STRIFE

Venkamsetty Venkata Jahnavi – 2022101118

1. System Overview

Strife is designed as a miniature version of Stripe. It consists of three primary components:

- **Bank Servers:**
  Each bank server represents a separate bank. They load account details (including username, password, and balance) from JSON files. Bank servers process payment preparations, commits, and aborts and provide balance inquiries. Multiple bank servers can be instantiated concurrently, each with a unique name and configuration.
- **Payment Gateway:**
  The payment gateway is the central coordinator between clients and bank servers. It manages client registration, processes payments using a two-phase commit (2PC) protocol, enforces idempotency, and maintains a persistent transaction history. The gateway also supports offline payments by allowing the client to queue payment requests when connectivity is interrupted.
- **Clients:**
  Clients interact with the payment gateway. They register themselves with the gateway, providing bank account details and credentials. After registration, clients can initiate payments, query their balance, view transaction history, or unregister. Communication is secured using mutual TLS (mTLS).

2. Secure Authentication, Authorization, and Logging

2.1 Authentication

- **Mutual TLS (mTLS):**
  All communications (client-to-gateway and gateway-to-bank) are secured using mTLS. Each component loads its own certificate and private key, and verifies the peer’s certificate using a trusted Certificate Authority (CA).
  - **Implementation:**
    - TLS certificates are generated with Subject Alternative Names (SANs) (not relying on the deprecated Common Name field).
    - The gateway uses its server certificate to secure incoming connections, and requires clients to present valid certificates.
    - Clients use their certificates to authenticate to the gateway.
    - Additionally, clients provide a username and password as metadata in gRPC calls; these credentials are validated against pre-loaded or registered users.

2.2 Authorization

- **Role-Based Authorization:**
  Once authenticated, only users with the correct permissions can access sensitive operations. For example, a user can view only their own balance or transaction history.
  - **Implementation:**
    - The gateway's gRPC interceptors (specifically, the `authorizationInterceptor`) examine the metadata attached to each request.
    - For balance or history queries, the interceptor verifies that the username provided in the request matches the authenticated username from the metadata.
    - Unauthorized requests receive a `PermissionDenied` error.

2.3 Logging

- **Verbose Logging via gRPC Interceptors:**
  Detailed logs are generated for every request, response, and error.
  - **Logged Information:**
    - The name of the invoked method (e.g., `ProcessPayment`, `GetBalance`, `GetTransactionHistory`).
    - The full payload of requests and responses.
    - The client certificate subject (for mTLS verification).
    - Any error messages along with their gRPC error codes.
  - **Benefits:**
    - This logging framework provides transparency into the payment flow, making it easier to debug issues, audit transactions, and trace the flow of data between clients, the gateway, and bank servers.

3. Idempotent Payments

3.1 Requirements and Challenges

- **Requirement:**
  Duplicate payment requests (e.g., due to network timeouts or retries) must not result in multiple deductions from the sender's account.
- **Challenge:**
  Simple timestamp-based deduplication is not allowed; the system must use a more scalable approach.

3.2 Implementation Approach

- **Idempotency Key:**
  The client supplies a unique `IdempotencyKey` (a UUID) with every payment request.
  - **Gateway Behavior:**
    - Upon receiving a payment request, the gateway checks an internal concurrent map (using `sync.Map`) to see if the idempotency key already exists.

        - If the key exists, the gateway immediately returns a duplicate response without re-processing the transaction.
        - If not, the key is stored (initially with a “processing” state), and the transaction is executed.
        - After successful commitment, the state is updated so that subsequent duplicate requests are identified.
- **Correctness Proof Sketch:**
    - **Uniqueness:** UUIDs provide a sufficiently high probability of uniqueness.
    - **Atomicity:** The use of concurrent maps ensures that even concurrent duplicate requests are detected reliably.
    - **Effect:** Only one instance of a payment is processed, and retries using the same idempotency key do not cause additional deductions.

4. Offline Payments

4.1 Requirements

- If the client cannot reach the payment gateway (e.g., due to network failures), payment requests must be queued.
- The client must automatically retry pending transactions when connectivity is restored.
- Transactions are persisted to a JSON file so that pending requests survive client restarts.

4.2 Implementation Approach

- **Client-Side Offline Queue:**
  The client maintains an in-memory queue of failed payment requests.
    - Each failed payment is converted into an `OfflineTransaction` structure and stored in a JSON file (`pending_transactions.json`).
    - A background goroutine periodically attempts to re-send queued transactions.
    - Upon successful processing, the transaction is removed from the queue (and the JSON file is updated).
- **Idempotency Interaction:**
  Since the gateway uses an idempotency key, retrying a failed transaction with the same key will not result in duplicate deductions.

5. Two-Phase Commit (2PC) with Timeout

5.1 Requirements

- The payment process must be executed as a distributed transaction.
- It involves two phases: the **Prepare Phase** (voting) and the **Commit Phase** (actual update).
- The transaction must be aborted if any participant (bank server) fails to respond within a configurable timeout.

5.2 Implementation Approach

- **Gateway as Coordinator:**
  The payment gateway initiates the 2PC protocol:

    - **Phase 1 (Prepare):**
        - The gateway sends `PreparePayment` requests to both the sender's and receiver's bank servers.
        - If either bank responds negatively or fails to respond within 5 seconds, the gateway aborts the transaction.
    - **Phase 2 (Commit):**
        - If both banks vote "commit," the gateway sends `CommitPayment` requests to both banks.
        - If a bank fails to commit, the transaction is aborted to avoid partial updates.
- **Timeout Handling:**
  The system uses context with a timeout (set to 5 seconds) to ensure that transactions are aborted if bank responses are delayed.

6. Implementation Details

- **Communication:**
  All inter-component communication is implemented using gRPC with clearly defined service definitions in the proto file.
- **TLS Security:**
  All communications are secured using mutual TLS. Each component loads its own certificate and verifies its peers using a trusted CA certificate.
- **Persistence:**
    - **Bank Servers:** Account details and balances are stored in JSON files.
    - **Gateway:** Transaction history is maintained in a persistent JSON file, and the client-side offline queue is also saved to a JSON file.
- **Fault Tolerance & Scalability:**
    - Concurrency is managed with Go's `sync.Map` and mutexes.
    - The system is designed to recover from failures, with persistent logs and transaction history for audit and recovery purposes.

7. Logging and Monitoring

- **Logging Interceptors:**
  Every gRPC call is intercepted to log:
    - The method name and payload.
    - The outcome (response, error codes, and error messages).
    - Client identification details (extracted from the TLS certificate).
- **Transaction Integrity:**
  Logging is used to verify the 2PC protocol flow and idempotency. In case of network errors, the system logs retries and offline queue updates.

8.2 Running the Tests

- **Automated Unit Tests:**
  Run using `go test -v` in the project directory.

- **Integration Test Script:**
  A shell script (`test.sh`) cleans and builds the project, generates TLS certificates, starts the payment gateway and bank servers, and simulates various client operations (registration, payment, offline retry, balance inquiry, transaction history, and unregister).

Assumptions:

- There is no explicit login phase; login details are sent with each request via metadata and validated by the bank.
- Data is persisted using JSON files, assuming that file operations are reliable and atomic in our test environment.
- Network interruptions are assumed to be transient, allowing offline payments to be retried automatically once connectivity is restored.

9. Conclusion

This implementation of Strife satisfies all assignment requirements:

- **System Components:**
  Clearly separated bank servers, a payment gateway, and client applications.
- **Secure Authentication, Authorization, and Logging:**
  Achieved via mutual TLS, metadata-based credential verification, role-based access control using gRPC interceptors, and detailed logging of all gRPC calls.
- **Idempotent Payments:**
  Implemented using a UUID-based idempotency key that ensures duplicate requests do not lead to multiple deductions.
- **Offline Payments:**
  Clients persist failed transactions to a JSON file and automatically retry them once connectivity is restored.
- **Two-Phase Commit with Timeout:**
  The gateway coordinates transactions using a two-phase commit protocol with a timeout, ensuring consistency even in the presence of network failures.

AI report:

<Question> <filestructure> implement this in go using grpc.