Fall 2021 CSci 4061:
Introduction to Operating Systems
Project #4 : Network Sockets

December 3, 2021

1 Background

Low–level network programming is accomplished using *network sockets*. Socket programming in a broader sense refer to a method of inter–process communication while network sockets allow for this inter–process communication to occur across separate hosts (i.e. computers) connected via a network.

This project is will not cover any aspects of underlying protocols or any in-depth aspects of the OSI and TCP/IP Models. We will only focus on network sockets in the context of TCP/IP.

2 Project Overview

In this project, you will be implementing a networked banking system. You will create a multi-threaded server, which will act as the coordinating authority of the entire system by managing account balances. The clients will function as local branches of the bank and will interact with this central authority by sending queries.

You will have two separate applications on separate hosts which are interacting with one another, or the same host using the loopback interface (see sec. 2.1.1). The client will send requests to the server, which will process and respond to them via the network; all communication will be governed by a protocol which you will implement. The three key components of this project are as follows:

1. Protocol will define several queries and dictate the responses and actions each one should produce from the client or the server.

2. Server will wait for incoming requests from the client and will process and respond to them accordingly. It will maintain the balances and other records for all other accounts, and also distribute cash to the clients. The clients will function as local branches of the bank.

3. Client will read from a provided input file and submit the requests to the server. It can submit requests to do things such as create accounts, perform transactions, and request information about the transaction history. It also maintains a store of cash, which may be refilled with cash requests from the central server.

Try not to overthink this project, the most difficult part is likely to be the emphasis on serializing/de–serializing messages. However, the functionality of socket programming is nearly identical to piping, which you should already be very familiar with. A basic diagram of the client–server system can be seen in Fig. 1

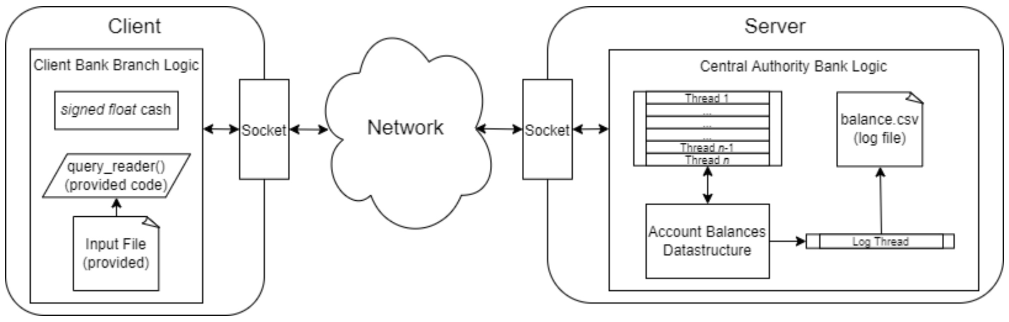

Figure 1: Overview of the client–server relationship

2.1 Assumptions

Once again, this course does not expect you to know a significant amount about networking and there are several assumptions you will be making. If you are interested in learning more, please consider taking CSci 4211.

2.1.1 IP Addresses

- Use the loopback interface (use IPv4 address 127.0.0.1) to run both the client and server on the same machine.

2.1.2 Port Numbers

- The server port number you use may need to be relatively unique (several students may be using the same machine) so free to calculate one it as 9001 plus the last three numbers of your student ID number.

2.1.3 Endianness/Byte Order

- Be consistent with your byte orderings, convert to network byte order (big–endian) when writing multi–byte data types to a socket. Convert them back to host byte order when reading from the socket.

2.1.4 General Bank Assumptions

- There will never exist more that 1023 accounts at any time
- Account numbers should start at 1 and increment with each new account
- Your server's main thread will never return, it will always continue listening for new connections
- Your code is to be written such that multiple clients may execute on the same machine at the same time. This is to simulate multiple bank branches.

- Names and usernames will never be longer than 64 characters, including the NULL terminator.

3 Implementation Details

3.1 Server

The client is launched with the following command–line arguments:

./server server_addr server_port num_workers

The server will begin by launching a log thread. The purpose of this thread is to do a timed wait (5 seconds) ~~(30 seconds)~~ (see Sec. 6.6 #2) and writing a log to a file. When logging, it should iterate over each account in a global balance datastructure in a thread–safe manner (NOTE: threads should be able to modify the balance immediately before and after an account is logged) to perform logging of all account balances to the output file *balances.csv* with each line being *account_number*,*balance*,*name*,*username*,*birthday* in the format (NOTE: the commas and lack of spaces):

"%d,%.2f,%s,%s,%ld\n"

The server will then create a socket and begin listening on it. For each incoming connection, the server will create a worker thread which will handle the connection (pass it the connection's file descriptor) and return to listening on the socket. A worker thread will parse each query received and reply with the appropriate response. If it modifies the global balance datastructure, it should signal the log thread's condition variable. This will continue until it receives a TERMINATE query from the client. It will then close the connection and return. ~~The maximum number of worker threads should be passed as a command–line argument.~~ (see Sec. 6.6 #1)

You are expected to implement all bank functionality and ensure that all operations are thread–safe. You may implement the global balance variables as whatever datastructure you so choose (HINT: use an array of structs with account data fields and individual locks).

3.2 Client

The client is launched with the following command–line arguments:

./client input_filename server_addr

The client should record the time and then attempt to connect to the server. It will then read a file containing various queries to send to the server. These queries are contained in a provided input file, one per line of the format *message_type*, *account_number*, *name*, *username*, *birthday*, *amount*, *num_transactions* for which you can use the following specifier (NOTE: the commas and lack of spaces):

""%d,%d,%s,%s,%ld,%f,%d\n"

The client will send the first query, and further queries dictated by the protocol, before finally ending the connection after submitting a TERMINATE message to the server. If there are new messages after the TERMINATE, the client should re–connect to the server and continue submitting queries. This will continue execution until the input file parser reaches and EOF, it will then print the total elapsed time in seconds to standard output in the format:

"Elapsed Time: %.2f\n"

Since the client also maintains a store of cash, it has a starting balance that is provided as a constant. Obviously, the cash on hand should never be negative. To prevent this, the client should not submit a transaction to the server immediately, it should first send a request for cash and then proceed with the transaction (NOTE: transmit the constant CASH_AMOUNT). Similarly, accounts should not be allowed to hold a negative balance.

3.3 Protocol

You will be implementing a protocol for your banking software to use. There are several potential queries that the client may send to which the server must respond in a pre–defined way. We provide you with several enumerated constants which you will find helpful for determining the type and size of messages that the client and server will each receive and casting them to the correct type.

At first glance, it may seem natural to create and transmit structs for these messages. However, thing such as pointers are no longer useful when transmit across a socket (they would point to addresses on a different host). Instead, you will be serializing all data to form your protocol. This means that you will be writing each message byte–by–byte to the socket and also reading them in this order and using your knowledge of the types involved and the general layout of the messages and their fields to correctly re–assemble them.

It is highly recommended that the enumerated type specifying the message is written to the socket before any other fields for a message. This will allow the recipient to read those four bytes (since an enum is by default the size of an int), and determine which message type must be handled, all other data may then be read in a predictable manner. An example of client–server interaction is show in Fig. 2.

3.3.1 Message Details

- *REGISTER*: create a new account
  - Expected Response: *BALANCE* ~~*ACCOUNT_INFO*~~ (see Sec. 6.2 #1)
  - Data Values:
    1. *char[]* username (*NULL*–terminated)
    2. *char[]* name (*NULL*–terminated)
    3. *time_t* birthday
- *GET_ACCOUNT_INFO*: request the information for a specific account
  - Expected Response: *ACCOUNT_INFO*
  - Data Values:
    1. *int* account_number
- *TRANSACT*: send a transaction for an account (make sure that the account balance and cash reserves are high enough first)
  - Expected Response: *BALANCE*
  - Data Values:
    1. *int* account_number
    2. *float* amount
- *GET_BALANCE*: get the balance of a specific account

  - Expected Response: *BALANCE*
  - Data Values:
    1. *int* account_number

- *ACCOUNT_INFO*: return the account information for the requested account
  - Expected Response: *NONE*
  - Data Values:
    1. *char[]* username (*NULL*–terminated)
    2. *char[]* name (*NULL*–terminated)
    3. *time_t* birthday

- *BALANCE*: return the balance for the requested account
  - Expected Response: *NONE*
  - Data Values:
    1. *int* account_number
    2. *float* balance

- *REQUEST_CASH*: request that the recipient is sent cash
  - Expected Response: *CASH*
  - Data Values:
    1. *float* amount

- *CASH*: send cash to the recipient (add to the recipient's cash reserves)
  - Expected Response: *NONE*
  - Data Values:
    1. *float* cash

- *ERROR*: generic error message to be sent when the enumerated message type does not match any within the protocol
  - Expected Response: *NONE*
  - Data Values:
    1. *int* message_type (the enumerated value)

- *TERMINATE*: signal to the recipient that no further messages are to be expected (NOTE: this does *not* shut down the server)
  - Expected Response: *NONE*
  - Data Values:
    1. *NONE*

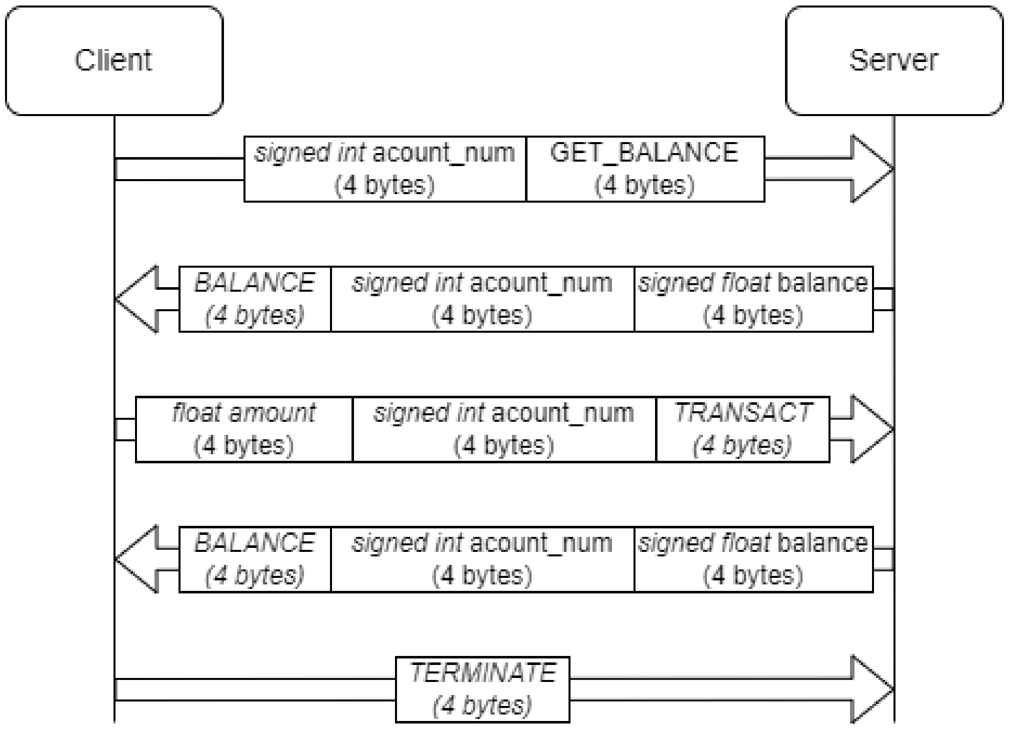
Client
Server
signed int acount_num (4 bytes)
GET_BALANCE (4 bytes)
BALANCE (4 bytes)
signed int acount_num (4 bytes)
signed float balance (4 bytes)
float amount (4 bytes)
signed int acount_num (4 bytes)
TRANSACT (4 bytes)
BALANCE (4 bytes)
signed int acount_num (4 bytes)
signed float balance (4 bytes)
TERMINATE (4 bytes)

Figure 2: An example of query–response exchange between the server and client

4 Submissions

Both your interim and final submissions must be in a .zip folder which unzips into a single new folder that is simply your group number (see Sec. 6.4 #2) in the current working directory. Deviation from the following directory structure will result in a reduced grade:

```
/
├── Makefile
├── README.md
├── include
│   ├── server.h
│   ├── client.h
│   └── utils.h
├── lib
└── src
    ├── server.c
    ├── client.c
    ├── utils.c
    └── launcher.c
```

Submissions should also include a README.md containing the following:

- Group number, member names, and x500 addresses
- Whether extra credit was attempted
- Members' individual contributions (or plan for the interim)

- Any additional assumptions which you made
- Instructions on compiling and running your program (via Makefile)

4.1 Interim

For the interim submission, implement sockets for the client and server so than a connection can be established. Have the client iterate through each enumerated value and send each in the order they are declared in **utils.h** to the server. The server will receive each of these, print their value to standard output, and re–transmit it to the client. The server will continue to do so until it receives the *TERMINATE* enum, in which case it will re–transmit it to the client, close the connection, and return. The client will print each response it receives to standard out as well and will close the connection and return after receiving and printing the echoed *TERMINATE*.

5 Grading Policy

1. (10%) Interim Submission
   (a) (5%) Report content
   (b) (5%) Echo server implementation
2. (5%) Appropriate Coding Style and Comments
3. (5%) Conformance Check
   (a) (3%) README.md content
   (b) (2%) Output file structure
4. (50%) Test Cases
   (a) (20%) Single client, single server thread (basic functionality)
   (b) (15%) Multiple clients, single server thread (race conditions)
   (c) (15%) Multiple clients, multiple server threads (performance via concurrency)
5. (30%) Code
   (a) (5%) Thread/socket creation, binding, socket/file I/O, and error handling for each
   (b) (5%) Implementation of client ~~and server~~ (see Sec. 6.6 #4) cash balance
   (c) (5%) Proper thread locking to avoid deadline and thread starvation
   (d) (5%) Serializing/de–serializing the queries/responses
   (e) (5%) Correct responses to each query
   (f) (5%) Logic for handling complex queries (ex: transactions)
6. (10%) Extra Credit
   (a) (5%) Thread Pool
   Rather than having your server create a worker thread for each new connection, have your server create a fixed number of worker threads which will persist across multiple connections. That is to say, the server will create the log thread, socket, and worker pool

of threads. The main thread will then wait on all other threads. The worker threads will attempt to get a lock on the socket, take a connection file descriptor, and then unlock the socket. A worker will then handle all queries on this connection until it receives the TERMINATE flag. Then, instead of returning, the worker will return to attempting to lock and get a new connection from the socket.

(b) (5%) Transaction History

Implement two additional messages to allow for a query to be sent that will request a specified number of transactions for an account. You will need to either modify the global balance array to also point to a datastructure which may grow to arbitrary size or some other data structure to store the history of transactions for each account. When the client receives the response with the transaction information, it is to print the account number and transactions (each on a separate line) to standard output.

You must also update the global balance logging to also include a transaction file for each account. Each file will be *account_#.csv* (where # is the account number) and will have a transaction amount per line. Each account will produce an output file with one transaction amount per line, i.e. with the formatter:

"%.2f\n"

The two new protocol messages are outlined is as follows:

- *REQUEST_HISTORY*: request the log of some number of the most recent transactions (requesting 0 transactions means provide them all)
  - Expected Response: *HISTORY*
  - Data Values:
    1. *int* account_number
    2. *int* number_transactions
- *HISTORY*: return the requested number of previous transactions for the requested account
  - Expected Response: *NONE*
  - Data Values:
    1. *int* account_number
    2. *int* number_transactions
    3. *float[]* transactions

6 FAQ/Clarifications

This section contains various answers to common questions, clarifications, and should take preference over the rest of this writeup.

6.1 Interim Submission

1. Submit a **README.mb** (no .PDF)
2. Submit a .PNG of your client and servers' output
3. Have your client/server output the enums with the following format:

   $$"\%s : \%d"$$

   This will help us catch any possible mismatch in their values and prevent misgrading.

6.2 Protocol

1. The response to REGISTER should instead be BALANCE.
2. **Always** convert from network order to host order when reading, and host order to network order when writing.

6.3 I/O Files

1. Your applications should assume all input files are in the first–level directory *input/* (similar to *include/*, *src/*, etc. Have your application append the directory to form the full path to the input file.
2. Not all the fields in a line of the input file will be relevant, use the first field (the message type) to determine which of them you will need for the message the line represents (HINT: look at the protocol).

6.4 Directory Structure

1. You may add any source or header files as you wish (place headers in *include/* and source files in *src/*).
2. For example, group #15 will submit **15.zip**, when extracted it will produce the directory *15/* which contains the file structure of Sec. 4 .

6.5 Client

1. You do not need to specify a port for the client, it will be assigned one automatically by the kernel.
2. The cash balance of the client is a local variable that all transactions are added to/subtracted from (think of an ATM withdrawl or a bank deposit). So the account must have enough money, and the client must have enough cash on hand to fulfill the a withdrawl. The cash balance of the client will increase with deposits and REQUEST_CASH queries, and decrease with withdrawls.
3. A TRANSACT message should be ignored if it will make the balance of an account negative.
4. A TRANSACT message may trigger one or more REQUEST_CASH messages if the client does not have enough cash on hand.
5. You cannot re–connect on a socket, you will need to re–declare it each time.

6.6 Server

1. The server does not need to have a bound on the number of threads it will create (except for the extra credit). However, your **server.c** should still take this command line argument, ignore it if you do not complete the extra credit.
2. Lower the log time to 5 seconds, as the input files we are using are quite small.
3. When the server REGISTER's a new account, its starting balance should be 0

4. The server will not maintain a balance of cash, it will always send a CASH reply with the same value that was in the REQUEST_CASH it recieved

6.7 Launcher

1. The launcher may not give your server adequate time to begin listening for connections. You may need to increase the duration of the *sleep()* on line #22 of **launcher.c**.
2. To reflect the decreased logging interval, feel free to reduce the sleep on line