EE450 Socket Programming Project,
Summer 2022 Due Date : Friday June 24th, 2022
11:59 PM (Midnight)
(The deadline is the same for all on-campus and DEN off-campus students)
Hard Deadline (Strictly enforced)

The objective of this assignment is to familiarize you with UNIX socket programming. **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 D2L. **You must discuss all project related issues on the DEN discussion forum**. We will give those who actively help others out by answering questions on the DEN discussion forum up to 10 bonus points.

Problem Statement:

Digital transactions and cryptocurrencies have been a hot topic lately. People prefer to use digital transactions due to their simplicity and they choose cryptos because they provide a sense of safety and anonymity that regular currencies cannot. While the latter are not bounded by a government or a country, their success relies on the distributed nature of their platforms. The technology behind them is known as blockchain, a chain of information blocks where we store all the transactions that have taken place since the start of the system. The amount of transactions stored on a block depends on the type of blockchain and the information can be provided to every member of the system (public blockchain) or to a few members of it (permissioned blockchain) . In this project we shall implement a simplified version of a blockchain system that'll help us understand how cryptocurrency transactions work. For this scenario, we will have three nodes on the blockchain where each stores a group of transactions. These will be represented by backend servers. While in the blockchain the transaction protocol deals with updating the digital wallet of each user, for this project we will have a main server in charge of running the calculations and updating the wallets for each user. Each transaction reported in the blockchain will include, in the following order, the transaction number, sender, receiver and amount being transferred.

In this project, you will implement a simplified version of a blockchain service called **Txchain**, where a client issues a request for finding their current amount of **txcoins** in their account, transfers them to another client and provides a file statement with all the transactions in order. These requests will be sent to a Central Server which in turn interacts with three other backend servers for pulling information and data processing.

The Main server (or Server-M) will connect to servers A, B and C, which have a group of transactions that are not in order. When required, the main server has to pull the transaction information from each backend server to find out the current wallet balance of the user, to transfer money from one user to another and to write back a file statement with all the transactions and store it on the Main server. The steps and explanation of each operation that will be required for the blockchain platform is provided in each phase of the project.

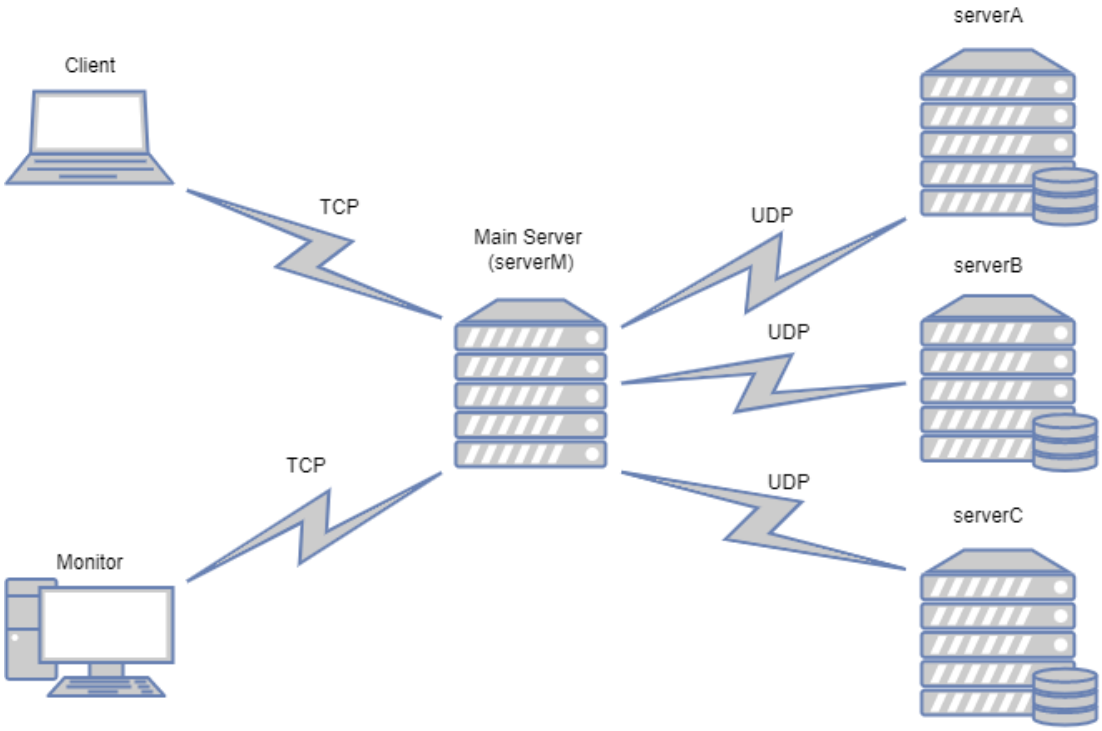

Figure 1. Illustration of the network

Server A has access to a file named block1.txt, server B has access to a file named block2.txt and server C has access to a file named block3.txt. The client, monitor and the main server communicate over a TCP connection while the communication between main server and the Back-Servers A, B & C is over UDP connections. This setup is illustrated in Figure 1.

Note:

Server-A will **not** be having access to block2.txt & block3.txt
Server-B will **not** be having access to block1.txt & block3.txt
Server-C will **not** be having access to block1.txt & block2.txt
The Main-Server or Server-M will **not** be having access to any block file.

Source Code Files

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

1. Server-M: You must name your code file: **serverM.c** or **serverM.cc** or **serverM.cpp** (all small letters except 'M'). Also you must include the corresponding header file (if you have one; it is not mandatory) serverM.h (all small letters except 'M').
2. Back-Server A, B and C: 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 include 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 or C), depending on the server it corresponds to. (Eg: serverA.cpp, serverB.cpp & serverC.cpp)

   Note: You are not allowed to use one executable for all four servers (i.e. a "fork" based implementation).
3. Client: The name of 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).
4. Monitor: The code file for the monitor must be called **monitor.c** or monitor.cc or monitor.cpp (all small letters) and the header file (if you have one; it is not mandatory) must be called **monitor.h** (all small letters).

Application Workflow Phase Description:

Phase 1: (30 points)

Establish the connection between the client and Main Server and the connection between the monitor and Main Server.

All four server programs (Main Server, Backend ServerA, Backend ServerB and Backend ServerC) boot up in this phase. While booting up, the servers must display a

boot message on the terminal. The format of the boot message for each server is given in **Table 4-7**. As the boot message indicates, each server must listen on the appropriate port information for incoming packets/connections.

Once the server programs have booted up, the client and monitor program should run. Each terminal displays a boot message as indicated in the onscreen messages table. Note that the client code takes an input argument from the command line that specifies the username(s).

Note: Once started, the main server and three backend servers should always be running (unless stopped manually). The client and monitor can terminate their connections with the main server once they receive the response for the operation they had requested.

For the first two phases, the client can perform two kinds of request operations, namely “CHECK WALLET”, “TXCOINS” (transfer coins). Which operation is performed depends on the number of input arguments that the code takes from the command line:

Operation1: CHECK WALLET

In this operation, the input argument should be the client’s own name and it will be used to fetch the client’s balance in the wallet, which will be described in the next section. The format for running client code in this operation is:

./client <username>

The username is the input for getting a specific user’s wallet balance. As an example, if the client wants to check Martin's wallet balance, the command should be run as follows:

./client Martin

Operation2: TXCOINS

In this operation, There will be three input arguments, namely the client name of the transfer out, the client name of the transfer in, and the amount of transfers. The main server will complete and record this transaction according to the input arguments, more details will be described in the next section. The command for running the client code in this operation is:

./client <username1> <username2> <transfer amount>

The usernames from the client are the inputs for transferring coins from one user to another. As an example, to transfer 100 txcoins from Martin to Chinmay should be run as follows

./client Martin Chinmay 100

After booting up, the client and monitor establish TCP connections with the server. After successfully establishing the connections, the client will send one of the requests described above to the main server. Once these are sent, the client should print a message in the format given in **Table 8 & 9.** This ends Phase 1 and now we can proceed to Phase 2.

Note: Before we proceed to phase-2 and phase-3, it should be noted that whenever the Main Server sends any transaction related information to the other backend servers (serverA, serverB and serverC), it should be encrypted based on the encryption scheme mentioned below:

- Offset each character and/or digit by 3.
- The scheme is case sensitive.
- Special characters (including spaces and/or the decimal point will not be encrypted or changed) symbol will not change
- All the backend servers should record any new transaction in an encrypted fashion based on the above scheme in their respective block files. The serial number/transaction number must NOT be encrypted.

Few examples of encryption are given below:

Example	**Original Text**	**Cipher Text**
#1	Martin	Pduwlq
#2	Chinmay	Fklqpdb
#3	Xiake	Aldnh
#4	Welcome to EE450!	Zhofrph wr HH783!
#5	100	433
#6	199	422
#7	0.27	3.40
#8	75	08

While sending the results back to the client and/or the monitor, the main server should decrypt all the encrypted messages and then send it back to the client/monitor. In other words, the client and/or the monitor must not receive any encrypted messages from the main server.

Phase 2: (40 points)

Phase 2A: Main Server connects to backend servers and proceeds to retrieve the information.

In Phase 1, you read what should be sent from Client to server M (main server) over the TCP connections. For Phase 2, Server M will send messages to the three back-servers (Server A, Server B and Server C) through UDP connections. The request will be sent to their respective back-end server depending on which information they need to get and which operation they need to execute.

Table 1: An example of the format for each Txchain transaction

Serial No.	Sender	Receiver	Transfer Amount
1	Racheal	John	45
2	Rishil	Alice	30
3	Oliver	Rachit	94
4	Ben	Victor	85
5	Chinmay	Oliver	129
6	Racheal	Alice	49
7	Martin	Luke	25
8	Rishil	Chinmay	10
9	Ali	Luke	155

初始都有1000　Name is not in the network, error check

Every member of the platform received **1000 txcoins** as incentive to join this blockchain.
Note: On Table 1 we can see the format of the transactions. Each row in the block files stored at the backend servers will be encrypted. So, for the set of transactions mentioned in table 1, the actual information stored in the block file will be as follows:

Table 1.1 : An example of transactions stored at blockfiles at the backend servers A,B&C			
Serial No.	**Encrypted Sender**	**Encrypted Receiver**	**Encrypted Transfer Amount**
1	Udfkhdo	Mrkq	78
2	Ulvklo	Dolfh	63
3	Rolyhu	Udfklw	27
4	Ehq	Ylfwru	18
5	Fklqpdb	Rolyhu	452
6	Udfkhdo	Dolfh	72
7	Pduwlq	Oxnh	58
8	Ulvklo	Fklqpdb	43
9	Dol	Oxnh	488

One thing to note in table 1.1 is that the serial numbers are not encrypted. All the three backend servers A,B&C will be having their own different log files with different encrypted transactions.

Operation1: CHECK WALLET

For this operation the main server will use the input *username* provided from client on phase 1 and check if the username is found on the transaction records while communicating with the backend servers. If the username is not found, the main server will provide a message as indicated below on the On-screen messages portion of the document. If the username exists we will calculate the amount the *username* has on Txchain and convey it back to the client.

Operation2: TXCOINS

For this operation the main server will use the input *username1*, *username2* and *amount* provided on phase 1 and check if both usernames can be found on the records

while communicating with the backend servers. If any of the usernames is not found the main server will provide a message as indicated below on the On-screen messages portion of the document. If both usernames exist, we will add the transaction to any of the block files that are located in the backend servers.

Phase 2B: Main Server computes the operation and replies to the client.

Once the connection server (or the main server) receives the relevant data for the desired operation from the other three servers (namely serverA, serverB and serverC), it will perform the required computation and send the results to the clients. The computation performed by the main server depends on the service requested by the client (i.e, CHECK WALLET or TXCOINS)

Operation1: CHECK WALLET

In this operation, the main server would be receiving only those transaction logs from each of the three backend servers (serverA, serverB and serverC) in which the user was involved. There can be multiple transaction logs related to a particular user distributed randomly across the three log files. Each backend server would be handling its own log file.

Based on the log data (mentioned above) received, the main server will compute the current balance of the client based on the following formula :

$$Current\ Balance\ =\ Initial\ Balance\ +\ \sum Coins\ Received\ -\ \sum Coins\ Sent$$

As already mentioned in phase 2A, initially all the members would be having 1000 txcoins. So, 'Initial Balance' term for everyone would be 1000.
After computing the current balance for a client, the main server will send this info to the client as response.

Operation2: TXCOINS

In this operation, the main server would notify the client about the transaction status. The transaction can either be successful or unsuccessful. In either case, the main server should notify the client about the status.

The transaction can be unsuccessful mainly because of two reasons. The first one being insufficient current balance of the sender. Transactions can also fail if either sender or recipient (or both) is not part of the network.

If the transaction is feasible (both the sender and the receiver are part of the network and the sender's balance is equal to or more than what he/she is intending to send), the main server would perform the following computations:

- The main server would figure out what should be the serial number of the current transaction. Every transaction would have a serial number which should be assigned in a contiguous fashion as they occur. In the log files, these transactions would not be arranged in some specific order of their serial number as the transaction log data is randomly distributed across three log databases handled by the three backend servers (serverA, serverB and serverC) separately. So, if the last transaction had a serial number 'n', then the current transaction should have the serial number of 'n+1'.
- The main server would then generate the encrypted log entry (which would be similar to the the other logs):

  <Unencrypted_Serial_No> <Encrypted_Sender_Username> <Encrypted_Receiver_Username> <Encrypted_Transfer_Amount>

- The main server then randomly selects any of the three backend servers (serverA, serverB and serverC) and sends this log entry to the selected server. The server which receives this log information records this log information as a new entry in its log database and sends a confirmation message to the main server.
- Upon receiving the confirmation, the main server enquiries for the updated balance of the sender and sends the transaction status along with the updated balance (current balance after this transaction) to the sender client.

Note: In the initial block files that we have provided you to begin with, there will not be any sort of ordering.

Phase 3: (30 points)

In phases 1 and 2 we have described two operations that required us to do calculations to check each user's account balance and also to add transactions to the blockchain. For phase 3, we will add one more operation named TXLIST which will be requested by the **monitor** (client will not be requesting for this operation)

Operation3: TXLIST

For this operation the monitor will send a keyword to indicate that the client is asking to get the full text version of all the transactions that have been taking place in Txchain and save it on a file named ”txchain.txt” located on the main server. The format of the operation will be as follows:

./monitor TXLIST

When the monitor runs this operation, the main server will receive this request and connect to the backend servers to gather all the information from the transactions. Main server will sort the list of transactions in the ascending (non-decreasing) order of the serial number (or transaction number) and generate the “txchain.txt” file with all the transactions made up to that point (including any new transactions made from the moment we booted up the servers). **All the transactions written in the txchain.txt file should be unencrypted (or decrypted)**. Students will have the freedom to choose any algorithm they prefer for this sorting operation.

Phase 4: (50 points extra, not mandatory)

In order to prevent the data leaks or stealing of any sensitive information, you need to improve existing encryption/cryptographic technique (presented in this project) which should make use of a **‘graph’** (similar to what you had in Lab 1) in such a way that it becomes nearly impossible to sabotage the txchain infrastructure. The graph will be given to you as input (graph.txt and score.txt) and should be constructed in a similar way you did in Lab 1 (using the compatibility score formula). Your design and implementation of the cryptographic scheme should make use of the elements of the graph and must be based on one or more following techniques:

- Cryptography algorithms such as RSA, etc.
- Statistical techniques (using mutual information, cross correlation, etc.)
- Randomization
- Learning techniques (graph learning, machine learning, deep Learning, etc.)
- Hashing

Note: If you decide to implement the extra credit part, you need to first come up with a proposal. You then need approval of your proposal by the instructor (Prof. Shahin) or the TA (Martin), or the mentors (Chinmay & Xiake).

Submission files for extra credit (along with the mandatory files): one client script and one server script. The server script would be having the access to all the block files (encrypted based on the scheme you would design). You would have to clearly explain and articulate each and every specific of your scheme in the Readme file.

Process Flow/ Sequence of Operations:

- Your project grader will start the servers in this sequence: serverM, serverA, serverB, serverC in four different terminals.
- As already mentioned, once all the servers are started, the servers should be continuously running unless stopped manually by the grader.
- The client/monitor can be exited after receiving and printing the results from the main server and will be re-executed every time while making a request to the main server.
- The sequence to request service from the main server is also fixed: Two operations by the client followed by one operation by the monitor.

Example:
./client Chinmay
./client Xiake
./monitor TXLIST
./client Chinmay Martin 20
./client Martin
./monitor TXLIST

.
.
.
.
.
<Other tests in the same sequence i.e., two client requests followed by one monitor request>
.
.
.
.
.
.

./client Oliver Victor 15
./client Victor Xiake 170
./monitor TXLIST

Few Important things to note:

You shouldn't expect more than a couple of thousand lines of transactions combined for the 3 block files together. **All block text files will end with a newline \n character. Make sure every time you append a new transaction to one of the block files to end it as well with \n.** If the total number of transactions among the three txt files is 15 then there will be transaction numbers from 1-15. We will not have a transaction number zero and there will not be any number skip for transactions.

Required Port Number Allocation

The ports to be used by the clients and the servers for the exercise are specified in the following table:

Table 3. Static and Dynamic assignments for TCP and UDP ports.

Process	Dynamic Ports	Static Ports
Backend-Server (A)	-	1 UDP, 21000+xxx
Backend-Server (B)	-	1 UDP, 22000+xxx
Backend-Server (C)	-	1 UDP, 23000+xxx
Main Server (M)	-	1 UDP, 24000+xxx 1 TCP with client, 25000+xxx 1 TCP with monitor, 26000+xxx
Client	1 TCP	<Dynamic Port assignment>
Monitor	1 TCP	<Dynamic Port assignment>

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: **21000+319 = 21319** for the Backend-Server (A). **It is NOT going to be 21000319.**

ON SCREEN MESSAGES: Table 4. Backend-Server A on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	“The ServerA is up and running using UDP on port <port number>.”
Upon Receiving the request from main server	“The ServerA received a request from the Main Server.”
After sending the results to the main server:	“The ServerA finished sending the response to the Main Server.”

ON SCREEN MESSAGES: Table 5. Backend-Server B on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	“The ServerB is up and running using UDP on port <port number>.”
Upon Receiving the request from Main Server	“The ServerB received a request from the Main Server.”
After sending the results to the main server:	“The ServerB finished sending the response to the Main Server.”

ON SCREEN MESSAGES: Table 6. Backend-Server C on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (Only while starting):	“The ServerC is up and running using UDP on port <port number>.”
Upon Receiving the request from main server	“The ServerC received a request from the Main Server.”
After sending the results to the main server:	“The ServerC finished sending the response to the Main Server.”

ON SCREEN MESSAGES: Table 7. Main Server on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up (only while starting):	“The main server is up and running.”
Upon Receiving the username from the clients for checking balance:	“The main server received input=<USERNAME> from the client using TCP over port <port number>.”
Upon Receiving the username from the clients for transferring coins:	“The main server received from <SENDER_USERNAME> to transfer <TRANSFER_AMOUNT> coins to <RECEIVER_USERNAME> using TCP over port <port number>.”
After querying Backend-Server i for checking balance (i is one of A,B, or C):	“The main server sent a request to server <i>.”
After receiving result from backend server i for checking balance (i is one of A,B, or C):	“The main server received transactions from Server <i> using UDP over port <PORT_NUMBER>.”
After querying Backend-Server i for transferring coins (i is one of A,B, or C):	“The main server sent a request to server <i>.”
After receiving result from backend server i for transferring coins (i is one of A,B, or C):	“The main server received the feedback from server <i> using UDP over port <PORT_NUMBER>.”
After sending the current balance to the client.(CHECK WALLET)	“The main server sent the current balance to the client.”
After sending the result of the transaction to the client. (TXCOINS)	“The main server sent the result of the transaction to the client.”
Upon receiving a request from the monitor for a sorted list (TXLIST)	“The main server received a sorted list request from the monitor using TCP over port <PORT_NUMBER>.”

ON SCREEN MESSAGES: Table 8. Client on screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	“The client is up and running.”
Upon sending the input to main server for checking balance	"<USERNAME> sent a balance enquiry request to the main server.”
Upon sending the input(s) to the main server for making a transaction.	“<SENDER_USERNAME> has requested to transfer <TRANSFER_AMOUNT> txcoins to <RECEIVER_USERNAME>.”
After receiving the balance information from the main server	“The current balance of <USERNAME> is : <BALANCE_AMOUNT> txcoins.”
After receiving the transaction information from the main server (if successful)	“<SENDER_USERNAME> successfully transferred <TRANSFER_AMOUT> txcoins to <RECEIVER_USERNAME>. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> txcoins.”
After receiving the transaction information from the main server (if transaction fails due to insufficient balance)	“<SENDER_USERNAME> was unable to transfer <TRANSFER_AMOUT> txcoins to <RECEIVER_USERNAME> because of insufficient balance. The current balance of <SENDER_USERNAME> is : <BALANCE_AMOUNT> txcoins.”
After receiving the transaction information from the main server (if one of the clients is not part of the network)	“Unable to proceed with the transaction as <SENDER_USERNAME/RECEIVER_USERNAME> is not part of the network.”
After receiving the transaction information from the main server (if both the clients are not part of the network)	“Unable to proceed with the transaction as <SENDER_USERNAME> and <RECEIVER_USERNAME> are not part of the network.”

ON SCREEN MESSAGES: Table 9. Monitor on-screen messages	
Event	**On Screen Message (inside quotes)**
Booting Up:	"The monitor is up and running."
Upon sending the input to main server for requesting a sorted list	"Monitor sent a sorted list request to the main server."
Main server confirms that the txchain.txt is generated	"Successfully received a sorted list of transactions from the main server."

Submission files and folder structure:
(Additionally, refer #2 of submission rules for more details)

Your submission should have the following folder structure and the files (the examples are of .cpp, but it can be .c files as well):

- **ee450_gabel_chinmay_gabelc.tar.gz**
  - **ee450_gabel_chinmay_gabelc**
    - **client.cpp**
    - **monitor.cpp**
    - **serverM.cpp**
    - **serverA.cpp**
    - **serverB.cpp**
    - **serverC.cpp**
    - **Makefile**
    - **Readme.txt**
    - **<Any additional header files>**

The grader will extract the tar.gz file, and will place all the block files in the same directory as your source files. The txchain.txt should also be generated in the same directory as your source files. So, after testing your code, the folder structure should look something like this:

- ee450_gabel_chinmay_gabelc

- client.cpp
- monitor.cpp
- serverM.cpp
- serverA.cpp
- serverB.cpp
- serverC.cpp
- Makefile
- Readme.txt
- client
- monitor
- serverM
- serverA
- serverB
- serverC
- block1.txt
- block2.txt
- block3.txt
- txchain.txt
- <Any additional header files>

Note that in the above example, the test block files (block1.txt, block2.txt and block3.txt) will be manually placed by the grader, while the 'make all' command should generate the executable files, and your code should generate the txchain.txt file (during TXLIST operation).

Example Output to Illustrate Output Formatting: For a sample CHECK WALLET operation:

Backend-Server A Terminal:

The ServerA is up and running using UDP on port 21319.

The ServerA received a request from the Main Server.

The ServerA finished sending the response to the Main Server.

Backend-Server B Terminal:

The ServerB is up and running using UDP on port 22319.

The ServerB received a request from the Main Server.

The ServerB finished sending the response to the Main Server.

Backend-Server C Terminal:

The ServerC is up and running using UDP on port 23319.

The ServerC received a request from the Main Server.

The ServerC finished sending the response to the Main Server.

Main Server Terminal:

The main server is up and running.

The main server received input="Racheal" from the client using TCP over port 25319.

The main server sent a request to server A.

The main server received transactions from Server A using UDP over port 21319.

The main server sent a request to server B.

The main server received the feedback from server B using UDP over port 21319.

The main server sent a request to server C.

The main server received the feedback from server C using UDP over port 21319.

The main server sent the current balance to the client.

Client Terminal:

The client is up and running.

"Racheal" sent a balance enquiry request to the main server.

The current balance of "Racheal" is : 906 txcoins.

Assumptions:

1. You have to start the processes in this order: **ServerM, ServerA, ServerB, ServerC. Request sequence: 2 Client operations followed by 1 Monitor operation.**

2. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.

3. You are allowed to use blocks of code from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to cite the copied part in your code.

4. When you run your code, if you get the message "port already in use" or "address already in use", **please first check to see if you have a zombie process** (see following). If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, please do mention it in your README file and provide reasons for it.

5. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command: `>>ps –aux | grep ee450.` Identify the zombie processes and their process number and kill them by typing at the command-line: `>>kill -9 processNumber.`

Requirements:

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table 3 to see which ports are statically defined and which ones are dynamically assigned.
2. The host name must be hard coded as **localhost (127.0.0.1)** in all codes.
3. Your client/monitor should terminate themselves after all is done. And the client can run multiple times to send requests. However, the backend

servers and the main server should keep being running and be waiting for another request until the TAs terminate them by Ctrl+C. If they terminate before that, you will lose some points for it.

4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.
5. You are not allowed to pass any parameter or value or string or character as a command-line argument except what is already described in the project document.
6. All the on-screen messages must conform exactly to the project description. You should not add anymore on-screen messages. If you need to do so for the debugging purposes, you must comment out all of the extra messages before you submit your project.
7. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming platform and environment:

1. All your submitted code **MUST** work well on the provided virtual machine Ubuntu.
2. All submissions will only be graded on the provided Ubuntu. TAs/Graders won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code works well on the provided Ubuntu. **"It works well on my machine" is not an excuse**.
3. Your submission MUST have a Makefile. Please follow the requirements in the following ""Submission Rules" section

Programming languages and compilers:

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):

http://www.beej.us/guide/bgnet/

(If you are new to socket programming please do study this tutorial carefully as soon as possible and before starting the project)

http://www.beej.us/guide/bgc/

You can use a unix text editor like emacs to type your code and then use compilers such as g++ (for C++) and gcc (for C) that are already installed on Ubuntu to compile your code. You must use the following commands and switches to compile yourfile.c or yourfile.cpp. It will make an executable by the name of "yourfileoutput".

gcc -o yourfileoutput yourfile.c g++
-o yourfileoutput yourfile.cpp

Do NOT forget the mandatory naming conventions mentioned before!

Also inside your code you need to include these header files in addition to any other header file you think you may need:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h
```

Submission Rules:

1. Along with your code files, include a **README file and a Makefile**. In the README file write
   a. Your **Full Name** as given in the class list
   b. Your Student ID
   c. What you have done in the assignment, if you have completed the optional part (suffix). If it's not mentioned, it will not be considered.
   d. What your code files are and what each one of them does. (Please do not repeat the project description, just name your code files and briefly mention what they do).
   e. The format of all the messages exchanged.
   g. Any idiosyncrasy of your project. It should say under what conditions the project fails, if any.
   h. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, say what functions and where they're from. (Also identify this with a comment in the source code.)

Submissions WITHOUT README AND Makefile WILL NOT BE GRADED.

Makefile tutorial:

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

About the Makefile: makefile should support following functions:

`make all`	Compiles **all** your files and creates executables
`./serverM`	**Run**s central server
`./serverA`	**Run**s server A
`./serverB`	**Run**s server B
`./serverC`	**Run**s server C
`./client ...`	Starts the client (with proper arguments)
`./monitor ...`	Starts the monitor (with proper arguments)

TAs will first compile all codes using **`make all`**. They will then open 6 different terminal windows. On 4 terminals they will start servers M, A, B and C . On the other two terminals, they will start client/monitor processes using **`./client <args> or ./monitor <args>`**. **Remember that servers should always be on once started.** The Client can connect again and again with different input values. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in tables in this project writeup.

2. Compress all your files including the README file into a single "tar ball" and call it: **ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz** (all small letters) e.g. my filename would be **ee450_gabel_chinmay_gabelc.tar.gz**. Please make sure that your name matches the one in the class list. Here are the instructions:

   a. On your VM, go to the directory which has all your project files. Remove all executable and other unnecessary files. **Only include the required source code files, Makefile and the README file**. Now run the following commands:

   b.
   >> tar cvf **ee450_YourLastName_YourFirstName_yourUSCusername.tar** *
   >> gzip **ee450_YourLastName_YourFirstName_yourUSCusername.tar**
   Now, you will find a file named "ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz" in the same directory. Please notice there is a star(*) at the end of the first command.
   Any compressed format other than .tar.gz will NOT be graded!

3. Upload "ee450_YourLastName_YourFirstName_yourUSCusername.tar.gz" to the Digital Dropbox on the DEN website (DEN -> EE450 -> My Tools -> Assignments -> Project). After the file is uploaded to the dropbox, you must click on the "**send"** button to actually submit it. If you do not click on "**send**", the file will not be submitted.

4. D2L will keep a history of all your submissions. If you make multiple submissions, we will grade your latest valid submission. Submission after the deadline is considered as invalid.

5. D2L will send you a "Dropbox submission receipt" to confirm your submission. So please do check your emails to make sure your submission is successfully

received. If you don't receive a confirmation email, try again later and contact your TA if it always fails.

6. Please take into account all kinds of possible technical issues and do expect a huge traffic on the DEN website very close to the deadline which may render your submission or even access to DEN unsuccessful.

7. Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen and you will miss the deadline. And a kind suggestion, if you still get some bugs one hour before the deadline, please make a submission first to make sure you will get some points for your hard work!

8. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. If the outcome is not what you expected, try to resubmit and confirm again. We will only grade what you submitted even though it's corrupted.

9. **There is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a zero for the project.**

Grading Criteria:

Notice: We will only grade what is already done by the program instead of what will be done.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.

2. Inline comments in your code. This is important as this will help in understanding what you have done.

3. Whether your programs work as you say they would in the README file.

4. Whether your programs print out the appropriate error messages and results.

5. If your submitted codes do not even compile, you will receive 5 out of 100 for the project.

6. If your submitted codes compile using make but when executed, produce runtime errors without performing any tasks of the project, you will receive 10 out of 100.

7. If you forget to include the README file or Makefile in the project tar-ball that you submitted, you will lose 15 points for each missing file (plus you need to send the file to the TA in order for your project to be graded.)

8. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose 10 points each.

9. You will lose 5 points for each error or a task that is not done correctly.

10. The minimum grade for an on-time submitted project is 10 out of 100, assuming there are no compilation errors and the submission includes a working Makefile and a README.

11. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend about 2 weeks on this project and it doesn't even compile, you will receive only 5 out of 100.

12. **You must discuss all project related issues on D2L Discussion Forum**. We will give those who actively help others out by answering questions on D2L up to 10 bonus points. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on D2L.)

13. The maximum points that you can receive for the project with the bonus points is 100. In other words the bonus points will only improve your grade if your grade is less than 100.

14. Your code will not be altered in any way for grading purposes and however it will be tested with different inputs. Your TA/Grader runs your project as is, according to the project description and your README file and then checks whether it works correctly or not. If your README is not consistent with the description, we will follow the description.

Cautionary Words:

1. Start on this project early!!!

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is *the **provided Ubuntu (16.04)***. It is strongly recommended that students develop their code on this virtual machine. In case students wish to develop their programs on their personal machines, possibly running other operating systems, they are expected to deal with technical and incompatibility issues (on their own) to ensure that the final project compiles and runs on the requested virtual machine. If you do development on your own machine, please leave at least three days to make it work on Ubuntu. It might take much longer than you expect because of some incompatibility issues.

3. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command: **`>>ps –aux | grep ee450`**
   Identify the zombie processes and their process number and kill them by typing at the command-line: **`>>kill -9 processnumber`**

Academic Integrity:

All students are expected to write all their code on their own.

Copying code from friends is called **plagiarism** not **collaboration** and will result in an F for the entire course. Any libraries or pieces of code that you use and you did not write must be listed in your README file. All programs will be compared with automated tools to detect similarities; examples of code copying will get an F for the course. **IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED ABOUT PLAGIARISM, TALK TO THE TA.** "I didn't know" is not an excuse.