Myles Greene-Beaupre 03/25/2022
C12195336
ECE 470

Digital Wallet System

NOTE: I did not get the "cancel pending transaction" operation signed off on during the demo, but I have since implemented it and there is a screenshot of it working in the "Verification" section of this report. The functionality for it is also in the source code for "common.c" at the end of the report.

Project Description/Goals

The goal of this project was to design and implement a user-friendly digital wallet system using the C programming language, unix network sockets, and the TCP protocol. While there were few architectural requirements, the end product must include several features such as a menu to be displayed to the user for selecting which operation they would like to perform, as well as feedback functionality where the user is informed if the server has encountered an error because of invalid input from the client. This, as well as intuitive, simple menu options and response messages, allows the system to be utilized by anyone with little or no instruction on how it works. After registering an account with the system and logging in, the user must be able to perform the following operations: check their current account balance, request a payment from another user, request a refund for a completed payment they made in the past, print the lines of the transaction ledger relevant to them, pay the currently pending payment request (displayed to them upon login), refuse the currently pending payment request, or cancel a pending payment request

they have submitted to the ledger. The digital wallet system must support multiple non-concurrent users, and allow each of them to perform these operations while receiving proper, sensical feedback from the server.

High-Level Architecture

The system functions based on a request/response loop in which the client receives input from the end user and constructs a message to send to the server, sends the message, waits for a response to give the user feedback, and then prompts the user for more input to determine the next operation they would like to perform. On the server side, a similar loop happens where the server waits for a message from the client, performs the operation if possible, and sends a message back to the client (this response message could be some data the user requested like their transaction history, or it could be a confirmation that the operation was successful or an error message informing the user that the operation could not be completed). The server then waits for another message from the client and continues in this loop until the user sends a logout message over the socket, in which case both the client and server loops are broken, the client program ends and closes the socket, and the server listens and waits to accept a new connection (possibly from another client). Both the client and server loops only start once the client has logged in, so if a loop is happening that means a client is currently logged in and authenticated (allowed to do operations). This means that the system does not support concurrent users, as this would require either multithreading so that multiple connections and operation loops can occur simultaneously, or the UDP protocol so that the authentication would be verified by the server upon receiving each message (since the message could be from any client, and they might not have been authenticated yet). The system architecture does not include objects or structs, and it is purely procedural

since it is essentially just a series of linear instructions which are organized into functions based on what they do on the client side, and they usually have an equivalent server side function that the server calls based on the contents of the message it received from the client.

Scenarios

1. REGISTER: This scenario is between one client and the server. The client is asked if they have an account, and if they do not then they are asked if they would like to create one. If they choose yes, they are prompted for a username and password, and the client constructs a register message and sends it to the server. When the server receives this message and determines that it is a register account request, it checks the users file to make sure that no user already exists with that username. If the name is already taken, the response sent back to the client will be an error message telling them so. If the name is not taken, the user is added to the users file with the username and password they entered, and a balance of zero dollars, and the response sent back to the client is a message that indicates that the account creation was successful. The client then displays the operations menu to the user, and they are considered logged in/authenticated at this point.

2. LOGIN: This scenario is between one client and the server. On the client side, the user is asked if they have an account, and if they say yes then they are prompted for a username and password. A login message is then constructed and sent to the server, and when the server receives it then it verifies that the username exists in the users file (if not an error response is sent back to the client) and that the password they entered is correct (an incorrect password results in an error response indicating so). If the

username and password are correct, a successful login response is sent to the client and the client then displays the operations menu. Additionally, if the server has found a pending request for money from this user in the ledgers file, the response it sends to the client will also contain this request so that the user knows, upon login, that someone has requested money from them.

3. GET BALANCE: This scenario is between one client and the server. The client selects the "Get current balance" in the operations menu, and a get balance message is constructed and sent to the server. When the server receives this message, it searches for the user's name in the users file (returning an error response to the client if it could not be found), and grabs the line two down from the username line, which contains the user's current balance. This line is appended to the response message and sent to the client, which displays the balance to the user.

4. REQUEST PAYMENT: This scenario is between two clients and the server. The client selects "Request a payment" in the operations menu, and they are prompted for the username they would like to request a payment from and for what amount. A request payment message is constructed with this information and sent to the server, which verifies that the user exists in the users file and sends an error response message back to the client if they do not. The server then makes an entry in the ledger indicating that a payment request is pending between the user and the other user they indicated for the specified amount. Since each time a user logs in the server also checks if they have a pending payment request in the ledger, the next time the user specified logs in this payment request will appear above their operations menu.

5. REQUEST REFUND: This scenario is between two clients and the server. The client selects “Request a refund” in the operations menu and they are prompted for some information about the past transaction that they would like to be refunded, such as who they sent money to and how much it was. A refund request message is then constructed and sent to the server, and the server verifies that a completed transaction with this information exists in the ledger, and if it does not then an error response message is sent back to the client. If this past transaction does exist in the ledger, the server constructs a new pending entry for the ledger of the exact same information, except the recipient and the person who will receive the request are swapped (ex. “C: Alice requested $42 from Bob” will become “P: Bob requested $42 from Alice”). This entry is placed into the ledger so that the user from which a refund is being requested is informed of this the next time they login.

6. PRINT TRANSACTIONS: This scenario is between one client and the server. The client selects “Print transactions” from the operations menu and a print transactions message is constructed and sent to the server. When the server receives this message, it goes through each line of the ledger and if the user’s name is in that line then it is appended to the response to be sent to the client. When the server is done searching the ledger, all of the lines found in the ledger that are relevant to the user requesting their transaction history are sent back to the client in a message to be printed to the user.

7. PAY CURRENT REQUEST: This scenario is between two clients and the server. The user selects “Pay current pending request” and the client checks a variable called pending, whose value is assigned based on the return value

of the login function, and if this variable is zero then the user is informed that they do not have any pending payment requests. If they do have a payment request, it should have been printed upon login, and a pay current request message is constructed and sent to the server. When the server receives this message, it verifies that the user has enough money in their account to pay the request (sending an error response message if not) and then deducts the amount from their balance in the users file and adds the amount to the recipient's balance in the users file. The relevant payment request is then updated in the ledger file from 'P:' to 'C:', indicating that it is now completed and no longer pending, and it should no longer pop up when the user logs in.

8. REFUSE CURRENT REQUEST: This scenario is between two clients and the server. Similar to "Pay current pending request," an error message is printed if they do not have a pending request, and if they do have a pending request a refuse current request message is constructed and sent to the server. When the server receives this message, it goes in the ledger and finds this request and changes the 'P:' to 'X:', indicating that the request has been refused by the requestee. A success message is sent back to the client.

9. CANCEL PENDING REQUEST: This scenario is between two clients and the server. The client selects "Cancel a pending payment request" and the user is prompted for some information about the request they made that they would like to cancel, such as who they requested the money from and for how much. A cancel pending request message is constructed and sent to the server, and when the server receives it it searches the ledger for a pending (not completed) request with this information in it, sending an error message

response back to the client if it could not be found. If it was found, the entry is deleted from the ledger so that it will no longer appear when the person who money was requested next logs in, and the user who requested this cancellation is sent a success message response.

10. LOGOUT: This scenario is between one client and the server. The client selects the "Logout" option from the operations menu and a logout message (which is simply an "X") is sent to the server, and the communication socket on the client side is closed and the program terminates. When the server receives this logout message, it breaks the while loop and waits for a new client connection before entering the loop again.

Detailed Design

```
#define PORT 2001
#define MESSAGE_SIZE 500

//both
int recvLoop(int csoc, char* data, const int size);

//client
void cget_balance(int commsoc, char* user);
int is_yes(char* input);
int create_socket();
char welcome();
int login(char header, int commsoc, char* user, char* pass);
void crequest_payment(int commsoc, char* user);

//server
int start_server();
FILE* open_file(char* path, char* mode);
int user_exists(char* user);
char* add_user(char* user, char* pass);
char* login_user(char* user, char* pass);
char* request_payment(char* msg);
char* get_balance(char* user);
double get_balance_f(char* user);
char* has_payment_request(char* user);
void set_bal(char* user, char* newbal);
void update_ledger(char* user, char character);
char* get_ledger(char* user);
```

Organization: This is the common.h file which contains all of the function declarations used by the client, server, and the one function which is used by both which is the recvLoop() function. It also includes all the necessary libraries, and definitions for the size of messages passed between client and server and for the port that they will use. This common header is included by client.c and server.c so that they may use their respective functions, and the common.c file has all the definitions (actual instructions/steps/return statements) for these functions. This allows for a very simple and readable project folder, as the entire project consists of client.c, server.c, common.c, common.h, and a users and ledger file. The users file acts as a database for storing the username, password and balance of each registered user, and the ledger file acts as a log of all transactions completed, pending and canceled.

```
msg203@rabbit:~ % ls
client          common.c        ledger          server.c
client.c        common.h        server          users
```

Messages: All messages passed between server and client are a fixed size of 500 bytes as defined in common.h. The messages that the client passes to the server are identified by type based on the first byte, so that the server knows what the message is requesting it to do. The table below is all the headers and their meanings.

Header Character (first byte)	**Meaning (interpreted by server)**
J	register new user
L	login existing user

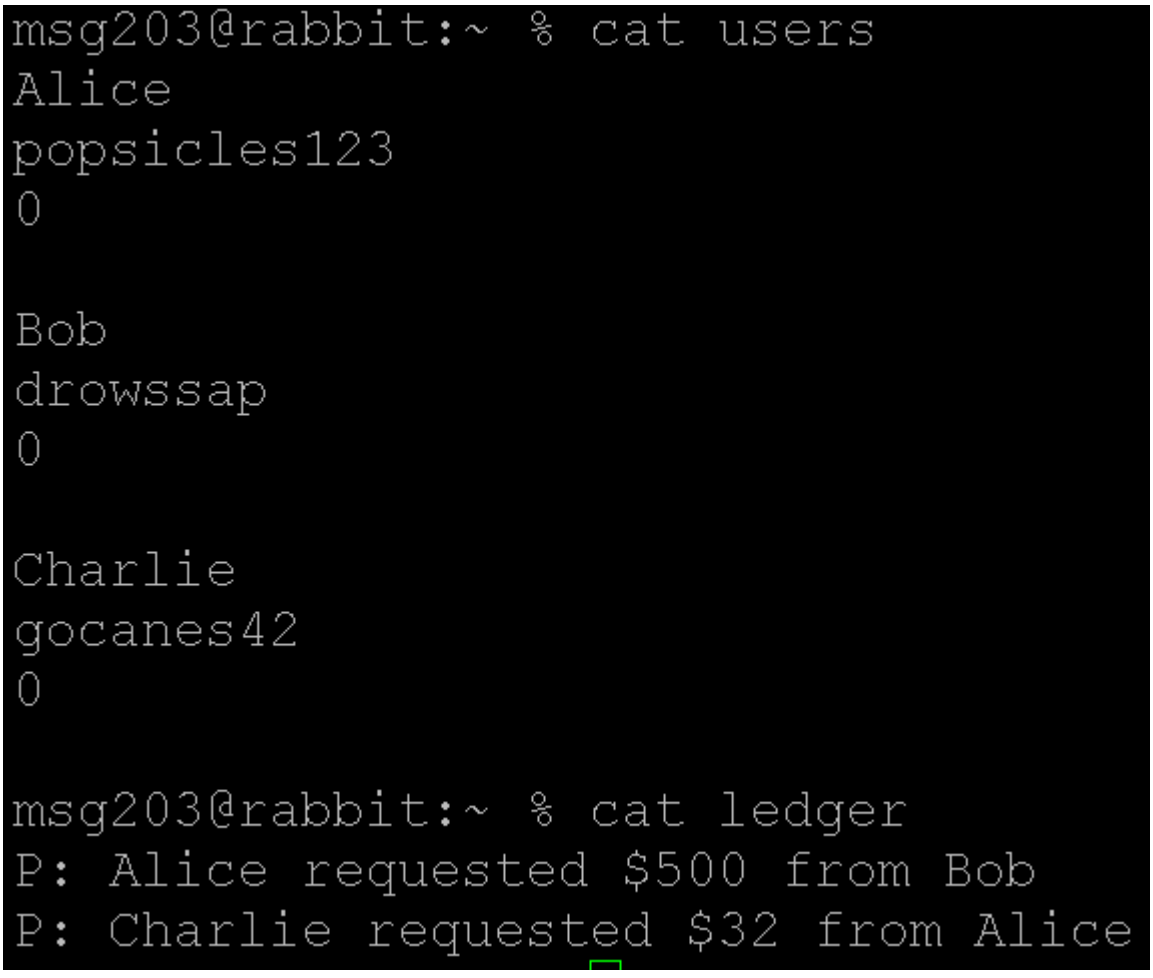
B	get balance
R	request payment
P	pay current pending payment request
N	refuse current pending payment request
T	print relevant transactions in ledger
G	request a refund
C	cancel a payment request you made
X	logout

When the server receives a message, it switches based on the first character and then removes the first character using memmove() so that the rest of the information in the message can be used to perform the operation it needs to do. The rest of this message is not explicitly outlined in the message protocol, so it can be any information it needs to be for the operation, such as a ledger entry (“P: Alice has requested $42 from Bob”) or a username and password (“myles 123”). In the case of a login request, for example, the full message the server receives before it manipulates it would be “Lmyles 123”.

```
    while(1) {
        response = "";
        recvLoop(commsoc, buf, MESSAGE_SIZE);

        switch(buf[0]) {
            case 'C':
                memmove(buf, buf+1, strlen(buf));
                response = cancel_payment_request(buf);
                break;
            case 'G':
                memmove(buf, buf+1, strlen(buf));
                response = request_refund(buf);
                break;
            case 'T':
                memmove(buf, buf+1, strlen(buf));
                response = get_ledger(buf);
                break;
            case 'N':
                memmove(buf, buf+1, strlen(buf));
                update_ledger(buf, 'X');
                response = "You have denied a pending transaction. The ledger has been updated.\n";
                break;
            case 'J':
                memmove(buf, buf+1, strlen(buf));
                user = strtok(buf, " ");
                pass = strtok(NULL, " ");
                response = add_user(user, pass);
```

Data: As for the data that the server deals with, the users file makes a four-line entry for each user, with the first line being their username, the second being their password, the third being their account balance, and the fourth being an empty line that separates them from the next user for easy readability. The ledger file has one transaction per line, and it always follows the format of STATUS: (user) requested (amount) from (other user). The STATUS can be P for pending, C for completed, or X for refused, the amount can be any double number, and the users must be valid members that exist in the users file.



State/Program Flow: The state of the program is described in the "High-level Architecture" section, but basically it consists of a loop on the client side, which does the following indefinitely until logout:

1. Take input from user
2. Send appropriate message to server
3. Wait for response from server
4. Print response
5. Go to step 1

The server has a similar loop:

1. Get message from client
2. Interpret message and do necessary operations using data in the message
3. Send response to client based on result of operation

On the client side, if the user logs out the program ends, while on the server side if the user logs out the server waits for a new connection from another client before going into the loop again.

Client/Server Operations: As for the operations, as detailed in the image above we have a list of client and server functions, but since the server does most of the work many times the instructions the client has to carry out are so few that they are simply put into the switch case, for example on the client side "Print transactions" looks like this:

```
case 4:
    printf("");
    char msgg[MESSAGE_SIZE] = "";
    msgg[0] = 'T';
    strcat(msgg, username);
    send(commsoc, msgg, MESSAGE_SIZE, 0);
    recvLoop(commsoc, msgg, MESSAGE_SIZE);
    printf("\n%s\n", msgg);
    break;
```

On the server side, though, it looks like this:

server.c

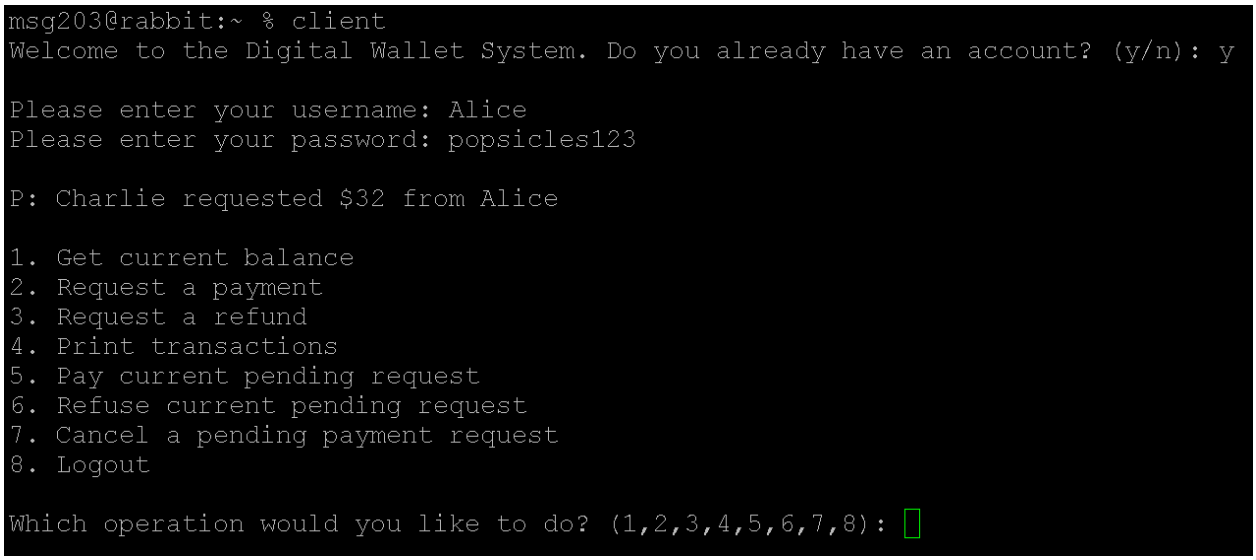
```
case 'T':
    memmove(buf, buf+1, strlen(buf));
    response = get_ledger(buf);
    break;
```

common.c

```
char* get_ledger(char* user) {
    char* ret = malloc(MESSAGE_SIZE);
    FILE* f = open_file("ledger", "r");
    char line[256];
    while(fgets(line, sizeof(line), f)) if(strstr(line, user)) strcat(ret, line);
    fclose(f);
    return ret;
}
```

All of the other operations are performed in a similar way, with most of the work being done on the server and the client simply constructs messages to send to the server and prints response messages it gets back.

User Menu: Finally, the user menu looks like the picture below, it is very simple and easy to use as you simply enter a number and you get a response back about the success of the operation or an error if something went wrong. There are no sub-menus, so everything you see below is what the user sees (besides the response messages, and prompts for input such as recipient and payment amount for operations like “Request a payment”). Before this menu is displayed, the user must first tell the program whether they want to register an account or login, and once they have done either of those they are now considered authenticated and they are presented with the operations menu.



Verification

Bob tries to make a payment but he doesn't have enough money

```
Which operation would you like to do? (1,2,3,4,5,6,7,8): 1

Your balance is $0.000000

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8): 5
P: Alice requested $500 from Bob

You do not have enough money in your account to make that payment!

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8):
```

Bob makes a $500 payment and his balance is changed from $800 to $300

```
Your balance is $800.000000

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8): 5
P: Alice requested $500 from Bob

You have paid the current pending transaction!
```

```
Your balance is $300.000000

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout
```

Bob requests a refund since he regrets making that payment

```
Which operation would you like to do? (1,2,3,4,5,6,7,8): 3

Who would you like to request a refund from?: Alice
What was the amount of this transaction?: 500

Your payment request has been submitted.

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8):
```

Alice refuses to pay his refund request since she is greedy

```
msg203@rabbit:~ % client
Welcome to the Digital Wallet System. Do you already have an account? (y/n): y

Please enter your username: Alice
Please enter your password: popsicles123

P: Bob requested $500 from Alice

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8): 6
P: Bob requested $500 from Alice

You have denied a pending transaction. The ledger has been updated.

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8):
```

Alice requests a payment from Charlie, but changes her mind and cancels it before he gets to pay it

```
Who would you like to request money from?: Charlie
How much money would you like to request?: 666

Your payment request has been submitted.

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8): 7

Who did you request payment from that you would like to cancel?: Charlie
What was the amount of this payment request?: 666

Your pending transaction has been removed from the ledger.

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout
```

Finally, Alice prints her transaction history

```
msg203@rabbit:~ % client
Welcome to the Digital Wallet System. Do you already have an account? (y/n): y

Please enter your username: Alice
Please enter your password: popsicles123

You are now logged in.

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8): 4

C: Alice requested $500 from Bob
C: Charlie requested $32 from Alice
X: Bob requested $500 from Alice

1. Get current balance
2. Request a payment
3. Request a refund
4. Print transactions
5. Pay current pending request
6. Refuse current pending request
7. Cancel a pending payment request
8. Logout

Which operation would you like to do? (1,2,3,4,5,6,7,8):
```

Conclusion

Given its intended purpose, which was to function as a simple and easy to use non-concurrent user digital wallet system, the project was a success. It is very easy to follow what is going on since none of the operations are abstracted away inside objects, so all you have to do to understand the flow of the program is go to the client, find the case in the switch statement for the operation you want to follow, see what the first character of the message that was constructed is, then go to the case in the switch statement on the server side that corresponds to that character, and follow the code from there which may call a server function or may be done inside the case. For some operations there is a client side function and a server side function, but once the user is authenticated and the input loop is running the program flow always follows the same pattern: go to the appropriate case in the client's switch statement, follow the code/functions and see what message is constructed, go to the corresponding case in the server's switch statement, follow the code/functions and see what response the server sends back for the client to print. Admittedly, using structs or objects could have made some parts of the program shorter/less time consuming to write, and in future projects I will try to consolidate and generalize certain functions so there aren't so many of them (especially on the server side). With multithreading, this architecture is extendable to a concurrent user system by running server operation loops on multiple threads, but if you wanted to do concurrent users with the UDP protocol then this entire architecture and messaging system would have to be redesigned. Overall though, the project was a success in that all the required operations were implemented and tested and it ended up being a very intuitive and user-friendly digital wallet system.

Source Code

client.c

```
#include "common.h"

int main() {
	char header = welcome();
	int commsoc = create_socket();
	int choice;

	char user[100];
   char pass[100];
   printf("\nPlease enter your username: ");
   scanf("%s", user);
   printf("Please enter your password: ");
   scanf("%s", pass);
	printf("\n");

	char* username = malloc(strlen(user)+1);
	strcpy(username, user);

	int pending = login(header, commsoc, user, pass);

	while(1) {
		printf("1. Get current balance\n");
		printf("2. Request a payment\n");
		printf("3. Request a refund\n");
		printf("4. Print transactions\n");
		printf("5. Pay current pending request\n");
		printf("6. Refuse current pending request\n");
		printf("7. Cancel a pending payment request\n");
		printf("8. Logout\n");
		printf("\nWhich operation would you like to do? (1,2,3,4,5,6,7,8): ");

		scanf("%d", &choice);

		switch(choice) {
			case 1:
				cget_balance(commsoc, username);
				break;
			case 2:
				crequest_payment(commsoc, username);
				break;
			case 3:
				printf("\nWho would you like to request a refund from?: ");
```

```
        char from[100];
        char amt[50];
        scanf("%s", from);
        printf("What was the amount of this transaction?: ");
        scanf("%s", amt);
        char msg3[MESSAGE_SIZE] = "";
msg3[0] = 'G';
char* str = "C: ";
strcat(msg3, str);
strcat(msg3, from);
str = " requested $";
strcat(msg3, str);
strcat(msg3, amt);
str = " from ";
strcat(msg3, str);
strcat(msg3, username);
        send(commsoc, msg3, MESSAGE_SIZE, 0);
        recvLoop(commsoc, msg3, MESSAGE_SIZE);
        printf("\n%s\n\n", msg3);
        break;
case 4:
        printf("");
        char msgg[MESSAGE_SIZE] = "";
        msgg[0] = 'T';
        strcat(msgg, username);
        send(commsoc, msgg, MESSAGE_SIZE, 0);
        recvLoop(commsoc, msgg, MESSAGE_SIZE);
        printf("\n%s\n", msgg);
        break;
case 5:
        pending = login(header, commsoc, user, pass);
        if(!pending) {
                printf("You have no pending transactions!\n\n");
                break;
        }
        char msg[MESSAGE_SIZE] = "";
        msg[0] = 'P';
        strcat(msg, username);
        send(commsoc, msg, MESSAGE_SIZE, 0);
        recvLoop(commsoc, msg, MESSAGE_SIZE);
        printf("%s\n", msg);
        break;
case 6:
        pending = login(header, commsoc, user, pass);
        if(!pending) {
                printf("You have no pending transactions!\n\n");
```

```
                        break;
                    }
                    char msg2[MESSAGE_SIZE] = "";
                    msg2[0] = 'N';
                    strcat(msg2, username);
                    send(commsoc, msg2, MESSAGE_SIZE, 0);
                    recvLoop(commsoc, msg2, MESSAGE_SIZE);
                    printf("%s\n", msg2);
                    break;
                case 7:
                    printf("\nWho did you request payment from that you would like to
cancel?: ");
                    char fromm[100];
                    char amount[50];
                    scanf("%s", fromm);
                    printf("What was the amount of this payment request?: ");
                    scanf("%s", amount);
                    char msg4[MESSAGE_SIZE] = "";
                    msg4[0] = 'C';
                    char* strr = "P: ";
                    strcat(msg4, strr);
                    strcat(msg4, username);
                    strr = " requested $";
                    strcat(msg4, strr);
                    strcat(msg4, amount);
                    strr = " from ";
                    strcat(msg4, strr);
                    strcat(msg4, fromm);
                    send(commsoc, msg4, MESSAGE_SIZE, 0);
                    recvLoop(commsoc, msg4, MESSAGE_SIZE);
                    printf("\n%s\n\n", msg4);
                    break;
                default:
                    send(commsoc, "X", MESSAGE_SIZE, 0);
                    close(commsoc);
                    return 0;
                    break;
            }
        }

        free(username);
        close(commsoc);
        return 0;
}
```

server.c

```
#include "common.h"

void listen_loop(int ssoc) {
	struct sockaddr_in fsin;
	int size = sizeof(fsin);
	int commsoc;
	char buf[MESSAGE_SIZE];
	char* response = malloc(MESSAGE_SIZE);
	char* user, *pass;

	NEWCONNECTION:

	if((commsoc = accept(ssoc, (struct sockaddr*)&fsin, &size)) < 0) {
		printf("accept error!\n");
		exit(-1);
	}

	while(1) {
		response = "";
		recvLoop(commsoc, buf, MESSAGE_SIZE);

		switch(buf[0]) {
			case 'C':
				memmove(buf, buf+1, strlen(buf));
				response = cancel_payment_request(buf);
				break;
			case 'G':
				memmove(buf, buf+1, strlen(buf));
				response = request_refund(buf);
				break;
			case 'T':
				memmove(buf, buf+1, strlen(buf));
				response = get_ledger(buf);
				break;
			case 'N':
				memmove(buf, buf+1, strlen(buf));
				update_ledger(buf, 'X');
				response = "You have denied a pending transaction. The ledger has
been updated.\n";
				break;
			case 'J':
				memmove(buf, buf+1, strlen(buf));
				user = strtok(buf, " ");
				pass = strtok(NULL, " ");
```

```
                response = add_user(user, pass);
                break;
            case 'L':
                memmove(buf, buf+1, strlen(buf));
                user = strtok(buf, " ");
                pass = strtok(NULL, " ");
                response = login_user(user, pass);
                char* request;
                if(response[0] == 'Y' && (request = has_payment_request(user)))
response = request;
                break;
            case 'B':
                memmove(buf, buf+1, strlen(buf));
                char strbal[50];
                snprintf(strbal, 50, "%f", get_balance_f(buf));
                response = strbal;
                break;
            case 'R':
                memmove(buf, buf+1, strlen(buf));
                response = request_payment(buf);
                break;
            case 'P':
                memmove(buf, buf+1, strlen(buf));
                char* req = has_payment_request(user);
                char* reqcopy = malloc(strlen(req)+1);
                strcpy(reqcopy, req);
                char* to = strtok(reqcopy, " ");
                to = strtok(NULL, " ");
                double bal = get_balance_f(buf);
                char* requestbalance = strtok(req, " ");
                for(int i = 0; i < 3; i++) requestbalance = strtok(NULL, " ");
                requestbalance++;
                double balreq = strtod(requestbalance, NULL);
                if(balreq > bal) {
                    response = "You do not have enough money in your account to
make that payment!\n";
                    break;
                }
                double res = bal-balreq;
                char param[50];
                snprintf(param, 50, "%f", res);
                set_bal(user, param);
                double floaty = get_balance_f(to);
                floaty += balreq;
                snprintf(param, 50, "%f", floaty);
                set_bal(to, param);
```

```
                        update_ledger(user, 'C');
                        response = "You have paid the current pending transaction!\n";
                        break;
                case 'X':
                        goto NEWCONNECTION;
                        break;
        }

        send(commsoc, response, MESSAGE_SIZE, 0);
    }
    close(commsoc);
}

int main() {
    int ssoc = start_server();

    listen_loop(ssoc);

    return 0;
}
```

common.h

```
#ifndef COMMON_H
#define COMMON_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 2001
#define MESSAGE_SIZE 500

//both
int recvLoop(int csoc, char* data, const int size);

//client
void cget_balance(int commsoc, char* user);
int is_yes(char* input);
int create_socket();
char welcome();
int login(char header, int commsoc, char* user, char* pass);
void crequest_payment(int commsoc, char* user);

//server
int start_server();
FILE* open_file(char* path, char* mode);
int user_exists(char* user);
char* add_user(char* user, char* pass);
char* login_user(char* user, char* pass);
char* request_payment(char* msg);
char* get_balance(char* user);
double get_balance_f(char* user);
char* has_payment_request(char* user);
void set_bal(char* user, char* newbal);
void update_ledger(char* user, char character);
char* get_ledger(char* user);
char* request_refund(char* req);
char* cancel_payment_request(char* request);

#endif
```

common.c

```
#include "common.h"

//both
int recvLoop(int csoc, char* data, const int size) {
    int nleft, nread;
    char* ptr = data;
    nleft = size;
    while(nleft > 0) {
        nread = recv(csoc, ptr, nleft, 0);
        if(nread < 0) {
            printf("Read error!\n");
            break;
        }
        else if(nread == 0) {
            printf("Socket closed!\n");
            break;
        }
        nleft -= nread;
        ptr += nread;
    }
    return size-nleft;
}

//client
void cget_balance(int commsoc, char* user) {
    char msg[MESSAGE_SIZE] = "";
    msg[0] = 'B';
    strcat(msg, user);
    send(commsoc, msg, MESSAGE_SIZE, 0);
    recvLoop(commsoc, msg, MESSAGE_SIZE);
    printf("\nYour balance is $%s\n", msg);
        printf("\n");
}

int is_yes(char* input) {
    char* yesarray[2] = { "y", "yes" };
    for(int i = 0; i < 2; i++) if(strcmp(input, yesarray[i]) == 0) return 1;
    return 0;
}

int create_socket() {
    struct addrinfo hints, *res;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
```

```
    hints.ai_socktype = SOCK_STREAM;
    if(getaddrinfo("localhost", NULL, &hints, &res)) {
        printf("getaddrinfo error!\n");
        exit(-1);
    }
    int ret = -1;
    if((ret = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0) {
        printf("socket error!\n");
        exit(-1);
    }
    struct sockaddr_in* inaddr = (struct sockaddr_in*)(res->ai_addr);
    inaddr->sin_port = htons(PORT);
    if(connect(ret, res->ai_addr, res->ai_addrlen) < 0) {
        printf("connect error!\n");
        exit(-1);
    }
    return ret;
}

char welcome() {
    char answer[3];
    printf("Welcome to the Digital Wallet System. Do you already have an account? (y/n):
");
    scanf("%s", answer);
    char ret = 'L';
    if(!is_yes(answer)) {
        printf("Would you like to create one? (y/n): ");
        scanf("%s", answer);
        if(!is_yes(answer)) exit(0);
        ret = 'J';
    }
    return ret;
}

int login(char header, int commsoc, char* user, char* pass) {
    strcat(user, " ");
    strcat(user, pass);
    char msg[MESSAGE_SIZE] = "";
    msg[0] = header;
    strcat(msg, user);
    send(commsoc, msg, MESSAGE_SIZE, 0);
    recvLoop(commsoc, msg, MESSAGE_SIZE);
    printf("%s\n", msg);
    if(strstr(msg, "Error")) {
            send(commsoc, "X", MESSAGE_SIZE, 0);
            close(commsoc);
```

```
            exit(-1);
        }
        if(msg[0] == 'P') return 1; else return 0;
}

void crequest_payment(int commsoc, char* user) {
    char from[100];
    char amount[100];
    printf("\nWho would you like to request money from?: ");
    scanf("%s", from);
    printf("How much money would you like to request?: ");
    scanf("%s", amount);
    char msg[MESSAGE_SIZE] = "";
    msg[0] = 'R';
    char* str = "P: ";
    strcat(msg, str);
    strcat(msg, user);
    str = " requested $";
    strcat(msg, str);
    strcat(msg, amount);
    str = " from ";
    strcat(msg, str);
    strcat(msg, from);
    send(commsoc, msg, MESSAGE_SIZE, 0);
    recvLoop(commsoc, msg, MESSAGE_SIZE);
    printf("\n%s\n\n", msg);
}

//server
int start_server() {
    struct sockaddr_in sin;
    int ssoc;
    memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons((unsigned short)PORT);
    if((ssoc = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        printf("socket error!\n");
        exit(-1);
    }
    if(bind(ssoc, (struct sockaddr*)&sin, sizeof(sin)) < 0) {
        printf("bind error!\n");
        exit(-1);
    }
    if(listen(ssoc, 5) < 0) {
        printf("listen error!\n");
```

```
        exit(-1);
    }
    return ssoc;
}

FILE* open_file(char* path, char* mode) {
    FILE* ret;
    if((ret = fopen(path, mode)) == NULL) {
        printf("Error opening file\n");
        exit(-1);
    }
    return ret;
}

int user_exists(char* user) {
    FILE* f = open_file("users", "r");
    int curr = 0;
    char line[256];
    while(fgets(line, sizeof(line), f)) {
        if((curr == 0 || (curr % 4) == 0) && strstr(line, user)) return curr+1;
            curr++;
    }
    fclose(f);
    return 0;
}

char* add_user(char* user, char* pass) {
    if(user_exists(user)) return "Error: that username is already taken!\n";
    FILE* f = open_file("users", "a+");
    fputs(user, f);
    fputs("\n", f);
    fputs(pass, f);
    fputs("\n0\n\n", f);
    fclose(f);
    return "Your account has been created with a balance of $0.\n";
}

char* login_user(char* user, char* pass) {
    int linenum;
    if((linenum = user_exists(user)) == 0) return "Error: that username does not exist!\n";
    FILE* f = open_file("users", "r");
    char line[256];
    for(int i = 0; i < linenum+1; i++) fgets(line, sizeof(line), f);
    strcat(pass, "\n");
    if(strcmp(line, pass) != 0) return "Error: password incorrect!\n";
    fclose(f);
```

```
    return "You are now logged in.\n";
}

char* get_balance(char* user) {
    FILE* f = open_file("users", "r");
    int curr = 0;
    char* line = malloc(256);
    while(fgets(line, sizeof(line), f)) {
        if((curr == 0 || (curr % 4) == 0) && strstr(line, user)) {
            fgets(line, sizeof(line), f);
            fgets(line, sizeof(line), f);
            break;
        }
        curr++;
    }
    fclose(f);
    return line;
}

double get_balance_f(char* user) {
    FILE* f = open_file("users", "r");
    int curr = 0;
    char line[256];
    while(fgets(line, sizeof(line), f)) {
        if((curr == 0 || (curr % 4) == 0) && strstr(line, user)) {
            fgets(line, sizeof(line), f);
            fgets(line, sizeof(line), f);
            break;
        }
        curr++;
    }
    fclose(f);
        double ret = strtod(line, NULL);
        return ret;
}

char* request_payment(char* msg) {
    char* msgorig = malloc(strlen(msg)+1);
    strcpy(msgorig, msg);
    char* from = strtok(msg, " ");
    for(int i = 0; i < 5; i++) from = strtok(NULL, " ");
    if(!user_exists(from)) return "Error: that user does not exist!";
    FILE* f = open_file("ledger", "a+");
    fputs(msgorig, f);
    fputs("\n", f);
    fclose(f);
```

```
    return "Your payment request has been submitted.";
}

char* has_payment_request(char* user) {
    FILE* f = open_file("ledger", "r");
    char line[256];
    while(fgets(line, sizeof(line), f)) {
        if(line[0] == 'P') {
            char* origline = malloc(strlen(line)+1);
            strcpy(origline, line);
            char* name = strtok(line, " ");
            for(int i = 0; i < 5; i++) name = strtok(NULL, " ");
            if(strstr(name, user)) {
                fclose(f);
                return origline;
            }
        }
    }
    fclose(f);
    return NULL;
}

void set_bal(char* user, char* newbal) {
    FILE* r = open_file("users", "r");
    FILE* w = open_file("users_temp", "w");

    char line[256];
    int curr = 0;
    int lineno;
    while(fgets(line, sizeof(line), r)) {
        fputs(line, w);
        if((curr == 0 || (curr % 4) == 0) && strstr(line, user)) {
            fgets(line, sizeof(line), r);
            fputs(line, w);
            fgets(line, sizeof(line), r);
            fputs(newbal, w);
                    fputs("\n", w);
            curr = curr + 2;
        }
        curr++;
    }
    fclose(r);
    remove("users");
    fclose(w);
    rename("users_temp", "users");
}
```

```
void update_ledger(char* user, char character) {
    FILE* r = open_file("ledger", "r");
    FILE* w = open_file("ledger_temp", "w");
    char line[256];
    int curr = 0;
    int lineno;
    while(fgets(line, sizeof(line), r)) {
        if(line[0] == 'P') {
                    char* linecpy = malloc(strlen(line)+1);
                    strcpy(linecpy, line);
                    char* name = strtok(linecpy, " ");
                    for(int i = 0; i < 5; i++) name = strtok(NULL, " ");
                    if(strstr(name, user)) line[0] = character;
                    free(linecpy);
        }
              fputs(line, w);
        curr++;
    }
    fclose(r);
    remove("ledger");
    fclose(w);
    rename("ledger_temp", "ledger");
}

char* get_ledger(char* user) {
    char* ret = malloc(MESSAGE_SIZE);
    FILE* f = open_file("ledger", "r");
    char line[256];
    while(fgets(line, sizeof(line), f)) if(strstr(line, user)) strcat(ret, line);
    fclose(f);
    return ret;
}

char* request_refund(char* req) {
    FILE* f = open_file("ledger", "r");
    char line[256];
    int found = 0;
    while(fgets(line, sizeof(line), f)) if(strstr(line, req)) found = 1;
    if(!found) return "A completed ledger as described could not be found!";
    char* from = strtok(req, " ");
    from = strtok(NULL, " ");
    char* amt = strtok(NULL, " ");
    amt = strtok(NULL, " ");
    char* to = strtok(NULL, " ");
    to = strtok(NULL, " ");
```

```
    char msg[MESSAGE_SIZE] = "";
    char* str = "P: ";
    strcat(msg, str);
    strcat(msg, to);
    str = " requested ";
    strcat(msg, str);
    strcat(msg, amt);
    str = " from ";
    strcat(msg, str);
    strcat(msg, from);
    return request_payment(msg);
}

char* cancel_payment_request(char* request) {
    FILE* r = open_file("ledger", "r");
    FILE* w = open_file("ledger_temp", "w");
    char line[256];
    int deleted = 0;
    while(fgets(line, sizeof(line), r)) if(strstr(line, request)) deleted = 1; else fputs(line, w);
    fclose(r);
    remove("ledger");
    fclose(w);
    rename("ledger_temp", "ledger");
    if(!deleted) return "A pending transaction as described could not be found!";
    else return "Your pending transaction has been removed from the ledger.";
}
```