ARIZONA STATE UNIVERSITY

CSE 434, SLN 11169 — Computer Networks — Spring 2021

Instructor: Dr. Violet R. Syrotiuk

Socket Programming Project

Available Sunday 01/31/2021; milestone Sunday 02/14/2021; full project due Sunday 03/07/2021

The objective of this project is for you to develop an instant messaging (IM) application that uses utilizes both a client-server and a peer-to-peer (P2P) approach to implement the communication of messages to a contact list. For simplicity, in this project, your IM application is restricted to text messages.

- You may only write your code in `C/C++`, in `Java`, or in `Python`. Each of these languages has a socket programming library that you **must** use for communication among processes. Sample client and server programs that use sockets written in `C` will be provided. (The book uses `Python`.)
- This project may be completed individually or in a group of size at most two people. Each group **must** restrict its use of port numbers as described in §3.2 to prevent application programs from interfering with each other.
- You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub. Your code repository must be *private* to prevent anyone from plagiarizing your work. It is expected that you will commit changes to your repository on a regular basis, and provide evidence of use.

The rest of this project description is organized as follows. Following an overview of the IM application and its architecture in §1, the requirements of the client-server protocol and the peer-to-peer protocol are provided in §2.1 and §2.2, respectively. §3 provides some suggestions on multi-threading your application, message format, and port assignments. Finally, §4 describes the requirements for the milestone and full project deadlines.

1 IM System Architecture

The architecture of the IM system is illustrated in Figure 1. It consists of a centralized server for contact list management. Each peer is identified by a name, an IP address, and a port number, and combines the functionality of a P2P client and P2P server. As a P2P client, a peer interacts with the server to, for example, create, join, and leave contact lists, query the server for contact lists, and obtain a contact list to use in sending a text message. As a P2P server, a peer participates in serving a text message sent by a P2P client to all contacts on the contact list organized as a cyclic overlay network. The specific queries and responses to be supported by the IM application are described in §2.

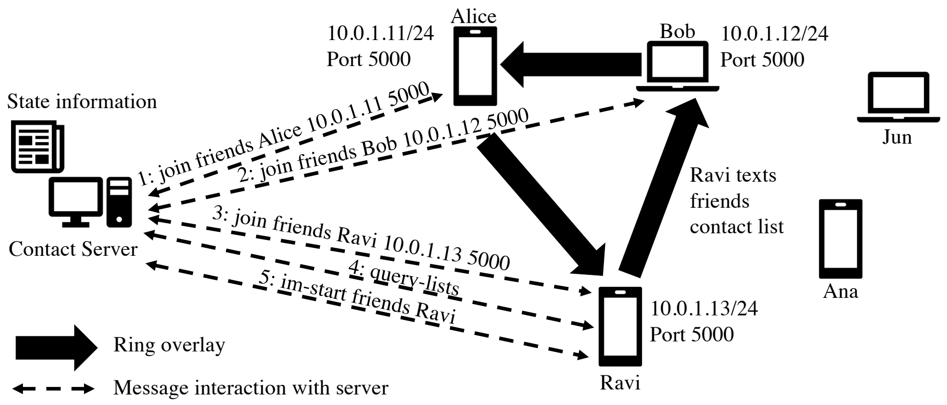

Figure 1: Architecture of the IM system.

In Figure 1, suppose that each of Bob, Alice, Ravi, Jun, and Ana have registered with the IM application. Suppose the Ravi creates a contact list `friends`, and subsequently joins it. Alice and Bob also join the contact list `friends` (messages 1, 2, and 3). Suppose that Ravi wants to send a text to those in the *friends* contact list. To do so, he first queries the contact server for the contact list *friends* (message 4). Upon receiving the server's response to starting an IM (message 5), he initiates the propagation of the text message around a cyclic overlay network of the contacts, with the text message delivered to each contact in turn (indicated by the thick arrows). An *overlay network* is constructed on top of another network. Nodes in the overlay network can be thought of as being connected by virtual or logical links, each of which corresponds to a path, perhaps through many physical links, in the underlying network.

2 Requirements of Socket Programming Project

This socket programming project involves the design and implementation of two programs:

1. The first program implements a the server-side of an "always on" contact server. Your server should read one integer command line parameter giving the port number at which the server listens for messages. A second optional command line parameter is the name of text file containing contact list information (see the `save` query for the format of this file). The commands to be supported by the server are described in §2.1.
2. The second program implements the client-side of the client-server protocol, as well as the peer-to-peer instant messaging protocol. Each client process should read two command line parameters, the first is the IPv4 address of the server in dotted decimal notation, and the second is the port number at which the server is listening. The messages to be supported by peers interacting with the server, and with other peers are described in §2.1-2.2.

2.1 The Contact Server

The contact server is to maintain a "database" of users and of contact lists. Each contact list is named and includes, for each contact on the list, the contact's name, and the IP address and port number where the contact process runs.

In the following, <> bracket parameters to the query, while the other strings are literals. There are 9 commands the contact server must support, *i.e.*, that can be issued by a P2P client. The server always responds to the client issuing the command. All communication between the contact server and P2P clients/peers must be through UDP sockets.

1. `register <contact-name> <IP-address> <port>`. This command registers the process with name `contact-name`, and adds it, along with its `IP-address` and `port`, in the database. A return code of `SUCCESS` indicates the named contact is registered successfully. A return code of `FAILURE` indicates failure to register the named contact, *e.g.*, the named contact already exists, or the number of contacts the application supports is exceeded.

2. `create <contact-list-name>`. This command initializes a contact list at the server with the name `contact-list-name` with zero contacts. A return code of `SUCCESS` indicates the contact list is created successfully at the server. A return code of `FAILURE` indicates failure to create the contact list, *e.g.*, due to the list name already in use, or the number of lists the application supports is exceeded.

3. `query-lists`. This command queries the server for the names of the contact lists. It returns a return code equal to the number of contact lists, and a list of their names. If no contact lists exist, the return code is set to zero and the list is empty.

4. `join <contact-list-name> <contact-name>`. This query allows the named contact to join the named contact list. The server looks up the `contact-name` in the list of registered contacts. If the contact is registered, its name, along with its `IP-address` and `port`, are added to the named contact list if is not already a member of the list. A return code of `SUCCESS` indicates the contact is successfully added; `FAILURE` is returned otherwise indicating, *e.g.*, a contact with the given name already exists in the list, the named contact list does not exist, the named user does not exist. In addition, `FAILURE` must be returned if the named contact is part of an ongoing instant message.

5. `leave <contact-list-name> <contact-name>`. This command removes the named contact and its associated information from the named contact list. A return code of `SUCCESS` indicates the contact is successfully removed. A return code of `FAILURE` indicates failure to remove the contact from the list, *e.g.*, removal from a non-existent contact list, or the named contact is not a member of the list. In particular, `FAILURE` must be returned if the named contact is part of an ongoing instant message.

6. `exit <contact-name>`. This command indicates to the server that the named user intends to exit the IM application. The `contact-name` is removed from list of active users, and also from each `contact-list` in which it is a member. A return code of `SUCCESS` indicates the user is successfully deleted from the list of active contacts and each contact-list of which it was a member, after which the process issuing this command must terminate (*i.e.*, exit the peer program). A return code of `FAILURE` must be returned if the named contact is part of an ongoing instant message.

7. `im-start <contact-list-name> <contact-name>`. This command indicates to the server that the named contact is initiating an instant message to those on the named contact list. It returns the number of contacts in `contact-list-name`, followed by a list of contacts including the contact's name, IP address, and port number. The first contact in the list **must** be the named contact; the order of the other contacts is immaterial. If `contact-name` is not in `contact-list-name`, the command fails, and the number of contacts returned must be zero. On receipt of a successful `im-start`, the client making the request then sends a text message to all contacts on the list (see §2.2). On completion of the text, the named contact must send a `im-complete` to the server to prevent modification to contact-lists or active users in the event of concurrent `join`, `leave` or `exit` commands.

8. `im-complete <contact-list-name> <contact-name>`. This command indicates to the server that the IM initiated by the named contact to the named list has completed. This command should pair with a corresponding `im-start` by the same named contact, in which case `SUCCESS` is returned. Otherwise `FAILURE` is returned.

9. `save <file-name>`, saves in the text file named `file-name.txt`:

   - A line containing the number n of active users (or contacts).
   - For each of the n contacts, a line containing the `contact-name`, its IP address, and port number.
   - A line containing the number ℓ of contact lists.
   - For each of the ℓ contact lists, a line containing the `contact-list-name` followed by the number k of contacts in the list.
     - For each of the k contacts, a line containing the `contact-name`, its IP address, and port number.

   A return code of `SUCCESS` is returned if the contact lists were saved successfully; `FAILURE` otherwise.

   The idea behind this query is that when the server program is started, it may be provided a file in this format to initialize the list of active users and contact lists to speed testing of your application. Of course, you then need to create processes with matching names, IP addresses, and ports!

   This file may also be generated manually. The use of such a file does not preclude the server being able to process all previous commands correctly.

2.2 Peer-to-Peer (P2P) Processes

A peer-to-peer (P2P) process can act as a client of the contact server and also as a server of IMs to other P2P processes. Such a peer provides a text-based user interface for it to interact with the contact server and its peers, *i.e.*, no fancy UI is expected. Through this interface, commands of the form listed in §2.1 are issued to the contact server; responses returned from the contact server are displayed on `stdout`.

All commands except the `im-start` are between a P2P client and the contact server, primarily for user and contact list management. The `im-start` command is handled differently, in that upon a successful response from the contact server, the named contact works together with the other P2P processes on the named contact list to propagate a text message from the named contact, to all contacts on the list following a circular overlay network. Specifically, when the number k of contacts is two or more in a response to a `im-start` command, the P2P peer then takes the following actions:

1. The user prompted to enter a text message m.
2. Recall, that the response to the `im-start` command contains a list of contacts including the contact's name, IP address, and port number on the `contact-list-name`, with the first contact in the list being the `contact-name` named on the `im-start`. That is, included in the response from the contact server is a list of the form:

name_1	IP-addr_1	port_1
name_2	IP-addr_2	port_2
$\vdots$	$\vdots$	$\vdots$
name_k	IP-addr_k	port_k

   Now, user_1 (which is also the named contact on the `im-start` command) sends the text message m to user_2 located at port_2 of IP-addr_2. On receipt of the text message, user_2 then sends m to user_3 located at port_3 of IP-addr_3, and so on until the message is sent from user_k back to its starting point, *i.e.*, , user_1 at port_1 of IP-addr_1. Each user must output details of the text message.

   For the example in Figure 1, the number of contacts in the `friends` contact list is three, *i.e.*, $k = 3$. Suppose the contact server returns the list of contacts in the following order as part of the response to an `im-start` command issued by Ravi:

Ravi	10.0.1.13	5000
Bob	10.0.1.12	5000
Alice	10.0.1.11	5000

   In this example, Ravi, Bob, and Alice are processes running on different end hosts and can therefore use the same port number. The message follows a cyclic overlay network of the contacts, delivering the text message from Ravi to Bob, from Bob to Alice (indicated in thick arrows in Figure 1), before being delivered back to Ravi from Alice. Again, each process receiving a text message should output details of it.

   On receipt of his own text message, Ravi sends the `im-complete` command to the server.

3 Implementation Considerations

You need to think about how many port numbers to assign to each peer process (the server only listens on one port). You may choose to reserve one port on a peer for communication with the server, and one or more for communication with other peer processes.

You may consider using a separate thread for handling each socket. Alternatively a single thread may loop, checking each socket one at a time to see if a message has arrived for the process to handle. If you use a single thread, you must be aware that by default the function `recvfrom()` is blocking. This means that when a process calls `recvfrom()` on an empty socket the process blocks and waits for a packet to arrive. Therefore, a call to `recvfrom()` will return immediately only if a packet has been written into the socket. This may not be what you want.

You can change the behaviour of `recvfrom()` to be non-blocking, *i.e.*, it returns immediately even if the socket is empty. This can be done by setting the `flags` argument of `recvfrom()` or by using the function `fcntl()`. See the `man` pages for `recvfrom()` and `fcntl()` for details; be sure to pay attention to the return codes.

3.1 Defining Message Exchanges and Message Format

Sections §2.1 and §2.2 describe the order of many message exchanges, as well as the actions taken on the transmission and/or receipt of message at a high level. As part of the design of your IM protocol, you must define the details.

Specifically, you need to define the format of all messages used in client-server and in peer-to-peer communication. This can be achieved by defining a structure with all the fields required by commands. For example, you could define the name of the command as an integer field and interpret other fields accordingly. Alternatively, you may prefer to define the command as a string with fields delimited by a specific character. Any choice is fine so long as you are able to extract the fields from a message and interpret them.

You should define *reasonable* upper bounds on the number of registered users that your application supports, among others. It is useful to define meaningful return codes to, *e.g.*, differentiate `SUCCESS` and `FAILURE`, and/or introduce other error conditions.

Due to their simpler multiplexing and demultiplexing, we use UDP sockets in this project. That means there is a possibility that messages may be lost or arrive out-of-order, but it is unlikely. You are not responsible for implementing reliable communication.

3.2 Port Numbers

UDP uses 16-bit integer port numbers to differentiate between processes. UDP defines a group of well known ports to identify well-known services. Clients on the other hand, use ephemeral, or short-lived, ports. These have port numbers 1024 through 655535 and are normally assigned to the client.

In this project, each group $G \geq 1$ is assigned a set of 500 unique port numbers to use in the following range. If $G \bmod 2 = 0$, *i.e.*, your group is even, then use the range: $\big[\big(\frac{G}{2} \times 1000\big) + 1000, \big(\frac{G}{2} \times 1000\big) + 1499\big]$. If $G \bmod 2 = 1$, *i.e.*, your group is odd, then use the range: $\big[\big(\lceil\frac{G}{2}\rceil \times 1000\big) + 500, \big(\lceil\frac{G}{2}\rceil \times 1000\big) + 999\big]$. That is, group 1 has range $[1500, 1999]$, group 2 has range $[2000, 2499]$, group 3 has range $[2500, 2999]$, group 4 has range $[3000, 3499]$, and so on. **Do not use port numbers outside your assigned range, as otherwise you may send messages to another group's server or peer process by accident that it is unlikely to interpret correctly, causing spurious crashes.**

4 Submission Requirements for the Milestone and Full Project Deadlines

All submissions are due before 11:59pm on the deadline date.

1. The milestone is due on Sunday, 02/14/2021. See §4.1 for the milestone submission requirements.
2. The full project is due on Sunday, 03/07/2021. See §4.2 for the full project submission requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas! An unlimited number of submissions are allowed. The last submission will be graded.

4.1 Submission Requirements for the Milestone

For the milestone deadline, you are to implement the following commands to the server: `register`, `create`, `query-lists`, and `join`, `save`, and partial functionality of `exit` (there is no IM to worry about yet).

Submit electronically before 11:59pm of Sunday, 02/14/2021 a zip file named `Groupx.zip` where `x` is your group number (do not include any spaces in your zip file name). **Do not** use any other archiving program except `zip`.

Your `zip` file must contain:

1. **Design document, commits, and video link in PDF format. 50%** Describe the design of your IM application program. Include a description your message format for each command implemented for the milestone. Include a time-space diagram for each command implemented to illustrate the order of messages exchanged between communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event. In addition, describe your choice of data structures and algorithms used, implementation considerations, and other design decisions.

Include a snapshot showing commits made in your choice of version control system.

Be sure to provide a *a link to your video demo* and that the link is accessible to graders.

2. **Code and documentation. 25%** Submit well-documented source code implementing the milestone of your IM application.

3. **Video demo. 25%** Upload a video of length at most 10 minutes to YouTube with no splicing or edits. This video must be uploaded and timestamped before the milestone submission deadline.

   The video demo your IM application must include:

   (a) Compile your server and peer programs (if applicable).
   (b) Run the freshly compiled programs on at least two (2) distinct end hosts.
   (c) First, start your server program. Then start 3 peers that each register with the server.
   (d) Create 2 contact lists, and have peers query and join lists, until each contact list has 2-3 contacts.
   (e) Save the configuration to a file.
   (f) Exit the peers; kill the server process.

   You must provide output that is a well-labelled trace the messages transmitted and received between processes so that it is clear what is happening in your IM application program.

4.2 Submission Requirements for the Full Project

For the full project deadline, you are to implement the all commands to the server listed in §2.1. This also involves implementation of texting among contacts on a contact list as described in §2.2.

Submit electronically before 11:59pm of Sunday, 03/07/2021 a zip file named `Groupx.zip` where `x` is your group number (do not include any spaces in your zip file name). **Do not** use any other archiving program except `zip`.

Your `zip` file must contain:

1. **Design document, commits, and experiment design in PDF format. 30%** Extend the design document for the milestone phase of your IM application program to include details for the remaining commands implemented for the full project. As before, for each command, include a description your message format, a time-space diagram showing actions taken on the transmission and/or receipt of a message or other event. In addition, describe your choice of data structures and algorithms used, and any implementation and design decisions that may have changed from the milestone phase of this project.

   Include a snapshot showing commits made since the milestone made in your choice of version control system.

   Design an experiment to demonstrate your IM application, describe it, and provide a *a link to a video demonstrating your application*; be sure that the link is accessible to graders.

2. **Code, demo, and documentation. 20%** Submit well-documented source code implementing the full requirements of the IM application.

3. **Video demo. 50%** Upload a video of length at most 20 minutes to YouTube with no splicing or edits. This video must be uploaded and timestamped before the full project submission deadline.

   Design an experiment to demonstrate all commands supported by your IM application and make a video of that demonstration. In particular, it must include:

   (a) Compile of your server and peer programs (if applicable).
   (b) Create processes on at least three (3) distinct end hosts.
   (c) Create both disjoint and intersecting contact lists, and show concurrent text messaging.
   (d) Illustrate handling of `join`, `leave`, and `exit` commands with concurrent text messaging.

   You must provide output a well-labelled trace the messages transmitted and received between processes so that it is clear what is happening in your IM application program.