1 Overview

Your task is to complete a UDP peer-to-peer client which contacts a registration server to obtain a peer ID. Your peer will also contact the registration server to download a list of available peers you can contact. With this contact list you can send messages between peers.

Your code must work in the UNIX environment.

You may use any programming language you wish, but like the previous programming assignment, we recommend you use Python as its the supported language in this class.

Remember Postel's Law/the Robustness Principle: *Be conservative in what you send, be liberal in what you accept.* Handle errors (for example, malformed messages) as gracefully as you can.

1.1 Message Format

Messages in this protocol will have pairs of field names and values. A field name and its value will be separated by a colon. In every message, these fieldname:value pairs will be separated by a semicolon.

All messages will have the following format:

```
SRC:###;DST:###;PNUM:#;HCT:#;MNUM:###;VL:xxx;MESG:yyy
```

In the above format (and throughout this document), # denotes a digit (0-9). xxx may be empty or may be a comma separated list of three digit numbers and yyy may be blank or a series of up to 200 ASCII characters. yyy can be made up of any printable ASCII characters *except* the following characters:

```
' " ; :
```

The fields in the message are as follows:

SRC denotes the three digit ID of the sender of the message and may have a value between 0 (000) and 999. 000 is a reserved ID, as is 999.

DST denotes the three digit ID of the intended recipient of a message and may have a value between 0 (000) and 999. Again, 000 is a reserved ID, as is 999.

PNUM denotes the protocol message number and may have a value between 0 and 9. The following is a list of what each protocol message number indicates:

PNUM value	Description
1	Registration request
2	Registration response
3	Data
4	Data confirmation
5	Request for current peer registry
6	Current Peer Registry
7	Serial Broadcast
8	Serial Broadcast confirmation
0	Error message

HCT denotes the hop count—the remaining number of times a message may be forwarded on and can be between 0 and 9. The HCT field is only relevant for messages being forwarded (discussed later).

MNUM a message ID number. Your client will generate these for every newly written message originating at your client. The message ID will help you match requests and responses. Even numbered PNUM messages are responses, and will have the same message ID as the request which prompted the reply. For example, if you send data (PNUM:3) with a message ID of 123, the data confirmation (PNUM:4) response should also have message ID 123. Generate message IDs in order, starting with whatever number you wish—we recommend starting with 100 so you already have a three digit value.

VL a comma separated list of three digit peer IDs which indicates which peers have already seen a forwarded message. Like HCT, this value is only important for messages being forwarded (discussed later).

MESG the message contents.

2 User Interface and Detailed Instructions

There are five "steps" to completing this project. Each is described in detail in the following subsections.

Upon starting, your program should:

1. Connect to the registration server at eig.isi.edu:63682, and register with the name server (Step #1).
2. Then, run a loop which waits for keyboard input and understands the following commands:

   ids as described in Step #2

   msg ### <string> as described in Step #3.

   all <string> as described in Step #4

Commands may be lower-case, upper-case or a mix of upper and lower case. Commands are entered via typing and then pressing the <enter> key.

2.1 Step #1, Register - worth 18 points

Send a message to the registration server at eig.isi.edu:63682, and register with the server. The server will give you a numeric name consisting of three digits.

The message for a registration request should be:

```
SRC:000;DST:999;PNUM:1;HCT:1;MNUM:###;VL:;MESG:register
```

where ### denotes any three digit number you generate for a message ID.

The server's response will be:

```
SRC:999;DST:XXX;PNUM:2;HCT:1;MNUM:###;VL:;MESG:registered as XXX, IP@port
```

where ### denotes the same three digit message ID number in your request, and XXX denotes a three digit number which is your new peer name. IP@port is what the registration server believes is the IP and port your peer can be reached at.

In the case of an error, the server *may* respond with a useful error message. Error messages have the following format:

```
SRC:999;DST:000;PNUM:0;HCT:1;MNUM:###;VL:;MESG:Error...
```

where again ### denotes the same three digit message ID number in your request, and ... may be some string of characters with a message (any printable ASCII without the characters ' " ; :).

Print out the MESG portion of any Error messages you receive from the registration server.

Upon successfully obtaining an ID, print out your obtained ID like so:

```
Successfully registered. My ID is: ###
```

2.2 Step #2, Pull Registry - worth 18 points

When the user requests a registry of available peers (by typing "ids"), pull down the registry from the registration server. The registry is the list of recently registered and potentially available peers.

The message to request the registry should be:

```
SRC:###;DST:999;PNUM:5;HCT:1;MNUM:###;VL:;MESG:get map
```

The response (if there are no errors) will be:

```
SRC:999;DST:###;PNUM:6;HCT:1;MNUM:###;VL:;MESG:ids=###,###,###,###,...and###=ip@port,###=ip@port,...
```

The "ids=" list is a list of peer IDs that have recently been assigned or seen as active. This indicates these peers may be available to contact.

The "and" is followed by a second list of *some* peer ids and their addresses in ip@port format. Each ip is an IPv4 address in dotted decimal format and port is a port number in decimal.

You will get a subset of addresses for available peers.

Print out the received info with similar format to the following:

```
********************
Recently Seen Peers:
###,###,###,###

Known addresses:
###     ip   port
###     ip   port
********************
```

2.3 Step #3, Send and Receive Data - worth 18 points

Implement the ability to send and receive short messages over UDP.

Loop waiting for either the user to ask to send a message (the "msg" command) or the "ids" command.

If a message arrives with PNUM:3 with a DST which matches your given ID, print it to the screen, and respond to the sender with an ACK as follows: Swap the values of the SRC and DST fields, keep the MNUM value the same, and respond with the following format:

```
SRC:###;DST:###;PNUM:4;HCT:1;MNUM:###;VL:;MESG:ACK
```

If the user indicates that they wish to send something (e.g. by typing "msg") followed by a destination (a 3-digit number from 001 to 998, presumably from the registry you showed the user in Step #2) followed by a message, send the message to the IP address and port via UDP if you have an IP and port which match with the indicated destination.

If you do not have an address for the ID indicated by the user, *for now*, skip sending anything. Once you complete Step #5 (forwarding), you can forward these messages on to peers of your choosing instead.

When you send the message (denoted by "message" in the following example) to the appropriate IP address and port, the format should be:

```
SRC:###;DST:###;PNUM:3;HCT:1;MNUM:###;VL:;MESG:message
```

Remember that messages should only contain certain characters and be no more than 200 characters in length. It's best if you double check and remove characters which are not allowed by protocol before sending, and truncate messages which are too long.

Repeat sending this message up to 5 times until you get a matching response with the SRC and DST identifiers swapped with the same MNUM field and PNUM:4 and MESG:ACK. Pause between retrying messages (hint: using select with a timeout is an easy way of doing this). If you do not get a confirmation after 5 tries, give up and print an error similar to:

```
********************
ERROR: Gave up sending to ###
********************
```

2.4 Step #4, Serial Broadcast - worth 18 points

Next, implement the ability to do serial broadcast. When a user requests to send a broadcast message by typing "all" followed by a message, send one copy to every peer in your registry which you have an IP/port address for (except yourself).

Just like any message which originates from your peer, you generate a three digit MNUM and use your given three digit ID as the value in the SRC field. The broadcast message copies will differ *only* in the DST field (which should match the ID of each peer you send to). When you broadcast the message (denoted by "message" in the following example) the format should be the following:

```
SRC:###;DST:###;PNUM:7;HCT:1;MNUM:###;VL:;MESG:message
```

Send the message up to 5 times to each of the peers you are able to contact until you get a confirmation response (with PNUM:8, SRC as your ID and MNUM matching the MNUM you originally sent out with your message). If you give up on sending to any of the peers, print an error.

If you receive a broadcast message (PNUM:7) correctly, respond to the sender by swapping the SRC and DST identifiers, keeping the MNUM the same as set by the broadcast sender, set PNUM:8 and MESG:ACK.

Also print any broadcast messages you receive to the screen following this example:

```
********************
SRC:### broadcasted:
(The message that you received from the broadcaster etc. etc. etc.)
```

2.5 Step #5, Forwarding - worth 18 points

In step 3, you implemented handling incoming messages with PNUM:3 that are to your peer ID. Augment your peer code to handle PNUM:3 messages which you receive which are *not* addressed to you (the DST value is not your peer's ID) by responding with a confirmation to the sender (swap the message's original SRC/DST identifiers and PNUM:4, HC:1 MESG:ACK) and then forward the message you received.

Forward messages by first checking the original message's HCT and do the following:

If the HCT field in the message you receive is 0, print out the message:

```
********************
Dropped message from ### to ### - hop count exceeded
MESG: (Change this to be the message from the dropped packet)
```

If HCT>0, check the VL field list.

If HCT>0 and your peer ID IS in the VL list, then print out the message similar to the following:

```
********************
Dropped message from ### to ### - peer revisited
MESG: (Change this to be the message from the dropped packet)
```

If HCT>0 and your node ID is NOT in the VL field list, then send a relay copy to up to 3 nodes from your registry list which you have addresses for EXCEPT yourself.

In this message you send out, decrement the HCT count in the message by 1, retain the original SRC/DST/PNUM/MNUM/MESG fields and append your three digit peer ID to the VL list.

Just as in the previous steps, if you do send a message out, resend this message up to 5 times to each peer you send to while looking for a confirmation before giving up. If you fail 5 times and give up, print an error.

In Step #3 if a user indicated they wanted to send to an ID you did not have an address for (using the msg command), you were instructed to drop the message. Once you've implemented forwarding of messages, you can now handle these messages by sending these out to up to 3 peers from your registry list which you do have an address for. In these messages, set HCT to 9 and add yourself to the VL list.

3 Turn In Instructions

You must submit a ZIP (.zip) or GZIP'ed TAR file (.tar.gz). These will be the *only* formats accepted. YOU MUST NAME YOUR ARCHIVE FILE USING YOUR USC USERNAME: E.G. xxx.tar.gz or xxx.zip, where xxx is your USC username.

Email your .zip or .tar.gz file to bartlett@isi.edu BEFORE 8pm on the deadline.

Included in your ZIP or TAR(.GZ) file should be the code for your peer and a README file (see below). *Do not turn in any binaries - only text files (eg no compiled code/executables).*

We suggest you use Python and name your main file "peerchat.py"

Your README file is worth points, so leave yourself time to complete the README. This README file should be plain text (no other format will be accepted) and should include the following headings/sections. Each of these items should be *clearly marked*:

1. AUTHOR: Your NAME and EMAIL.
2. BUILD: Instructions on how to compile your code under UNIX. Use a Makefile if possible. If you use Python (or some other scripting language), simply put "None" for this item in your README.
3. RUN: Instructions on how to run your code. For example:

   ```
   % python3.2 peerchat.py
   ```

4. BUGS: List things you know you were not able to complete or do correctly.

4 Hints

1. Your program must simultaneously be listening for incoming network messages *and* listening for input from the keyboard. Think through where your program is spending its time! For example, if you have a single thread of execution, and you're blocking on a socket read, or spending time in a sleep function, your program is not listening for input from stdin!
2. There are many ways to complete this assignment. Using a single select() or poll() call in a while loop is a really good way to go (and will allow you to listen for input from multiple sources such as stdin and your open UDP socket), but you can also choose to approach this with multiprocessing/multithreading.
3. There is a wealth of information online as to how to write a simple networking program. Take advantage of resources such as online tutorials, http://stackoverflow.com/, https://docs.python.org/ etc.
4. Non-blocking socket reads/writes are highly appropriate for this assignment.
5. Note that numbers used in messages have a set number of digits, regardless of how large or small the number value is. The message ID, TO/FROM IDs, IDs in the VL list all are three digits long, so if the numerical value of these fields is less than 100, you must pad the number out. For example, ID "3" would be "003". Be aware of how your programming

language of choice handles such strings in conversion to integer values (if you choose to convert such padded strings to numerical values). In Python, you can (if you need to), convert an integer to a zero padded string via "format". For example the following will print '004':

```
x='{0:03d}'.format(4)
print(x)
```

6. Note that fields in the messages are separated via colons. If you use Python to complete this assignment, you can use split() to divide up the message into parts, *however* if the message includes quote characters (which it shouldn't, but again, there are no guarantees), split() could have problems. It's always best to check for malformed messages! When composing a message, you may find Python's join() function useful.

7. Your program will be accepting incoming UDP packets. You may be connected to the Internet behind a firewall which will block such connections or you may be behind a Network Address Translation (NAT) server which will mean your machine's IP will not be reachable to the outside world (without coordination with the NAT server). You can always check the registration server's message to see what outside IP you're coming from. If this outside IP does not match what IP you think you have, it's a good chance incoming UDP packets will not reach you.

   Both USC's Guest Wireless and Secure Wireless networks are behind NAT boxes, so you will not be able to receive messages from other peers when connected via these networks UNLESS you also use USC's VPN service while connected. For details see http://itservices.usc.edu/vpn/

   With the VPN you will have a VPN address, which will be reachable.