecting packet2
[1353035731.072] packet2 c discarded
[1353035731.079] packet3 d received
[1353035731.084] ACK1 sent, expecting packet2
[1353035731.095] packet4 e received
[1353035731.101] ACK1 sent, expecting packet2
[1353035731.106] packet5 f received
[1353035731.113] ACK1 sent, expecting packet2
[1353035731.118] packet6 g received
[1353035731.120] ACK1 sent, expecting packet2
[1353035731.559] packet2 c received
[1353035731.562] ACK2 sent, expecting packet3
[1353035731.579] packet3 d received
```

```
[1353035731.581] ACK3 sent, expecting packet4
[1353035731.590] packet4 received
[Summary] 1/10 packets dropped, loss rate = 0.1
```

3 Distance-Ve[illegible]rithm (40pt)

3.1 Description

The objective of this por[illegible]o implement a simplified version of a routing protocol in a static network. You are requi[illegible]which builds its routing table based on the distances (i.e., edge weights) to other nodes i[illegible]an-Ford algorithm should be used to build and update the routing tables. The UDP protoc[illegible]ge the routing table information among the nodes in the network.

3.2 Network Model

You may assume that that all the nodes (instances of your program) 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. No need to support dynamic link/node status changes. The link distance is the same for either direction (e.g. between node A and B, Distance$_{A \to B}$ = Distance$_{B \to A}$.)

3.3 Protocol

For this assignment, we will use the UDP protocol to exchange the routing information among the nodes. Each node will send its most recent routing table by building a UDP packet. The destination port is the listening port of the node to which the packet is being sent. The source port is an arbitrary UDP port the node is using to send the packet. The data field of a packet should include 1) the sending node UDP listening port number (to identify the sending node to its neighbor) and 2) the most recent routing table.

3.4 Routing Information Exchange

You are free to design the format and structure of the routing table kept locally by each node and exchanged among neighboring nodes.

1. Upon the activation of the program, each node should construct the initial routing table from command line info, and keep it locally.
2. Once the link and distance information for all the nodes are specified, the routing table information will be exchanged among network neighbors. Each node should send its routing table information to its neighbors *at least once*. (See the command syntax section for more detail on how the process starts). This is the base case before there are any table updates.
3. Using the Bellman-Ford algorithm, each node will keep updating its routing table as long as neighboring nodes send their updated routing tables information.
4. If there is any change in the routing table, a node should send the updated information to its neighbors.

Due to the nature of the UDP protocol, packets may be lost or delivered out of order to the application. Thus you may consider to add some kind of *time stamp* or *sequence* information to each packet (i.e. each routing table), so that each node can update its own routing table accordingly. Note that if the process is implemented properly, the routing information exchange should converge (stop) as soon as all nodes in the network obtain the routing tables with the shortest pathes information.

3.5 Command Syntax

The program name should be `dvnode`. For each node in the network, the command syntax is:

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

`dvnode` Program name.

`<local-port>` umber (1024-65534) of the node.

`<neighbor#-port>` umber (1024-65534) of one of the neighboring nodes.

`<loss-rate-#>` ink distance to the `<neighbor#-port>`.
resents the probability of a packet being dropped on that link.
signment you do not have to implement dropping of packets.
`neighbor-port>` and `<loss-rate>` for all your neighboring nodes.

`last` e information of the network. The proram should understand this arg as optional (he
nent, the routing message exchanges among the nodes should
kick in.

`ctrl+C (exit)` Use ctrl+C to exit the program.

3.6 Status Messages

The followin status messages should be displayed at least for the following events. The example of the messages are also shown below.

1. Routing message sent:

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

2. Routing message received:

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

3. 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>
⋮
```

3.7 Cases

Your program should deal with the following cases:

Distance Vector	Routing tables and their updates should follow DV algorithms	(20pt)
DV Updates	DV updates should be sent out to all neighbors once the routing table is changed	(10pt)
Convergence	Routing tables should eventually converge and all output should cease	(10pt)

3.8 Example

The example below illustrates the command inputs, routing message exchanges and the computation of the routing table at each node, given the network configuration described in Figure 3.
The command to start a program is:

C	`$ ./dvnode <local-port> <neighbor1-port> <loss-rate-1> <neighbor2-port> <loss-rate-2> ... [last]`
Java	`$ java dvnode <local-port> <neighbor1-port> <loss-rate-1> <neighbor2-port> <loss-rate-2> ... [last]`
Python	`$ python dvnode.py <local-port> <neighbor1-port> <loss-rate-1> <neighbor2-port> <loss-rate-2> ... [last]`

From the command line prompts type:

```
$ ./dvnode 1111 2222 .1 3333 .5
```

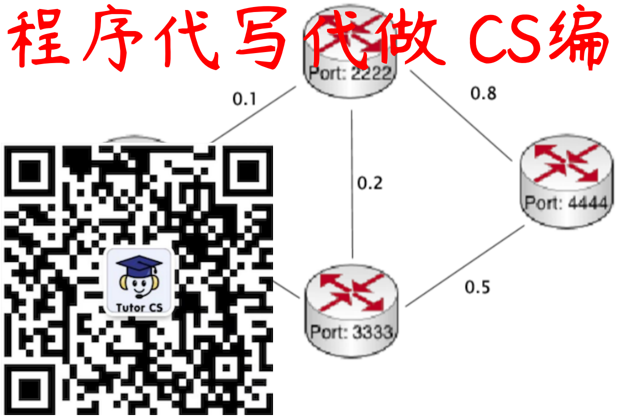

Figure 3: Network Configuration Example

```
$ ./dvnode 2222 1111 .1 3333 .2 4444 .8
$ ./dvnode 3333 1111 .5 2222 .2 4444 .5
$ ./dvnode 4444 2222 .8 3333 .5 last
```

As soon as the command for node 4444 is entered the following process should kick in:

1. Node `4444` send its routing information to node `2222` and `3333`.

2. Node `2222` and `3333` update their routing tables according to the routing information sent from node `4444`. Since both node `2222` and `3333` have not sent their routing information to anyone, node `2222` sends its routing information to node `1111`, `3333` and `4444`. Node `3333` will send its routing information to node `1111`, `2222` and `4444`.

3. Node `1111` should update its routing table according to the message received either from node `2222` or `3333` whichever came first. Node `1111` will send its updated routing information to node `2222` and `3333`. (At this point, all nodes have sent their routing information to their neighbors at least once.)

4. Each node will update its routing table according to the messages received from its neighbors. However, it will send messages only if its routing table is updated.

5. The process converges as each node has the most updated routing table which includes the shortest distances to all the nodes in the network and the next hop on the shortest path to each node.

Once the algorithm converges, there should be no more status messages. The last status message from the nodes will be something like (note: each node will print its own table):

```
[1353035852.173] Node 1111 Routing Table
- (.1) -> Node 2222
- (.3) -> Node 3333; Next hop -> Node 2222
- (.8) -> Node 4444; Next hop -> Node 2222

[1353035852.192] Node 2222 Routing Table
...

[1353035852.239] Node 3333 Routing Table
...
```

```
[1353035852.287] Node 4444 Routing Table
- (.5) -> Node 3333
- (.7) -> Node 2222; Next hop -> Node 3333
...
```

4 Combination

4.1 Description

As an ultimate goal for [illegible] 3N and DV protocols should be implemented and take effect in the emulated network. [illegible] eate reliable links, and the DV algorithms should determine the shortest paths over these [illegible] nment, it is only required to generate the correct routing table for each node using the two protocols. The distance of each link will be the packet loss rate on that link *calculated* by the GBN protocol.

4.2 Loss Rate Calculation

This is the incorporation of section 2.6, so it should not be additional work. To simplify the problem, we put some more specifications on the GBN protocol:

- Window size is always 5.
- The data packet will only be dropped in a probabilitic way.
- ACK will never be dropped.
- Timeout is still 500ms.

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

$$Link\ cost = \begin{cases} 0, \text{(initial value)} & \text{if no probe packets have been sent} \\ \frac{Total\ number\ of\ dropped\ packets}{Total\ number\ of\ sent\ packets}, & \text{otherwise} \end{cases}$$

A counter is needed for every link to get those numbers.

4.3 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. You should implement a sending interval. 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” in the arguments. 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.

4.4 Routing Information Exchange

The requirements for routing information exchange are similar to those in Section 3, except when the routing updates should be sent out to neighbors. The loss rate calculated for each link will keep changing rapidly, but you should not send updates to the network in the same frequency, as those packets would flood the entire network. For this assignment, we have the following requirements:

- The distances in the routing table should be rounded to the second demical 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 neighboring nodes, including all probe senders and receivers.

- The probe receive ated distance of the corresponding links through the updates sent by the probe send

4.5 Command Sy

Your program in this sec node" and should be started with the following command:

```
cnnode <local-port> rec oss-rate-1> <neighbor2-port> <loss-rate-2> ... <neighborM-port>
       <loss-rate-M> s  <neighbor(M+2)-port> ... <neighborN-port> [last]
```

`cnnode` Again, the program name.

`<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` The optional argument. 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.

`ctrl+C (exit)` Use ctrl+C to exit the program.

4.6 Example

For example, to start the network shown in Figure 4: The user should use the following commands:

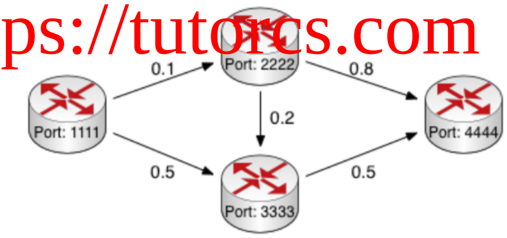

Figure 4: Network Configuration Example

```
$ ./cnnode 1111 receive send 2222 3333 (receiving list is empty)
$ ./cnnode 2222 receive 1111 .1 send 3333 4444
$ ./cnnode 3333 receive 1111 .5 2222 .2 send 4444
$ ./cnnode 4444 receive 2222 .8 3333 .5 send last (sending list is empty)
```

When node 4444 is started, it will send DV update (initially all with loss rate of 0) to both node 2222 and node 3333.

After receiving the DV updates, node 2222 and 3333 should both update their routing table, send out the changes,

and start sending probe packets. Node 2222 will send DV update to node 1111, 3333, and 4444, and start sending probe packets to 3333 and 4444. Node 3333 will send DV update to node 1111, 2222, and 4444, and start sending probe packets to only 4444.
When node 1111 receives the DV update from either node 2222 or node 3333, it should update its own routing table, send the DV update to node 2222 and 3333, and also start sending probe packets to node 2222 and 3333.
After that, there will be [illegible] DV [illegible] o routing tables will be changed. All nodes have started to count and calculate the loss rat[illegible] later, the DV updates will be sent out again and all routing tables will be updated. Note: *[illegible] nt of the loss rates input on the command line. The information enters the tables via the [illegible] tions.*

4.7 Status Messag[illegible]

The status messages sho[illegible] on 3.
Also, the packet loss rat[illegible] displayed every 1 second with the following format:

```
[<timestamp>] Link to <neighbor-port>: <sent-count> packets sent, <lost-count> packets lost, loss rate <loss-rate>
```

For example:

```
[1353035952.259] Link to 2222: 10 packets sent, 5 packets lost, loss rate .50
```

4.8 Cases

Your program should deal with the following cases:

Counter	The number of packets should be counted at the sender side	(5pt)
Loss Rate	The loss rate calculated at the sender should vary at first but eventually converge	(5pt)
Distance Vector	The routing tables should be updated periodically following the DV algorithm	(5pt)
DV algorithm	The routing tables should vary at first but eventually converge to the result of Section 3	(5pt)

5 Program Structure

The program should run an independent process per node. Within one process, there are two ways to operate:

- *single thread* operation - The process should listen to the port, receive an incoming packet, parse the packet data, and take actions (to update the local routing table and send out packets to its neighbors, or to check the sequence number and send back an ACK). Once all the actions are completed, the process goes back to port listening mode. The downsize of this approach is that normally there is a limited size input queue associated with each UDP port. This implies that while the process is busy taking above actions, incoming packets may be queued up and some of them may be lost. If you select this approach you have to verify that the routing tables still converge.

- *multi-threading* - The main thread within a process is always listening to its port. Once a UDP packet arrives to the port, a sub-thread is created to parse the message and take actions. By doing this, the main thread is always listening to the port so that you will not miss any incoming packet. (Obviously there are many ways to assign tasks to different sub-threads though.)

The program structure is left for you to design and implement, but make sure you test it thoroughly.

6 Submission Instructions

Please use C, Java, or Python for developing the emulation programs. You should develop on Ubuntu 14.04 LTS, which is available on the Google Cloud platform.
Before you begin programming, install the required packages for your choice of the languages above. Before proceeding, be sure to run `sudo apt-get update`.
If using C, use this command in the terminal to download gcc: `sudo apt-get install build-essential`

If using Java, use `sudo apt-get install default-jdk` to install the java 1.7.0 jdk. Please do not use Java 8.

If using Python, the Google Cloud Ubuntu 14.04 LTS images already have Python 2.7.6 as well as Python 3.4.3 installed. If using Python 3, clearly state so in the README. Otherwise, it will be assumed that Python 2 is used.

Make sure that your p[illegible] runs on a Google Cloud Ubuntu 14.04 LTS instance before you submit the packag[illegible] [illegible]ackage should include the following deliverables:

- `README`: The file co[illegible]entation, program features, usage scenarios, brief explanation of algorithms or data [illegible] description of any additional features/functions you implemented. This should be a t[illegible] *[illegible]ts if you submit a PDF, .rtf, or any other format.*

- `Makefile`: This file [illegible]lication. If you have written the programs in C (Unix), the output file names should be `gbnnode`, `dvnode`, and `cnnode`. If you used Java, the file names should be `gbnnode.class`, `dvnode.class`, `cnnode.class`. You do not need a Makefile if you used Python, but please name your files `gbnnode.py`, `dvnode.py`, and `cnnode.py`.

- Your source code.

- `test.txt`: This file should contain some output samples on several test cases. This would help others to understand how your programs work in each test scenario. It is optional to include this, as part of your `README` document.

The submission should be made via Courseworks. Zip all the deliverables mentioned above, and name the zip file as `<your UNI>_PA2.zip` (e.g. `gz2136_PA2.zip` for Professor Zussman). The zip file should expand to **one** directory containing all deliverables. Upload the file in the Courseworks, under `Assignments -> Programming Assignment 2`. It is highly recommended after submitting to download your submission, unzip it and test it on Ubuntu 14.04 again!

In the grading of your work, we will take the following points into account.

- The documentation clearly describes your work and the test result.

- The source code is complied properly by using the `Makefile` and generate appropriate output files.

- The programs run properly, including 1) take appropriate commands and arguments, 2) handle different situations and support required functions, and 3) display necessary status messages.

- The programs follow the command line formats exactly, and that the deliverables are presented in the format specified by the assignment.

Happy Coding and Good luck!!

□