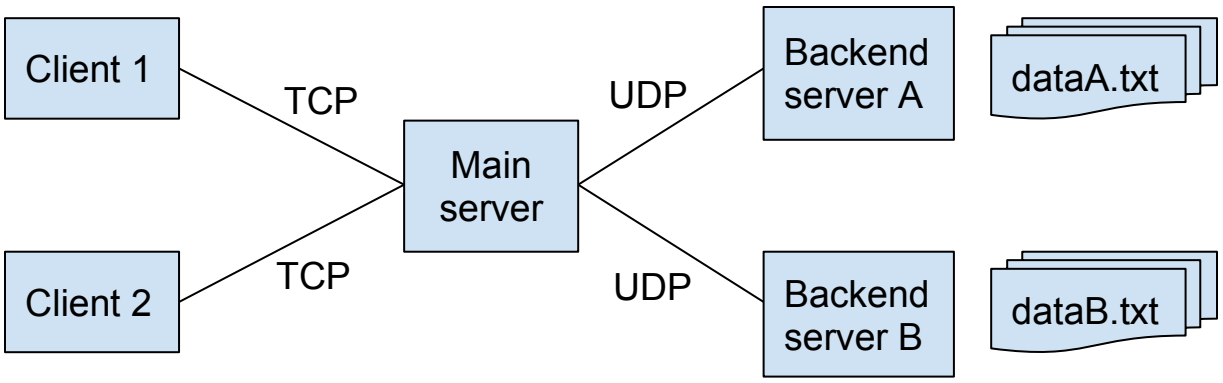
OBJECTIVE

The objective of project is to familiarize you with UNIX socket programming. This assignment is worth **4%** of your overall grade in this course. **It is an individual assignment, and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).** If you have any doubts/questions, post your questions on Blackboard -> Discussion Board, email TA your questions or come by TA's office hours. **You can ask TAs any question about the content of the project, but TAs have the right to reject your request for debugging.**

PROBLEM STATEMENT

Nowadays, social recommendation is an important task to enable online users to discover friends to follow, pages to read and items to buy. Big companies such as Facebook and Amazon heavily rely on high-quality recommendations to keep their users active. There exist numerous recommendation algorithms --- from the simple one based on common interest, to more complicated one based on deep learning models such as Graph Neural Networks. The common setup for those algorithms is that they represent the social network as a graph and perform various operations on the nodes and edges.

In this project, you will implement a simple application to generate customized recommendations based on user queries. Specifically, consider that social media users in different countries want to follow new friends. They would send their queries to a social media server (i.e., main server) and receive the new friend suggestions as the reply from the same main server. Now since a social network is so large that it is impossible to store all the user information in a single machine. Thus, we consider a distributed system design where the main server is further connected to many (in our case, two) backend servers. Each backend server stores the social network for different countries. For example, backend server A may store the user data of Canada and the US, and backend server B may store the data of Europe. Therefore, once the main server receives a user query, it decodes the query and further sends a request to the corresponding backend server. The backend server will search through its local data, identify the new friend(s) to be recommended and reply to the main server. Finally, the main server will reply to the user to conclude the process.



The detailed operations to be performed by all the parties are described with the help of Figure 1. There are in total 5 communication endpoints:

- Client 1 and Client 2: representing two different users, possibly in different countries
- Main server: responsible for interacting with the clients and the backend servers
- Backend server A and Backend server B: responsible for generating the new friend based on the query
    - For simplicity, user data is stored as plain text. Backend server A stores dataA.txt and Backend server B stores dataB.txt in their local file system.

The full process can be roughly divided into four phases (see also "DETAILED EXPLANATION" section), the communication and computation steps are as follows:

Bootup

1. [Computation]: Backend server A and B read the files dataA.txt and dataB.txt respectively and store the information in data structures.
    - Assume a "static" social network where contents in dataA.txt and dataB.txt do not change throughout the entire process.
    - Backend servers only need to read the text files once. When Backend servers are handling user queries, they will refer to the data structures, not the text files.
    - For simplicity, there is no overlap of countries between dataA.txt and dataB.txt.
2. [Communication]: after step 1, Main server will ask each of Backend servers which countries they are responsible for. Backend servers reply with a list of countries to the main server.
3. [Computation]: Main server will construct a data structure to book-keep such information from step 2. When the client queries come, the main server can send a request to the correct Backend server.

Query

1. [Communication]: Each client sends a query (a country name and a user ID) to the Main server.

    - A client can terminate itself only after it receives a reply from the server (in the Reply phase).
    - Main server may be connected to both clients at the same time.
2. [Computation]: Once the Main server receives the queries, it decodes the queries and decides which backend server(s) should handle the queries.

Recommendation

1. [Communication]: Main server sends a message to the corresponding backend server so that the Backend server can perform computation to generate recommendations.
2. [Computation]: Once the query user ID is received, Backend server recommends users who have the same interest as the query user. You need to implement an algorithm on Backend servers to find all users that have common interest with the query user.
3. [Communication]: Backend servers, after generating the recommendations, will reply to Main server.

Reply

1. [Computation]: Main server decodes the messages from Backend servers and then decides which recommendation correspond to which client query.
2. [Communication]: Main server prepares a reply message and sends it to the appropriate Client.
3. [Communication]: Clients receive the recommendation from Main server and display it. Clients should keep active for further inputted queries, until the program is manually killed.

The format of dataA.txt or dataB.txt is defined as follows. Below the name of country 1, there are multiple lines representing different interest groups. Each interest group (a line) contains the users that have the same interest, and different lines represents different kinds of interest. A user is in at least one group and may in multiple groups.

```
<name of country 1>
<ID of user 1-1> <ID of user 1-2> … < ID of user 1-k1>
<ID of user 2-1> <ID of user 2-2> … <ID of user 2-k2>
...
<name of country 2>
...
```

Let's consider an example:

Let's say there are three countries, "A", "Canada" and "xYz". In country A, there are three users with ID 0, 1 and 2. In country Canada, there are five users with IDs 78, 2, 8, 3 and 11. In country xYz, there are three users with IDs 1, 0 and 3. Although both country A and country xYz have the same user ID 0 and 1, they are not the same user (See Assumption 3 below). In country A, user 0

and 1 are in the same interest group, and user 1 and 2 are in another interest group. Assume dataA.txt stores the user data for countries A and xYz, and dataB.txt stores the user data for country Canada.

Example dataA.txt:

```
A
0 1
1 2
0
xYz
1 3
0 3
```

Example dataB.txt:

```
Canada
3 8
2 8
11
8 78 2 3
78 8
```

Assumptions on the data file:

1. A user is in at least one interest group and may be in multiple groups. There may be only one user in a group. (e.g., user 11 in country Canada).
2. There are at most 20 interest groups.
3. The pair (country name, user ID) uniquely identifies a user around the world.
    - Users in different countries may have the same ID.
    - Users in the same country do not have the same ID.
4. Country names are letters. The length of a country name can vary from 1 letter to at most 20 letters. It may contain only capital and lowercase letters but does not contain any white spaces or other characters. Country names “Abc” and “abc” are different.
5. There are at most 10 countries in total of dataA.txt and dataB.txt.
6. User IDs are non-negative integer numbers and are separated by only one white space in a line. The maximum possible user ID is (2^31 - 1). The minimum possible user ID is 0.
    - This ensures that you can always use int32 to store the user ID.
    - Backend servers may also want to re-index the user IDs.
7. Within the same country, user IDs do not need to be consecutive and may not start from 0. i.e., if a country contains N users, their IDs do not need to be 0, 1, 2, …, N-1. See the case of Canada and xYz.

8. There is no additional empty line(s) at the beginning or the end of the file. That is, the whole dataA.txt and dataB.txt do not contain any empty lines.
9. For simplicity, there is no overlap of countries between dataA.txt and dataB.txt.
10. There is no repeated user ID in an interest group.
11. The user IDs in the text are not sorted.
12. dataA.txt and dataB.txt will not be empty.
13. A country will have at least one user, and at most 100 users.

An example dataA.txt and dataB.txt is provided for you as a reference. Other dataA.txt and dataB.txt will be used for grading, so you are advised to prepare your own files for testing purposes.

Source Code Files

Your implementation should include the source code files described below, for each component of the system.

1. Main Server: You must name your code file: **servermain.c** or **servermain.cc** or **servermain.cpp** (all small letters). Also, you must name the corresponding header file (if you have one; it is not mandatory) **servermain.h** (all small letters).

2. Backend-Server A and B: You must use one of these names for this piece of code: **server#.c** or **server#.cc** or **server#.cpp** (all small letters except for #). Also, you must name the corresponding header file (if you have one; it is not mandatory) **server#.h** (all small letters, except for #). The "#" character must be replaced by the server identifier (i.e. A or B), e.g., serverA.c.

3. Client: The name for this piece of code must be **client.c** or **client.cc** or **client.cpp** (all small letters) and the header file (if you have one; it is not mandatory) must be called **client.h** (all small letters). **There should be only one client file!!!**

Note: Your compilation should generate separate executable files for each of the components listed above.

DETAILED EXPLANATION

Phase 1 (10 points) -- Bootup

All three server programs (Main server, Backend servers A & B) boot up in this phase. While booting up, the servers must display a boot up message on the terminal. The format of the boot up message for each server is given in the onscreen message tables at the end of the document. As the boot up message indicates, each server must listen on the appropriate port for incoming packets/connections.

Backend servers should boot up first. A backend server needs to read the text file and store the country names and user IDs in appropriate data structures. There are many ways to store them, such as dictionary, array, vector, etc. You need to decide which format to use based on the requirement of the problem. You can use **any** format if you can generate the correct recommendation.

Main server then boots up after Backend servers are running and finish processing the files of dataA.txt and dataB.txt. Main server will request Backend servers for country lists so that Main server knows which Backend server is responsible for which countries. The communication between Main server and Backend servers is using UDP. For example, Main server may use an unordered_map to store the following information as in part 2.

```
std::unordered_map<std::string, int> country_backend_mapping;
country_backend_mapping[“xYz”] = 0;
country_backend_mapping[“Canada”] = 1;
country_backend_mapping[“A”] = 0;
```

Above lines indicate that “xYz” and “A” stored in dataA.txt in Backend server A (represented by value 0) and “Canada” is stored in dataB.txt in Backend server B (represented by value 1). Again, you could use **any** data structure or format to store the information.

Once the server programs have booted up, two client programs run. Each client displays a boot up message as indicated in the onscreen messages table. Note that the client code takes no input argument from the command line. The format for running the client code is:

```
./client
```

After running it, it should display messages to ask the user to enter a query country name and a query user ID (e.g., implement using std::cin):

```
./client
…
Enter country name:
Enter user ID:
```

For example, if the client 1 is booted up and asks for the friend recommendation for user ID 78 in Country Canada, then the terminal displays like this:

```
./client
…
Enter country name: Canada
Enter user ID: 78
```

After booting up, Clients establish TCP connections with Main server. After successfully establishing the connection, Clients send the input country name and user ID to Main server. Once this is sent, Clients should print a message in a specific format. Repeat the same steps for Client 2.

Each of these servers and the main server have its unique port number specified in "PORT NUMBER ALLOCATION" section with the source and destination IP address as localhost/127.0.0.1.

Clients, Main server, Backend server A and Backend server B are required to print out on screen messages after executing each action as described in the "ON SCREEN MESSAGES" section. These messages will help with grading if the process did not execute successfully. Missing some of the on-screen messages might result in misinterpretation that your process failed to complete. Please follow the exact format when printing the on-screen messages.

Phase 2 (40 points) -- Query

In the previous phase, Client 1 and Client 2 receive the query parameters from the two users and send them to Main server over TCP socket connection. In phase 2, Main server will have to receive

requests from two Clients. If the country name or user ID are not found, the main server will print out a message (see the "On Screen Messages" section) and return to standby.

For a server to receive requests from several clients at the same time, the function **fork()** should be used for the creation of a new process. Fork() function is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (***parent process***). This is the same as in Project Part 1.

For a TCP server, when an application is listening for stream-oriented connections from other hosts, it is notified of such events and must initialize the connection using accept(). After the connection with the client is successfully established, the accept() function returns a non-zero descriptor for a socket called the child socket. The server can then fork off a process using fork() function to handle connection on the new socket and go back to waiting on the original socket. Note that the socket that was originally created, that is the parent socket, is going to be used only to listen to the client requests, and it is not going to be used for communication between client and Main server. Child sockets that are created for a parent socket have the identical well-known port number IP address at the server side, but each child socket is created for a specific client. Through using the child socket with the help of fork(), the server can handle the two clients without closing any one of the connections.

Phase 3 (40 points) -- Recommendation

In this phase, each Backend server should have received a request from Main server. The request should contain a country name and a user ID. A backend server will generate one recommendation per request based on the common interest. If a user is in the same interest group as the query user, it should be taken as potential friends and be recommended to the query user. If the query user is in different interest groups, all users in these groups except the query user should be recommended. The recommendation result could be either None or ID(s) of other users.

Recall the example in the "PROBLEM STATEMENT" section. When the user queries 1 in A, user 0 and 2 should be recommended. When the user queries 11 in Canada, none of user can be recommended. When the user queries 0 in xYz, user 3 should be recommended. There should be no repeated user IDs in recommendation results. For example, when the user queries 78 in Canada, the recommendation result should be user 2, 3, 8, rather than 2, 3, 8, 8.

Phase 4 (10 points) -- Reply

At the end of Phase 3, the responsible Backend server should have the recommendation result ready. The result is the recommended user IDs. The result should be sent back to the Main server using UDP. When the Main server receives the result, it needs to forward all the result to the

corresponding Client using TCP. The clients will print out the recommended user IDs and then print out the messages for a new request as follows:

```
...

User 2, User 3, user 8 is/are possible friend(s) of User 78 in
Canada

-----Start a new request-----
Enter country name:
Enter user ID:
```

See the ON SCREEN MESSAGES table for an example output table.

PORT NUMBER ALLOCATION

The ports to be used by the client and the servers are specified in the following table:

Table 1. Static and Dynamic assignments for TCP and UDP ports

Process	**Dynamic Ports**	**Static Ports**
Backend-Server A		UDP: 30xxx
Backend-Server B		UDP: 31xxx
Main Server		UDP(with server): 32xxx TCP(with client): 33xxx
Client 1	TCP	
Client 2	TCP	

NOTE: xxx is the last 3 digits of your USC ID. For example, if the last 3 digits of your USC ID are "319", you should use the port: **30319** for the Backend-Server (A), etc. Port number of all processes print port number of their own.

ON SCREEN MESSAGES

Table 2. Backend-Server A on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	Server A is up and running using UDP on port <server A port number>
Sending the country list that contains in "dataA.txt" to Main Server:	Server A has sent a country list to Main Server
For friends searching, upon receiving the input query:	Server A has received a request for finding possible friends of User<user ID> in <Country Name>
If this user ID cannot be found in this country, send "not found" back to Main Server:	User<user ID> does not show up in <Country Name>
	Server A has sent "User<user ID> not found" to Main Server
If this user ID is found in this country, searching possible friends for this user and send result back to Main Server:	Server A found the following possible friends for User<user ID> in <Country Name>: <user ID1>, <user ID2>...
	Server A has sent the result to Main Server

Table 3. Backend-Server B on-screen messages

Event	**On-screen Messages**
Booting up (Only while starting):	Server B is up and running using UDP on port <server B port number>
Sending the country list that contains in "dataB.txt" to Main Server:	Server B has sent a country list to Main Server
For possible friends finding, upon receiving the input query:	Server B has received request for finding possible friends of User<user ID> in <Country Name>
If this user ID cannot be found in this country, send "not found" back to Main Server:	User<user ID> does not show up in <Country Name>
	The server B has sent "User<user ID> not found" to Main Server
If this user ID is found in this country, searching possible friends for this user and send result(s) back to Main Server:	Server B found the following possible friends for User<user ID> in <Country Name>: <user ID1>, <user ID2>...
	The server B has sent the result(s) to Main Server

Table 4. Main Server on-screen messages

Event	On-screen Messages
Booting up(only while starting):	Main server is up and running.
Upon receiving the country lists from Server A:	Main server has received the country list from server A using UDP over port <Main server UDP port number>
Upon receiving the country lists from Server B:	Main server has received the country list from server B using UDP over port <Main server UDP port number>
List the results of which country server A/B is responsible for:	Server A <Country Name 1> <Country Name 2> Server B <Country Name 3>
Upon receiving the input from the client:	Main server has received the request on User <user ID> in <Country Name> from client <client ID> using TCP over port <Main server TCP port number>
If the input country name could not be found, send the error message to the client:	<Country Name> does not show up in server A&B
	Main Server has sent "<Country Name>: Not found" to client <client ID> using TCP over port <Main server TCP port number>
If the input country name could be found, decide which server contains related information about the input country and send a request to server A/B	<Country Name> shows up in server <A or B>
	Main Server has sent request of User <user ID> to server A/B using UDP over port <Main server UDP port number>
If this user ID is found in a backend server, Main Server will receive the searching results from server A/B and send them to client1/2	Main server has received searching result of User <user ID> from server<A or B>
	Main Server has sent searching result(s) to client <client ID> using TCP over port <Main Server TCP port number>
If this user ID cannot be found in a backend server, send the error message back to client	Main server has received "User <user ID>: Not found" from server <A or B>
	Main Server has sent message to client <client ID> using TCP over <Main Server TCP port number>

Table 5. Client 1 or Client 2 on-screen messages

Event	**On-screen Messages**
Booting up(only while starting)	Client is up and running
	Enter country name: Enter user ID:
After sending User ID to Main Server:	Client has sent <Country Name> and User<user ID> to Main Server using TCP over port <dynamic TCP port>
If input country not found	<Country Name>: Not found
If received message from main server saying that input User ID not found	User <user ID>: Not found
If input User ID and country can be found and the result is received:	User<user ID1>, User<user ID2>, … is/are possible friend(s) of User<user ID> in <Country Name>
After the last query ends:	-----Start a new request----- Enter country name: Enter user ID: