Bryan Mullen

National College Of Ireland

Student #21154490

Distributed Systems - GRPC

CA - Report

Scenario and Services

The main scenario for this project is based around smart farming. The applications have been created to simulate the operations of a smart automated environment based around farming. A total of three services have been created - one based around Milk Production, one based around Feed Monitoring, and one based around Report Generation. The services are interconnected by means of client applications which connect to server applications. It is envisaged that each server application would be connected to one or more sensors which would allow for the reading of and generation of meaningful values, eg weighing scales, volume sensors etc. For the purposes of demonstration, these readings have been simulated by means of random value generators where appropriate.

The server devices publish themselves by means of Multicast Domain Name System (MDNS) implemented in the dependency JMDNS. The corresponding client devices then discover these services using the same technology. All communication is done over gRPC and all methods have been implemented in the Java programming language.

The scenario for each of the services is outlined in the project proposal document which has already been submitted to NCI as part of the first set of deliverables.

Project Architecture

The project architecture differs slightly from the project architecture demonstrated in the tutorials. Gradle has been used as the build tool in place of Maven. The advantage of this is that there is far less boilerplate to navigate due the the fact gradle does not use xml to define dependencies. It also gives access to convenient tools for building proto files and integrates very well with the developers choice of IDE - IntelliJ. The build architecture also allows for very clear differentiation between auto-generated files which are created by the generateProto script, and files written by the developer. In particular, auto-generated files are stored in the 'out' directory which is not commited to source control. This allows for a far smaller package to be commited and means that commits only track meaningful changes to the code, as opposed to changes generated by scripts. These files can always be rebuilt upon pulling the repository by simply running the generateProto script. This is the same architecture that is implement in Clement-Jean's excellent gRPC for Java course which the developer referenced in implementing this technology.

Two further deviations in structure have been made. The developers 'Server' files are only responsible for creating and tearing down the server, and implementing interceptors, SSL, and mounting the actual service implementation. This differs from the demonstrated content which placed these responsibilities in the same file as the actual method definitions. In the case of this project - these method definitions occur in the ServiceImpl files, which extend the ServiceImplBase files. These files are responsible for implementing the actual remote procedures and generating responses to requests. Finally, In both the case of the client and the server files, a separate app class has been created which is repsonsible for running the client or server. This is in line with the taught architecture in Semester 1.

Service Definitions

Milking Service

The milking service is defined in the milking.proto file located at ./src/main/proto/milking.proto. In this file it is defined that the file should use the proto3 syntax, export multiple files, the java package is defined to match the project structure and the java outer class name is defined as MilkingService. 6 message types are defined for the 3 services - a request for each service and a response for each service. While it is currently true that the request for each service in this file contains the same information, this might not always be true. In the interests of allowing for backward compatibility in the future without causing conflicts, these have been explicitly defined.

The milk collection procedure is envisaged to allow the client to read the current volume of liquid in the milk collection container, to easily allow the caller to know whether it is nearing the need to be emptied. It is

envisaged some kind of phyical sensor would be attached to this unit. This method has been implemented using the server streaming method, since it is envisaged the client would want to send a single request and be able to monitor the level on an ongoing basis. At the request stage this method accepts the name of the person checking - assuming there are multiple people working on the farm and responsibility requires attributing. At the response phase the server returns the current volume of collected milk, the capacity of the unit, and a boolean to tell whether this unit is currently full or not.

The milk production procedure is envisaged as being able to log the milk produced by a particular cow in the current session. To fully implement this in a working capacity there would require some kind of database or persisted log of the milk produced from which this method could read, but for the purposes of this example, these values are simulated. The method has been implemented using the Unary method, since this could be a simple request/response call similar to a traditional REST API. At the request stage we submit the name of the checker for accountability logging. At the response stage the id of the cow currently being milked is returned as well as the milk volume logged for this session.

The milk current cow procedure is envisaged as a way to check which cow is currently being milked. To fully implement this there would need to be some kind of a sensor to read to cows tag to return the id, as well as some kind of persisted log to verify the start time of the current session. These values have been simulated for the purposes of demonstration. This method is implemented using the server streaming approach since it is envisaged that the caller would wish to send a single response and receive multiple updates back in case the current cow being milked changed. At the request stage the checker name is submitted for accountability logging. At the response stage, the id of the cow currently being milked is returned as well as the time this milking session started as well as the duration the current milking session has been going on. The start time and duration are both returned using google well-known-types as the data structure.

Feed Service

The feed service is defined in the feed. proto file located at ./src/main/proto/feed.proto. In this file it is defined that the file should use the proto3 syntax, export multiple files, the java package is defined to match the project structure and the java outer class name is defined as FeedPackage. 6 message types are defined for the 3 services - a request for each service and a response for each service.

The add to feed procedure is envisaged as a way to automatically open the feed chute to add more feed to the available feed for the farm animals. It is envisaged that some kind of mechanical device to open the chute on the server's command would be required. It is also assumed that the feed tray would have a sensor capable of returning the weight of the current feed in the tray, to allow the caller to know whether it needed replenishing or not. The service is assumed to be smart enough to prevent adding feed if there is a possibility of overflow. This method is implemented using bi directional streaming since it is assumed the caller may wish to add feed multiple times in a session and receive multiple responses back from the server to monitor whether the feed is full or not over a period of time. At the request stage the amound of feed to add is defined, as well as the name of the person adding the feed for accountability logging. At the response stage, the current feed mass in the tray is returned, as well as the total feed mass added in this particular call, as well as a message to allow the server to let the client know if the tray is full.

The current water available method is envisaged as a way to read the amount of water available in the water tray for the animals. It is assumed that some kind of sensor to measure the current volume of water would be connected to the server in order to implement this. For the purposes of example, these values have been generated using dummy values. The method has been implemented using server streaming since it is assumed that the caller would wish to send a single response and receive updates as the water level drops, until such time as it is empty, indicating that it should be refilled. At the request stage the name of the checker is submitted for accountability logging. At the response stage the current water is returned as a double to allow for fractional measures, as well as a boolean to display whether the water tray is currently empty or not.

The feed consumption service is envisaged as a way to calculate the rate of feed consumption on the farm between two given dates. It is assumed that some kind of persisted storage of feed logs would be necessary to implement this fully using a database, however for the purposes of example these values have been faked. This method is implemented using the Unary method since it is assumed that this can be achieved using a simple request/response similar to a tradional REST API. At the request stage the start date of the period to search and the end date of the period to search are submitted, as well as the checkers name for accountability logging. At the response stage a value for the amount of feed consumed during this period is returned as a double to allow for fractional values as well as a message. The Dates in the request are structured using google well-known-types for Timestamp.

Report Service

The report service is defined in the report.proto file located at ./src/main/proto/report.proto. In this file it is defined that the file should use the proto3 syntax, export multiple files, the java package is defined to match the project structure and the java outer class name is defined as ReportService. 4 message types are defined for the 2 services - a request for each service and a response for each service.

The cow report service is envisaged as a way to retrieve specific information about a particular cow. It is assumed that some kind of persisted database of the cows on the farm would exist to make full implementation of this possible. However for the purposes of demonstration these values are faked. This method is implmented using the Unary method since it is assumed that this would be a simple request/response similar to a traditional REST api. At the request stage the name of the checker is accepted for accountability logging and the id of the cow to retrieve the report of. At the response stage, the cowId is returned back to the caller with the cow name, weight and amount of milk that it has produced this month.

The herd report service is envisaged as a way to retrieve specific information about a number of cows in one go. It is assumed again that some kind of persisted database of the cows would exist to make this possible. The method is implemented using client streaming since it is assumed that the caller may wish to submit multiple calls and receive a single response back. At the requst stage the name of the checker is accepted for accountability logging and the id of the cow to check. At the response stage, a herd report response object is returned, but inside the herd report response there is a repeated field of multiple cow report responses, showing the embedding of one message type in another. The response returns the same data as a regular cow report for each cow, as well as an average weight and average milk produced for the batch of cows reported on.

Service Implementations

Server Implementations

Server Implementations have been achieved by creating a ServerBase abstract class to house common functionality of the servers, and then extending this class for each of the discrete Server Implementations.

The ServerBase abstract class is responsible for calling the methods to read the server properties from the properties files stored in the ./src/main/resources folder. These files contain information regarding the service type, name, description and port, and is used in the creation of the service registration with JMDNS. The server base is also responsible for triggering the registration of each service using the ServiceRegistration class. The ServerBase has a run method which starts the server, logs that the server was started to the console, initialises the registration and calls a further method to put the server in a listening state, awaiting requests. The serverListen method is responsible for putting the server in this listening state and initializing a shutdown hook which will gracefully tear down the server in the case that the JVM is shutdown. The graceful teardown includes unregistering the service from JMDNS, closing the JMDNS instance, and shutting the server down before killing the JVM.

Each individual Server file (MilkingServer, FeedServer, Report Server) is then responsible for accepting the properties file path and the bindable service (the ServiceImpl file) and passing those through the superclass

constructor to be initialized. The discrete Server files are also responsible for using the ServerBuilder to create an instance of a server, and configuring each server to use the relevant port, use SSL with the relevant public and private keys, mount the ServiceImpl instance, implement any interceptors for metadata, and instantiate the server. Finally the servers run method should call the superclass' run method.

Each discrete Server file has a ServerApp file which is used to kick off the process. These files simply create an instance of the Server and call the run method.

Client Implementations

Similarly the Client implementations are based on a common ClientBase abstract class which merges shared functionality. This ClientBase file is used by both CLI and GUI clients alike. The ClientBase class is responsible for accepting the properties file of the service to be discovered, and using them in the getService method to search the network and discover the relevant IP and Port of the service in question. Once the service is discovered, the ClientBase class uses this to create a channel for the service. This is done using the NettyChannelBuilder, which allows setting of SSL context for client-server authentication, using the servers public key. The discrete client files then implement logic for sending the request to the server by means of either a blocking or async stub, depending on the method in quesiton. The GUI clients also implement logic for the creation of the user interface in order to provide some input validation and allow the user a pleasant user interface to call the server from.

MilkingServiceImpl Implementation

The MilkingServiceImpl class contains the logic for handling all requests to the MilkingServer. This file extends the MilkingServiceImplBase class which is automatically generated by the protobuf file using the generateProto script in gradle. It implements 3 override methods, one each for each RPC defined in the proto file.

The milkCollection method is implemented using server streaming. Variables are initialised to track the current volume and whether the unit is full or not. As long as the unit is not full, the current volume will be updated by calling a getCurrentVolume() method - this method would in reality call a sensor to find this information. It then sends a response by calling the responseStreamObserver's onNext() method and passing in a response object using the response object builder pattern. For the sake of demonstration, a period of time is allowed to elapse between responses. Once the unit is full, the logic breaks out of the loop and calls the responseStreamObservers onCompleted() method to finish the response.

The milkProduction method is implmented using the Unary Method. Variables are initialised to track the current volume as well as the last logged volume. These values would be taken from some kind of persisted storage in reality but here dummy methods are called to generate them. A reply is built using the MilkProductionResponse builder pattern, and the milk volume logged is calculated by subtracting the current volume from the last logged volume. As this is a unary method expecting a single resonse, the responseStreamObserver is simply called with a single response object before calling the observer's onCompleted() method to close the procedure.

The milkCurrentCow method is implemented using server streaming. A variable is initialised to pull the cow currently being milked - in reality this value would be pulled from a sensor but here we use a dummy method to populate this value. A timer is started to track the current session using the Timestamp type's builder pattern. Here it is assumed that we wish to receive responses at a rate of 1 per second for 6 seconds. Therefore within a for loop the current time is stored using a similar Timestamp, and a response is generated using the MilkCurrentCowResponse's builder pattern and setting the duration to a value calculated as the difference between the two timestamps, wrapped in a Duration builder. As this is a server streaming call, we can call the responseStreamObserver's onNext() method multiple times (In this case 6) to send multiple responses. Once we break out of the loop we finally call the observer's onCompleted method to close the procedure.

FeedServiceImpl Implementation

The FeedServiceImpl class contains the logic for handling all requests to the FeedServer. This file extends the FeedServiceImplBase class which is automatically generated by the protobuf file using the generateProto script in gradle. It implements 3 override methods, one each for each RPC defined in the proto file.

The addToFeedAvailable method is implemented using Bi Directional Streaming. First an if check is run to check if the current weight of the feed tray plus the requested additions (as well as the amount added totally in this call) would exceed the capacity. If this is the case a response is immediately sent using the responseStreamObserver's onNext() method and the AddToFeedResponse's builder pattern to let the caller know that no feed has been added and the tray is full. In the case that the tray currently is not full then the addToFeed private method would be called. In reality this would trigger feed being added but here this is just simulated. The totalFeedAdded variable is updated and a response is send to the client using the responseStreamObserver's onNext() method. In the case of an error on the server, the responseStreamObserver's onError method is fed back the throwable so that the client will receive the error. In the case of the server's onCompleted method being called, the total feed added will be logged and the responseStreamObserver's onCompleted() method will be called to close the procedure.

The currentWaterAvailable method is implemented using Server Streaming. Variables are initialised to track whether the tray is empty or not, and what the current volume of water in the tray is. While the tray is not empty the method continuously updates the variables by calling the getWaterVolume() method which in reality would be drawn from a sensor, and then comparing this value to the water capacity constant. In the case that there is less than 10% water in the unit, it will be considered to be empty. This is to allow for margin of error and to allow time for it to be refilled. Once the unit is empty the loop breaks and the responseStreamObservers onCompleted() method is called to close the procedure.

The feedConsumption method is implemented using the Unary Method. First variables are initialised to track the start and end date in Timestamp format. Next, a database is queried feeding in these two values as parameters. In reality a real database would be called but here a simple dummy method is called to simulate this. Once the feedLogs are retrieved a response is built using the FeedConsumptionResponse's builder pattern. Since this is using the unary method only a single response can be sent, so the response is simply fed into the responseStreamObserver's onNext() method before immediately calling the onCompleted() method to close the procedure.

ReportServiceImpl Implementation

The ReportServiceImpl class contains the logic for handling all requests to the ReportServer. This file extends the ReportServiceImplBase class which is automatically generated by the protobuf file using the generateProto script in gradle. It implements 3 override methods, one each for each RPC defined in the proto.

The cowReport method is implemented using the Unary Method. Note in the proposal this was originally intended to be client streaming, however upon reflection a decision was made that Unary would be far more appropriate as we have no need to multiple requests to find data relating to a single cow. Variables are initialised to store the cow's weight, milkVolume and name. These values would be pulled from persistant storage such as a database. For example purposes simple dummy methods have been created to simulate these calls. Since this is using the unary method only a single response can be sent, so the response is simply fed into the responseStreamObserver's onNext() method before immediately calling the onCompleted() method to close the procedure.

The herdReport method is implemented using Client Streaming. As such we immediately return a requestStreamObserver and place the logic inside the onNext(), onError, and onCompleted() methods. In the requestObservers onNext() method we initialize variables for a particular cow, to store the weight, milkVolume and name. Here we use the same helper methods as the cowReport() method, except this time we are doing it in a loop for each cow. Still within the loop, a cowReportResponse object is initialized and populated with the relevant data. It is then appended to a herdReport list which is used to store all the cowReport objects before later

constructing the herdReport response. In the requestObservers onError() method we simply feed the throwable back to the client so that errors are returned to the client. In the requestObservers onCompleted() method a herdReportResponse object is created using the relevant builder pattern. Variables are initialized to track the average milk volume and weight. Next the list of cowReport objects is iterated over and used to calculate the total milk volume and weight, which in turn is used to calculate the averages. Finally these values are set on the response, the response is built and the responseStreamObserver's onNext() method is called with the response before calling the responseStreamObserver's onCompleted() to close the procedure.

Naming Services

Service Registration

Service registeration is handled by a dedicated class which accepts service properties taken from a properties file located in ./src/main/resources/<service>.properties. These properties are read in to the ServerBase class using the dedicated PropertiesReader class which creates the properties object by reading in the file and then returns them in a single object.

The ServiceRegistration accepts the service type, name, description and port as parameters in the constructor and initializes constants to store each of these. The register method then creates a JmDNS instance using the localhost address. Note - on mac this InetAddress.getLocalHost() method fails unless the wifi is turned off on the developer's home network for reasons unconfirmed. However it works fine on other networks, and has been tested to run stably on linux. Next the jmdns.registerService() method is called with the service info to register the service on the network.

Two other methods are implemented. The getJmDNS() method is used in the ServerBase file to get a handle on the JmDNS instance and gracefully shut it down in the shutdown hook. The getServiceInfo() method is used in the same ServerBase file to allow messages to be logged to show which service is shutting down.

Once the service is registered it can be verified to be running using the mac's Discovery application available on the Apple Store which displays active mdns services on the network. This is useful for troubleshooting.

Service Discovery

Service discovery is handled by a dedicated class which accepts a string of the service type to search for. Again, these properties are programmattically read in by the ClientBase file from the properties file using the PropertiesReader class. In the case of discovery we only read the serviceType, and use this to search the network for a corresponding service. First a serviceInfo variable is initialized to null. Next a JmDNS instance is created using the localhost. Next a service listener is created using a helper class, which implements the ServiceListener interface, designed to update the serviceInfo variable when a service is resolved. Next, the service listener is registered with JMDNS and left time to discover the service. Currently 1 second is allowed, which is adequate on the developers machine. In the case services are not being discovered in time, this value can be tweaked. In the case that no service is found a null object will be returned. Otherwise service info containing the host and port and other details of the desired service on the network will be returned, allowing the client to call the service without hard coding the host and port.

The abstraction of the registration and discovery logic into separate classes allows these methods to be reused across all three servers and clients, to reduce repeated code. In addition, the invokation of the serviceRegistraion and serviceDiscovery in the abstract ClientBase and ServerBase files allows even less repetition of code, keeping the repository more maintainable.

Remote Error Handling & Advanced Features

Error Handling & Input Validation

Where possible error handling has been implemented to prevent a fatal crash of the client or server in the case of an unexpected value being submitted, or something going wrong. In the case of reading in the properties file for example in the PropertiesReader class for example, this exception is thrown upwards to the ServerBase, to the Server, before finally being thrown by the ServerApp class. The reason this error is thrown is because without the properties file we cannot expect the server to function as expected and so it is better to return an error than to catch is in this case.

In other cases, try/catch has been used to allow the program to gracefully continue on and print error messages. For example in the ServiceRegistration class, when initializing the JMDNS instance, these lines are surrounded by try/catch. In the case that the jmdns instance is not created an error message will be printed to the log and the program will continue on.

In terms of remote error handling, in the case of the client streaming and bi directional streaming calls, in the event of an error in the request observer, the throwable has been returned back to the client so that the client can see the error message, for example in ReportServiceImpl and FeedServiceImpl.

Within the clients, remote error handling is heavily implemented such that the logs get updated with the error received by the onError() methods, and the text response area is updated with the error, allowing the client to see the error in question. This logic is implemented in the panels found in ./src/main/java/com/bryanmullen/services/client/gui/panels/<Panel Name>.

In terms of input validation, this is handled extensively in the Client Gui Panel Files. Panels have been chosen to force the user to input valid values (for example a Date Picker has been implemented so that a user can not possibly enter an invalid string for a date). Where text fields are used for other data types (for example for integers in the FeedAddToFeedPanel) the string is first verified to be a number using a simple regex and checking the text matches. In the case of this not matching, the UI will be updated with an error message.

Input validation also occurs on all text fields to ensure that they are not empty, making it impossible for the user to submit a request with empty values in these fields.

MetaData, Deadlines, Authentication

MetaData has been implemented in the project by means of interceptors. A client and a server interceptor have been written to demonstrate basic metadata and can be found at ./src/main/java/com/bryanmullen/interceptors/<Interceptor Class>.

The client interceptor works by implementing the io.grpc.ClientInterceptor interface. This has an interceptCall method which is overridden. This takes an outgoing request and calling a middleware function on it before allowing it to continue. In the case of this projects Client Interceptor, a string is initialized to track the hostname of the requesting computer, and is then populated with the localhost hostname. In the case that this fails, some error handling is in place to set this to 'Unknown'. Thereafter, the outgoing request headers are updated to include a new piece of metadata containing the hostname of the requesting machine. A second piece of metadata is added to include the timestamp. This process could be used in other ways also, for example transmitting a jwt or other information. The client interceptor is added to the requests via the stubs, which can be found in the GUI client panels - '.withInterceptors(new ClientInterceptor()).

The Server Interceptor acts in a corresponding way, by implementing the io.grpc.ServerInterceptor interface, which has an interceptCall method which is overridden. Within this call we create a string builder by concatenating a human readable sentence with details pulled from the header object of the incoming request, and feeding them to the logger before allowing the request to continue to its destination.

Deadlines have been implemented on all gui client calls. The deadlines work by exiting the request if the server doesn't respond after a set amount of time. For example in FeedFeedConsumptionPanel, a deadline is

implemented on line 116 with the syntax '.withDeadlineAfter(10, TimeUnit.SECONDS)', which means after ten seconds the request will be exited.This is implemented across all stubs in the client guis.

Authentication has been implemented allowing the client to verify the client by means of SSL authentication. This means that no other machine can claim to be the client machine without having the private key that matches the public key stored on the client. These keys are generated by a simple shell script located at ./src/main/resources/ssl.sh. The shell script generates a key pair using the openssl tool, valid for 1 year, for the host 'localhost'. This shell script has been verified to run stably on mac and ubuntu, but cannot be guaranteed to also work in windows. The key paid generated are found also in the resources folder as my-private-key.pem and my-public-key-cert.pem. These files are saved as pem files as this is the format grpc requires. These files are NOT committed to source control, and therefore these MUST be generated new if the project is cloned. It is generally considered bad practice to commit private keys to source control. If these keys do not exist the project will fail to launch.

These keys are used in the discrete server files (eg MilkingServer) when creating the server using the server builder. For example in MilkingServer, line 29, when calling the server builder we use the method .useTransportSecurity() to enable SSL encryption and pass in the public and private keys. Correspondingly, in ClientBase, when creating the channel on line 79 we use the NettyChannelBuilder to create the channel instead of the default grpc one, and then call the .sslContext() method passing in the public key of the server to it. This allows the client to authenticate that the server is who the server is claiming to be, as only the genuine server would have the private key which matches.

Client - Graphical User Interface (GUI)

The Client Graphical User Interface has been implemented using swing to create frames, panels, tabs etc in order to allow the user to view, control and invoke services and devices. There is a client for each of the three services. Within each GUI client, there is an individual tab for each RPC allowing for clear separation between the services and for a varying amount of inputs to be accepted for each RPC. The client guis can be found at ./src/main/java/com/bryanmullen/services/client/gui/<ClientGUI>.

Each GUI client first calls the superclass constructor of JPanel, from which each one extends. Thereafter a TabbedPane is created, allowing us to separate each service. Icons are loaded in from the resources folder using a helper method. Then a single panel is created for each RPC method needed in each GUI, so for example, the MilkGUIClient has 3 tabs as it has 3 RPC methods, wheras the ReportGuiClient has 2.

Each GUI client has a createAndShowGUI method which initializes a frame, sets the default close operation, adds the full GUI to the frame and packs it, making sure the frame will be large enough to accomodate the default size of the panels. It is then set to visible. Finally, each GUI class has a main method which allows the entire process of constructing the GUI to be kicked off, and from the user's point of view, 'launches' the program.

Behind the scenes, each panel is infact a client in its own right, extending from ClientBase, allowing it a dedicated channel. When the panels are initialized they first call the superclass method feeding in the path of the properties file for the relevant service. Thereafter, depending on the needs of the particular GUI a varying amount of labels, text fields, date pickers etc are created and added to the panel. Each panel has a send request button which is bound to an action listener. On clicking the button, and event is fired which calls the method relevant to this particular panel.

Within each of the client methods triggered by the click listener, the same logic that is implemented for the CLI client is executed with a number of tweaks specifically for the GUI, including updating the response textarea when a response is received instead of printing to the console, as well as updating the response textarea when errors are received as a means of error handling. Furthermore, each panel contains the specific input validation discussed earlier relevant to the panel itself, which is called before sending the requst.

Github

Git has been used from start to finish on this project At every stage, small defined commits have been made to ensure a detailed, regular commit history exists, as opposed to a last minute code dump. In total 99 commits were made over the course of the project, and the entire git history can be viewed by running the 'git log' command in the terminal to verify the existance of the commits. Furthermore, at each commit a descriptive comment has been left to make it clear exactly what will change if the repo is checked out at this location in time.

Github has been used as per the specification as a remote repository to mirror the local git repo to. The entire codebase has been commited to the remote repository and it can be verified that no updates have occurred since the due date.

The full github repository is available at https://github.com/bryansmullen/distributedSystemsCA

The github repository is a private repo and will only be available to invited collaborators as per the specification. To this end the repo has been shared with the lecturer Rohit Verma using his handle rohit253. Should there be any issues with this, please do not hesitate to contact to request access.

It is strongly advised if cloning the code base to do the following before trying to run the project:

- Generate a key pair using the ssl.sh script

- Use gradle to install all project dependencies

- Generate grpc files by running the generateProto script in gradle

Video Link

As per the specification a video demonstration not exceeding 10 minutes has been created and uploaded to vimeo to allow streaming for the lecturer without the need to download. The video is available at the following private link.

https://vimeo.com/739424140

Password: 6xELw-Uz