COMP 4320 Introduction to Computer Networks

Programming Assignment (**Group,** 100 pts)

IMPORTANT:

1) *Your code will be tested and graded on the Engineering Unix (Tux) machines. If the code does not work on those machines, you will not get **any** credit even if your code works on every other machine in the universe.*
2) *A late submission will get a 50% penalty if submitted at 10:00pm or later on the due day. After 10:00pm the next day, you cannot submit the lab.*
3) *One submission per group.*
4) *Writing and presentation of your report are considered to grade your lab.*
5) *The quality of your code will be evaluated.*

Programming Assignment (Turned in by one group mate)

First, it is assumed that by now, 1) you have an engineering Unix account, 2) you can edit, 3) you can compile, and 4) you can execute C programs on the Unix (Tux) machines. You can use any personal computer or computing lab to remotely access the Engineering Unix machines.
Second, it is assumed that you have a group and a group ID (**GID**). If you do not yet have a group ID, you must ask for now. If working alone, you will get a **penalty of 5 points per lab**: I want you to learn to work on a team and deal with **human** problems. You must have a group ID to avoid "port numbers" conflicts with other groups. If you do not have a **GID** at LEAST one week prior the submission day, you will get 20 points off this assignment.

Lab 2: **Establish a Circle (ring) of Trust (Phase I)**

Lab must meet two key requirements:

1) your client must interoperate/work with any (working) server of other classmates, and
2) your server must interoperate/work with any (working) client of other classmates.

Languages:

1) **you must use C, C++, or Java for the server AND any other language of your choice for the client.**
2) **The client cannot be written using C, C++, or Java.**
3) **Your code must ultimately compile and execute on Tux engineering machines.**
4) **The TA will use default packages on the Tux machines. He will not install new packages.**

Creating A Virtual Ring

The objective of this lab is to create a *virtual* ring of nodes over the Internet. After joining the ring, the nodes on the ring MUST use only the ring to communicate. The ring is managed by a **master** which is part of the ring and has Ring ID 0 (zero). All the other nodes of the ring are slave nodes (clients) with a ring ID assigned by the master.

The implementation of the ring will take two phases:

Phase I (Lab 2): set up the virtual ring
Phase II (Lab 3): enable communications within the ring.

Phase I is the assignment for Lab 2 and Phase II is the assignment for Lab 3.

We will assume that once a node joins, it will never leave the ring, shutdown, or fail.

Ring Set Up

- Initially, the ring consists only of the **master**. The master is a stream (TCP) server that manages the ring: it assigns to each joining node a ring ID *myRID* and the IP address of their successor node (*nextSlaveIP)*. The master binds to Port Number 10010+(GID%30)*5 where GID is the group ID of the group who implemented the master. For example, if the master's GID is 2, then the master must bind to Port 10020.
- The master can run anywhere on the Internet (if no filtering!).

a) **Slave Node** Operations

Write a **client** (**Slave.xxx**) in any language other than Java, C, or C++. The Client must

i. accept a command line of the form: ***Slave MasterHostname MasterPort#*** where
   1. ***Slave*** is the executable,
   2. ***MasterHostname*** is the master's hostname,
   3. ***MasterPort#*** is the master's port number.

ii. form and send a ***Join Request*** following the protocol described below**.**

iii. set itself as a slave node on the ring following the protocol described below and obtain its ring ID (***myRID***) from the master**.**

iv. Display the GID of the master, its own ring ID, and the IP address (in dotted decimal format) of its next slave.

v. (**Phase 2** only), repeatedly prompt the user for a ring ID ***RID*** and a message ***m***.

vi. (**Phase 2** only), send the message ***m*** to the trusted node with ring ID ***RID*** following the protocol described below.

- (**Phase 2** only), **display** any message received in a packet that contains ring ID ***myRID***.
- (**Phase 2** only), **forward** any message destined to a node that has a ring ID different from ***myRID***. Forward only message with a ***TTL*** (time to live) higher than 1.

How does a Slave Node Join the Ring?

First, the slave node must send a **Join Request** to the master at *MasterHostname* on Port Number *MasterPort*. The join request must have the following format:

1 byte	4 bytes
GID	0x4A6F7921

where ***GID*** is the group ID of the group who implemented the slave node (Slave.c) and ***0x4A6F7921***[*] is a magic number (used by the nodes to test the validity of

[*] Find out what this magic number means. Each byte is the ASCII code of a character.

messages using this protocol). The master will ignore the request if the message is not valid (different from 5 bytes or not containing the magic number).

Second, the slave must wait for a response from the Master node. This response must have the following format:

1 byte	4 bytes	1 byte	4 bytes
GID	0x4A6F7921	yourRID	nextSlaveIP

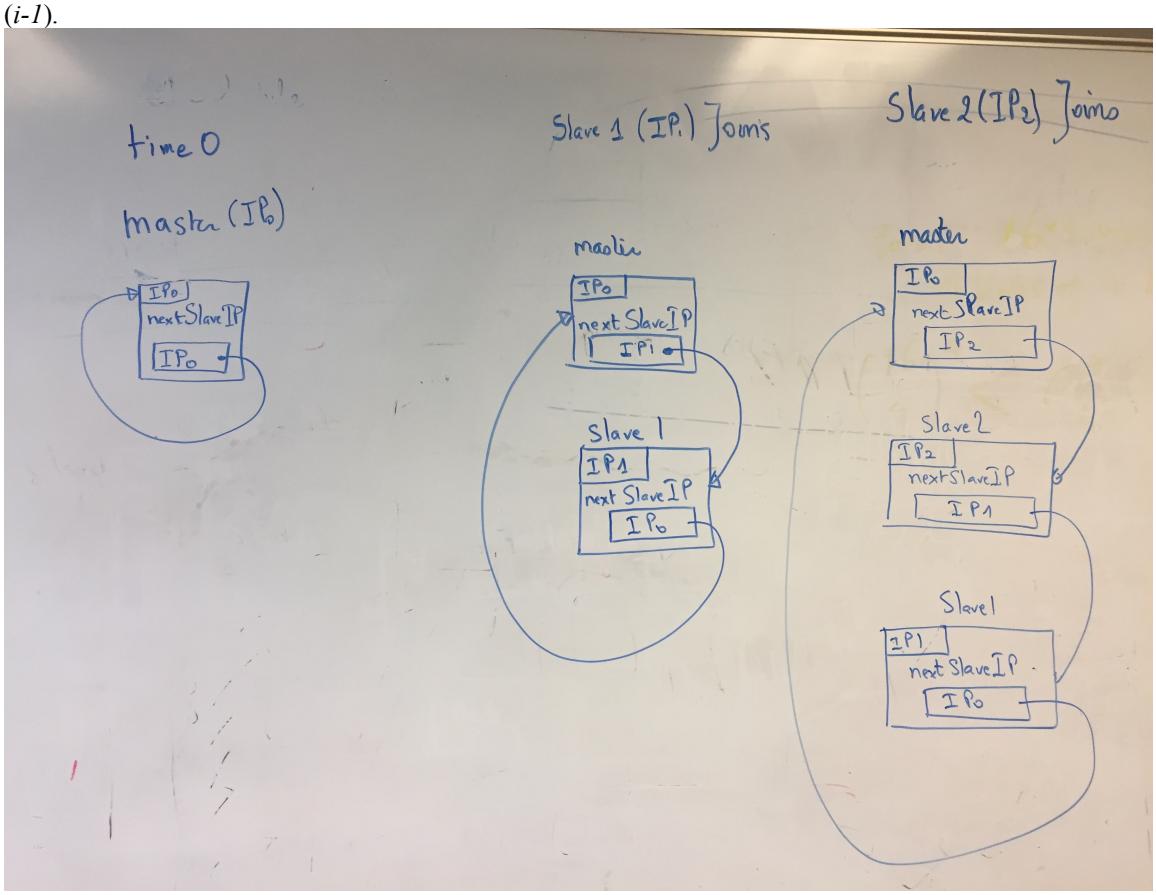
Where 1) ***GID*** is the group ID of the group who implemented the Master, ***0x4A6F7921*** is the magic number, 3) ***yourRID*** is the Ring ID that the Master assigns to the requesting slave, and 4) ***nextSlaveIP*** is the IP address of the slave just after the requesting slave on the ring

Whenever a slave ***S*** receives a message not meant to it, it must forward it to its successor *NextSlave* on the ring. Slave *S* has the IP address of the *successor* slave in its variable ***nextSlaveIP***. *nextSlaveIP* is determined and assigned by the master node as follows: the IP address *nextSlaveIP* that the Master sends to the requesting node is the IP address of the latest node that successfully joined the ring.

Scenario

Let us consider a scenario to explain how *nextSlaveIP* is maintained and assigned by the master. Let the master have IP address $\boldsymbol{IP_0}$. Initially, the master is the only node on the ring (see Figure), so it is the latest node successfully joining a ring of one node. Its *nextSlaveIP* variable is initially set to its own IP address IP_0. Let a slave node S_1 with IP address $\boldsymbol{IP_1}$ requesting to join the ring. When the master node receives the request to join, it will perform two actions: 1) it will send to Node S_1 the address IP_0 as the *nextSlaveIP* for S_1, and 2) it will set its own variable *nextSlaveIP* to IP_1 (i.e., the IP address of the latest node to successfully join the ring – namely the node S_1). Now the ring has two nodes: the master (Ring ID = 0, IP address = IP_0, *nextSlaveIP* = IP_1) and Node S_1 (Ring ID = 1, IP address = IP_1, *nextSlaveIP* = IP_0). Suppose now that Node S_2 with IP address IP_2 requests to join. Similarly, the master will perform two actions: 1) it will send to Node S_2 the address IP_1 (IP address of the latest node to successfully join) as the *nextSlaveIP*, and 2) it will set its own variable *nextSlaveIP* to IP_2 (see Figure). Now Node S2 is set as: Ring ID = 2, IP address = IP_2, *nextSlaveIP* = IP_1.

Note that if a slave node has ring ID *i*, then its successor on the ring has always ring ID (*i-1*).



Third, after receiving the valid response (with magic number and not corrupted) from the master, the slave must establish a forwarding (defined later) **datagram** server at Port *10010+(GID%30)*5+yourRID* where *GID* is the Group ID contained in the response from the master. In other words, *GID* is the group ID of the group who implemented the **master**. For example, if the GID of the master is 1 and the master sent you a response with *yourRID* equal to 4, then you must set up a forwarding datagram server at Port 10019.

Forwarding/Reception Service (Phase II, Lab 3)

Recall that each slave *S* establishes a Forwarding Datagram Server. Whenever this Forwarding Server of Node *S* receives a message with a ring ID ***RID*** as destination, Node *S* checks whether the datagram is not corrupted (checksum) and checks whether *RID* is equal to its own ring ID. If yes, then this means that the datagram is meant for this receiving slave *S* and *S* will display the string payload m carried by the datagram, otherwise Node *S* will forward the full (AS IS) received datagram to the node slave with IP address *nextSlaveIP*. Example, consider a node ***N*** that has ring ID 3 and *nextSlaveIP* set to IP_2. Suppose Node *N* receives a datagram with destination RID 1. Since Node *N* has RID 3, it must forward the datagram to the node with IP address IP_2.

Sending Service (Phase II, Lab 3)

Recall that after setting up itself as a slave, the node must repeatedly prompt the user to ask a ring ID ***RID*** and a message ***m*** where RID is a ring ID. When the user provides *RID* and *m*, the slave ***S*** must send to its successor a datagram formed as follows:

1	4 bytes	1 byte	1 byte	1 byte	Up 64 byt.	1 byte
GID	0x4A6F7921	TTL	RID Dest	RID Source	Message *m*	Checksum

where 1) ***GID*** is the group ID of the group that implemented the slave, 2) **0x4A6F7921** is the magic number, 3) ***TTL*** is the time to live, 4) ***RID Dest*** is the ring ID of the destination node, 5) ***RID Source*** is the ring ID of source node ***S***, 6) the message m, and 5) a checksum to detect a corrupted datagram.

The ***checksum*** of a datagram *dgm* is computed as follows: consider the datagram as an array of bytes. The checksum must be computed as an 8-bit Internet checksum. The 16-bit Internet checksum is described on Page 212 (textbook). Let me know if you have a problem computing the checksum. See exercise page 253, # 15. Here is what is peculiar about computing the checksum: *let Sum be the ONE BYTE current sum. Whenever the addition (Sum+dgm[i]) produces a carry, add this carry to Sum before processing the next byte*. Ultimately, *Checksum is the* bitwise one complement of *Sum*.

IT IS CRITICAL that all groups compute correctly (and the same way) the checksum. Otherwise, a receiver from a different group will reject/drop your messages.

b) **Master Node** Operations

Write a **Master** (**Master.c**). The Master must

i. accept a command line of the form: ***Master MasterPort#*** where
   1. ***Master*** is your executable,
   2. ***MasterPort#*** is the port number (where the master must bind)

ii. set itself as a stream (**TCP**) server. The master must maintain ring IDs and the variable *nextSlaveIP* to assign to slave nodes on the ring following the protocol described above**.** The master's ring ID is 0. The master must maintain two variables: *nextRID* and *nextSlaveIP*.

The variable ***nextRID*** is initially set to 1. Whenever a slave joins the ring, the master will send it the value contained in Variable *nextRID* and will increment *nextRID*.

The variable ***nextSlaveIP*** is initially set to the IP address of the machine on which the master is running. Whenever a slave *S* joins the ring, the master will send it the IP address contained in Variable *nextSlaveID* and will set its variable *nextSlaveIP* to the IP address of Node *S* (who just requested to join).

iii. (For Phase 2 only), repeatedly prompt the user to ask a ring ID ***RID*** and a message ***m***.

iv. (For Phase 2 only), send the message ***m*** to the trusted node with ring ID ***RID*** following the protocol described below.

v. (For Phase 2 only), **display** any message received in a packet that contains ring ID ***0***.

vi. (For Phase 2 only), **forward** any message destined to a node that has an RID different from ***myRID***.

Grading:

1) 50 points per program (One client and one server)
2) Code does not compile on a Tux machine: 0% credit
3) Code compiles on Tux machines but does not work: 5% credit
4) Code compiles on Tux machines, works somewhat but does not meet all requirements: 30% credit,
4) Code meets all requirements and interacts correctly with counterpart from the **same** group: 70% credit
5) Code meets all requirements and interacts correctly with counterpart from **other** groups: 100% credit

Advice to complete these exercises:

Lab 2 requests much more effort than Lab 1. This is not a lab you can complete in the last two days before the deadline: many issues must be solved well ahead. Break the lab in pieces that can be independently developed by each teammate. Insure that teammates are satisfactorily progressing. It is the responsibility of each teammate to timely deliver working programs meeting all requirements.

"How to get started?":
This is just an introduction to socket programming: I advise to work **ACTIVELY** to implement these programs.

Step 1: download, compile, and execute Beej's sample programs (UDP-client.c and UDP-server.c)
Step 2: get familiar with this code: study key socket programming calls/methods/functions
Step 3: improve the server to echo (send back) the string it received.
Step 4: improve the client to receive the echo and print it out.
Step 5: SAVE the improved versions (you may need to roll back to them when things will go bad)
Step 6: All you need now is "forming" your messages based on the specified format (lab 1 protocol).

Help Tools:
If needed, refer to the following programs:

1) **TCP-server.c** (Beej's stream server)
2) **TCP-client.c** (Beej's stream client)
3) **UPD-server.c** (Beej's datagram server)
4) **UDP-client.c** (Beej's datagram client)
5) **TCPServerDisplay.c** : this program is a modification of Beej's TCP server: it takes as input a port number to run on and displays in **hexadecimal** what it receives.
6) **UDPServerDisplay.c :** this program is a modification of Beej's UDP server: it take as input a port number to run on and display in **hexadecimal** what it receives.
7) **packedStruct.c** : this program demoes how to create a "packed" struct in C.

Do not hesitate to ask in class or during office hours about how to use any of the above programs.

What to turn in?

1) **Electronic copy** of your report (**standalone**) and source code of your programs (**standalone**). **IN ADDITION,** all source code programs must be put in a **zipped folder** named lab1-name1-name2-name3 where *name1*, *name2,* and *name3* are the last names of three teammates. Zip the folder and post it **TOO** on Canvas.
   Make sure to submit **separately** (not inside the zipped folder) the report as a Microsoft or PDF file as well as the source code. **A penalty of 25 points will be applied if these instructions are not followed.**

2) Your report must:
   a. state whether your code works
   b. Clearly explain how to compile and execute your code. If the TA needs your presence to compile and execute your code, then a **30% penalty** will be applied.
   c. **If** needed, report/analyze (as appropriate) the results. The quality of analysis and writing is critical to your grade.

   Good writing and presentation are expected.

If the TA is unable to access/compile/execute your work based on your report, 30% will be applied.
If the turnin instructions are not followed, 25 pts will be deducted.