

Linda On AMQP

Martina Cavallucci

`martina.cavallucci@studio.unibo.it`

Simone Golinucci

`simone.golinucci@studio.unibo.it`

13 aprile 2020

L'obiettivo del progetto è quello di definire e realizzare un'interfaccia che permetta l'interazione con una o più tuple space Linda. Un vincolo posto al progetto è l'utilizzo del protocollo *AMQP* per lo scambio di messaggi. *AMQP* è un protocollo programmabile per lo scambio di messaggi basato su broker. Con protocollo programmabile si fa riferimento alla feature che permette alle applicazioni stesse di definire le entità e gli schemi di routing e non dall'amministratore del broker.

Il broker è definito da più componenti: l'exchange, le code e il bind tra i due. L'exchange ha il compito di distribuire i messaggi alle varie code a cui sono registrati i client.

L'interfaccia che si vuole realizzare fungerà da tramite tra i client connessi al broker *AMQP* ed uno o più tuple space Linda.

La comunicazione tra i due sistemi sarà possibile grazie ad un client speciale *LindaServer* connesso al broker *AMQP*. Il *LindaServer* rimarrà in ascolto dei messaggi provenienti dai client *LindaClient*, con il compito di tradurli e inoltrarli come richieste al sistema Linda. Linda produrrà poi la risposta che sarà tradotta in un messaggio *AMQP* ed inoltrata al *LindaClient* richiedente.

Date le diverse tipologie di exchange del protocollo si è scelto di utilizzare come canale principale di richiesta l'exchange Topic, dove si sono specificati i formati accettabili. Invece per la risposta da parte di *LindaServer* si è scelto l'exchange Direct che appunto non segue specifici topic per la risposta al client in attesa.

1 Obiettivi e Requisiti

1.1 Requisiti

Si vuole realizzare un sistema che permetta la coordinazione di uno o più dispositivi remoti, come ad esempio nell'ambito IoT, basato sul protocollo *AMQP*.

Una specifica che questo sistema deve seguire, impone una coordinazione mediante un tuple space Linda condiviso e accessibile da tutte le entità che necessitano della suddetta coordinazione. Altresì, si vuole realizzare un' interfaccia lato client che sia in grado di nascondere il più possibile i dettagli tecnici dell'implementazione, mostrando all'utilizzatore soltanto quelle primitive che fanno parte degli spazi di tuple Linda.

Ci si aspetta che venga realizzato un prototipo di servizio che implementi le primitive base del tuple space Linda e che sia accessibile da più dispositivi che siano in grado di comunicare con il servizio offerto. Si vuole anche realizzare un' applicazione esemplificativa per mostrare il funzionamento e l'efficacia del sistema prototipato.

1.2 Obiettivi

Definizione del protocollo L'obiettivo primario è quello di realizzare una specifica del protocollo definendo le modalità di iterazione tra il *LindaClient* e il *LindaServer* . Nella sua definizione si esporranno le caratteristiche quali: formato dei messaggi, regole di interpretazione e traduzione, sequenze di interazione ammissibili. Tale protocollo verrà definitivo al di sopra di *AMQP* nella pila protocollare come si vede in Figura 1

Realizzazione componente server Una volta definito il protocollo di comunicazione tra l'applicazione client e il *LindaServer* si realizzerà tale componente. Esso implementerà sia la specifica del protocollo di comunicazione, sia la gestione del tuple space vero e proprio. Il *LindaServer* sarà sviluppato con il linguaggio Java ma grazie al protocollo *AMQP* è possibile interfacciarsi al *LindaServer* mediante diversi linguaggi.

Realizzazione componente client Tra gli obiettivi vi è quello della realizzazione di un' interfaccia client che possa essere utilizzata per realizzare un' applicazione che faccia uso di uno o più tuple space presenti nel *LindaServer* .

Configurazione del broker Nonostante il progetto faccia uso di un broker *AMQP* , non fa parte del progetto la sua realizzazione. Pertanto si utilizzerà un' implementazione già esistente chiamata *RabbitMQ* . Tale implementazione è particolarmente interessante poiché fornisce delle API di interazione con il broker realizzate con diversi linguaggi di programmazione.

Realizzazione Applicazione Client Mediante la componente client realizzata precedentemente si vuole realizzare un esempio di applicazione che faccia uso delle primitive del tuple space. Le primitive utilizzate in questa applicazione dipenderanno dal contesto applicativo nella quale si vuole realizzare l'applicazione.

Obiettivi opzionali Implementazione di interfaccia per le primitive Bulk e le primitive predicative.

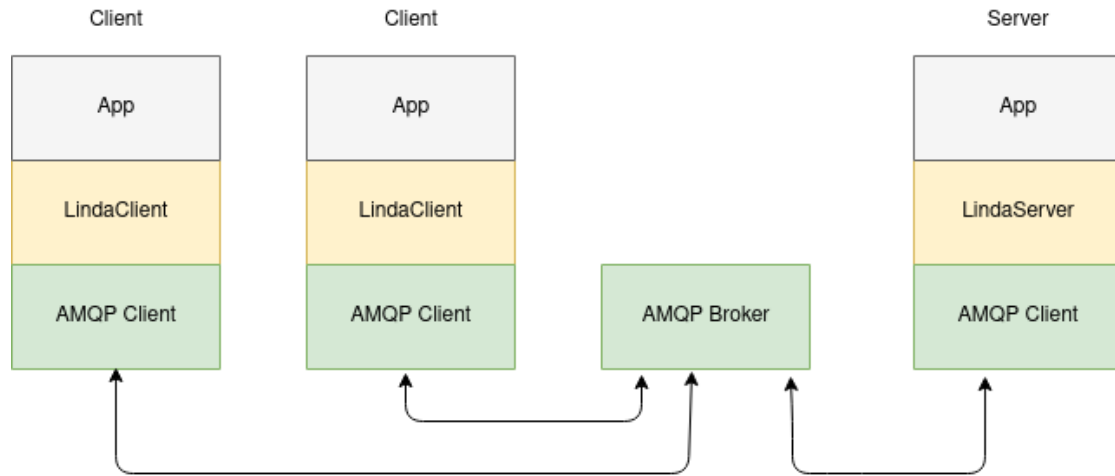


Figura 1: Architettura logica del sistema proposto

1.3 Risultati attesi

Il risultato principale è la creazione di un sistema che permetta l'utilizzo di un tuple space Linda in modo distribuito, mantenendo la semantica di Linda. Tale necessità deriva dalla volontà di voler collegare più dispositivi in ambito IoT allo stesso tuple space per permettere una coordinazione tra di essi. Per supportare questa distribuzione su più dispositivi si sfrutterà il protocollo *AMQP*.

Questo tuple space in un primo momento supporterà le operazioni base di Linda. Se le tempistiche lo permettono si lavorerà per implementare le primitive bulk.

Si vuole inoltre realizzare un'interfaccia client che permetta l'interazione con il *LindaServer*.

1.4 Scenari di utilizzo

L'utilizzatore ha a disposizione 2 modi per interagire con il sistema da noi progettato. Il primo modo, quello più semplice, è mediante l'utilizzo della nostra interfaccia client che permette di comunicare con il server mediante le primitive classiche definite nel tuple space Linda. Il secondo metodo è più complesso poiché implica la comunicazione diretta con il broker su cui si appoggia il nostro servizio. Bisogna assicurarsi che tale comunicazione sia effettuata nel rispetto del protocollo di interazione progettato e che sarà implementato sul servizio Linda.

Si consiglia l'utilizzo della libreria già implementata poiché essa implementa già il protocollo di interazione tra *LindaClient* e il *LindaServer*.

1.5 Politiche di autovalutazione

- La qualità del software prodotto dovrà essere valutata tenendo in considerazione il grado di incapsulamento raggiunto dalle primitive Linda all'interno dei moduli, all'attinenza del comportamento atteso e di come tali moduli vengono suddivisi. La correttezza e l'affidabilità del software prodotto sarà garantita dalla corretta esecuzione dei test: verranno definite diverse unità di testing. La correttezza del codice sarà garantita dal soddisfacimento dei singoli test mentre per l'affidabilità sarà garantita eseguendo gli stessi test varie volte.
- Si cercherà di individuare l'architettura del sistema migliore, e di scrivere codice uniforme e corretto in base alla nostra esperienza.

2 Analisi dei requisiti

2.1 Requisiti funzionali

Il sistema che si andrà a definire, dovrà offrire un metodo di interazione con un tuple space remoto comunicando esclusivamente attraverso broker *AMQP*. Per far questo vi è la necessita di mantenere il servizio che gestisce i tuple space in remoto sia costantemente attivo per accettare le richieste che arrivano.

Oltre al servizio, anche il broker *AMQP* dovrà essere sempre attivo poiché è l'unico canale di comunicazione che ha il servizio per parlare con i vari client connessi. Per garantire un'efficace comunicazione tra le parti, si dovrà realizzare anche una specifica di messaggi che dovrà essere implementata sia nel *LindaClient* sia nel *LindaServer*.

Per la realizzazione dell'applicazione, si realizzerà anche una libreria a se stante che permetta l'interazione con il tuple space e che sia in grado di nascondere tutti i dettagli implementativi riguardo la connessione.

2.2 Requisiti non funzionali

Tra i requisiti non funzionali spicca la necessità di realizzare una buona documentazione del protocollo di comunicazione, per dare la possibilità a un programmatore terzo di poter implementare la propria libreria client.

Il sistema dovrà offrire risposte in tempi ragionevoli nonostante le primitive con semantica sospensiva. Inoltre va tenuto in considerazione il fatto che la comunicazione viene per mezzo della rete e che in base al contesto applicativo in cui ci troviamo questa rete può essere più o meno affidabile. Bisogna altresì tenere a mente la volontà che la libreria client che si vuole realizzare potrà essere messa in campo su dispositivi IoT. Questo aspetto è importante poiché non sempre i dispositivi IoT hanno una grande quantità di risorse, di conseguenza la libreria dovrà essere efficiente sotto questo punto di vista.

Non è da sottovalutare la necessità di estendere la libreria con tipologie di tuple space differenti. Lo stesso vale per il protocollo di comunicazione che verrà realizzato.

2.3 Requisiti tecnologici

Come definito da requisito, l'infrastruttura di comunicazione dovrà essere basata sul protocollo *AMQP*.

Poiché non fa parte del progetto l'implementazione del protocollo *AMQP*, si è optato per utilizzare una soluzione già esistente sul mercato. La scelta è ricaduta sull'implementazione *RabbitMQ* [4] che mette a disposizione un server che ha funzione di broker e delle librerie client che permettono l'interazione con il server da parte di uno o più client.

Il broker dovrà essere installato e avviato durante la fase di installazione del sistema. Nonostante la diffusione di tale protocollo e di tale implementazione di *AMQP*, è possibile che esista già un *RabbitMQ* server in esecuzione. In questo caso si può riutilizzare quella istanza del broker già in esecuzione.

Visto la natura dei dispositivi su cui utilizzare l'interfaccia client, si dà per scontato che tali dispositivi siano in grado di comunicare mediante la rete internet.

Il *LindaServer* non è pensato per essere eseguito su dispositivi IoT, poiché può necessitare di grandi quantità di memoria e di risorse calcolo in base al numero di dispositivi che si connettono e dal tipo di richieste che vengono effettuate.

2.4 Modelli, paradigmi e tecnologie scelte

Considerando le necessità del sistema di essere eseguito su dispositivi con configurazione variabile e non ben definita a priori, si è deciso di avere un occhio a riguardo della portabilità del codice, soprattutto per quanto riguarda il *LindaServer*. Di conseguenza è stato deciso di creare l'implementazione mediante il linguaggio Java, che offre una elevata portabilità del software. Nonostante ciò Java implica anche dei problemi relativi all'efficienza causati proprio dal suo essere estremamente portabile. Nonostante questo, poiché il *LindaServer* non è pensato per essere in esecuzione su sistemi IoT, abbiamo deciso di accettare questo compromesso.

Data la scelta di utilizzare Java come linguaggio di programmazione, è stato adottato il paradigma di programmazione ad oggetti ma con qualche miglioramento, principalmente relativo alla programmazione asincrona, utilizzata per reagire ai messaggi che ci arrivavano dal broker.

3 AMQP Overview

In questa sezione è dedicata a una digressione generale sul protocollo *AMQP*, necessaria per introdurre delle parole che ci serviranno successivamente per definire meglio il protocollo di interazione usato tra il *LindaServer* e i client.

3.1 Cosa è AMQP?

AMQP [2] (*Advanced Message Queuing Protocol*) è un protocollo a scambio di messaggi che si appoggia su un broker. Il Broker ha la funzione di ricevere i messaggi dai *Publisher* e di reindirizzarli ai *Consumer*, che sono quelle applicazioni che devono processare i suddetti messaggi.

3.2 Modello del protocollo

Senza entrare nello specifico, il protocollo segue il seguente flusso[1]: un *Publisher* pubblica il suo messaggio a un *Exchange*, che a sua volta distribuisce i messaggi nelle varie *code* in base alle regole di re-indirizzamento chiamate *bindings*. Successivamente il broker consegna i messaggi ai *Consumer* che hanno sottoscritto una di queste *code*; tale comportamento si evince dalla figura 2.

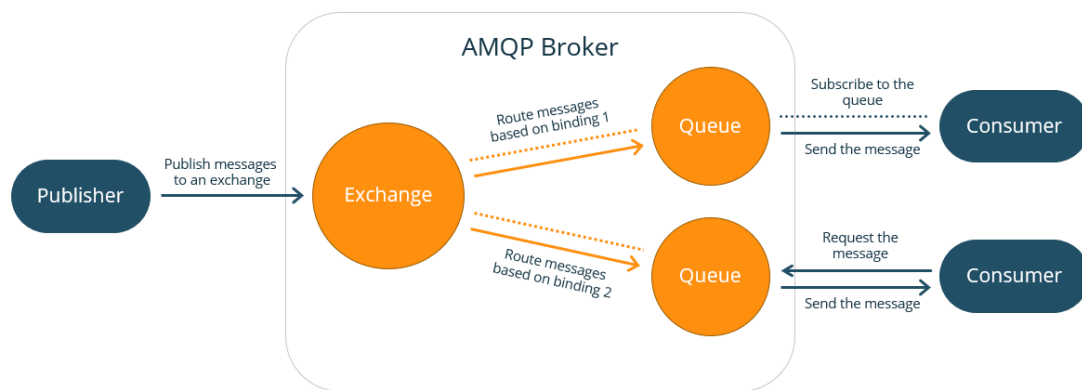


Figura 2: Logica del protocollo *AMQP*

Il protocollo *AMQP* è considerato molto più affidabile rispetto ad altri protocolli. Questa affidabilità è data dalla nozione di *message acknowledgements* ovvero, quando il messaggio viene consegnato, viene notificato al broker che la consegna è avvenuta.

Nel protocollo: le *code*, gli *exchange* e i *bindings* sono definite entità *AMQP* e non semplici dettagli implementativi.

3.2.1 Exchange

L'*Exchange* è il punto di ingresso del messaggio nel broker e ha il compito principale di smistare i messaggi nelle relative code, che possono essere uno o più. Visto il ruolo che svolge può essere associato a un ufficio postale.

L'algoritmo utilizzato per questo smistamento dipende dal tipo di Exchange adottato e dai bindings che vengono effettuati.

Le tipologie di Exchange sono diverse ed ognuna è indicata per diversi contesti applicativi. Le tipologie principali sono:

- *Direct*: ideale per l'instradamento diretto di messaggi;
- *Fanout*: ideale per inviare messaggi a tutte le code associate all'Exchange;
- *Topic*: ideale per l'instradamento multicast di messaggi su più code;
- *Headers*: permette un instradamento basato su più attributi definiti nell'intestazione del messaggio.

In questo progetto sono stati utilizzati nello specifico 2 tipologie di Exchange:

Direct Ideale per l'indirizzamento diretto dei messaggi. Il suo funzionamento è molto semplice: una coda esegue quello che è definito bind di una routing-key all'exchange, ovvero dice all'exchange quale routing-key deve avere un messaggio per essere consegnata sulla coda stessa. Successivamente quando un messaggio viene pubblicato all'exchange, quest'ultimo confronta la routing-key del messaggio con quella registrata dalla coda. Se corrisponde, la coda riceve una copia del messaggio. Se ne mostra un esempio in figura 3.

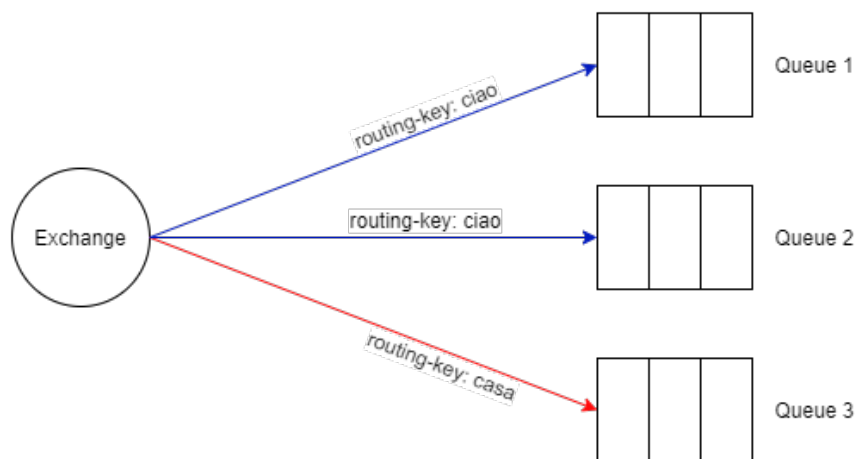


Figura 3: Direct exchange

Topic Questo tipo di exchange, molto utilizzato per implementare varianti del pattern publish/subscribe, permette di eseguire un binding di una coda con un pattern. Il messaggio viene inviato in una specifica coda se la routing-key associata al messaggio rispetta la forma del pattern associato alla bind.

Il pattern può essere visto come un URI, dove i vari livelli sono divisi da punti (.) invece che dal carattere slash (/).

Ad esempio, un pattern può essere:

a.b.c

Questo pattern è formato da 3 livelli e accetta soltanto *a.b.c* come topic.

Viene anche definito un modo per avere dei pattern più flessibili, utilizzando dei caratteri speciali come asterisco (*) e il cancelletto (#). Per esempio il pattern:

a..c*

permette di eseguire il match con di tutti i topic formati da 3 livelli dove il primo livello è il carattere *a* e il terzo livello è il carattere *c*, mentre il secondo livello può assumere qualsiasi valore alfanumerico.

Il pattern:

a.#.c

sembra simile a quello precedente, ma non è così. Sebbene * permette di definire qualsiasi valore su un singolo livello, # permette di definire qualsiasi valore su un qualsiasi numero di livelli. Questo significa che il pattern *a.#.c* sarà valido per tutte le routing-key che hanno come primo livello il carattere *a* e come ultimo livello il carattere *c* come nell'esempio precedente. La differenza tra i due è che in questo secondo caso # non rappresenta un singolo livello come *, ma rappresenta una moltitudine di livelli.

Ad esempio, delle routing-key valide per il pattern di quest'ultimo esempio, potrebbero essere: *a.b.c*, *a.b.d.c*, *a.b.d.e.c*, etc...

In breve è valida la relazione:

$$a.\#.c = a(.*)+.c$$

4 Progettazione

4.1 Definizione del protocollo

Per permettere l'interazione con lo spazio di tuple è necessario definire sia il modo con la quale il client e il server si inviano i messaggi, sia il formato dei messaggi, specificando la loro struttura e il contenuto in base alle informazioni da trasmettere.

Scambio di messaggi Lo scambio di messaggi presenta una struttura diversa per le richieste e le risposte. Per una richiesta, viene utilizzata una struttura di messaggio stile *Topic*, come nel protocollo MQTT. Per la risposta viene utilizzato uno stile definito *direct* poiché i messaggi vengono inviati dal server al client in modo diretto. Questa differenza di metodo tra lo spedire e ricevere messaggi è possibile grazie al concetto di *Channel* presente nel protocollo *AMQP*, che permette di utilizzare una singola connessione fisica per avere più connessioni logiche.

La struttura che si utilizzerà per lo scambio di messaggi è riassunta nella Figura 4.

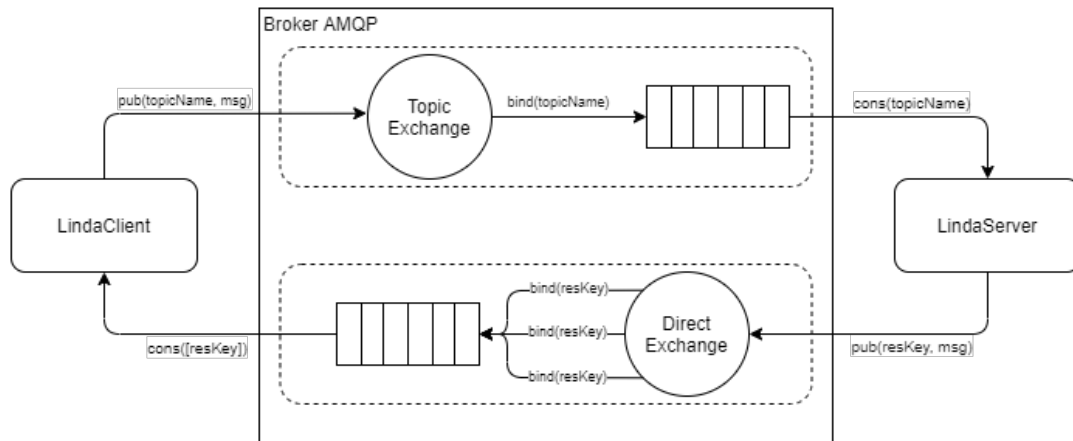


Figura 4: Struttura del broker del protocollo realizzato

4.2 Messaggio di richiesta

I messaggi di richiesta, vengono inviati utilizzando il pattern publish-subscribe come nel protocollo *MQTT*. Il server si mette in ascolto su un particolare topic che gli permette di accedere a tutte le richieste che seguono il pattern definito. Tale pattern ha la seguente struttura:

MQLINDA.serverID.*.*.*

il topic così definito significa che il server è in ascolto su un topic formato da 5 livelli. Il primo livello è statico per il protocollo di comunicazione e per ogni istanza del server. Il secondo livello è statico per ogni istanza del server poiché esprime un nome identificativo

del server stesso. Abbiamo assegnato un nome per ogni livello al fine di poter specificare, per ognuno di questi, il proprio significato:

MQLINDA.serverID.clientID.tupleSpace.dataType

Le parole del topic rappresentano:

- **MQLINDA**: rappresenta la root dell'applicazione, identifica la singola applicazione nel caso si utilizzi lo stesso Exchange per più applicazioni;
- **serverID**: nome identificativo del server a cui si vuole effettuare la richiesta;
- **clientID**: nome identificativo del client che effettua la richiesta;
- **tupleSpace**: nome del tuple space a cui si vuole fare accesso;
- **dataType**: indica il tipo di tuple space. Per ora è supportato lo spazio di tuple basato su stringhe, ma facilmente estendibile.

4.3 Messaggio di risposta

Il messaggio di risposta viene inviato dal *LindaServer* verso il client, sfruttando un differente *Channel* basato su un *exchange* di tipo *direct*. Il client, ogni volta che invia una richiesta al server, esegue anche un bind di una nuova routing-key nel exchange dedicato per le risposte. Il server manda un messaggio utilizzando la suddetta routing-key e poi sarà compito del broker a reindirizzare il messaggio verso il giusto client.

Quindi per il messaggio di risposta è necessario impostare una buona funzione che sia in grado di calcolare una routing-key univoca, e che tale algoritmo sia presente sia sul client che sul server.

Attualmente la funzione utilizzata per calcolare la routing-key è:

$$h(serverID + clientID + operationID)$$

dove:

- **+**: indica una concatenazione tra stringhe;
- *serverID* e *clientID*: sono i nomi identificatori rispettivamente del server e del client specificati nel topic della richiesta;
- *OperationID*: è un identificativo della richiesta univoco per ogni client. Sarà definito meglio successivamente nel payload del messaggio.

4.4 Payload

Il formato del payload dei messaggi è stato definito da noi e segue il seguente pattern Regex:

```

OperationID:([0-9]+)\|
OperationType:(RD|IN|OUT|GET|GET_SIZE|RD_ALL|IN_ALL|OUT_ALL)\|
(MessageType:(ACK|NACK|RESPONSE)\|)?
Content:(.*)

```

Quando un messaggio viene ricevuto, deve essere tradotto da stringa ad un'istanza del nostro messaggio. A livello applicativo abbiamo 2 tipi di messaggi, uno di richiesta e uno di risposta. Quello di richiesta viene generato dal client e spedito al server, mentre quello di risposta viene generato dal server e spedito al client. Il payload di entrambi i messaggi hanno i seguenti campi:

- **OperationID**: identificativo della richiesta. Necessario al server per definire la response-key, una routing-key utilizzata per spedire la risposta al client, permettendo inoltre di associare una richiesta a una risposta del server;
- **OperationType**: indica il tipo di operazione da eseguire sullo spazio di tuple. I suoi valori verranno mostrati successivamente quando tratteremo le varie primitive implementate;
- **Content**: rappresenta il contenuto della richiesta o alternativamente della risposta. In alcuni casi può essere vuoto altrimenti può essere una tupla, una collezione di tuple o un template.

A differenza del messaggio di richiesta, il messaggio di risposta ha un campo in più, definito con il nome di *MessageType* che rappresenta il tipo di messaggio di risposta. Questo può assumere i seguenti valori:

- **RESPONSE**: indica un messaggio di risposta che ha come contenuto i valori richiesti con il messaggio di richiesta;
- **ACK**: questo tipo indica che il server ha ricevuto correttamente la richiesta del client e che potrà procedere all'elaborazione di tale richiesta;
- **NACK**: questo tipo di messaggio viene inviato se il server riscontra qualche problema nella esecuzione della richiesta. Parte della semantica di questo tipo di risposta implica che il server abbia abortito l'operazione.

4.5 Primitive Linda

Le primitive Linda sono definite come una sequenza prestabilita di operazioni publish e subscribe tra componente server e client. Di seguito si descrive la struttura dei messaggi da spedire per eseguire le primitive di Linda, sia quelle classiche che quelle bulk.

Primitive classiche Sono le primitive originali di Linda: Rd, In, Out. Come da specifica Linda le primitive Rd e In hanno una semantica sospensiva, mentre la primitiva Out no. Affinché il server elabori correttamente la richiesta di queste operazioni, essa dovrà essere formata nel seguente modo:

Rd, In

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

OperationID	long
OperationType	"RD" "IN"
Content	Template

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

OperationID	long
OperationType	"RD" "IN"
MessageType	ACK NACK
Content	""

OperationID	long
OperationType	"RD" "IN"
MessageType	RESPONSE
Content	Tuple

Out

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

OperationID	long
OperationType	"OUT"
Content	Tuple

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

OperationID	long
OperationType	"OUT"
MessageType	ACK NACK
Content	""

OperationID	long
OperationType	"OUT"
MessageType	RESPONSE
Content	Tuple

Primitive bulk Le primitive bulk sono una variante delle primitive classiche che permettono di inserire, togliere, leggere, più di una tupla nello spazio di tuple con una sola chiamata. A differenza delle primitive classiche, queste primitive non hanno semantica sospensiva.

Rd_All, In_All

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

```

OperationID          long
OperationType  "RD_ALL"|"IN_ALL"
Content            Template

```

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

```

OperationID          long
OperationType  "RD_ALL"|"IN_ALL"
MessageType          ACK|NACK
Content              ""

```

```

OperationID          long
OperationType  "RD_ALL"|"IN_ALL"
MessageType          RESPONSE
Content            List<Tuple>

```

Out_All

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

```

OperationID          long
OperationType  "OUT_ALL"
Content            Tuple

```

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

```

OperationID          long
OperationType  "OUT_ALL"
MessageType          ACK|NACK
Content              ""

```

```

OperationID          long
OperationType  "OUT_ALL"
MessageType          RESPONSE
Content            List<Tuple>

```

Primitive aggiuntive Vengono inoltre aggiunte 2 primitive molto utili soprattutto per la fase di test, che permettono di ritornare tutte le tuple presenti nel tuple space, e un'altra che ritorni il numero di tuple presenti nel tuple space.

GET

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

OperationID	long
OperationType	"GET"
Content	Template

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

OperationID	long
OperationType	"GET"
MessageType	ACK NACK
Content	""

OperationID	long
OperationType	"GET"
MessageType	RESPONSE
Content	List<Tuple>

GET_SIZE

Client Publish: MQLINDA.ServerID.ClientID.TupleSpaceName.DataType

OperationID	long
OperationType	"GET_SIZE"
Content	Template

Server Publish: response-key, calcolato mediante la formula in sezione 4.3.

OperationID	long
OperationType	"GET_SIZE"
MessageType	ACK NACK
Content	""

OperationID	long
OperationType	"GET_SIZE"
MessageType	RESPONSE
Content	integer

4.6 Struttura

Il progetto si suddivide in 3 sotto progetti; si è effettuata tale scelta per rendere i sotto progetti indipendenti. In particolare il progetto *LindaMessage* viene utilizzato da *LindaServer* e *LindaClient* come libreria. Tale operazione ci ha permesso di non duplicare il codice ed avere un unico punto di riferimento per la rappresentazione dei messaggi con cui le due entità comunicano.

4.6.1 Architettura LindaMessage

Si sono definiti due tipi di messaggi per il protocollo: **RequestMessage** e **ResponseMessage**. Il loro utilizzo per la comunicazione si basa su una visione più astratta rispetto ai messaggi diretti al TupleSpace, essi rappresentano infatti le richieste e le risposte che coinvolgono la parte applicativa. Tali entità contengono le informazioni per eseguire le operazioni sullo spazio di tuple remoto; ognuno di essi avrà una specifica diversa rappresentazione per la costruzione del **MQMessage** che rappresenta il messaggio che *LindaServer* e *LindaClient* si scambieranno. In particolare il **ResponseMessage** conterrà anche l'informazione riguardo al tipo di messaggio di risposta e una *responseKey* che rappresenta la codifica di un'insieme di informazioni che servirà per riconoscere tale richiesta tra la coda delle richieste; tali dettagli si evincono dalla Figura 5.

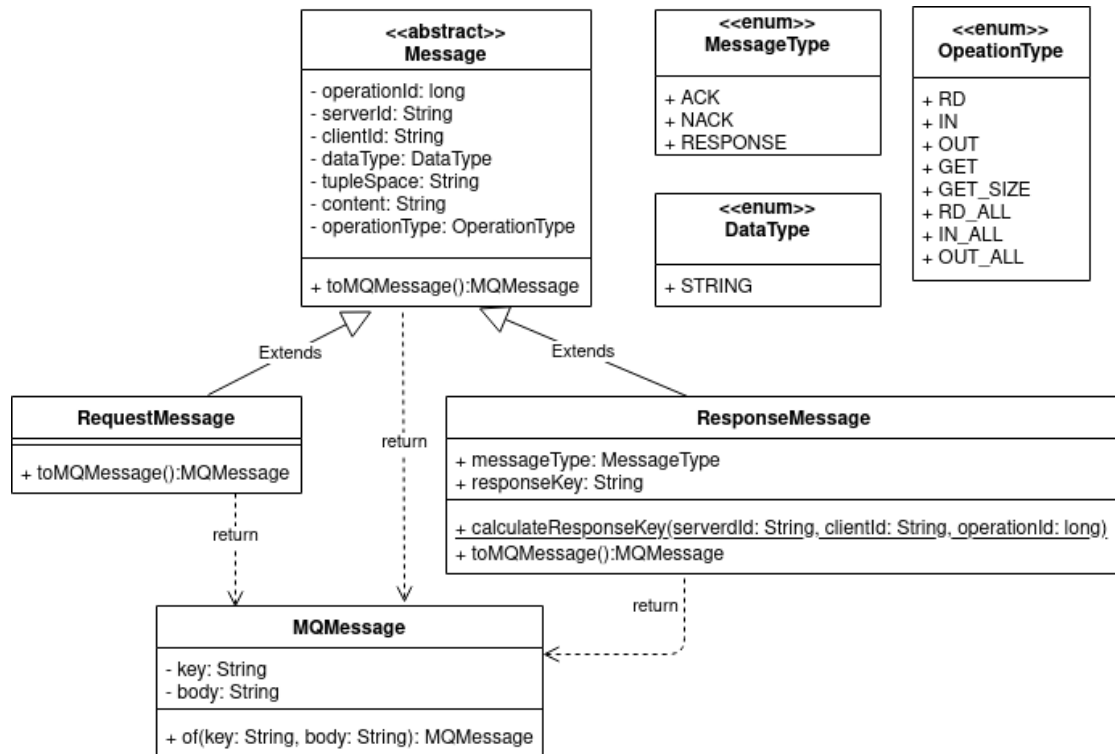


Figura 5: Rappresentazione dei messaggi di richiesta e di risposta

4.6.2 Architettura LindaServer

Il servizio principale è costituito dall'interfaccia **Service** che espone i metodi principali per l'avvio e la conclusione del servizio. Tali metodi astratti vengono implementati nella classe **LindaService** che incorpora la logica di accettazione della richiesta e di risposta al client. Le operazioni vengono delegate allo spazio di tuple richiesto, e se non esiste tale spazio di tuple, allora ne verrà creato uno. Il suo compito principale è quello di creare le connessioni al broker e di reagire ai messaggi in arrivo.

All'arrivo di un **RequestMessage** si occupa di inviare un **ResponseMessage** di tipo ACK e notificare lo specifico **TupleSpace**. La classe astratta **GenericTupleSpace** implementa le primitive classiche e quelle bulk su qualsiasi spazio di tuple, mentre la sua specializzazione **TextualTupleSpace** implementa il metodo per mappare il messaggio in richiesta sullo spazio di tuple. Tale comportamento si evince dalla Figura 6.

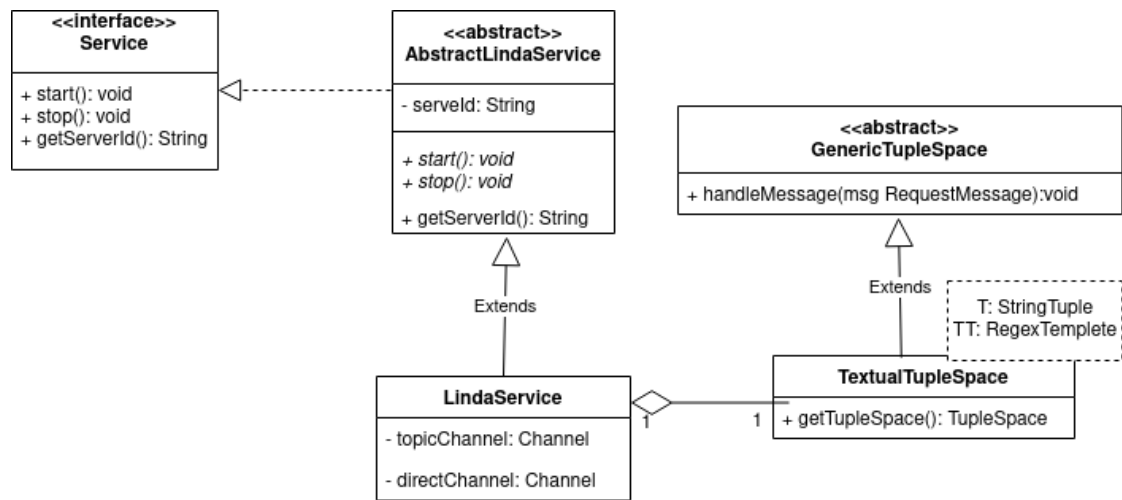


Figura 6: Rappresentazione della componente *LindaServer*

4.6.3 Architettura LindaClient

La classe `BrokerConnection` è la componente che permette l'interazione con lo spazio di tuple da parte del *LindaServer* ; in particolare tale classe si occupa di gestire e pubblicare i messaggi sul *Channel* dedicato ai topic. All'atto della pubblicazione della richiesta, dovrà essere generata una *responseKey* che sarà utilizzata per eseguire il *binding* sulla coda dove riceve le risposte da parte del server. L'interazione con il tuple space è favorita da un'interfaccia comune di un generico tupleSpace(`TupleSpace`), essa definisce i metodi invocabili su uno spazio di tuple, dove il tipo di *Tuple* e di *Tempate* non sono definiti ma sono generici. La classe `AbstractTupleSpace` implementa le primitive classifiche e bulk su un generico tuplespace lasciando invece alla classe `TextualTupleSpace` solo le operazioni che necessitano la dichiarazione dello specifico tipo di *Tuple* e di *Tempate*. In particolare la classe `TextualTupleSpace` aggancia una callback alla connessione al broker in modo tale da gestire nel modo opportuno il tipo dei dati in risposta. La rappresentazione della componente *LindaClient* si evince dalla Figura 7.

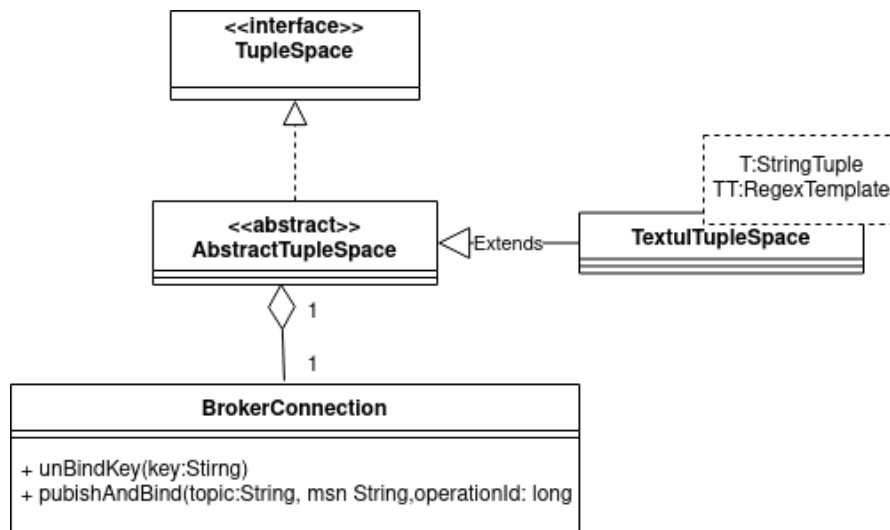


Figura 7: Rappresentazione della gerarchia di spazi di tuple

4.6.4 Struttura dei Package

Il servizio *LindaServer* e *LindaClient* seguono la stessa struttura di package; in particolare all'interno di *MQLinda.lindaServer* si implementano le classi principali **LindaService** e **AbstractLindaService** mentre nel package *linda* che si suddivide in *core* e *textual* sono definite, rispettivamente, le interfacce per il tuple space implementate dalla classe **GenericTupleSpace** e l'implementazione di **TextualTupleSpace** che rappresenta la classe principale che contiene lo spazio di tuple testuale e le primitive disponibili.

Mentre *MQLinda.lindaClient* contiene la classe **BrokerConnection** e il package *linda*. Il primo rappresenta la connessione con il broker e il package contiene una factory per la creazione di spazi di tuple. In questo package è presente anche una interfaccia per utilizzare il tuple space con semantica linda.

Il package *Message* del progetto *LindaMessage* contiene una serie di entità che rappresentano i messaggi che si scambiano i client con il *LindaServer*. Tali entità restituiscono, inoltre, delle rappresentazioni del messaggio che seguono la specifica del protocollo da noi creato. All'interno del package *util* si definiscono le classi di parsing e un builder, dove il primo realizza un parsing del messaggio secondo specifica, mentre il secondo è uno strumento di creazione utile per istanziare un nuovo messaggio. Vi è un ulteriore package, nominato *type*, dove sono definiti i tipi di messaggi, operazioni e dati.

Tale struttura si evince in Figura 8.

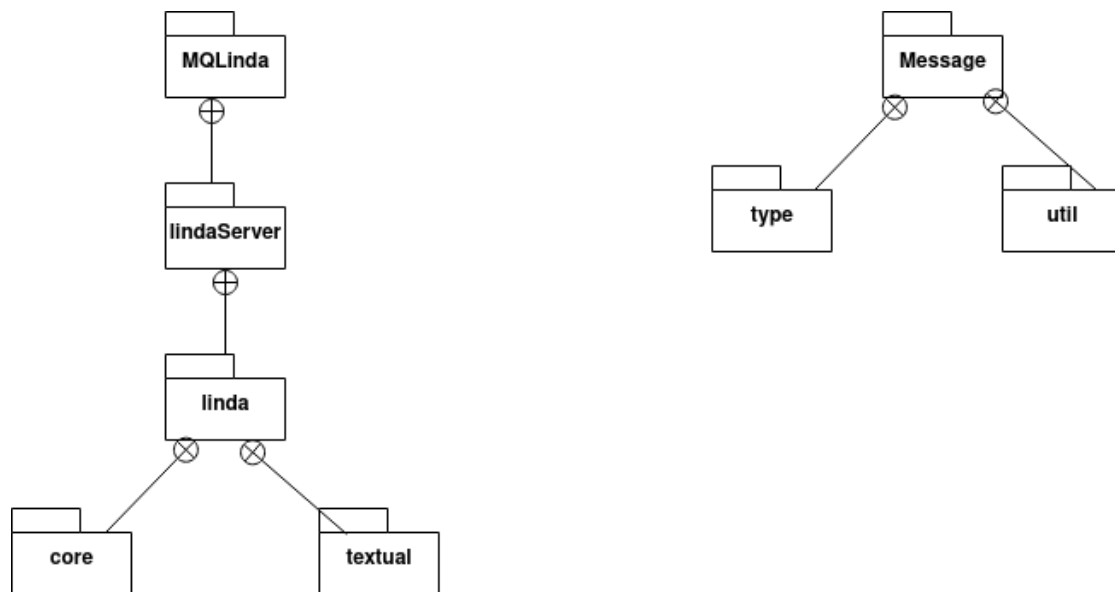


Figura 8: Diagramma dei package

4.6.5 Diagramma di Deploy

Il progetto sviluppato definisce un sistema distribuito in cui le diverse componenti software possono essere messe in esecuzione su diversi dispositivi fisici differenti all'interno di una rete di comunicazione. Le principali componenti di cui può essere fatto il deploy su diversi devices sono:

- Broker *AMQP*;
- *LindaServer* e i tuple space;
- Applicazione client;

La Figura 9 mostra il caso in cui *LindaServer* ed il broker *AMQP* sono in esecuzione sulla medesima macchina; mentre in figura 10 prevede che ciascuna delle componenti sia in esecuzione su un dispositivo differente. Tale scenario risulta utile nel caso si disponga già di un broker *AMQP* in funzione e lo si voglia utilizzare anche per questo sistema.

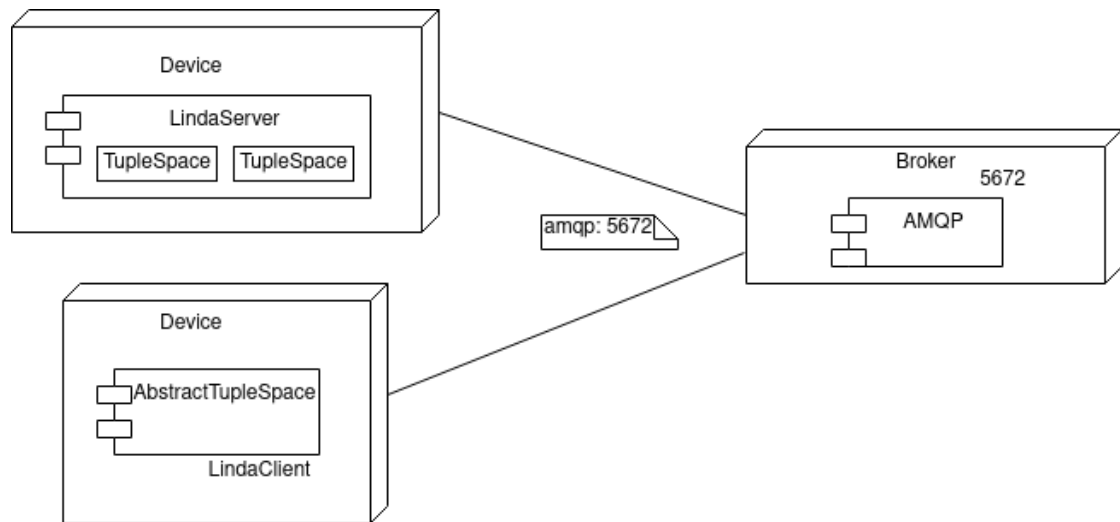


Figura 9: Diagramma di deploy in cui il broker e componente server sono in esecuzione su stessa macchina

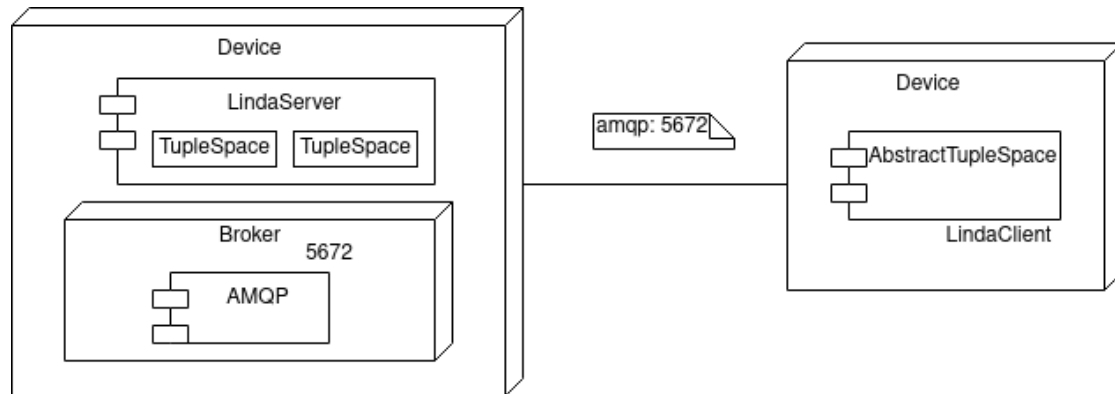


Figura 10: Diagramma di deploy in cui il broker *AMQP* , componente server e componente client sono in esecuzione su macchine diverse

4.7 Comportamento

Per rappresentare il comportamento del sistema, si è deciso di utilizzare il diagramma di attività poiché permette di rappresentare meglio le attività che devono svolgere le varie entità del sistema considerando anche la dimensione temporale. Questo tipo di diagramma è anche più utile rispetto ad altri perché ci permette di esprimere al meglio anche l'interazione tra le componenti, visto che è fortemente presente.

Nella figura 11 di seguito, si mostra un diagramma di attività del sistema che esegue una qualsiasi richiesta. Anche se le richieste hanno semantiche diverse, ad esempio le **Rd** e le **In** sono sospensive mentre le altre no, l'interazione tra le componenti è la medesima, quindi presentiamo un diagramma che è valido per ogni tipo di richiesta che viene effettuata dal client.

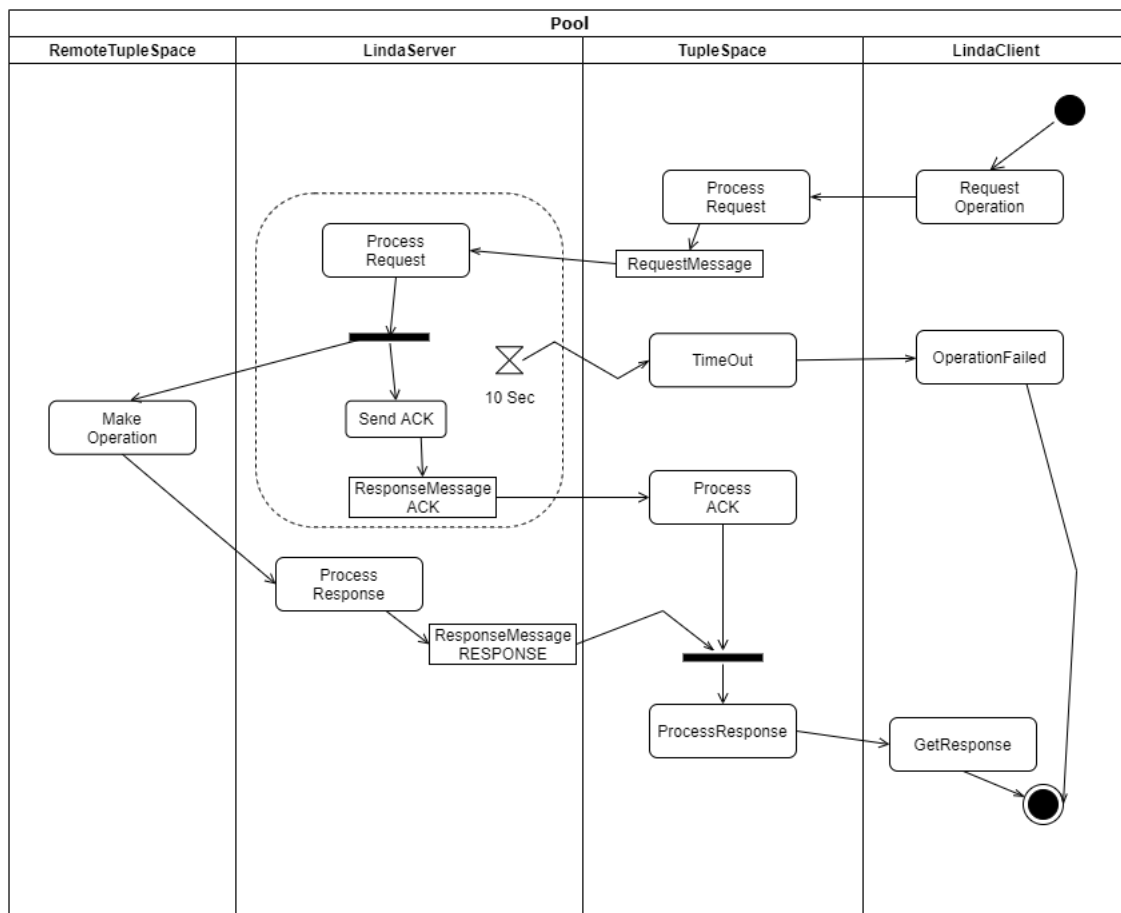


Figura 11: Diagramma di attività che rappresenta l'esecuzione di una operazione nel nostro sistema

4.8 Interazione

Durante la progettazione abbiamo studiato come sfruttare le varie possibilità del protocollo *AMQP* per fare interagire le varie componenti software. Lo scopo dell'interazione era quello di mettere in collegamento la rappresentazione del tuple space del client con il tuple space remoto. Nella figura 12 viene mostrato il diagramma di sequenza delle operazioni che vengono eseguite quando si crea un tuple space e cosa succede quando si effettua una chiamata al tuple space remoto. Nonostante la presenza di primitive con diversa semantica, l'interazione con il server rimane pressoché identica. Bisogna fare notare che l'ACK viene inviato prima che il server inizi l'elaborazione della richiesta poiché l'ACK significa che la richiesta è arrivata e il corpo del messaggio è formattato correttamente, di conseguenza la richiesta può essere processata. Nel Client, con la nostra soluzione, tutte le comunicazioni passano per il *BrokerConnection*, poiché tale entità gestisce le comunicazioni verso e da il Broker *AMQP*.

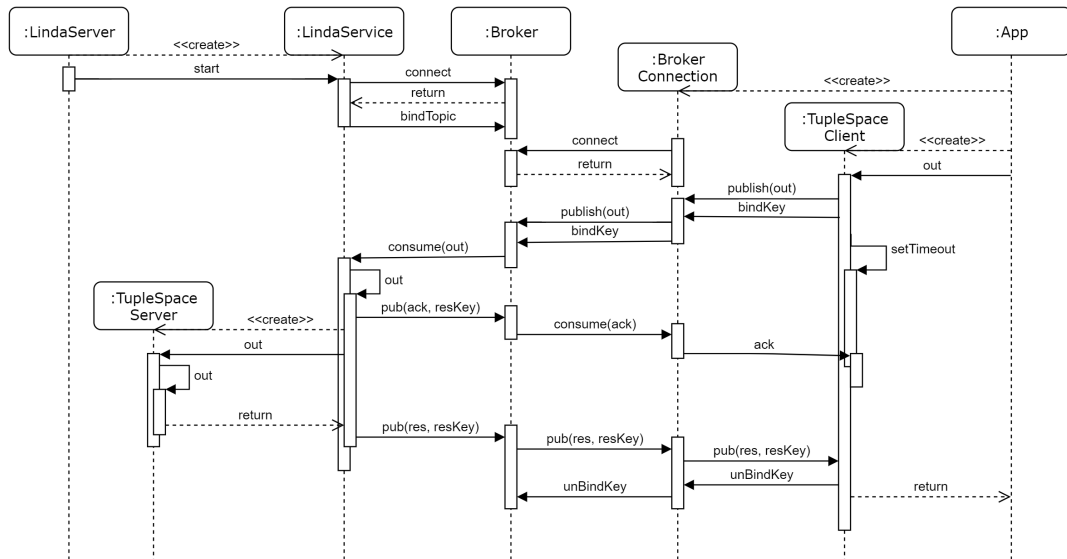


Figura 12: Diagramma di sequenza che mostra l'interazione tra la libreria client e il server

5 Dettagli implementativi

Uno dei obiettivi che ci eravamo proposti nel progetto era quello rendere più semplice possibile l'estensibilità del codice al fine di poter gestire spazi di tuple con differenti tipologie di dato.

Generalizzazione La facilitazione da questo punto di vista, proviene dalla generalizzazione di tutto quel codice che si sarebbe dovuto riscrivere per ogni spazio di tuple.

LindaServer Questo ha portato alla creazione di una classe astratta e generica nel *LindaServer* chiamata **GenericTupleSpace**. Per ogni spazio di tuple si dovrà creare una classe dedicata che estende il **GenericTupleSpace**, ne definisce i tipi, e implementi la funzione `handleMessage`. Questa funzione accetta come parametro una messaggio di richiesta e ritorna una **CompletableFuture** di tipo stringa, e quando sarà completata, dovrà essergli assegnato il contenuto del messaggio di risposta.

LindaClient Una soluzione analoga è stata fatta nel *LindaClient*. Nel *LindaClient* la classe astratta ha il nome **AbstractTupleSpace**. Le classi che la estendono hanno come compito quello di assegnare la callback alla coda dei messaggi di risposta del server. Per fare questa assegnazione, è stata appositamente creata la funzione `setUpConsume()` che accetta come parametro proprio una istanza di **DeliverCallback** che viene eseguita dal thread dei messaggi di risposta di *RabbitMQ*.

Creazione TupleSpace Quando riceve una richiesta, il *LindaServer* deve ottenere l'istanza del tuple space che fa riferimento a quel nome. Questo ci ha portato alla creazione di una funzione statica nel *TextualTupleSpace* che controlla se esiste una istanza della tupla con un determinato nome, se esiste ritorna tale istanza, altrimenti ritorna un nuovo tuple space.

AbstractTupleSpace A differenza del server, che quando vengono chiamate le operazioni esse vengono eseguite sullo spazio di tuple, nel client la classe *AbstractTupleSpace* si occupa di creare un *RequestMessage* ad-hoc per ogni operazione. Le richieste vengono effettuate utilizzando sia i valori utilizzati durante la creazione del TupleSpace, sia i valori passati nella chiamata a funzione della operazione.

La classe *AbstractTupleSpace* ha il duplice compito di eseguire le richieste e mantenere un timer per controllare, periodicamente, se qualche richiesta eccede il timeout.

La classe *TextualTupleSpace*, che estende *AbstractTupleSpace*, deve definire la callback per gestire le risposte. Un limite al momento presente nella libreria è che con una istanza di *BrokerConnection* si può gestire soltanto un TupleSpace.

6 Autovalutazione/Validazione

Per la validazione del processo sono stati utilizzati test e l'approccio utilizzato è stato quello di portare i test a pari passo con lo sviluppo per quanto concerne le funzionalità principali. Sono stati sviluppati sia test di unità per alcune classi sia test di integrazione che verificano il funzionamento globale del sistema. In particolare:

- In *LindaServer* si testano le primitive Linda che il server mette a disposizione su un tuple space in locale.
- In *LindaClient* viene testata l'implementazione dello spazio di tuple delle stringhe ed il metodo *match()* del *Template*. Inoltre si è effettuato un test di integrazione, per cui è necessario che venga lanciata prima di esso un'istanza del *LindaServer* e si verifica, quindi, il corretto funzionamento delle primitive e la comunicazione tra di essi.
- In *Message* vengono testati i messaggi di alto livello del protocollo, in particolare viene testato il metodo *toMQMessage()* per la conversione in messaggi direttamente utilizzabili dal *LindaClient*. Inoltre viene testato il corretto funzionamento del metodo *build()* del *MessageBuilder* che si occupa della creazione di un messaggio di risposta o di richiesta in base agli attributi che il metodo riceve. Il parser dei messaggi viene testato attraverso test sulle possibili eccezioni che verrebbero lanciate nel caso di formattazioni errate.

Per verificare la coverage si è utilizzato il plugin Jacoco [3] che ci ha permesso di valutare la percentuale di copertura del codice e dei i branch. In particolare si ottiene una copertura delle istruzioni del 77% e del 59% sui branch. Tutti i test passano e vengono realizzati con il framework JUnit versione 4.

7 Istruzioni di deployment

Per mettere in campo il prototipo è necessario:

- Installare il Java development kit v11 o superiore;
- Clonare il progetto dal seguente link: <https://gitlab.com/pika-lab/courses/ds/projects/sd-project-cavallucci-golinucci-aa1920.git>
- Scaricare e installare *RabbitMQ* server che avrà la funzione di broker;
- Aprire il terminale e posizionarsi dentro `./LindaServer`;
- Eseguire il comando `.gradlew build` per creare la build dal codice sorgente;
- Eseguire il comando `.gradlew test` per lanciare i test del progetto (ogni sottoprogetto ha i suoi test);
- Eseguire il comando `.gradlew run build` per avviare il server.

Una volta eseguite queste operazioni il servizio è attivo e operabile. Successivamente sarà necessario avviare l'applicazione client che interagirà con il server.

8 Esempi di utilizzo

Di seguito verranno descritti i passaggi per utilizzare il software proposto.

Configurazione LindaServer Il server è considerato come un servizio in esecuzione. Pertanto esso presenta un file che permette di definire alcune configurazioni nel server. Le configurazioni sono espresse in formato JSON ed esse sono:

- *brokerIp*: indirizzo IP del broker a cui il server si collega;
- *brokerPort*: porta TCP del broker a cui il server si collega;
- *brokerUser*: nome utente che il server utilizza per connettersi con il broker;
- *brokerPassword*: password che il server utilizza per connettersi con il broker;
- *requestExchange*: nome dell'exchange che gestisce le richieste del client verso il server;
- *responseExchange*: nome dell'exchange che gestisce le risposte del server verso il client;
- *serverID*: codice identificativo del server.

La configurazione di default prevede i seguenti valori:

NAME	TYPE	VALUE
brokerIp	string	127.0.0.1
brokerPort	number	5672
brokerUser	string	guest
brokerPassword	string	guest
requestExchange	string	topic_req
responseExchange	string	direct_resp
serverID	string	first

la configurazione viene caricata all'avvio del server. Di conseguenza un cambiamento della configurazione diventa effettivo una volta riavviato il servizio.

Libreria LindaClient L'utilizzo della libreria client è molto semplice. In un primo momento si ha la necessità di creare una connessione con il broker. Per avere una connessione con il broker basta creare una istanza della classe *BrokerConnection*

```
1 String brokerIp = "127.0.0.1";
2 int brokerPort = 5672;
3 String serverId = "first";
4 BrokerConnection connection = new BrokerConnection(
5     brokerIp,
```

```

6     brokerPort ,
7     serverId ,
8     clientId ,
9     EXCHANGE_NAME ,
10    RESPONSE_EXCHANGE);

```

Una volta che è stata creata una istanza della connessione, possiamo creare il tuple space.

```

1 String tupleSpaceName = "tp1";
2 TextualTupleSpace tupleSpace = new TextualTupleSpace(
3     connection ,
4     tupleSpaceName ,
5     Executors.newFixedThreadPool(1));

```

Ora che abbiamo l'istanza del tuple space, possiamo fare eseguire tutte le primitive implementate mediante l'istanza stessa. Di seguito ci sarà un esempio delle primitive classiche.

```

1 // primitiva out
2 StringTuple tp = StringTuple.of("foo");
3 CompletableFuture<StringTuple> operationFuture =
4     tupleSpace.out(tp);
5 System.out.println("out response" +
6     operationFuture.get().getValue());
7
8 // primitiva rd
9 CompletableFuture<StringTuple> operationFuture =
10    tupleSpace.rd(RegexTemplate.of(".*"));
11 System.out.println("rd response" +
12    operationFuture.get().getValue());
13
14 // primitiva in
15 CompletableFuture<StringTuple> operationFuture =
16    tupleSpace.in(RegexTemplate.of(".*"));
17 System.out.println("in response" +
18    operationFuture.get().getValue());

```

Ogni chiamata ritorna una istanza della classe `CompletableFuture<T>` dove T, in questo caso, è la classe `StringTuple`.

Questa semplicità è data dalla volontà di voler esporre soltanto quella che è la semantica di Linda, nascondendo quanto più possibile la logica implementativa e rendere minimo l'impatto che si ha con il broker.

Applicazione Demo Abbiamo sviluppato una demo dimostrativa tramite JavaFX nella quale mostriamo un possibile utilizzo del prototipo. In particolare tramite l'applicazione si vuole dare la possibilità di prenotare auto appartenenti ad una flotta aziendale.

Si mettono a disposizione le seguenti azioni:

- Acquisizione dell'identificativo un auto dal garage aziendale(per semplicità si fornisce il codice identificativo);
- Restituzione dell'auto precedentemente acquisita.

Si sono quindi creati due TupleSpace: il primo contenente le auto disponibili e uno dove memorizzare le auto assegnate ai dipendenti; il popolamento del TupleSpace con le auto a disposizione avviene da parte del LindaServer lanciando `./LindaServerApp initDemo`.

9 Conclusioni

Il sistema realizzato nel corso del progetto risulta essere coerente con gli obiettivi stabiliti per il progetto, ovvero la creazione di uno spazio di tuple remoto che sia accessibile a più utenti, mediante protocollo *AMQP*.

I componenti principali utilizzati sono:

- **LindaServer**: tale componente rimane in attesa di ricevere dei messaggi da parte dei client, e reagisce alla loro ricezione. Esso inoltra le richieste ai vari tuple space e una volta ottenuta la risposta la impacchetta e la inoltra al client. Tale servizio deve essere sempre attivo;
- **LindaClient**: questo componente è una libreria Client che permette l'interazione con il server e lo spazio di tuple, nascondendo l'interazione presente tra questi, ma mostrando una semantica in linea con quella definita da Linda;
- **LindaMessage**: componente realizzato per unificare la struttura dei messaggi sia nel client che nel server. Risulta comodo poiché implementa tutte quelle funzioni di parsing di stringhe e funzioni per convertire un messaggio in stringa, secondo specifica di protocollo da noi realizzata.

Come da specifica di progetto, le componenti utilizzano un broker *AMQP* per comunicare. Mediante opportuna configurazione, si può utilizzare un broker già esistente e in esecuzione, oppure si può avviare una nuova istanza del broker. Il broker è un elemento che deve restare sempre attivo come il *LindaServer* in quanto unico canale di comunicazione tra questi.

Il codice realizzato, ha sfruttato noti pattern di progettazione, ad esempio le varie Factory e Builder realizzati. Inoltre il codice realizzato è stato opportunamente controllato mediante Test. Tali test non sono sicuramente esaustivi, ma mediante l'analisi della coverage dei test abbiamo cercato di coprire quelle che ritenevamo le parti più delicate del codice.

La libreria *LindaClient*, sviluppata in Java nonostante le basse prestazioni, soprattutto in contesti IoT, ci permette una grande portabilità. Questa adozione è stata anche giustificata dal fatto che le operazioni che svolge non sono CPU intensive o bloccanti (il processo che elabora la richiesta non si blocca in attesa della risposta da parte del server).

Mediante questa relazione, inoltre si è prodotta una buona documentazione che percorre tutto il progetto, spiegandone le fasi e tutte le specifiche realizzate per realizzare tale progetto. Infatti siamo andati a descrivere lo scambio dei messaggi, il formato dei messaggi, e la struttura delle varie componenti, e la loro interazione.

9.1 Sviluppi futuri

Gli sviluppi futuri primari che prevediamo sono:

- realizzazione delle primitive predicative (goal opzionale);
- realizzazione di una libreria client che sfrutti linguaggi diversi da Java;
- utilizzo di una singola istanza di *BrokerConnection* per più *TupleSpace*;
- integrazione degli aspetti di sicurezza informatica;
- integrazione di ulteriori formati di messaggi per il protocollo (es. JSON);
- integrazione di ulteriori test per aumentare la affidabilità del codice.

9.2 Cosa abbiamo imparato

Con questo progetto abbiamo:

- imparato a modellare e implementare una specifica di protocollo personalizzato alle nostre esigenze;
- l'importanza di avere un sistema di testing affidabile;
- appreso il protocollo di comunicazione *AMQP*;
- migliorata la conoscenza dello strumento di Gradle importando librerie esterne e utilizzando plugin.

Riferimenti bibliografici

- [1] Modello AMQP. <https://www.rabbitmq.com/tutorials/amqp-concepts.html>.
- [2] Sito AMQP. <https://www.amqp.org/>.
- [3] Sito Jacoco. <https://www.jacoco.org/jacoco/>.
- [4] Sito RabbitMQ. <https://www.rabbitmq.com/>.