Project 1: Banking with TCP

Authors: Tom Cardy & Mario Delprete

Class: CISC-450: Computer Networks

Date: 4/2/2017

Summary of Programs

Implement a Server which manages a bank account and a client that interacts with the server by using TCP to request transactions on the account.

Purpose of the Server

The server maintains a single checking account and savings account, which the client has access too. When the server launches, the checking and savings account are initialized at 0. The client can then make four requests to the server:

- Check the Balance of either account
- Deposit a specified amount from an account
- Withdraw a specified amount from an account
- Transfer a specified amount from one account to another

All of the transactions a user may request from the server are done in whole dollars, and vice versa. There are some artificial limitations applied to the server. The checking and savings account will not store decimal values, like $20.34. Furthermore, the amounts requested from the client cannot exceed what the account has, and if the amount is over $1000000. The client can only request withdrawals in multiples of 20. The client will also not be allowed to make withdraws from the savings account. Lastly, a transfer account will only be allowed if the source account has the funds available to allow this. When the server catches one of these errors made, it will not complete the request and send a message back to the user notifying why.

Purpose of the Client

The client navigates through an interface menu that creates and tailors a message to be sent to the server holding the type of transaction request and the amounts requested to be manipulative if required by a certain transaction. The client then sends this message, displays the size and details about the message, and begins to wait for a response from the server. After the message is received, the client then displays the user the results of the transaction, displaying the size and details about the message. The client can then continue to make more requests on the

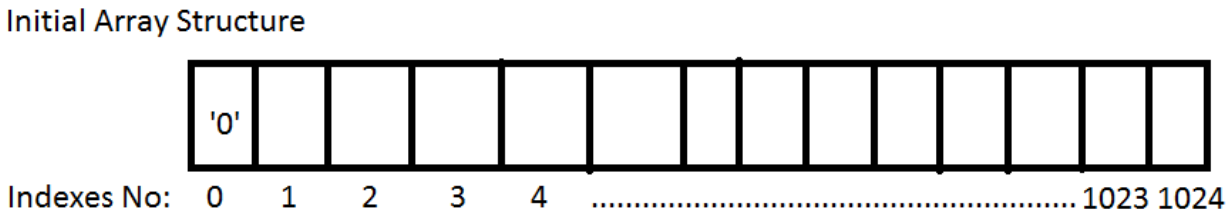
same connection. If the user is done with transaction, they can select the exit option to close the client and the connection.

Message Format

The format of the messages being sent is a character array. This array, taken from a skeleton program supplied by the instructor of the course, is able to hold and send 1024 characters. The size of a message being sent can range based on the transaction. These sizes can be compared below throughout the walkthrough of the program.

Data Structure Concept

Looking at the template for our message we can dissect these indexes it into multiple parts



Our message can have up to 1024 different characters in its array, but we use at most, 43 indexes. That being said, of those 43, the first 4 indexes are of the upmost importance.

Index 0: This is our error flag character. We initialize this index to ‘0’ each time and expect to see the same character when the message returns. The error codes that may be returned are as follows:

1. The client attempted to withdraw from a savings account
2. The amount requested in a withdraw exceeds the amount in the account
3. The amount requested for a withdraw was not in multiples of 20
4. The amount requested for transfer exceeds the balance of the source account

Index 1: This is our Transaction type. This index is initialized with a character number between 0-3. This allows the server to proceed with transactions based on what character is in this index

Index 2: This is our Source Account Index. This index is initialized with either a character of 0 or 1. This will notify which Account the client wanted to interact with.

Index 3: This is our Destination Account Index. This Index is only used for Transfers. This lets the server know what Account the destination of the transfer will occur. It is generally initialized with a 0 character in non-transfer transactions to maintain the integrity of the message.

Index 4-23: This is our Binary Digits Indexes. These indexes will hold 20 binary digits. These Binary digits will represent the amount of money the client wants to withdraw, deposit, or transfer. Regardless of the size of the amount requested, each of these array indexes will contain a 0 or a 1 to maintain the integrity of the message. Even if the number is 20. The number will be represented as “00000000000000010100”. Like above, these extra 0s at the front of the binary number are kept to maintain integrity of the message

Index 24-43: These 20 indexes serve the exact same purpose as indexes 4-23, but for the balance of the transfer destination. These indexes are only used for transfers essentially. The server is the only part of the project that inputs characters in this part of the array. The client only reads these characters and reports them back as an integer value.

Server-Side Functions and Classes:

1. Class: Account
    a. 3 Objects
        i. int account (Account Number)
        ii. char type (Checking or Saving)
        iii. int balance (funds available)
    b. 4 Functions
        i. Int Deposit
        ii. Int Withdraw
        iii. Int Check_balance
        iv. Int transfer
2. Init_account(Account& a, int arg1, char arg2, int arg3)
    a. This function will initialize the accounts
        i. For test runs, this will be 0
3. change_amount_sentence(char sentence[], int amount, int index)
    a. Function that changes the data type of the returned Integer of getAmount()
        i. First from Integer to a Bitset
        ii. Then from a Bitset to a Binary String
    b. The String is then loaded into the character message array
4. Process_check_balance(char sentence[])
    a. This function will access the account class object and get the balance using two functions
        i. Checking.check_balance
        ii. Savings.check_balance
    b. Ends with tailoring message to be sent back by doing step a in reverse
5. process_withdraw(char sentence[])
    a. This function will take the message received by casting characters in indexes 4-23 cast them into a string and place that binary string in a bitset
    b. The function will access the account class object and perform withdraws using two functions
        i. Checking.withdraw

        ii. Savings.withdraw (Which should report an error)
    c. Ends with tailoring message to be sent back by doing step a in reverse
6. process_deposit(char sentence[])
    a. This function does the same as process_withdraw (5.a)
    b. The function will access the account class object and perform deposits using two functions
        i. Checking.deposit
        ii. Savings.deposit
    c. Ends with tailoring message to be sent back by doing step a in reverse
7. process_transfer(char sentence[])
    a. This function does the same as process_withdraw (5.a)
    b. The function will then access the account class object and perform transfers using two functions
        i. Checking.transfer
        ii. Savings.transfer
    c. Ends with tailoring message to be sent back by doing step a in reverse
        i. Transfers though will report back the balance in both account so this process will be done twice
8. process_transaction(char sentence[])
    a. This function enables the functions above by looking at index 1 of the message received and passing it to the appropriate process function. These functions are
        i. Process_check_Balance
        ii. Process_deposit
        iii. Process_withdraw
        iv. Process_transfer

Client-Side Functions:

1. displayMainMenu(char sentence[])
    a. Function that displays the menu interface and requests the user to make a selection.
    b. Begins to Tailor the Message by:
        i. Initializes a 0 in the error control
        ii. Initializes the transaction type
    c. Will return an integer
        i. 0 to continue more transactions
        ii. 1 to break loop in main and break connection and exit
2. accountSelectMenu(string s)
    a. Function that displays a request for what type of account the client wants to interact with
    b. Returns the choice
        i. 0 for Checking
        ii. 1 for Savings
3. getAmount()

    a. Function that prompts the User to enter the amount of money they are requesting
    b. Whole Dollars only
    c. Returns the amount as an integer
4. Change_amount_sentence(char sentence[], int amount, int index)
    a. Function that changes the data type of the returned Integer of getAmount()
        i. First from Integer to a Bitset
        ii. Then from a Bitset to a Binary String
    b. The String is then loaded into the character message array
5. get_amount_sentence(char mSentence[], int index)
    a. Same concept as change_amount_sentence function but does it in reverse
    b. Creates a String from the proper indexes of the received message and places it in a bitset
    c. The bitset is then casted into an int using the <bitset>.to_long() function
    d. The Function returns an integer of the number
6. processCheckBalance(char sentence[])
    a. This simple function tailors the message to be sent with characters required to send a check balance request.
    b. The 0 index of the array was already set to 0 from the displayMainMenu() function
    c. The 1 index of the array was already set to 0 as well from displayMainMenu()
    d. Index 2 is initialized based on whether the user selected 0 for checking or 1 for savings from accountSelectMenu()
    e. Index 3, used for transfers only is initialized to 0 just to maintain the structure's integrity
    f. The same integrity rule applies for Index 4, which is only required for withdrawing, deposits, and transfers.
7. processDeposit(char sentence[])
    a. Deposit Function initializes indexes the same way as processCheckBalance(), but for Deposits
        i. The difference would be that index 1 would have a 1 representing Deposits
    b. Index 2 is initialized based on whether the user selected 0 for checking or 1 for savings from accountSelectMenu()
    c. Index 3, used for transfers only is initialized to 0 just to maintain the structure's integrity
    d. Index 4, and the indexes that follow, rely on the getAmount() method for the user to input how much money they want to deposit
    e. getAmount()'s integer returned gets passed to the change_amount_sentence(), so it can be converted to a binary string and loaded into our message to be sent.
8. processWithdraw(char sentence[])
    a. It is exactly the same function as processDeposit but index 1 of our message will contain a 2 for withdraws, given to us by the method accountSelectMenu()
9. processTransfer(char sentence[])

    a. This function tailors the message for Transfers
        i. Index 1 will contain a 3, representing Transfer request.
    b. Using the accountSelectMenu() function, the user will specify which is the source (src) account or the destination (dst) account
    c. The Source account will be represented as a 0 (Checking) or 1(Savings) and be placed in Index 2
    d. The Destination account will be represented as a 0 (Checking) or 1 (Savings) and be placed in Index 3.
    e. Index 4, and the indexes that follow, rely on the getAmount() method for the user to input how much money they want to deposit
    f. getAmount()'s integer returned gets passed to the change_amount_sentence(), so it can be converted to a binary string and loaded into our message to be sent.
10. processRequest(char sentence[])
    a. This simple method will determine which transaction the user is interested in and will follow it up with the proper processXXXXXX() method
        i. Index 1 determines this since it represents our transaction types
11. processSuccessReturn(char mSentence[], string s)
    a. This method verifies that the transaction was received with no errors and prints out the transaction was a success and the current balance of that account
    b. The amount is given by passing our received message to get_amount_sentence
        i. We start at index 4 because that's where the binary characters for the balance of that account begins
12. processSuccessTransfer(char mSentence[])
    a. This method does the same as processSuccessReturn, but is for Transfers only
    b. We use the same get_amount_sentence to get the amount but do it twice
        i. The first is the same as processSuccessReturn's get_amount_sentence function calling, starting at 4 because that's where the binary characters for the balance begins for the source account
        ii. The second call does the same as above but starts at index 24 because that's where the binary characters for the balance begins for the destination account
13. processReturn(char mSentence[])
    a. This method checks the first index of the received message
    b. If a character assigned in index 0 is not a 0, an error occurred
        i. The program will print an error based on what character is in the index
    c. If the no error has occurred, the function will look at the transaction type and then point to the proper function
        i. processSuccessReturn for Checking Balances, Deposits, and Withdraws
        ii. processSuccessTransfer for Transfers

Client Interface Basics

Our Port Number: 5045

A client will boot up this application and will be prompted to enter a hostname and port number, given above. If there is a server with the matching hostname and port, the server will respond and the connection will be established. The User is then prompted with a basic switch statement menu asking them what are they interested in doing today. The options for choices are as follows:

- 0 for Check Balance
- 1 for Deposits
- 2 for Withdraws
- 3 for Transfers
- 4 to exit and close the connected socket

It is important to note that the numbers 0-3 in this switch menu, will be loaded as a character in the second index of our character array, to represent which transaction they wanted to go through. No matter which of the four transactions that the user selects, the first index, representing error control, will be initialized with a 0 to represent zero errors. After the user has selected their transactions, they will be taken to the appropriate function that will tailor the message appropriately.

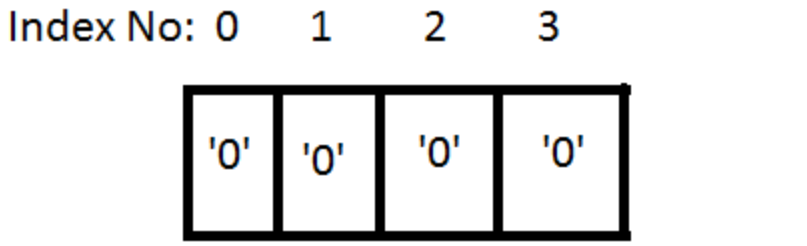
Basic Scenarios – Presented from Client Side Perspective

Before Scenarios can be presented, it is important to understand that all scenarios essentially begin and end with the same processes. This would be initializing the 0 index of a message with a 0 character to represent zero errors in our error control index. The 1 index of the message will contain a character between 0-3 representing what type of transaction they want. These two indexes will be applied like so for every scenario.

As for the end processes, all messages received go through the same processReturn(). Almost all of these transactions are handled then by processSuccessReturn(), except for transfers which are handled by proccessSuccessTransfer(). These two functions simply report back the success of the transaction, and print out the balances. The balances are found by doing a reverse process with getting the characters in indexes 4-23, putting them in a bitset, and casting that bitset back into an integer. If the error flag is not set to 0 in the first index, the processReturn() function will notify the user what happened and they can retry again if they wish.

Going further, these scenarios are told from the perspective of the client. That being said, the server repeats the process in reverse

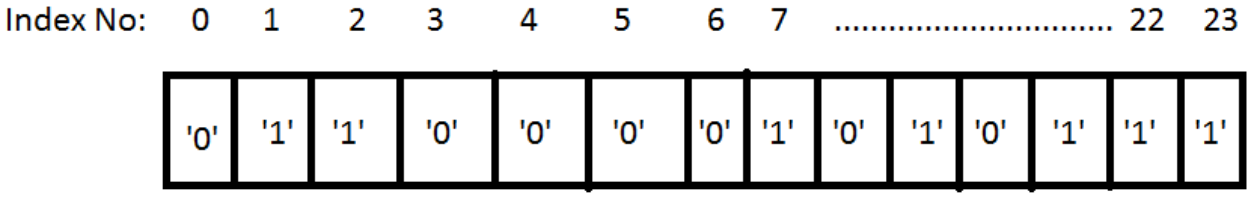
A Check Balance Scenario



Check Checking Account Sample

So if the user selected that they wanted to check a balance, the client will then prompt them which account they want to check based on the AccountselectMenu() Function. The choices will be a 0 for a Checking Account and a 1 for a Savings Account. Using the proccessRequest() function, the function checks index 1 of our message and sees that it contains a 0 character, which means the client wants to check a balance, so it begins the processCheckBalance() function. Just like the switch statement we used to ask them for what transaction, the client takes the 0 or 1 the user just inputted from AccountselectMenu() function for which account they wanted to check and places this character in index 2 of the message. Indexes 3 and 4 are set to 0 only to ensure the integrity of the message we send will be maintained. The message is now completed and ready to be sent. The client takes the size of the message and sends it along with the message through the socket, using the send function for tcp. The system will then wait on the server to respond back. Once the server has returned the message, it will begin the reading process, found above in paragraph 2 of "Basic Scenarios."

A Deposit/Withdraw Scenario

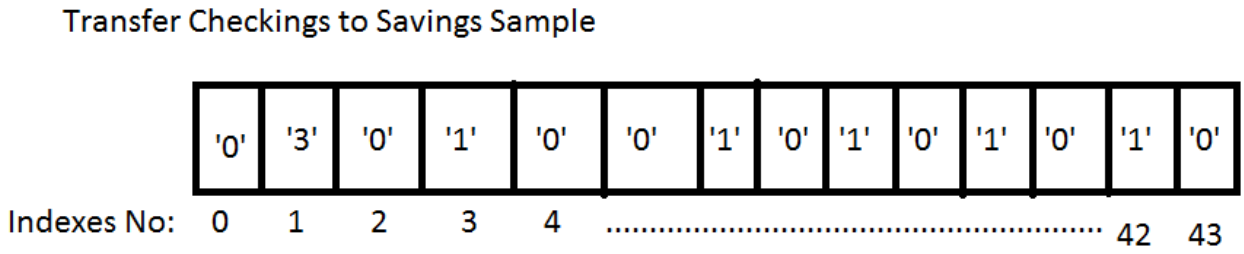


Savings Account Deposit Sample

These two scenarios are so similar that one scenario description should suffice. If the user selected 1 or 2 for Deposit/Withdraw in our Main Menu, then a 1 or 2 will be found in the 1 index of our message. Like in the Check Balance Scenario, The user will be taken to the AccountselectMenu(), where they will input a 0 (Checking) or 1 (Saving). That character will then be placed in index 2 of our message. Then the user is taken to processWithdraw() or processDeposit(). During this process, the client also interacts with the getAmount() method, where they will be asked what amount they wish to withdraw or deposit, only in whole dollars and in multiples of 20. Once the number is inputted, it is then put through the change_amount_sentence() method that will cast it into a bitset and then load it in the character

array from index 4 to 23. Index 3 is essentially ignored but initialized to 0 since it is used only for transfers. The message is ready at this point and it is sent and the client begins to wait for a response. Once the message is received, the message is analyzed as described in paragraph 2 of "Basic Scenarios" found above.

A Transfer Scenario



This scenario follows the same structure as the other transactions, but with an additional requirement. While the user is selecting what account they wish to interact with during the AccountselectMenu(), the client checks if the user chose the transfer option by checking for a 3 in index 1 of our message. If it does find a 3, it prompts a different account selection choice, asking for a 0 or 1 for the source account or a 0 or 1 for the destination account. The source account is stored in index 2, while the destination is stored in index 3. Then the process continues like in a normal withdraw/deposit but asking the amount they wish to transfer instead, all stored in indexes 4-23. After this is done, it gets sent like a normal message. This message though is filled with almost twice the amount of characters than normally. Unlike a normal message, the server included an additional 20 characters from 24-43. This is why during the processReturn() function, we pass our message to a special function just for transfers called proccessSuccessTransfer(), which will take the balance of both accounts involved in the transfer, by taking the binary digits found in indexes 4-23 and 24-43, and print out the balances for both.