Two options
1. Write a report and document
2. Record a video walking through all functions

**COMP 41720 Distributed Systems Lab 1: Synchronous Communication Patterns**

**Lab Overview and Learning Objectives**

This comprehensive lab guide will help you implement a distributed system with synchronous communication using both REST and gRPC approaches. You'll work with both Java and Python to understand different implementation patterns.

**Learning Objectives:**

- Understand synchronous communication in distributed systems
- Implement both RESTful API and gRPC services
- Work with build tools (Maven) and containerization (Docker)
- Compare different communication patterns and their trade-offs
- Gain practical experience with Java and Python in distributed contexts

**Phase 1: Socket-Based Client-Server Application**

Java Implementation with Maven

Project Structure:

```text
socket-lab/
├── src/
│   ├── main/java/
│   │   ├── Server.java
│   │   └── Client.java
│   └── test/java/
├── pom.xml
└── Dockerfile
```

pom.xml (Maven Configuration):

```xml
<project>
    <modelVersion>4.0</modelVersion>
    <groupId>edu.distributed-systems</groupId>
    <artifactId>socket-lab</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Server.java Pseudo Code:

```java
// Create ServerSocket on port 8080
// While server is running:
//   - Accept client connection
//   - Create input/output streams
//   - Read client message
//   - Process request (e.g., convert to uppercase)
//   - Send response back to client
//   - Close connection
```

Client.java Pseudo Code:

```java
// Create Socket connection to server
// Create input/output streams
// Send message to server
// Wait for response
// Print server response
// Close connection
```

Python Implementation

Project Structure:

```text
python-socket-lab/
├── server.py
├── client.py
├── requirements.txt
└── Dockerfile
```

server.py Pseudo Code:

```python
# Import socket library
# Create TCP/IP socket
# Bind to localhost:8080
# Listen for connections
# While True:
#   - Accept client connection
#   - Receive data from client
#   - Process data
#   - Send response back
#   - Close connection
```

**Phase 2: RESTful API Implementation**

Java Spring Boot Implementation

Project Structure:

```text
rest-api-lab/
├── src/
│   ├── main/java/
│   │   └── com/
│   │       └── example/
│   │           ├── Application.java
│   │           ├── controller/
│   │           │   └── ApiController.java
│   │           ├── model/
│   │           │   └── User.java
│   │           └── service/
│   │               └── UserService.java
│   └── test/java/
├── pom.xml
└── Dockerfile
```

pom.xml Dependencies:

```xml
```

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.7.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <version>2.7.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

ApiController.java Pseudo Code:

```java
@RestController
@RequestMapping("/api/users")
public class ApiController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getUsers() {
        return userService.getAllUsers();
    }

    @GetMapping("/{id}")
    public User getUser(@PathVariable String id) {
        return userService.getUserById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable String id, @RequestBody User user) {
        return userService.updateUser(id, user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable String id) {
```

```
        userService.deleteUser(id);
    }
}
```

Python Flask Implementation

Project Structure:

```text
python-rest-lab/
├── app.py
├── models.py
├── requirements.txt
└── Dockerfile
```

app.py Pseudo Code:

```python
from flask import Flask, jsonify, request
from models import User

app = Flask(__name__)

@app.route('/api/users', methods=['GET'])
def get_users():
    # Return list of all users
    pass

@app.route('/api/users/<id>', methods=['GET'])
def get_user(id):
    # Return specific user
    pass

@app.route('/api/users', methods=['POST'])
def create_user():
    # Create new user from request data
    pass

@app.route('/api/users/<id>', methods=['PUT'])
def update_user(id):
    # Update existing user
    pass

@app.route('/api/users/<id>', methods=['DELETE'])
def delete_user(id):
    # Delete user
```

```
    pass
```

**Phase 3: gRPC Implementation**

Java gRPC Implementation

Project Structure:

```text
grpc-lab/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── example/
│   │   │           ├── UserServiceGrpc.java
│   │   │           ├── UserServer.java
│   │   │           └── UserClient.java
│   │   └── proto/
│   │       └── user_service.proto
│   └── test/java/
├── pom.xml
└── Dockerfile
```

user_service.proto:

```proto
syntax = "proto3";

option java_package = "com.example";
option java_outer_classname = "UserServiceProto";

service UserService {
    rpc GetUser (UserRequest) returns (User);
    rpc CreateUser (CreateUserRequest) returns (User);
    rpc UpdateUser (UpdateUserRequest) returns (User);
    rpc DeleteUser (UserRequest) returns (Empty);
}

message UserRequest {
    string id = 1;
}

message CreateUserRequest {
    string name = 1;
    string email = 2;
}
```

```
message UpdateUserRequest {
  string id = 1;
  string name = 2;
  string email = 3;
}

message User {
  string id = 1;
  string name = 2;
  string email = 3;
}

message Empty {}
```

UserServer.java Pseudo Code:

```java
public class UserServer {
  private Server server;

  public void start() throws IOException {
    server = ServerBuilder.forPort(8080)
      .addService(new UserServiceImpl())
      .build()
      .start();
  }

  static class UserServiceImpl extends UserServiceGrpc.UserServiceImplBase {
    @Override
    public void getUser(UserRequest request, StreamObserver<User> responseObserver) {
      // Implement getUser logic
    }

    // Implement other methods
  }
}
```

Python gRPC Implementation

Project Structure:

```text
python-grpc-lab/
├── proto/
│   └── user_service.proto
```

```
├── generated/
├── server.py
├── client.py
├── requirements.txt
└── Dockerfile
```

requirements.txt:

```text
grpcio==1.47.0
grpcio-tools==1.47.0
protobuf==3.20.1
```

server.py Pseudo Code:

```python
import grpc
from concurrent import futures
import user_service_pb2
import user_service_pb2_grpc

class UserService(user_service_pb2_grpc.UserServiceServicer):
    def GetUser(self, request, context):
        # Implement GetUser
        pass

    # Implement other methods

def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    user_service_pb2_grpc.add_UserServiceServicer_to_server(UserService(), server)
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()
```

Docker Setup for All Components

Java Dockerfile:

```dockerfile
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/*.jar app.jar
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]
```

Python Dockerfile:

```dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python", "app.py"]
```

docker-compose.yml:

```yaml
version: '3.8'
services:
  java-rest-service:
    build: ./rest-api-lab
    ports:
      - "8080:8080"

  python-rest-service:
    build: ./python-rest-lab
    ports:
      - "5000:5000"

  java-grpc-service:
    build: ./grpc-lab
    ports:
      - "50051:50051"

  python-grpc-service:
    build: ./python-grpc-lab
    ports:
      - "50052:50052"
```

**Testing and Validation**

Test Cases for Each Implementation

1. Socket Implementation Tests:
    - Test connection establishment
    - Test message sending/receiving
    - Test error handling for invalid connections
2. REST API Tests:
    - CRUD operations test cases

    - HTTP status code validation
    - Request/response format validation
3. gRPC Tests:
    - Service method invocation tests
    - Data serialization/deserialization tests
    - Error handling tests

Performance Comparison

Create a simple benchmarking script to compare:

```
python
# benchmark.py pseudo code
import time
import requests
import grpc

def benchmark_rest():
    start = time.time()
    # Make multiple REST API calls
    end = time.time()
    return end - start

def benchmark_grpc():
    start = time.time()
    # Make multiple gRPC calls
    end = time.time()
    return end - start

# Compare results
```

Additional Resources

1. Maven Documentation: https://maven.apache.org/guides/
2. Docker Documentation: https://docs.docker.com/
3. gRPC Official Documentation: https://grpc.io/docs/
4. Spring Boot Guide: https://spring.io/guides/gs/rest-service/
5. Flask Documentation: https://flask.palletsprojects.com/

## Deliverables and Rubric

### Deliverables:

- Source Code: Submit all source code for your application.
- Design & Explanation: A brief document OR a video demonstration (not to exceed 10 minutes) is expected.
  - Document: This should be a brief document or a dedicated section in your README file detailing your system's design, how it demonstrates synchronous communication, and your choice of technology.
  - Video: The video should show your application/services running and clearly explain the design choices and how they fulfil the requirements of the lab.

### Assessment Rubric

Your submission will be assessed based on the following criteria:

- Correctness and Functionality (50%): Does the application run successfully and correctly demonstrate synchronous request-response communication?
- Design and Implementation Quality (30%): Is the system design clear, and is the code well-structured and readable?
- Documentation and Explanation (20%): Are the instructions clear, and does the documentation effectively explain your design and how it fulfils the lab's requirements?

### Conclusion

Upon completing this lab, you will have a solid foundation in using Maven for building Java applications, Docker for containerization, and a practical understanding of synchronous communication patterns.

Good luck with the lab! If you run into any issues, don't hesitate to reach out for assistance on the module's communication channels.