Strife Report

1. Overview

Our system simulates a distributed payment service similar to Stripe. The primary components are:

- **Payment Gateway Server (Coordinator):** Orchestrates payment transactions between clients and bank servers using a two-phase commit (2PC) protocol. It also enforces security (SSL/TLS mutual authentication), authentication/authorization (via gRPC interceptors), and fault tolerance.
- **Bank Servers (Voters):** Each bank (e.g., HDFC, ICICI, Axis, etc.) runs a single instance that maintains account balances (persisted in SQLite) and processes debit and credit operations.
- **Clients:** Users interact with the system via a client application that authenticates, views balances, and initiates transactions. Communication is secured with mutual TLS.

---

2. gRPC Service Definitions for All Interactions

Our system uses Protocol Buffers to define the interactions between clients, the Payment Gateway, and bank servers. These definitions form the contract that each component adheres to.

Key Snippet from payment_gateway.proto:

```
service PaymentGateway {
  rpc RegisterClient(ClientInfo) returns (RegisterResponse) {}
  rpc ProcessPayment(PaymentRequest) returns (PaymentResponse) {}
  rpc ViewBalance(BalanceRequest) returns (BalanceResponse) {}
}
```

And for bank servers:

```
service BankService {
  rpc PrepareTransaction(TransactionRequest) returns
(TransactionResponse) {}
  rpc CommitTransaction(TransactionRequest) returns
(TransactionResponse) {}
```

```
  rpc AbortTransaction(TransactionRequest) returns
(TransactionResponse) {}
  rpc Ping(Empty) returns (PingResponse) {}
  rpc GetBalance(BalanceRequest) returns (BalanceResponse) {}
}
```

These definitions ensure that every component has a clearly defined interface.

---

3. Core Concepts

3.1 Idempotency

Definition:
Idempotency is the property of an operation whereby performing it multiple times produces the same result as performing it once. In the context of payment transactions, this means that if a client retries a transaction (for example, because of a network fault), the system must ensure that the transaction is not processed more than once (i.e., funds are not deducted multiple times).

Implementation in Our System:

- Every transaction is assigned a unique idempotency key (a transaction ID).
- The Payment Gateway maintains a record (persisted in SQLite) of processed transaction IDs along with their status (e.g., "committed").
- Before processing a new transaction, the Payment Gateway checks if the transaction ID is already recorded as committed. If yes, it immediately returns a successful response without reprocessing.
- **Role of Hashing:**
  While our simulation stores the transaction IDs as keys in a Python dictionary (and in the SQLite table), in a production-scale system you might hash these keys for:
    - **Efficient Lookup:** Hashing enables constant-time lookups in hash tables.
    - **Uniform Distribution:** A good hash function distributes keys evenly, reducing collisions in distributed caches or databases.
    - **Security:** In some cases, hashing may help protect sensitive transaction identifiers.
- In our current implementation, the idempotency key (transaction ID) is assumed to be unique and is used directly for comparison. Hashing could be applied if we need to store or index very large volumes of transaction keys.
- **Proof of Correctness (Conceptual):**
- Assume two identical transactions (same transaction ID) are received due to a network fault. The first transaction goes through and is marked "committed" in the dictionary and

the database. When the duplicate arrives, the system checks the transaction ID, finds it committed, and returns the same success response without reprocessing. Thus, no additional funds are deducted, ensuring idempotency.

- **Key Code Snippet:**

  ```
  if transaction_id in processed_transactions and processed_transactions[transaction_id] == "committed":

      return PaymentResponse(success=True, message="Transaction already processed (idempotent).")
  ```

3.2 Two-Phase Commit (2PC)

Definition:
The two-phase commit protocol is a distributed transaction protocol that coordinates all the participants (in our case, the bank servers) to either commit or abort a transaction. It ensures that a transaction is executed atomically across all participants.

Phases:

1. **Prepare Phase:**
   The coordinator (Payment Gateway) sends a "prepare" message to each participant (both the sender's and recipient's bank servers). Each participant checks whether it can commit (e.g., for debit operations, it verifies sufficient funds) and responds with a vote (ready or not ready).

2. **Commit Phase:**
   If all participants respond positively (i.e., "ready") within a configurable timeout (in our code, `TWO_PHASE_TIMEOUT` is 5.0 seconds), the coordinator then sends a "commit" message to all. Otherwise, it sends an "abort" message to roll back any partial changes.

Timeout and Fault Handling:

- Our implementation uses a timeout (`TWO_PHASE_TIMEOUT = 5.0` seconds) for each phase. If a bank server does not respond within this period, the Payment Gateway aborts the transaction.
- This prevents indefinite waiting and ensures that no partial transaction is committed in the presence of network delays or server failures.

---

4. Security: Authentication, Authorization, and Mutual TLS

4.1 Mutual TLS (mTLS)

Purpose:
Mutual TLS secures the communication channel by ensuring both the server and the client authenticate each other using digital certificates. This prevents man-in-the-middle attacks.

Implementation:

Server-Side:
The Payment Gateway server loads its certificate and private key and also loads the CA certificate to verify client certificates. The critical configuration is:

```
with open('certs/server.key', 'rb') as f:
    private_key = f.read()
with open('certs/server.crt', 'rb') as f:
    certificate = f.read()
with open('certs/ca.crt', 'rb') as f:
    client_ca_cert = f.read()  # CA certificate that signed client
certificates

server_credentials = grpc.ssl_server_credentials(
    [(private_key, certificate)],
    root_certificates=client_ca_cert,
    require_client_auth=True  # Enforces mutual TLS
)
server.add_secure_port('[::]:50050', server_credentials)
```

Client-Side:
The client loads its own certificate and private key along with the CA certificate to verify the server's certificate:

```
def get_secure_channel(address):
    with open('certs/ca.crt', 'rb') as f:
        trusted_certs = f.read()
    with open('certs/client.key', 'rb') as f:
        client_key = f.read()
    with open('certs/client.crt', 'rb') as f:
        client_cert = f.read()
    credentials = grpc.ssl_channel_credentials(
```

```
        root_certificates=trusted_certs,
        private_key=client_key,
        certificate_chain=client_cert
    )
    return grpc.secure_channel(address, credentials)
```

Certificate Generation:
New certificates must be generated (if not already available) using OpenSSL:

- Generate CA certificate, then server and client certificates signed by the CA. Terminal commands for this were provided previously and are documented in README.md .

4.2 Authorization

Purpose:
Role-based authorization ensures that only authenticated clients with proper roles can access and perform sensitive operations.

Implementation:

- **User Data:**
  Dummy user data is stored in a JSON file (`users.json`), which includes credentials, bank, role, and account number.

- **Registration:**
  Clients call `RegisterClient` with their credentials. The Payment Gateway validates these credentials and issues an authentication token.

gRPC Interceptor (AuthInterceptor):
A dedicated interceptor examines metadata on every RPC (except registration). It checks for the presence and validity of the `client-id` and `auth-token` metadata and verifies that the client has the required role (in our case, "client").

---

5. Fault Tolerance and State Persistence with SQLite

5.1 Payment Gateway State Persistence

Why SQLite?

- **Fault Tolerance:**
  To prevent loss of state during crashes, we persist critical state (processed transactions) in a SQLite database (`gateway_state.db`).
- **State Recovery:**
  On startup, the Payment Gateway loads previously processed transactions, ensuring idempotency even across restarts.
- **Implementation:**
  Functions such as `init_db()`, `load_state_from_db()`, and `update_transaction_in_db()` are used to manage this persistent state.

5.2 Bank Server State Persistence

Why Persistence in Bank Servers?

- **Consistency:**
  Bank servers maintain account balances, which must not reset after a crash. We use SQLite databases (one per bank, e.g., `bank_HDFC.db`) to persist account balances.
- **Implementation:**
  On startup, a bank server checks if its database exists:
  - If yes, it loads current balances from the database.
  - If not, it loads initial data from `users.json` and then saves it.
  - After each transaction commit (debit or credit), the bank server updates the database with the new balances.

---

6. Logging

Logging Configuration

We use Python's logging module to capture detailed logs for debugging and monitoring. Logs are written both to the console and to a dedicated file (`logs/server.log`).

Key logged information includes:

- **Method Name:** The gRPC method being invoked (e.g., ProcessPayment).
- **Client ID:** Extracted from metadata.

- **Transaction Amount:** Included in the request details.
- **Errors/Exceptions:** Any errors encountered during processing are logged with error codes and messages.
- **Retry Events:** When offline payments or retries occur, these events are logged.

---

7. Failure Handling: Offline Payments and 2PC Timeouts

Offline Payments and Their Failure Handling Mechanism

Offline payments address the scenario where the client is unable to reach the Payment Gateway—whether due to network faults or the gateway being down. In our design, the client handles offline payments using the following approach:

- **Local Queuing:**
  When a client initiates a transaction and the Payment Gateway is unreachable (for example, the gRPC call returns a UNAVAILABLE error), the client catches this exception and enqueues the payment request in an offline queue. This prevents the transaction from being lost and allows the client to continue operating locally.

```
except grpc.RpcError as e:
    if e.code() == grpc.StatusCode.UNAVAILABLE:
        print(f"[Client {client_id}] Payment gateway is down, queuing transaction for retry.")
        offline_queue.append(payment_request)
```

- **Periodic Retry:**
  A background thread runs periodically (every 5 seconds) to process any transactions in the offline queue. This thread attempts to resend the queued transactions. If the gateway becomes available, the transactions are retried. If they succeed, the offline queue is cleared for those items; if not, they remain queued.

```
def retry_offline_payments():
    while True:
        if offline_queue:
            # Attempt to resend offline transactions
            response = stub.ProcessPayment(payment_request, metadata=metadata)
        time.sleep(5)
```

- **Failure Handling:**
    - **Network Faults:** If the Payment Gateway is down, the retry mechanism continues to requeue the transaction until the gateway is back up.
    - **Idempotency:** Even if a transaction is retried multiple times due to repeated network faults, the use of a unique transaction ID ensures that it is processed only once.
    - **User Notification:** The client prints friendly messages (e.g., "Payment gateway is down. Please try again later") so the user understands that the transaction is pending until connectivity is restored.

This design ensures that even if the gateway is temporarily unreachable, the client does not lose any payment requests, and once connectivity is restored, those pending transactions are processed without causing duplicate operations.

---

ii) Failure Handling Mechanisms in Two-Phase Commit (2PC)

The two-phase commit (2PC) protocol in our Payment Gateway coordinates the distributed transaction between the sender's and recipient's bank servers. The failure handling mechanisms for 2PC are critical to ensure atomicity—either both parts of the transaction commit or neither does.

- **Phase 1: Prepare Phase**
    - The Payment Gateway sends a `PrepareTransaction` RPC to both bank servers.
    - Each bank server checks if it can process its part of the transaction (e.g., for debits, it verifies if the sender has sufficient funds).
    - **Timeout:** A global timeout (`TWO_PHASE_TIMEOUT = 5.0` seconds) is set for these RPC calls. If any bank server fails to respond within this period, the coordinator interprets it as a failure.
      `prep_sender = sender_stub.PrepareTransaction(txn_request, timeout=TWO_PHASE_TIMEOUT)`
    - `prep_recipient = recipient_stub.PrepareTransaction(txn_request, timeout=TWO_PHASE_TIMEOUT)`
    - `if not (prep_sender.ready and prep_recipient.ready):`
    - `    # Abort transaction if either response is not ready`
- **Phase 2: Commit Phase**
    - If both banks respond positively during the prepare phase, the Payment Gateway sends `CommitTransaction` RPCs.

    - Again, if a bank server does not respond within the timeout, or if its commit fails, the coordinator will abort the transaction.

```
commit_sender = sender_stub.CommitTransaction(txn_request,
timeout=TWO_PHASE_TIMEOUT)
```

- `commit_recipient = recipient_stub.CommitTransaction(txn_request, timeout=TWO_PHASE_TIMEOUT)`
- `if not (commit_sender.ready and commit_recipient.ready):`
- `    # Abort the transaction if commit phase fails`
- **Abort Mechanism:**
    - If any part of the transaction fails (either during the prepare or commit phase), the Payment Gateway sends an `AbortTransaction` RPC to both banks to roll back any preliminary changes.
    - The transaction is marked as “aborted” in the persistent store (SQLite) to ensure idempotency.
      `sender_stub.AbortTransaction(txn_request, timeout=TWO_PHASE_TIMEOUT)`
    - `recipient_stub.AbortTransaction(txn_request, timeout=TWO_PHASE_TIMEOUT)`
    - `processed_transactions[transaction_id] = "aborted"`
    - `update_transaction_in_db(db_conn, transaction_id, "aborted")`

This design ensures that if any participant (bank server) fails to respond or explicitly indicates it cannot process the transaction, the entire operation is aborted. The use of timeouts prevents the system from hanging indefinitely due to network delays or server crashes, and the abort mechanism maintains the consistency of the distributed transaction.

In summary, our system’s failure handling for offline payments and 2PC ensures that:

- Offline transactions are safely queued and retried once the gateway becomes available.
- The 2PC protocol uses timeouts to abort transactions if any participant fails to respond, ensuring atomicity.
- Idempotency guarantees that retried transactions are not processed multiple times.

This comprehensive failure handling approach enhances both the robustness and reliability of our distributed payment service.

8. System Components and CLI Interface

- **Multiple Bank Servers:**
  Our system includes six bank servers—HDFC, ICICI, Axis, Canara, BankOfIndia, and Kotak. Each bank server is responsible for maintaining its own account balances (persisted in SQLite) and handling debit/credit operations. The bank server loads only the accounts relevant to its bank from a JSON file (users.json), which contains client details like username, password, and a unique 14-digit account number.

- **Payment Gateway Server:**
  The Payment Gateway acts as the transaction coordinator. It does not maintain account balances but uses a two-phase commit protocol to ensure atomic transactions. It enforces authentication and role-based authorization via gRPC interceptors and persists transaction statuses in SQLite for fault tolerance.

Clients:
Clients are end-users whose credentials (username, password), bank affiliation, and unique 14-digit account number are stored in the JSON file. They interact with the system via a CLI. For example, a client can run:

```
python3 client.py --client_id client1 --username user1 --password pass1 --bank HDFC
```

- Once running, the client enters interactive mode where they can:
  - **view:** Check their current balance.
  - **transfer <amount> <recipient_account_number> [transaction_id]:** Initiate a funds transfer to another client by specifying the amount and a valid 14-digit recipient account number.
  - **exit:** End the session.

This CLI interface allows multiple clients to perform authentication, view balances, and initiate secure transactions concurrently, while the Payment Gateway and Bank Servers ensure that all operations are processed reliably and securely.

---

9. Conclusion

Our system design implements a robust payment service with the following features:

- **gRPC Communication:**
  All interactions between the Payment Gateway, Bank Servers, and Clients are defined

in a clear proto file.

- **Mutual TLS Security:**
  Both server and client certificates are used for authentication at the transport layer. The Payment Gateway is configured with its certificate and key, and clients load their certificate and key along with the CA certificate.

- **Authentication and Authorization:**
  Clients authenticate with username and password, and are issued an auth token. Authorization is enforced using gRPC interceptors that verify metadata and role information.

- **Idempotency:**
  Each transaction uses a unique idempotency key to ensure that retries do not result in duplicate processing. Our correctness proof is based on the fact that once a transaction is marked "committed" in our persistent store, any subsequent request with the same key will be recognized as already processed.

- **Two-Phase Commit:**
  Our Payment Gateway coordinates debit and credit operations across bank servers using a two-phase commit protocol. A configurable timeout (`TWO_PHASE_TIMEOUT = 5.0 seconds`) ensures that if any part of the protocol fails, the transaction is aborted across all participants.

- **Fault Tolerance and Persistence:**
  The Payment Gateway persists transaction statuses in SQLite to recover state after crashes. Similarly, each Bank Server persists account balances so that state is maintained across restarts.

- **Logging:**
  Verbose logging is implemented via gRPC interceptors, with logs written both to the console and to a separate log file (`logs/server.log`). This logging captures method names, client IDs, transaction amounts, errors, and retry events, aiding in debugging and system monitoring.