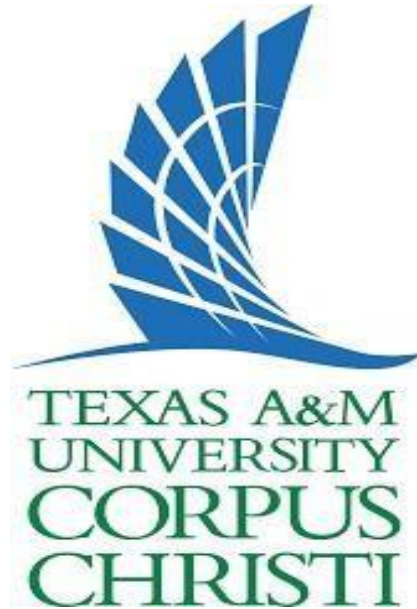


FINTECH FORTRESS A DISTRIBUTED BANKING SYSTEM

DEPARTMENT OF COMPUTER SCIENCE



COSC 6352 - ADVANCED OPERATING SYSTEMS

Instructor - Dr. Yongzhi Wang

Report by:

Yasaswi Polasi

ABSTRACT

Distributed Banking System is a secure, fault-tolerant financial system designed to enhance the reliability and security of modern banking services. This service combines zero-knowledge proof (ZKP) authentication, protecting sensitive information in an authentication process, it uses the Schnorr protocol to ensure strong and private user authentication. Also in context, it says real-time database synchronization to ensure continuous service availability and data fraud.

Key functions include account management, secure transaction processing (deposit, withdrawal, and transfer), access to customer history, and information services for notifying users of account activity and transactions. To be with improved reliability, the system has database replication and automatic failover mechanisms to provide resilience against disruptions. This project demonstrates how ZKP-based authentication, fault tolerance, and information processing have been successfully applied to create secure, reliable and user-friendly banking solutions.

Keywords: Distributed Banking System, Zero-Knowledge Proof (ZKP), Schnorr Protocol, Secure Authentication, Fault-Tolerant Architecture, Real-Time Database Synchronization, Database Replication, Automated Failover, Secure Transaction Processing, Account Management, Transaction History, Notification Services, Financial Security, Data Consistency, Resilient Banking Solutions.

1. INTRODUCTION

In today's digital age where financial transactions increasingly rely on technology, banking systems have become a fundamental need to ensure security, reliability and accessibility. Distributed banking meets these needs through security improved processing and error-free processes that result in a more robust and user-friendly financial system.

This project uses zero-knowledge proof (ZKP) authentication, uses the Schnorr protocol to protect sensitive user information during authentication, and ensures strong user authentication and it is confidential to use. With real-time database synchronization, the system ensures continuous service availability and safeguards against data fraud, providing maximum reliability in disruption even in unexpected situations.

Key features of the system include account management, secure transaction processing (such as deposits, withdrawals, and transfers), access to customer history, and information processing to create debtors keep the user informed of audit activity. Improved reliability is achieved through database replication and automatic failover mechanisms.

Distributed banking systems combine advanced authentication mechanisms, error-proof systems, and compliance functionality to demonstrate the ability to create secure, efficient and portable banking solutions reliable designed to meet the challenges of modern financial services.

2. SYSTEM ARCHITECTURE

Overview of Distributed System Design:

Distributed system architecture enables multiple independent components (server, client, database, etc.) to work together in a coordinated manner to provide a seamless service to the user. Ensures continuity work and protect critical data, the system is designed for high availability, scalability, security and fault tolerance. In this architecture, web-based applications communicate with users through different clients and are supported by redundant servers, synchronized databases, and failover mechanisms. These distributed systems use many critical resources to manage user requests, store data, and provide reliable services.

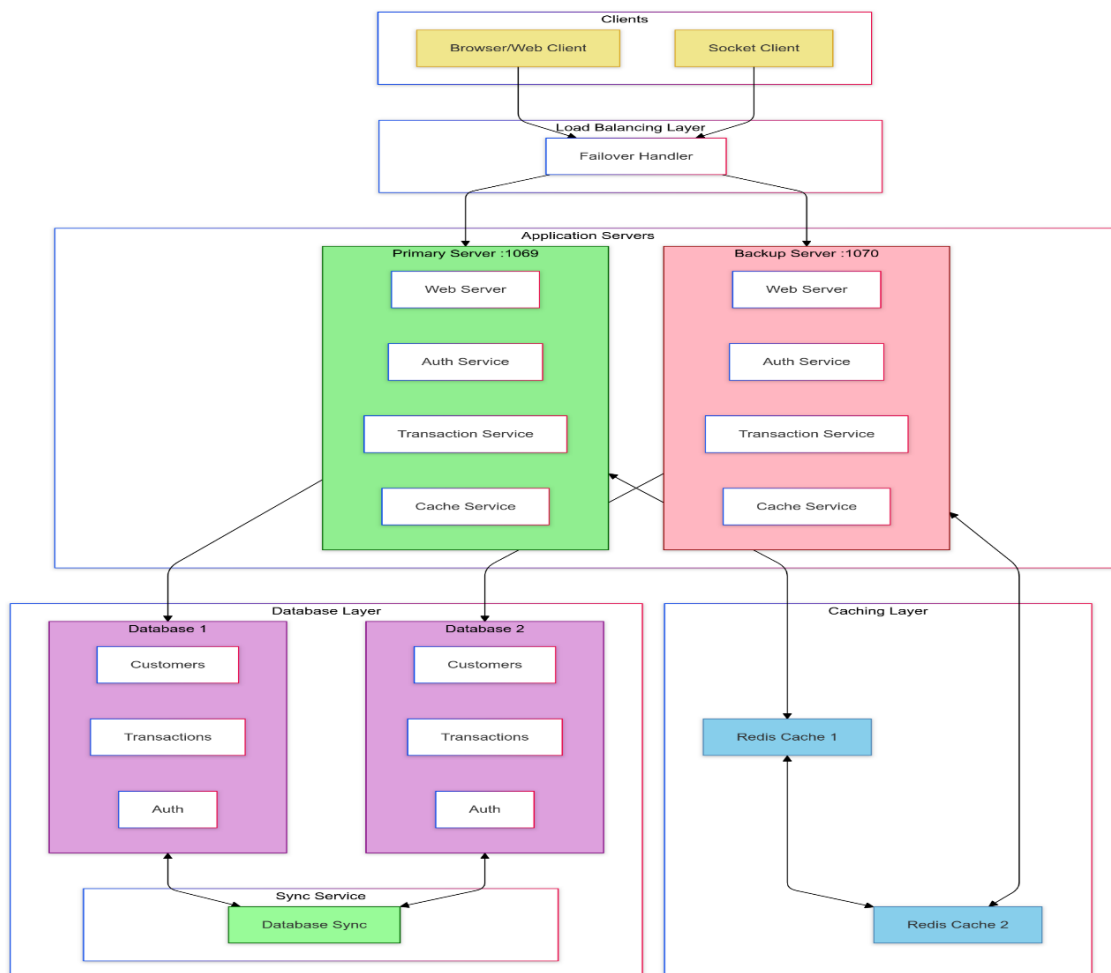


Fig-1: Design of the System

i. Dual-Server Architecture:

The distributed system in this design follows a dual-server model, where two application servers are used to ensure high availability and fault tolerance. This design ensures that if one server fails, another server can process the request without interrupting the service.

- **Primary Server (1069):** This is the primary server that handles most of the incoming client requests. It handles communication with clients, connects to databases, and communicates with the caching layer for efficient data recovery.
- **Backup Server (1070):** This server acts as a backup for the primary server. When a primary server fails or overruns, the backup server takes over to ensure uninterrupted service. This redundancy reduces downtime and increases system reliability.

This dual-server approach ensures that the system can maintain service availability even when the server is down.

ii. Zero-Knowledge Proof (ZKP) Authentication:

Zero-knowledge proof (ZKP) authentication is used to enhance security and protect sensitive data. In this process, it ensures that users can authenticate without revealing their real credentials such as passwords or private keys. This method uses cryptographic protocols, where one party (the swallower) can convince the other party (the broker) that they know the secret without revealing it.

- **Schnorr Protocol:** ZKP, implemented using protocols such as Schnorr Protocol, is used to effectively authenticate users while maintaining privacy. In a banking or e-commerce system, this ensures that sensitive user information (such as login credentials) will never be encountered by malicious people.

This level of security prevents unauthorized access and protects sensitive information, ensuring that only legitimate users can interact with the system.

iii. Database Synchronization and Replication:

To ensure high availability, robustness, and fault tolerance, the system uses database synchronization and replication. The system uses two databases to store customer and

transaction information, which are updated periodically to synchronize the data between them.

- Database 1: Stores data about primary users and transactions.
- Database 2: Stores a mirrored copy of Database 1's data.
- Synchronization: The synchronization service periodically ensures that any changes made in database 1 are reflected in database 2. This prevents inconsistent data between the two databases and ensures that both databases are updated.
- Replication: Replication ensures that the same data exists in both databases, improving data redundancy and fault tolerance. If one database fails, another can continue to provide the necessary data, ensuring system continuity.

Database replication and synchronization maintain data availability while protecting against potential data loss from server crashes or failures.

iv. Automated Failover Mechanisms:

The system was designed with automatic failover mechanisms to ensure service continuity in the event of a server or component failure. When the primary server becomes unavailable due to hardware failure, overload, or network issues, the system automatically forwards jobs to the backup server without requiring manual intervention

- Failure Procedure: If the primary server fails, the load balancer diverts all incoming traffic to the backup server (1070), which has the same application logic and data state as the primary server. This automatic switchover occurs without difficulty, and there is no service interruption for users.
- Load Balancer: A load balancer not only distributes requests across servers but also detects primary server failures and automatically redirects requests to backup servers. This ensures that the system can handle traffic spikes and recover quickly from component failures.

Automated failover mechanisms reduce downtime and system performance, ensuring fewer disruptions to users.

Key Interactions and Flow:

- **Clients:** Clients interact with the system through web browsers or direct socket connections. The load balancer receives incoming requests from clients and forwards them to the application servers.
- **Load Balancer:** The load balancer distributes client requests to either the primary or backup application server, depending on availability and load. If a failure is detected, it redirects traffic to the backup server.
- **Application Servers:** Application servers process the client requests, retrieve data from the databases, and interact with the cache layers. If the primary server fails, the backup server takes over the responsibilities.
- **Database Layer:** The databases store critical data, such as customer information and transactions, and are replicated to ensure consistency and availability.
- **Caching Layer:** Redis caches store frequently accessed data from the databases to reduce load and improve response times. This caching layer helps ensure fast data retrieval and minimizes database query times.
- **Sync Service:** The sync service ensures that both databases are synchronized, keeping data consistent across the system and mitigating potential data loss during failover or server transitions.

3. SYSTEM FEATURES

a. Accounting Management:

It allows users to create, update and manage their accounts more efficiently. This includes user profiles, account information, and ensuring data privacy.

b. Transaction Processing:

Facilitates secure and efficient handling of financial transactions, ensuring accuracy and integrity in all operations.

- **Deposits:**

Allows users to add funds to their accounts through various methods, ensuring the amount is reflected in real time for immediate availability.

- **Withdrawals:**

It allows users to securely withdraw funds from their accounts, validate the credentials before processing them, and check for balances.

- **Fund Transfers:**

It allows for easy transfers between accounts in the same bank or banks with secure authentication mechanisms.

- c. Transaction History:**

It gives users a detailed view of past transactions, including dates, amounts and details, to create transparent and easy record keeping.

- d. Real-Time Notifications:**

It immediately sends alerts to users about account activity such as deposits, withdrawals and transfers, ensuring that they are notified and that unauthorized activities can be detected quickly.

4. DISTRIBUTED FEATURES

- a. Coordination:**

Communication in a distributed system ensures seamless communication between all parties to efficiently meet customer requests. The load balancer plays a key role by distributing incoming requests across servers, optimizing resource utilization, and preventing overload. The synchronization service ensures data accuracy by synchronizing changes in updated databases and ensuring consistency of the stored information. In addition, dual-server operation increases system reliability by providing primary backup servers that enable the primary backup servers to work together, with the backup server immediately ready to take over in the event of a primary server failure. This approach this precise design ensures uninterrupted service and high efficiency.

- b. Consistency & Replication:**

Consistency in a distributed system ensures that all users have access to the same data, even when multiple objects connect to it at the same time. To achieve this, replication is used, and copies of the data are maintained in multiple databases to increase availability and reliability. The system uses data replication mirroring information between database 1 and database 2, ensuring redundancy. In addition, the synchronization service periodically

updates changes to these databases, preserving data integrity and assuring the accuracy of the updated data for a consistent experience.

c. Fault Tolerance:

The system is designed to effectively manage the failure of components, allowing for smooth operation. Dual-server architecture provides redundancy, whereby the backup server automatically takes over operations if the main server fails. In addition to this, automatic failover mechanisms in the load balancer detect real-time server failures and redirect requests to business components, reducing downtime and database redundancy by providing replication of data an availability guarantee, ensuring that even if a database is inaccessible, the system operates smoothly without loss of data or disruption of service.

d. Security:

The system places great emphasis on protecting sensitive user data through advanced security measures. “Zero-knowledge proof (ZKP) authentication”, which uses the Schnorr protocol, ensures secure user transactions without revealing sensitive credentials. “Data encryption” protects information during transmission and storage, preventing unauthorized access. Additionally, “Real-time notifications” allow users to see account activity, allowing them to quickly identify any unauthorized activity. Strict “Access control” policies further prevent unauthorized interactions with the system. By combining these features, the system provides a secure, reliable and flexible platform that ensures data integrity and ensures continuous availability.

5. DESIGN AND DEVELOPMENT FRAMEWORK

a. TECHNOLOGIES USED:

- Backend: Python, Flask
- Database: SQLite3
- Distributed Technologies: TCP/IP, ZKP, Multi-threading

b. SECURITY PROTOCOLS:

• **Schnorr Protocol for ZKP:**

The Schnorr Protocol has been implemented for user validation. It allows users to safely authenticate without revealing sensitive credentials and ensures that sensitive information (e.g. passwords) is never revealed during authentication. This process enhances, and crashes privacy protect against identity theft or compromise of credentials.

- **Database Encryption:**

All sensitive data stored in the system's databases is encrypted. This prevents access to user information, even if the database has been breached. Encryption is applied to sensitive data areas such as account information, transaction records and personal identification to maintain privacy.

- **Secure Sessions:**

Secure sessions are established for communication between clients and servers. These sessions use privacy protocols (e.g. HTTPS/TLS) to ensure that data transmitted during the transaction or accounting activity cannot be intercepted or tampered with. This is important for the security of user transactions, inter alia and login settings, connection requests, and notifications.

6. IMPLEMENTATION & RESULTS

Server 1:

```
(venv) chinmayi@DELL-PCN: /mnt/c/Users/chinm/aos/distributed-banking-system/config$ cd ..
(venv) chinmayi@DELL-PCN: /mnt/c/Users/chinm/aos/distributed-banking-system$ python3 server/server_manager.py 0
Initializing Redis cache...
Redis cache initialized
Loading menus...
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/adminMenu.txt
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/loginMenu.txt
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/customerMenu.txt
All menus loaded successfully
Menus loaded successfully
Initializing database...
Database initialized successfully
Backup database initialized successfully
Starting database synchronization...
Database paths: Primary=/mnt/c/Users/chinm/aos/distributed-banking-system/server/database1.db, Backup=/mnt/c/Users/chinm/aos/distributed-banking-system/server/database2.db
Clearing Redis cache...
Redis cache cleared successfully
Database synchronization started
Database synchronization active
Starting server 0...
Server 0 started on 127.0.0.1:1069
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
```

Fig-2: The above terminal displays Server 1

**** Note:** Server 1 is named as Server 0 in the snippets and Server 2 is named as Server 1. ******

This terminal output indicates the initialization and operation of the distributed banking system. The script `server_manager.py` is created in a Python environment, likely to check a server instance marked 0. Redis starts properly, meaning it is used for caching, while menu files like `adminMenu.txt`, `LoginMenu.txt`, and `customer Menu is txt` loaded, shows support for different user functions. The system installs the backup mechanism under the primary database

(database1.db) and enables database synchronization to ensure data consistency. The server starts at 127.0.0.1:1069, and updates are applied to cache tables such as TRANSACTIONS, CUSTOMERS, and AUTH during the synchronization cycle. These results reflect the robust architecture of the system with caching, synchronization and activity-based management of banking services.

Server 2:

```
pip install pycrypto
(venv) chinmayi@DELL-PCN:/mnt/c/Users/chinm/aos/distributed-banking-system$ python3 server/server_manager.py 1
Initializing Redis cache...
Redis cache initializedisfied: colorama in ./venv/lib/python3.12/site-packages (0.4.6)
Loading menus...DELL-PCN:/mnt/c/Users/chinm/aos/distributed-banking-system$
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/adminMenu.txt
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/loginMenu.txt
Loading menu from: /mnt/c/Users/chinm/aos/distributed-banking-system/server/menu/customerMenu.txt
All menus loaded successfully
Menus loaded successfully
Initializing database...
Database initialized successfully
Starting server 1...
Server 1 started on 127.0.0.1:1070
```

Fig-3: The above terminal displays Server 2

The “Server 2” terminal output indicates its origin in the distributed banking system. It initializes its Redis cache, loads the user-specific menu files, and synchronizes with the main database (database1.db). Server 2 updates the TRANSACTIONS, CUSTOMERS, and AUTH tables in its collection, ensuring data consistency by synchronizing with other servers. It works on unique IPs and ports such as `127.0.0.1:1070`, and checks for updated data on all instances during the synchronization cycle.

Client:

```
chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × + v
(venv) chinmayi@DELL-PCN:/mnt/c/Users/chinm/aos/distributed-banking-system/client$ python3 main.py|
```

Fig-4: The above terminal displays client

This means that the generated script is a client-side component of the distributed banking system project. The location of the script in the "Customers" subdirectory indicates that it is responsible for handling operations or logic in the customer-facing part of the banking process.

Server Connection:

```
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
New connection from ('127.0.0.1', 35006)
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
```

Fig-5: Server Connection

The log entries indicate periodic synchronization between the server and its database, ensuring data consistency across tables like CUSTOMERS, TRANSACTIONS, and AUTH. Additionally, they highlight a new connection from a local host (127.0.0.1) on port 35006, suggesting that a local application or service is interacting with the server. The repeated cache updates optimize performance by reducing direct database queries, likely part of a routine maintenance or real-time data update process.

Login Menu:

```
chinmayi@DELL-PCN: /mnt/  X  chinmayi@DELL-PCN: /mnt/  X  chinmayi@DELL-PCN: /mnt/  X  +  v
BANK,
FINTECH FORTRESS
-----
Login Menu

a. Login to your account
b. Exit

Enter your choice:
|
```

Fig-6: Login Menu

The user is interacting with an information-based, command-line banking system, which is likely the customer side of a larger distributed banking system. The title "Bank, Fintech Fortress" and login menu indicate that the app emphasizes financial technology and security. The menu offers two options: "Log in to your account" and "Log out".

The user logs into the administrator account with the assigned credentials: the account number "0" and the password "Administrator". Zero-knowledge proofs (ZKP) are used for authentication, to ensure that the password is not directly revealed and remains confidential in the process in which case the password used for authentication is "admin". ZKP improves security by allowing the server to verify user authentication without revealing sensitive information such as passwords.

Authentication:

```
Cache updated for table: Admin
Database synchronization cycle completed

=====
ZKP Authentication Process for Admin Login
=====

1. Generating ZKP proof
-----
• Commitment: 77986681504172860031116096307861683298690188266910494325699990346151092990482
• Challenge: 11896421290199352286329670665740718374500916869267672527927395265142669501341
• Response: 38957456274304347709571629543826001355799060644179678555210992829394389778889

2. Verifying with Admin's Public Key
-----
• Public Key: 87822928321045780135625680249974613439941446352992507191949337239695297913081

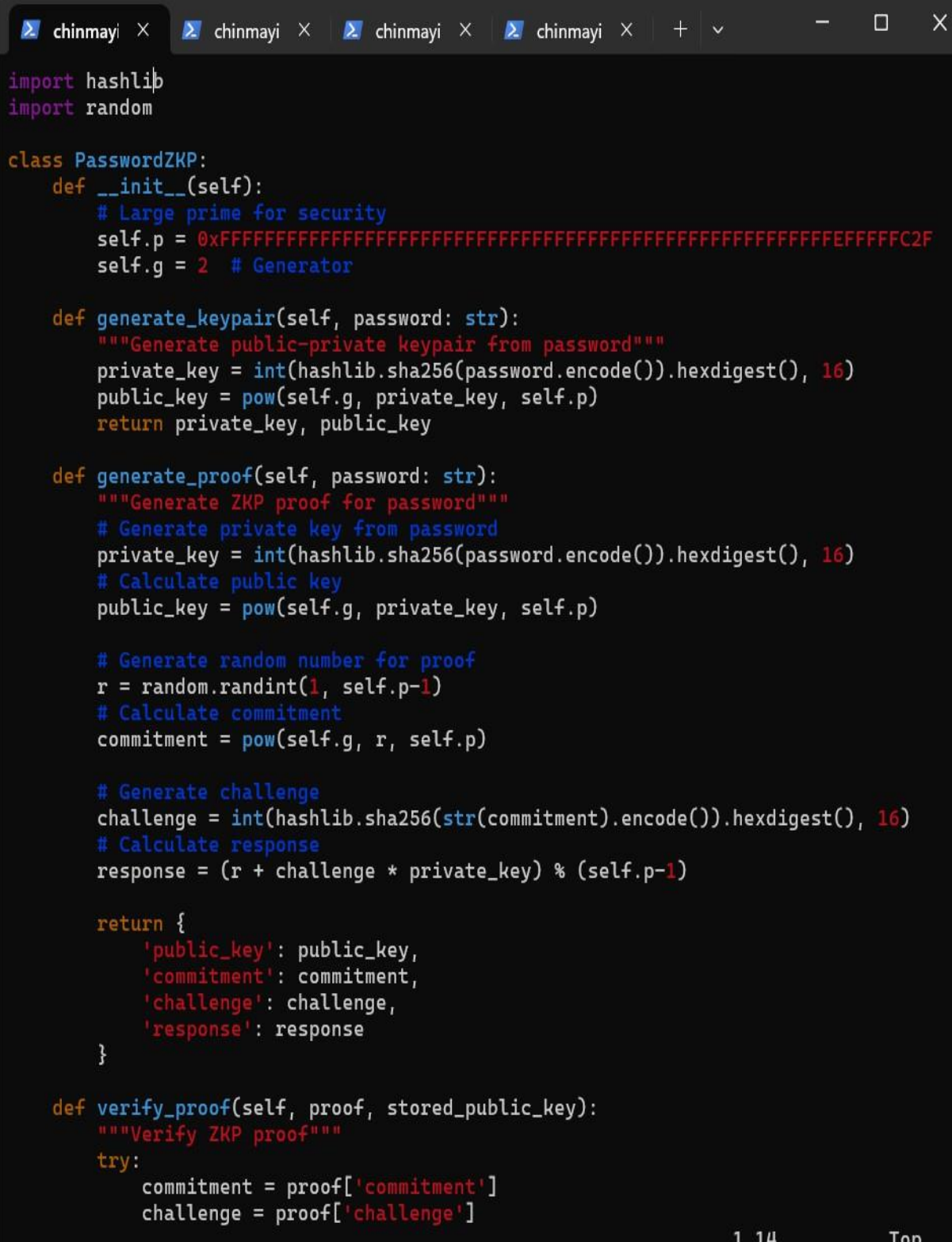
3. Verification Process
-----
• Status: SUCCESS - Valid admin authentication

=====
```

Fig-7: Authentication was done

We can see that the authentication is done as the Status shows SUCCESS – Valid admin authentication.

To check for class PasswordZKP:

The image shows a code editor window with four tabs, each labeled 'chinmayi'. The code is written in Python and defines a class named 'PasswordZKP'. The class has several methods: '__init__' for initialization with a large prime 'p' and generator 'g'; 'generate_keypair' for generating a public-private key pair from a password; 'generate_proof' for generating a ZKP proof from a password; and 'verify_proof' for verifying a ZKP proof. The code uses the hashlib library for SHA-256 hashing and the pow function for modular exponentiation. The editor interface includes standard window controls at the top and a status bar at the bottom right showing '1 1/4' and 'Top'.

```
import hashlib
import random

class PasswordZKP:
    def __init__(self):
        # Large prime for security
        self.p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC2F
        self.g = 2 # Generator

    def generate_keypair(self, password: str):
        """Generate public-private keypair from password"""
        private_key = int(hashlib.sha256(password.encode()).hexdigest(), 16)
        public_key = pow(self.g, private_key, self.p)
        return private_key, public_key

    def generate_proof(self, password: str):
        """Generate ZKP proof for password"""
        # Generate private key from password
        private_key = int(hashlib.sha256(password.encode()).hexdigest(), 16)
        # Calculate public key
        public_key = pow(self.g, private_key, self.p)

        # Generate random number for proof
        r = random.randint(1, self.p-1)
        # Calculate commitment
        commitment = pow(self.g, r, self.p)

        # Generate challenge
        challenge = int(hashlib.sha256(str(commitment).encode()).hexdigest(), 16)
        # Calculate response
        response = (r + challenge * private_key) % (self.p-1)

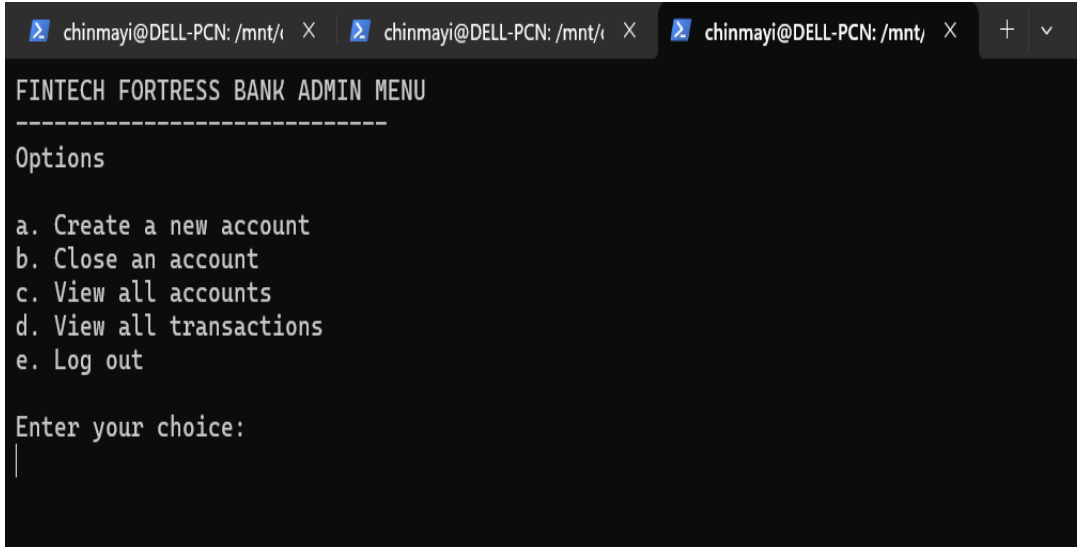
        return {
            'public_key': public_key,
            'commitment': commitment,
            'challenge': challenge,
            'response': response
        }

    def verify_proof(self, proof, stored_public_key):
        """Verify ZKP proof"""
        try:
            commitment = proof['commitment']
            challenge = proof['challenge']
```

Fig-8: Code Snippet to generate ZKP proof for password.

Admin Menu:

The below options are displayed after the user logs in as an admin.



```
chinmayi@DELL-PCN: /mnt/ι ×  chinmayi@DELL-PCN: /mnt/ι ×  chinmayi@DELL-PCN: /mnt/ι ×  +  v
FINTECH FORTRESS BANK ADMIN MENU
-----
Options

a. Create a new account
b. Close an account
c. View all accounts
d. View all transactions
e. Log out

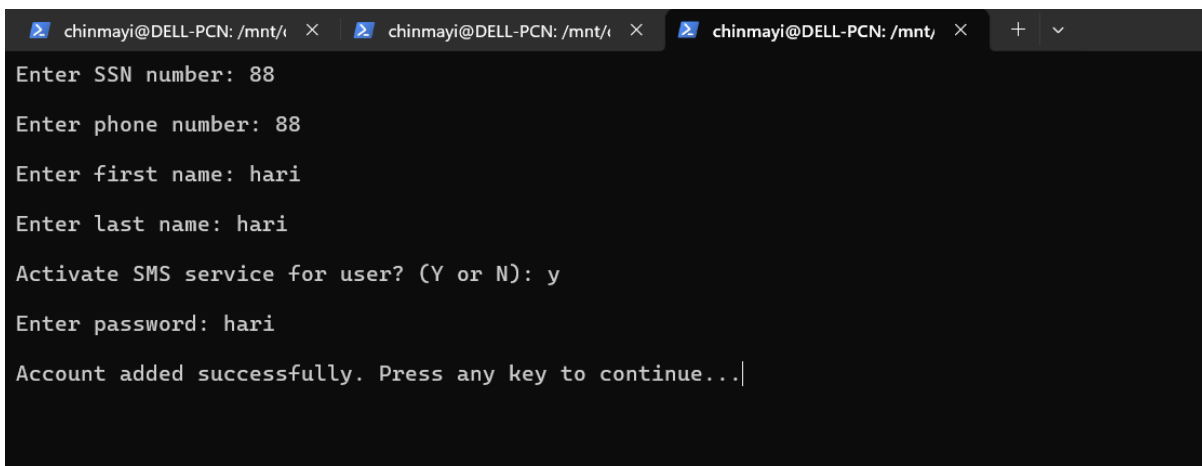
Enter your choice:
|
```

Fig-9: Admin account menu

From the above options, we can choose any one option according to our necessity.

For now, let us choose Option a, i.e. Create a new account.

Create a new account:

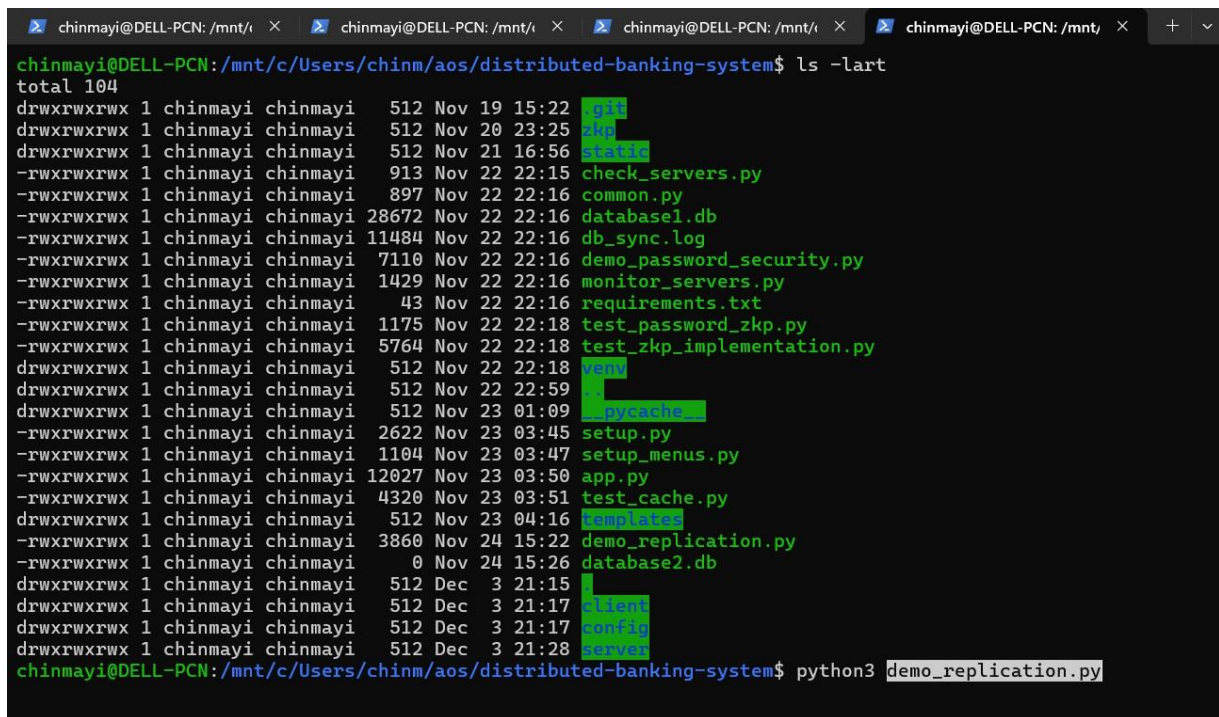


```
chinmayi@DELL-PCN: /mnt/ι ×  chinmayi@DELL-PCN: /mnt/ι ×  chinmayi@DELL-PCN: /mnt/ι ×  +  v
Enter SSN number: 88
Enter phone number: 88
Enter first name: hari
Enter last name: hari
Activate SMS service for user? (Y or N): y
Enter password: hari
Account added successfully. Press any key to continue...|
```

Fig-10: Details asked while creating a new account.

When we create a new account, it is going to ask our details like SSN number, Phone Number, first name, last name, Option to activate SMS service and password.

Check Databases:



```
chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × + v
chinmayi@DELL-PCN: /mnt/c/Users/chinm/aos/distributed-banking-system$ ls -lart
total 104
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 19 15:22 .git
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 20 23:25 zkp
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 21 16:56 static
-rwxrwxrwx 1 chinmayi chinmayi 913 Nov 22 22:15 check_servers.py
-rwxrwxrwx 1 chinmayi chinmayi 897 Nov 22 22:16 common.py
-rwxrwxrwx 1 chinmayi chinmayi 28672 Nov 22 22:16 database1.db
-rwxrwxrwx 1 chinmayi chinmayi 11484 Nov 22 22:16 db_sync.log
-rwxrwxrwx 1 chinmayi chinmayi 7110 Nov 22 22:16 demo_password_security.py
-rwxrwxrwx 1 chinmayi chinmayi 1429 Nov 22 22:16 monitor_servers.py
-rwxrwxrwx 1 chinmayi chinmayi 43 Nov 22 22:16 requirements.txt
-rwxrwxrwx 1 chinmayi chinmayi 1175 Nov 22 22:18 test_password_zkp.py
-rwxrwxrwx 1 chinmayi chinmayi 5764 Nov 22 22:18 test_zkp_implementation.py
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 22 22:18 venv
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 22 22:59 ...
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 23 01:09 ...pycache...
-rwxrwxrwx 1 chinmayi chinmayi 2622 Nov 23 03:45 setup.py
-rwxrwxrwx 1 chinmayi chinmayi 1104 Nov 23 03:47 setup_menus.py
-rwxrwxrwx 1 chinmayi chinmayi 12027 Nov 23 03:50 app.py
-rwxrwxrwx 1 chinmayi chinmayi 4320 Nov 23 03:51 test_cache.py
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 23 04:16 templates
-rwxrwxrwx 1 chinmayi chinmayi 3860 Nov 24 15:22 demo_replication.py
-rwxrwxrwx 1 chinmayi chinmayi 0 Nov 24 15:26 database2.db
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:15 ...
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:17 client
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:17 config
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:28 server
chinmayi@DELL-PCN: /mnt/c/Users/chinm/aos/distributed-banking-system$ python3 demo_replication.py
```

Fig-11: Checking for databases

From the above screenshot, we can see the two available databases i.e. database1.db and database2.db.

To Check Database Entries:

For checking the Database entries, there is a separate file named “demo_replication.py”.



```
chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × + v
=== Database Replication Demonstration ===
1. Show contents of both databases
2. Create new account (through Server 1)
3. Make a transaction (through Server 1)
4. Stop Server 1 (simulate failure)
5. Verify Backup Server (Server 2)
6. Exit
Enter your choice (1-6): |
```

Fig-12: Terminal output to check database entries

Suppose if we choose 1. Show content of both databases, then it is going to display the following details as shown below.


```

chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x
1. Show contents of both databases
2. Create new account (through Server 1)
3. Make a transaction (through Server 1)
4. Stop Server 1 (simulate failure)
5. Verify Backup Server (Server 2)
6. Exit
Enter your choice (1-6): 1
----- Primary Database (Server 1) -----
CUSTOMERS TABLE:
Account_Num | First_Name | Last_Name | SSN | Phone | SMS | Balance
-----
1 | yashu | yashu | 11 | 1 | N | 100000.0
2 | chinmayi | sony | 123 | 123 | N | 100500.0
3 | harishma | valli | 99 | 99 | N | 99500.0
4 | hari | hari | 88 | 88 | Y | 100000.0
TRANSACTIONS TABLE:
Trans_ID | From_Acc | To_Acc | Amount | Type | Date
-----
1 | 3 | 2 | 500.0 | TRANSFER | 2024-11-25 17:55:53
----- Backup Database (Server 2) -----
CUSTOMERS TABLE:
Account_Num | First_Name | Last_Name | SSN | Phone | SMS | Balance
-----
1 | yashu | yashu | 11 | 1 | N | 100000.0
2 | chinmayi | sony | 123 | 123 | N | 100500.0
3 | harishma | valli | 99 | 99 | N | 99500.0
4 | hari | hari | 88 | 88 | Y | 100000.0
TRANSACTIONS TABLE:
Trans_ID | From_Acc | To_Acc | Amount | Type | Date
-----
1 | 3 | 2 | 500.0 | TRANSFER | 2024-11-25 17:55:53
Press Enter to continue...|

```

Fig-13: Data replication

Here, we can see that data entered in server 1 but replicated in both databases. It ensures data replication and consistency.

Code Snippet Data Replication (Database Synchronization):

```

chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x  chinmayi@DELL-PCN: /mnt/i  x
class DatabaseSync:
    def __init__(self):
        self.running = True
        self.sync_interval = 2 # 2 seconds interval
        self.primary_db = os.path.join(current_dir, "database1.db")
        self.backup_db = os.path.join(current_dir, "database2.db")
        print(f"Database paths: Primary={self.primary_db}, Backup={self.backup_db}")
        # Initialize Redis cache
        self.cache = CacheManager()

    def sync_databases(self):
        while self.running:
            try:
                # Connect to both databases
                primary_conn = sqlite3.connect(self.primary_db)
                backup_conn = sqlite3.connect(self.backup_db)

                # Tables to sync
                tables = ['CUSTOMERS', 'TRANSACTIONS', 'AUTH']

                for table in tables:
                    self._sync_table(primary_conn, backup_conn, table)
                    # After syncing, update cache
                    self._update_cache(table, primary_conn)

                primary_conn.close()
                backup_conn.close()
                print("Database synchronization cycle completed")

            except Exception as e:
                print(f"Synchronization error: {e}")

            time.sleep(self.sync_interval)

    def _sync_table(self, primary_conn, backup_conn, table):
        cursor = primary_conn.cursor()
        cursor.execute(f"SELECT * FROM {table}")
        records = cursor.fetchall()

        # Cache the entire table
        cache_key = f"table_{table}"

```

Fig-14: code snippet for Data replication & Synchronization

```
chinmayi@DELL-PCN: /mnt/ X chinmayi@DELL-PCN: /mnt/ X chinmayi@DELL-PCN: /mnt/ X chinmayi@DELL-PCN: /mnt/ X + v
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
Cache updated for table: CUSTOMERS
Cache updated for table: TRANSACTIONS
Cache updated for table: AUTH
Database synchronization cycle completed
^C
Server shutting down...
Database synchronization stopped
^CException ignored in: <module 'threading' from '/usr/lib/python3.12/threading.py'>
Traceback (most recent call last):
  File "/usr/lib/python3.12/threading.py", line 1622, in _shutdown
    lock.acquire()
KeyboardInterrupt:
(venv) chinmayi@DELL-PCN:/mnt/c/Users/chinm/aos/distributed-banking-system$ |
```

Fig-15: Data Synchronization

Here, Client automatically connected to server 2 and for Fault Tolerance we need to kill server 1.

Fault Tolerance and Dual Server Architecture:

```
chinnmayi x chinnmayi x chinnmayi x chinnmayi x + - □ ×

SERVERS = [
    {'ip': '127.0.0.1', 'port': 1069}, # Primary server
    {'ip': '127.0.0.1', 'port': 1070} # Backup server
]

class BankClient:
    def __init__(self):
        self.current_server_index = 0
        self.client_socket = None
        self.key = random.randint(0, 255)

    def clearScreen(self):
        os_name = platform.system()
        if os_name == 'Windows':
            os.system('cls')
        else:
            os.system('clear')

    def connect_to_server(self):
        # Try all servers starting from the current index
        start_index = self.current_server_index

        for i in range(len(SERVERS)):
            server_index = (start_index + i) % len(SERVERS)
            server = SERVERS[server_index]

            try:
                if self.client_socket:
                    self.client_socket.close()

                self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

                print(f"Trying to connect to server at {server['ip']}:{server['port']}...")
                self.client_socket.connect((server['ip'], server['port']))
                print("Connection was successful!")
                self.current_server_index = server_index
                return True
            except ConnectionRefusedError:
                print(f"Server at port {server['port']} is not available, trying next server...")
                continue
        except Exception as e:
            print(f"Exception: {e}")
```

Fig-16: Code Snippet for Fault Tolerance

```
chinnmayi@DELL-PCN: /mnt/ x chinnmayi@DELL-PCN: /mnt/ x chinnmayi@DELL-PCN: /mnt/ x chinnmayi@DELL-PCN: /mnt/ x + - □ ×

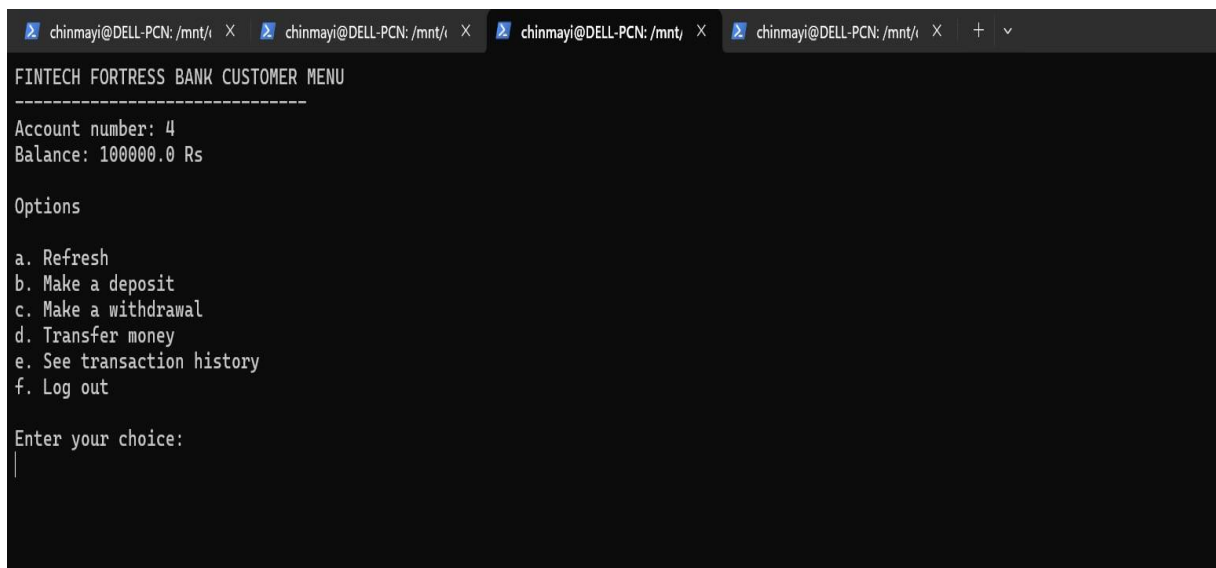
Reading state information... Done
redis-server is already the newest version (5:7.0.15-1build2).
0 upgraded, 0 newly installed, 0 to remove and 47 not upgraded.
Warning: The unit file, source configuration file or drop-ins of redis-server.service changed on disk. Run 'systemctl daemon-reload' to reload unit files.
Requirement already satisfied: flask in ./venv/lib/python3.12/site-packages (3.1.0)
Requirement already satisfied: flask-login in ./venv/lib/python3.12/site-packages (0.6.3)
Requirement already satisfied: flask-session in ./venv/lib/python3.12/site-packages (0.8.0)
Requirement already satisfied: Werkzeug>=3.1 in ./venv/lib/python3.12/site-packages (from flask) (3.1.3)
Requirement already satisfied: Jinja2>=3.1.2 in ./venv/lib/python3.12/site-packages (from flask) (3.1.4)
Requirement already satisfied: itsdangerous>=2.2 in ./venv/lib/python3.12/site-packages (from flask) (2.2.0)
Requirement already satisfied: click>=8.1.3 in ./venv/lib/python3.12/site-packages (from flask) (8.1.7)
Requirement already satisfied: blinker>=1.9 in ./venv/lib/python3.12/site-packages (from flask) (1.9.0)
Requirement already satisfied: msgspec>=0.18.6 in ./venv/lib/python3.12/site-packages (from flask-session) (0.18.6)
Requirement already satisfied: cachelib in ./venv/lib/python3.12/site-packages (from flask-session) (0.13.0)
Requirement already satisfied: MarkupSafe>=2.0 in ./venv/lib/python3.12/site-packages (from Jinja2>=3.1.2->flask) (3.0.2)
OK

^C
(venv) chinnmayi@DELL-PCN:/mnt/c/Users/chinnmayi/aos/distributed-banking-system$
^C
(venv) chinnmayi@DELL-PCN:/mnt/c/Users/chinnmayi/aos/distributed-banking-system$
pip install pycrypto
(venv) chinnmayi@DELL-PCN:/mnt/c/Users/chinnmayi/aos/distributed-banking-system$ python3 server/server_manager.py 1
Initializing Redis cache...
Redis cache initialized: colorama in ./venv/lib/python3.12/site-packages (0.4.6)
Loading menus...DELL-PCN:/mnt/c/Users/chinnmayi/aos/distributed-banking-system$
Loading menu from: /mnt/c/Users/chinnmayi/aos/distributed-banking-system/server/menu/adminMenu.txt
Loading menu from: /mnt/c/Users/chinnmayi/aos/distributed-banking-system/server/menu/loginMenu.txt
Loading menu from: /mnt/c/Users/chinnmayi/aos/distributed-banking-system/server/menu/customerMenu.txt
All menus loaded successfully
Menus loaded successfully
Initializing database...
Database initialized successfully
Starting server 1...
Server 1 started on 127.0.0.1:1070
```

Fig-17: Shows Fault Tolerance

Here, we can see that the client is connected to server 2 and satisfies fault tolerance.

Customer Menu:



```
chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × + v
FINTECH FORTRESS BANK CUSTOMER MENU
-----
Account number: 4
Balance: 100000.0 Rs

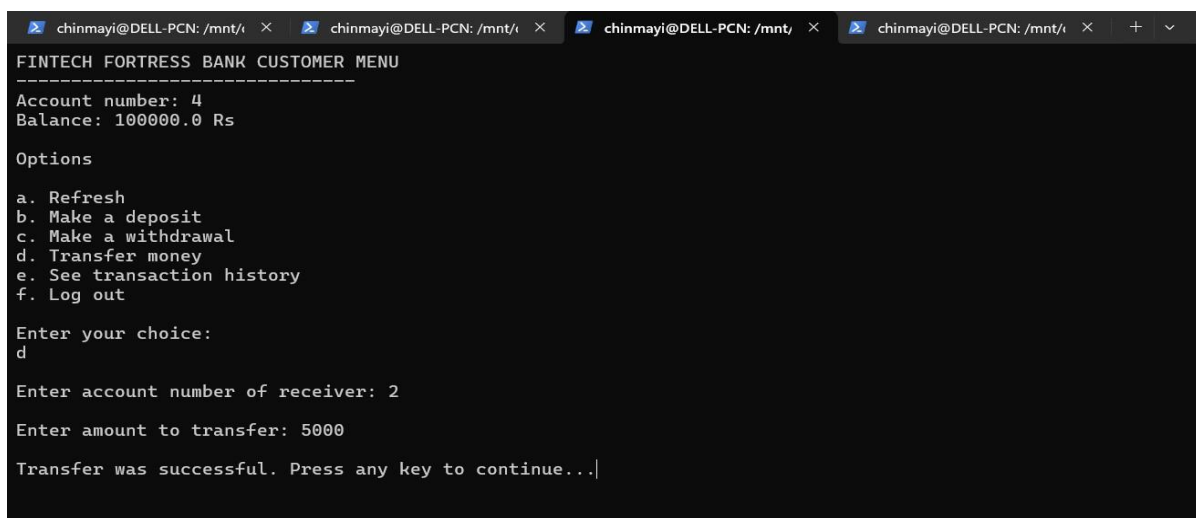
Options

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

Enter your choice:
|
```

Fig-18: The output terminal shows the Customer Menu.

Let's say, we want to transfer money. So, we need to hit option d.



```
chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × chinmayi@DELL-PCN: /mnt/ι × + v
FINTECH FORTRESS BANK CUSTOMER MENU
-----
Account number: 4
Balance: 100000.0 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:
d

Enter account number of receiver: 2

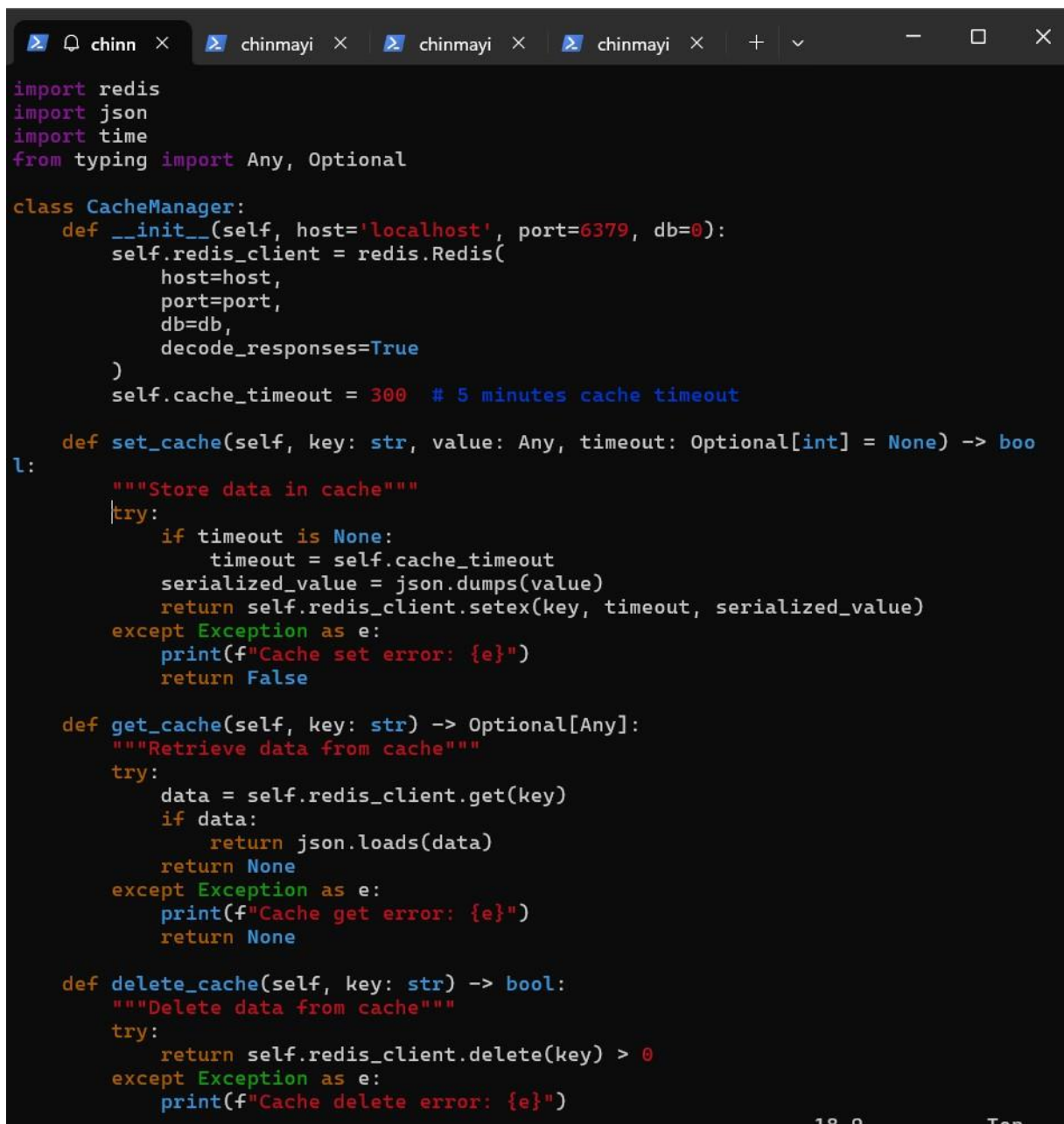
Enter amount to transfer: 5000

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

Fig-19: The output terminal shows asking the recipient details

Cache Manager:

To check for class Cache Manager, the code is saved as cache_manager.py.



```

import redis
import json
import time
from typing import Any, Optional

class CacheManager:
    def __init__(self, host='localhost', port=6379, db=0):
        self.redis_client = redis.Redis(
            host=host,
            port=port,
            db=db,
            decode_responses=True
        )
        self.cache_timeout = 300 # 5 minutes cache timeout

    def set_cache(self, key: str, value: Any, timeout: Optional[int] = None) -> bool:
        """Store data in cache"""
        try:
            if timeout is None:
                timeout = self.cache_timeout
            serialized_value = json.dumps(value)
            return self.redis_client.setex(key, timeout, serialized_value)
        except Exception as e:
            print(f"Cache set error: {e}")
            return False

    def get_cache(self, key: str) -> Optional[Any]:
        """Retrieve data from cache"""
        try:
            data = self.redis_client.get(key)
            if data:
                return json.loads(data)
            return None
        except Exception as e:
            print(f"Cache get error: {e}")
            return None

    def delete_cache(self, key: str) -> bool:
        """Delete data from cache"""
        try:
            return self.redis_client.delete(key) > 0
        except Exception as e:
            print(f"Cache delete error: {e}")

```

Fig-20: Code snippet for cache manager

Example flow:

1. Client requests data:

Client -> Server -> Check Redis Cache

2. If data in cache (Cache Hit):

Server <- Redis Cache (Fast)

3. If data not in cache (Cache Miss):

Server -> Database (Slow)

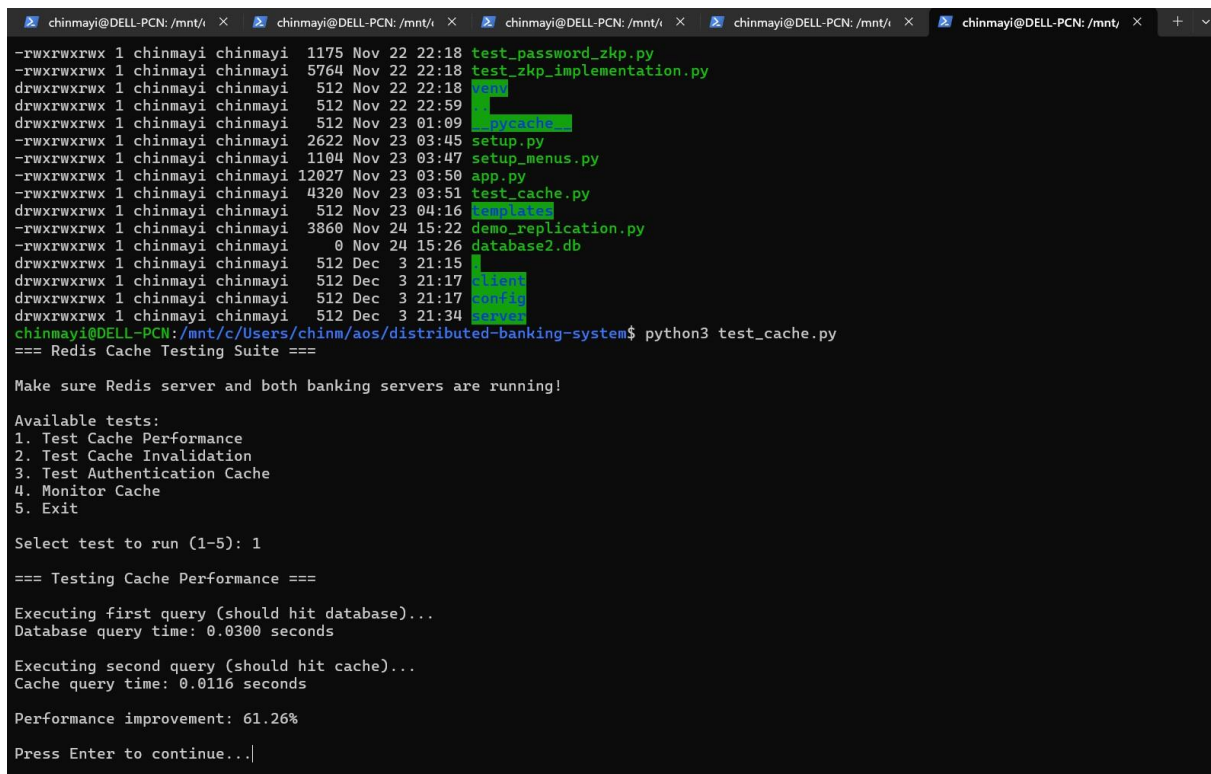
Server -> Store in Redis

Server -> Return to Client

4. When data changes:

Server -> Update Database

Server -> Clear Related Cache



```
chinmayi@DELL-PCN: /mnt/i x chinmayi@DELL-PCN: /mnt/i x chinmayi@DELL-PCN: /mnt/i x chinmayi@DELL-PCN: /mnt/i x chinmayi@DELL-PCN: /mnt/i x
-rwxrwxrwx 1 chinmayi chinmayi 1175 Nov 22 22:18 test_password_zkp.py
-rwxrwxrwx 1 chinmayi chinmayi 5764 Nov 22 22:18 test_zkp_implementation.py
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 22 22:18 view
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 22 22:59 redis
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 23 01:09 redis-cache
-rwxrwxrwx 1 chinmayi chinmayi 2622 Nov 23 03:45 setup.py
-rwxrwxrwx 1 chinmayi chinmayi 1104 Nov 23 03:47 setup_menus.py
-rwxrwxrwx 1 chinmayi chinmayi 12027 Nov 23 03:50 app.py
-rwxrwxrwx 1 chinmayi chinmayi 4320 Nov 23 03:51 test_cache.py
drwxrwxrwx 1 chinmayi chinmayi 512 Nov 23 04:16 templates
-rwxrwxrwx 1 chinmayi chinmayi 3860 Nov 24 15:22 demo_replication.py
-rwxrwxrwx 1 chinmayi chinmayi 0 Nov 24 15:26 database2.db
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:15 client
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:17 client
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:17 config
drwxrwxrwx 1 chinmayi chinmayi 512 Dec 3 21:34 server
chinmayi@DELL-PCN:/mnt/c/Users/chinm/aos/distributed-banking-system$ python3 test_cache.py
=== Redis Cache Testing Suite ===

Make sure Redis server and both banking servers are running!

Available tests:
1. Test Cache Performance
2. Test Cache Invalidation
3. Test Authentication Cache
4. Monitor Cache
5. Exit

Select test to run (1-5): 1

=== Testing Cache Performance ===

Executing first query (should hit database)...
Database query time: 0.0300 seconds

Executing second query (should hit cache)...
Cache query time: 0.0116 seconds

Performance improvement: 61.26%

Press Enter to continue...|
```

Fig -21: Cache Testing

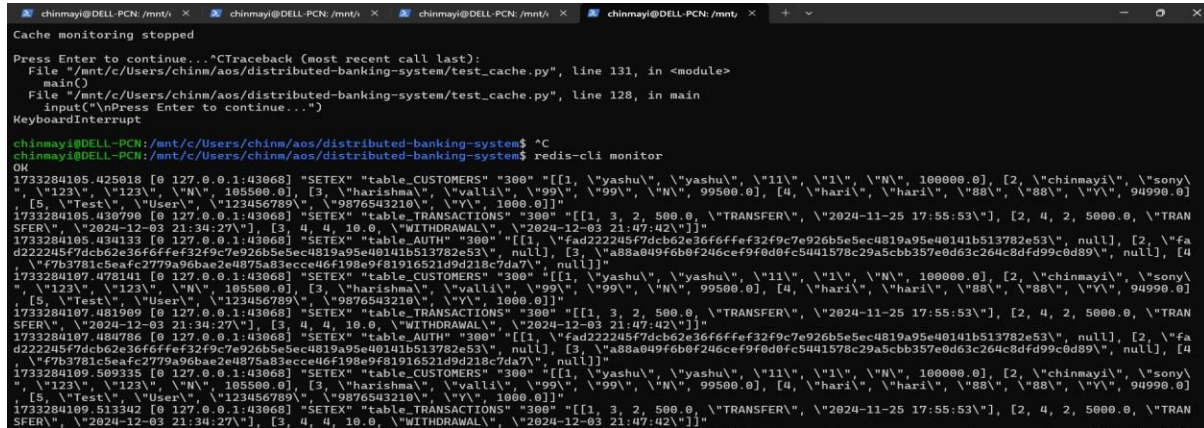
For Cache testing, we need to ensure Redis Servers and Banking servers are running. Here we chose to test cache performance.

Faster data retrieval (57-67% improvement), Reduced database load, Quicker authentication are benefits of performance testing.

There is separate script to test cache test_cache.py.

Data caching is used to conserve time and make banking system more efficient and faster.

Even the command, (redis-cli monitor) can also be used for cache testing.



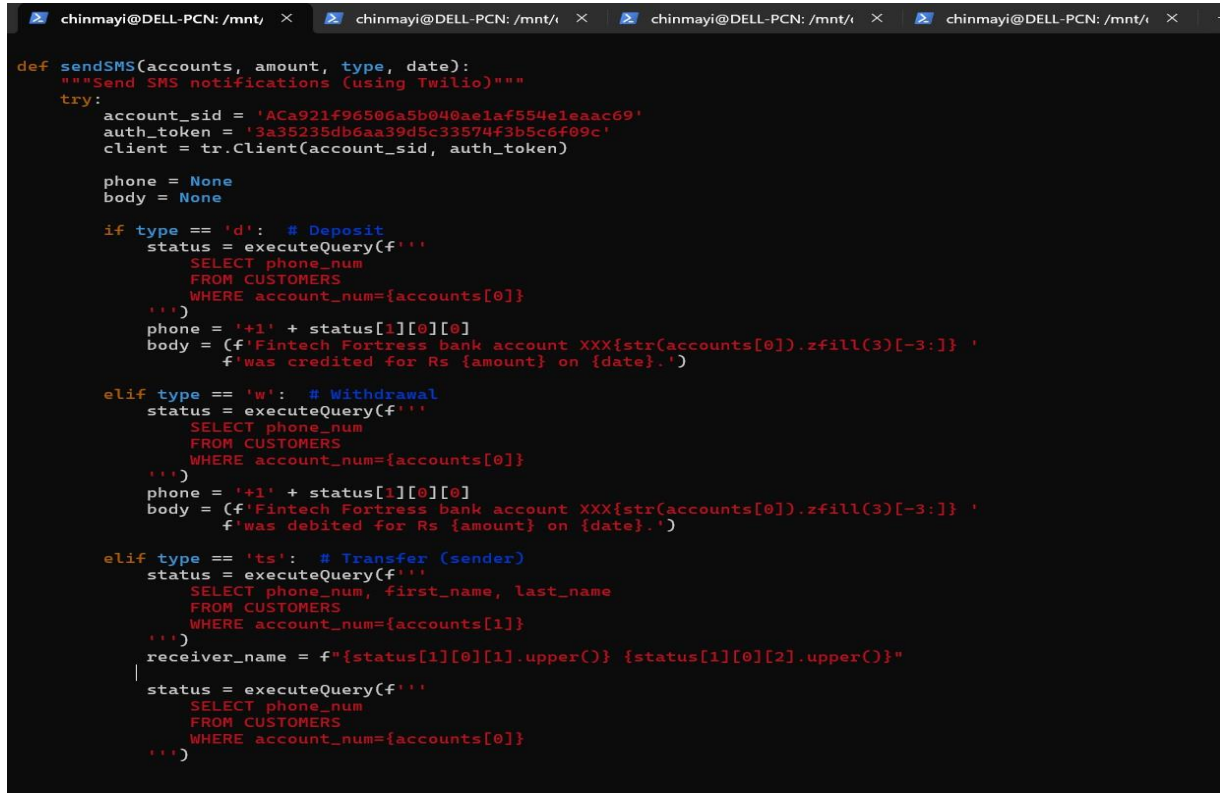
```
chinmayi@DELL-PCN: /mnt/
Cache monitoring stopped
Press Enter to continue...C:\Traceback (most recent call last):
  File "/mnt/c/Users/chinn/aos/distributed-banking-system/test_cache.py", line 131, in <module>
    main()
  File "/mnt/c/Users/chinn/aos/distributed-banking-system/test_cache.py", line 128, in main
    input("Press Enter to continue...")
KeyboardInterrupt

chinmayi@DELL-PCN: /mnt/c/Users/chinn/aos/distributed-banking-system$ ^C
chinmayi@DELL-PCN: /mnt/c/Users/chinn/aos/distributed-banking-system$ redis-cli monitor
OK
1733284105.425018 [0 127.0.0.1:43068] "SETEX" "table_CUSTOMERS" "300" "[[1, \"yashu\", \"yashu\", \"11\", \"1\", \"N\", 10000.0], [2, \"chinmayi\", \"sony\", \"123\", \"123\", \"N\", 105500.0], [3, \"harishma\", \"valli\", \"99\", \"99\", \"N\", 99500.0], [4, \"hari\", \"hari\", \"88\", \"88\", \"Y\", 94990.0]]"
1733284105.430790 [0 127.0.0.1:43068] "SETEX" "table_TRANSACTIONS" "300" "[[1, 3, 2, 500.0, \"TRANSFER\", \"2024-11-25 17:55:53\"], [2, 4, 2, 5000.0, \"TRANSFER\", \"2024-12-03 21:34:27\"], [3, 4, 4, 10.0, \"WITHDRAWAL\", \"2024-12-03 21:47:42\"]]"
1733284105.434133 [0 127.0.0.1:43068] "SETEX" "table_AUTH" "300" "[[1, \"fad222245f7dcb62e36f6ffef32f9c7e926b5e5ec4819a95e40141b513782e53\", null], [2, \"fad222245f7dcb62e36f6ffef32f9c7e926b5e5ec4819a95e40141b513782e53\", null], [3, \"a8a049f6b0f246cccf9f8d0fc5441578c29a5cbb357e0d63c264c8df99c0d89\", null], [4, \"f7b3781c5eafc2779a96bae24875a83ecce46f198e9f81916521d9d218c7da7\", null]]]"
1733284107.478141 [0 127.0.0.1:43068] "SETEX" "table_CUSTOMERS" "300" "[[1, \"yashu\", \"yashu\", \"11\", \"1\", \"N\", 10000.0], [2, \"chinmayi\", \"sony\", \"123\", \"123\", \"N\", 105500.0], [3, \"harishma\", \"valli\", \"99\", \"99\", \"N\", 99500.0], [4, \"hari\", \"hari\", \"88\", \"88\", \"Y\", 94990.0]]"
1733284107.481989 [0 127.0.0.1:43068] "SETEX" "table_TRANSACTIONS" "300" "[[1, 3, 2, 500.0, \"TRANSFER\", \"2024-11-25 17:55:53\"], [2, 4, 2, 5000.0, \"TRANSFER\", \"2024-12-03 21:34:27\"], [3, 4, 4, 10.0, \"WITHDRAWAL\", \"2024-12-03 21:47:42\"]]"
1733284107.484786 [0 127.0.0.1:43068] "SETEX" "table_AUTH" "300" "[[1, \"fad222245f7dcb62e36f6ffef32f9c7e926b5e5ec4819a95e40141b513782e53\", null], [2, \"fad222245f7dcb62e36f6ffef32f9c7e926b5e5ec4819a95e40141b513782e53\", null], [3, \"a8a049f6b0f246cccf9f8d0fc5441578c29a5cbb357e0d63c264c8df99c0d89\", null], [4, \"f7b3781c5eafc2779a96bae24875a83ecce46f198e9f81916521d9d218c7da7\", null]]]"
1733284109.509335 [0 127.0.0.1:43068] "SETEX" "table_CUSTOMERS" "300" "[[1, \"yashu\", \"yashu\", \"11\", \"1\", \"N\", 10000.0], [2, \"chinmayi\", \"sony\", \"123\", \"123\", \"N\", 105500.0], [3, \"harishma\", \"valli\", \"99\", \"99\", \"N\", 99500.0], [4, \"hari\", \"hari\", \"88\", \"88\", \"Y\", 94990.0]]"
1733284109.513342 [0 127.0.0.1:43068] "SETEX" "table_TRANSACTIONS" "300" "[[1, 3, 2, 500.0, \"TRANSFER\", \"2024-11-25 17:55:53\"], [2, 4, 2, 5000.0, \"TRANSFER\", \"2024-12-03 21:34:27\"], [3, 4, 4, 10.0, \"WITHDRAWAL\", \"2024-12-03 21:47:42\"]]"
```

Fig-22: Using command for cache testing

Notification Services:

check for “def sendSMS” in db_exec.py



```
chinmayi@DELL-PCN: /mnt/
def sendSMS(accounts, amount, type, date):
    """Send SMS notifications (using Twilio)"""
    try:
        account_sid = 'ACa921f96506a5b040ae1af554e1eaa69'
        auth_token = '3a35235db6aa39d5c33574f3b5c6f09c'
        client = tr.Client(account_sid, auth_token)

        phone = None
        body = None

        if type == 'd': # Deposit
            status = executeQuery(f'''
                SELECT phone_num
                FROM CUSTOMERS
                WHERE account_num={accounts[0]}
            ''')
            phone = '+1' + status[1][0][0]
            body = (f'Fintech Fortress bank account XXX{str(accounts[0]).zfill(3)[-3:]} '
                    f'was credited for Rs {amount} on {date}.')

        elif type == 'w': # Withdrawal
            status = executeQuery(f'''
                SELECT phone_num
                FROM CUSTOMERS
                WHERE account_num={accounts[0]}
            ''')
            phone = '+1' + status[1][0][0]
            body = (f'Fintech Fortress bank account XXX{str(accounts[0]).zfill(3)[-3:]} '
                    f'was debited for Rs {amount} on {date}.')

        elif type == 'ts': # Transfer (sender)
            status = executeQuery(f'''
                SELECT phone_num, first_name, last_name
                FROM CUSTOMERS
                WHERE account_num={accounts[1]}
            ''')
            receiver_name = f'{status[1][0][1].upper()} {status[1][0][2].upper()}'
            status = executeQuery(f'''
                SELECT phone_num
                FROM CUSTOMERS
                WHERE account_num={accounts[0]}
            ''')
```

Fig-23: Code Snippet for Notification Services

```

    '''
    phone = '+1' + status[1][0][0]
    body = (f'Fintech Fortress bank account XXX{str(accounts[0]).zfill(3)[-3:]} '
           f'was debited for Rs {amount} on {date}. '
           f'{receiver_name} credited.')

    elif type == 'tr': # Transfer (receiver)
        status = executeQuery(f'''
            SELECT phone_num
            FROM CUSTOMERS
            WHERE account_num={accounts[1]}
        ''')
        phone = '+1' + status[1][0][0]
        body = (f'Fintech Fortress bank account XXX{str(accounts[1]).zfill(3)[-3:]} '
               f'was credited for Rs {amount} on {date}.')

    if phone and body:
        client.messages.create(
            from_='+18663125015',
            body=body,
            to='+13617654010'
        )
except Exception as e:
    print(f"SMS sending error: {e}")

```

Fig-24: Continuation code snippet for Notification Services

And there is also config file to update the two fields: twilio_config.py in config directory

```

# Twilio Configuration
TWILIO_CONFIG = {
    'account_sid': 'ACa921f96506a5b040ae1af554e1eaac69',
    'auth_token': '3a35235db6aa39d5c33574f3b5c6f09c',
    'from_phone': '+18663125015' # Format: +1xxxxxxxx
}

```

Fig-25: Config File

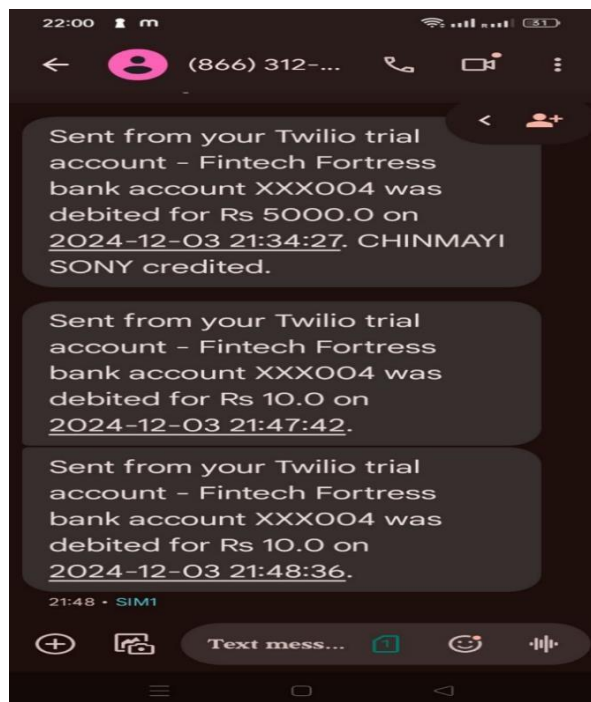


Fig-26: Output of SMS

In addition to this, we have created a webserver. The screenshots of the webserver are shown below:

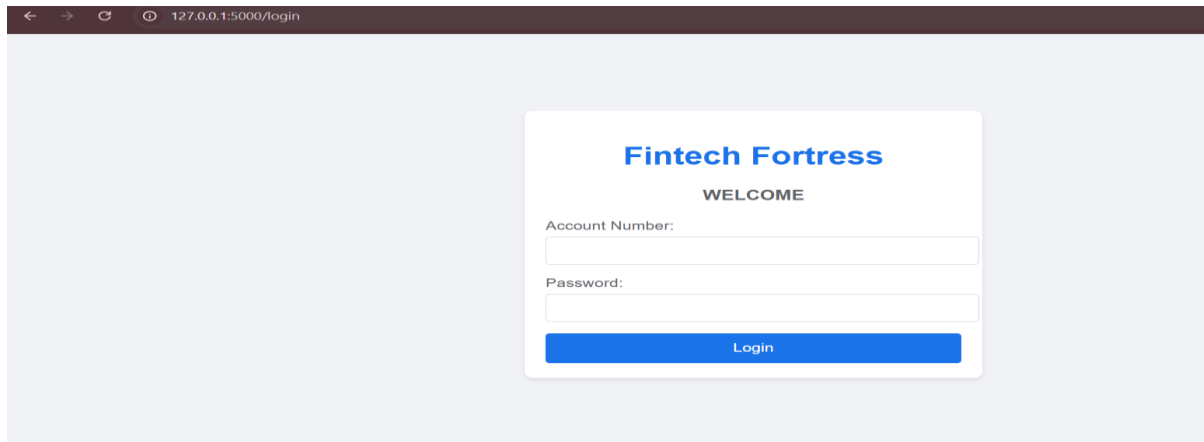


Fig-27: Login page of Webserver

1) Login as Administrator:

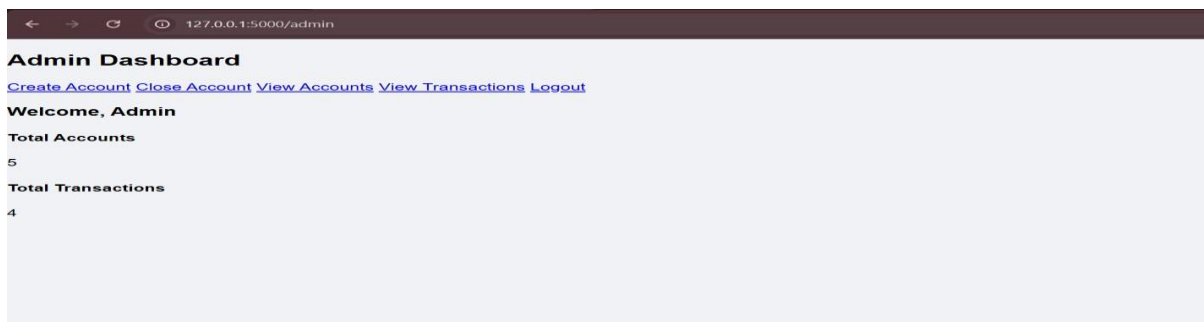
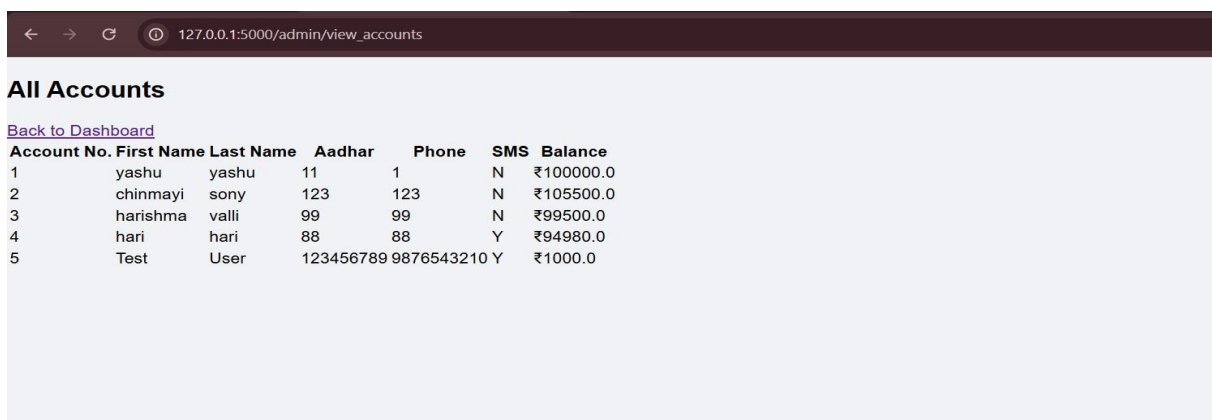
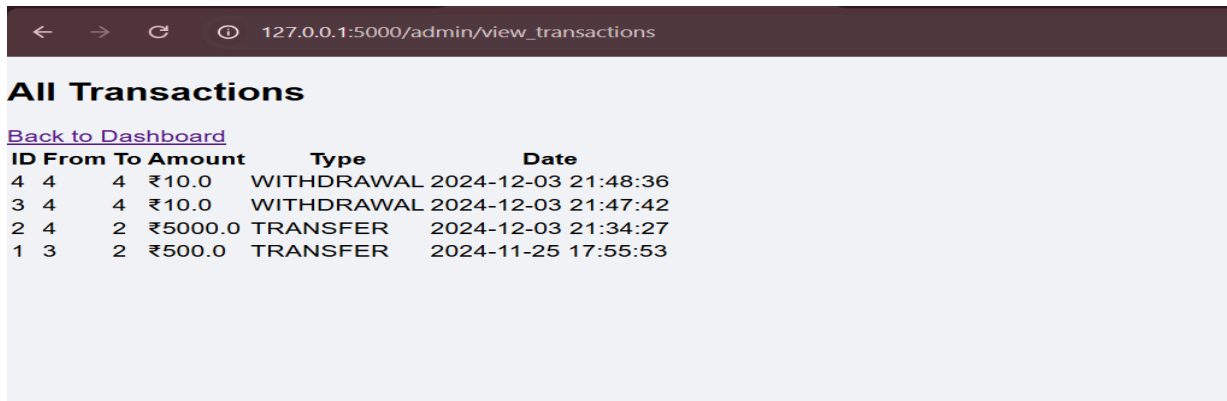


Fig-28: Admin Dashboard



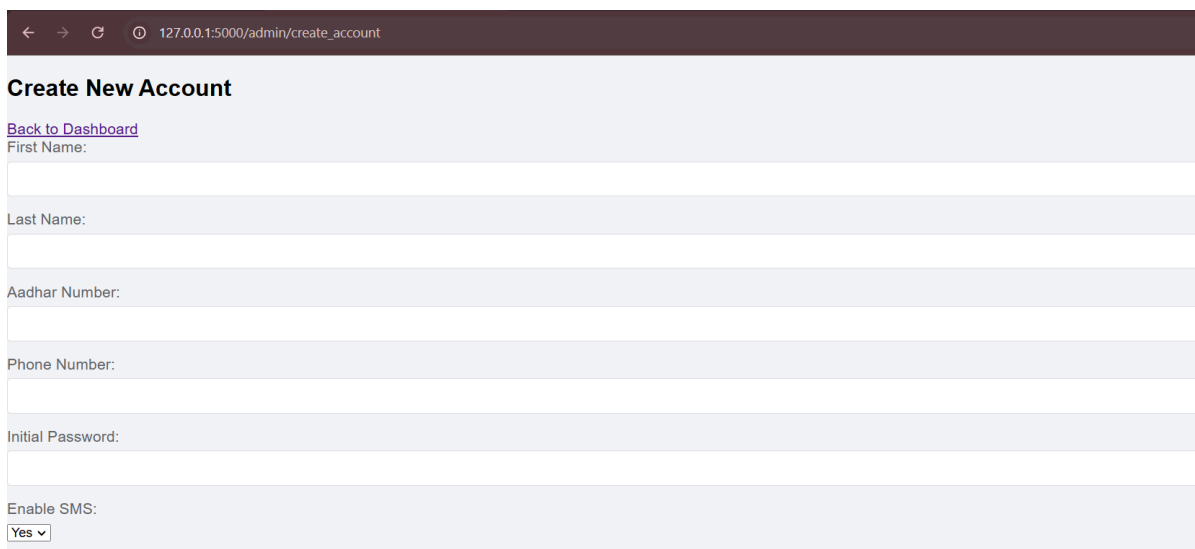
Account No.	First Name	Last Name	Aadhar	Phone	SMS	Balance
1	yashu	yashu	11	1	N	₹100000.0
2	chinmayi	sony	123	123	N	₹105500.0
3	harishma	valli	99	99	N	₹99500.0
4	hari	hari	88	88	Y	₹94980.0
5	Test	User	123456789	9876543210	Y	₹1000.0

Fig-29: To view accounts screen



ID	From	To	Amount	Type	Date
4	4	4	₹10.0	WITHDRAWAL	2024-12-03 21:48:36
3	4	4	₹10.0	WITHDRAWAL	2024-12-03 21:47:42
2	4	2	₹5000.0	TRANSFER	2024-12-03 21:34:27
1	3	2	₹500.0	TRANSFER	2024-11-25 17:55:53

Fig-30: Transaction Screen in Admin View



[Back to Dashboard](#)

First Name:

Last Name:

Aadhar Number:

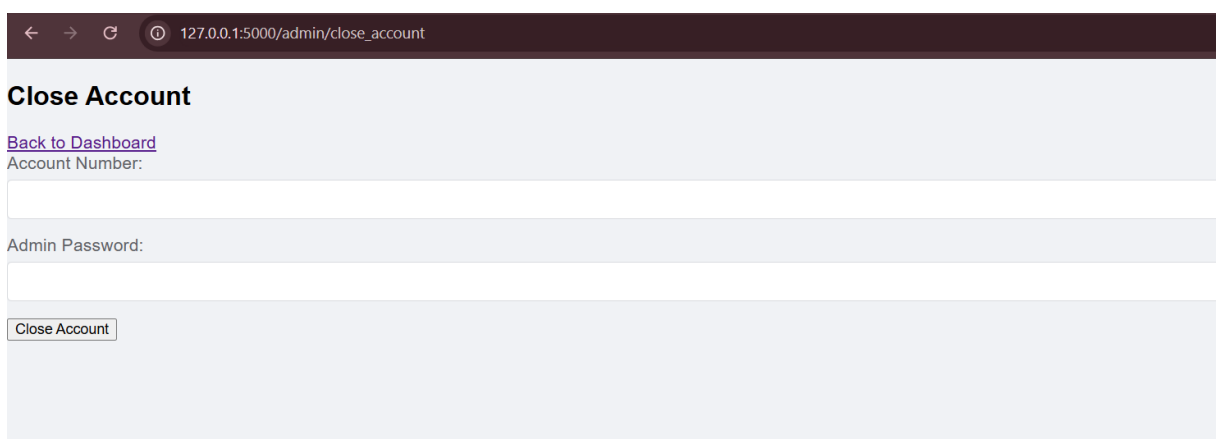
Phone Number:

Initial Password:

Enable SMS:

Yes ▾

Fig-31: To create an account screen



[Back to Dashboard](#)

Account Number:

Admin Password:

Close Account

Fig-32: Account Closure Screen

2. Login as Client:

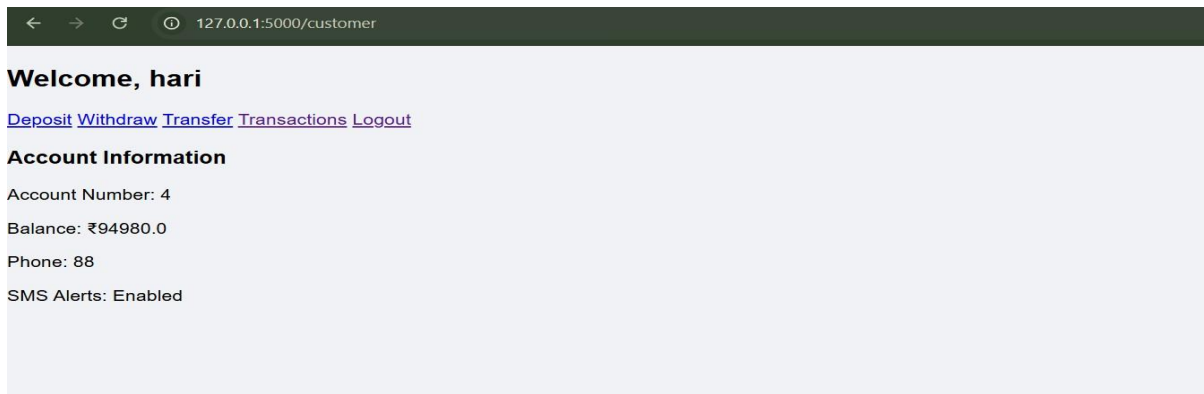


Fig-33: Client Account Dashboard

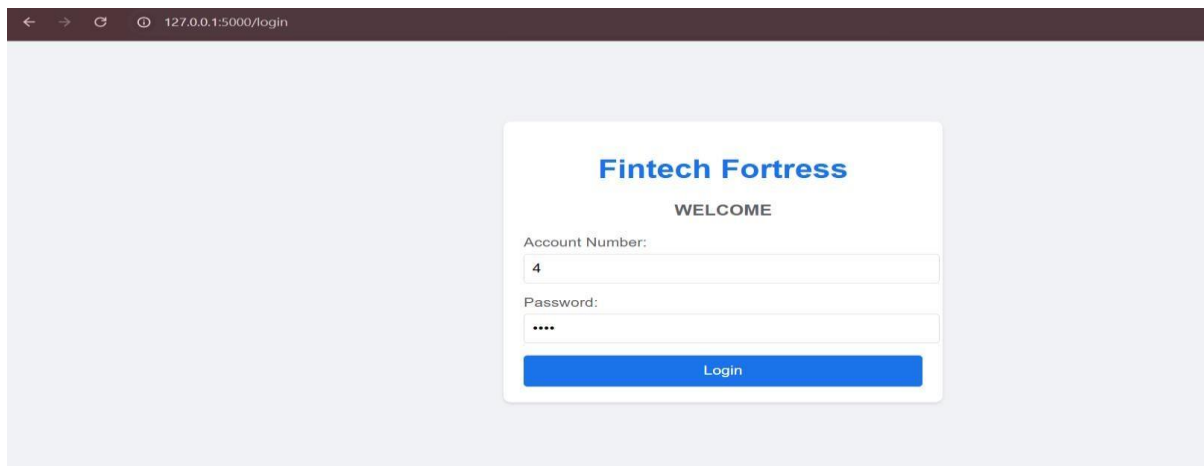


Fig-34: Login as Client Account

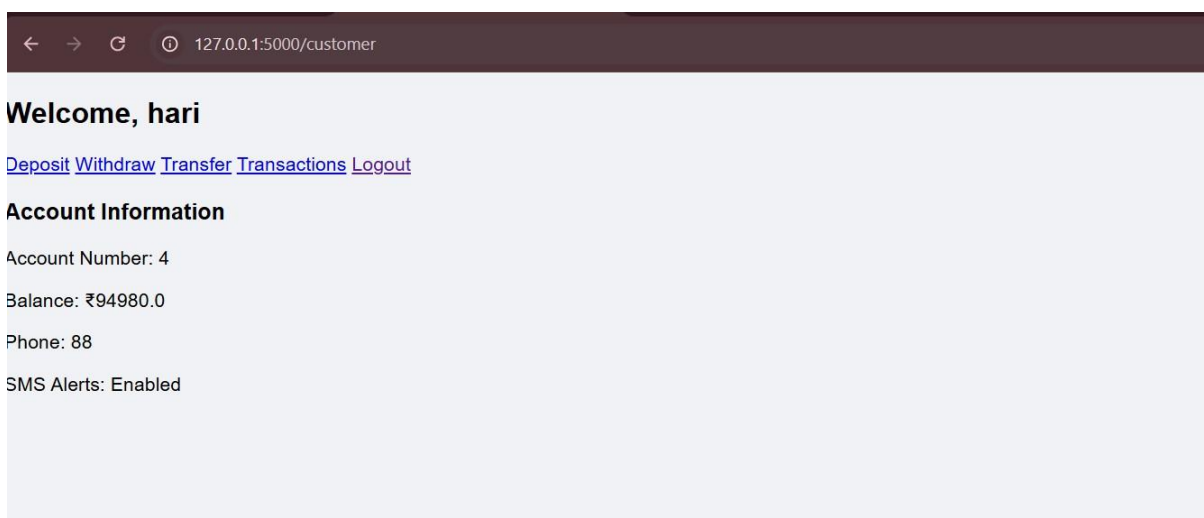


Fig-35: Customer Dashboard Screen

127.0.0.1:5000/customer/deposit

Deposit Money

[Back to Dashboard](#)

Amount:

Deposit

Fig-36: Deposit Screen

127.0.0.1:5000/customer/transfer

Transfer Money

[Back to Dashboard](#)

Recipient Account Number:

Amount:

Transfer

Fig-37: Transfer money screen

127.0.0.1:5000/customer/transactions

Transaction History

[Back to Dashboard](#)

Date	Type	Amount	Details	Balance
2024-12-03 21:48:36	WITHDRAWAL	₹10.0	WITHDRAWAL	₹10.0
2024-12-03 21:47:42	WITHDRAWAL	₹10.0	WITHDRAWAL	₹10.0
2024-12-03 21:34:27	TRANSFER	₹5000.0	To: Acc#2	₹5000.0

Fig-38: Transaction History Screen

7. CHALLENGES & SOLUTIONS

- a. **Security Implementation:** The main challenge in this project is to generate integrity and transaction management. Traditional password hashing was vulnerable to attack, and sensitive data needed to be protected. The solution included the use of zero-knowledge proofs (ZKP) for authentication, eliminating the need to store passwords to provide secure authentication. This cryptographic approach improved security by monitoring that important information was never found on it. Additionally, it prioritized secure connection management and database security to protect data integrity across distributed systems, ensuring that all transactions were handled securely.
- b. **Database Management:** The challenge of maintaining a synchronized database across multiple servers was solved by implementing continuous synchronization. Data consistency across all servers must be maintained to prevent inconsistencies with a distributed system. The solution was to implement a consistent policy, ensuring that changes made on one server were immediately reflected on all others. Database replication techniques were used to further stabilize the data and prevent problems with concurrent synchronization. Additionally, Redis was used for efficient storage management, ensuring data consistency while optimizing system performance.
- c. **Server Synchronization:** The critical challenges are maintaining database consistency across servers. If not properly configured, contradictions can occur, resulting in unreliable data. To overcome this, the DatabaseSync class was implemented, which ensured continuity across servers. This approach provided real-time data synchronization, prevented data inconsistencies, and increased system reliability.
- d. **Password Security:** Handling passwords correctly without direct storage obviously presented a significant challenge. The solution was to use Schnorr's zero-knowledge proof (ZKP) protocol, which enables password less authentication. This cryptographic approach provided a way to authenticate users without revealing passwords, thereby increasing security by eliminating traditional associated risks.
- e. **Authentication System:** Traditional password hashing schemes were sloppy and posed security risks. The implementation of zero-knowledge proof (ZKP) provided a password

less authentication mechanism without the need for password storage, and significantly enhanced security by eliminating password-related threats.

- f. **Error Handling:** The service was interrupted due to an unexpected system crash during operation. The solution was to implement comprehensive error handling that captured and dealt with errors efficiently. This enabled the system to run smoothly even in the event of an outage, improving reliability and uptime.

8. CONCLUSION

A distributed banking system effectively integrates distributed core computing resources to ensure secure and reliable operation. Including fault tolerance to handle component failures, data replication for enhanced availability, and zero-knowledge proofs (ZKP) for secure authentication, it protects user credentials during transactions. Dual server architecture ensures maximum availability by automating failover mechanisms so, providing uninterrupted service.

It is optimized with Redis caching that store frequently accessed data, reduces database load, and improves response time. Additionally, real-time transaction information is available in the system, keeping users accountable for account activity and enhancing security against unauthorized actions.

This scalable and robust solution demonstrates the ability to meet today's banking needs, combining security, convenience and high performance to deliver a seamless and reliable user experience.

9. FUTURE ENHANCEMENTS

Future developments in distributed banking systems could include the integration of smart contracts for businesses to automate and control contract terms, improving efficiency and improving reliability. Immutable transaction records will increase transparency and security by ensuring that all transactions are recorded in an unalterable ledger.

Decentralized accounting can strengthen user confidence and improve privacy by reducing reliance on intervening authorities. In addition, interbank transactions will enable secure and easy payments between financial institutions, remove barriers, reduce fees and improve the convenience of international payments. These upgrades will improve system security, performance and global connectivity.

10. REFERENCES

1. Gomber, P., Koch, J, A and Siering, M Digital Finance and Fintech: current research and future research directions, J Bus Econ, Vol. 87, p.p. 537-580, 2017
2. Sarhan, H. Fintech: An Overview. Available online: <https://www.researchgate.net/publication/342832269>
3. L. Philosophy and A. Pathak, DigitalCommons @ University of Nebraska - Lincoln Bibliometric survey on Zero-Knowledge Proof for Authentication, 2021.
4. L. Chuat, S. Plocher, A. Perrig and E. Zurich, Zero-Knowledge User Authentication: An Old Idea Whose Time Has Come, 2020.
5. Pagnia Henning and Felix C. Grtner, "On the impossibility of fair exchange without a trusted third party" in , Darmstadt, Germany:Darmstadt University of Technology, Department of Computer Science, 1999.
6. Austin Mohr, A survey of zero-knowledge proofs with applications to cryptography, Carbondale:Southern Illinois University, pp. 1-12, 2007.
7. Jeremy Jedlicka, and Emanuel S. Grant, "Data Privacy through Zero-Knowledge Proofs," 2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT), Mandya, India, pp. 1-7, 2022. [CrossRef] [Google Scholar] [Publisher Link]