gRPC Assignment

Deadline: March 6 2025, 11:59 PM

Introduction

In this assignment, you will construct distributed solutions to certain problems and implement them using the gRPC(gRPC Remote Procedure Calls) framework.

gRPC is an open-source remote procedure call (RPC) framework developed by Google that uses HTTP/2 for transport and Protocol Buffers for interface definition. It enables efficient, strongly typed communication between client and server applications, supporting features like authentication, bidirectional streaming, and flow control. GRPC simplifies building scalable, distributed systems and microservices by providing a unified communication mechanism.

You can use either **Python** or **Go** language for this assignment.

- Installation:
  - https://grpc.io/blog/installation/
- Documentation:
  - General: https://grpc.io/docs/
  - Python: https://grpc.io/docs/languages/python/
  - Go: https://grpc.io/docs/languages/go/basics/
  - Protobuf: https://protobuf.dev/overview/
- Resources:
  - https://github.com/grpc-ecosystem/awesome-grpc

Important Points

- This assignment will have **manual evaluations**, where you will give a demo of your solutions, and then questions will be asked about:
  - Concepts tested in the assignment.(gRPC, protobuf, streaming, authentication, load balancing, interceptors among others)
  - Implementations for the given problem.
- There are 3 compulsory problems in this assignment, which are Problems **1, 2 and 3**. Along with this, there is also a Bonus, Problem **4** which you can attempt to recover any marks lost **within this assignment**.
- **Do not use existing code bases** from anywhere as we will be running plagiarism checkers. In case you are caught, you will be penalized.
- **You can take your own reasonable assumptions where things aren't mentioned. Mention any such assumption with the reason in your Report.**

1 Balance Load? (20 Points)

You have to implement a basic client-server system where the focus will be on load-balancing.

Load balancing is essentially the process of distributing computational load between multiple computers or processors. This is done to avoid overloading at a single server while other servers are idle. There are multiple options when you want to implement a load balancer; we will use **look-aside load balancing** (also known as *one-armed balancing*).

1.1 Look-Aside Load Balancing

In this type of load balancing, every client asks the load balancing server(let's call it LB server), which server(s) they should use. The LB server is thus easily able to implement any complex load balancing it wants. Clients query the lookaside LB and the LB responds with best server(s) to use. The heavy lifting of keeping server state and implementation of LB algorithm is consolidated in the lookaside LB. gRPC defines a protocol for communication between client and LB using this model.

For more details about load balancing with gRPC, you may refer the official documentation on their website: https://grpc.io/blog/grpc-load-balancing/.

1.2 Load Balancing Policies

3 different load balancing strategies have to be implemented in the LB server:

- **Pick First:** Selects the first available server from the list.
- **Round Robin:** Distributes requests cyclically among available servers.
- **Least Load:** Route requests to the server with the lowest current load (Use few servers with CPU heavy tasks for testing).

1.3 Components

1.3.1 Servers

Servers handle client requests and perform computational tasks. They register with the LB server and periodically report their status (load, availability, etc.). For dynamic server discovery at the LB server, existing service discovery clients such as **etcd** can be used to maintain an updated list of available backend servers.

1.3.2 Clients

Clients send computational requests and query the LB server to determine which backend server to use.

1.3.3 Load Balancer

The LB server receives client queries and responds with the most suitable backend server(s) based on the chosen balancing policy. The discovery of the LB server itself can be managed using service discovery mechanisms like **DNS-based service discovery, etcd, or Consul**, ensuring that clients always have access to an up-to-date LB endpoint.

1.4 Tasks

1. Define appropriate services and RPCs using gRPC. Explain your choices in the report.
2. Implement a Look-Aside Load Balancer server that:
    - Maintains a list of available servers and their load status.

- Implements load balancing policies mentioned in Section 1.2.
- Responds to client queries with the best available server.

3. Implement the backend servers that:
   - Register with the LB server.
   - Periodically report their load status.
   - Handle client requests.
4. Implement the client that:
   - Queries the LB server for the best server.
   - Sends computational requests to the selected server.
5. Scale test the system using multiple clients and servers.
   - Simulate high request loads with multiple concurrent clients.
   - Evaluate the performance of three load balancing policies: Pick First, Round Robin, and Least Load.
   - Measure response time, server utilization, and system throughput.
   - Analyze how effectively the system distributes load under different scenarios.

1.5 Marks Distribution

1.5.1 Load Balancer Functionality (6 Points)

- Maintaining a **list of available servers**.
- Implementing **server registration and load reporting**.
- Responding to client queries with a **valid server selection**.

1.5.2 Load Balancing Policies (9 Points)

- **Pick First Policy** correctly implemented.
- **Round Robin Policy** cycles requests evenly.
- **Least Load Policy** selects the least-loaded server.

1.5.3 Scale Testing and Analysis (5 Points)

- Successfully simulating a **multi-client, multi-server** test case.
- Measuring and analyzing **response time, throughput, and load distribution**.
- Provide brief **observations/conclusions** based on test results(provide graphs wherever necessary).
- An optimal load cab be to spawn 10 - 15 servers, along with an order of 100 clients(bombarding concurrent requests to the servers). Show how the LB server is managing this load. **Note** that this is just a suggestive number, you can increase or decrease this as per the support on you local systems.

Note: The approach taken for scale testing, along with any necessary scripts or test code, must be submitted in the repository. For evaluations, you can setup a dummy client-server setup to show that the load-balancer is functioning properly. Also, keep the load-balancing property as a CLI argument for ease of testing.

2 MapReduce (30 points)

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key [1].

In this problem, you need to implement a **distributed MapReduce** that will call applications **Map** and **Reduce** and handles reading and writing of files, along with a **master** process which will give tasks to the workers.

Note: For the scope of this problem, you do not need to handle master or worker failures.

2.1 Components

2.1.1 Master

The master node is responsible for assigning **Map** and **Reduce** tasks to the worker nodes. It will also monitor the progress of the different tasks along with handling worker failures. The entire data flow across the **Map** and **Reduce** tasks is also handled by the **Master**.

2.1.2 Workers

Worker nodes execute tasks assigned by the master. Each worker can be a Mapper or a Reducer. The worker reads input data and applies the Map function. It will store intermediate key/value pairs and keep the master informed. It will also fetch assigned data and apply the Reduce function, finally writing the output.

Note: We strongly recommend you to read the paper [1] for better understanding of this system. We are only aiming to replicate a small version of this system with a few basic tasks.

2.2 Tasks

The system will be having a **single Master** and every task would be parallelized between the **Workers**. The number of mapper and reducer tasks will be specified through the CLI while running the code. You will be implementing the following **MapReduce** tasks:

2.2.1 Word Count

Each Mapper reads an input file and produces `(word, count)` pairs. The Reducer aggregates counts for the same word and outputs the final word count.

2.2.2 Inverted Index

Inverted indices are widely used in computer science, and are particularly useful in document searching. Broadly speaking, an inverted index is a map from interesting facts about the underlying data, to the original location of that data. For example, in the context of search, it can be a map from keywords to documents that contain those words. For this task, each **Mapper** reads a file and produces `(word, filename)` pairs. The **Reducer** aggregates all filenames where the word appears and outputs `(word, [list of filenames])`. For more details on this task refer to Section 2.3 of [1].

2.2.3 Distributed MapReduce

You will be using multiple worker threads for the scope of this assignment. This will ensure that the computational tasks are run in parallel. To coordinate the parallel execution of tasks, you should use a separate master thread, which hands out work to the workers and waits for them to finish. The master

should only communicate with the workers via gRPC. We leave the remaining design decisions on the students(document any major design decisions in your Report).

2.2.4 Important Notes:

- You will be using gRPC for defining various tasks across the scope of this problem. Properly document the different services and RPC usage in your Report.
- The map phase should divide the intermediate keys into buckets for `numReduce` reduce tasks, where `numReduce` is the number of reduce tasks.
- Each Mapper should create `numReduce` intermediate files for consumption by the Reduce tasks.
- The worker implementation should put the output of the `X`'th reduce task in separate intermediate output files.
- The intermediate output file should contain one line per Reduce function output in an appropriate **key/value** format. You can keep these temporary file names unique so that you don't have to keep removing them during every run.
- The worker should put intermediate Map output in files in the current directory, where it can later read them as input to Reduce tasks.
- When a job is completed, the worker processes should exit.

2.3 I/O Format

There is no specific I/O format for this problem. You can flexibly employ different output formats for the different tasks and document it in your README. However, there are certain things you should keep in mind. The **numReduce** tasks should be kept as a CLI argument. For the intermediate tasks, we have already given specific notes, those need to be followed. We will be providing you with certain input files on which you can document your outputs and mention in the Report.

Acknowledgements

This problem is inspired originally from MIT's 6.824 course, along with references taken from Princeton's COS 418 course. We would like to sincerely thank the authors of this problem (Frans Kaashoek, Robert Morris, and Nickolai Zeldovich).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04), 2004. Available: `https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf`

3 Strife (50 Points)

Your task is to build a miniature version of Stripe, which is a payment gateway that interfaces with various bank servers to manage transactions.

3.1 System Components (10 points)

- **Bank Servers**
  - Implement a bank server that represents an individual bank.
  - Multiple bank servers will be instantiated, each with a unique name.
- **Clients**
  - Implement a client that interacts with the payment gateway.
  - Clients must register with the gateway and provide bank account details.
  - Multiple clients will be run during testing.
- **Payment Gateway**
  - The central entity that interfaces between clients and bank servers.
  - Handles transactions between clients and banks.

3.2 Secure Authentication, Authorization, and Logging (10 points)

3.2.1 Authentication

A secure and robust payment service requires strict authentication and authorization mechanisms to ensure only authorized users can access the system and perform transactions. In a Stripe-like service, clients must authenticate before initiating payments. The system will employ **SSL/TLS mutual authentication** for communication security and implement **role-based authorization** to restrict access to sensitive operations.

- Clients must authenticate themselves with the Stripe service before making transactions.
- Clients provide their **username** and **password**, which are validated against stored credentials (You can load some users at the server side before hand and these will serve as existing users in the system. You can also choose to implement an admin which can add users based on details such as their name, account no, balance etc which is relevant for this problem).
- You can keep a setup file (csv, json, yaml) with dummy data for simulation purposes. The bank servers would need to read from that file and load the information relevant *only* to that bank. This is if you are not implementing an admin who can add users manually. We suggest this method for simplicity.
- Communication between the client and server is secured using **SSL/TLS** to prevent eavesdropping. You can use certificates for this.
- **gRPC Configuration:** The gateway server will configure SSL/TLS credentials using its own server certificate and private key. Similarly, the client will verify the server's certificate using the trusted Certificate Authority (CA).
- The specific details about how you want to perform authentication(using appropriate certificate choice, implementation details) is upto you.(Document your approach in the Report).

You can refer to the official documentation of gRPC for authentication: https://grpc.io/docs/guides/auth/

3.2.2 Authorization

Authorization ensures that only authenticated users with the correct permissions can access certain resources.

For example, a client can:

- View their balance.
- Initiate transactions (only within available funds).
- View their transaction history. (Optional)

You are required to implement **authorization** using gRPC interceptors so that only a client with appropriate permissions can access their information.

3.2.3 Logging

Logging is essential for monitoring the health of the system and troubleshooting potential issues. In companies, the logs are used for recovery as well but for the scope of this assignment we are just focussing on capturing logs and **will not be** utilizing them for any task. gRPC interceptors provide an ideal mechanism to log every request, response, and error, which is important for both debugging and other tasks as well.

- **Verbose Logging with gRPC Interceptors:** To capture detailed logs for each gRPC request, you need to implement a logging interceptor. This interceptor will log the incoming requests, the server's response, the status of the response, and any errors that occur. This logging mechanism will provide full transparency into the payment flow, helping to track down any issues in real-time.
- **Logging Information:** The logs should include the following information(you can change as you feel necessary):
  - The transaction amount.
  - Client identification information (e.g., client ID, IP address).
  - Method name (for e.g., `ProcessPayment`).
  - Any errors or exceptions, including error codes and messages.
- **Transaction Integrity and Retry Logging:** In case of network failures or timeouts during payment processing, retries may be attempted. The interceptor will also log when a transaction is retried, ensuring that the payment gateway correctly handles idempotency.

3.3 Idempotent Payments (10 points)

When you're designing some systems, changes are inherently idempotent e.g. updating user information repeatedly yields the same result (think of any PUT request). In other cases however, this might not be true – consider adding courses in registration. If you do it twice, there will either be an error or you will end up with two versions of the course... unless you set up idempotent additions.

That is what you have to do here, and it will be a much more critical feature than for a course registration system. You have to make sure that if the same transaction gets retried (perhaps the earlier one failed to return in the specified timeout and the server resent the transaction), the application of these changes is idempotent. So if the transaction is of Rs. 100, the system shouldn't accidentally deduct Rs. 200 from the sender's account.

There are various ways of doing this. The simplest would be to compare timestamps – if equal, consider them the same payment. And this is *not* allowed in this assignment, because while the system you are building is small, there's no reason to lock that in and not make it scaleable.

See the linked document for one way of ensuring payments that occur exactly once. If you feel like coming up with your own strategy, you're free to do so – include a proof of correctness in your Report (remember that network faults are a thing).

To summarize, the **requirements** are:

- Ensure payments are **idempotent**—retrying a transaction should not cause multiple deductions.
- Simple timestamp-based deduplication is **not allowed**.
- Implement a scalable approach (refer to the provided document or propose your own).
- If using a custom approach, provide a **proof of correctness** in the report.

3.4 Offline Payments (10 points)

Say the client initiates a payment, but the payment gateway is down (or you're offline) and you're unable to send the request i.e. it fails.

To handle such situations, the client must maintain a queue of payments when it's offline and (re)send the pending ones when the connection comes back up. (Tip: Does the idempotent assumption simplify anything for you in this feature?)

Requirements

- Clients must queue payments when they are **offline**.
- Pending payments should be automatically resent once connectivity is restored (You may choose to keep a periodic check, but the period in that case must be appropriately chosen).
- Only handle cases where the **client is offline**, not the payment gateway.
- Notify clients about the **success/failure** of processed payments – offline or otherwise.

3.5 2PC with Timeout (10 points)

Fairly straightforward. Payments need to be implemented using 2 phase commit transactions. The voters are going to be the sending and receiving bank servers and the coordinator is the payment gateway. Follow the algorithm discussed in class or the one linked below.

Note that transactions must be aborted after a certain timeout (keep this a configurable parameter) and shouldn't go through on any end if any of the participants ultimately aborts.

3.6 Implementation Details

- Use **gRPC** for communication between components.
- Clearly define **gRPC service definitions** for all interactions.
- Ensure fault tolerance (e.g. servers should be able to recover from crashes, with the previous state) and scalability in the system.

3.7 Report Requirements

- Explain the **design choices** for each requirement.
- Justify the **idempotency approach** with a correctness proof.
- Discuss failure handling mechanisms for **offline payments** and **2PC timeouts**.

3.8 Resources

1. Stripe blog on idempotency
2. Martin Fowler's blog on 2PC

4 (Bonus) Byzantine Failure Resolution (10 points)

Overview: The Byzantine Fault Problem occurs in distributed systems when some nodes provide incorrect or malicious responses, potentially leading to a system failure. To reach consensus despite faulty nodes, a Byzantine Fault Tolerance (BFT) algorithm is required.

Problem Statement

Given n generals (of which one is the designated commander), of which at most t are traitors, implement a simulation where:

- Each general starts with an initial decision (e.g., Attack or Retreat).
- Messages are exchanged synchronously with bounded delay.
- A consensus is reached despite the presence of traitors.

Solution Approach

The system must satisfy:

- **Redundant Communication:** Each general forwards received orders to all others.
- **Majority Voting:** Decisions are finalized based on the majority after $t + 1$ rounds.
- **Consistency:** Honest generals must reach the same decision.

Algorithm

1. The commander sends an order to all $n - 1$ lieutenants.
2. Each lieutenant forwards the received message to others.
3. After $t + 1$ rounds, each general applies majority voting.
4. If the majority decision is "Attack," all agree on "Attack"; otherwise, "Retreat."

Rounds of Simulation

The consensus process occurs over multiple rounds to ensure agreement among honest generals. Each round refines the shared knowledge and eliminates uncertainty caused by traitorous generals.

- **Round 1: Commander Broadcast**
  - The commander sends an order (e.g., "Attack" or "Retreat") to all lieutenants.
- **Round 2: Cross-Verification**
  - Each general asks every other general what order they received from the commander.
  - This helps detect inconsistencies caused by a dishonest commander.
- **Round 3: Secondary Verification**
  - Each general asks all others what responses they received in Round 2.
  - This step increases reliability by filtering out misinformation.
- **Rounds 4 to $t + 1$: Continued Propagation**
  - Messages continue to be exchanged among generals to refine collective agreement.
  - Honest generals converge on the same decision using majority voting.

- **Final Decision: Consensus**
  - After $t + 1$ rounds, each general finalizes a decision.
  - The majority result determines the action (e.g., "Attack" or "Retreat").

Expected Output

The program should take n (number of generals) and t (number of traitors) as input and simulate the consensus process, printing the final agreed decision and the intermediate steps. There is no restriction on the formatting of the output, but it should be readable since this is a simulation.

Note: Ensure correctness by meeting the conditions $n > 3t$ and enforcing synchronized communication.

Task: Implement a simulation of a Byzantine Fault Tolerance algorithm to demonstrate how consensus can be reached.

- Implement a simulation that solves the **Byzantine Fault Problem**.
- The input parameters are:
  - **n**: The number of processes involved.
  - **t**: The number of faulty nodes, where $t < 4$.
- The program should determine whether consensus can be reached despite faulty nodes.
- The output should clearly illustrate the process of reaching consensus or failing to do so.
- Ensure proper logging and debugging information to show steps in reaching agreement.

Report Requirements

- Provide an analysis of the **Byzantine Fault Tolerance** solution with a discussion on its effectiveness.

5 Submission Format

We'll be manually checking the assignment with evals.

Your submission is expected to be a ⟨ `RollNumber` ⟩.zip file containing a directory with the same name as your ⟨ `RollNumber` ⟩ (ex: 2022101001).

A sample submission is shown in Figure 1.

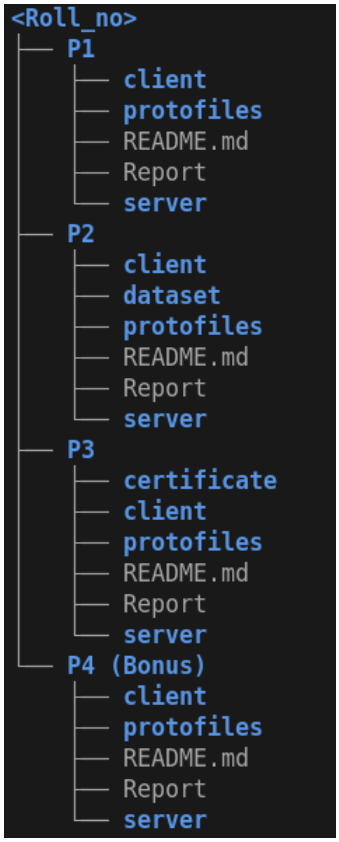

Figure 1: The directory structure for the submission