Term Project 2

You need to work on this project individually. You should implement this project using C, C++ or Java since performance is not a concern.
Note: more details are still being added to this descriptions. Please let me know if you notice any missing or inconsistent or incorrect information.
I strongly encourage you to start this project ASAP since it has a number of details to sort out.

Overview: Your goal in this project is to implement a peer-to-peer bulletin board (forum) client application based on a ring overlay. Individual clients run as separate processes on the same machine (ix-dev). The basic idea is to organize participating clients (or nodes) into a ring-shaped overlay. This overlay is used for the sequential delivery of a post message (in a specific direction) from the posting client to all other clients along the ring. More specifically, a post by client c1 is only sent to its next hop neighbor c2 along the direction of the ring (say clock-wise). c2 then posts the message in its local board and relays the message to the next node on the ring. This process continues until c1 receives the same message which indicates that all other clients have successfully received the message. At that point c1 concludes that the message is properly delivered to all other nodes and does not forward this message any further. We only consider the case where all clients are running at different processes on the same computer. Each client uses a UDP socket to send (receive) its control and data messages to (from) its next (previous) hop in the ring. All (control and data) messages are sent and received through a single UDP socket by each client. We assume message transmission between nodes is reliable (i.e. no message is lost) since all processes are on a single computer. But your client should be able to properly deal with the arrival of new nodes and departure of existing nodes from the overlay by properly and quickly restructuring the overlay and also coping with the loss of messages between two clients as a result of node dynamics (i.e. a message does not going completely through the ring since a node has departed and caused the ring to be broken during the delivery). Below we provide further details about different components of this protocol.

Token Ring Mechanism: Clients use a token-ring mechanism to coordinate posting of their messages. There is a single (control) token message in the ring. Once a node receives the token from its upstream node, it can post any pending (available) messages on the board by sending them to the next hop node along the direction of the ring "one message at a time". Once a client ensures that all of its pending messages are successfully delivered, it sends the token to its next hop node in the ring. Messages (posts) are sent through the ring sequentially - that means a client should ensure the delivery of message m to all other nodes (by getting the message back through the ring) before it sends message m+1. This also implies that at any point of time, only one post message or the token is traveling through the ring. The token ring mechanism ensures that only a single client can post messages at any point of time.

Ring Formation Procedure: To form a ring, participating clients first need to discover each other. To make this procedure rather simple, we explicitly limit the nodes/clients that can participate in a ring and assign a globally unique ID to these nodes. These node IDs must be used for their ordering along the ring. Each client takes a configuration file as an input that specifies (among other things) a range of (at least 10) consecutive port numbers that can be used by participating clients on the same forum. The configuration file also provides the specific port that is assigned to that specific process/client (and must be used as its ID). Therefore, clients must be ordered along the direction of the ring (i.e. the direction of sending messages) in the order of increasing port number. Note that not all the available IDs would be used at the same time, e.g. there could be 20 available port numbers (IDs) but there are only 5 clients in the ring that use some random IDs from the available range of ports. In other words, consecutive nodes in the ring may not have consecutive port numbers.

Node Discovery: To form a ring-shaped overlay, each client is responsible to discover the proper next hop node in the ring that has the next larger port number among the participating nodes. To discover other participating (alive) nodes with higher port numbers, client x with port number p sequentially probes higher port numbers (starting from p+1 till the end of the range of assigned ports) until it finds a client. A probe from node x to another node y is a control message that contains the ID of the sending node x. Node y needs to quickly respond to a request to confirm that it is alive. A probing client x sets a short timeout to detect whether a probing node y exists (is alive) or not (i.e. no response within the timeout period means that node y does not exist). Once client node x finds the proper next hop y, x considers y as its next hops and forward all the received posts/messages to y. y also keeps the ID of x as its last (previous) hop and *only* forwards the messages that it receives from x. If y receives a discovery probe from another node ID z (where x is smaller than z, and z is smaller than y), then y considers z as a better previous hop. We discuss this case in more details later. Also if node x possibly sends a discovery probe to node z but z already has a more appropriate prior node y (i.e.

the ID for these nodes are x < y < z), then z sends a negative response to x. Given a collection of participating clients in the forum, each client should be able to find its proper next hop using the above discovery method. The collective effect of this step by all participants leads to the formation of a properly structured ring-shaped overlay.

Election Mechanism: In order to ensure that only a single token exists in the system, participating nodes need to run an election to select a leader. Once the leader is selected, it creates the only token in the ring. Every time that the ring is restructured/reformed due to the arrival/departure of one or more nodes, a new election must be conducted to create the only token in the ring. Once a node identifies its next hop, it waits for a random period of time and then starts an election. The random delay is selected as a random value (in second) between [0,1] second, e.g. 0.36s. To start an election, a node simply sends an election (control) message to its next hop and includes its ID (port number) as a candidate leader in that message. When another node receives an election message, it changes the ID of candidate leader in the message to its own ID if its own ID is larger than the one in the message. Otherwise, it simply forwards the message to its next hop if it has one. If a node does not have a next hop yet (because it is still looking for the proper next hop), it simply drops an election message and causes the election process to be incomplete. An election message goes through the ring in the usual direction until a node x receives an election message with its own ID as the selected leader in the message. In this case, x considers itself as a leader and sends an elected (control) message through the ring. An elected (control) message contains the ID of the elected leader and is sent to the next hop until the elected message returns to node x indicating that all nodes have received the elected message and thus are aware of the elected leader. Note that an elected message should not be forwarded by node x (that perhaps joined the ring after the election is completed) if x's ID is larger than the ID of elected node. Each election message should contain a unique identification so it can be identified as it goes through the ring. The node that starts an election by sending an election message should include a random election ID (from 0-100,000 range) in the message. Since an election or elected message could be dropped along the ring, a client that sends either of these messages needs to wait for proper turn around time for the message to come back through the ring. If it does not receive its own message within the proper turn around time, it assumes that the message is dropped. It waits for a random period of time and then starts a new election. Note that a participating node may receive duplicate copy of a control (or data) message. Therefore, each data or control message should include a per sender sequence number (of the original sender) that allows each participant to identify and ignore duplicate messages. Once the leader is elected and its identity is shared with all other participants, it generates the only token in the system and uses the election ID that led to its selection as the token ID. Then, the leader checks and sends all of its pending messages before passing the (newly generated) token to its next hop/client. The token message should not change as it travels through the ring and token ID (along with the ID of the leader that generated the token) is used to distinguish a specific token.

Managing Node Dynamics: We run all participating clients as separate processes on the same computer. Therefore, we assume that no packet is lost between two adjacent nodes in the ring. This implies that a client/node should receive a message or the token once every round in a properly-connected round. The complete traveling of a message through the entire ring is an indication that it is connected and lack of message arrivals suggests that the ring is broken (i.e. some nodes have left the ring). Each node maintains a timer that is reset every time that a control or data message arrives. If the timer at node x reaches the duration of a round, node x concludes that the ring is broken (i.e. one or more nodes have arrived or departed). This triggers node x to start the node discovery and ring formation procedure, wait for a random delay and if it does not receive an election message, it starts a new election, i.e. receiving a new election message would suppress a node from starting an election. Note that different nodes in the ring may detect that the ring is broken at the same (or different time). But the random delay after discovery and associated suppression mechanism would prevent all/many nodes to start separate elections at the same time. This prevents an election message by a node to go around through the ring. Therefore, each node should resend its election message until a leader is selected. If the election message does not go through the ring (i.e. ring is still broken), the timer at some nodes expire and at least one node starts a new election. This process continues until one election message goes through the ring and confirms that the ring is properly formed. When a new node x wants to join the ring, it simply tries to break the ring and force the discovery and ring (re)formation procedure. To this end, node x runs the discovery method to identify its proper next hop (node y) and requests y to be its next hop (as nodes do during the ring formation). Since each node accepts better previous node, this causes node y to only forward messages from node x and stop forwarding messages from its current previous node in the ring (i.e. basically breaking the ring). This in turn triggers various nodes in the ring to start the node discovery and election.

Input and Configuration Files: The format of your configuration, input and output files should be consistent with the provided description here. Having a consistent format allows us to test your code with a new input and configuration files. We will provide sample input and configuration files so you can test your code with our configuration files as well.

Estimating Timeout: Clearly having a long timeout value makes the recovery slower but prevents unnecessary/premature recovery. To simplify the design, each client can conservatively initialize the timeout value to a large value, say 2 seconds. But when a client posts a message, it should measure the time between sending the message to the next node until it receives the same message from previous node (i.e. the time for the message to go through the ring). This measured time provides a sample of "turn around time" (or TAT) for a message through the ring. The client can set the timeout=2*TAT as a conservative value. Keep in mind that we need to re-measure TAT for any new post since the size of the ring may change as nodes join/leave, and then use the most recent value to estimate timeout. Note that if none of the nodes has anything to send during an interval, the token message should be forward by all nodes rather quickly. Therefore, there is always a post or control message that regularly goes through different nodes of a connected ring.

Configuration File: Each node takes a separate configuration file as an input which contains the following information for each node:

- client_port: the valid range of valid port numbers for all participants,
- my_port: specific port (from the valid range) that is assigned to this client,
- join_time: join time shows the time when the client should join the ring,
- leave_time: leave time shows the time when the client should leave the ring.

the provided join and leave time allows you/us to run a test with a few clients with the need to manually start or end individual client code. Each line of the input file includes one of the above keywords (client_port, my_port, ...) and then ":" and then the proper information. Here is an example:

```
client_port: 3451-3461
my_port: 3455
join_time: 1:34
leave_time: 2:44
```

here is a sample of configuration file that you can download to test your code. Your code should be able to accept our configuration files with the same format.

Input File: To avoid the need for having a user to type post messages at each client, we provide a separate input file that contains a collection of posts and their associated timestamps to each client. Each client checks the local clock to measure the elapsed time. When a client receives the token, it examines the input file and posts all the unsent messages that their timestamp is smaller or equal that the local clock. Each line of the input file shows a timestamp for a post, and a tab and the text for the post, and CR and LF. Here is an example:

```
1:23 I am looking for an apartment around university
1:31 Is anyone interested in a group study for the final exam
2:01 I am selling my bike for $150
```

here is a sample of an input file that you can download to test your code. Your code should be able to accept our input files with the same format.

All the time information (for join, leave time in configuration and timestamp of posts in the input file) have the format of mm:ss where mm shows minute and ss shows seconds.

Output File: Each client process should dump its output in a separate file using the file name that is provided in the command line (with -o flag). The output file consists of the following status information that are appended to the output file on a separate line (one status information per line) when the corresponding event occurred at a client. The status information should **use the exact text** that we have specified below. The following list provides the list of status information (in italic font) and the description of the event that triggers the status information (the text after "-").

- *[time]: next hop is changed to client [port#]* - when this client detects a better (or its first) next hop
- *[time]: previous hop is changed to client [port#]* - when this client is contacted by client [port#] which is a better option (has a closer port number) than the current one to be this client's previous hop.
- *[time]: started election, send election message to client [port#]* - when this client starts an election and sends the first message to next hop client [port#]
- *[time]: relayed election message, replaced leader* - when this client send the election message to next hop and replaces the suggested leader in the message to itself

- *[time]: relayed election message, leader: client [port#]* - when this client relays an election message that contains client [port#] as the leader
- *[time]: leader selected* - when this client is selected as the leader
- *[time]: new token generated [#RandomTokenID]* - this should be reported by the leader when it generates a token with token ID of #RandomTokenID
- *[time]: token [#RandomTokenID] was sent to client [port#]* - when this client sends the token to next hop node which is client [port#]
- *[time]: token [#RandomTokenID] was received* - when this node receives the token
- *[time]: post "CCCCC" was sent* - indicating that this client has posted a message with the content of CCCC
- *[time]: post "CCCCC" from client [port#] was relayed* - indicating that this client receives a post that was original sent by client [port#] with the content of CCCC and relayed it to its next hop
- *[time]: post "CCCCC" was delivered to all successfully* - indicating that this post by this client was delivered back to this client after going through the entire ring
- *[time]: ring is broken* - when this client detects that the ring is broken

In the above status information: [time] indicates the local time when an event occurs at a client; client [port#] specifies a particular client (depending on the message this would be a different client). **Note:** You do NOT need to include [ ] around time or port#. Time should be reported in mm:ss format. (e.g. 1.32) and [port#] should be replaced with an integer that shows the corresponding port number.Similarly [#RandomTokenID] should be replaced with an integer that represents the corresponding token ID. "CCCCC" should be replaced by the text of a post with " on both ends (and no extra white space). Your code should closely follow this formatting convention and do NOT produce any other (unnecessary) output message.

Packet Format

You are required to use a consistent packet format for all the data and control messages that your client sends to other clients. This would facilitate the inter-operability of your code with other clients. Each message begins with a 32-bit type identifier. This identifier indicates what type of message the UDP datagram contains. By examining this type, the application can determine how to parse and interpret the rest of the message. The type identifier is shown in parenthesis.

- **Post(0)**: The message contains a 32-bit source ID for the client that originates the post and a 32-bit source-specific sequence number along with the string of character for a client's post

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (0)             |
+---------------------------------------------------------------+
|                 32-bit source ID                              |
+---------------------------------------------------------------+
|                32-bit source-specific sequence number         |
+---------------------------------------------------------------+
|                 user-post[string of char]                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Probe(10)**: The message contains a 32-bit source ID for the client that originates the discovery probe, a 32-bit source-specific message ID so the source can distinguish the responses to different requests.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (10)            |
+---------------------------------------------------------------+
|                 32-bit source ID                              |
+---------------------------------------------------------------+
|                32-bit source-specific message ID              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Probe ACK(11)**: The message contains a 32-bit source ID for the client that is directly responding to a probe as well as message ID of the probe indicating that the request for being the next hop is accepted.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (11)            |
+---------------------------------------------------------------+
|                       32-bit source ID                        |
+---------------------------------------------------------------+
|                32-bit source-specific message ID              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Probe NAK(12)**: The message contains a 32-bit source ID for the client that is directly responding to a probe as well as message ID of the probe indicating that the request for being the next hop is rejected.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (12)            |
+---------------------------------------------------------------+
|                 32-bit source ID                              |
+---------------------------------------------------------------+
|                32-bit source-specific message ID              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Election(20)**: The message contains a 32-bit ID for the client that is initiated the election, unique election ID, and 32-bit ID of the best client candidate (highest ID) so far.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (20)            |
+---------------------------------------------------------------+
|         32-bit client ID that initiated the election          |
+---------------------------------------------------------------+
|                     32-bit Election ID                        |
+---------------------------------------------------------------+
|          32-bit ID of the best candidate client so far        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Elected(21)**: The message contains a 32-bit ID for the client that is initiated the election, unique election ID, and 32-bit ID of the elected client.

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (21)            |
+---------------------------------------------------------------+
|         32-bit client ID that initiated the election          |
+---------------------------------------------------------------+
|                     32-bit Election ID                        |
+---------------------------------------------------------------+
|                32-bit ID of the elected client                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- **Token(30)**: The message contains a 32-bit ID for the client that generates the token, unique token ID (reusing the corresponding election ID).

```
0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                32-bit message type identifier (30)            |
+---------------------------------------------------------------+
|         32-bit client ID that created the token               |
+---------------------------------------------------------------+
|                      32-bit Token ID                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Some suggestions: We suggest that you take the following steps to gradually build your code:

- develop the code for two nodes to communicate with each other through UDP socket, and then add a timer to your code so your client can manage sending and receiving message through the same UDP socket while reacting to a timer as well.
- create a three node chain and implement the message sending and forwarding among these nodes, and add the ability to obtain information from the config and input files.
- implement the discovery and ring formation procedure
- implement and test the random delay and election mechanism.
- implements the robustness against node addition and node removal.

Running your code: We use a simple script (such as the following) to create multiple instances of your client and pass the input, configuration and output file to each instance of your program

```
%ring -c cfg1.txt -i in1.txt -o out1.txt &
%ring -c cfg2.txt -i in2.txt -o out2.txt &
%ring -c cfg3.txt -i in3.txt -o out3.txt &
...
```

here "ring" is the name of your executable file for client, -c flag indicates the configuration file, -i indicates the input file and -o indicates the output file. You must use this name for your executable and your program should be able to parse these input parameters that specify the names for input, output and configuration files.

Evaluation & Grading: The grading of your program is based on the following elements:

- your code complies with the provided packet formats: 10%
- your code behaves properly in the absence of node dynamics: 30%
  - your code can form the proper ring: 10%
  - your code is able to conduct an election: 10%
  - your code is able to deliver posts from each client to all clients: 10%
- your code behaves properly when new nodes are added to the ring: 30%
  - your code can form the proper ring: 10%
  - your code is able to conduct an election: 10%
  - your code is able to deliver posts from each client to all clients: 10%
- your code behaves properly when nodes are removed from the ring: 30%
  - your code can form the proper ring: 10%
  - your code is able to conduct an election: 10%
  - your code is able to deliver posts from each client to all clients: 10%

Note: There is 20% penalty if your code generates outputs with improper format or generates unnecessary messages (other than those we specified).

Turning in your Assignment: Make sure that your executable has the proper file and your code accepts the input file formats and can take all file names on the command line as we described earlier to avoid any problem during its evaluation. To submit your project, you should email me single tar file as attachment. Untaring the tar file should create a directory with your last name that contains the following

- all your files,
- completed readme2.txt file, and
- a Makefile to compile your code.

The subject line of your email should be "CIS630: Term Project II".

Good Luck!