Report: Building a Miniature Stripe-like Payment System

Overview

This project implements a simplified payment gateway inspired by Stripe, interfacing clients with bank servers to manage transactions securely and reliably. The system comprises bank servers, clients, and a payment gateway, with features including secure authentication, idempotent payments, offline processing, and 2PC with timeouts. Below, I detail the design choices for each requirement and provide justifications where required.

3.1 System Components

Bank Servers

- Design Choice: Each bank server is implemented as a gRPC service representing an individual bank. Multiple instances are instantiated, each identified by a unique name (e.g., "bank_a", "bank_b"). They manage accounts, process transactions, and participate in 2PC as voters.
- Implementation: Bank servers store account data (account number, balance) in a persistent store (e.g., a JSON file or database) loaded at startup from a setup file. This ensures recovery from crashes by restoring the last known state.
- Justification: Unique names enable the payment gateway to route transactions to the correct bank, while persistence ensures fault tolerance.

Clients

- Design Choice: Clients are gRPC clients that interact with the payment gateway to register, authenticate, check balances, and initiate transactions. They provide a username, password, and initial balance during registration.
- Implementation: Clients maintain a local queue for offline payments and use session keys for authenticated requests post-login.
- Justification: Registration and authentication align with real-world payment systems, ensuring only legitimate users can transact. The offline queue enhances reliability.

Payment Gateway

- Design Choice: The payment gateway is a central gRPC service coordinating between clients and bank servers. It handles authentication, transaction initiation, and 2PC coordination.
- Implementation: It validates client credentials, issues session keys, and manages transaction states for idempotency and 2PC.
- Justification: A centralized coordinator simplifies transaction management and ensures consistency across distributed bank servers.

3.2 Secure Authentication, Authorization, and Logging

3.2.1 Authentication

- Design Choice: Authentication uses SSL/TLS mutual authentication for secure communication and a username-password mechanism for client verification, with key being sent to authenticate for requests. Key is sent to the client when running login function.
- Implementation:
    - SSL/TLS: gRPC's built-in support secures client-gateway and gateway-bank communication using server certificates and a trusted Certificate Authority (CA).
    - Client Login: Clients submit a username and password, validated against a preloaded JSON setup file containing user credentials and account details (e.g., `{"username": "alice", "password": "pass123", "account_no": "123", "balance": 1000}`).
    - Session Keys: Upon successful login, the gateway issues a unique session key (e.g., a UUID), which clients include in subsequent requests.
- Justification: SSL/TLS prevents eavesdropping, while session keys reduce the need for repeated credential transmission, enhancing security and efficiency. This helps enforce Idempotency

3.2.2 Authorization

- Design Choice: Role-based authorization is enforced using gRPC interceptors to validate session keys and restrict operations.
- Implementation:
    - Interceptors check the session key in each request's metadata.
    - Permissions:

        - View Balance: Allowed only for the account owner (session key matches the account).
        - Initiate Transactions: Permitted if the session key is valid and the account has sufficient funds.
- Justification: Interceptors provide a scalable, centralized way to enforce access control, ensuring only authorized clients access sensitive operations.

3.2.3 Logging

- Design Choice: Verbose logging is implemented via gRPC interceptors to monitor system health and debug issues.
- Implementation:
    - Interceptor: Logs every request and response on the gateway and bank servers.
    - Log Content:
        - Transaction amount (e.g., "100 INR").
        - Client ID (from session key) and IP address.
        - Method name (e.g., `ProcessPayment`).
        - Errors (e.g., "Insufficient funds, code: 5").
    - Retry Logging: Logs retry attempts to track idempotency enforcement.
- Justification: Detailed logs provide transparency into payment flows and aid troubleshooting, aligning with industry practices (though recovery from logs is out of scope here).

3.3 Idempotent Payments

Design Choice

- Payments are made idempotent using unique transaction IDs and state tracking, avoiding simple timestamp-based deduplication for scalability.

Implementation

- Transaction IDs: Each client generates a unique transaction ID (e.g., a UUID) for every payment request.
- State Tracking:
    - The gateway maintains a persistent store (e.g., in-memory map or Redis) of transaction states: `INITIATED`, `COMMITTED`, `ABORTED`.

        - Bank servers also track transaction IDs to prevent duplicate processing.
- Deduplication Logic:
    - On receiving a transaction:
        - If the ID exists and is `COMMITTED` or `ABORTED`, return the stored result.
        - If `INITIATED`, wait for completion or abort if timed out.
        - If new, process the transaction via 2PC.

Correctness Proof

- Property: A transaction with ID `T` deducts funds exactly once, regardless of retries.
- Proof:
    i. Initial Request: Client sends `T` with amount `A`. The gateway records `T: INITIATED` and starts 2PC. If successful, `T: COMMITTED`; if failed, `T: ABORTED`. Banks deduct `A` only on `COMMIT`.
    ii. Retry Request: Client resends `T` due to timeout or network failure:
        - If `T: COMMITTED`, gateway returns success, and banks ignore duplicates (ID already processed).
        - If `T: ABORTED`, gateway returns failure, and no deduction occurs.
        - If `T: INITIATED`, gateway waits or aborts if timed out, ensuring no double deduction.
    iii. Network Faults: Persistent state ensures the gateway and banks agree on `T`'s outcome post-recovery.
- Conclusion: The use of unique IDs and state tracking ensures idempotency, preventing multiple deductions.

Justification

- This approach scales better than timestamps, as it handles distributed systems and network delays without clock synchronization issues.

3.4 Offline Payments

Design Choice

- Clients queue payments locally when offline and resend them upon reconnection, leveraging idempotency for safe retries.

Implementation

- Queue: If a gRPC call fails (e.g., `UNAVAILABLE` status), the client stores the payment request (transaction ID, amount, recipient) in a local queue.
- Retry Mechanism:
    - Check connectivity every 5 seconds (configurable period).
    - Use exponential backoff (e.g., 5s, 10s, 20s) to avoid overwhelming the gateway.
- Processing: On reconnection, resend queued payments. Idempotency ensures no duplicates.
- Notifications: Post-processing, clients receive success/failure messages via gRPC responses.

Failure Handling

- Offline Detection: gRPC errors trigger queuing.
- Retry Success: Idempotency guarantees safe retries, even if the original request partially succeeded.
- User Feedback: Notifications inform users of outcomes, enhancing usability.

Justification

- Queuing and periodic retries ensure payments are not lost, while idempotency simplifies handling partial failures.

3.5 2PC with Timeout

Design Choice

- The payment gateway acts as the 2PC coordinator, with bank servers as participants, incorporating configurable timeouts for fault tolerance.

Implementation

- Phase 1 (Prepare):
    - Gateway sends `PREPARE` messages to sending and receiving banks with transaction details.
    - Banks vote `YES` (sufficient funds, valid transaction) or `NO`.

- Phase 2 (Commit/Abort):
    - If all vote `YES`, gateway sends `COMMIT`; banks deduct/add funds.
    - If any vote `NO` or timeout occurs, gateway sends `ABORT`; no changes are applied.
- Timeout: Configurable (e.g., 10 seconds per phase). If votes or acknowledgments are not received, the transaction aborts.

Failure Handling

- Timeouts: Prevent indefinite waiting by aborting stalled transactions.
- Crash Recovery: Banks log transaction states (e.g., `PREPARED`, `COMMITTED`) to disk, ensuring consistency post-restart.

Justification

- 2PC ensures atomicity across banks, while timeouts and logging enhance reliability in failure-prone environments.

3.6 Implementation Details

- gRPC: Used for all communication with defined services (e.g., `ClientService`, `BankService`).
- Fault Tolerance: Persistent storage (files/databases) for bank states and transaction logs.
- Scalability: Load balancing for gateway instances and sharding bank servers by name.

Conclusion

This design delivers a secure, reliable, and scalable payment system meeting all requirements. SSL/TLS and session keys ensure security, transaction IDs guarantee idempotency, offline queuing enhances usability, and 2PC with timeouts ensures transactional integrity. The use of gRPC and interceptors provides a robust foundation for communication and monitoring.