CSE 434: COMPUTER NETWORKS

PROJECT-1: CHECKPOINT - Complete Project Design Document
Group #15

Team Members:
Narasimha Arun Oruganti - 1223956669 (norugant@asu.edu)
Sai Rahul Reddy Kandula - 1225500107 (skandu12@asu.edu)

Project description:

The primary goal of this project is to create a Socket project that implements the CHECKPOINT application using the Client-server architecture and peer-to-peer architecture. We have two main processes in this application: Bank and Customer. The Checkpoint application is a distributed application in which we implement a checkpoint to ensure that the global state is consistent across all customers and banks. The bank process acts as a server, allowing customers to open accounts, create cohorts, delete cohorts, and exit applications. Customers can issue commands from the terminal using the customer process, which runs as clients. These commands are sent to the bank server by the customer processes. Customers have the ability to deposit and withdraw funds from their accounts. Money can be transferred among customers who are part of a cohort. Customers use checkpoints to save globally consistent states. When there is a lost transfer or any inconsistent state, the customer can also force the peers in the cohort to roll back to the latest checkpoint to ensure a consistent global state is established between the peers.

In milestone 1, we have already implemented the bank process and multiple customer processes and also performed the actions to open a bank account, create a cohort, delete a cohort, and exit the process. For the final project, the main goal is to establish peer-to-peer communication between customer processes, perform deposits, withdrawal, and transfer operations and maintain a consistent global state with the ability to roll back to a previous consistent state.

System Architecture:

The CHECKPOINT algorithm is implemented using a distributed system that consists of multiple nodes, including the bank server and multiple customer clients. The bank server provides a centralized location where all customer data is stored, maintains the cohort details, and also additionally the exit state of the customers for fast access to the data. The customer clients allow customers to access their accounts and the cohort details to perform transactions within the cohort. The bank server listens continuously on a single port only unlike the customer process. Each customer process is bound to two nodes out of which one is used for communication with other customers (peer-to-peer architecture) while the other port is used to communicate with the bank (Client-Server architecture). We have implemented a looping mechanism where the client process checks the ports in a loop to see if there is data available.

Bank Process:

The bank process runs as a server and provides services to customers, such as opening accounts, creating cohorts, deleting cohorts, and exiting the checkpoint application. The bank process communicates with customers using UDP sockets. When a customer sends a request to the bank process, the bank process reads the request, updates the customer's data in its database, and sends a response to the customer. The bank process uses a CSV file to store customer data as the main database, and it reads and writes data to the file when it updates a customer's data.

Customer Process:

The customer process can communicate with the bank to Open a bank account, create a cohort with the existing customers, delete a cohort, and exit the checkpoint application. The customer can communicate with the peers present in the cohort. If they are not in the cohort the peers cannot communicate with each other. The customer can perform self-functions like deposit and withdrawal. While the transfer, lost transfer, checkpoint, and rollback operations are performed with the peers in the cohort.

Communication Protocol:

The Bank and Customer System uses a custom communication application-layer protocol for communication between the bank process and customer processes. The underlying network protocol is implemented using UDP protocol. For each process, the running program creates a UDP socket and starts communicating through the specified user defined ports to communicate with other processes.

As per the project description, each group has designated ports calculated with group number. For our group, the group number is **15.** The port numbers calculated for our group are from **8500 - 8999**. Care is taken in the code to only accept the ports between the given range.

The protocol is a text-based protocol, where each message consists of a command and a set of parameters separated by spaces. The protocol supports the following commands:

Message Format for the Commands at Bank:

1. **open <name> <balance> <ip_address> <port1> <port2>**: opens a new bank account with the given name, initial balance, and network information. If the account is successfully created, the bank process returns **“SUCCESS”** code to the customer, else it returns **“FAILURE”** code. When the bank account is created successfully, the details are stored in the object of the class “cohortCustomerClass”.

   Below is the example command:
   open arun 100 10.120.70.146 8500 8501. The command tells that there is a customer “arun” who wants to “open” a bank account, with a minimum balance of “100” USD and the

“ipv4-address” of the customer is “10.120.70.105”, port1 - 8500 (used for communicating with bank) port2 - 8501 (used for communicating with peer customer).

2. **new-cohort <name> <n>**: Initiates the new-cohort by the customer with the given name with subset customers of size n. The bank checks the list of existing customers in the database who are not part of any cohort. If there are sufficient customers to form the cohort of size n, the bank goes ahead and forms the cohort. Once the cohort is formed successfully, the bank returns a **cohort tuple** which is converted to a string and sent as a response to the customer. Else, the bank sends a **“FAILURE”** message to the customer. At the customer end, the customer checks for a **“FAILURE”** message. If the response is not a failure message, the customer sends the tuple to the other customers present in the cohort.

   Below is the example command:
   new-cohort arun 2. This command tells the bank that the user “arun” wants to start a new-cohort consisting of 2 customers in cohort (including himself).

3. **delete-cohort <name>:** Deletes the cohort with the given name if he is in any cohort. When the bank process receives the delete-cohort command, the bank first checks if the user is present in any of the cohorts. If the user is not in any cohort, the bank sends a **“FAILURE”** message. Else the bank sends a delete-cohort command to users in the same cohort. Once the other customers in the cohort receive the message, they delete the cohort details at their end and send a **“SUCCESS”** message to the bank. Upon receiving all the **“SUCCESS”** messages from every customer, the bank then sends a **“SUCCESS”** message to the original customer who initiated the delete-cohort. Else the bank sends a **“FAILURE”** message to the original customer.

   Below is the example command:
   delete-cohort arun. This command tells that the user “arun” wants to delete the cohort to the bank. The bank checks if he is in any cohort and does the operation accordingly.

4. **exit <name>:** Exits the checkpoint application with the given name of account holder if he is not in any cohort. When the bank receives the exit command, the bank checks if the customer is part of any cohort or not. If he is a part of any cohort, then the bank sends a **“FAILURE”** message to the customer process. Else, the bank sends a **“SUCCESS”** message to the customer. Upon receiving the **“SUCCESS”** message, the bank sets the exit_flag and exits from the application.

   Below is the example command:
   exit arun. This command tells the bank that customer arun wants to exit the application. The application checks if the user is in any active cohort. If the customer is in the cohort, then the exit command fails. Else, the customer is exited from the application.

Message format for the commands at Customer:

Below are the command formats that are used at the customer end to communicate between the peers.

1. **deposit <amount>:** This command is used to deposit the amount into the bank account of the customer. The object cohortCustomer.currentBalance is updated to existing bank account balance.

   Below is the example command:
   deposit 10. When the user enters the command, the currentBalnce is updated by 10 and is displayed on the command line.

2. **withdrawal <amount>:** When user enters this command, the current bank balance is checked and if the entered amount is less than the current balance, then the withdrawal is performed. Or else the withdrawal operation will fail.

   Below is the example command:
   withdrawal 100. The currentBalance of the customer is checked if it is less than 100 or not. If the balance is less than 100, then the withdrawal is not performed. If the balance is greater than 100, then the withdrawal will be successful and the updated balance is displayed on the command line.

3. **Transfer <amount> <q> <label>:** The user at the time of entering the command will not know what label to enter. So, when user inputs the command, he/she enters it as **transfer <amount> <receiver>**. The currentBalance of the sender is checked if the transfer can be performed or not. By comparing the receiver in the **cohortCustomer** object, the **firstLableSent** for the respective **receiver** is put in the **<label>** field and is sent to the respective customer. The **<q>** here refers to the receiver process which we are sending the money. At the receiving end, the checkpoint cohort is updated and the sender is added into the checkpoint cohort. The lastLabelreceived field is updated with the **<label>** field in the command. The firstLabelSent variable in the command is compared with the lastLabelReceived variable at the receiver end to find out if there is any inconsistency in the system. Based on the comparison, the variable **“oKToTakeChkPoint”** is updated. If there is any inconsistency, the variable is set to **“No”** or else **“Yes”**.

   Below is the example command:
   User entered command at customer process C2: transfer 20 c1.
   When the user enters the above command, at the C2 end, the currentBalance is checked with 20 and if the balance is greater than 20, the transfer operation is performed. The sender then checks the object and updates his firstLabelSent filed of customer c1 and includes it in the new command that is sent to the receiver. Let this be 1 for suppose. At the receiver, the lastLabelReceived field of the sender is updated by the value 1 which is included in the command. While sending, the command would be something like **transfer 20 c1 1**.

4. **lost-transfer <amount> <q>:** This command is used to simulate the lost event of a transfer message. When a transaction is lost, the system is forced into an inconsistent state. Similarly when the user enters a lost transfer command, the local variables at the sender end are updated but the transfer will not happen. In the above message format, the amount is deducted from the current bank balance of the customer and the firstLabelSent is incremented at the sender side but it is never sent to the receiver.

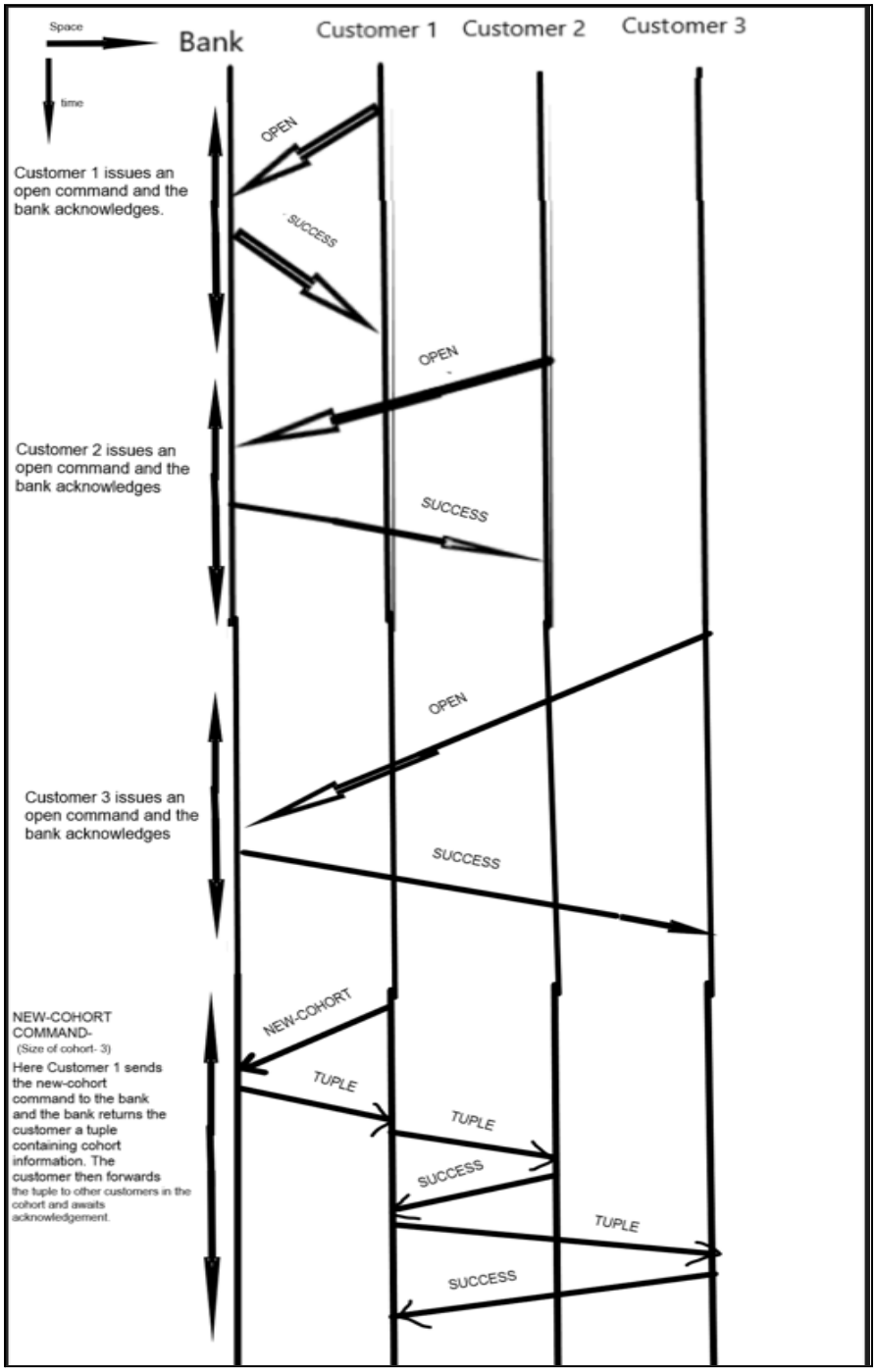
   Below is the example command:
   lost-transfer 20 c1. Here the customer C2 is trying to send a lost-transfer message to customer C1. The bank balance is deducted by 20 only of the currentBalance is greater that 20. Once it is deducted, the firstLabelSent for customer C1 is incremented.

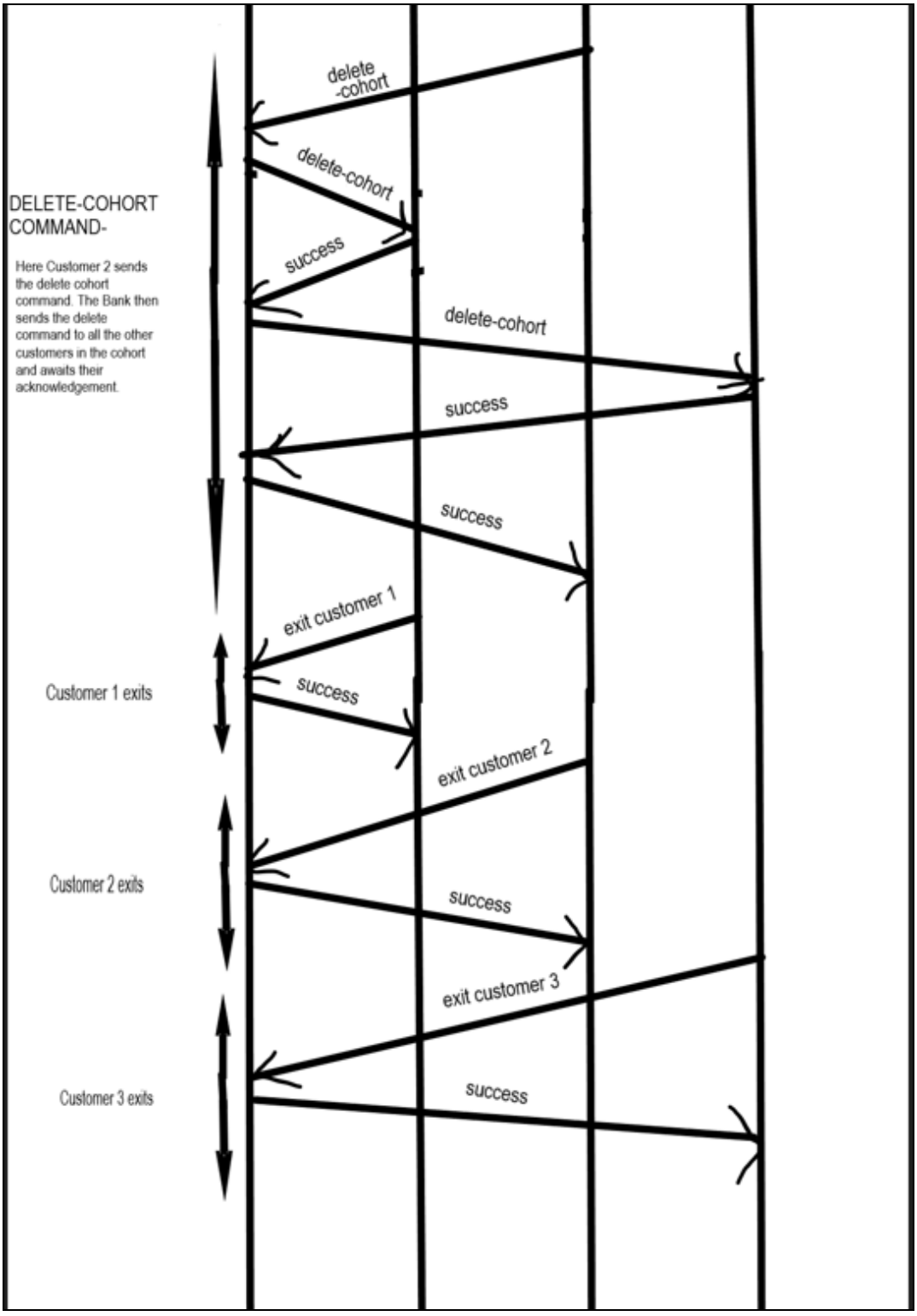
5. **checkpoint:** The format of this command is just the word checkpoint. When user enters this command, the customer will send a command called **"take_tentative_check_point"** message to the customers in it's checkpoint cohort. The customers, who receive the message **"take_tentative_check_point"** along with **<name of process>**, **<lastLabelReceived>**, will save the state variables into the object called **"tentativeCheckPoint"**. After taking the checkpoint, the customer will in-turn send the "take_tentative_check_point" message to the members in its cohort. Finally, the customers will reply either **"Yes"** or **"No"** to their respective sender customers in a chain fashion. Once all the customer processes replies **yes**, then the sender will automatically send **"make_tentative_check_point_permananent"** message to the customers processes in the checkpoint cohort. Upon receiving the make permanent checkpoint message, the customer processes save the state variables to an object called **"permanentCheckPoint"**. If the customer sends a **no** to **take_tentative_checkpoint**, then the sender sends a **"undo_tentative_check_point"** to all the customers in the checkpoint cohort. After all processes take a permanent checkpoint then the **chkptCohort** is deleted.

6. **rollback:** This command is used to perform a rollback operation when there is any inconsistency in the global state. When the user enters the rollback command, the sender will form a rollback cohort and store it in the variable called **rollCohort** and will send a **"prepare_to_rollback"** with a variable **"lastLabelRecvd"** message to all the customers in the rollback cohort. The rollback cohort is nothing but all customers in the **cohort_tuple**. When a customer receives the **prepare_to_rollback** message, the customers check for the **willingToRollBack, resumeExecution** flag and compares the **lastLabelRecvd** to **lastLabelSent**. Based on the results, receiver customer sends **"prepare_to_rollback"** to all the customers in the rollback cohort. The receiver customer can send either **yes** or **no** to **prepare_to_rollback** message. If they reply **yes**, the sender process will now send **"roll_back"** message to all customers in the rollback cohort to roll back to a consistent permanent checkpoint. If they reply no, the sender will send **"do_not_roll_back"** message to customers in rollback cohort. **rollCohort** is deleted after the rollback is successful.

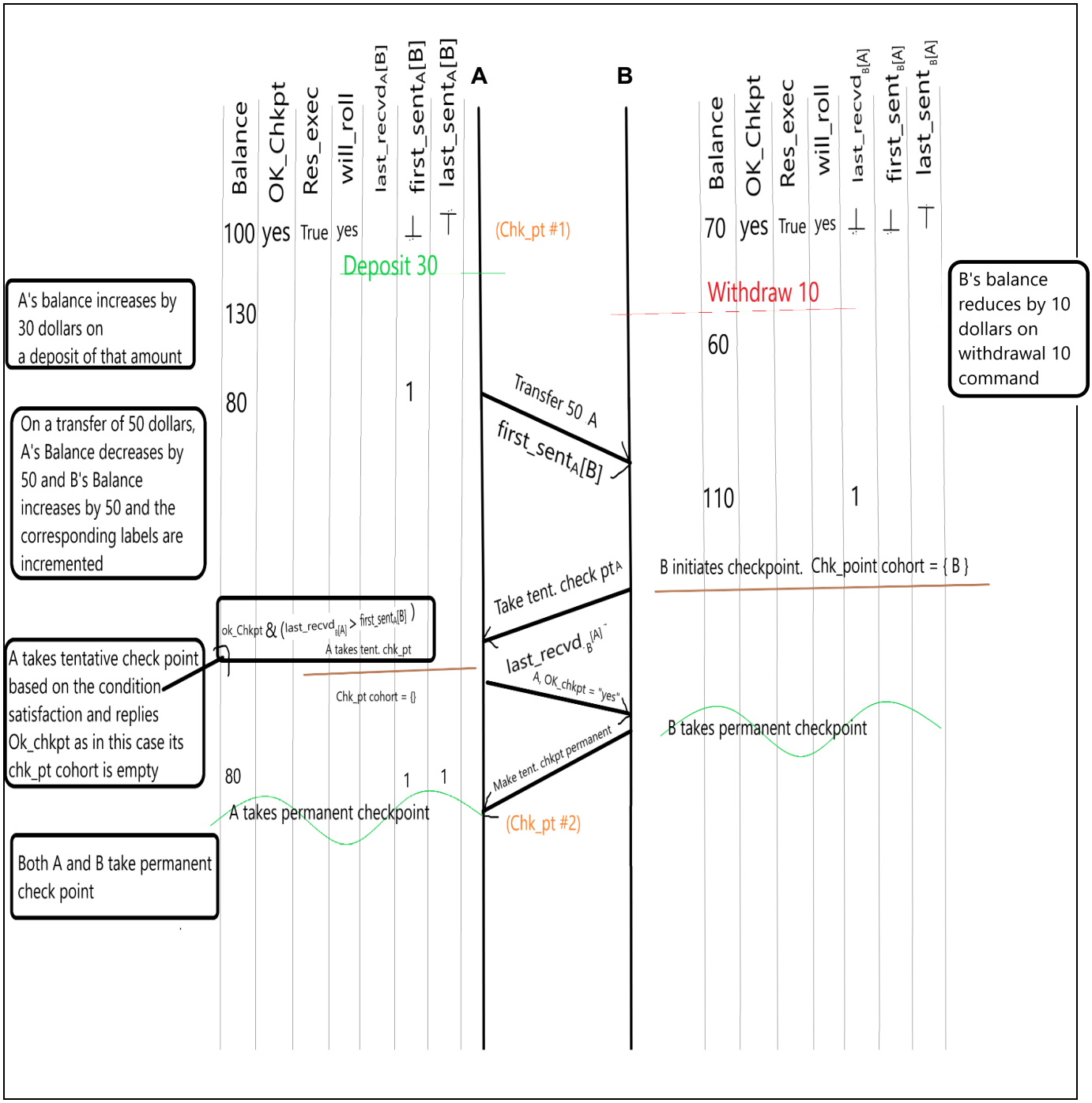
Time Space Diagram:

Below is the time space diagram explaining the application testing and how the commands are exchanged between the bank and customers processes.



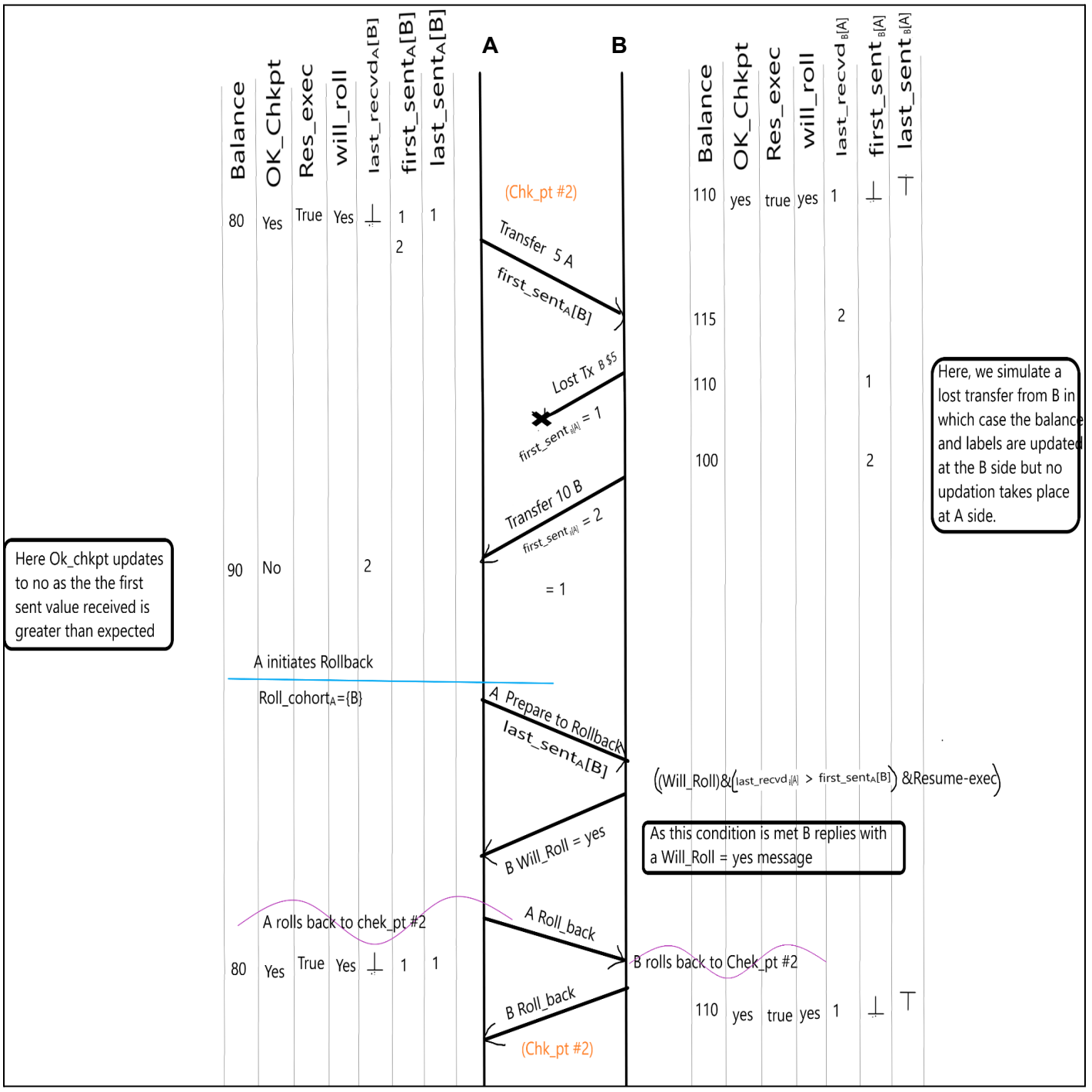

DELETE-COHORT COMMAND-
Here Customer 2 sends the delete cohort command. The Bank then sends the delete command to all the other customers in the cohort and awaits their acknowledgement.
delete -cohort
delete-cohort
success
delete-cohort
success
success
exit customer 1
success
Customer 1 exits
exit customer 2
Customer 2 exits
success
exit customer 3
Customer 3 exits
success

DEPOSIT, WITHDRAW AND CHECKPOINT OPERATIONS (between customers A and B)



LOST TRANSFER AND ROLLBACK OPERATIONS

(Below space diagram is a continuation of the previous time space diagram from Check Point #2)



Design decisions and Algorithm Implementation:

Our team implemented the project in python. There are two files **1. Bank_server.py** and **2. Customer_client.py.**

The **Bank_server.py** utilizes the socket module for network communication, and CSV files for storing customer data. The server-side of the banking system performs different operations such as opening a new account, creating a new cohort of customers, deleting a cohort of customers, and exiting the application.

The design decisions include the use of UDP socket programming for communication between the server and the client, the use of CSV files for storing customer data, and the use of Python dictionaries for storing customer records.

The main functions in the code are:

1. **open_customer(name, balance, ip_address, port1, port2):** This function adds a new customer to the CSV file with the specified customer details, such as name, balance, IP address, and port numbers.
2. **new_cohort(name, n):** This function creates a new cohort of customers by randomly selecting n customers (excluding the specified customer name) who are not part of any cohort and assigning them to a new cohort group.
3. **delete_cohort(name):** This function deletes the cohort for the specified customer name by assigning a cohort number of 0 to all the customers present in the specified customer's cohort group.
4. **exit_customer(name):** This function checks if the customer is in any cohort. If the customer is not part of the cohort then the customer can exit the application or else the function sends a failure message.

The **Customer_client.py** implements the client side functionality that communicates with a bank server and other peers in a network. The code has two main functions: **bankWorker()** and **peerWorker()**, which respectively handle communication with the bank server and with peers in the network.

The **bankWorker()** function takes the user's input as a command and sends it to the bank server over a socket. The function waits for a response from the bank server and prints it to the console. If the user inputs the "new-cohort" command and receives a successful response from the bank server, the bankWorker() function calls sendCohortDetailsToPeers() to send the details of the new cohort to all other peers in the network. If the user inputs the "delete-cohort" or "exit" command and receives a successful response from the bank server, the function sets the cohort_tuple to an empty list and sets the exit_flag to True.

The **peerWorker()** function takes the user's input as a command and sends it to other peers in the network over a socket. The function waits for a response from the peers and prints it to the console.

The **sendCohortDetailsToPeers()** function sends the details of the new cohort to all other peers in the network. The function first gets the IP address and port number of the client socket connected to the bank server. It then iterates over the list of other peers in the network and sends the cohort details to each peer that is not the current client socket. If a peer receives the details successfully, the function prints a success message to the console.

The main part of the code creates two sockets, one for communicating with the bank server and one for communicating with peers in the network. It then enters a loop that listens for user input and data on both sockets. When a user inputs a command on the console, the code checks if it is a valid command for either the bank or the peer. If it is a valid command, the code calls the appropriate function to handle the command.

The code uses the select module to listen for data on both sockets and the console simultaneously. The select function waits for a socket to become readable before continuing the loop. This prevents the code from blocking while waiting for a response from a socket.

The **peerWorker()** functions utilizes the following functions.

1. **self_functions(input_command):** The deposit and withdrawal operations are performed in this function. This is local to the customer_client.py
2. **checkpoint():** This is the function that is called when the user enters the checkpoint command in the command line. It triggers the sending of the take checkpoint messages to all the customers in the check point cohort starting with the initiator process. If all processes reply with "yes", the tentative checkpoint is made permanent. Otherwise, the tentative checkpoint is undone.
3. **take_tentative_chkpt(senderPeer, last Received):** At all processes, when a message is received to take a tentative checkpoint, the process checks if it is okay to take a checkpoint and if the last label received is greater than or equal to the first label sent. If so, the process takes a tentative checkpoint and sends a message to all processes in the checkpoint cohort to take a tentative checkpoint. If all processes reply with "yes", the process can take a checkpoint, otherwise, it cannot.
4. **local_tentative_chkpt():** helper function to save the cohortCustomer object data to tentativeCheckPoint object.
5. **local_permanent_chkpt():** helper function to save the tentativeCheckPoint object data to permanentCheckPoint object.
6. **make_permanent_chkpt():** When a message is received to make a tentative checkpoint permanent, the process makes the checkpoint permanent and sends a message to all processes in the checkpoint cohort to do the same. After making the permanent checkpoint, all the label variables are set to default and the checkpoint cohort is deleted.
7. **undo_tentative_chkpt():** When a message is received to undo a tentative checkpoint, the process undoes the checkpoint and sends a message to all processes in the checkpoint cohort to do the same.
8. **rollback():** The initiator process sends a "Prepare to Rollback" message to all processes in its rollback cohort. If all processes reply with "yes," then the initiator process sends a

"Rollback" message to all processes in its rollback cohort. If any process replies with "no," then the initiator process sends a "Do Not Rollback" message to all processes in its rollback cohort.

9. **prepare_to_rollback(senderPeer, lastSent):** When a process receives a "Prepare to Rollback" message from another process, it checks its state variables to determine if it is willing to roll back. If it is willing to roll back and the last label received from the sender is greater than the last label sent by the receiver, then the process sets its state variable to "no" and sends a "Prepare to Rollback" message to all processes in its rollback cohort. If all processes reply with "yes," then the process sets its state variable to "yes" and sends a "willing to roll" message to the sender. Otherwise, it sets its state variable to "no" and sends a "not willing to roll" message to the sender.
10. **loc_roll_back():** helper function to rollback to the previously taken permanent checkpoint.
11. **peer_roll_back():** If a process receives a "Rollback" message and its state variable is set to "false," then it restarts from its permanent checkpoint and sends a "Rollback" message to all processes in its rollback cohort.
12. **do_not_roll_back():** If a process receives a "Do Not Rollback" message, then it sets its state variable to "true," resumes execution, and sends a "Do Not Rollback" message to all processes in its rollback cohort.

Data Structures used:

Below are the data structures used in the **Bank_server.py**:

Lists:

1. customers: to store the customer data read from the CSV file, and to keep track of new customers. It is a list of dictionaries.
2. availablecohorts: to keep track of the customers available for cohort formation.
3. cohortTuples: to store the tuples formed in cohort creation.

Dictionaries:

1. customer: it is a temporary dictionary used to loop through each dictionary in the customers list.

Strings and integers: to store various input/output messages and numeric values used in the program.

Below are the classes data structures used in the **Customer_client.py**:

Classes:

1. **cohortCustomerClass**: a class contating all the state variables needed for running the checkpoint algorithm and rollback recovery algorithm.
   Below are the members and member function of the class with brief description of the data structures used.
   - **name**: A string that represents the customer's name.
   - **ipAddress**: A string that represents the customer's IP address.
   - **currentBalance**: A float that represents the current balance of the customer's bank account.

- **lastLabelrecvd**: A dictionary that stores the last label received from each cohort peer.
- **firstLabelSent**: A dictionary that stores the first label sent to each cohort peer.
- **lastLabelSent**: A dictionary that stores the last label sent to each cohort peer.
- **oKToTakeChkPoint**: A string that indicates whether the customer is okay to take a checkpoint. The default value is "Yes".
- **willingToRollBack**: A string that indicates whether the customer is willing to roll back. The default value is "Yes".
- **resumeExecution**: A string that indicates whether the customer is ready to resume execution. The default value is "Yes".
- **rollCohort**: A list that stores the customers in the roll cohort.
- **chkptCohort**: A list that stores the customers in the checkpoint cohort.
- **print_data():** This method is used to print the customer data, and the initializeData method is used to initialize the customer data by updating the dictionaries firstLabelSent, lastLabelrecvd, and lastLabelSent with the appropriate values for each cohort peer. The cohort_tuple variable is assumed to be a global variable that contains information about the cohort peers.
- **initializeData():** This method is used to initialize the customer data. It iterates over the cohort_tuple list, which is assumed to be a global variable that contains information about the cohort peers, and updates the firstLabelSent, lastLabelrecvd, and lastLabelSent dictionaries with the appropriate values for each cohort peer. For each peer in the cohort_tuple list, the firstLabelSent dictionary is updated with the value 0, the lastLabelrecvd dictionary is updated with the value 0, and the lastLabelSent dictionary is updated with the value 999.

Objects of Classes:

1. **cohortCustomer**: it is used to store and update the local variables when there are local operations performed at the customer end like deposit, withdrawal and transfer operations.
2. **tentativeCheckPoint**: it is used as part of checkpoint algorithm to store the local state variables when ever the customer is instructed to take a tentative checkpoint.
3. **permanentCheckPoint**: it is used as part of checkpoint algorithm to store the tentative checkpoint to permanent checkpoint when ever the customer is instructed to make a tentative checkpoint permanent. It is also used by the rollback algorithm to rollback to a consistent global state.

Lists:

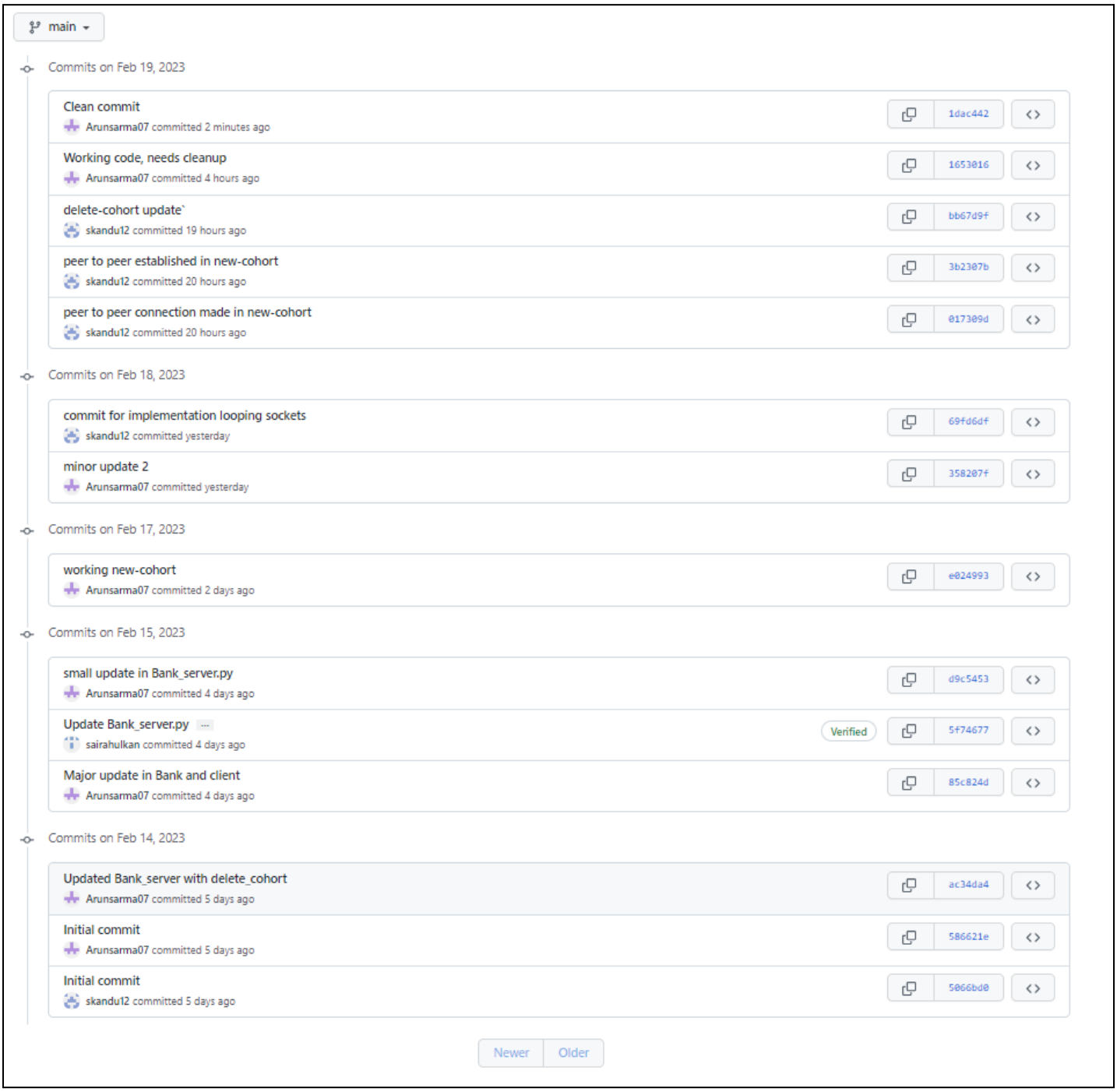
1. cohort_tuple: a list containing dictionaries with details about the different peers in the cohort.
2. sockets: a list containing the client's socket objects.

Strings and boolean flags:

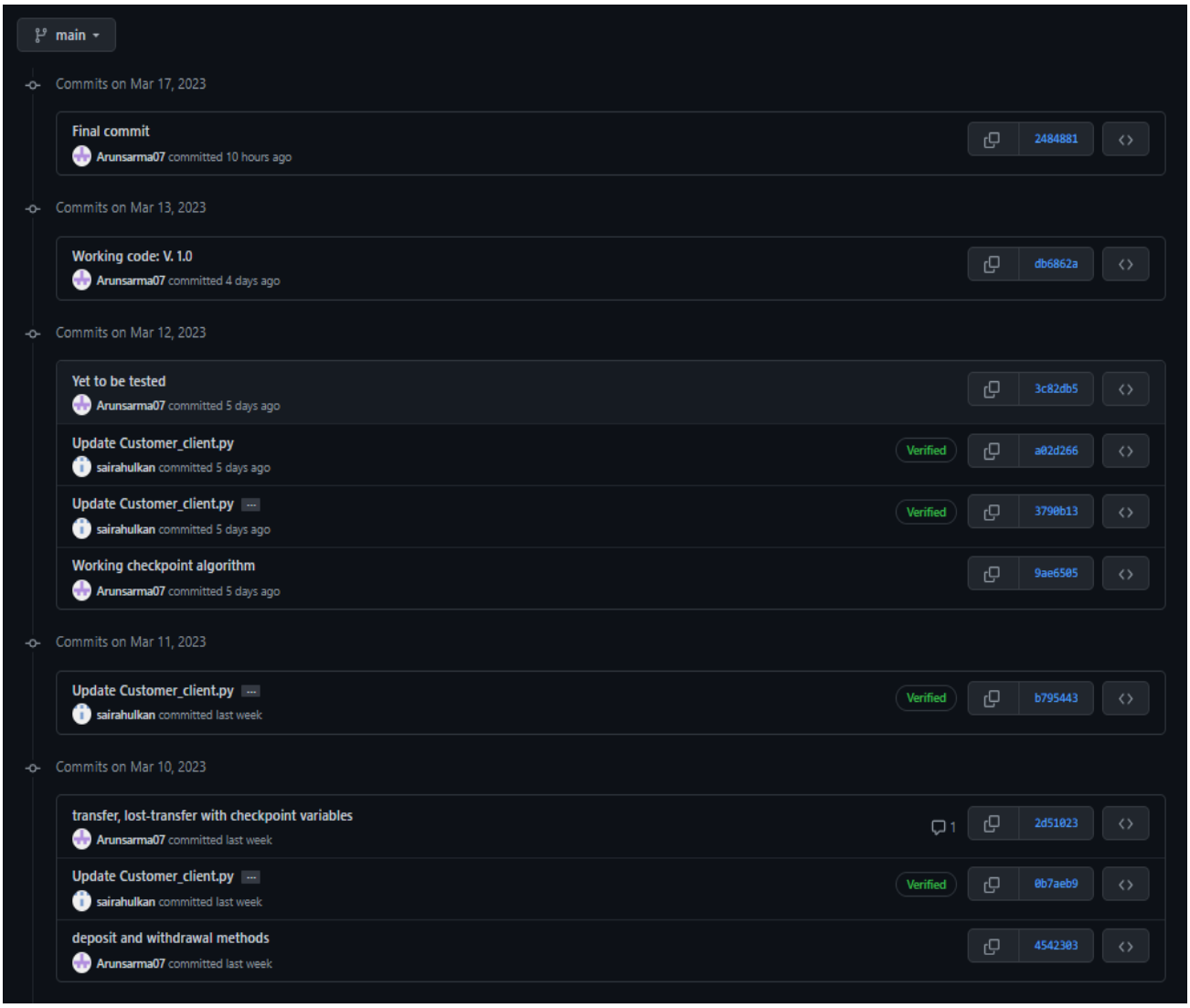
1. exit_flag: a boolean flag indicating whether the process should exit.
2. input_command: a string representing the user's command input.

Screenshot of version control system:

For this project our team chose to use git to maintain the code. Below is the screenshot of the commits made during the development of the project during milestone 1.



Below is the screenshot of the commits made during the development of the project after milestone1 till the completion of project.



Project demonstration video link:

Below is the youtube link for the complete project video demonstration.

Link: https://youtu.be/EOo-s11fU5k