Mitalee Minde - mminde@umass.edu
Aishwarya Sahoo - absahoo@umass.edu

Introduction

While constructing distributed systems, it's beneficial to maintain a logical separation through multiple-tiers for better scalability and maintainability. We incorporate a two-tier architecture. Frontend has one microservice which is an http_service while the backend has two microservices: one for catalog (catalog_service) and one for order (order_service) implemented using gRPC. We also containerize all the services using Docker, and also build a pipeline to bring-up or tear-down the entire application using Docker-Compose.
We were able to verify the following things:

1. Catalog microservice handling all three cases (invalid, insufficient quantity, valid) by unit tests
2. Order microservice handling three cases (invalid, insufficient, valid) by unit tests
3. Multithreading in catalog and order microservices via ThreadPool Framework
4. Multithreading in HTTP microservice via ThreadingHTTPServer
5. Interprocess communication between client and http service using HTTP requests. Interprocess communication between HTTP service and microservices using remote procedure calls.
6. Docker containerization for all three microservices
7. Orchestration of all three microservices using a single docker-compose file. We were also able to test persistence of all the disk files when a container is built and tear down through mounting of volumes.

Objective

1. Learn how multi-tier architecture is designed and implemented.
2. Performing POC of gRPC/gRPC and gRPC/HTTP based communication.
3. Containerizing an application.
4. Perform analysis of how communication overheads add up in a complex system.
5. Understanding how HTTP Servers work, how they handle multiple requests.

Statement of Scope

1. Part 1
    a. Creating a microservices architecture which supports three different services Frontend service, Catalog Service and Order Service
    b. Frontend Service is implemented using HTTP REST architecture
    c. Catalog and Order Service is implemented using GRPC
    d. Addressing synchronous query issued by clients using read-write locks.
    e. Handling concurrency using thread-per-session model

2. Part 2

    a. Containerizing the application by creating independent dockerfile for each application
    b. Writing a docker-compose file that can bring up (or tear down) all three services using one docker-compose up (or docker-compose down) command.
3. Part 3
    a. Analyzing latency behavior for servers under varying client loads (1-5 clients).
    b. Comparing latency performance between the two requests.
    c. Evaluating the performance difference between Query and Buy operations.

Solutions Overview

1. Frontend Service

Files:
—-http_service.py
—-productBL.py
—-grpc_channel_manager.py

1. http_service.py

This file handles the request handler and controller methods (GET & POST) for GET /products/{product_id} and POST /orders REST apis. The frontend service makes requests to the server – catalog and order microservice via business logic layer calls.

Design Choices

1. **HTTP Server Implementation:** The code uses Python's **ThreadingHTTPServer** to implement a simple HTTP server. This allows handling multiple incoming requests concurrently using threads. This allows for concurrent handling of requests.
2. **Request Handling:** Requests are handled within the HttpReqHandler class, which subclasses BaseHTTPRequestHandler. This class provides methods for handling various HTTP request methods such as GET and POST.
3. **Separation of Business Logic:** The business logic related to product orders and catalog retrieval is encapsulated within the ProductBL class. This separation allows for better organization and maintenance of code, thus making it more modular and testable.
4. **Error Handling:** The code includes basic error handling by returning appropriate HTTP status codes (e.g., 404 for not found, 304 for code not modified) and error messages in JSON format when encountering errors.
5. **Use of Regular Expressions:** Regular expressions are utilized to match and parse URLs, allowing for flexible routing and handling of different types of requests.
6. **Cookie Handling:** The code parses and extracts the client ID from the incoming HTTP request's cookies, allowing for session tracking or other client-specific functionality.

7. **JSON Serialization:** Data is exchanged between the client and server in JSON format, which is a common and lightweight data interchange format suitable for web applications.
8. **Environment Configuration:** The server host and port are configurable through environment variables, providing flexibility in deployment and configuration across different environments.
9. **Graceful Shutdown:** The server is designed to gracefully handle interruptions (e.g., KeyboardInterrupt) and close the server socket when shutting down.

---

2. productBL.py

1. **gRPC Integration:** The code integrates with gRPC-based services using generated stubs (order_pb2_grpc.OrderStub and catalog_pb2_grpc.CatalogStub). This allows for efficient and type-safe communication between services.
2. **Singleton Pattern:** The ProductBL class is implemented as a singleton, ensuring that only one instance of ProductBL is created throughout the application's lifecycle. This pattern is commonly used to manage global state or resources.
3. **Error Handling:** The code includes error handling for gRPC RPC errors (grpc.RpcError). It catches exceptions and prints error details, providing basic error handling and graceful degradation of service.
4. **Separation of Channel Manager:** Logic related to channel management is separated from Business Logic. gRPC channels for order and catalog services are obtained from OrderManager and CatalogManager respectively. This encapsulation helps manage the creation and reuse of gRPC channels, potentially improving performance and resource utilization.
5. **Graceful Error Handling:** In case of gRPC errors, the methods return appropriate default responses (order_pb2.BuyResponse(OrderNumber=-1, Message="Could not place an order") and catalog_pb2.QueryResponse(Name="")). This ensures that the client receives a response even in error scenarios, preventing application crashes or unexpected behavior.
6. **Use of Protocol Buffers:** Protocol Buffers (.proto files) are used to define the service interfaces (order.proto and catalog.proto). This enables language-agnostic communication between services and ensures data consistency and compatibility.

---

3. grpc_channel_manager.py

1. **Singleton Pattern:** Both OrderManager and CatalogManager classes are implemented as singletons. This ensures that only one instance of each manager is created throughout the application's lifecycle, promoting resource efficiency and preventing unnecessary duplication.

2. **Lazy Initialization:** The singletons are lazily initialized using the __new__ method. They are created only when they are first accessed, reducing unnecessary overhead during application startup.
3. **gRPC Channel Creation:** The managers are responsible for creating gRPC channels (grpc.insecure_channel) for communication with the respective services. Insecure channels are used, which do not provide transport security, assuming the communication is within a trusted network.
4. **Separation of Manager Classes for order and catalog:** The managers encapsulate the logic for obtaining channels to interact with the order and catalog services. This separation isolates the networking-related concerns, making the codebase more modular and maintainable.
5. **Class Methods:** The get_channel method is implemented as a class method, allowing access to the channel without needing to instantiate the manager explicitly. This promotes a simpler and more intuitive usage pattern. Security Considerations:

2. Catalog Microservice

catalog
—catalog_service.py

1. **gRPC Server Implementation:** The file implements a gRPC server using Python's grpc library. The server listens for incoming requests from clients and handles them using the defined CatalogServicer.
2. **Threading Model:** The server uses a multi-threaded approach to handle incoming requests concurrently. This is achieved by utilizing futures.ThreadPoolExecutor with a configurable maximum number of worker threads (MAX_WORKER = 4). This design choice allows the server to handle multiple requests simultaneously, improving scalability and responsiveness.
3. **Service Implementation:** The CatalogServicer class implements the gRPC service interface defined in catalog_pb2_grpc.CatalogServicer. It contains methods for querying toy information (Query) and buying items from the toy store (Buy).
4. **Data Storage:** Toy information is stored in a pandas DataFrame (self.db) initially loaded from a CSV file (catalog.csv). This allows for efficient data manipulation and querying.
5. **Concurrency control using Reader-Writer Lock:** The file uses a reader-writer lock implementation from the **readerwriterlock** module (rwlock.RWLockFairD). Reader lock (self.read_lock) and writer lock (self.write_lock) are created to ensure thread-safe access to shared resources (DataFrame) during read and write operations, respectively. This helps prevent data corruption and race conditions in a multi-threaded environment. Fair allocation is used as there are cases where there would be more readers than writers so it prevents starvation as against writers priority or readers priority.
6. **Graceful Shutdown:** The server is designed to gracefully handle interruptions (e.g., KeyboardInterrupt) and shutdown. Upon receiving a termination signal, the server stops accepting new requests and waits for ongoing requests to complete before shutting down.

3. Order Microservice:

order
—order_service.py

1. **gRPC Server Implementation:** The file implements a gRPC server using Python's grpc library. The server listens for incoming requests from clients and handles them using the defined CatalogServicer.
2. **Threading Model:** The server uses a multi-threaded approach to handle incoming requests concurrently. This is achieved by utilizing futures.ThreadPoolExecutor with a configurable maximum number of worker threads (MAX_WORKER = 4). This design choice allows the server to handle multiple requests simultaneously, improving scalability and responsiveness.
3. **Service Implementation:** The OrderServicer class implements the gRPC service interface defined in order_pb2_grpc.OrderServicer. It contains methods for buying items from the toy store (BuyRequest).
4. **Data Storage:** Toy information is initially loaded from a CSV file (order.csv). This allows for efficient data manipulation and querying.
5. **Concurrency control using Lock:** The file uses a lock implementation from the **threading** module. Locks are created to ensure thread-safe access to shared resources (file) during write operations, respectively. This helps prevent data corruption and race conditions in a multi-threaded environment.
6. **Graceful Shutdown:** The server is designed to gracefully handle interruptions (e.g., KeyboardInterrupt) and shutdown. Upon receiving a termination signal, the server stops accepting new requests and waits for ongoing requests to complete before shutting down.

API Architecture

Frontend Service

1. **Query API**
    1. Endpoint : /products/<toyName>
    2. Returns:
        a. Case 1 toy exits

```
{
    "data": {
        "name": "Tux",
        "price": 15.99,
        "quantity": 100
    }
}
```

        b. Case 2: toy does not exist

```
{
```

```
            "error": {
                "code": 404,
                "message": "product not found"
            }
        }
```

2. **Buy API**
    1. Endpoint: /orders
    2. Request Body:

```
{
  "name": "Tux",
  "quantity": 1
}
```

    3. Response:
        a. Case 1: Order is successfully placed

```
{
   "data": {
       "order_number": 4
   }
}
```

        b. Case 2: Invalid Toyname

```
{
   "error": {
       "code": 304,
       "message": "could not post order"
   }
}
```

        c. Case 3 Invalid Quantity

```
{
   "error": {
       "code": 304,
       "message": "could not post order"
   }
}
```

List of known issues

[1] **Client ID - Thread ID Mappings:** We were able to check if thread ID and client ID mappings are preserved across sessions, and realized they were not preserved in some cases. This could be due to the original thread already being allocated to some other client ID by the time the original client ID made a request.
Solution: To solve the above problem we could store the mappings between thread IDs and client IDs persistently, such as in a database or a file system or in memory. If a client associated thread is already allocated, we could requeue the thread in the waiting queue to let it wait for some time.

Testing

1. **Catalog and Order microservice unit testing:** We test the various test cases in catalog like invalid toy name, insufficient stocks and valid test case using unittest library of python. The files are in /src/server/catalog/unit_test.py and /src/server/order/unit_test.py. We also test these cases manually using Postman - gRPC interface.
2. **HTTP Service testing:** We test the APIs of our HTTP service (GET and POST) using Postman interface. We test invalid query, invalid product name, insufficient stocks and valid test cases.We have also written unit test cases for HTTP service which tests call cases mentioned previously, this file /src/server/catalog/unit_test.py and /src/frontend/unit_test.py
3. **Containerization persistence testing:** The main use-case in our containerization was making sure the disk files remained persisted even after deleting the containers using volume mounting. We ensure this by checking if order ID and quantity of toy remains persisted even after mounting, unmounting and remounting all the containers.
4. **Load Testing for various clients:** We also perform load testing of our clients by randomizing products in GET /products/<product_id> endpoint, and calling POST /orders query if a random probability is less than or equal to probability = 0.5. This ensures that 50% of the POST requests reach the server which has sufficient quantity. We run this load testing on Edlab machines to ensure that there is a remote server and client using Edlab and our remote PC, and use containerization and non-containerization approaches to validate our hypotheses/results. This is run for clients ranging from 1 to 5.

References:

1. For catalog and order microservice (official documentation of gRPC) https://github.com/grpc/grpc/tree/master/examples/python/route_guide
2. For singleton pattern https://python-patterns.guide/gang-of-four/singleton/
3. Reader Writer Lock https://pypi.org/project/readerwriterlock/