ReLINDA: Un'interfaccia per l'utilizzo di LINDA in remoto

Giacomo Tontini
giacomo.tontini@studio.unibo.it

Daniele Rossi
daniele.rossi18@studio.unibo.it

Febbraio 2019

Lo scopo del progetto è quello di modellare ed implementare un'interfaccia lato server per LINDA ed integrarla con gRPC.

Utilizzando gRPC un'applicazione client può direttamente invocare metodi su un'applicazione server remota come se la chiamata avvenisse su un oggetto locale, ciò favorisce lo sviluppo di applicazioni distribuite e l'integrazione di sistemi sviluppati in linguaggi di programmazione differenti.

Lato server, vengono esposti tramite interfacce i metodi che possono essere invocati e viene eseguito un processo (gRPC server) per la gestione delle richieste da parte del client.

gRPC agevola notevolmente il processo di definizione delle interfacce avvalendosi di un linguaggio IDL (Interface Description Language) chiamato **proto** (giunto alla versione 3). In tale linguaggio, metodi appartenenti al medesimo concetto applicativo vengono racchiusi all'interno dello stesso servizio.

Lato client, attraverso uno stub sarà possibile interrogare i servizi (con relativi metodi) disponibili sul server.

gRPC utilizza i **protocol buffer** per la serializzazione e la de-serializzazione dei dati, agevolando lo scambio di informazioni tra gli end point applicativi. Tramite i protocol buffer si definiscono le strutture dati che verranno fornite e restituite ai/dai servizi remoti.

L'interfaccia lato server per LINDA deve esporre le primitive di comunicazione essenziali:

- **in(template)** o **take(template) -** Legge e consuma una tupla, che corrisponde al template indicato, dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta è sospesa fino al suo inserimento.

- **rd(template)** o **read(template) -** Legge, senza consumarla, una tupla che corrisponde al template indicato dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta è sospesa fino al suo inserimento.
- **out(tuple)** o **write(tuple) -** Inserisce una nuova tupla nello spazio di tuple.

In aggiunta, potrà esporre:

- Le **primitive di accesso (take e read) predicative**, in grado di ritornare un messaggio di fallimento anziché sospendersi.
- Le primitive **bulk**, in grado di gestire più tuple alla volta.
- Le **primitive di accesso duali**.

1 Terminologia

1.1 Interface Definition Language - IDL

Un linguaggio di definizione dell'interfaccia (IDL), è un linguaggio di specifica utilizzato per descrivere le API (Application Programming Interface) di un componente software. Gli IDL descrivono un'interfaccia in modo completamente indipendente dal linguaggio di programmazione utilizzato per l'implementazione dei vari componenti software, consentendo ad entità che sono stati sviluppate con linguaggi differenti di comunicare. Si può pertanto definire come un linguaggio di interazione universale.

I linguaggi IDL sono comunemente sfruttati per le chiamate remote di procedure (RPC). In questi casi, le entità alle estremità della comunicazione possono potenzialmente appoggiarsi a sistemi operativi e linguaggi di programmazione diversi. IDL non si occupa della comunicazione, bensì solo ed esclusivamente della specifica di uno schema comune dei messaggi scambiati. La libreria gRPC utilizzata in questo progetto si avvale dei Protocol Buffers, descritti in seguito, per la definizione degli schemi dei messaggi e fornisce un'astrazione per la creazione di canali di comunicazione sui quali transitano tali messaggi.

1.1.1 Protocol Buffers

Protocl Buffers sono un'insieme di linguaggi IDL che forniscono dei meccanismi per la generazione automatica di sorgenti utili alla creazione dei messaggi e appositi canali tra client e sever utilizzati per la comunicazione in accordo con il linguaggio di programmazione adottato.

2 Obiettivi e Requisiti

L'obiettivo principale consiste nel realizzare un server che esponga i servizi necessari per operare su un tuple space rendendo tale interazione il più semplice possibile da parte

del client ed evitando che quest'ultimo debba conoscere i dettagli di funzionamento di LINDA per interagirvi. Si desidera pertanto realizzare un mapping delle primitive LINDA in corrispondenti metodi gRPC esposti attraverso opportuni servizi.

Verrà adottato un approccio model-first e test-driven, per questo, dopo aver accuratamente progettato il sistema, verranno sviluppati test automatizzati che ne guideranno l'implementazione.

Gli studenti si pongono come obiettivo quello di realizzare, per ognuna dalle tre primitive di base, un'interfaccia server gRPC che si integri con LINDA, correlata da una chiara documentazione ed un insieme di test automatizzati.

Nel caso in cui tutto ciò venga portato a termine in un numero di ore inferiore a quello richiesto verranno aggiunte:

- le primitive predicative;
- le primitive bulk;
- le primitive duali.

Ovviamente, essendo l'approccio adottato test-driven, lo sviluppo di ogni punto sarà guidato dai relativi test.

Per raggiungere l'obiettivo prefissato è necessaria la realizzazione dei seguenti punti:

1. Definizione delle strutture dati che modellano i messaggi di richiesta e di risposta al/dal server attraverso linguaggio *proto*.
2. Definizione dei servizi esposti e conseguente raggruppamento logico dei metodi di interazione seguendo le specifiche dei protocol buffers.
3. Realizzazione di unità di testing per verificare il funzionamento corretto dei singoli servizi sulla base delle interfacce precedentemente definite. In particolare, verranno sviluppati test per verificare la corretta comunicazione tra i due servizi gRPC Server e Linda Server Interface e che l'implementazione delle primitive LINDA sia coerente con lo stato dell'arte (ad es. test della semantica sospensiva di read e take, test sull'idem-potenza della read ...). Questi test guideranno l'implementazione del sistema, da qui l'approccio test-driven.
4. Implementazione degli handler delle richieste per l'integrazione con LINDA.
5. Verifica del corretto funzionamento globale del sistema mediante l'impiego delle API offerte da gRPC. Arrivati a questo punto dello sviluppo il sistema sarà già integrato con LINDA ed i test effettuati mirano alla verifica della corretta ed efficiente comunicazione gRPC.

2.1 Scenari

Nonostante l'obiettivo di questo progetto sia quello di curare lo sviluppo della parte server e non quello del client, l'utilizzo del sistema da parte dell'utente finale attraverso

quest'ultimo può essere descritto, a grandi linee, dal seguente diagramma dei casi d'uso [1].

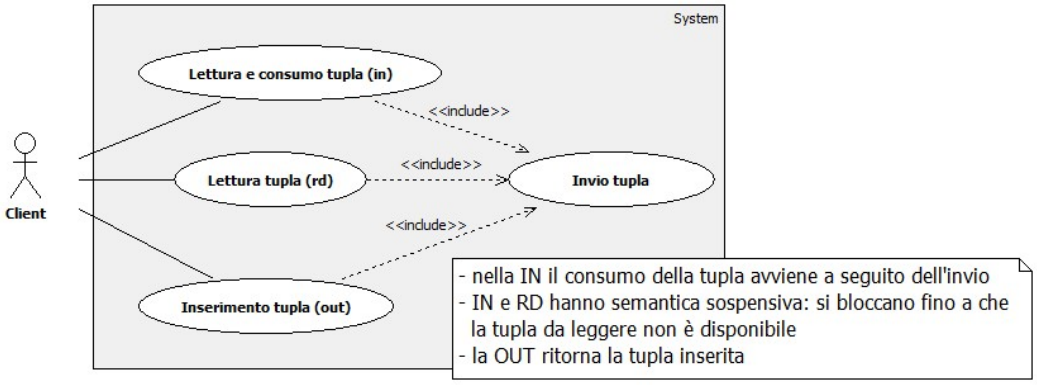

Figure 1: Scenari di utilizzo del sistema con primitive base

Inoltre, come si evince dall'immagine sopra-riportata, il sistema sviluppato esporrà le tre primitive base di LINDA, il diagramma dei casi d'uso di un sistema completo, ovvero che codifica anche le primitive predicative, bulk e duali, è riportato nella figura sottostante [2].

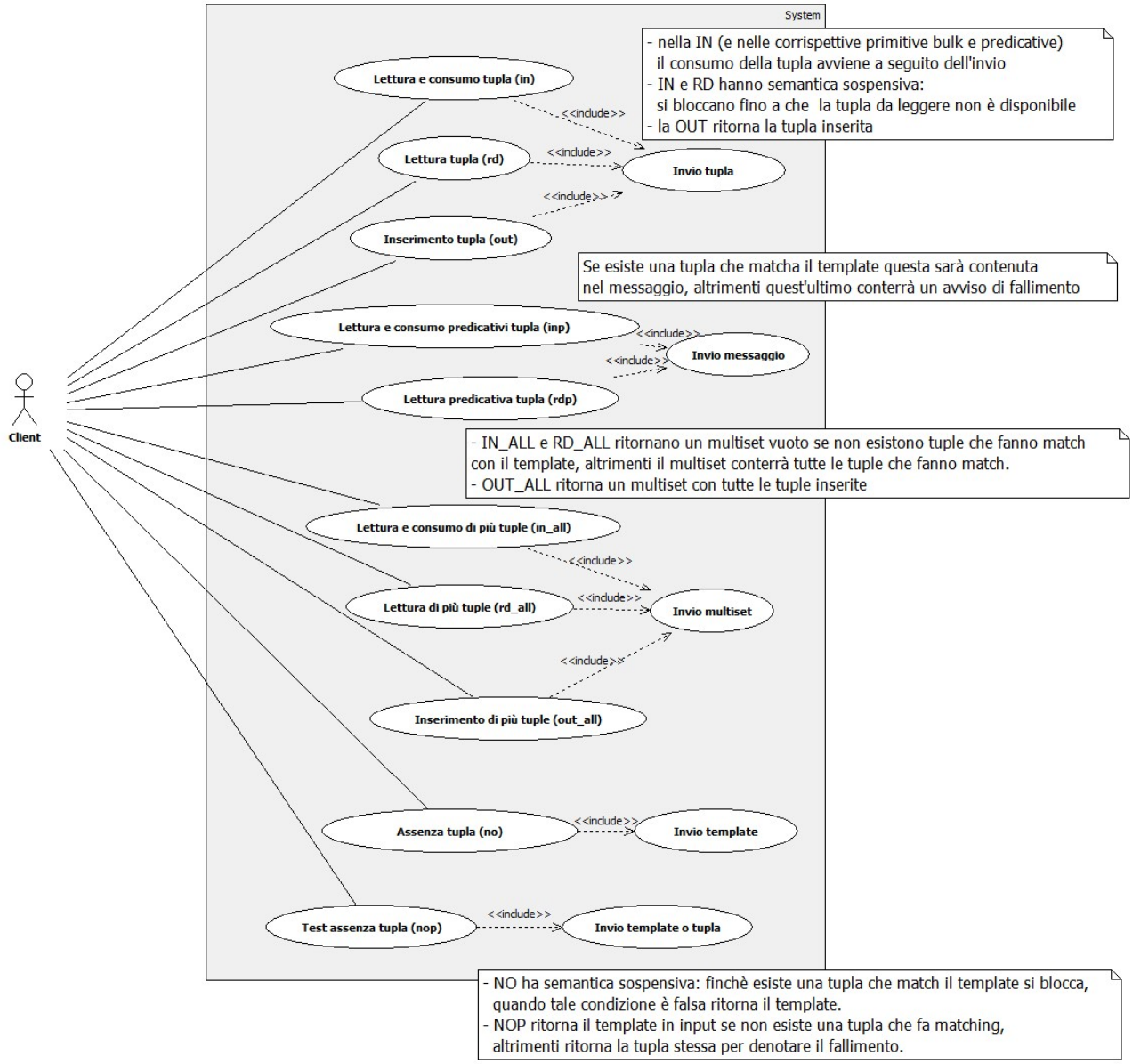

Figure 2: Scenari di utilizzo del sistema completo

Per quanto riguarda lo sviluppo della controparte client, agli sviluppatori sarà reso disponibile uno stub attraverso il quale interagire col server sviluppato, adottando come linguaggio di comunicazione quello del sistema (definito mediante specifiche *proto*).

2.2 Politche di autovalutazione

- La *qualità del software* prodotto dovrà essere valutata tenendo in considerazione il grado di incapsulamento raggiunto delle primitive LINDA all'interno dei moduli gRPC, all'attinenza del comportamento delle chiamate ai metodi remoti rispetto al comportamento atteso dalle primitive LINDA ed infine, il grado con cui i vari moduli software saranno suddivisi ed organizzati.

- L'*efficacia dei risultati* del progetto è definita dalla facilità con il quale sarà possibile interagire con il server, dal grado di efficienza con cui quest'ultimo sarà in grado di fornire risposte ai client (rapidità delle risposte, gestione elevata di richieste ...) e dalla complessità raggiungibile dai potenziali client sviluppati.

  I test sviluppati dagli studenti mireranno principalmente a verificare il grado di qualità del software.

3 Analisi dei requisiti

Risulta essere di fondamentale importanza determinare in modo accurato la struttura dei messaggi che andranno a mascherare le richieste LINDA, nello specifico, definire che astrazione utilizzare per gestire richieste e risposte che necessitano l'inclusione di più tuple nel proprio corpo. I protocol buffers mettono a disposizione differenti modalità per perseguire il medesimo scopo.

Il sistema nel suo complesso dovrà mascherare la distribuzione delle componenti, apparendo all'utente finale come se fosse un'unica entità locale.

Per quanto concerne la modalità di comunicazione tra client e server gRPC questa è stata implementata in maniera asincrona: il client a fronte di una richiesta restituisce una *CompletableFuture* che verrà completata al giungere della risposta da parte del server.

Tra i requisiti non funzionali di questo progetto figura la necessità di evadere le richieste da parte del client in modo veloce ed efficace quando queste non sono bloccanti.

4 Design

Essendo l'obiettivo di tale lavoro quello di effettuare il deployment di LINDA in un contesto remoto, il sistema risultante sarà un sistema distribuito. Come intuibile dalle considerazioni preposte in fase di analisi, al fine di mascherare la distribuzione del sistema all'utente finale è opportuno utilizzare diverse modalità di interazione tra client e server gRPC.

Poiché il progetto mira ad implementare le tre primitive di base si utilizzerà sia una comunicazione sincrona che asincrona; come da specifiche *gRPC* il tipo di richieste scambiate tra le entità sarà di tipo unario: il client invia una richiesta al server e riceve una singola risposta; questo non significa che il messaggio contenga un solo valore (potrebbe infatti contenere un insieme di elementi), ben diverso dal concetto di stream di risposte (si veda come riferimento Streaming RPC). Per le primitive *bulk* dovrà pertanto essere

effettuata una scelta tra le due opzioni appena descritte, tenendo presente come più elementi nella stessa risposta rallentino l'invio di quest'ultima fino a quando essa non sarà completa; nel caso dello stream invece si rende possibile una comunicazione fluente consentendo al client di elaborare progressivamente le risposte mano a mano che queste sopraggiungono.

L'architettura funzionale del sistema complessivo è riportata nella figura sottostante [3]. Le entità: *gRPC Server*, *Linda Server Interface* e lo spazio delle tuple *(Data Space)* sono in esecuzione su un unico host.

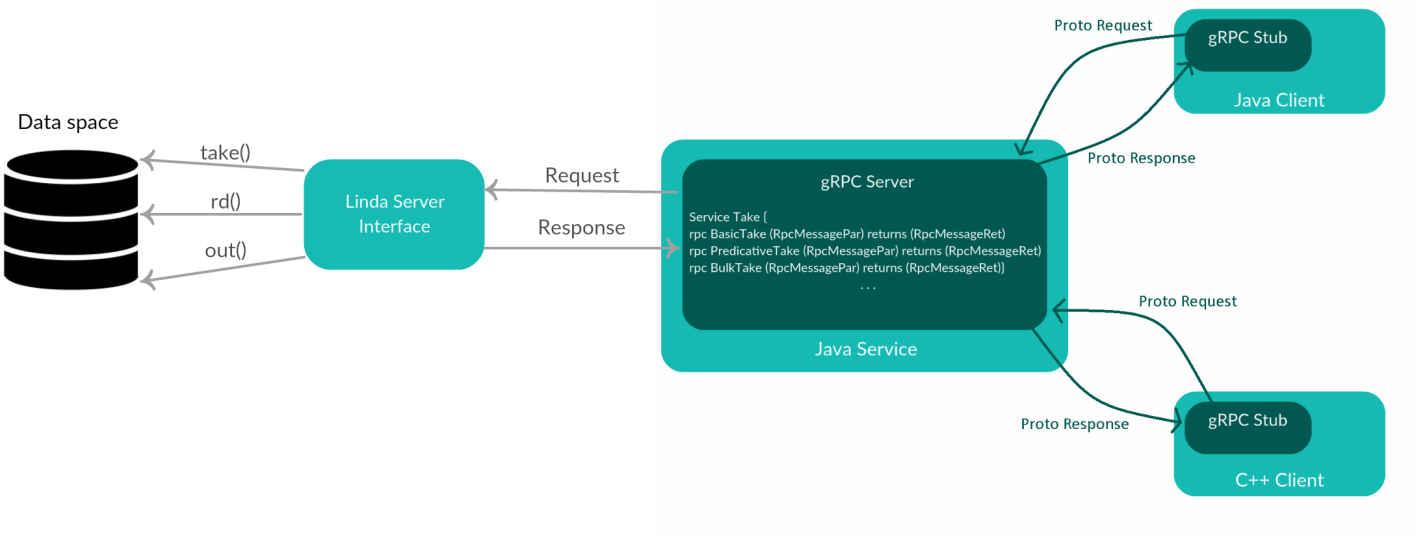

Figure 3: Architettura funzionale del sistema ReLINDA

In aggiunta, viene mostrata l'architettura client-server del sistema complessivo ad un livello di astrazione più alto [4].

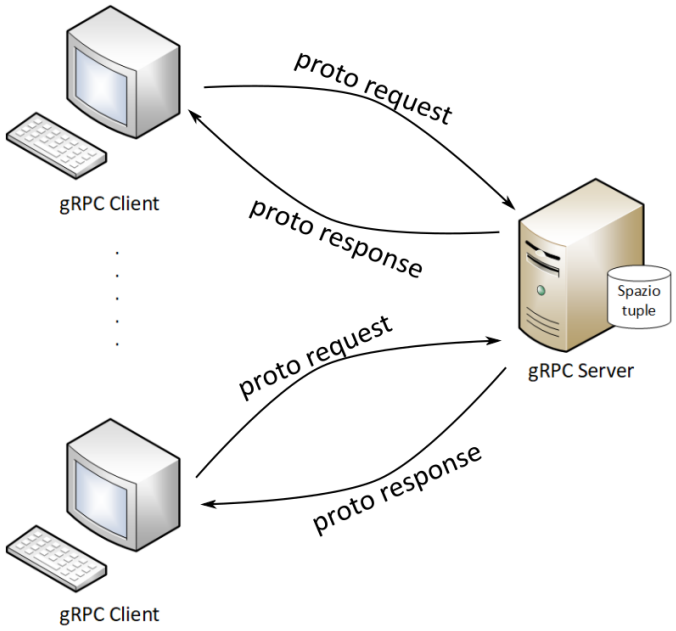

Figure 4: Architettura client-server del sistema ReLINDA

4.1 Architettura

Le principali entità che compongono il sistema in questione sono:

- Le classi **ReaderGrpc**, **WriterGrpc** e **TakerGrpc** che mettono a disposizione i metodi, utilizzati dai client, per ottenere lo stub dei servizi. Queste entità contengono al proprio interno rispettivamente: una classe astratta **ReaderImplBase**, **TakerImplBase** o **WriterImplBase** che devono essere estese definendo l'handler delle richieste **ReaderService**, **WriterService** e **TakerService** [5] . L'implementazione di tutte le classi presenti nei diagrammi riportati, eccetto le *Service, avviene in maniera automatica a seguito della compilazione, da parte del compilatore *protoc*, dei file *.proto* definiti. La struttura di tali file è definita al seguente link [struttura file proto].

- Le classi **LogicTupleSpaceApi** e **StringTupleSpaceApi** che espongono i metodi per inserire, leggere e consumare una o più tuple; in particolare, i metodi permettono di specificare, attraverso un argomento, se la primitiva è di tipo predicativo, bulk o negata. Queste classi sfruttano, rispettivamente, i metodi *getLogicSpace* e *getStringSpace* esposti da **TupleSpaces** per ottenere lo spazio di tuple a partire dal suo nome. Le classi *Service sfruttano **AbstractTupleSpaceApi** (che astrae LogicTupleSpaceApi e StringTupleSpaceApi) per realizzare l'integrazione di LINDA con gRPC. [6]

- La classe, modellate nel diagramma delle classi come **RelindaClient**, che fa uso degli stub offerti [7].

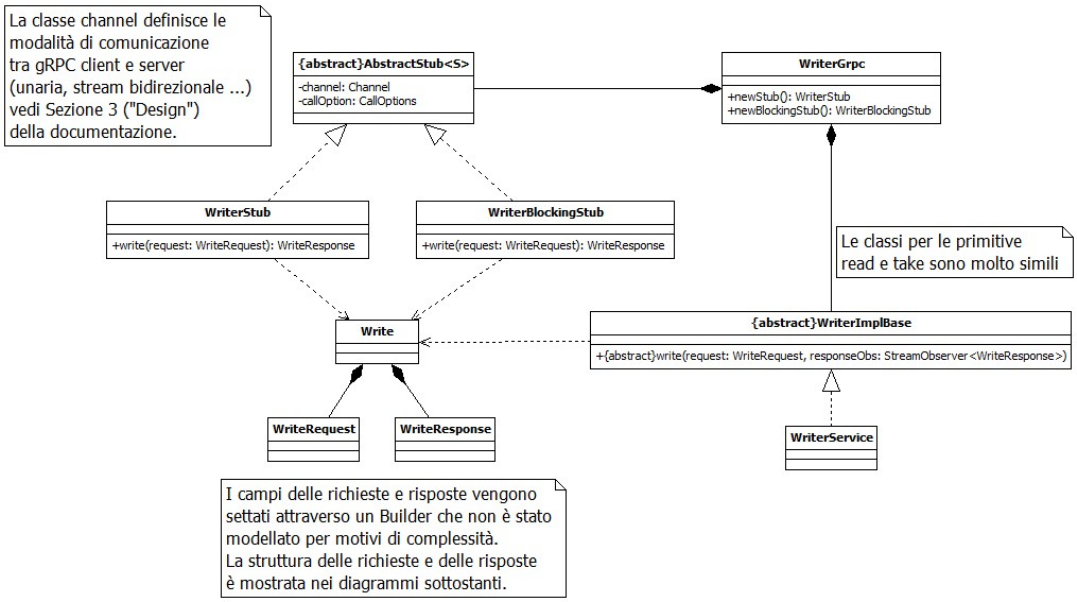

Figure 5: Diagramma delle classi utili per l'implementazione della primitiva write.

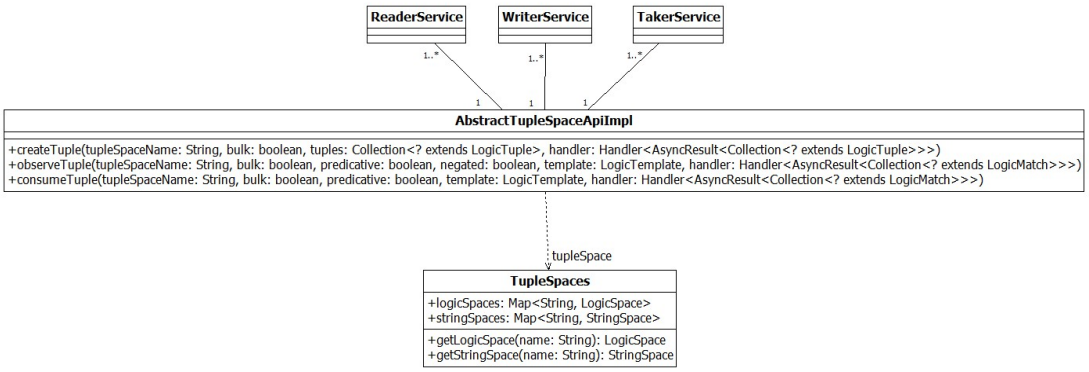

Figure 6: Integrazione tra gRPC e LINDA.

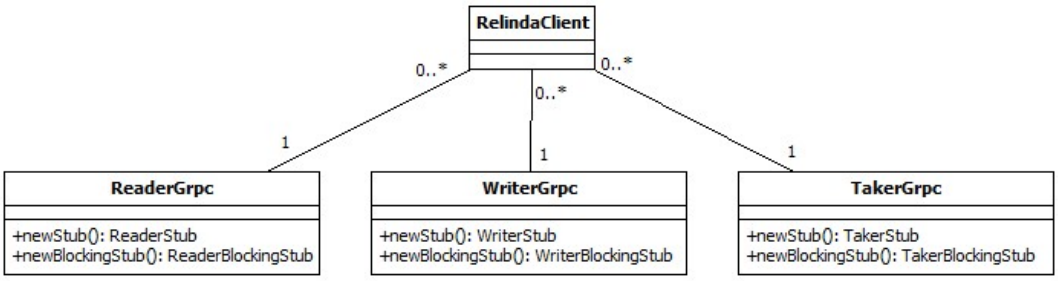

Figure 7: Relazione tra client e le classi che espongono i servizi gRPC

4.2 Comportamento

Il client gRPC inizialmente deve richiedere lo stub dei servizi affinché possa interagire con il server. Una volta fatto ciò, è abilitato ad inviare richieste gRPC che al proprio interno incapsulano i parametri della richiesta LINDA. Se la richiesta è sincrona, il client rimane bloccato in attesa di ricevere una risposta, a fronte della quale potrà effettuare nuove richieste ([8]).

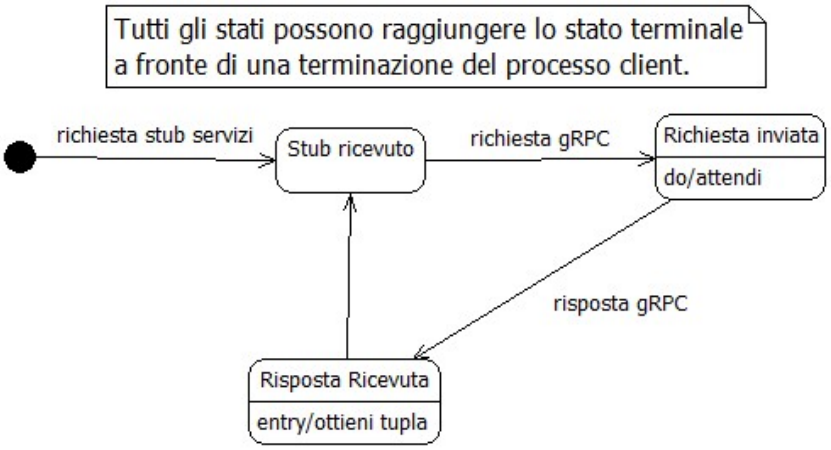

Figure 8: Diagramma di stato dell'entità client per una primitiva NON predicativa

Il caso di richiesta asincrona è stato modellato mediante diagramma di attività in quanto risulta maggiormente comprensibile il comportamento non bloccante delle varie entità [9].

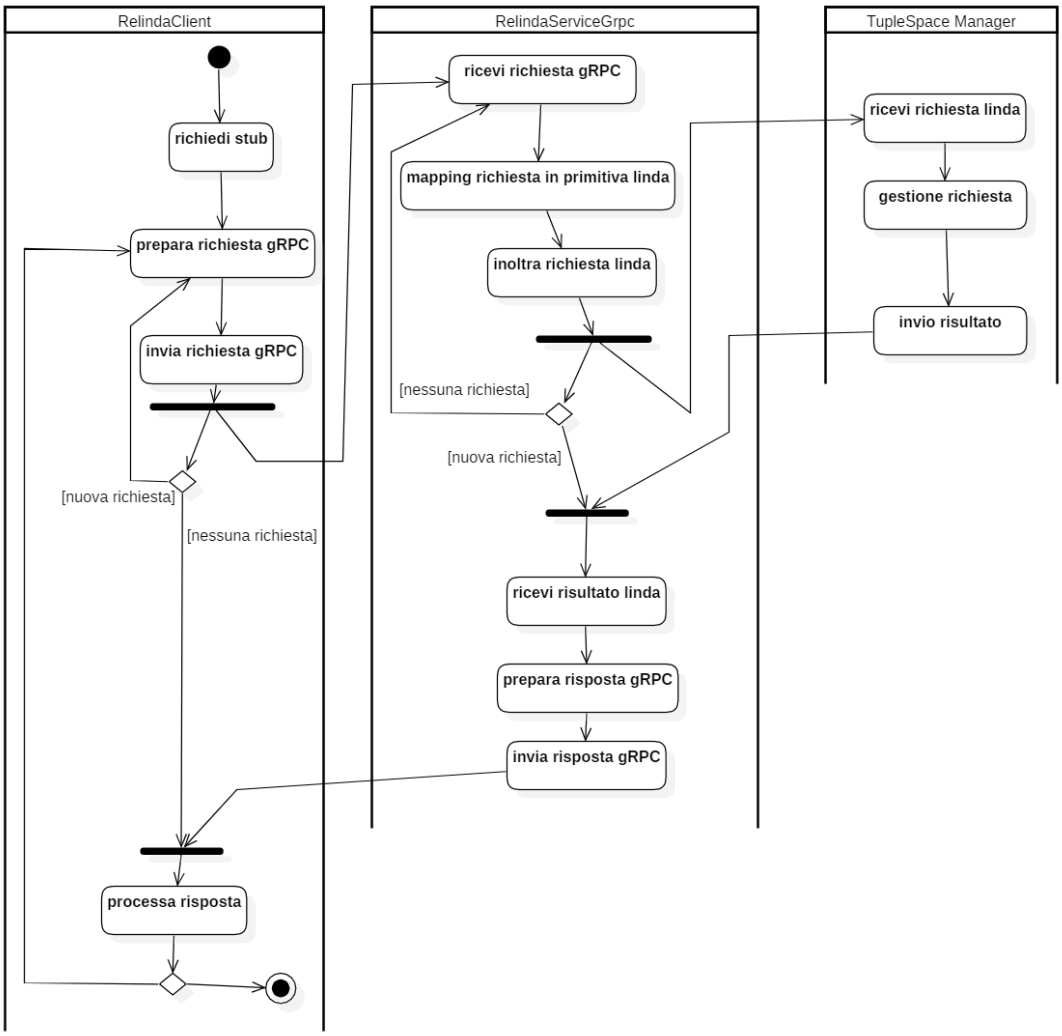

Figure 9: Diagramma di attività per una primitiva predicativa

Il server gRPC rimane inizialmente in attesa di ricevere delle richieste, all'arrivo di una di queste estrae ed analizza da essa il suo tipo e procede inoltrando al gestore di data-space LINDA il mapping della richiesta sotto forma di primitiva. Ricevuto un risultato, viene preparata una risposta gRPC e rispedita al client, ritornando infine allo stato idle iniziale ([10]).

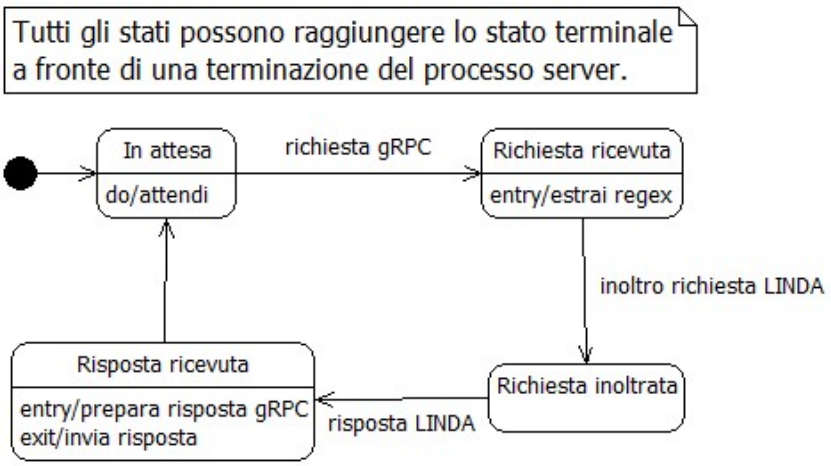

Figure 10: Diagramma di stato dell'entità server

4.3 Interazione

I diagrammi che seguono ci consentono di analizzare come avvengono le interazioni fra le varie entità che compongono il sistema lungo l'asse del tempo. Considerata la somiglianza tra le varie primitive di base di seguito verranno proposti solamente i diagrammi per la primitiva read, in particolare nel diagramma [11] è possibile osservare una richiesta sincrona (bloccante) mentre nel diagramma [12] è preso in esame il caso di una richiesta predicativa asincrona (non bloccante).

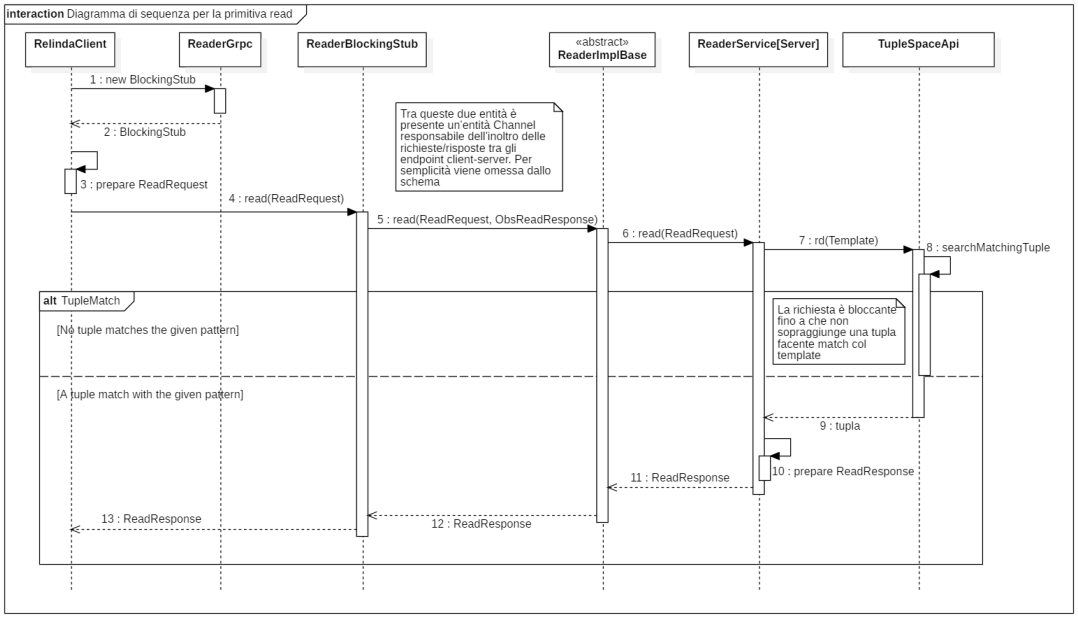

Figure 11: Diagramma di sequenza relativo ad una primitiva read

La prima operazione, effettuata a fronte di una qualsiasi richiesta, viene svolta dal client, il quale richiede lo stub al Grpc che gli consentirà di interloquire con il server.

Una volta ottenuto lo stub, una serie di metodi remoti saranno invocabili; per come sono stati strutturati i file proto precedentemente descritti, ogni servizio espone un solo metodo. Nel caso in esame, ovvero una richiesta di lettura non predicativa, lo stub interrogato sarà di tipo bloccante: *BlockingStub*.

Una volta preparata la richiesta e quindi impostati correttamente tutti i campi, mediante lo stub, la richiesta verrà inoltrata al server. In particolare, questa verrà recapitata, attraverso un *Channel* di comunicazione dedicato, all'entità **ReaderService** che fornisce un'implementazione alla classe **ReaderImplBase** ottenuta in maniera automatica.

Il server provvederà a gestire la richiesta interrogando uno specifico data-space (indicato in quest'ultima) mediante le api offerte da gtextbfAbstractTupleSpaceApiImpl.

La primitiva read, se non è presente nessuna tupla facente match col template incluso nella richiesta, rimarrà sospesa fino a quando una tupla che fa 'match' non sopraggiunge nel data space. A fronte di ciò, la tupla verrà restituita al client ripercorrendo la catena di chiamate al contrario.

Mediante questo diagramma è possibile percepire le potenzialità di questa libreria: si stanno invocando metodi in esecuzione su macchine distribuite sulla rete come se fossero istanze locali, mascherandone quindi la dislocazione.

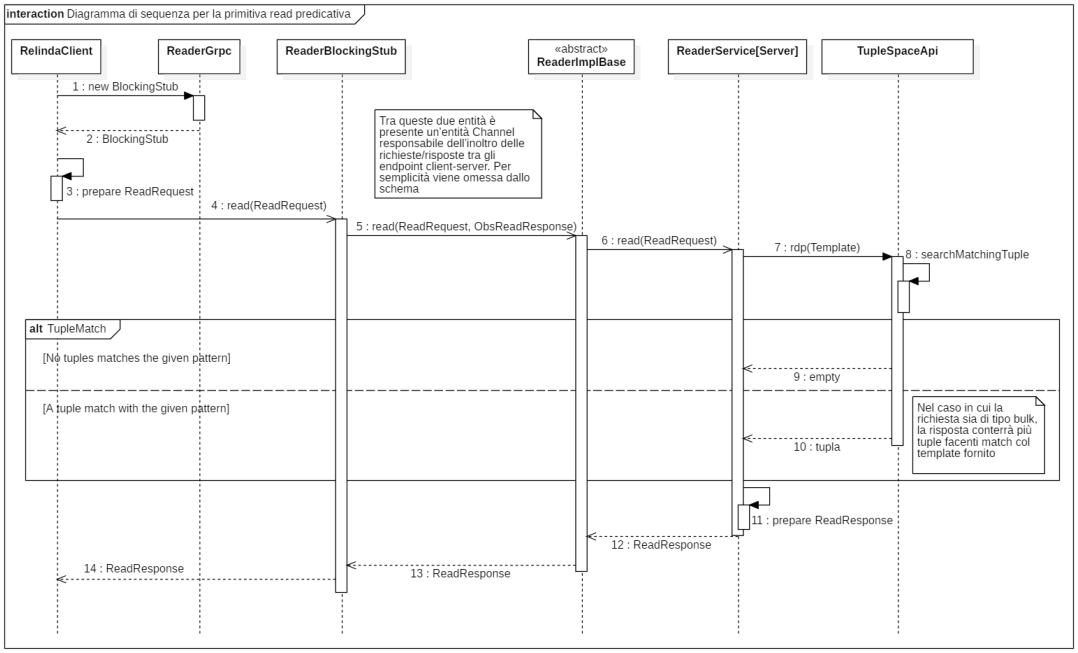

Figure 12: Diagramma di sequenza relativo ad una primitiva read predicativa

Le richieste predicative vengono gestite in maniera simile al caso precedente, tuttavia qui il client si avvale di uno stub non bloccante (*ReaderStub*) e le richieste che includono un template non soddisfatto da nessuna tupla vengono comunque portate a termine restituendo una risposta 'vuota'. La comunicazione istanziata attraverso Channel è pertanto asincrona.

4.3.1 Potenziale inconsistenza di comunicazione

Una possibile incongruenza, che potrebbe insorgere durante l'utilizzo del sistema da parte del client, consiste nell'inoltrare al server una richiesta sincrona (ad esempio una read non predicativa) attraverso uno stub non bloccante. Il server gestirà la richiesta mediante le api di LINDA, che tratteranno la richiesta come specifica la semantica (bloccante); di conseguenza, nel caso in cui non vi siano tuple facenti match col template incluso nella richiesta si avrà il canale di comunicazione (per la richiesta corrente) bloccato in attesa che sopraggiunga una tupla nel data-space, nonostante lo stub utilizzato dal client sia non bloccante. Lato server, durante l'handling della richiesta non è possibile conoscere quale tipo di comunicazione sia stata scelta pertanto non è possibile arginare questo 'problema' (virgolettato in quanto ne risulta un'inconsistenza logica ma non funzionale).

5 Dettagli implementativi

Nelle fasi di design erano stati previsti tre servizi per la gestione delle richieste linda ognuno con i metodi relativi alle primitive base, predicative, bulk e negate. Durante lo sviluppo ci si è resi conto che con una separazione in più servizi e la conseguente generazioni automatica delle classi dedicata alla gestione di tali richieste comportava una una proliferazione di codice non indifferente. Inoltre, era necessaria la definizione di più messaggi, di richiesta e risposta, molto simili tra loro e la definizione di tre diversi client per poter richiedere le tre operazioni, read, take e write, al server. Per tali motivi, si è deciso di utilizzare un solo metodo per ogni tipologia di richiesta (read, take, write) e di organizzarli sotto il medesimo servizio *RelidaService*. Di seguito il diagramma delle classi che rappresenta il risultato di questo cambiamento, si noti come tale diagramma offra le stesse operazioni dei diagrammi precedentemente esposti [5], ma con un numero di classi notevolmente ridotto [13]. A seguito di questa riorganizzazione per ogni primitiva ci si avvale di un solo messaggio di richiesta ed un solo messaggio di risposta, ognuno dei quali include i parametri specifici per l'operazione desiderata.

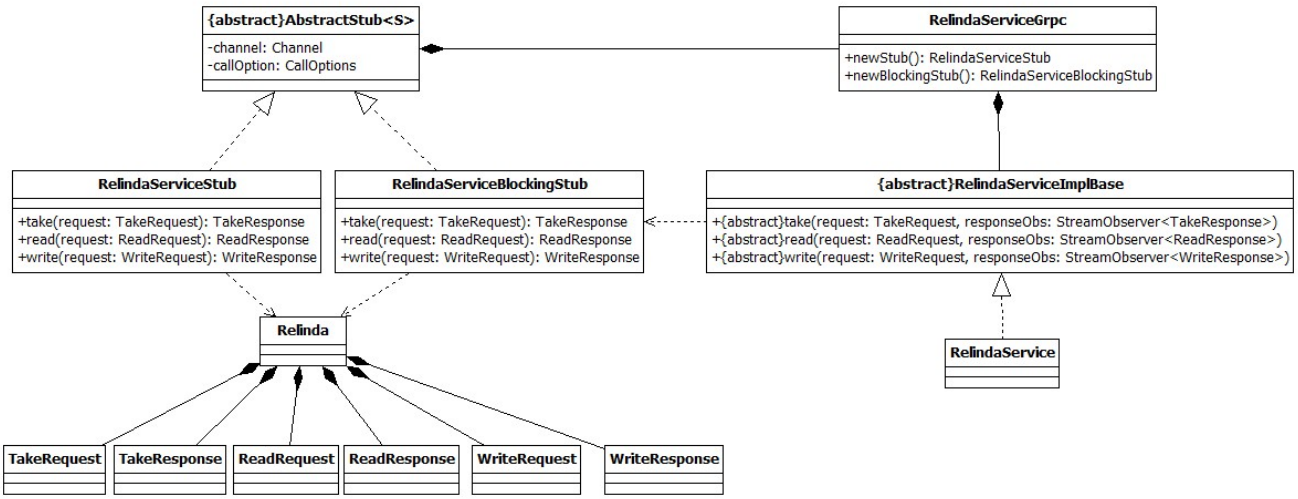

Figure 13: Diagramma delle classi delle classi utili per l'implementazione delle primitive Linda.

Altro cambiamento in corso d'opera coinvolge la struttura dati responsabile di contenere le tuple e i template all'interno dei messaggi. Inizialmente si era pensato di codificare template e tuple come semplici stringhe testuali e sfruttare l'opzione *repeated* offerta dalle specifiche *proto* per occorrenze multiple di queste ultime (utile nelle richieste bulk). Successivamente, a fronte di un cambio di specifiche, è stata sviluppata una classe di utilità per serializzare e de-serializzare, attraverso le api *linda-presentation*, tuple e template e codificarli in formato json. Il prefisso *repetead* è stato pertanto rimosso ed il campo utilizzato è rimasto una semplice stringa. La struttura definitiva dei messaggi

proto e l'organizzazione in servizi dei metodi remoti è documentata qui.

Per quanto concerne il codice sviluppato, gli aspetti implementativi degni di nota risultano nella **AbstractTupleSpaceApi**, un'interfaccia funzionale che interagisce con i tuple space e mette a disposizione i metodi d'interazione con questi ultimi all'handler **RelindaService** delle richieste LINDA. Tale interfaccia, sfrutta un metodo astratto per ottenere il tupleSpace sul quale operare (*LogicSpace* o *StringSpace*). Il tuple space viene fornito dall'utilizzatore di queste api (in questo caso l'handler sopracitato), che deve preoccuparsi di reperirlo da **TupleSpaces** avvalendosi del metodo *wrap* definito nelle interfacce funzionali **LogicTupleSpaceApi** e **StringTupleSpaceApi** che estendono **AbstractTupleSpaceApi**.

```
@FunctionalInterface
interface TupleSpaceApi<T extends Tuple, TT extends
   Template, K, V> {

        public default Collection<T>
           createNewTuples(boolean bulk, Collection<T>
           tuples) {// body}

        public default Collection<Match<T, TT, K, V>>
           observeTuples(boolean bulk, boolean predicative,
           boolean negated, TT template) {//body}

        public default Collection<Match<T, TT, K, V>>
           consumeTuples(boolean bulk, boolean predicative,
           TT template) {// body}

        public default CompletableFuture<MultiSet<T>>
           getAllTuples() {// body}

        public default CompletableFuture<Integer>
           getTupleSpaceSize() {// body}

        // reperimento dell'extended tuple space specifico
           sul quale operare (LogicTupleSpace o
           StringTupleSpace)
        ExtendedTupleSpace<T, TT, K, V> getTupleSpace();
}
```

```
@FunctionalInterface
public interface LogicTupleSpaceApi extends
   AbstractTupleSpaceApi<LogicTuple, LogicTemplate, String,
   Term> {
        // this is the only non-default method
        public LogicSpace getTupleSpace();

        public static LogicTupleSpaceApi wrap(LogicSpace
           ts) {
                return () -> ts;
        }
}
```

Doveroso risulta essere l'analisi di **RemoteTupleSpace**: una classe astratta che funge da wrapper del client gRPC **RelindaClient**. Tale wrapper, è a tutti gli effetti un tuple space di *linda-core* e ne espone di conseguenza tutti i metodi di interazione. La peculiarità di questa classe, consiste nel fatto che *Relinda* può essere utilizzato come un qualunque sistema locale nascondendone la sua natura distribuita. All'atto pratico, quello che svolge consiste nell'inoltrare le richieste a RelindaClient, che le incapsulerà in messaggi gRPC, e gestirne le risposte. Essendo una classe astratta, **RemoteTupleSpace** necessita di essere concretizzata attraverso due classi **RemoteLogicTupleSpace** e **RemoteStringTupleSpace** che operano rispettivamente su tuple space logici e testuali, consentendo alla classe astratta di conoscere il tipo specifico dei template e delle tuple con le quali operare.

6 Autovalutazione e convalida

Il criterio adottato per valutare la qualità del progetto sono i test automatizzati; sono stati sviluppati test infrastrutturali per costruire una base di conoscenza sul funzionamento di gRPC e riusati test per valutare la corretta semantica delle primitive LINDA implementate.

I test infrastrutturali sono contenuti nella classe **GrpcInfrastructureTest** e sono stati implementati nelle prime fasi del progetto (dopo la definizione dei messaggi proto). Questi test, avvalendosi del framework Mockito per l'implementazione di un server fittizio, verificano che i messaggi del client giungano al server integri e che quest'ultimo sia in grado di rispondere al client. Sono stati utilizzati per guidare lo sviluppo del sottosistema lato server e verificare la correttezza della classe **Utility** per la serializzazione/deserializzazione dei template e delle tuple. I test per la valutazione della semantica delle primitive mirano a verificare che queste ultime abbiano un comportamento analogo a quello atteso.

7 Istruzioni per il deployment

Per eseguire il software prodotto è necessario soddisfare i seguenti requisiti:

- aver installato sulla propria macchina un sistema operativo compilato a 64bit, in caso contrario la build automatica del progetto non andrebbe a buon fine, per maggiori info leggere questa issue.
- aver installato il Java Development Kit in versione 10 o successiva.

Essendo un progetto gradle, la compilazione del software è automatizzata e risulta sufficiente eseguire il comando `./gradlew build`.

Va ricordato che il progetto consiste in un'interfaccia di interazione, di conseguenza non può essere eseguito fino a quando non si produce il software che utilizza tale sistema.

8 Esempi di utilizzo

Per poter sviluppare software che sfruttino Relinda è necessario definire prematuramente un'istanza del server **RelindaServer** indicando su quale *porta* renderlo disponibile ed avviarla mediante il metodo *.start*(). Lato client invece, è sufficiente istanziare una classe concreta di **RemoteTupleSpace** (**RemoteLogicTupleSpace** o **RemoteStringTupleSpace**) passando come argomenti al suo costruttore *indirizzo IP*, *porta* ai quali contattare il server e il nome del tuple space che si vuole utilizzare per le future richieste. Attraverso quest'ultima classe sarà poi possibile richiamare le varie primitive linda come se fossero eseguite in tuple space locali.

9 Conclusioni

Per riassumere, la realizzazione del progetto non si è limitata agli obiettivi base, ma ha ambito a ricoprire anche quelli opzionali; in particolare, Relinda supporta tutte le primitive Linda: base, predicative, bulk e negate. Inoltre, come evidenziato in precedenza, le tuple e i template all'interno delle richieste/risposte gRPC sono state serializzate/deserializzate e rappresentate in formato json. Il lavoro è passato attraverso le seguenti fasi:

- sono stati definite le strutture dei messaggi proto utilizzati per la comunicazione;
- sono state sviluppate le classi per la gestione delle richieste LINDA lato server e definite le api per l'interrogazione dei tuple space;
- in parallelo al punto precedente sono state sviluppate le api per interagire con il tuple space remoto come se fosse un'istanza locale. Queste, sfruttano una classe che rappresenta un client Relinda e ha il compito di inoltrare le richieste e ricevere le relative risposte al/dal servizio server remoto;

- sono stati definiti un insieme di test strutturali per verificare il funzionamento corretto dell'architettura gRPC ed attraverso i test presenti in *linda-test*, verificato il funzionamento corretto del tuple space remoto.

9.1 Lavori futuri

Tra i possibili sviluppi futuri figura la possibilità di re-ingegnerizzare la struttura del sistema per ottenerne uno più efficiente e di utilizzare messaggi di trasmissione generici, non strettamente legati al tipo di richiesta: interamente codificato in json e non limitati alle sole tuple e template.

Potrebbe essere interessante verificare il funzionamento e l'efficienza del sistema a fronte di tuple space di dimensioni notevoli. Data la dimensione limitata dei messaggi a 4 MB, a fronte della trasmissione di numerose tuple, il sistema potrebbe non funzionare correttamente; una possibile soluzione potrebbe essere quella di avvalersi di stream di messaggi e non di messaggi unari.

Inoltre, potrebbe essere interessante utilizzare le funzionalità offerte da gRPC per l'autenticazione e la costruzione di canali tra client e server criptati.

9.2 Conoscenze acquisite

Gli studenti hanno potuto comprendere più a fondo gli argomenti affrontati duranti il corso, in particolare: Linda, gli spazi di tuple logici e testuali ed il building automatizzato di progetti mediante gradle. Inoltre, trattandosi di un progetto che coinvolge più persone, è stato utilizzato un software di controllo versione distribuito, git, che gli studenti già conoscevano, ma di cui hanno potuto sperimentare nuove feature come: creazione di nuovi branch, sincronizzazione di un fork con il repo originale...

Come nuove conoscenze, figurano la comprensione del funzionamento delle seguenti librerie: *gRPC* e *Mockito*. Il primo consente l'invocazione di metodi in un'istanza remota (potenzialmente sviluppata in un linguaggio diverso dal chiamante), ed è pienamente coerente con gli argomenti trattati a lezione inerenti ai sistemi distribuiti. Il secondo è un framework usato per la generazione di mock (mockup: classi fittizie) di classi non ancora sviluppate in modo da poter predisporre test anticipati di determinate funzionalità, particolarmente utile quando si adotta un approccio test-driven.

References

[1] Giovanni Ciatto. Deep into linda. https://iol.unibo.it/mod/resource/view.php?id=135631, 2018.

[2] Giovanni Ciatto. Logic tuple spaces. https://iol.unibo.it/mod/resource/view.php?id=138863, 2018.

[3] Giovanni Ciatto. Process algebrae and other ways for hurting people. https://iol.unibo.it/pluginfile.php/273070/mod_resource/content/7/SD1819_Lab_5.pdf, 2018.

[4] David Muto. gRPC protoc doc generator. https://github.com/pseudomuto/protoc-gen-doc.

[5] Lorenzo Rizzato. Coordination as a web service: una moderna implementazione del modello linda. https://amslaurea.unibo.it/16791/1/rizzato_lorenzo_tesi.pdf, 2018.

[6] Sconosciuto. gRPC guideline. https://grpc.io/docs/guides/.