REPORT

Instant Payment Gateway

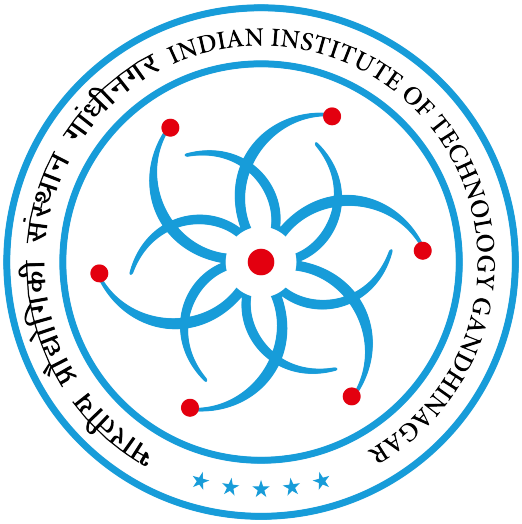
PARALLEL AND DISTRIBUTED SYSTEMS

CS 616

Authors:

Aman Kumar Gupta
Chirag Modi
Subhrajit Das

May 10, 2024
Sem-II, 2023-2024



Contents

Introduction

Instant Transaction Gateway (ITG) provides instant transaction of payment between any two payment parties. With the simplified interface, parties of different banks just use a unique digital payment ID to transact the funds, without the requirement of account number, IFSC codes etc.

Stakeholders Involved

- **User Interface Provider**: Interface provider for the parties collecting the Digital Payment ID of the payer and payee, and security pin from the payer.
- **Digital Payment ID Resolver**: Resolves the unique digital payment id collected from the payee/payer with the corresponding bank account details. Communicates with the Transaction Processing Gateway.
- **Bank**: Respective Remitter/Beneficiary Bank Authority responsible for handling bank transactions of the individuals.
- **Transaction Processing Gateway**: Provides mechanisms for verifying payments enabling communications between different stakeholders.

1 Use Case Diagram: Payment System

Here's a Use Case Diagram of the Instant-Payment-Gateway system: Figure 1.

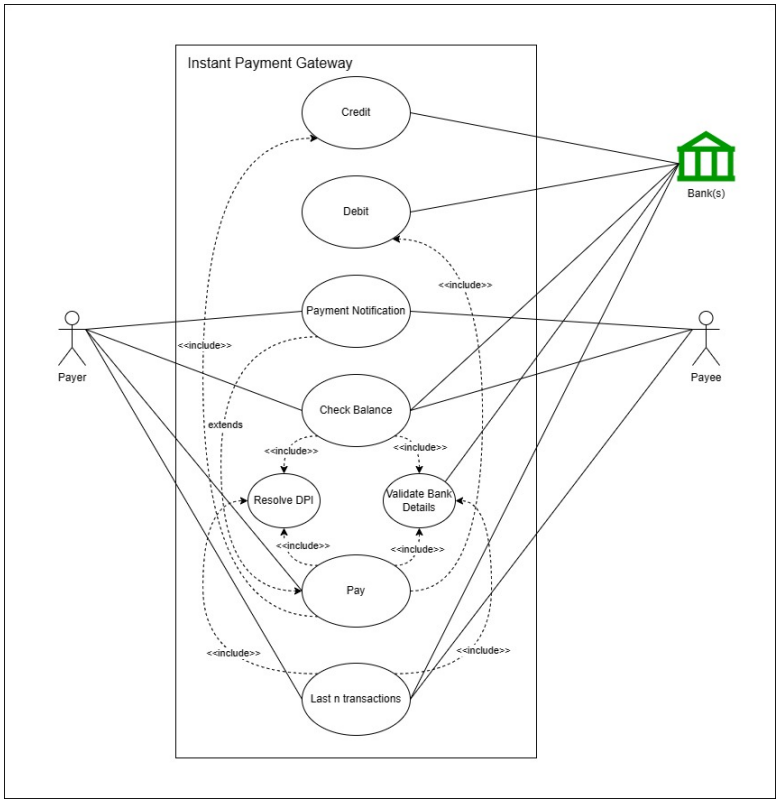


Figure 1: Use Case Diagram: Instant-Payment-Gateway

Actors:

- Payer: The entity initiating a payment transaction (e.g., a customer, individual).
- Payee: The entity receiving a payment (e.g., a merchant, individual).
- Bank(s): The financial institution(s) responsible for processing the payment.

Use Cases:

- Check Balance: Allows payers and payees to verify their available account funds.
- Pay: The primary use case where the payer initiates a payment.
- Resolve DPI: Addresses potential issues resolving the Digital Payment ID.
- Validate Bank Details: Ensures the payer's banking information is correct for transaction processing.
- Debit: The process of deducting funds from the payer's account.
- Credit: The process of depositing funds into the payee's account.
- Payment Notification: Confirms a successful transaction to both the payer (debit) and the payee (credit).
- Last n Transactions: Provides the payer with a view of recent transaction history.

Use Case Relationships:

- Includes: Denotes a mandatory dependency between use cases.
- The "Check Balance", "Pay", and "Last n Transactions" use cases include the "Resolve DPI" and "Validate Bank Details" use cases.
- The "Debit" and "Credit" use cases are included within the "Pay" use case.

Actor Interactions:

- Payer: Engages with "Check Balance", "Pay", "Last n Transactions", and receives "Payment Notification" (debits).
- Payee: Interacts with "Check Balance" and receives "Payment Notification" (credits).
- Bank(s): Facilitates "Debit", "Credit", "Check Balance", and "Validate Bank Details" processes.

2 Sequence Diagram

Here's a basic sequence diagram listing the possible sequences that can take place in case of successful Debit-Credit in the Instant-Payment-Gateway system, Figure 2.

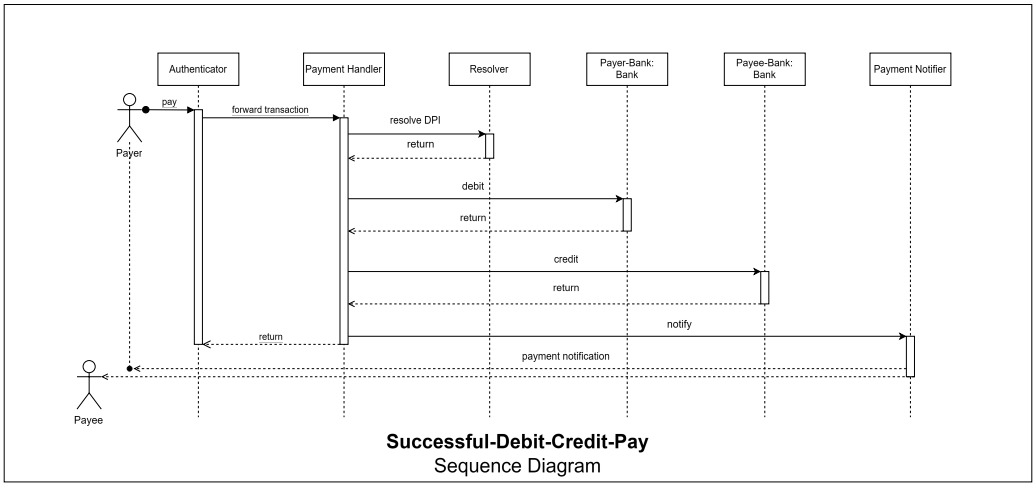


Figure 2: Sequence Diagram - Successful Debit-Credit

The components involved in the sequence diagram are as follows:

- **Authenticator:** Responsible for authentication of valid users.
- **Payment Handler:** Coordinates with Resolver, Banks, Schedulers, and Notifier to perform transactions.
- **Resolver:** Takes virtual payment ID and provides corresponding bank details of the payer and the payee.
- **Banks:** Entities beyond the Transaction Processing Gateway meant to check account status and perform debit and credit operations.
- **Scheduler:** Tracks failed transactions and ensures reversal of partially performed transactions.
- **Notifier:** Sends corresponding messages to the payer and payee after transaction completion.

The joint operation of these components is an important aspect of the component sequence diagram.

Transaction Process

After authentication and resolving, the payment handler initiates the debit command to the payer's bank. This request is retried three times if no success or failure is returned.

Debit Process

- If successful, the payment handler proceeds to credit the payee's bank account.
- If the debit is unsuccessful, the notifier reports the failure to the payer.
- In case of a timeout, the scheduler is notified to reverse the debit transaction.

Credit Process

- On successful debit, the payment handler attempts to credit the payee's bank account, with multiple retry attempts.
- Successful credit results in notifications to both payer and payee.
- If credit fails:
  - The scheduler is prompted to reverse the debit transaction if the credit was unsuccessful.
  - If there was a timeout, both debit and credit transactions are reversed, with the notifier reporting to the payer.

3 Architecture Diagram

Here's the architecture diagram for the Instant-Payment-Gateway, Figure 3.

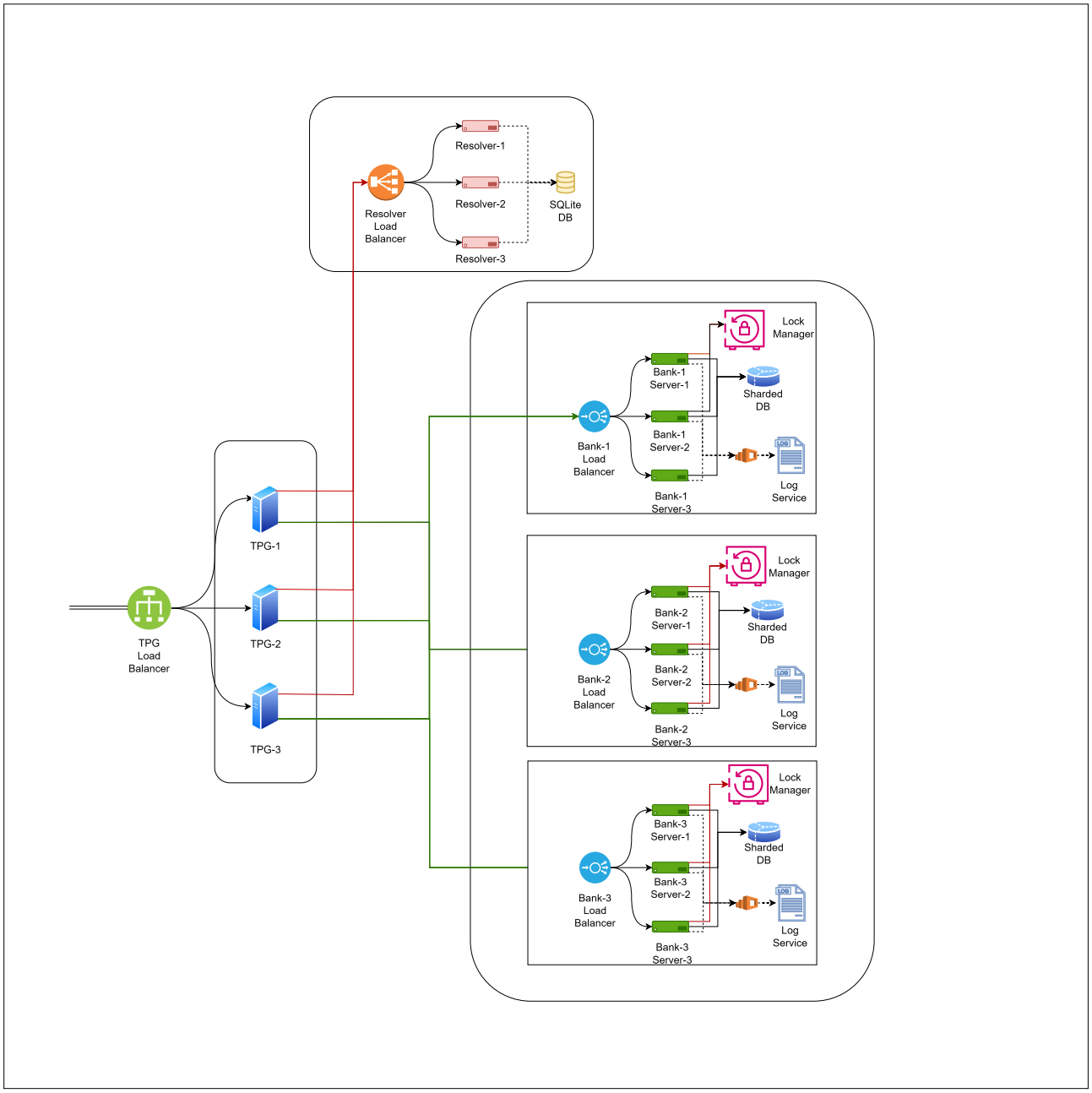


Figure 3: Architecture Diagram : Instant-Payment-Gateway

The architecture of the Instant-Payment-Gateway consists of the following components:

- **TPG Load Balancer:** Responsible for handling the load received from the users. It tries to distribute the load evenly among the TPG servers.
- **TPG**(Transaction Processing Gateway): Mainly entity for handling all the transactions. It basically communicates with the Resolver Load Balancer and Bank Load Balancer components to facilitate transactions.
- **Resolver Load Balancer:** Responsible for handling the load received from the TPG servers. It tries to distribute the load evenly among the Resolver servers.
- **Resolver:** Resolves the Digital Payment ID with Bank Account Details querying from a SQLite file.
- **Bank Load Balancer:** Responsible for handling the load received from the TPG servers. It tries to distribute the load evenly among the Bank servers.

- **Bank:** Bank Servers are mainly responsible for processing the Debit and Credit requests. Multiple instances of a bank server use a sharded DB, along with elastic search into Log service for facilitating failure scenarios. Additionally, **mutual exclusion** of database services among bank server instances is ensured through a **Coordinator-based approach**.

4 Working Mechanism

4.1 Resolving Payment IDs

After initiation the transaction from the user, the TPG's payment handler service first contacts the resolver to resolve the bank details from the payment IDs. The resolver queries the payment ID(s) into its SQLite DB to find bank details (Account number, Holder Name, IFSC Code) if they exist and returns them to the TPG payment handler. These calls are routed to the resolver-loadbalancer which is responsible to transferring the request to one of the running instance of the resolver. Ref: Figure 4.

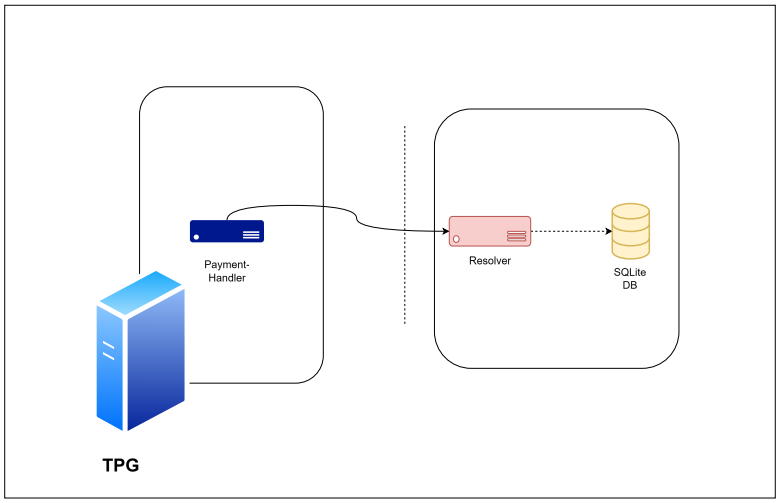


Figure 4: Resolving Bank-Details from Payment-ID

4.2 Debit Request

After getting a valid response from the Resolver, the payment handler of TPG initiates a debit request to the particular bank server based on its IFSC code and waits for a debit. If debit is successful after certain retrial attempts, it moves on to credit. Ref: Figure 5.

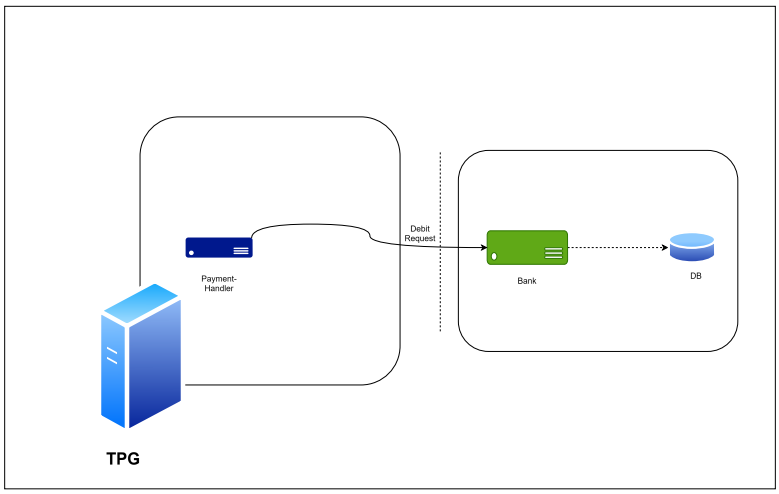


Figure 5: Debit Request

4.3 Failure of Debit

In case a debit fails(with retries), the payment handler initiates a reversal request by appending it to the Queue (file-based). The scheduler service of TPG is a process running in background that kicks in after a certain time interval, reads form the file based queue and tried to reverse the transaction through the bank server. This reverse scheduler tried to revert back the transaction until it specifically get a success message. This implementation gives us an approach where the reversal of the transactions is guarenteed. Ref: Figure 6.

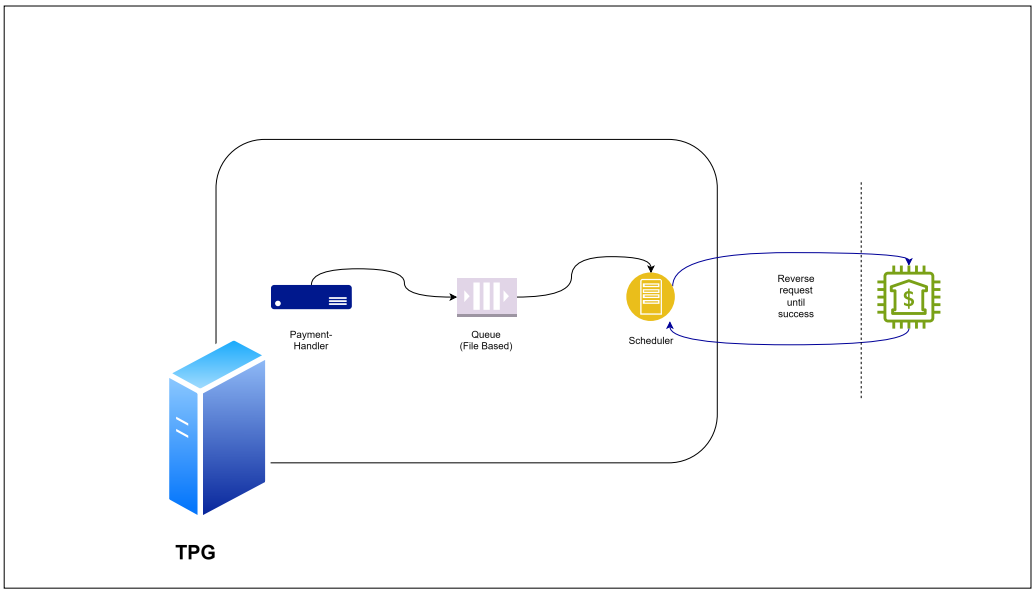


Figure 6: Debit Failure Handling

4.4 Credit Request

In the case of a successful debit response, the payment handler communicates with the respective bank server to process the credit to Payee's bank account; upon successful communication, the payment handler acknowledges the successful transaction to both parties. Meanwhile, the bank server processes the pending credit requests in batch. The credit calls are accumulated in the bank side. Later, based on lock availability for that particular user, it updates pending credits in a batch in the database and updates it to the log service it uses. This approach gives us a guarantee that we will be able to make count of every credit request possible after a certain time of delay.

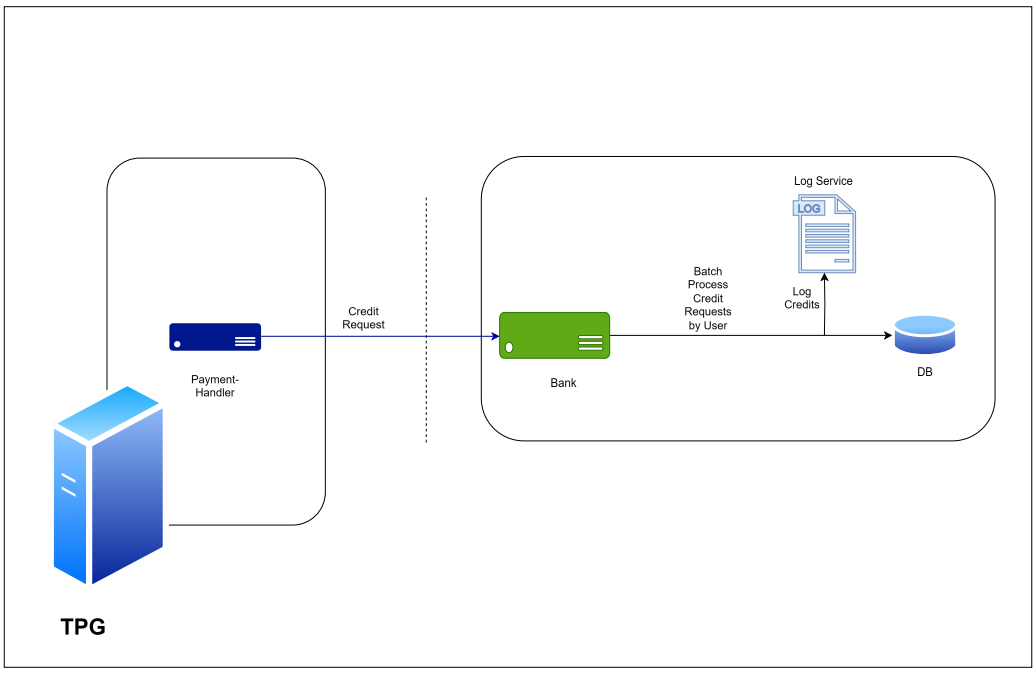


Figure 7: Credit Request

4.5 Mutual-Exclusion among Bank Server Instances

We are following a centralized locking mechanism for the mutual-exclusion. We are giving locks based on the account number. This centralized algorithm is comparatively easy to implement when put aside to the quorum based algorithms.

4.6 Data Sharding

The database has been shared based on the account number. A modulo-n operation gives us a number from 0 to n-1. Based on this remainder, a particular database is chosen and a call to the respective database is sent. I our case, we have done a modulo-3 operation and divided the database into three portions. We had this as an intuitive approach that there will be lesser number of connections to each database and we will be able to handle more connections at the bank server end without having to wait. But with our setup, we we not able to see any significant difference in our operations. This might be because of the reason that we are processing very less number of requests compared to what the databases are able to handle. We also have a bottleneck at acquiring locks that is slowing us further down.

4.7 Elastic Search

For retrial and reversal of debit requests and updation of credit requests, the bank server uses Elastic Search. To avoid invalid debits or any other odd situation, it confirms with the log it maintained through Elastic search. We used elastic search as it uses a reverse index technique and parallel computing, leading to theoretical O(1) complexity, compared to file-based, which theoretically leads to O(n) complexity.

5 Tech-Stack and Deployment

For deployment, we utilized Docker Containerized Micro-services. Each component, along with its instances (if any), such as TPG, TPG Load-balancer, Bank, Bank Load-balancer, Bank DB, Log Service, Resolver, and Resolver Load-balancer are containerized using Docker in a single server. Go gRPC is used to communicate among the servers, using multiple channels to withstand heavy load. (Default gRPC could not withstand concurrent requests(see Statistics section 6); to tackle the issue, we have incorporated multiple channels in a round-robin fashion).
Here's the repository for the Instant-Payment-Gateway: github link

5.1 Server Configuration

The server configuration includes:

- **Processor:** Intel(R) Xeon(R) E-2314 CPU @ 2.80GHz, 4 Threads.

5.2 Tools Used

We employed the following tools for this project:

- **Programming Language:** Go
- **Go gRPC, protobuf:** For communication between services using RPC calls
- **Docker:** For containerized microservices
- **Nginx:** For Load Balancing
- **ELK Stack:** For Elastic Search in Sharded DB
- **MySQL:** For Bank DB
- **SQLite:** For Resolver mappings
- **wrk, wrk2:** For HTTP Benchmarking

Table 1: Performance Results for Debit-Credit Stub Based Approach

Nodes	Connections	2m testing Success?	Requests/s	Avg. Latency (ms)	Stdev	Max	P/M Stdev (%)
1	4	Yes	216.80	18.97	12.75	78.86	
2	4	Yes	293.62	15.13	11.72	201.35	
2	20	Yes	328.56	71.63	85363.0	1.34s	
2	100	Yes	343.12	320.90	271.71	2s	
2	500	Yes	339.20	605.70	423.06	2s	
2	1000	No	Nil	Nil	Nil	Nil	

6 Statistics

Below we have shown the performance of the IPG. See figure 1 and 2

Table 2: Performance Results of the Single Mutex Lock Based Approach

Nodes	Connections	2m testing Success?	Requests/s	Avg. Latency (ms)	Stdev	Max	P/M Stdev (%)	Avg. Req/S	Stdev
3	10	Yes	18.57	444.47	419.77	1.8	80.5	8.26	7.82
3	20	Yes	45.72	448.33	514.2	2	79.9	17.73	13.82
3	25	Yes	35.77	364.73	515.79	2	Nil	Nil	Nil
3	30	No	Nil	Nil	Nil	Nil	Nil	Nil	Nil

7 Conclusion

Based on our extensive testing, we evaluated our system under various scenarios by adjusting parameters such as the number of bank server nodes and resolver nodes, while employing multiple threads to simulate simultaneous client requests. Our analysis revealed a mix of outcomes: in certain instances, the system exhibited stability with all servers operational, while in others, some servers experienced downtime.

To address these challenges, we iteratively refined our codebase, employing different approaches to pinpoint and resolve bottlenecks. This systematic approach enabled us to identify and rectify issues, leading to improvements in system reliability and performance.