Implementing gRPC in the TuSoW project

Filippo Cavallari

Abstract

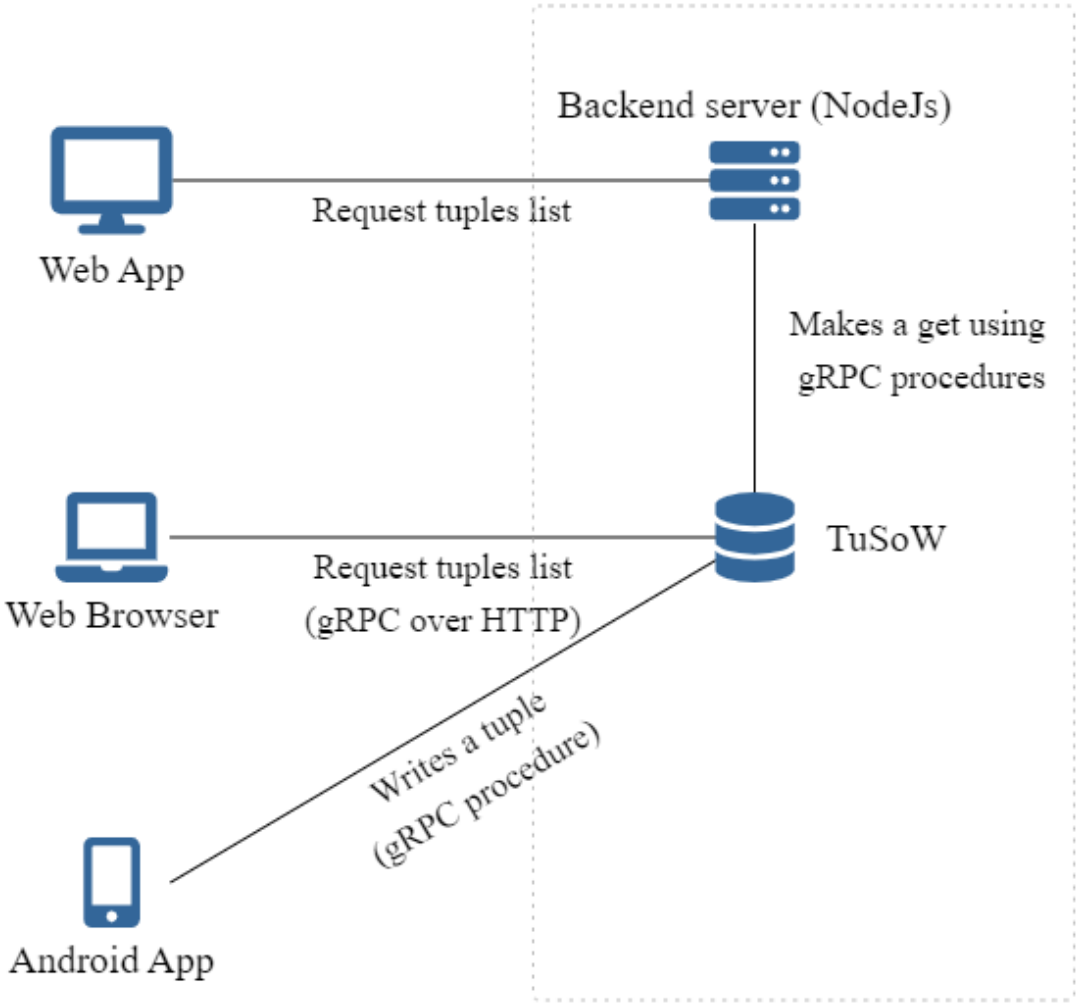
This project aims to add a new way of remotely accessing tuple spaces by using Google Remote Procedure Call [1]; by using gRPC it will be possible to use all the supported languages and platforms (i.e. Web, Android) giving more choices to the developers and also improving interoperability between different systems.

Goal/Requirements

As stated in the abstract section, the main goal is to add support for gRPC inside TuSoW and at the same time maintain the current modular structure of the project; if the project succeed, which means that all the unit tests pass, lots of new ways of accessing TuSoW will be available and this feature could be merged in the main repository on GitLab.

Scenarios

Let's take the following case diagram to clarify those concepts: we have three external actors (i.e. two http clients and an Android app) of which two of them want to retrieve all the tuples from a tuple space (e.g. classroom's students list) while the third one wants to add a new tuple (e.g. a new student); the first actor is using a Web App that communicates with a backend server (e.g. a NodeJs instance) using HTTP requests, so no uses of gRPC procedures yet; the NodeJs server uses its gRPC library to establish a channel with the TuSoW instance, makes a readAll request (gRPC procedure) and then forwards the response to the web app; the second actor makes a direct request to TuSoW using the javascript APIs generated by protoc; the last actor is an Android app that basically acts like the second actor but using auto-generated Java classes instead of JS APIs; in the last two situation a TLS authentication is highly recommended before calling gRPC procedures since they are not operating on localhost.



Requirements Analysis

To be able to actually comprehend the project a basic understanding of tuple spaces [2] and of protobuf [3] is required.

It is mandatory that the gRPC implementation does not alter the modular structure of the TuSoW project; that's necessary because not all the modules might be needed in an infrastructure to be able to use the main TuSoW's functionalities (accessing linda's tuple spaces from remote).

The project source code has to be published on a fork of the TuSoW repository on GitLab and, in order to merge it in the official repository, appropriate unit tests must be implemented.

The gRPC services must support working with both logic (prolog based) and textual (regex based) tuple spaces.

There are twentytwo procedures (eleven for each tuple space types) that have to be implemented:

- tuple space validation: used to verify that the given tuple space id matches an existing tuple space
- tuple space creation: used to create a new tuple space given the id and the type
- read: used to read a tuple from the given tuple space and matching a given template
- take: used to take (read and then remove) a tuple from the given tuple space and matching a given template
- write: used to add a given tuple to the given tuple space

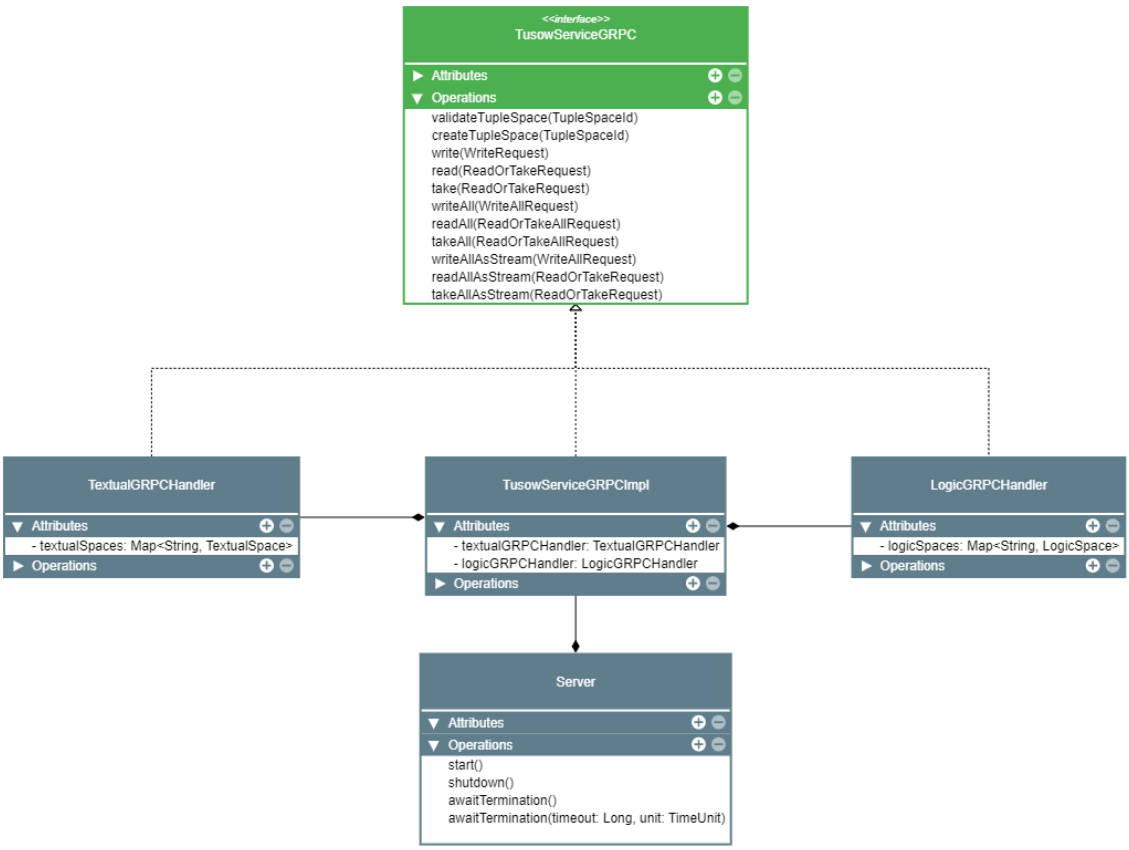
- read all/take all/write all: like the read/take/write procedure but accepts a list of tuples instead of just one tuple and the single responses are grouped in one list that is then sent as one unique response
- read all as stream/take all as stream/write all stream: like the read/take/write procedure but accepts a list of tuples instead of just one tuple and the single responses are sent back one by one as a stream

Design

Structure

Entities structure

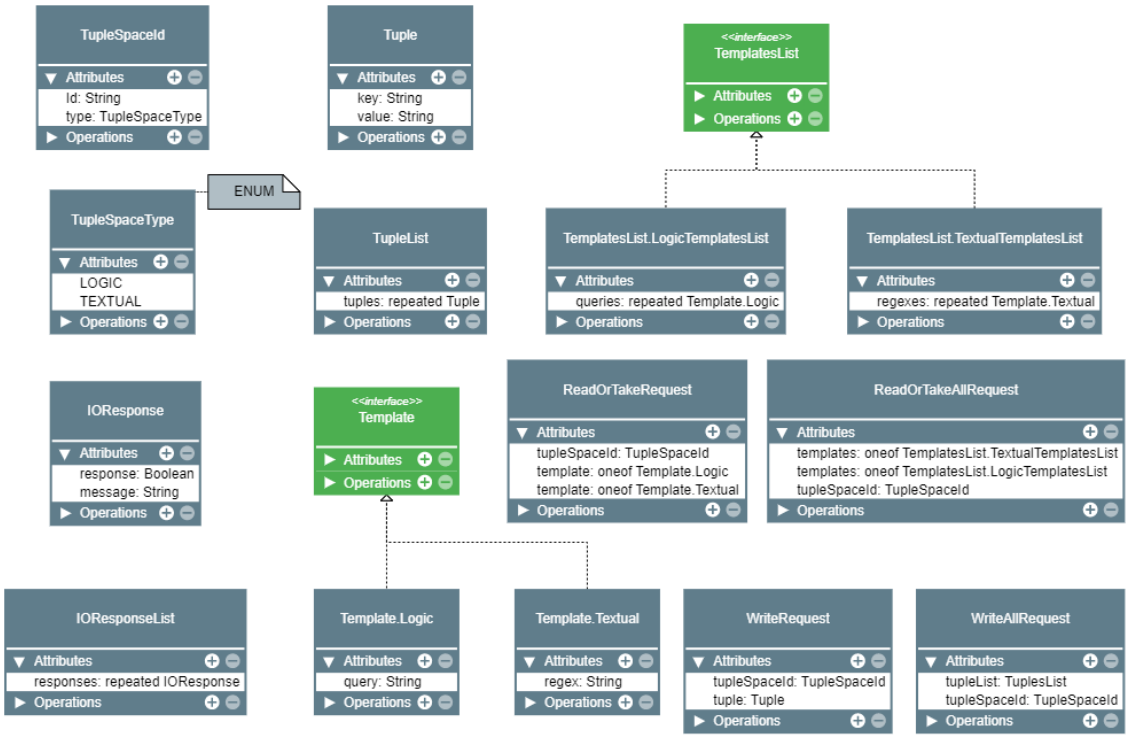
The design is heavily influenced by the interface generated from the protobuf file; truthfully that interface is already implemented by an also generated class which is then extended by the TusowServiceGRPCImpl class, but I have hidden such extension to simplify the UML (also because it doesn't really matter to show classes that are auto generated and cannot be modified).



The inherited methods of these classes allow to create new tuple spaces, verify if a tuple space exists and do I/O operations, even with multiple tuples per request (<write/take/read>All and <write/take/read>AllAsStream methods); a more detailed description of the methods is provided in the definition of the services (Services structure chapter).

The Server class is a wrapper of the gRPC's server class that simplifies its usage.

Messages structure



A quick background on the syntax used in the above UML:

- `repeated` means that the field might contain one value up to an infinite amount of values
- `oneof` means that the message can contain only one of the various *oneof* fields
- the `interface` entities are used to represent messages that include sub-messages
- the label `ENUM` is used to identify enum messages

TupleSpaceID

The *TupleSpace* message represents the ID of a *tuple space* and its type (*logic* or *textual*).

TupleSpaceType

That message is a simple *enum* used to classify tuple spaces types.

IOResponse

The *IOResponse* message is returned from input operations and gives a feedback on the outcome of such operation using a boolean flag and a string for a more detailed description.

IOResponseList

That message is a list of *IOResponse* messages created using the `repeated` keyword.

Tuple

The *Tuple* message abstracts the concept of tuple, defining two attributes, one for the template used to perform the query and one containing the associated value.

TupleList

That message is a list of *Tuple* messages created using the `repeated` keyword.

Template

The *Template* message defines two sub-messages:

- *Template.Logic*: represents a template to be used with a logic tuple space, in other words a query;
- *Template.Textual*: represents a template to be used with a textual tuple space, in other words a regular expression.

TemplatesList

The *TemplatesList* message defines two sub-messages:

- *TemplatesList.LogicTemplatesList*: a list of *Template.Logic* messages;
- *TemplatesList.TextualTemplatesList*: a list of *Template.Textual* messages.

ReadOrTakeRequest

That message is used to retrieve a value from a tuple space, thus it contains a field for specifying the target tuple space plus a template to perform the query.

ReadOrTakeAllRequest

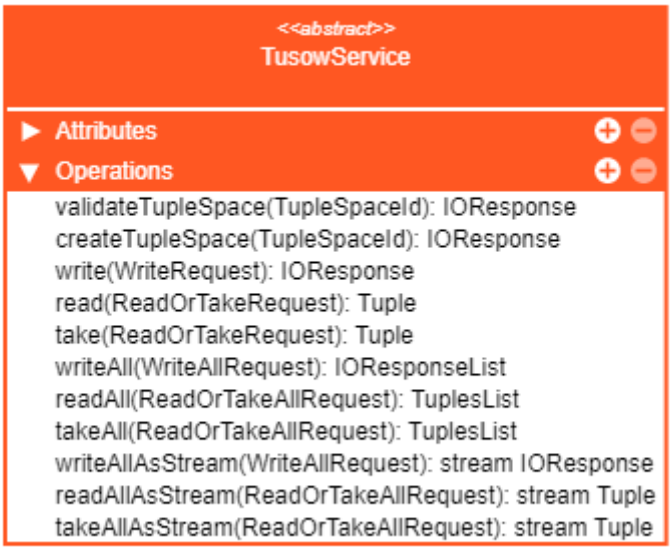
That message is very similar to the *ReadOrTakeRequest*, except that it contains a list of *Template*.

WriteRequest

The *WriteRequest* message contains a *Tuple* and the *TupleSpace* where the tuple will be added.

WriteAllRequest

Such message is very similar to the *WriteRequest* but differs by the usage of a list of tuples instead of just one tuple.



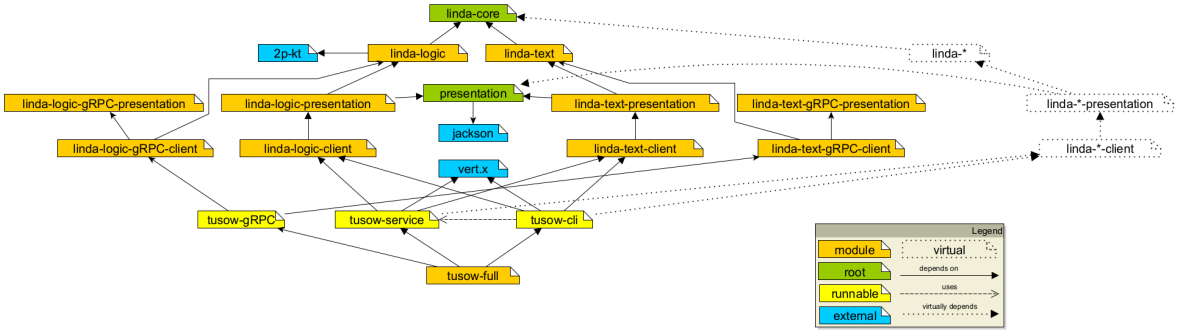
Services structure

The *TusowService*, as the name suggests, is not a *message* but a *service*, which means it defines procedures that can be called by a client stub. Those procedures are:

- `validateTupleSpace`: used to check if a tuple space as already been created;
- `createTupleSpace`: used to created a new tuple space;

- `write`: adds a tuple to a tuple space
- `read`: retrieves a tuple from a tuple space matching the provided template;
- `take`: retrieves and removes a tuple from a tuple space matching the provided template;
- `writeAll`: writes all the provided tuples in a tuple space;
- `readAll`: retrieves a list of tuples from a tuple space matching the provided templates list;
- `takeAll`: retrieves and removes a list of tuples from a tuple space matching the provided templates list;
- `<write/read/take>AllAsStream`: *streamed* variant of the *writeAll*, *readAll*, *takeAll* procedures; *streamed* means that the response is sent value by value instead of being grouped in a single list.

Modules structure



Initially the project was designed to have a total of five new modules, two on presentation level, two on the client level and one on the service level with their scope described in the next paragraphs:

tusow-gRPC module

The tusow-gRPC module will contain the service implementation the service methods will then use the classes of the linda-*-gRPC-client module, depending on the type of request (logical or textual).

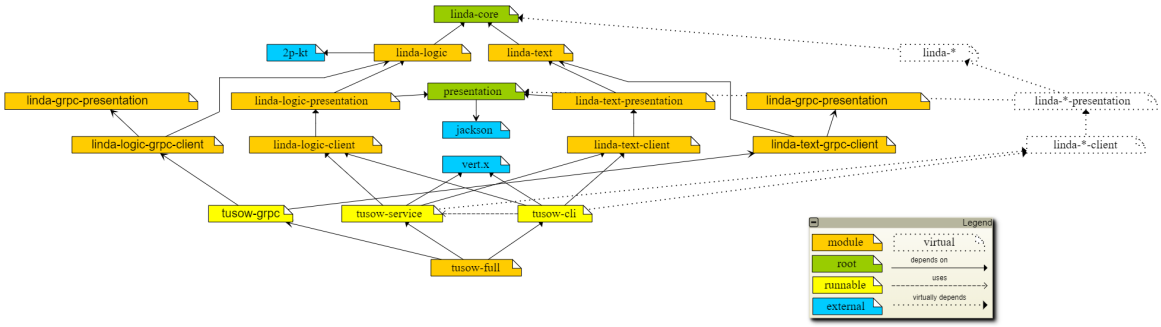
linda-*-gRPC-presentation

The linda-*-gRPC-presentation modules will contains the *.proto files where messages and services are defined.

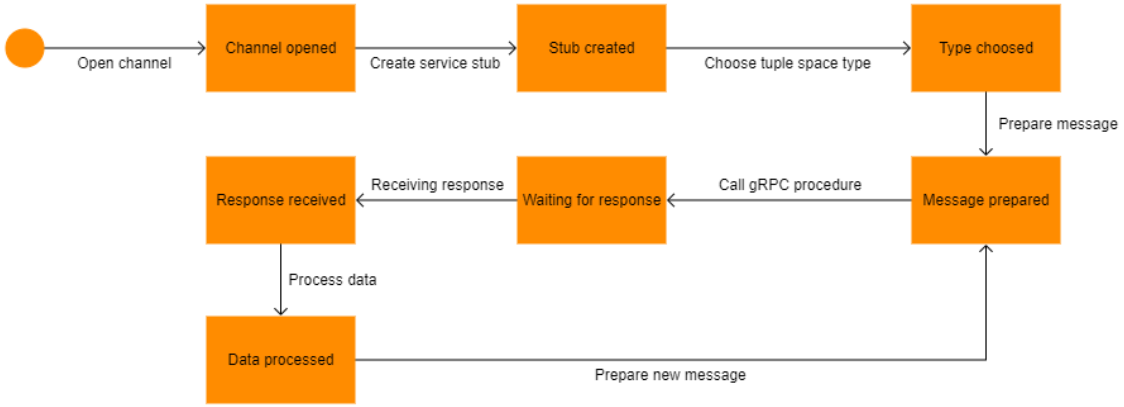
linda-*-gRPC-client

The linda-logic-grpcC-client and the linda-text-grpc-client modules will contains all the classes that are actually needed to perform operations on the tuple spaces; they will make extensive use of the already existing classes/interfaces of the upper layer (linda-*) and of the linda-grpc-presentation modules generated proto classes.

During the actual implementation I have encountered a complication related to how protobuf works, that is the absence of any kind of inheritance or polymorphism between protobuf messages; that forced me to create a unique presentation module used both by the logic and textual modules; the new project map appears as follow:



Behaviour

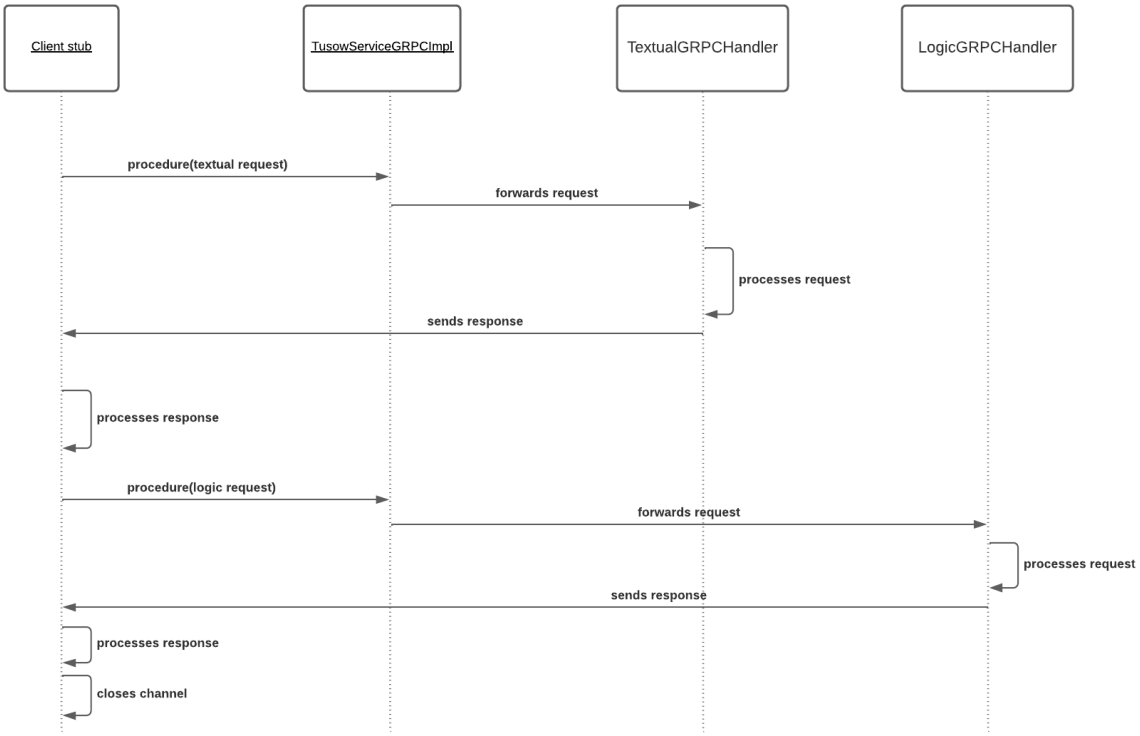


The service workflow is pretty straightforward:

1. at first a channel is opened (a channel is a connection between the client and the service's host based on TCP sockets)
2. then the channel is assigned to a new stub specific for the service being used
3. at this point the client has to choose the type of tuple space he wants to work with (logic or textual)
4. once the tuple space type has been chosen a message (specific for each procedure of the service) has to be prepared
5. the message is then sent via a call to the relative procedure
6. the client will be waiting for a response from the service
7. when such response arrives it will be processed by the client as it likes
8. the connection can then be closed or if further data are needed then a new message can be prepared and the workflow will repeat itself from point 5

Interaction

To avoid a too much complicated diagram all the possible procedures (e.g. write(), read(), takeAll(), ecc.) have been grouped in one generic *procedure(request)* message. The diagram explanation is pretty straightforward: the *client* calls a procedure providing a message containing the request data; the *TusowServiceGRPCImpl* class dispatch the request based on the tuple space type; one of the two handler classes (i.e. *TextualGRPCHandler* and *LogicGRPCHandler*) processes the request, creates a response message and sends it back to the *client*; upon receiving the response the client might want to call another procedure or terminate the process.



Implementation Details

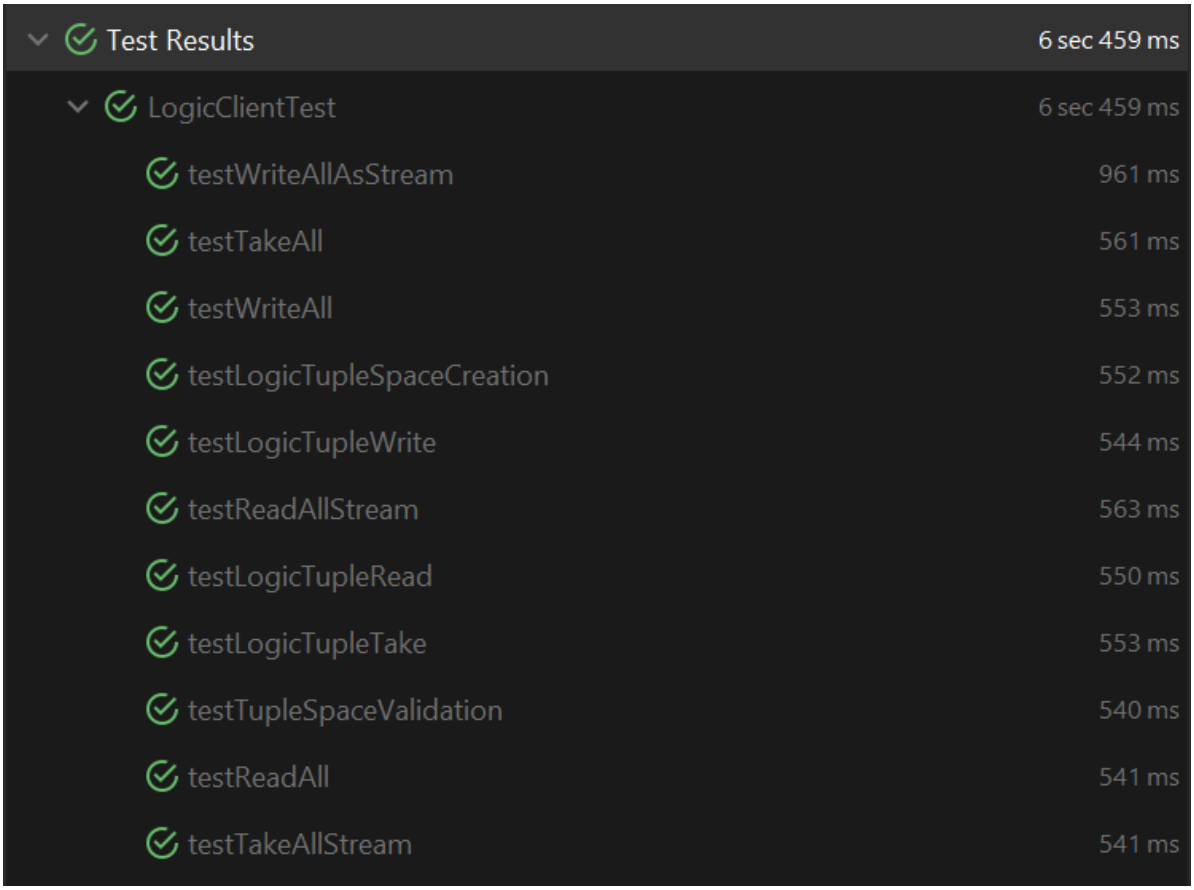
To make a clear distinction in the protobuf messages between logic and textual spaces, specific fields and children messages have been created (e.g. `Template.Textual` and `Template.Logic`); it's important to clarify this concept because by taking a closer look at the code it might seems that, since all the messages content is plain text, only one type of messages would have been necessary; that would have worked but at same time would have made the calls to the procedures more chaotic; with my approach the developer must choose a specific message depending on the tuple space type he wants to work on reducing the chance of making mistakes.

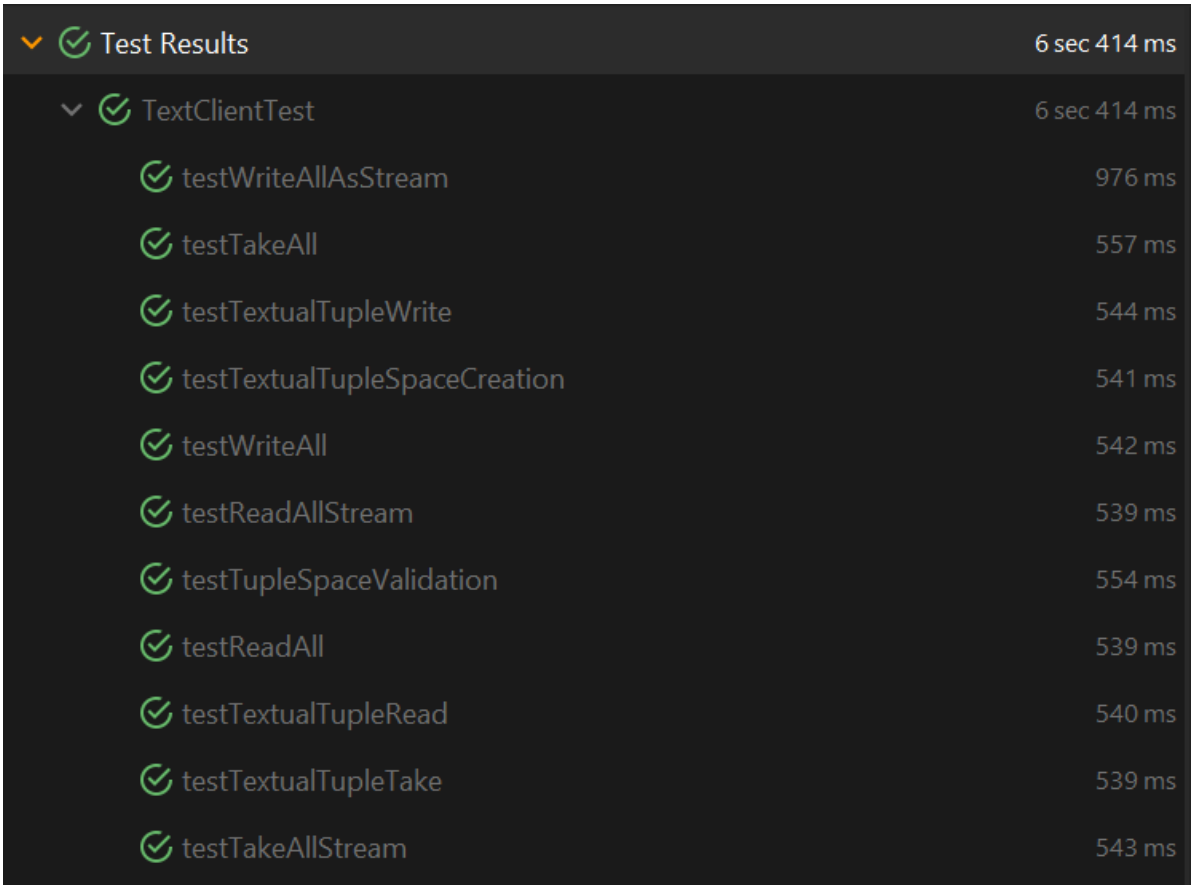
There are three classes that "implements" the TusowServiceGRPC interface, one for each tuple space type (i.e. logic and textual) and one that acts like a router by redirecting requests based on the tuple space type to previous two classes; if only one type of tuple space is required (like in the test classes), it is possible to create a server using the classes TextualGRPCHandler and LogicGRPCHandler, instead if both are required the class TusowServiceGRPCImpl must be used; they both uses the TupleSpace interface for interoperability between different implementations.

All the service elaborations prior to sending any response have been made asynchronous to avoid blocking the entire service in case of a task taking to long time to be completed.

Self-assessment

To maintain the high quality of the project I have created a test case for each procedure of the TuSoW service, divided in two classes, one for the logic related procedures and one for the textual ones.





As shown in the above pictures, all the tests have successfully passed.

Deployment Instructions

System requirements:

- Git
- Java 11+

To start a TuSoW service with gRPC capabilities you first need to clone the repository using `git clone https://gitlab.com/pika-lab/courses/ds/projects/ds-project-cavallari-ay2122.git tusow-grpc` and then run the following commands:

```
cd tusow-grpc
.\gradlew tusow-grpc:run -Pport=<port>
```

Usage Examples

The implementations differs from the language being used; in a scenario with the client based on Kotlin the first step will be to generate the Java classes from the protobuf file by using either the appropriate Gradle task or the command `protoc --proto_path=src --java_out=build/gen linda-grpc-presentation/src/main/proto/TusowGRPC.proto` (requires protocol buffer compiler installed); then an instance of a stub is required, which can be created like in the following example:

```
private fun createChannel(ip: String, port: Int) =
ManagedChannelBuilder.forAddress(ip, port).usePlaintext().build()

private fun createStub(ip: String, port: Int) =
TusowServiceGrpc.newStub(createChannel(ip, port))

fun main(){
    //Change ip and port to match your TuSoW instance ip:port
    val ip = "localhost"
    val port = 8000
    val stub = createStub(ip, port)
}
```

When the stub is created it will be possible to call any of the available service procedures, e.g. for creating a new tuple space:

```
[...]

val tupleSpaceType = TupleSpaceType.TEXTUAL //or TupleSpaceType.LOGIC
val tupleSpaceName = "hello world space"
val tupleSpaceId =
TupleSpaceId.newBuilder().setId(tupleSpaceId).setType(tupleSpaceType).build()

stub.createTupleSpace(tupleSpaceId, object : StreamObserver<IOResponse> {
    override fun onNext(value: IOResponse) {
        if(value.response){
            println("Success")
        }else{
            println("Failure. Reason: ${response.message}")
        }
    }

    override fun onError(t: Throwable?) {
        t.printStackTrace()
    }

    override fun onCompleted() {
    }
```

```
})}

[...]
```

Conclusion

I have spent the first hours studying how TuSoW and Linda worked, by looking and the code and reading the project paper, then I have defined the gRPC services and messages which are the foundations for the subsequent development of the project; with the protobuf classes generated I have defined the entities and their interactions using UML diagrams and started programming; for each implemented method of the service, I coded a specific test to be sure that everything was working properly.

Future works

I have a lot of ideas for hypothetical improvements to the code structure that I still need to evaluate (pros and cons) now that I am assessing the course of *programming and development paradigms*; one idea is to gather all the similar lines of code of the test cases in an abstract class and make the two test classes extend it and overriding only the methods that actually differ from it, thus respecting the DRY principle.

The whole TuSoW project really got my interest and, if allowed by the professors, I might keep working on the open issues on GitLab.

What did I learn

I got a deeper understaing of tuple spaces and improved my knowledge of gRPC and of the protobuf language; lastly but non the less important, I have got better at approaching large projects created by other people.

---

1. gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services. ↩

2. https://en.wikipedia.org/wiki/Tuple_space ↩

3. https://developers.google.com/protocol-buffers/docs/proto3 ↩