Distributed System - Assignment-2 Q3: Strife

Overview

This project implements a miniature version of Stripe, a distributed payment gateway system that interfaces with clients and multiple bank servers to manage secure transactions. It leverages gRPC for communication, Consul for service discovery, and SSL/TLS for security, providing a robust framework for authentication, transaction processing, and failure handling.

Components

Client

The client (`client.py`) enables users to perform banking operations such as logging in, checking balances, initiating transfers, and viewing transaction history. It also manages offline payments by queuing transactions when the gateway is unreachable.

Bank

The bank server (`bank.py`) represents individual banks, each with a unique name. It handles user authentication, balance management, and transaction processing using a two-phase commit (2PC) protocol, maintaining user data and transaction history locally.

Gateway

The gateway (`gateway.py`) serves as the central coordinator between clients and banks. It manages user authentication, issues JWT tokens, and orchestrates transactions across banks using the 2PC protocol, ensuring security and consistency.

Files

- **client.py**: Implements client-side logic for user interactions and offline transaction queuing.
- **bank.py**: Manages bank-specific operations, including authentication and transaction processing.
- **gateway.py**: Coordinates communication and transactions between clients and banks.
- **service.proto**: Defines gRPC services and message structures for all interactions.
- **README.md**: This report documenting the implementation and design choices.

Running the Project

1. **Start the Gateway**
   Launch the gateway service:

   ```
   python gateway.py
   ```

2. **Start the Bank**
   Run a bank instance with a unique name:

```
python bank.py --name <bank_name>
```

3. **Run the Client**
   Interact with the gateway, optionally providing credentials:

```
python client.py [<bank_name> <username> <password>]
```

Report Requirements

Design Choices

System Architecture

- **Separation of Responsibilities**:
  - **Clients**: Handle user interactions (login, balance checks, transfers), assign unique transaction IDs (UUIDs), and queue transactions during offline scenarios using a background thread (`offline_txn_worker`).
  - **Bank Servers**: Manage user authentication, store account balances and transaction history, and participate in the 2PC protocol to ensure transaction integrity.
  - **Payment Gateway**: Acts as an intermediary, authenticating users, issuing JWT tokens, and coordinating multi-bank transactions via 2PC.
- **Communication**:
  - All interactions use **gRPC with SSL/TLS secure channels** to ensure data confidentiality and integrity.
  - Certificates are generated for mutual authentication between clients, the gateway, and banks, with public keys stored in Consul.
- **Service Discovery**:
  - **Consul** enables dynamic discovery of bank servers and the gateway, registering services with health checks.
  - It also serves as a trusted authority for certificate distribution, ensuring secure communication setup.

Authentication

- **Mechanism**:
  - Clients authenticate by sending username, bank name, and password to the gateway (`Login` method).
  - The gateway forwards credentials to the appropriate bank for validation (`Authenticate` method).
  - Upon success, the gateway issues a **JWT token** (signed with a secret key, valid for 1 hour), stored in a token registry for subsequent validation.

- **Implementation**:
  - Bank servers verify passwords against a hashed version loaded from `user.json`.
  - The gateway uses a `JwtInterceptor` to validate tokens for all protected methods (e.g., `ViewBalance`, `TransferMoney`).

Authorization

- **Mechanism**:
  - Authorization is enforced via the `JwtInterceptor`, which checks token validity and presence in the token registry.
  - Tokens encode user ID and bank name, allowing the gateway to restrict actions to authenticated users (e.g., viewing their own balance or initiating transfers within available funds).
- **Implementation**:
  - Only authenticated users with valid tokens can access protected endpoints, ensuring role-based access control.

Logging

- **Mechanism**:
  - Both the gateway and banks implement a `LoggingInterceptor` to capture detailed logs of gRPC requests and responses.
  - Logs include method names, request data, response data, and errors, written to files (e.g., `gateway_interceptor.log`, `<bank_name>_interceptor.log`).
- **Purpose**:
  - Provides transparency for debugging and monitoring, though not used for recovery in this scope.

Idempotency Approach and Correctness Proof

Approach

- **Unique Transaction IDs**:
  - Each transaction is assigned a unique UUID by the client (`initiate_transfer` in `client.py`).
  - The gateway maintains a `transaction_ids` set to track processed transactions, rejecting duplicates in `TransferMoney`.
  - Banks similarly track transaction IDs in `prepare_transaction` during the prepare phase, preventing duplicate processing.

Implementation

- **Gateway Check**: Before initiating 2PC, the gateway verifies the transaction ID isn't in `transaction_ids`.
- **Bank Check**: During `PrepareDebit` and `PrepareCredit`, banks reject requests with IDs already in `prepare_transaction`.

Correctness Proof

- **Uniqueness**: UUIDs ensure each transaction ID is globally unique.
- **Detection**:
  - If a client retries a transaction, the gateway detects the duplicate ID and returns an error ("Duplicate transaction ID").
  - Banks independently reject prepare requests for known IDs, ensuring no double processing.
- **Persistence**: Completed transaction IDs are saved to `gateway_data.json` on gateway shutdown, preventing reuse post-restart (though ongoing transactions aren't persisted—see assumptions).
- **Guarantee**: Since checks occur before any state change, and IDs are unique and persistent for completed transactions, each transaction affects balances exactly once, even with retries.

Failure Handling Mechanisms

Offline Payments

- **Queuing**:
  - When the gateway is unreachable (`create_channel` fails), the client queues transactions in `transaction_queue` (`initiate_transfer` in `client.py`).
- **Retry Mechanism**:
  - A background thread (`offline_txn_worker`) periodically (every 0.2 seconds) checks gateway availability via Consul and resends queued transactions.
  - Results are stored in `queue_output` for user notification (success/failure, including token expiration errors).
- **Idempotency**: Retries are safe due to the unique transaction ID checks, preventing duplicate deductions.

2PC with Timeout

- **Protocol**:
  - The gateway coordinates transactions using 2PC in `TransferMoney`:
    1. **Prepare Phase**: Sends `PrepareDebit` and `PrepareCredit` to source and destination banks with a timeout (`TIMEOUT2PC` from `config.json`).
    2. **Commit Phase**: If both prepare calls succeed, sends `CommitDebit` and `CommitCredit`; otherwise, sends `AbortDebit` and `AbortCredit`.
- **Timeout Handling**:
  - If any call times out (gRPC `RpcError`), the gateway aborts the transaction, ensuring no partial commits.
  - Aborts restore balances (e.g., `AbortDebit` adds funds back) if the prepare phase succeeded but commit failed.
- **Consistency**: The configurable timeout ensures transactions either fully complete or fully abort, maintaining bank state consistency.

Fault Tolerance Notes

- **State Persistence**:
  - The gateway saves completed `transaction_ids`, user data, and tokens to `gateway_data.json` on shutdown.
  - Banks save balances and history to `user.json`.

    - Ongoing transactions (in `prepare_transaction` or mid-2PC) aren't persisted, so crashes abort them implicitly.
- **Recovery**: Post-restart, completed transactions are preserved, but clients must retry aborted ones.

Assumptions

- **Uniqueness**: Usernames are unique within each bank, with one account per user.
- **Consul Reliability**: Consul is always available and secure, serving as a trusted authority for service discovery and certificates.
- **Crash Handling**: Server crashes abort ongoing transactions; clients must retry. Completed states are recovered from saved files.
- **Token Expiry**: Offline transactions with expired tokens are skipped, requiring re-authentication.

Additional Design Highlights

- **Scalability**: Consul enables dynamic bank registration, supporting system growth.
- **Security**: JWT tokens and SSL/TLS provide stateless, secure authentication and communication.
- **Non-Blocking**: The client's offline thread avoids blocking user interactions, enhancing usability.
