Secure Distributed Payment Gateway: Strife

1. Project Overview

This project implements a Secure Distributed Payment Gateway called Strife using gRPC. It allows multiple bank servers and clients to securely process transactions via Two-Phase Commit (2PC) and ensures system reliability with robust authentication, logging, and offline transaction handling. Strife is designed to simulate the functionality of Stripe, handling transactions between clients and bank servers in a secure, reliable, and scalable manner.

The system addresses the following requirements:

- Implementation of bank servers, clients, and a payment gateway using gRPC
- Secure authentication, authorization, and logging using SSL/TLS and gRPC interceptors
- Idempotent payment processing to prevent duplicate deductions
- Offline payment queuing and automatic resending upon reconnection
- Transaction integrity using the 2PC protocol with configurable timeouts
- Fault tolerance and scalability, with persistent state for recovery

2. System Architecture

The system consists of three main components:

2.1 Payment Gateway (`payment_gateway.py`)

- Acts as the central hub managing transactions between clients and bank servers
- Implements `PaymentGatewayServicer` with gRPC services for client authentication, payment processing, balance queries, and transaction history
- Uses interceptors for logging and authorization
- Serves as the coordinator in the 2PC protocol
- Persists transaction states in `transactions.json` for recovery after crashes
- Supports the following gRPC methods:
    - `AuthenticateClient` : Authenticates users and issues JWT tokens
    - `LogoutClient` : Invalidates active JWT tokens
    - `ProcessPayment` : Coordinates transactions using 2PC
    - `CheckBalance` : Retrieves account balances from banks
    - `ViewTransactionHistory` : Returns transaction history for authenticated users
    - `RegisterBank` : Registers banks with the payment gateway, updating `banks.json`

2.2 Bank Servers (`bank_server.py`)

- Each bank runs an independent gRPC server handling account operations
- Implemented as `BankServicer` that manages accounts and participates in transactions
- Accounts are loaded from `bank_data.json` at startup, and updates are persisted to disk
- Supports the following gRPC methods:
  - `GetBalance` : Retrieves the balance of an account
  - `ProcessTransaction` : A legacy method for direct transactions (not used in 2PC)
  - `PrepareTransaction` , `CommitTransaction` , `AbortTransaction` : Implements the 2PC protocol for transaction integrity
- Banks register with the payment gateway, storing their port mappings in `banks.json`

2.3 Clients (`client.py`)

- Users interact with the system, initiating payments and checking balances
- Provides an interactive command-line interface for authentication, payments, balance checks, and transaction history
- Handles offline payments by queuing them in `offline_payments.json`
- Implements retry logic with configurable parameters (`MAX_RETRIES = 3` , `RETRY_DELAY = 3s`) for transient failures
- Runs a background thread (`connection_monitor`) to detect reconnection and process offline payments

3. Authentication & Security

3.1 JWT-based Authentication

- Clients authenticate using username and password, receiving a JWT token upon success
- The payment gateway issues a JWT token with a 5-minute expiration upon successful authentication (`generate_jwt` in `payment_gateway.py`)
- Users are pre-loaded from `users.json` at startup, supporting multiple bank accounts per user but adhering to the assumption of one account per bank

3.2 SSL/TLS Encryption

- Secure communication between all components using SSL/TLS
- Mutual authentication enforced with certificates in the `certs/` directory (`require_client_auth=True`)
- Each component (client, payment gateway, bank) uses its respective certificates, trusting the CA (`ca.crt`)

- Banks must have valid SSL certificates (`*.crt` , `*.key`) signed by a trusted Certificate Authority (CA) before registering with the Payment Gateway

3.3 Role-Based Access Control

- Implemented via gRPC Interceptors (`AuthorizationInterceptor` in `interceptor.py`)
- The interceptor verifies the JWT token for all requests except `AuthenticateClient` and `RegisterBank`
- For `CheckBalance` , the interceptor ensures the requested bank is linked to the user's account
- For `ProcessPayment` , the interceptor logs the transaction but does not restrict access beyond authentication
- The payment gateway ensures that users cannot transfer money to their own account within the same bank

4. Transaction Processing (2PC)

The system implements Two-Phase Commit (2PC) for secure transaction handling:

4.1 Prepare Phase

- Payment gateway calls `PrepareTransaction` on sender and receiver banks
- Banks lock funds and check account validity before committing
- Transaction state is persisted as "initiated" in `transactions.json`

4.2 Commit Phase

- If all parties agree (both banks prepare successfully), the payment gateway calls `CommitTransaction`
- The transaction is finalized and state updated to "completed"

4.3 Abort Phase

- If any step fails (insufficient funds, timeout), the payment gateway calls `AbortTransaction`
- Locks are released and transaction state updated to "aborted"

4.4 Timeout Mechanism

- Configurable timeout (`TRANSACTION_TIMEOUT = 5s`) for each phase
- If a bank does not respond within timeout, transaction is aborted
- Ensures system remains responsive in presence of failures
- The timeout is configured to balance responsiveness and reliability

5. Idempotent Payment Processing

5.1 Implementation

- Each payment uses a unique `transaction_id` (UUID) generated by the client (`process_payment` in `client.py`)
- Payment gateway checks for duplicate transactions before processing (`ProcessPayment` in `payment_gateway.py`)
- Duplicates are rejected with `grpc.StatusCode.ALREADY_EXISTS` , informing the client of the prior attempt's outcome
- Transaction states are persisted in `transactions.json`

5.2 Rationale

- **Scalability**: UUIDs are globally unique, making the approach scalable across distributed systems without relying on timestamps, which can collide in high-concurrency scenarios
- **Correctness**: Persisting transaction states ensures that even after crashes, the system can detect and reject duplicate transactions, guaranteeing exactly-once processing
- **Simplicity**: The approach leverages gRPC's built-in error handling to inform clients of duplicate attempts, simplifying client-side retry logic

5.3 Proof of Correctness

To prove that the idempotency approach ensures exactly-once payment processing, consider the following formal analysis:

5.3.1 Assumptions

- Each transaction is identified by a unique `transaction_id` (UUID), which is statistically unlikely to collide due to the large UUID space (2^128)
- Transaction states are persisted atomically to `transactions.json` using a file-based approach
- Network faults may cause retries, but the client always sends the same `transaction_id` for a given payment

5.3.2 Execution Model

- **Initial Request**: Client generates a `transaction_id` and sends a `ProcessPayment` request
- **Duplicate Check**: Payment gateway checks if `transaction_id` exists; if so, rejects with `ALREADY_EXISTS`
- **Processing**: If `transaction_id` is new, the payment gateway initiates 2PC protocol,

persisting state as "initiated"
- **Completion**: Upon successful 2PC, the state is updated to "completed"; if 2PC fails, state is updated to "aborted"

5.3.3 Correctness Properties

1. **Safety (No Double Deduction)**:
    - If a client retries a payment with the same `transaction_id`, the payment gateway rejects it after the first attempt
    - **Proof**: Let T be a transaction with ID TID. If T is processed successfully, its state is persisted as "completed". Any subsequent request with TID is rejected. If T fails, its state is persisted as "aborted", and retries are rejected similarly. Thus, T is applied exactly once.
2. **Liveness (Progress)**:
    - The 2PC protocol ensures atomicity, so partial completions are rolled back via `AbortTransaction`
    - **Proof**: If T is in progress (state "initiated"), a retry with TID is rejected as a duplicate. If T completes, retries are rejected. If T is interrupted, the state is recovered, and processing resumes or aborts, ensuring progress.
3. **Fault Tolerance**:
    - If the payment gateway crashes after persisting the transaction state, the client can retry safely
    - **Proof**: Let T be interrupted after persisting state S. Upon restart, S is reloaded, and T is either completed (if S = "initiated") or rejected (if S = "completed" or S = "aborted"). Thus, T is applied exactly once, even under crashes.

6. Offline Payments & Retry Mechanism

6.1 Client-Side Queuing

- If the Payment Gateway is down, clients queue transactions locally in `offline_payments.json`
- Payments are queued if the gateway is unreachable, detected via `grpc.StatusCode.UNAVAILABLE` errors (`process_payment` in `client.py`)
- Queue is persisted to survive client restarts, ensuring no loss of pending transactions

6.2 Automatic Retries

- Background thread (`connection_monitor`) checks connectivity every 30 seconds (`RECONNECT_INTERVAL`)
- Once the gateway is online, pending transactions are automatically retried
- Successful payments are removed from queue
- Failed payments (e.g., insufficient funds) are logged but not retried, notifying the client of the outcome

6.3 Failure Handling for Offline Payments

- **Permanent Offline**: If the payment gateway remains offline, payments stay queued indefinitely and are logged in `client.log` . The client is notified of the queued status.
- **Insufficient Funds**: Upon reconnection, if a payment fails due to insufficient funds, it is removed from the queue and logged, preventing the queue from growing indefinitely with unprocessable payments.
- **Duplicate Payments**: Idempotency ensures that duplicate attempts are rejected, preventing double deductions. If a queued payment has already been processed, it is rejected with `ALREADY_EXISTS` and removed from the queue.

6.4 Failure Handling for 2PC Timeouts

- **Timeout During Prepare**: If a bank does not respond within the timeout, the transaction is aborted, and the client is notified (`PaymentResponse` with `success=False`). The payment gateway calls `AbortTransaction` on all participants.
- **Timeout During Commit**: If a bank commits but another times out, the transaction is marked as "partial_commit_recovery_needed" in `transactions.json` . This is logged as a critical error, notifying the client of the partial failure.
- **Crash Recovery**: After a crash, the payment gateway reloads transaction states and aborts any incomplete transactions, ensuring consistency and preventing orphaned locks or inconsistent updates.

7. Logging & Monitoring

7.1 Comprehensive Logging

- All transactions (successful & failed) are logged using multiple gRPC interceptors:
  - `LoggingInterceptor` : General request and response logging
  - `TransactionLoggingInterceptor` : Detailed transaction logging
  - `RegisterBankInterceptor` : Bank registration logging

7.2 Log Files

- `banktransactions.log` : Bank server activities
- `transactions.log` : Payment gateway activities
- `interceptortransactions.log` : Interceptor activities (requests, responses, errors)
- `client.log` : Client activities, including offline payment queuing

7.3 Transaction Tracing

- Logs include transaction amounts, client IP addresses, method names, errors, and retry attempts
- Transaction retries are logged to ensure idempotency is correctly handled
- Provides transparency into retry attempts and their outcomes

8. Assumptions

The following assumptions are made in the design and implementation:

1. **`users.json` contains predefined users; no new registrations**: User credentials and account mappings are static and loaded from `users.json` at startup, without support for dynamic user registration.
2. **A user cannot have multiple accounts in the same bank**: Each user is associated with at most one account per bank, ensuring unambiguous account selection during transactions.
3. **Banks must have SSL certificates before registering**: Each bank server must have valid SSL/TLS certificates (`*.crt` , `*.key`) signed by a trusted Certificate Authority (CA) before registering with the payment gateway.
4. **Clients & Banks already know the Payment Gateway's address**: The payment gateway's address (`localhost:50052`) is hardcoded in client and bank server implementations, eliminating the need for dynamic service discovery.

9. Future Enhancements

- Implement an Admin Dashboard for transaction monitoring
- Support multi-bank transactions for cross-bank payments
- Improve system scalability with load balancing
- Add database persistence with ACID guarantees instead of file-based storage