Deployment of Financial Services With Load Balancer and Auto Scaler

for

Cloud Computing (CS-G527)

by

Kaustab Choudhury, 2020A7PS0013P
Shashank Shreedhar Bhatt, 2020A7PS0078P
Harsh Priyadarshi, 2020A7PS0110P

GitHub Repo: Load Balancer AutoScaler

1. Introduction

In personal finance, having a versatile tool to compute various financial metrics swiftly is invaluable. The Financial Utility Project is designed to cater to the essential needs of individuals seeking quick and accurate calculations for common financial scenarios. This user-friendly programme encompasses a range of functionalities, offering solutions for:

1. Simple Interest Calculation: Swiftly determines the interest accrued on a principal amount over a specified period.

2. Tax Computation: Easily compute taxes based on your income and prevailing tax rates, ensuring accurate financial planning.

3. EMI Calculation: Facilitate hassle-free planning of Equated Monthly Installments (EMIs) for loans, aiding in budget management.

4. FD Returns Calculation: Project future returns on Fixed Deposits by inputting principal, interest rate, and tenure details.

5. Currency Conversion: Effortlessly convert between currencies, providing real-time exchange rates for seamless financial transactions.

This Financial Utility Project aims to empower users with a simplified and efficient tool for managing their financial calculations, offering a comprehensive solution for everyday financial scenarios.

2. System Overview

- System Architecture

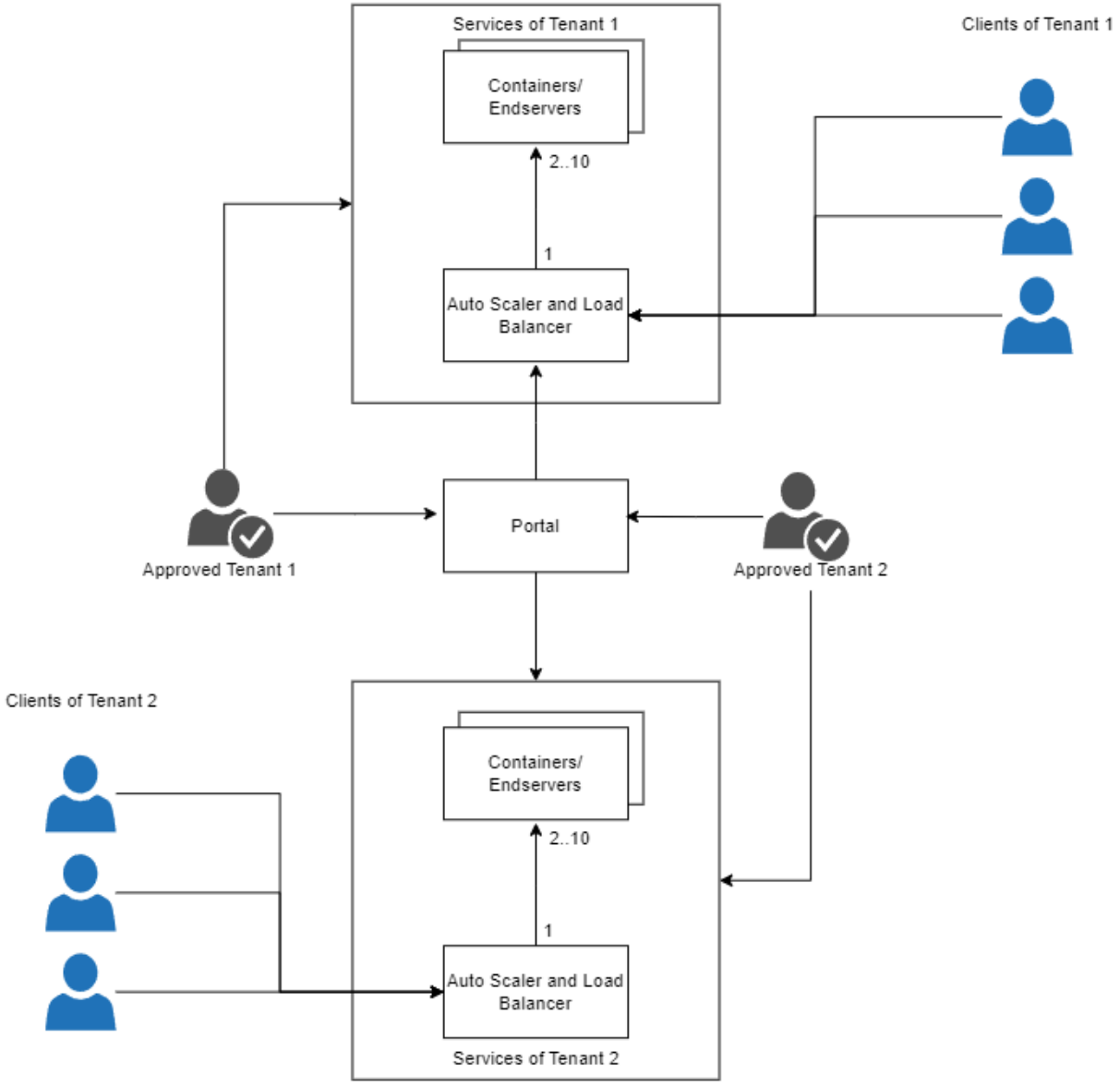

Fig 1: A system design showing two tenants using services with multiple clients

- Key Components
  - Portal
    - The *Initialiser* class acts as the entry point or portal for user interactions. It provides methods for creating user accounts, handling user logins, and initialising server instances.
    - The *CreateAccount* method adds user credentials (email and password) to a CSV file, allowing the creation of user accounts.
    - The *Login* method verifies user credentials by checking against the stored data in the CSV file, providing a simple authentication mechanism.
    - The *CreateInstance* method initialises a new server instance, invoking the *InitialiseServer* method. It supports the creation of up to 10 server instances.
  - Load Balancer
    - The *InvokeMethod* method in the Server class serves as the load balancer, distributing incoming requests among multiple end server containers.
    - It supports various load-balancing strategies, including:
      - **Least Connections:** Select the end server container with the least number of active connections.
      - **Random Choice:** Randomly selects an end server container.
      - **Power of Two Choices:** Randomly select two containers and choose the one with fewer active connections.
      - **Round Robin:** Cycles through end server containers in a round-robin fashion.
      - **IP Hash:** Hashes the client's IP address to determine the end server container.
    - The load balancing logic ensures that requests are distributed evenly among available end server containers.
  - Auto Scaler
    - The *LeAutoScaler* method in the Server class implements auto-scaling logic in two ways:

        - **Threshold-Based Analysis:** Monitors CPU usage, and if it exceeds a threshold (80%), it adds a new end server container.
        - **Queueing Theory:** Monitors the number of active connections. It adds a new end server container if it exceeds a threshold (7 requests in a container).
    - The auto-scaling logic aims to dynamically adjust the number of end server containers based on workload, ensuring efficient resource utilisation.
- Endserver/Container
    - These are the end servers that are running in dockerized containers and can run on the same machine or a different machine.
    - The *Server* class in the '*end.py*' file acts as the end server component that performs specific financial computations based on client requests.
    - It includes the method *RelayClientMessage*, which receives requests from the intermediate node and processes them.
    - The *serve* function initializes a gRPC server, adds the *Server* class as a service, and starts listening on port 50051 for incoming requests.
    - The end server sleeps for 5 seconds after processing each request, simulating some computational load.
- Clients
    - It prompts the user to select a financial service (e.g., simple interest calculation, tax computation) and inputs relevant data for the chosen service.
    - The client then sends a gRPC request to the intermediate node, invoking the selected financial service.
    - The user's input, along with the selected function and the client's IP address, is encapsulated in a gRPC message and sent to the intermediate node.
    - The client logs the request and the server's response, providing a record of the interaction.

3. Design Rationale

- Design Principles
  - *Open for extension and closed for modification*: The project allows for extension without modification. For instance, new end server functions can be added without changing existing code by simply extending the *Server* class and implementing the new functionality.
  - *Single responsibility principle:* Each class has a single responsibility. For example, the *Server* class in 'Scaler.py' is responsible for handling incoming gRPC requests, managing containers, and implementing load balancing and auto-scaling logic.
  - *Separation of Concerns:* The code separates concerns related to different aspects of the system. The *Server* class is responsible for scaling and load balancing, while the *Initialiser* class focuses on handling client requests for creating instances.
  - *Dynamic Scaling:* The system incorporates dynamic scaling logic in the *LeAutoScaler* method, adjusting the number of containers based on the current load and specified conditions
  - *Load Balancing:* The code implements various load balancing strategies, including least connections, random choice, power of two choices, round-robin, and IP hash, giving flexibility in choosing an appropriate strategy based on the requirements.
- Technology stack
  - Docker
  - Python
  - gRPC

4. Security - E-Mail Authentication

There is an authentication mechanism implemented for user registration and login. The portal.py script includes a user registration process that requires users to provide an email address and a password. This password then gets hashed and stored inside a credentials.csv file. Whenever a user decides to authenticate next, their entered password is securely hashed once again and

compared against the stored hash on the intermediate server's end. Based on this match or mismatch, a user is let in or denied access.

5. Execution Instructions

- You must have Docker and Python installed on your machine. We used Docker version 4.24 and Python version 3.10 for testing.
- Make sure you have the following Python libraries installed:
  - gRPCio
  - gRPCio tools
  - docker
- Clone the repository (either from the GitHub repo or by simply downloading the submitted folder).
- Create a docker image named 'endserversleep' via instruction
  `docker build -t endserversleep .`
- Run scaler.py
  `python scaler.py`
- Run portal.py to register a tenant via email and password. Then choose the services, load balancer and auto scaler types.
  `python portal.py`
  Two containers are started as soon as a tenant deploys his services for clients to use. A port number is also issued to the tenant, which is to be passed to the client.
- Run client by passing the port number (as given by tenant) as a command line argument as follows
  `python client.py <PORT_NUMBER>`

6. Conclusion

In summary, the project's design revolves around a distributed system architecture that seamlessly integrates intermediate servers, end servers, and clients, fostering scalability, responsiveness, and security. Leveraging Docker for containerization, our approach allows the dynamic creation and removal of end server instances, adapting to fluctuating workloads. The use of gRPC for

communication encapsulates specific functionalities within services, promoting modularity and ensuring maintainability over time.

The system employs diverse load balancing strategies, such as least connections, random choice, and round-robin, to efficiently distribute incoming requests among end servers. An auto-scaling mechanism further optimizes resource utilization by dynamically adjusting the number of end server instances based on predefined conditions. Security considerations, including secure password hashing, coupled with comprehensive logging throughout the system, enhance robustness and aid in debugging and monitoring.

For future development, the project can benefit from improvements in error handling, dynamic configuration management, and the incorporation of a user interface to enhance user interaction. As we continue refining and testing the system, these considerations will contribute to the ongoing evolution of our project, ensuring its adaptability and sustainability in diverse scenarios.