Project Assignment 1: Implementing the SWIM Protocol Using Docker and gRPC

Members:

Fahim Shahriar Khan (1002242739)

Nowshin Tabassum (1002226577)

Course: CSE 5306

Group: 6

Q1: Intro to Docker:

1. Include the link to your shared image on Docker Hub, as stated in Part 3.
    a. Nowshin Tabassum
    nowshiin/getting-started-todo-app:latest

    b. Fahim Shahriar Khan
    fahimshahriar52/getting-started-todo-app:latest

2. Include the link to the other shared image on Docker Hub related to Part 7.2
    a. Nowshin Tabassum
    nowshiin/workshop-getting-started:latest

    b. Fahim Shahriar Khan
    fahimshahriar52/getting-started:latest

Docker Commands to explain:

- docker compose watch: automatically updates the changes made to source code to the running services.
- docker compose up: start and run an entire application defined in a Docker Compose file (docker-compose.yml)
- docker build: creates a Docker image from a Dockerfile
- docker image ls: lists all images that exists locally
- docker image history: view the logs/history of the image or how the image was created
- docker run: used to build a container from image that was build using docker build command
    - -v/--mount: used to attach a volume to a container when running it
- docker push: Pushes a local Docker image to DockerHub
- docker ps: used to view running containers
- docker rm/stop: used to remove or stop a running container
- docker network:
- docker volume: persist data created and used by a docker container. The data can be accessed by other containers and it persists even if a container is deleted.
    - create: Create a new volume.
    - ls: List all volumes.
    - rm: Remove a volume.
    - prune: Remove all unused volumes.

Q2: Intro To gRPC:

Java: Quick Start

First, we must "g*it clone -b v1.70.0 --depth 1 https://github.com/grpc/grpc-java*" and move to the /example ("*cd grpc-java/examples*") directory. To run the example, we must.

a. Compile the client and server

```
(ds_assignment) fahimkhan@FWJ70PVW91 examples % date; whoami; ./gradlew installDist
Sun Feb 23 19:48:53 CST 2025
fahimkhan
Starting a Gradle Daemon (subsequent builds will be faster)

> Task :compileJava
warning: [options] source value 8 is obsolete and will be removed in a future release
warning: [options] target value 8 is obsolete and will be removed in a future release
warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
Note: Some input files use or override a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
Note: /Users/fahimkhan/Documents/Coursework/2nd semester/CSE 5306/assignment_1/grpc-java
/examples/src/main/java/io/grpc/examples/customloadbalance/CustomLoadBalanceClient.java
uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
3 warnings

BUILD SUCCESSFUL in 9s
41 actionable tasks: 41 executed
```

b. Run the server

```
(ds_assignment) fahimkhan@FWJ70PVW91 examples % date; whoami; ./build/install/examples/b
in/hello-world-server
Sun Feb 23 19:55:11 CST 2025
fahimkhan
Feb 23, 2025 7:55:12 PM io.grpc.examples.helloworld.HelloWorldServer start
INFO: Server started, listening on 50051
```

c. From another terminal, run the client. We can see that we get a response from the server.

```
(base) fahimkhan@FWJ70PVW91 examples % date; whoami; ./build/install/example
s/bin/hello-world-client
Sun Feb 23 19:57:55 CST 2025
fahimkhan
Feb 23, 2025 7:57:55 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Will try to greet world ...
Feb 23, 2025 7:57:55 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Greeting: Hello world
```

Now we want to update the gRPC service and to do that we must.

a) Add an extra server method to the *"helloworld.proto"* file. We add the **SayHelloAgain** rpc to the **Greeter** service.

```
14  syntax = "proto3";
15
16  option java_multiple_files = true;
17  option java_package = "io.grpc.examples.helloworld";
18  option java_outer_classname = "HelloWorldProto";
19  option objc_class_prefix = "HLW";
20
21  package helloworld;
22
23  // The greeting service definition.
24  service Greeter {
25    // Sends a greeting
26    rpc SayHello (HelloRequest) returns (HelloReply) {}
27    // We added this New method.
28    rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
29  }
30
31  // The request message containing the user's name.
32  message HelloRequest {
33    string name = 1;
34  }
35
36  // The response message containing the greetings
37  message HelloReply {
38    string message = 1;
39  }
```

b) Now we must update "*HelloWorldServer.java"* to implement the newly added rpc, as shown below.

```
82    static class GreeterImpl extends GreeterGrpc.GreeterImplBase {
83
84      @Override
85      public void sayHello(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
86        HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.getName()).build();
87        responseObserver.onNext(reply);
88        responseObserver.onCompleted();
89      }
90
91      @Override
92      public void sayHelloAgain(HelloRequest req, StreamObserver<HelloReply> responseObserver) {
93        // Generate another greeting message for the new method.
94        HelloReply reply = HelloReply.newBuilder().setMessage("Hello again " + req.getName()).build();
95
96        // Send the reply back to the client.
97        responseObserver.onNext(reply);
98
99        // Indicate that no further messages will be sent to the client.
100       responseObserver.onCompleted();
101     }
102   }
103 }
```

c) Now we must update the "*HelloWorldClient.java*" to call the newly implemented **SayHelloAgain** rpc, as shown below.

```java
public void greet(String name) {
  // Log a message indicating the intention to greet a user.
  logger.info("Will try to greet " + name + " ...");

  // Creating a request with the user's name.
  HelloRequest request = HelloRequest.newBuilder().setName(name).build();
  HelloReply response;
  try {
    // Call the original method on the server.
    response = blockingStub.sayHello(request);
  } catch (StatusRuntimeException e) {
    // Log a warning if the RPC fails.
    logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    return;
  }

  // Log the response from the original method.
  logger.info("Greeting: " + response.getMessage());

  try {
    // Call the new method on the server.
    response = blockingStub.sayHelloAgain(request);
  } catch (StatusRuntimeException e) {
    // Log a warning if the RPC fails.
    logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    return;
  }

  // Log the response from the new method.
  logger.info("Greeting: " + response.getMessage());
}
```

Now we want to run the updated app. To do this, we must.

a) Compile the updated client and server

```
(ds_assignment) fahimkhan@FWJ70PVW91 examples % date; whoami; ./gradlew installDist
Sun Feb 23 20:15:03 CST 2025
fahimkhan

> Task :compileJava
warning: [options] source value 8 is obsolete and will be removed in a future release
warning: [options] target value 8 is obsolete and will be removed in a future release
warning: [options] To suppress warnings about obsolete options, use -Xlint:-options.
Note: /Users/fahimkhan/Documents/Coursework/2nd semester/CSE 5306/assignment_1/grpc-java/examples/
src/main/java/io/grpc/examples/customloadbalance/CustomLoadBalanceClient.java uses unchecked or un
safe operations.
Note: Recompile with -Xlint:unchecked for details.
3 warnings

BUILD SUCCESSFUL in 3s
41 actionable tasks: 39 executed, 2 up-to-date
```

b) Run the updated server

```
(ds_assignment) fahimkhan@FWJ70PVW91 examples % date; whoami; ./build/install/examples/bin/hello-world-server
Sun Feb 23 20:16:45 CST 2025
fahimkhan
Feb 23, 2025 8:16:45 PM io.grpc.examples.helloworld.HelloWorldServer start
INFO: Server started, listening on 50051
```

c) From another terminal, run the updated client. We can see that we do get a response from the newly implemented **SayHelloAgain** rpc.

```
(base) fahimkhan@FWJ70PVW91 examples % date; whoami; ./build/install/examples/bin/hello-world-client
Sun Feb 23 20:18:25 CST 2025
fahimkhan
Feb 23, 2025 8:18:26 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Will try to greet world ...
Feb 23, 2025 8:18:26 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Greeting: Hello world
Feb 23, 2025 8:18:26 PM io.grpc.examples.helloworld.HelloWorldClient greet
INFO: Greeting: Hello again world
```

Java: Basics tutorial

First, we must "g*it clone -b v1.70.0 --depth 1 https://github.com/grpc/grpc-java*" and move to the /example ("*cd grpc-java/examples*") directory.

First, we must define our services in the "*route_guide.proto"* file as shown below.

```
// Interface exported by the server.
service RouteGuide {
  // A simple RPC.
  //
  // Obtains the feature at a given position.
  //
  // A feature with an empty name is returned if there's no feature at the given
  // position.
  rpc GetFeature(Point) returns (Feature) {}

  // A server-to-client streaming RPC.
  //
  // Obtains the Features available within the given Rectangle.  Results are
  // streamed rather than returned at once (e.g. in a response message with a
  // repeated field), as the rectangle may cover a large area and contain a
  // huge number of features.
  rpc ListFeatures(Rectangle) returns (stream Feature) {}

  // A client-to-server streaming RPC.
  //
  // Accepts a stream of Points on a route being traversed, returning a
  // RouteSummary when traversal is completed.
  rpc RecordRoute(stream Point) returns (RouteSummary) {}

  // A Bidirectional streaming RPC.
  //
  // Accepts a stream of RouteNotes sent while a route is being traversed,
  // while receiving other RouteNotes (e.g. from other users).
  rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
}
```

a) We define "*rpc GetFeature(Point) returns (Feature) {}*" which is a simple RPC where the client sends a request to the server using the stub and waits for a response to come back, just like a normal function call.
b) We define "*rpc ListFeatures(Rectangle) returns (stream Feature) {}"* which is a server-side streaming RPC where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages.
c) We define *"rpc RecordRoute(stream Point) returns (RouteSummary) {}"* which is a client-side streaming RPC where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them all and return its response.
d) We define *"rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}"* which is a bidirectional streaming RPC where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes.

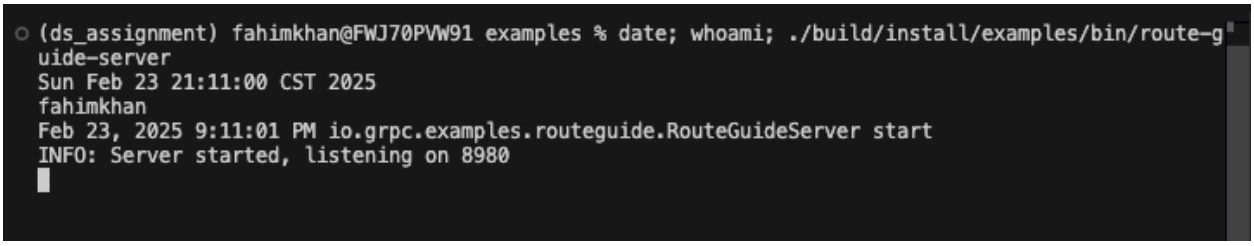
And then we have our message definitions in the "route_guide.proto" file shown below.

```
// If a feature could not be named, the name is empty.
message Feature {
  // The name of the feature.
  string name = 1;

  // The point where the feature is detected.
  Point location = 2;
}

// Not used in the RPC.  Instead, this is here for the form serialized to disk.
message FeatureDatabase {
  repeated Feature feature = 1;
}

// A RouteNote is a message sent while at a given point.
message RouteNote {
  // The location from which the message is sent.
  Point location = 1;

  // The message to be sent.
  string message = 2;
}

message RouteSummary {
  // The number of points received.
  int32 point_count = 1;

  // The number of known features passed while traversing the route.
  int32 feature_count = 2;

  // The distance covered in metres.
  int32 distance = 3;

  // The duration of the traversal in seconds.
  int32 elapsed_time = 4;
}
```

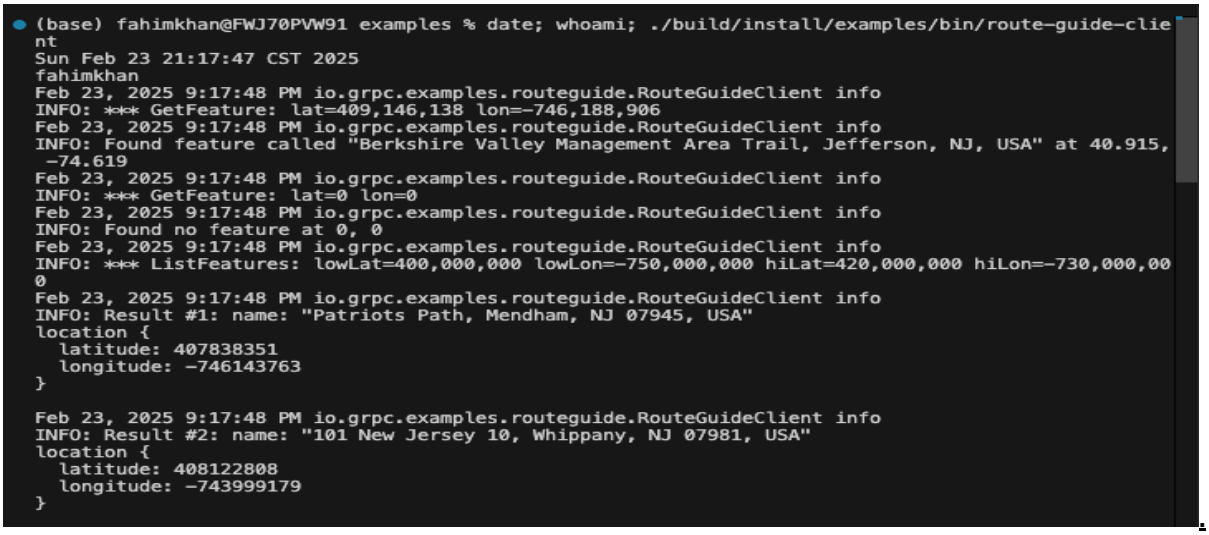
```
message Point {
  int32 latitude = 1;
  int32 longitude = 2;
}

// A latitude-longitude rectangle, represented as two diagonally opposite
// points "lo" and "hi".
message Rectangle {
  // One corner of the rectangle.
  Point lo = 1;

  // The other corner of the rectangle.
  Point hi = 2;
}

// A feature names something at a given point.
//
// If a feature could not be named, the name is empty.
message Feature {
  // The name of the feature.
  string name = 1;

  // The point where the feature is detected.
  Point location = 2;
}
```

Next, we must generate the gRPC server and client interfaces from our .proto service definition. We do this using **Gradle**.

Now we must implement the rpc methods defined in our .proto file inside "*./src/main/java/io/grpc/examples/routeguide/RouteGuideServer.java".* After we have defined the 1. Simple RPC, 2. Server-side streaming RPC, 3. Client-side streaming RPC, and 4. Bidirectional streaming RPC, we must start the server.



Now we must define our clients to Instantiate stubs to call the services define inside "*./src/main/java/io/grpc/examples/routeguide/RouteGuideServer.java".* After, we implemented the 1. Simple RPC (GetFeature), 2. Server-side streaming RPC (ListFeatures), 3. Client-side streaming RPC (RecordRoute), and 4. Bidirectional streaming RPC (RouteChat) in the "./src/main/java/io/grpc/examples/routeguide/RouteGuideClient.java" file we run the client on another terminal.

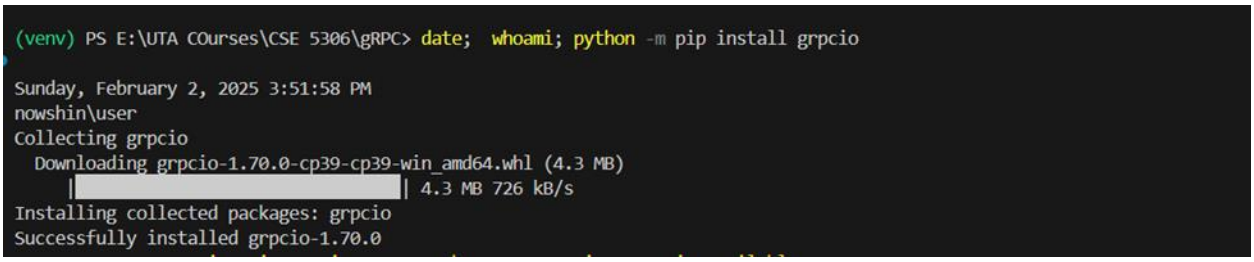


(output truncated in the screenshot)

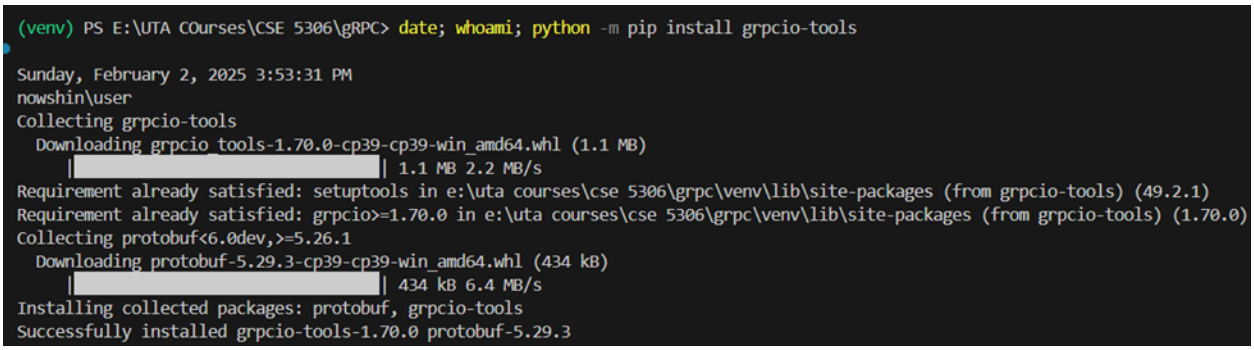
Python: Quick Start:

a) Installing necessary modules:

Installing gRPCio

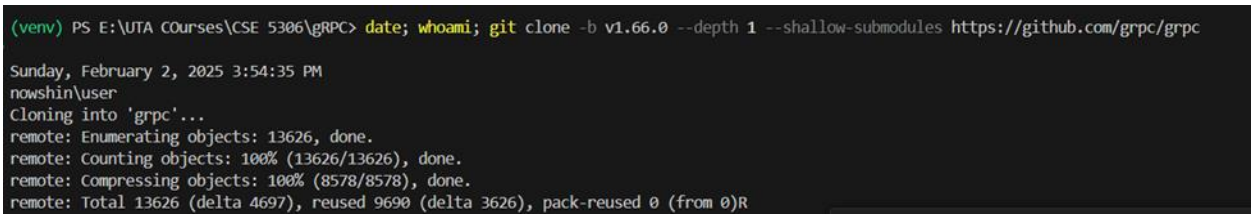


Installing gRPC-tools:



b) Running a Python Client-Server Application

A GitHub repository containing gRPC example code is cloned.



Now, To run a python client-server application with gRPC:

c) Starting the Server

The greeter_server.py script is ran in one terminal, listening on port 50051.

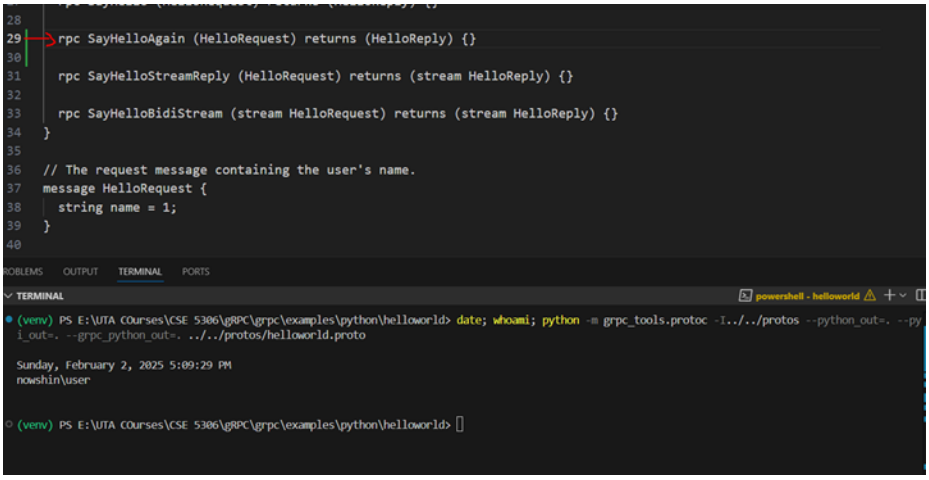
d) **Running the Client:**

The greeter_client.py script is ran in another terminal, making a HelloRequest to the server.

Client made Hello Request to server on port 50051, server received the request, formulated the message and sent to client. Client printed the final message: ‘ Hello Nowshin’. This is shown is the screenshot below.

e) **Updated the helloworld.proto file:**

The helloworld.proto file is updated to add new method at **line 29**.



Then the following command is ran to get the python code for updated proto file

python -m grpc_tools.protoc -I../../protos --python_out=. --pyi_out=. --grpc_python_out=. ../../protos/helloworld.proto

The new ‘HelloAgain’ method is added to both server and client file.

Server File:

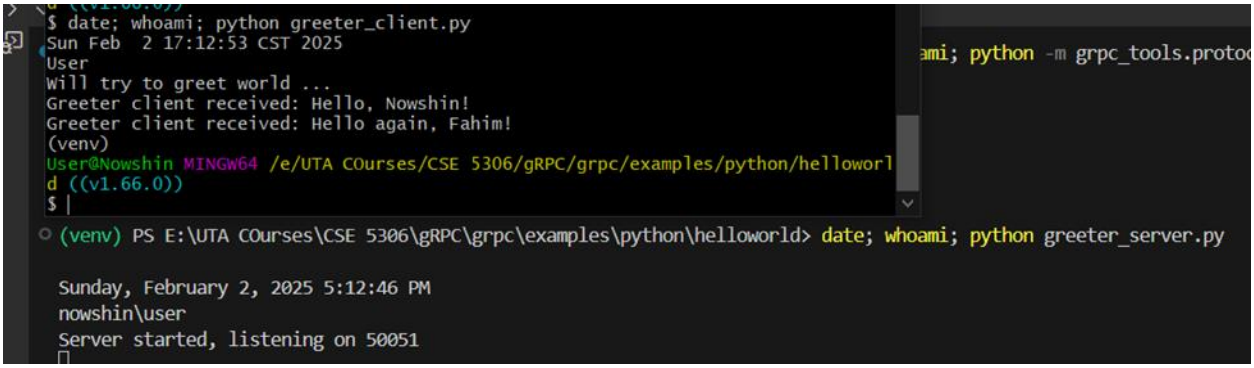
```python
class Greeter(helloworld_pb2_grpc.GreeterServicer):
    def SayHello(self, request, context):
        return helloworld_pb2.HelloReply(message="Hello, %s!" % request.name)
    def SayHelloAgain(self, request, context):
        return helloworld_pb2.HelloReply(message=f"Hello again, {request.name}!")
```

Client file:

```python
def run():
    # NOTE(gRPC Python Team): .close() is possible on a channel and should be
    # used in circumstances in which the with statement does not fit the needs
    # of the code.
    print("Will try to greet world ...")
    with grpc.insecure_channel("localhost:50051") as channel:
        stub = helloworld_pb2_grpc.GreeterStub(channel)
        response1 = stub.SayHello(helloworld_pb2.HelloRequest(name="Nowshin"))
        response2 = stub.SayHelloAgain(helloworld_pb2.HelloRequest(name='Fahim'))
    print("Greeter client received: " + response1.message)
    print("Greeter client received: " + response2.message)
```

f) Running the updates client-server application

Then ran the greeter_server in one terminal and greeter_client in another terminal with the modifications and the hello messages were printed



Python: Basics Tutorial:

A) Defining the service

From basics Tutorial, after clonic the github repository of gRPC, we redirect to the \grpc\examples\python directory. The gRPC services with four rpc methods is already defined in the route_guide.proto file under proto folder.

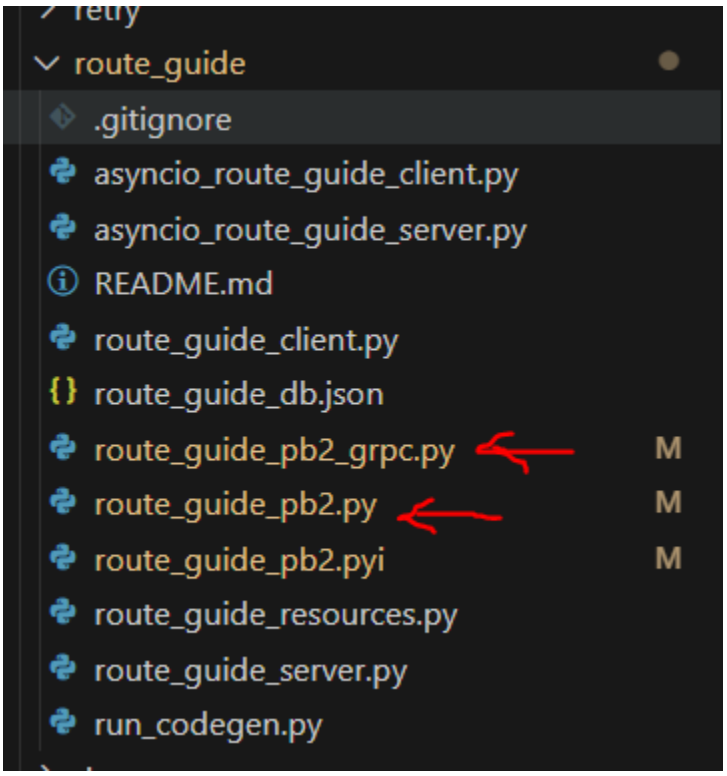
B) Generating client and server code

Running the following command generated the Python code from route_guide.proto file

python -m grpc_tools.protoc -I../../protos --python_out=. --pyi_out=. --grpc_python_out=. ../../protos/route_guide.proto

```
PS E:\UTA COurses\CSE 5306\gRPC\grpc\examples\python\route_guide> date; whoami; python -m grpc_tools.protoc -I../../protos --python_out=. --pyi_out=. --gr
pc_python_out=. ../../protos/route_guide.proto

Sunday, February 23, 2025 7:50:02 PM
nowshin\user
```

These are the python codes that were generated from the proto file



C) Creating the server

The *route_guide_server.py* file under route_guide folder implemented the RouteGuide service methods that was defined in the .proto file.

There were 4 service methods:

1. Simple RPC
2. Response Streaming RPC
3. Request Streaming RPC
4. Bidirectional RPC

To start the server, a serve function is written in the *route_guide_server.py,* that starts the server in 50051 port.

```
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    route_guide_pb2_grpc.add_RouteGuideServicer_to_server(
        RouteGuideServicer(), server
    )
    server.add_insecure_port("[::]:50051")
    server.start()
    server.wait_for_termination()
```

D) Creating the client

The *route_guide_client.py* file under route_guide folder is the client that will call the servoce methods defined in the server file.

To call the service methods defined in server, at first a stub needs to be created using the RouteGuideStub class of the route_guide_pb2_grpc module, generated from the.proto.

```python
def run():
    # NOTE(gRPC Python Team): .close() is possible on a channel and should be
    # used in circumstances in which the with statement does not fit the needs
    # of the code.
    with grpc.insecure_channel("localhost:50051") as channel:
        stub = route_guide_pb2_grpc.RouteGuideStub(channel)
        print("-------------- GetFeature --------------")
        guide_get_feature(stub)
        print("-------------- ListFeatures --------------")
        guide_list_features(stub)
        print("-------------- RecordRoute --------------")
        guide_record_route(stub)
        print("-------------- RouteChat --------------")
        guide_route_chat(stub)
```

E) Running the server:

Running the python route_guide_server.py, starts the server and keeps waiting for a client to call its service methods.

```
PS E:\UTA COurses\CSE 5306\gRPC\grpc\examples\python\route_guide> date; whoami; python route_guide_server.py

Sunday, February 23, 2025 8:05:56 PM
nowshin\user
Server start at port 50051
```

F) Running the client:

Running the python route_guide_client.py, the client connects with server through the same port and calls necessary service methods defined in the server and gets output returns from the server.

```
(venv)
User@Nowshin MINGW64 /e/UTA COurses/CSE 5306/gRPC/grpc/examples/python/route_gui
de ((v1.66.0))
$ date; whoami; python route_guide_client.py
Sun Feb 23 20:12:46 CST 2025
User
-------------- GetFeature --------------
Feature called 'Berkshire Valley Management Area Trail, Jefferson, NJ, USA' at latitude: 409146138, longitude: -746188906
Found no feature at latitude: 0, longitude: 0
-------------- ListFeatures --------------
Looking for features between 40, -75 and 42, -73
Feature called 'Patriots Path, Mendham, NJ 07945, USA' at latitude: 407838351, longitude: -746143763
Feature called '101 New Jersey 10, Whippany, NJ 07981, USA' at latitude: 408122808, longitude: -743999179
Feature called 'U.S. 6, Shohola, PA 18458, USA' at latitude: 413628156, longitude: -749015468
Feature called '5 Conners Road, Kingston, NY 12401, USA' at latitude: 419999544, longitude: -740371136
Feature called 'Mid Hudson Psychiatric Center, New Hampton, NY 10958, USA' at latitude: 414008389, longitude: -743951297
Feature called '287 Flugertown Road, Livingston Manor, NY 12758, USA' at latitude: 419611318, longitude: -746524769
Feature called '4001 Tremley Point Road, Linden, NJ 07036, USA' at latitude: 406109563, longitude: -742186778
Feature called '352 South Mountain Road, Wallkill, NY 12589, USA' at latitude: 416802456, longitude: -742370183
Feature called 'Bailey Turn Road, Harriman, NY 10926, USA' at latitude: 412950425, longitude: -741077389
Feature called '193-199 Wawayanda Road, Hewitt, NJ 07421, USA' at latitude: 412144655, longitude: -743949739
Feature called '406-496 Ward Avenue, Pine Bush, NY 12566, USA' at latitude: 415736605, longitude: -742847522
Feature called '162 Merrill Road, Highland Mills, NY 10930, USA' at latitude: 413843930, longitude: -740501726
Feature called 'Clinton Road, West Milford, NJ 07480, USA' at latitude: 410873075, longitude: -744459023
Feature called '16 Old Brook Lane, Warwick, NY 10990, USA' at latitude: 412346009, longitude: -744026814
Feature called '3 Drake Lane, Pennington, NJ 08534, USA' at latitude: 402948455, longitude: -747903913
Feature called '6324 8th Avenue, Brooklyn, NY 11220, USA' at latitude: 406337092, longitude: -740122226
Feature called '1 Merck Access Road, Whitehouse Station, NJ 08889, USA' at latitude: 406421967, longitude: -747727624
Feature called '78-98 Schalck Road, Narrowsburg, NY 12764, USA' at latitude: 416318082, longitude: -749677716
Feature called '282 Lakeview Drive Road, Highland Lake, NY 12743, USA' at latitude: 415301720, longitude: -748416257
Feature called '330 Evelyn Avenue, Hamilton Township, NJ 08619, USA' at latitude: 402647019, longitude: -747071791
Feature called 'New York State Reference Route 987E, Southfields, NY 10975, USA' at latitude: 412567807, longitude: -741058078
Feature called '103-271 Tempaloni Road, Ellenville, NY 12428, USA' at latitude: 416855156, longitude: -744420597
Feature called '1300 Airport Road, North Brunswick Township, NJ 08902, USA' at latitude: 404663628, longitude: -744820157
Feature called '' at latitude: 407113723, longitude: -749746483
Feature called '' at latitude: 402133926, longitude: -743613249
Feature called '' at latitude: 400273442, longitude: -741220915
Feature called '' at latitude: 411236786, longitude: -744070769
Feature called '211-225 Plains Road, Augusta, NJ 07822, USA' at latitude: 411633782, longitude: -746784970
```

Q3: Two basic server-client pairs using the same proto file but with different languages.

We defined a *"service.proto"* file to get weather updates from both:

1. Python server
2. Java server.

The Python client can ping both the Python and the Java server to get weather updates and a greeting message (same for a Java client).

The *"service.proto"* file is shown below.

```
syntax = "proto3";

package grpc_example;

option java_package = "com.example.grpc";
option java_outer_classname = "ServiceProto";

// Define the request message
message RequestMessage {
  string name = 1;
}

// Define the response message
message ResponseMessage {
  string message = 1;
}
message WeatherRequest {
  string city = 1;
}

message WeatherResponse {
  string update = 1;
}

// Define the gRPC Service
service MyService {
  rpc GetResponse(RequestMessage) returns (ResponseMessage);
  rpc GetWeatherUpdates(WeatherRequest) returns (stream WeatherResponse);
}
```

We define two RPC methods

1. Unary Request-Unary Response (greeting message)
2. Unary Request-Streaming Response (weather update)

How to run the code and relevant details of the code and containerization techniques are discussed in the README.md file of the project.

Q4 + Q5: The Failure Detection and Failure Dissemination component of the SWIM protocol.

For our implementation we used Python to write the Failure Detection service of each node and Java to write the Failure Dissemination component of each node. We define a common .proto file for both the Failure Detection service and the Failure Dissemination component of each node. The service we define in the swim.proto file is shown below.

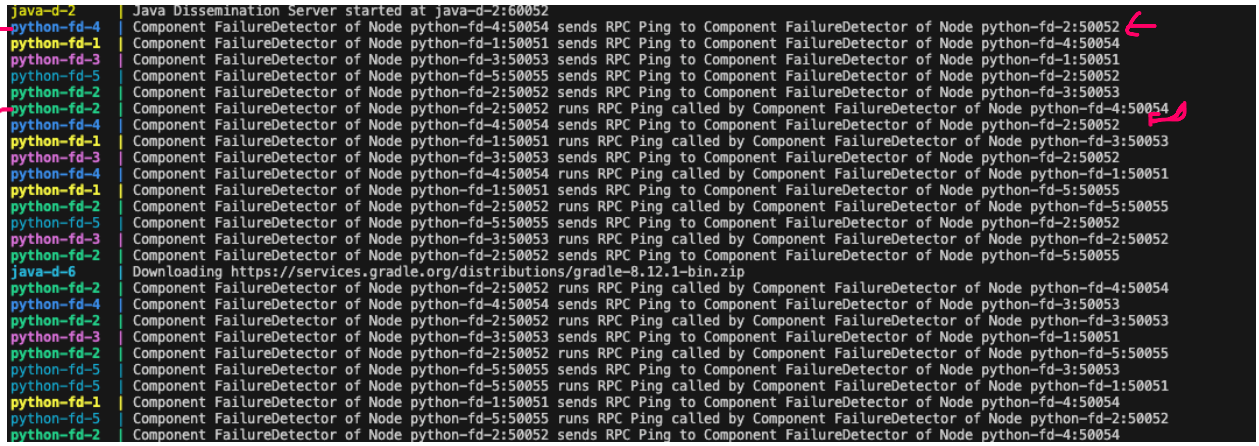
```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "swim";

package swim;

// Failure Detector Service
service FailureDetector {
  rpc Ping (PingRequest) returns (PingAck);
  rpc IndirectPing (IndirectPingRequest) returns (IndirectPingAck);
  rpc RemoveFailedNode (FailedNodeRemovalRequest) returns (FailedNodeRemovedAck);
  rpc JoinNewNode (NewNodeJoinRequest) returns (NewNodeJoinAck);
}

// Dissemination Service
service Dissemination {
  rpc NotifyFailure (NotifyFailureRequest) returns (NotifyFailureAck);
  rpc Join (JoinRequest) returns (JoinResponse);
}
```

a) A node uses the **rpc Ping** in the **FailureDetector** component *(written in Python)* to periodically contact another node in its membership to determine if it is alive.
b) If a source node does not get a response for a target node using the **rpc Ping** it uses the **rpc IndirectPing** in the **FailureDetector** component to contact other nodes in its membership list to ping the target node.
c) When a node gets a positive acknowledgement from a target node, it updates its last_heard_from timestamp for that target node.
d) When a node does not get a response for both **rpc Ping** and **rpc IndirectPing** for a target node within a timeout, it marks the target node as failed.
e) Then the source node removes the failed node from its membership list and uses the **rpc NotifyFailure** in its **Dissemination** component *(written in Java)* to let other nodes know that a particular target node has failed.
f) The nodes getting their **rpc NotifyFailure** invoked also remove the failed node from their respective membership list.

g) When a node gets their **rpc NotifyFailure** invoked they use the **rpc RemoveFailedNode** in the **FailureDetector** component *(written in Python)* to remove the failed node from the **FailureDetector** component of that node as well.
h) When a new node wants to join the cluster, they invoke the **rpc Join** in the **Dissemination** component of a bootstrap node and they get added to the cluster once the new node gets the **JoinResponse** from the bootstrap node.
i) Then the bootstrap node uses the **rpc Join** to multicast **JoinRequest** of the new node to all the other nodes in its membership list so that all the nodes update their corresponding membership list.
j) When a node gets their **rpc Join** invoked they use the **rpc JoinNewNode** in the **FailureDetector** component *(written in Python)* to add the new node in the **FailureDetector** component of that node as well.

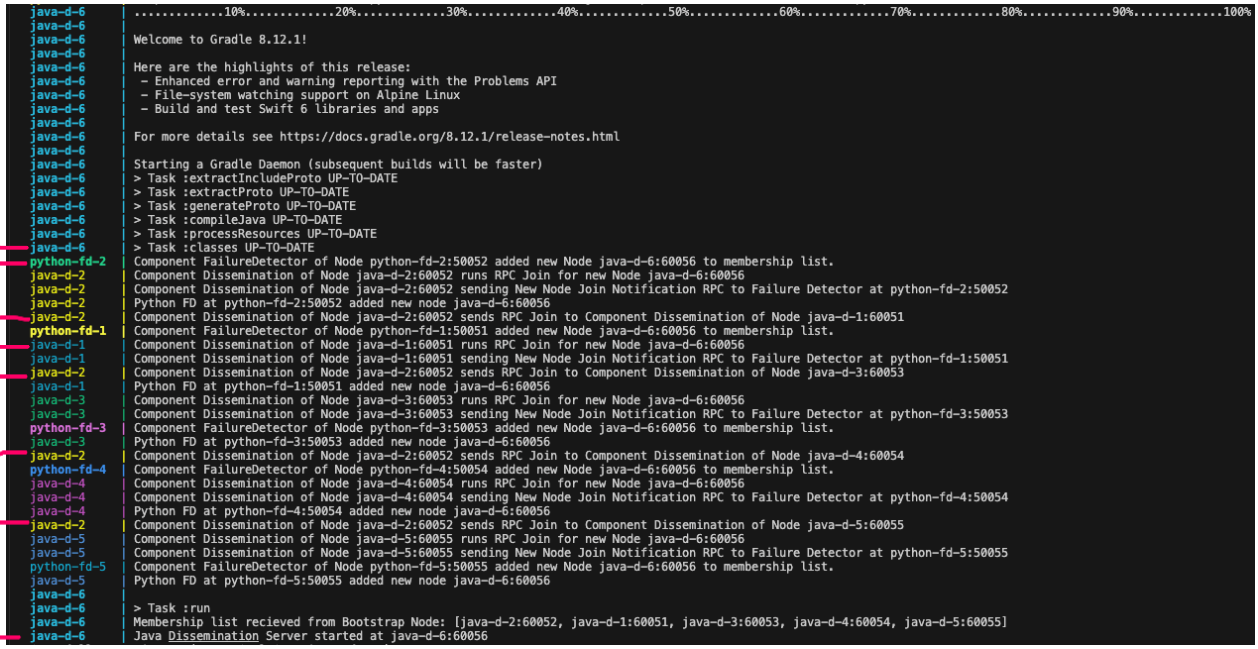
Q6: Test Cases

Test Case 1: When all the nodes are up and running, each node pings a target node in its membership list and gets a positive acknowledgement and updates their last heard from timestamp for that target node.



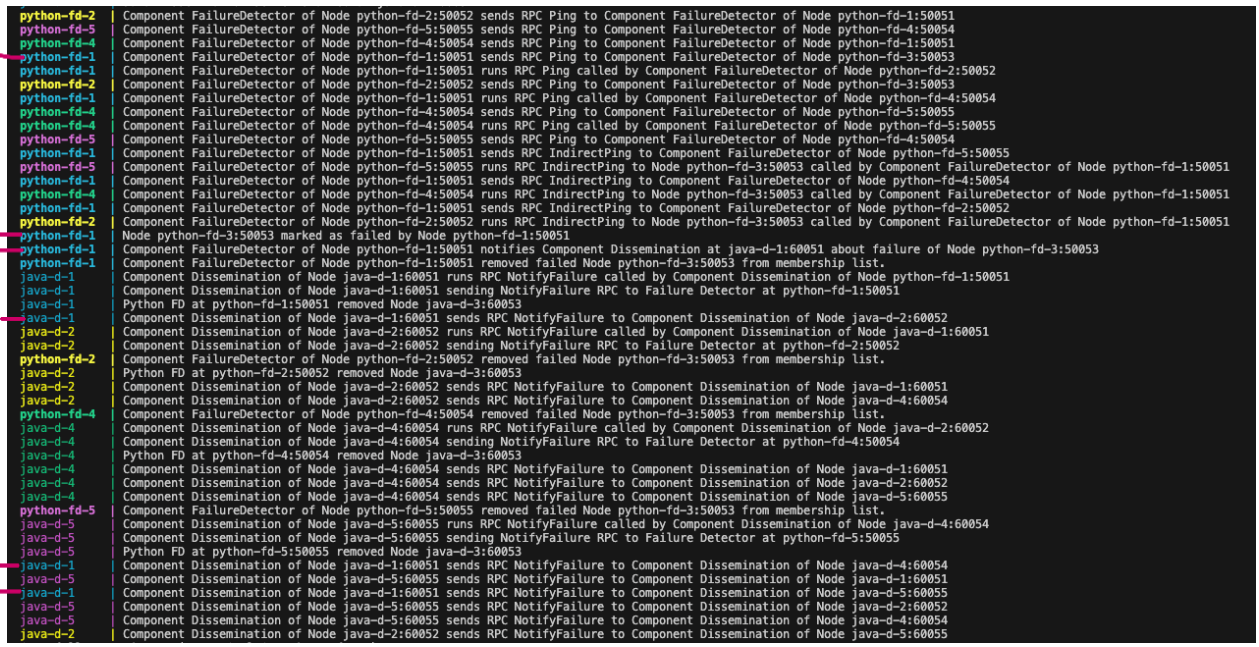
- Here, node4(python-fd-4) sends ping request to node2 (python-fd-2) and node2 runs the Ping Request and Acknowledges it.
- Same goes for all other Ping Requests by all other nodes.

Test Case 2: When a new node joins the cluster, the bootstrap node responds with an updated membership list and multicasts a join notification to all members in the cluster. Each node then updates its membership list accordingly.



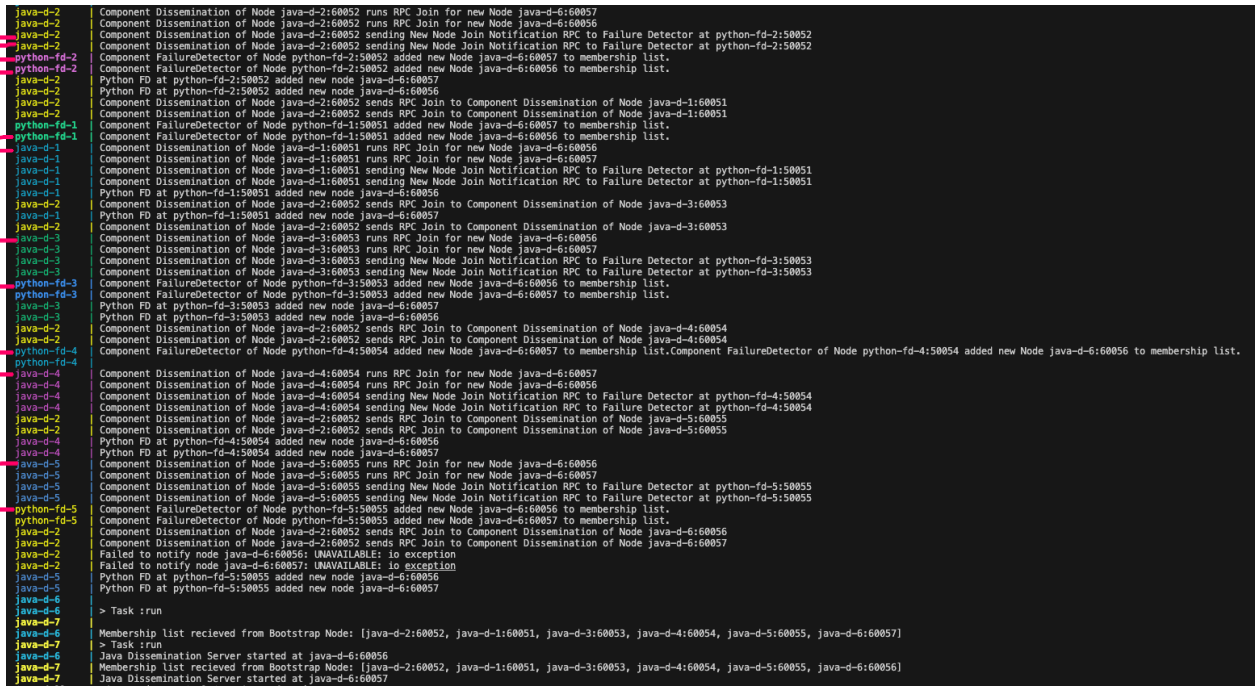
- Here the Dissemination Server of New Node (java-d-6:60056) started and invoked the RPC Join for the bootstrap Node (java-d-2:60052).
- Java-d-2:60052 at first disseminates the join Notification (RPC JoinNewNode) to its own Failure Detector Component (python-fd-2) and updates the membership list to add java-d-6:60056.
- Then bootstrap node multicasts the join Notification to all other nodes in the membership list (java-d-1:60051, java-d-3:60053, java-d-4:60054, java-d-5:60055) and every node updates their membership list in both Dissemination and Failure Detection Component.
- After that, bootstrap Node returns with an updated membership list to the new Node. Then, New node(java-d-6) print the recieved membership list.
- The print statement of Join Multicasting and updated membership list has been marked.

Test Case 3: When a node is detected as failed through IndirectPing, the failed node's ID is multicasted to all members in the cluster except the failed node and ensures that every node removes the failed node from their respective membership lists.

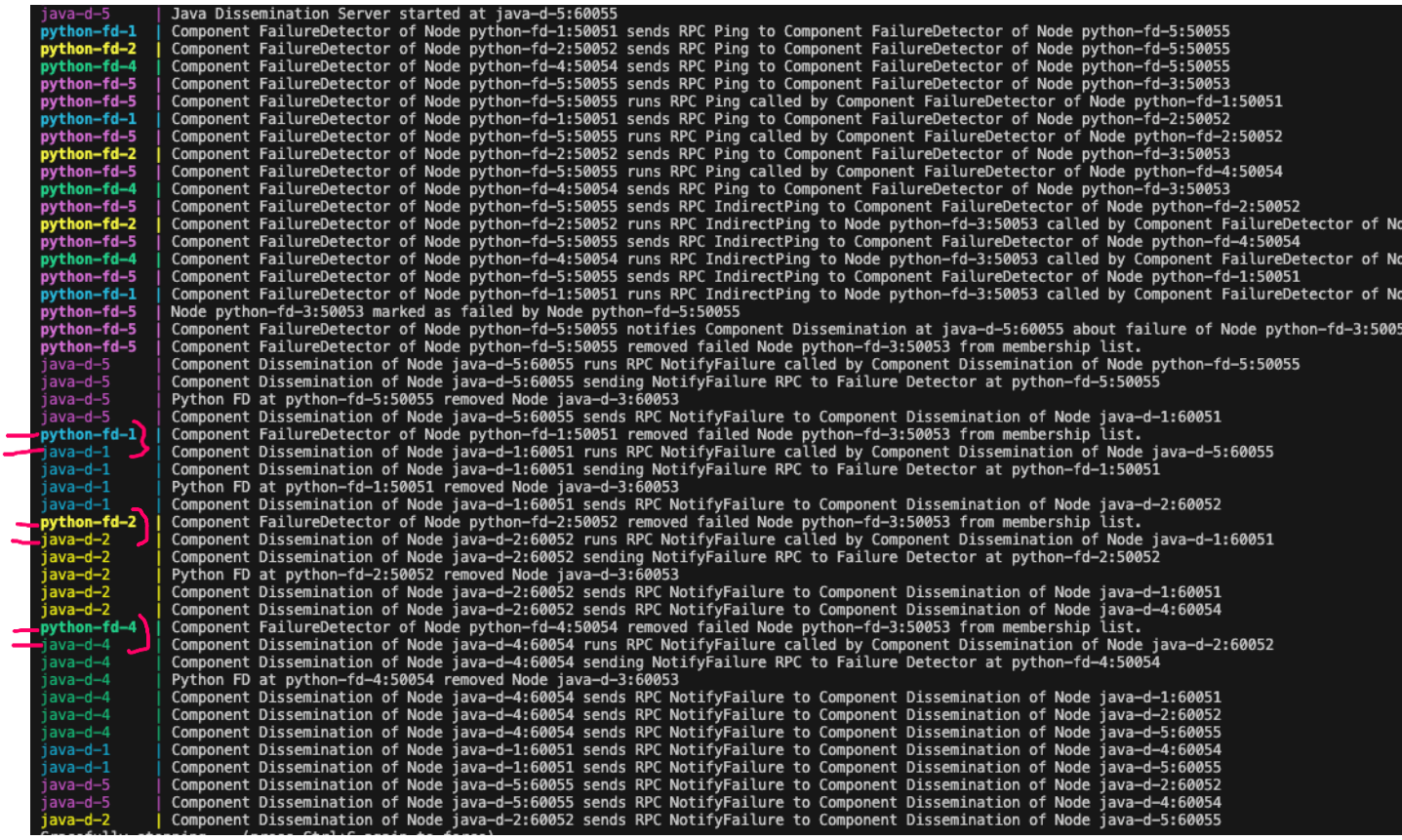


- Here, the node3 (python-fd-3: 50053) is down.
- Firstly, node1 (python-fd-1:50051) tries to contact python-fd-3: 50053 through RPC Ping and RPC IndirectPing.
- After no reponse, the node3 is marked as failed by node1.
- After that, Failure Detector Component of node1 notifies its own Dissemination component (java-d-1:60051) about the failed node3.
- Then, Dissemination component of node1 multicasts the failed node to all other nodes (node2, node4 and node5)
- The print statements where the Failure is Multicasted has been marked.

Test Case 4: Maintain synchronization between the Failure Detection and Dissemination Components of a node. When the Dissemination Component receives a failure/join notification and updates its membership list, the Failure Detection Component also reflects the same update. For example, if n1 receives a failure notification for n3, it removes n3 from the membership lists of **both** components.

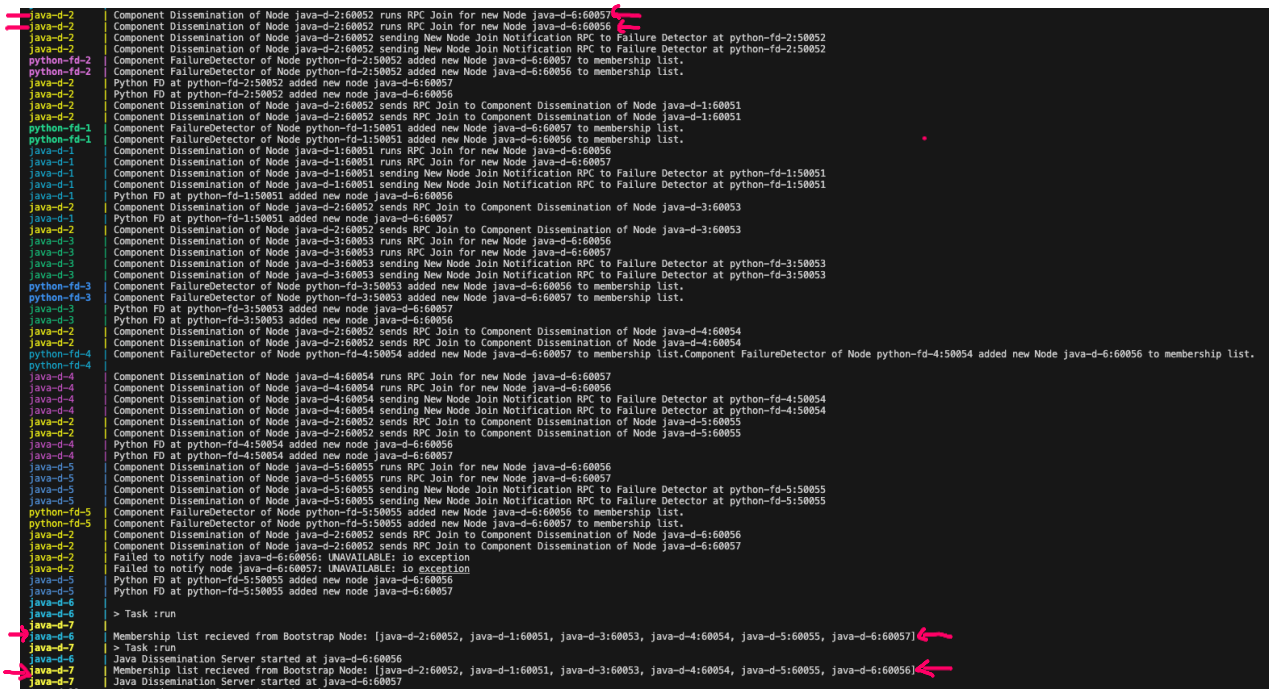


- When a new join request is invoked and multicasted, each node not only updates its membership list for Dissemination Component, but also updates membership list in the Failure Detection Component
- Example: When Java-d-1 runs RPC Join, the python-d-1 also runs NewNodeJoin and both update their membership list
- When java-d-3 runs RPC Join, the python-d-3 also runs NewNodeJoin and both update their membership list.
- So, the communication between Failure Detection Component and Dissemination Component is synchronized.



- Same goes for Failed Node Removal.
- When java-d-1 runs Notify Failure and removes the failed Node (java-d-3) from its membership list, python-d-1 also removed the failed node.
- When java-d-2 runs Notify Failure and removes the failed Node (java-d-3) from its membership list, python-d-2 also removed the failed node.
- So, the communication of failed node removal is synchronized among both the Failure Detection and Dissemination Component.
- All the relevant print statements are marked

Test Case 5: When 2 nodes want to join the cluster simultaneously, they receive a consistent and synchronized membership list.



- When node7 (java-d-7:60057) and node6 (java-d-6:60056) both want to join the cluster simultaneously and invokes the bootstrap node (java-d-2:60052), they receive a synchronized and consistent membership list.
- That is, node7 has node6 in its membership list and node6 has node7 in its membership list.
- The relevant print statements have been marked.

Contributions

a. Fahim Shahriar Khan: Q1, Q2, Q3- Java Server and Java Client, Q4-monitor nodes (server-side code), Q5- Multicast Failure, debugging of the whole code, containerization, Q6- Building test cases, Screenshot taking and marking
b. Nowshin Tabassum: Q1, Q2, Q3- Python Server and Python Client, Q4-Ping Indirect Ping (client-side code), Q5- Multicast Join, debugging of the whole code, containerization, Q6- Building test cases, Explaining the screenshots.

Implementation

Our implementation is available in our GitHub repository along with the README.md file.