MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DISTRIBUTED SYSTEMS (CSE -3261) MINI PROJECT REPORT ON

Distributed Banking System- FlexiBank Connect

SUBMITTED TO

Department of Computer Science & Engineering

by

Anurag Nayak, 210905016, anurag.nayak@learner.manipal.edu
Adruti Onam, 210905190, adruti.onam@learner.manipal.edu
Vishal Puneyani, 210905037, vishal.puneyani@learner.manipal.edu
Kuber Lamba, 210905070, kuber.lamba@learner.manipal.edu

6th A

Venkatesh Bhandage

Name & Signature of Evaluator 2

(Jan 2024 – May 2024)

Table of Contents

Chapter 1. Introduction

This client-server application implements an intuitive distributed financial system using sockets and the client-server architecture. The client program and the server program are the two main parts of the project.

The server program responds to client queries and keeps the banking system functioning. It utilizes Python's socket module to accept and handle client connections while listening on a socket for incoming connections. The server responds to each client request and keeps track of the status of the banking system using a dictionary data structure that associates account numbers with account objects.

The distributed-bank management system is intended to serve as an easy-to-understand demonstration of how to create a distributed application utilizing sockets and client-server architecture. It can serve as a springboard for creating more intricate distributed systems or as a tool for instructing students on how to use Python to learn about distributed systems and network programming.

1.1 Introduction to the Problem Statement

In the context of distributed systems, the problem statement for this project is to create a banking management system that can manage several clients at once and enable them to open new accounts, make deposits and withdrawals, move money across accounts, and check their balances. The system should be scalable and fault-tolerant, which means it should be able to recover from errors and manage rising traffic volumes without losing data.

In the context of a distributed banking application, database replication plays a crucial role in enhancing data availability and fault tolerance. By replicating data across multiple servers, the system ensures that in the event of a server failure, the application can continue to operate seamlessly, as users can be redirected to another server where a replica of the data exists. This redundancy is vital for banking applications, where data availability is critical for transaction processing and customer satisfaction.

1.2 Motivation

Given that a distributed banking system can offer a number of advantages, such as greater performance and scalability, increased dependability and availability, and higher availability, this problem statement is best understood in the context of distributed systems. Furthermore, to prevent data loss in the event of a failure, the server keeps a log of all transactions.

1.3 Approach

The banking management system can be created as a distributed application, where the financial services allow clients to communicate with the server and carry out various banking transactions, the project makes use of sockets and client-server architecture. Utilizing a dictionary data structure that associates account numbers with account objects, the server keeps track of the status of the banking system and responds to client queries. The client program offers a command-line interface through which users can submit requests to the server, which executes them, and receives responses from the server. This project, on its whole, serves as a fundamental illustration of how to construct a distributed system using sockets and client-server architecture.

Implementing database replication brings forth significant challenges, particularly in achieving consistency and synchronization across the distributed system. Ensuring that all replicas are updated simultaneously without causing data conflicts or inconsistencies requires sophisticated coordination mechanisms. The distributed nature of these applications complicates the scenario further because network latency and partitioning can lead to scenarios where not all replicas can be updated at the same time, potentially leading to outdated or conflicting data being served to users. Balancing consistency with availability and latency, often framed within the boundaries of the CAP theorem, is a complex but necessary endeavor for maintaining the integrity and reliability of banking applications.

Chapter 2. Background Theory and Literature Review

2.1 Background Theory

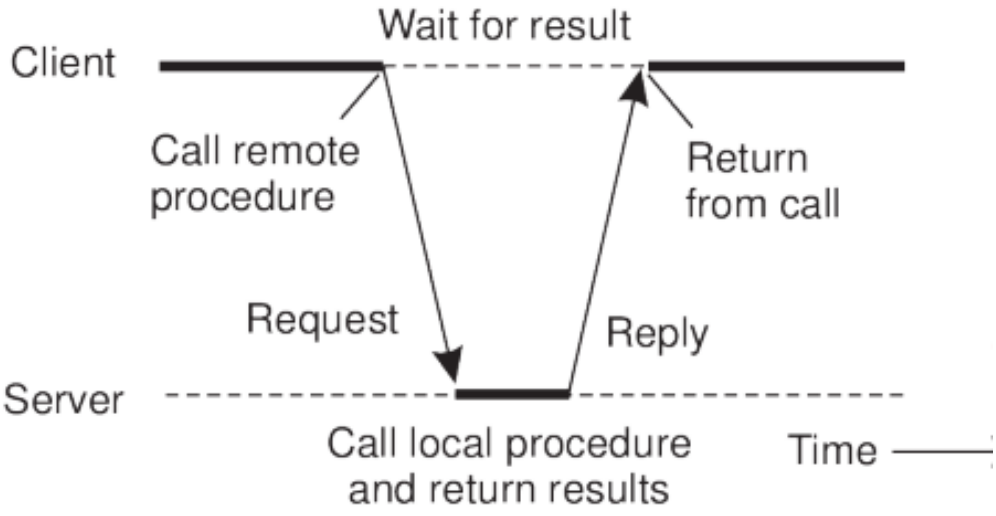
2.1.1 RPC

A distributed system's RPC (Remote Procedure Call) protocol enables communication between various processes or services executing on various computers. It enables a client process to call a distant server's procedure or method as if it were local.

In an RPC system, a client submits a request to a different server, which responds to the request by processing it and returning the outcome to the client. A network connection, such as the Internet, is typically used to communicate between the client and server. The methods and data structures used in communication are

defined by the client and server using a common interface description language (IDL).

Fig: 2.1 The principle of RPC between a client and server program[1]



2.1.2 ACID Properties

ACID stands for four characteristics that guarantee dependable and consistent transactions. Atomicity, consistency, isolation, and durability are these characteristics[2].

Atomicity: When a transaction is seen to be a single, indivisible unit of labor, it is said to be atomic. Either everything occurs, or nothing does. All changes made during a transaction revert to a previous state if it is interrupted in the midst, keeping the database consistent.

Consistency: When a transaction moves the database from one legal state to another, it is said to be consistent. The database constraints and integrity guidelines are respected throughout the transaction, guaranteeing the accuracy and consistency of the data. A transaction is rolled back, and the database is left untouched if it breaks any rules. The database is kept in a valid state after each transaction.

Isolation: Isolation refers to the ability of several transactions to run simultaneously without affecting one another. Only knowing about modifications made by other transactions once committed, each transaction operates as though it were the only one active. This attribute ensures that each transaction accesses consistent data not impacted by concurrently executing transactions.

Durability: When a transaction is durable, its modifications are kept after it is committed and will withstand subsequent system failures. In order to prevent loss in the case of a power outage or system failure, the modifications performed by the transaction are permanently recorded on and written to the disc. This characteristic ensures the database can recover from system faults without losing data.

2.1.3 Data Security

Data security is the collection of policies and procedures to guard against unauthorized access, theft, change, and data destruction. Data confidentiality, integrity, and availability are all critical components of information security.

Data security encompasses many different methods and steps, which include:

i)Access control: Access control entails granting authorized individuals or systems access to data. Access control lists, biometrics, encryption, and passwords can all be used to do this.

ii) Encryption: Data is transformed into a secret code through encryption to prevent unauthorized access. Both data at rest and data in transit may be protected with encryption.

2.1.4 Locking Mechanisms and Database Replication

A locking mechanism controls access to shared resources in a concurrent environment, such as in multi-threaded programs or distributed systems. When multiple threads or processes attempt to access a shared resource simultaneously, a locking mechanism can ensure that only one thread or process can access the resource at a time. This helps prevent race conditions, deadlocks, and other synchronization issues when multiple entities try to modify the same resource concurrently.

Database replication techniques and consistency models are critical components in the design of distributed systems, particularly in financial applications where data integrity and availability are paramount. Two primary consistency models are eventual consistency and strong consistency:

Eventual Consistency is often used in systems where high availability is prioritized over immediate consistency. It allows for temporary discrepancies between

replicas, with the guarantee that all copies will eventually become consistent. This model is beneficial for applications that can tolerate some degree of data staleness but require high availability and partition tolerance.

Strong Consistency, on the other hand, ensures that all replicas are consistent at all times. Any read operation following a write operation will always return the updated value. This model is crucial for financial systems where even minor inconsistencies can lead to significant issues, such as incorrect account balances or transaction failures.

2.1.5 Client-Server Architecture

Any software application, mobile app, web browser, or another device that can connect to a network and request services or data from a server qualifies as a client. Usually, the client connects to the server first and makes a request for a specific piece of data or service. The server, on the other hand, is a computer system that offers the client services or data. Clients submit queries to the server, and it answers with the relevant data. When a centralized system that can manage numerous client requests concurrently is required, the client-server architecture is frequently utilized. Through the use of a network and a communication protocol like TCP/IP, HTTP, or SMTP, the client, and server can communicate with one another under this architecture. Scalability, dependability, and security are advantages of client-server architecture.

2.2 Literature Review

The pros and cons of adopting a distributed database system in online banking are covered in [3]. Longer uptime, quicker performance, and reduced expenses are benefits. The complexity of duplicating data over several servers, assuring data security, and maintaining database integrity are some drawbacks. The paper suggests that these problems can be solved by implementing sophisticated security features and controls.

The paper [4] discusses the development of distributed applications, particularly in online banking, using Enterprise Java Technologies and its distributed architectures such as PageCentric, Page-View, and Model 1. However, the authors suggest that utilizing the Model View Controller 2 architecture would be more efficient in creating reusable, robust, and object-oriented applications. They emphasize that J2EE, a distributed component architecture, is suitable for developing web applications.

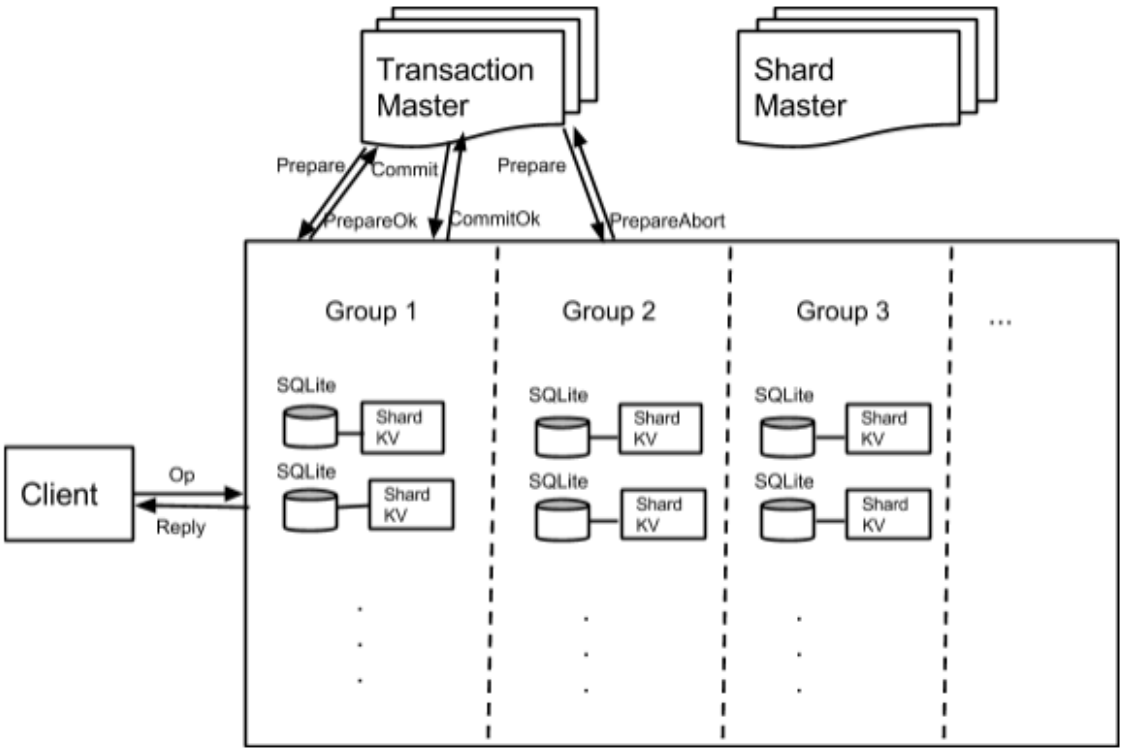


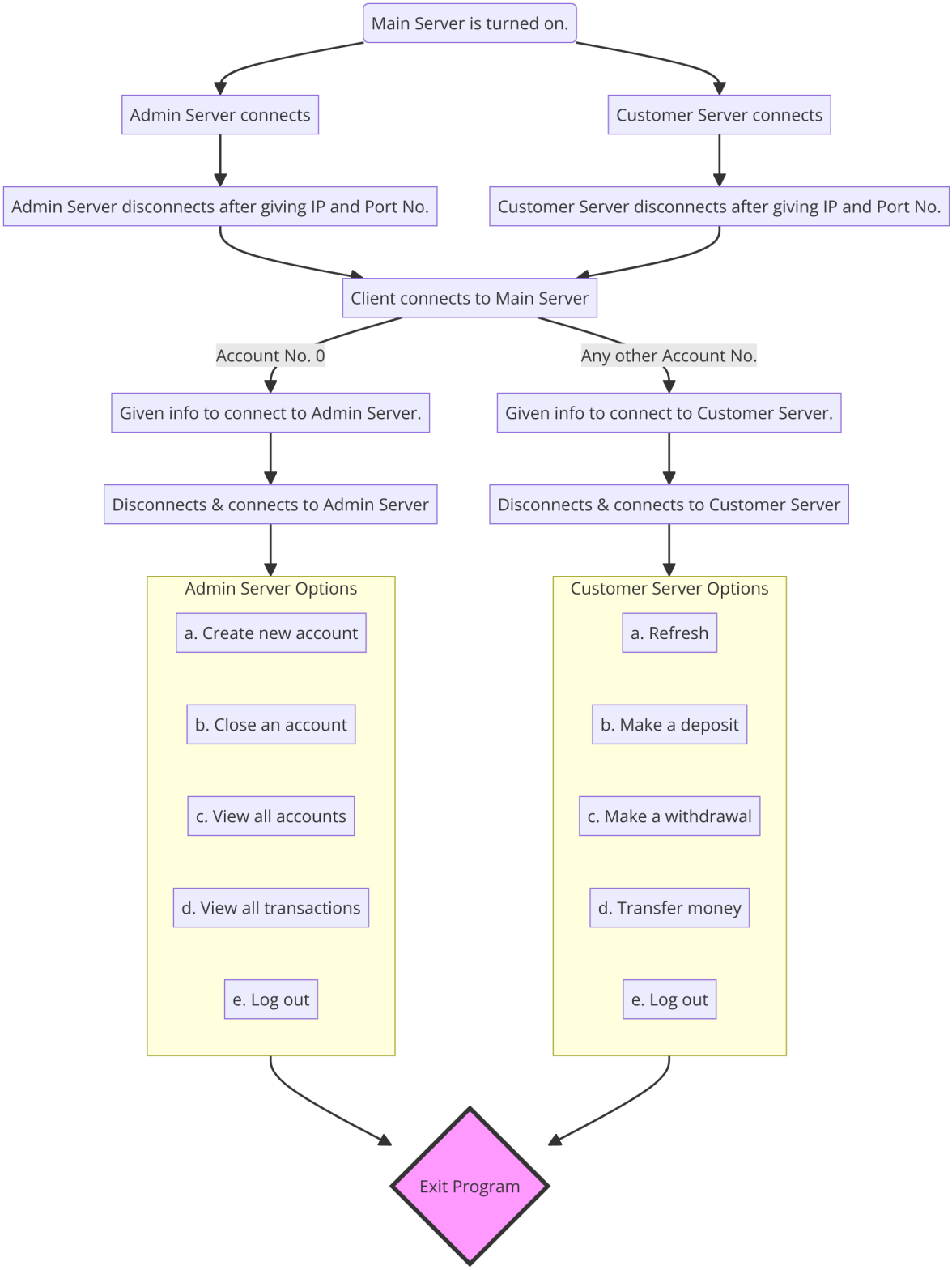
Fig 2.2 : The architecture described in Fig. 2. 2 [5]

A distributed system design for a banking system that strives to ensure consistency and dependability in data storage and access is described in [5]. The drawbacks of centralized banking systems are first discussed, along with the advantages of employing distributed systems. They then provide a brand-new distributed system design that uses a consensus mechanism to guarantee data consistency among several network nodes. The CDBS system is made to efficiently perform various banking tasks, including account setup, transaction processing, and balance checking.

The findings demonstrate that the CDBS system can preserve data consistency and integrity even in heavy transaction loads withstanding network failures and node crashes.

Chapter 3. Methodology

3.1 Outline



The Distributed Banking System employs an SQLite DBMS to facilitate the storage and retrieval of data on the backend. The front end, on the other hand, relies on Python for the output presented on the terminal. The system adheres to the Model-View-Controller (MVC) architectural pattern, wherein the user interface, data model, and controller are separated into distinct files. The view component not only displays the output but also captures input from the user. Subsequently, the controller executes specific actions on the database based on the captured input, and the model code handles the database operations. This separation of concerns provides a high degree of flexibility, as each file's code can be modified freely, as long as the interface remains the same.

For synchronization and ensuring data consistency across replicas for each of the server(main, admin and customer), a comprehensive strategy involves:
Conflict Resolution Mechanisms: Implementing logic to handle conflicts when concurrent updates occur. This could include "last writer wins" policies or more sophisticated merge strategies based on timestamps or transaction logs.
Versioning: Keeping track of data versions can help in identifying and resolving conflicts, ensuring that the most recent updates are preserved and replicated accurately.

3.2 Implementation

The data model employed in the Distributed Banking System is comprised of three tables: CUSTOMERS, TRANSACTIONS, and AUTH. The CUSTOMERS table contains information about each customer, such as their account number, Aadhar number, balance, first and last name, and preference for SMS or transaction notifications. Additionally, each user is assigned a password. To access their details, the user is required to provide their account number and password. However, before storing the password in the database, it is first hashed using SHA256 and saved in the AUTH table along with the username. The use of SHA256 encryption makes it considerably more challenging for unauthorized parties to brute force the password.

The TRANSACTION table contains data about each transaction, including the transaction number, sender account, receiver account, amount, and date. In cases where a deposit or withdrawal is made, the same account number is used for both the sender and receiver accounts. The transaction number in the TRANSACTIONS table and the account number in the CUSTOMERS table are both primary keys of the integer auto-increment type. This ensures that each record can be uniquely identified.

The login page of the Distributed Banking System provides access to two types of users: regular customers and admins once they have entered the correct username and password in the main server. If the user enters the admin account number and password, they are redirected to the admin page, where they can add and delete accounts and view the complete CUSTOMERS and TRANSACTIONS tables and is handled by the admin server. However, if the user enters the account number and password of a regular customer, they are directed to the customer menu instead and this is handled by the customer server.

On the customer menu which is handled by the customer server after being redirected by the admin server, the user can view their account number and current balance. Additionally, the menu provides options for depositing, withdrawing, and transferring funds, as well as viewing all transactions associated with the user. To ensure that the latest balance value is displayed, a refresh option is available, which fetches the most recent value of the balance in case the account is being accessed by multiple processes.

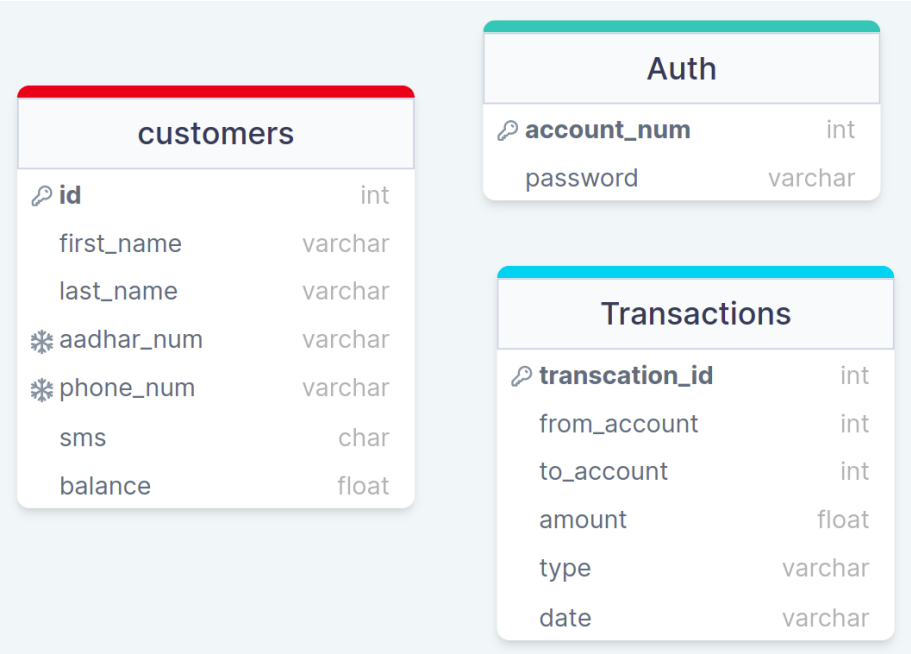
The Distributed Banking System employs socket programming to enable client-server communication. Concurrency is achieved through the use of Threading library functions. Upon initial connection to the server, the client sends an integer value, which is subsequently used as the encryption and decryption key for all messages exchanged between the server and the client. This ensures that even if a malicious user intercepts the packets, they cannot decrypt the messages without knowledge of the encryption key.

The menu is implemented using infinite while loops in the dbs_view.py file. The client can only display the menu it receives and send input. The server handles all decision-making processes, including redirecting users, validating input, and executing actions. Most actions return a list of values, where the first value is a boolean indicator of success, and the second value contains relevant data. The server may send messages to the client beginning with the symbol '@;', followed by the action to be performed (e.g., '@CLEAR' to clear the screen or '@PASS' to accept passwords).

If a user has opted to receive SMS notifications for each transaction, the system uses the Twilio library to send SMS messages to the user's associated phone number. The Twilio library code accesses the Twilio web API to send SMS messages.

Tabulate library is used to format the output of tables. This library provides several functions for displaying a 2D list in various styles. The database schema is shown in Fig. 3.2.1.

Figure 3.2.1: Database Model.



Chapter 4. Results and Discussion

This section presents an overview of the results obtained and shows the output.

4.1 Discussion

Database replication and synchronization strategies significantly impact system performance, particularly in terms of latency and throughput. Synchronous replication can increase latency due to the need for transactions to be confirmed across multiple nodes, potentially reducing throughput. Asynchronous replication, while improving throughput by allowing transactions to proceed without immediate replication, can lead to data being temporarily inconsistent across nodes.

These systems often prioritize consistency and availability, necessitating a careful balance to ensure transactions are processed reliably and accurately, even in the event of network partitions or server failures. Achieving this balance requires thoughtful design and implementation of replication and synchronization mechanisms to meet the stringent demands of financial systems.

FlexiBank Connect is a well-structured client-server application written in Python that demonstrates the use of sockets and threading for concurrent communication. It provides a simple command-line interface for users to manage bank accounts, including depositing and withdrawing funds and transferring funds to other accounts. The code follows standard Python conventions and makes use of various Python modules for efficient serialization/deserialization of objects. The project serves as a valuable resource for developers looking to learn about network programming in Python and provides a solid foundation for building more complex distributed systems. Figures 4.2.1 - 4.2.13 shows the code output.

4.2 Results and Output

Figure 4.2.1: Server Initiated

Figure 4.2.2: Login Menu

```
MIT BANK ADMIN MENU
MIT BANK,

 Bank of surity since 2024
 -------------------------
 Login Menu

 a. Login to your account
 b. Exit

 Enter your choice: a

 Enter your account number: 8
 Enter your password:
```

Figure 4.2.3: Admin Menu

```
MIT BANK CUSTOMER MENU
 ------------------------------
 Account number: 8
 Balance: 99897.0 Rs

 Options

 a. Refresh
 b. Make a deposit
 c. Make a withdrawal
 d. Transfer money
 e. See transaction history
 f. Log out

 Enter your choice:
```

Figure 4.2.4: Customer menu

```
Balance: 99897.0 Rs

Options

a. Refresh
b. Make a deposit
c. Make a withdrawal
d. Transfer money
e. See transaction history
f. Log out

Enter your choice: b

Enter amount to deposit: 1000

Deposit was successful. Press any key to continue...
```

Figure 4.2.5: Customer Deposit

TRANSACTIONS table

transaction_id	from_account	to_account	amount	type	date
11	8	8	100	DEPOSIT	2024-02-17 14:04:49
12	8	8	100	WITHDRAWAL	2024-02-17 14:05:56
13	8	9	100	TRANSFER	2024-02-17 14:10:29
14	8	9	3	TRANSFER	2024-02-17 14:10:40
15	8	8	1000	DEPOSIT	2024-02-21 17:04:56
16	8	8	100	WITHDRAWAL	2024-02-21 17:05:15
17	8	11	50000	TRANSFER	2024-02-21 17:06:52
18	8	11	5000	TRANSFER	2024-02-21 17:07:30
19	8	11			

Figure 4.2.11: Customer view transaction history

```
Options

a. Refresh
b. Make a deposit
c. Make a withdrawal
d. Transfer money
e. See transaction history
f. Log out

Enter your choice: d

Enter account number of receiver: 12

Enter amount to transfer: 5000

Transfer was successful. Press any key to continue...
```

Figure 4.2.12: Customer transfer money

Figure 4.2.13: SMS update to mobile

Chapter 5. Conclusions and Future Enhancements

5.1 Conclusions

FlexiBank Connect is a distributed system designed to manage the operations of a bank, and it highlights the significance of implementing distributed system concepts in complex and critical systems. The system is designed to be scalable, fault-tolerant, and efficient while ensuring the security and privacy of user data.

One of the critical features of FlexiBank Connect is its use of a client-server architecture. This allows for separating concerns between the user interface and the back-end processing, enabling efficient and secure communication between the two components. Additionally, FlexiBank Connect implements remote procedure calls (RPCs) to facilitate communication between the client and server, ensuring that requests and responses are transmitted quickly and reliably.
Security is of utmost importance in a banking system, and FlexiBank Connect considers this by incorporating encryption-based security mechanisms. All sensitive data, such as user account information and transaction details, are encrypted using robust encryption algorithms to prevent unauthorized access. Moreover, FlexiBank Connect implements access controls, authentication mechanisms, and other security features to ensure the integrity of user data.

Integrating database replication and synchronization in a distributed banking system: It offers significant benefits, including enhanced data availability, fault tolerance, and system resilience. However, it also introduces challenges such as increased complexity in managing data consistency and potential performance trade-offs.

Fault tolerance is another important aspect of distributed systems, and FlexiBank Connect is designed to be fault-tolerant.

To ensure that all transactions are processed correctly, FlexiBank Connect employs a robust consistency model. This model ensures that all transactions are executed in a consistent order and that all replicas of the data are updated in a timely and consistent manner. Using a robust consistency model prevents inconsistencies and data corruption, which can lead to errors and financial loss.
In conclusion, FlexiBank Connect is a distributed system that demonstrates the importance of implementing distributed system concepts for the use-case of a bank management system. Its client-server architecture, use of RPCs, encryption-based security mechanisms, fault-tolerance mechanisms, and robust consistency model all work together to ensure the system is efficient, secure, and reliable.

5.2 Future Enhancements

Some prospective enhancements in future versions of the project can include:

1) Incorporation of a GUI interface: While the current command-line interface is functional, implementing a GUI interface would enhance user experience, the same can be achieved by redesigning the client component of the project keeping Human-Computer Interaction principles in mind.
2) Increase system scalability: Many additional banking features including but not limited to loan application facilitation, credit card management system etc can be incorporated. Implementing support for multiple currencies and digital currency can make the system globally scalable.
3) Fraud detection system: Incorporate a fraud detection system to monitor suspicious transactions.
4) Support for recurring transactions: Incorporate recurring transactions in the processing to ease convenience
5) Implement support for batch transactions: Adding support for batch transactions, which allows users to perform multiple transactions at once, would save users time and make the system more efficient.
6) Support multi-factor authentication: Adding support for multi-factor authentication (MFA), which requires users to provide more than one authentication factor, would provide an additional layer of security to the system.
7)Integrating database replication and synchronization in a distributed banking system: Adopting newer database technologies that offer improved performance and built-in support for advanced replication and synchronization mechanisms can be beneficial. Further, exploring advanced conflict resolution strategies that can automatically handle data inconsistencies and employing machine learning algorithms to predict and mitigate potential synchronization issues could significantly improve data consistency and system reliability.

References

[1] Van Steen, Maarten, and Andrew S. Tanenbaum. *Distributed systems*. Leiden, The Netherlands: Maarten van Steen, 2017.
[2] Silberschatz, Abraham, Henry F. Korth, and Shashank Sudarshan. "Database system concepts." (2011).
[3] C.S., Arjun, and Midhun Omanakuttan. "A Review on Distributed Systems and Online Banking." n.d., Santhigiri College of Computer Science.
[4] Mahmood Akthat. “ Distributed Online Banking”
[5] Shirley Lu, Min Zhang, Chongyuan Xiang “Consistent Distributed Banking System (CDBS)”.CSAIL, MIT