Utilizzo del protocollo *STOMP* in un sistema ad agenti *Linda*-based

Riccardo Maldini
riccardo.maldini2@studio.unibo.it

Francesco Gorini
francesco.gorini@studio.unibo.it

Maggio 2020

Sommario

Nell'ambito dei sistemi distribuiti, sempre più si fa largo l'esigenza di scambiare dati velocemente tramite web browser. Applicazioni come giochi online multiplayer, ad esempio, richiedono sempre più throughput elevato e latenza ridotta; le connessioni devono rimanere attive per lunghi periodi di tempo, con update frequenti e bidirezionali. Protocolli quali *HTTP*, anche nelle versioni più recenti, presentano pesanti limitazioni sotto questo punto di vista. Risulta inoltre difficile separare lo strato middleware dal resto del sistema, rendendo trasparente la logica sottostante.

Tra i possibili approcci pensati per superare queste limitazioni, ne esiste uno molto particolare e poco conosciuto, che sfrutta in combinazione le peculiarità di tre di differenti tecnologie: il protocollo *WebSocket*, il protocollo *STOMP* e i *broker di messaggistica*.

In primo luogo, *WebSocket* supporta comunicazioni bi-direzionali, e consente di mantenere aperta la connessione per lunghi periodi di tempo. Risulta inoltre facilmente e ampiamente applicato nelle comunicazioni asincrone via browser.

STOMP è invece un protocollo progettato per permettere uno scambio asincrono di messaggi tra client, sfruttando un nodo server centrale che agisce da mediatore. La sua implementazione standard agisce su TCP, ma una sua estensione molto popolare poggia su *WebSocket* (*StompOverWebsocket*), estendendo quest'ultimo con le peculiarità del modello publish-subscribe.

Quindi uno stato middleware, basato su *broker di messaggistica*, consente di ottenere prestazioni elevate in termini di throughput e latenza.

Questo progetto si propone in particolare di approfondire il secondo aspetto citato: il protocollo *STOMP*, dimostrando come applicarlo in modo specifico ad un sistema *Linda*-based ad agenti. Ciò permette di mostrare caratteristiche e peculiarità del protocollo, e di fornire un modello di partenza per altri progetti basati su queste tecnologie.

Indice

Elenco delle figure

1 Obiettivi/Requisiti

Il progetto si pone come obiettivo quello di approfondire il protocollo *STOMP*, e di applicarlo nello sviluppo di un sistema ad agenti centralizzato. Rappresenta inoltre un punto di partenza per l'implementazione concreta del modello ad agenti su web browser, nel caso in cui si decidesse di estendere l'elaborato utilizzando le altre tecnologie della soluzione citata nell'abstract. Analizzando la problematica, si sono posti inoltre i seguenti requisiti nello sviluppo del sistema:

Requisiti architetturali Architettura composta da un modulo con il ruolo di *broker*, che agisce da centro di smistamento dei messaggi, e da differenti *nodi client*. Uno di questi agisce da gestore degli spazi di tuple, mentre gli altri rappresentano agenti in grado di interagire con tali spazi tramite *Linda* operation. Le primitive *Linda* verranno implementate sopra il protocollo *STOMP*, andando a definire un'architettura inquadrabile per ogni componente in diversi strati (o livelli di astrazione).

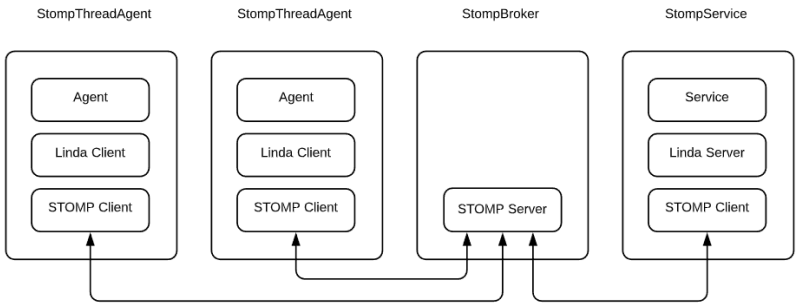


Figura 1: Architettura logica del sistema.

Requisiti software Utilizzo del linguaggio di programmazione Java, e processo di build automatizzato tramite il sistema *Gradle*. Utilizzo della libreria *Vertx-stomp* per l'implementazione dei client, in quanto facilmente integrabile nel sistema di automazione.

Requisiti di validazione Sviluppo di test che agiscono primariamente come *test di integrazione*, tali da coinvolgere tutti i moduli, con l'obiettivo principale di dimostrare la corretta implementazione delle primitive *Linda* tramite *STOMP*.

1.1 Scenari

La nostra soluzione andrà di fatto a comporre un'interfaccia per l'utente, al quale vengono rese disponibili in questo modo le primitive *Linda*. L'interfaccia, e più in generale l'intera architettura, può essere utilizzata in qualunque tipologia di sistema che basi la comunicazione su tale primitive. Questa si adatta in modo particolare ad applicazioni fruibili da browser, come chat, videogiochi online o web app, nel caso in cui venga utilizzata un'implementazione *STOMP* basata su *WebSocket*. Il progetto da noi proposto è stato sviluppato in Java, ma rappresenta un modello per un'implementazione equivalente in Javascript, più facilmente utilizzabile nei contesti sopracitati.

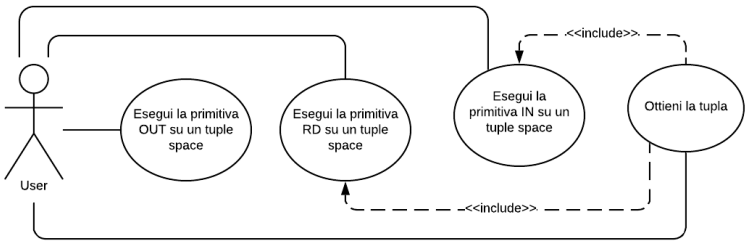


Figura 2: Diagramma dei casi d'uso.

L'interfaccia può però essere utilizzata, in un ambito più generale, in qualunque applicazione composta da più componenti che comunicano in rete, in maniera interattiva.

Per quanto tale protocollo si adatti ad applicazioni web, o comunque rivolte all'interazione finale con un utente "umano", *STOMP* non si adatta bene ad essere utilizzato in sistemi di componenti low energy o battery-optimized. Non va quindi inquadrato in scenari di ambito Industria 4.0 o IoT.

1.2 Politica di autovalutazione

Il nostro progetto è stato sviluppato in linguaggio Java, ed è stato utilizzato *jUnit 4* come framework per i test.

Per verificare il corretto funzionamento dei moduli, si sono creati dei test di integrazione, tali da coinvolgere tutti i principali componenti. Questi hanno l'obiettivo di verificare le proprietà delle primitive *Linda* (anche su molteplici tuple space in contemporanea), il corretto funzionamento dei singoli moduli, e alcune proprietà di interesse derivanti dall'utilizzo di *STOMP*. La correttezza dei risultati di progetto è quindi verificata e garantita dall'esito positivo dei test. I test portati avanti sono trattati in maniera più approfondita al capitolo 5.

La qualità del software è inoltre garantita da vari accorgimenti. Primo fra tutti il rispetto, nei punti critici dell'elaborato, di pratiche TDD (*Test Driven*

Development): per alcuni moduli, abbiamo portato avanti degli unit test, sia per verificarne la correttezza durante lo sviluppo, sia per mostrarne più chiaramente le caratteristiche.

Si è inoltre cercato di rispettare il più possibile pattern architetturali comuni (come Factory e Strategy), si sono aggiunti commenti alle sezioni di più difficile comprensione, e si sono adottate in maniera continuativa pratiche di refactoring.

2 Analisi dei requisiti

2.1 Concetti di base

Alla luce dei requisiti individuati, si è approfondita l'analisi, studiando gli aspetti teorici alla base del problema.

Il modello di comunicazione *Linda* Nell'ambito di questo progetto, il protocollo *STOMP* agisce come medium per l'implementazione di primitive di un più alto livello di astrazione. Queste fanno parte di *Linda* (1), un modello di coordinazione utilizzabile per la comunicazione tra diversi componenti di un sistema distribuito.

Linda permette di disaccoppiare le varie parti della comunicazione, andando a creare l'astrazione di memoria condivisa (*tuple space*), particolarmente utile in sistemi aperti ed eterogenei. Ogni spazio di tuple è paragonabile ad una lavagna, nella quale i vari componenti (*agent*) possono andare a leggere, modificare o aggiungere dati (*tuple*). L'interazione all'interno di questa può essere ottenuta sfruttando tre primitive di base:

- OUT: permette a un agente di inserire una tupla all'interno di un dato tuple space;
- RD: dà all'agente la possibilità di verificare se all'interno di un tuple space sia presente una determinata tupla (andando ad effettuarne pattern matching con un elemento, fornito dall'agente, denominato *template*); quindi, in caso affermativo, di leggerne il contenuto;
- IN: concede all'agente la possibilità di "consumare" una tupla, andando prima a verificarne la presenza nel tuple space tramite un template; quindi, in caso affermativo, di leggerne il contenuto ed eliminarla dallo spazio condiviso.

Il modello Linda può inoltre essere espanso con delle primitive a contorno, quali GET (che agisce in maniera analoga a RD, dando inoltre la possibilità di leggere tutte le tuple dello spazio condiviso, nel caso venga omesso il template) o GET_SIZE (che restituisce il numero di tuple all'interno dello spazio condiviso).

Va inoltre ricordato che la semantica di *Linda* deve tener conto di requisiti ben precisi, legati all'associatività dell'accesso alle tuple, al non determinismo delle operazioni, oltre all'essere di tipo generativo (una volta creata una tupla, questa è indipendente dallo stato dell'agente) e sospensivo (RD e IN vengono sospese fin quando una tupla combaciante con il template non è presente nello spazio condiviso).

Message-oriented Middleware Il middleware rappresenta l'infrastruttura software, più o meno complessa, necessaria per coordinare e gestire la comunicazione tra i vari componenti di un sistema distribuito (2). In particolare, Un middleware di tipo *Message-Oriented* (MOM) sfrutta, nella comunicazione, il concetto di messaggio. Ciò permette di implementare operazioni più complesse della semplice invocazione remota di una funzione a distanza, come ad esempio accade con middleware RPC (3).

MOM introduce il concetto di sistema di messaggistica distribuito (*distributed messaging*): un sistema tale da fornire un nodo centrale per la memorizzazione e lo smistamento dei messaggi da scambiare tra le varie componenti.

Message broker Il nodo centrale di un sistema del genere prende il nome di *message broker* (4). Tale componente permette di gestire i messaggi tramite due principali modelli di messaggistica:

- *Point-to-point*: sfrutta l'utilizzo di una queue in modaltà FIFO. Molteplici client possono pubblicare messaggi sulla stessa coda, mentre di norma un singolo client si pone in ascolto su di questa. I messaggi vengono quindi inoltrati secondo la politica FIFO al destinatario.

- *Publish-subscribe*: modello più articolato, tale da permettere comunicazioni many-to-many. Sfrutta il concetto di topic (argomento di interesse): molteplici client possono notificare al broker la volontà di sottoscrivere un determinato topic; quindi quest'ultimo provvederà ad inoltrare i messaggi pubblicati su tale topic a tutti i client interessati.

Esistono molteplici soluzioni pre-implementate, framework e librerie che permettono di sfruttare le caratteristiche dei *message broker*.

Le varie implementazioni disponibili di *message broker* supportano molteplici protocolli di comunicazione. Protocolli per la comunicazione middleware vanno in genere a collocarsi nel *livello applicazione* dello stack (TCP/IP o ISO/OSI), e si appoggiano al livello inferiore prevalentemente su TCP. Largamente diffusi in questo contesto sono i protocolli *MQTT*, *AMQP*, e *STOMP*, ognuno con le sue caratteristiche peculiari.

2.2 Il protocollo *STOMP*

L'intero progetto verte sull'utilizzo del protocollo *STOMP* (5); si è quindi ritenuto opportuno effettuare una panoramica sullo stato dell'arte del protocollo, e sulle principali funzionalità.

STOMP, acronimo di *Simple Text-Oriented Message Protocol*, trae le sue origini dalla necessità di far comunicare linguaggi di scripting, come Ruby, Python o PERL, con dei broker di messaggistica, principalmente per scopi industriali. In ambienti simili, le operazioni portate avanti dai singoli componenti sono relativamente semplici: l'invio di un messaggio, o il consumo di tutti i messaggi provenienti da una singola destinazione. Nasce quindi come alternativa a protocolli come *AMQP*, o altri più application-specific, come *OpenWire* per i broker JMS. Si distingue dai primi, principalmente per la sua *semplicità* e *leggerezza*. Va a coprire soltanto un subset delle operazioni di messaggistica, quelle più comunemente utilizzate, piuttosto che andare a definire un'API wire-level, globalmente completa. Semplicità che ne garantisce, all'atto pratico, un'amplissima *interoperabilità*:

- Viene adottato nativamente, o tramite estensioni in una vasta gamma di broker;
- Moltissime librerie e framework lo supportano, o prevedono estensioni che ne garantiscono il supporto. Anche creare un client o un server *STOMP* "from scratch", in assenza di soluzioni pre-implementate, non risulta particolarmente complesso;
- Può essere applicato come una vera e propria "estensione" ad altri protocolli, come nel caso di *WebSocket* (6).

2.2.1 Panoramica

STOMP è un protocollo frame-based, dove i frame sono modellati sulla falsa riga di *HTTP*. Un frame si compone di un comando, un set di header opzionali e un body opzionale.

La codifica di default utilizzata nei frame è UTF-8, seppur vengono supportate specifiche alternative per il body (nel progetto si andrà a utilizzare la codifica JSON). Da qui la dicitura text-oriented, essendo il frame, al pari di *HTTP*, human-readable.

Un generico client *STOMP* può agire, anche simultaneamente, sia come produttore che come consumatore.

La specifica più recente del protocollo è la 1.2; sostanzialmente retrocompatibile con le versioni precedenti, introduce alcune rifiniture ulteriori.

2.2.2 Struttura dei messaggi

Come detto, un frame *STOMP* si compone di un comando, degli header opzionali e di un body opzionale. Comando, header e body sono separati da caratteri EOL (carriage return opzionale e carattere spazio); il body è separato dagli header da un ulteriore spazio, e termina con il carattere speciale ^@. Gli header sono sempre opzionali, seppur sono presenti alcuni header standard (destination, heart-beat, content-length, content-type e altri) aventi lo scopo di aiutare il destinatario ad interpretare il messaggio. Il server può prevedere una dimensione limite per ogni frame, e rispondere in caso di dimensione eccessiva con un frame ERROR.

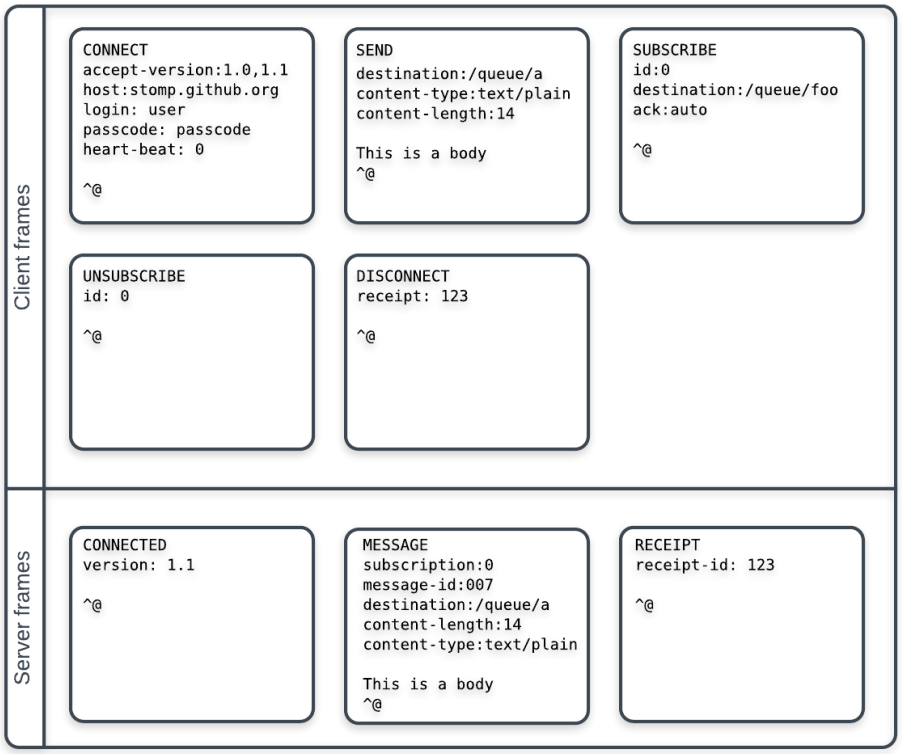


Figura 3: Rappresentazione dei principali frame che possono essere spediti tramite il protocollo *STOMP*, dai componenti client e server, comprensivi degli header più significativi.

2.2.3 Fase di connessione-disconnessione

Lo scambio di frame tra client e server presuppone la presenza di una connessione affidabile bi-direzionale sottostante, quale TCP. Lo stream viene inizializzato tramite una richiesta che parte dal client:

1. Il client invia un frame CONNECT al server (o in alternativa un frame STOMP, a partire dalla specifica 1.2). Il server può prevedere dei meccanisemi di autenticazione di base. A tal scopo, *STOMP* permette di includere in maniera non mandatoria gli header login e passcode. Possono anche essere stabiliti meccanismi di heart-beat tramite l'omonimo header.

2. Quindi, il server risponde con un frame CONNECTED in caso di avvenuta connessione.

Con un analogo meccanismo, il client può decidere di interrompere la connessione:

1. Il client invia al server un frame DISCONNECT.

2. Il server risponde con un frame RECEIPT, avente come header receipt-id l'id del messaggio di richiesta.

2.2.4 Scambio di messaggi

Una volta stabilita la connessione, i vari client possono comunicare tra loro in maniera indiretta, sfruttando il server come nodo di mediazione. Il flusso di comunicazione che permette di ottenere ciò è il seguente:

1. Un client A può porsi in ascolto su una destinazione nel server, inviando a questo un frame SUBSCRIBE, includendo l'header mandatorio destination[1].

   Un eventuale header ack all'interno del frame SUBSCRIBE permette di definire politiche di acknowledgment relativamente della ricezione dei frame, tramite l'utilizzo dai frame ACK - NACK. Di deafult, tali politiche sono settate a auto (nessun ack).

2. Quindi, un client B può spedire un messaggio in direzione di una destinazione, anche non avendo una sottoscrizione attiva su di questa, inviando un frame SEND al server. Questo dovrà includere l'header mandatorio destination, tale da indicare la destinazione, ed un body.

3. Alla ricezione del frame SEND, il server andrà a spedire, per ogni client in sottoscrizione alla destinazione indicata, un frame MESSAGE. Questo

[1] Per destinazione si intende il nome del topic o della queue al quale il messaggio è diretto. Nella pratica, la comunicazione diretta agisce soltanto tra client e server, e non in maniera diretta tra client. Le destinazioni non hanno inoltre una semantica prestabilita. Eventuali meccanismi gerarchici tra destinazioni, differenti modelli di messaggstica come queue o topic, vengono gestiti in maniera trasparente dal server.

erediterà dal frame SEND il body e gli eventuali header content-length e content-type. In questo esempio, il client A riceverà quindi il frame MESSAGE indicato.

Con un frame UNSUBSCRIBE, al quale è associato l'header obbligatorio destination, il client può terminare l'osservazione di una destinazione. Il protocollo permette inoltre di gestire *transazioni*, tramite i frame BEGIN, COMMIT e ABORT.

2.3 Tecnologie utilizzate

In seguito dell'analisi teorica è stata posta in atto una fase più operativa, andando a confrontare varie tecnologie e framework, selezionando quelle più adatte a questo progetto. Sono state descritte le principali tecnologie utilizzate, e il processo che ha portato alla scelta di queste.

Essendo lo scopo del progetto utilizzare il protocollo *STOMP*, e non di crearne una libreria from scratch, si è subito scartata l'ipotesi di implementare il protocollo da zero. Ci si è concentrati invece sulla ricerca di broker tale da supportare il protocollo, facilmente integrabile in un progetto Java, oltre ad una libreria che ne consentisse facilmente la creazione un client.

Sono state confrontate varie alternative broker pre-implementate, quali *RabbitMQ* e *ApacheMQ*. Ci si è però presto resi conto che l'utilizzo di un broker "completo" all'interno del progetto presentava problemi generali di integrazione e compatibilità.

La ricerca si è quindi spostata sui framework Java, i quali spesso forniscono librerie di interazione web per implementare facilmente client e server *STOMP*.

2.3.1 *Vertx* e *Vertx-stomp*

La scelta finale è ricaduta sull'utilizzo del framework *Vertx* (7), ampiamente utilizzato per lo sviluppo di applicazioni reattive, event-driven, sulla JVM. È un framework estremamente modulare: i vari moduli estendono il core con ulteriori funzioni di comunicazione web, supporto a differenti protocolli, tecnologie e linguaggi sulla JVM.

Vertx è stato selezionato in particolare per la presenza di un modulo che integra il supporto al protocollo di interesse per il progetto: *Vertx-stomp* (8). Questo fornisce in particolare delle implementazioni per server e client, ben documentate e altamente efficienti.

Lato server, ciò ci ha dato la possibilità di creare facilmente un server *STOMP* tale da poter gestire le politiche topic e queue, e di avere le funzionalità di un broker "minimale" per il protocollo in esame.

Lato client, si ha un'implementazione pronta del protocollo; abbiamo potuto quindi concentrarci sugli aspetti di maggiore interesse (quali l'implementazione delle primitive *Linda*).

2.3.2 *Jackson*

Jackson (9) è un processore JSON ad alte prestazioni per Java. Nell'ambito del progetto, è stata utilizzata questa libreria per effettuare il parsing e l'unparsing del body dei vari frame *STOMP* scambiati nel sistema. Per scelta progettuale, il body dei frame *STOMP* scambiati nel sistema segue la codifica JSON.

3 Design

3.1 Struttura

Per risolvere le problematiche di progetto, rispettando il protocollo in esame, si è deciso di agire andando a modellare tre principali entità:

- *StompThreadAgent*: un modulo avente il ruolo di agente generico, tale da poter interagire con uno o più spazi di tuple, tramite primitive *Linda*. L'agente è modellato come un Java Thread, ed espone all'utente la possibilità di stabilire dall'esterno il proprio comportamento (che sarà, per l'appunto, una sequenza di primitive *Linda*).

- *StompService*: modulo che agisce come gestore degli spazi di tuple centralizzato. L'idea di base è che l'utente abbia dall'esterno solo la possibilità di avviare e fermare tale servizio: i tuple space verranno gestiti in maniera automatizzata, in base all'interazione con i vari agenti.

- *StompBroker*: le due entità sopracitate, incorporano al loro interno dei client *STOMP*. Per consentire l'interazione tra i vari client, questo modulo implementa un server *STOMP*, che agisce come un nodo di smistamento dei messaggi.

Si è deciso all'atto pratico di strutturare il codice predisponendo tre principali package, che si potrebbe definire "core", tali da rappresentare le tre entità sopracitate. Altri package a contorno vanno poi a raccogliere funzionalità comuni a questi. Di seguito se ne andrà a descrivere la struttura interna, per poi andare a visualizzare le relazioni presenti tra i vari componenti.

3.1.1 *StompThreadAgent*

Questo modulo si occupa di implementare un'entità attiva capace di interagire con uno o più spazi di tuple, memorizzati in remoto, all'interno del Service. Nello specifico, l'utente può configurare il comportamento di un agente indicandogli di eseguire delle primitive *Linda*: operazioni utili allo scopo di creare, modificare o cancellare le informazioni all'interno degli spazi di tuple.

Ogni entità memorizza al proprio interno dei riferimenti agli spazi di tuple sui quali l'utente richiede operazioni. Qualora sia operata una qualsiasi *Linda* operation su uno spazio di tuple non noto dall'agente, questo provvederà in maniera automatica (lazy) a crearne un riferimento e a inoltrare una rischiesta al Service.

Il concetto di agente viene modellato con la classe *StompThreadAgent*, estendendo di fatto la classe *ThreadAgent* vista in laboratorio. Ad ogni agente viene associato un client *STOMP*, che gestisce a basso livello le comunicazioni con il Service, tramite questo protocollo.

I riferimenti agli spazi di tuple vengono modellati tramite la classe *RemoteTextualSpaceImpl*. Questa classe rappresenta l'implementazione di un'interfaccia di comunicazione con uno spazio di tuple remoto, e si occupa di effettuare richieste relative al tuple space collegato.

Le operazioni richieste dall'utente sono di fatto bloccanti: questa classe si occupa di memorizzare le richieste pendenti, e di sbloccarle nel momento in cui il Service inoltra ad esso la risposta.

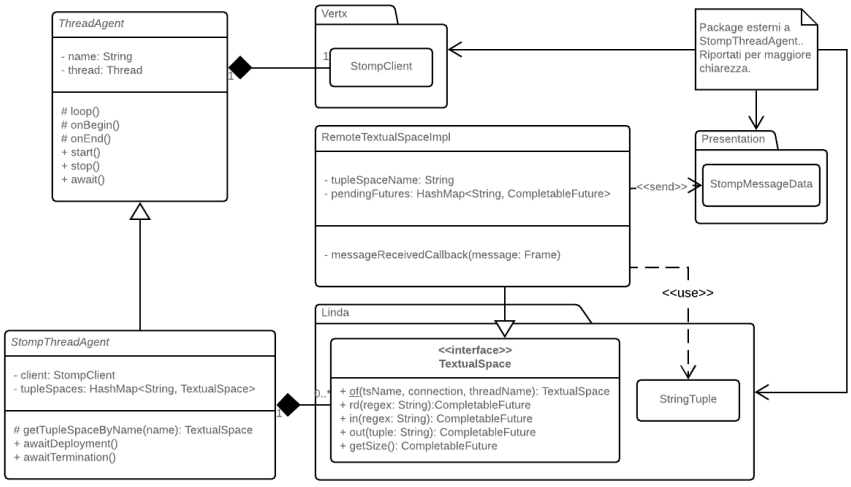


Figura 4: Diagramma delle classi del package *StompThreadAgent*.

3.1.2 *StompService*

La classe *StompService*, contenuta all'interno dell'omonimo modulo, rappresenta una sorta di servizio web, che ha lo scopo di porsi in ascolto su una queue dedicata in attesa di *Linda* request, processarle e rispondere adeguatamente. Estende la classe *VertxVerticle*, implementando quindi un servizio indipendente che una volta avviato deve rimanere costantemente attivo in attesa di messaggi.

StompService detiene quindi il ruolo di gestore degli spazi di tuple. Al suo interno presenta un'istanza di *TextualSpaceStorage*, un modulo che ha come fine quello di emulare lo storage dei vari spazi di tuple (nel progetto non viene gestita la persistenza). Il modulo dà la possibilità di creare ed accedere a diverse istanze di *TextualSpaceImpl*, che in maniera analoga a quanto visto nelle esercitazioni, implementa il concetto di tuple space e ne gestisce in maniera ordinata le operazioni.

Per poter interagire con i vari agenti, il servizio contiene un client *STOMP* dedicato; questo si pone in ascolto, collegandosi con il broker, su una queue adibita alla ricezione di richieste. Alla ricezione di queste, il servizio va ad agire come da indicazioni sugli spazi di tuple, e ne ricava i dati necessari per generare una risposta. La logica delle varie operazioni che possono essere in atto nello storage viene gestita da un'istanza di *TupleSpaceHandler*. Il modo in cui agisce l'implementazione di questa interfaccia è in parte assimilabile a *TupleSpaceApi* visto in laboratorio.

Quindi, una volta generata la risposta, questa viene inoltrata, sotto forma di frame *STOMP* sul topic associato al rispettivo tuple space.

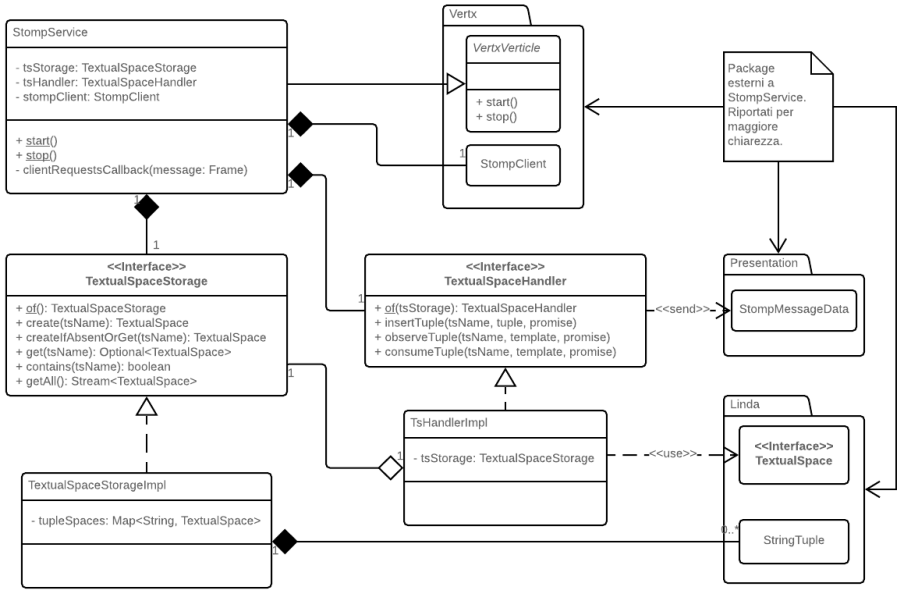


Figura 5: Diagramma delle classi del package *StompService*.

3.1.3 *StompBroker*

Questo componente si è reso necessario per permettere, a più basso livello, lo smistamento dei vari messaggi tra le entità. Come descritto in precedenza, Service ed Agent implementano al loro interno dei client *STOMP*, che per loro natura hanno bisogno di un nodo centrale che agisca da broker. *StompBroker* è la classe utilizzata a tal scopo. Questa implementa un *VertxVerticle*: è quindi di fatto un servizio indipendente, che incapsula al suo interno un'istanza della classe *StompServer*.

Questo servizio è necessario per effettuare test e simulazioni sul sistema: va avviato come primo componente e deve rimanere attivo nell'arco di ogni test, permettendo lo smistamento dei messaggi tra i componenti tramite modelli di messaggistica basati su topic e queue.

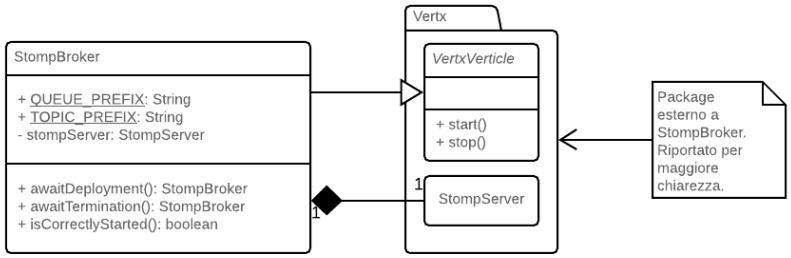


Figura 6: Diagramma delle classi del package *StompBroker*.

3.1.4 Presentation layer

All'interno del progetto, si è reso necessario comporre i messaggi in maniera strutturata. *STOMP* permette ai client di inviare messaggi in direzione di date destinazioni, tramite un frame SEND; quindi il server inoltra, tramite dei frame MESSAGE, il messaggio in direzione dei sottoscrittori della destinazione. In questo procedimento, tutte le informazioni devono essere contenute all'interno del body del messaggio: porre informazioni tra gli header non è sicuro, in quanto, a seconda del modo in cui *STOMP* viene implementato dalla rispettiva libreria, questi potrebbero andare persi nel passaggio da SEND a MESSAGE.

Si è reso quindi necessario andare a definire una struttura anche all'interno del body, per poter rappresentare dati eterogenei in maniera strutturata. Si è deciso in particolare di utilizzare JSON come formato del body del messaggio. Tutti i messaggi circolanti nella rete hanno come body un file JSON con dei campi fissi, in formato campo-valore, che all'atto pratico possono essere utilizzati per implementare le varie *Linda* request e le rispettive risposte. I campi devono essere sempre presenti, ma non è obbligatorio porre un valore al loro interno. I campi utilizzabili sono descritti nella tabella 1.

Si è creato una sorta di "protocollo intermedio", che consente di utilizzare ognuna delle primitive *Linda* tramite uno scambio di messaggi request-response tra agente e service. Ogni agente ha la possibilità di inoltrare un frame di richiesta in direzione del service, e questo andrà a rispondere con il frame complementare. Le possibili richieste e risposte utilizzabili all'interno del sistema sono descritte nella tabella 2.

All'interno del progetto, tutto ciò che concerne il layer di presentazione è contenuto nel package presentation. Questo di fatto si struttura intorno alla classe *StompMessageData*: lo scopo di questa è modellare il corpo dei frame *STOMP* da scambiare nel sistema, fornendone metodi per il parsing-unparsing da formati come JSON, YAML e XML a POJO e viceversa. Per far ciò, viene esteso il comportamento della classe Data vista in laboratorio, utilizzando quindi per la serializzazione la libreria *Jackson*.

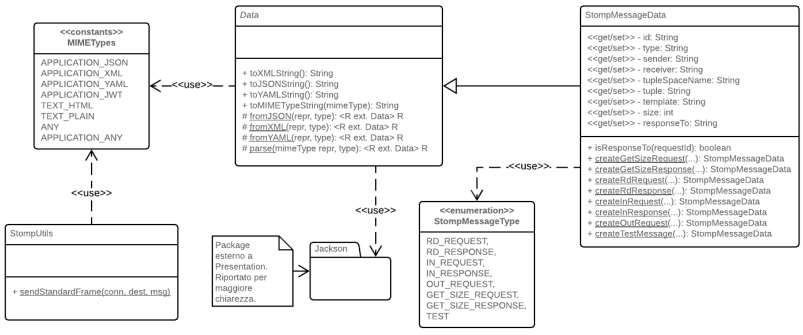


Figura 7: Diagramma delle classi del package Presentation.

Nome del campo	Descrizione	Valori
id	L'identificativo univoco del messaggio nella rete.	Valori in formato UUID.
type	La tipologia di operazione che il messaggio implementa.	Ad esempio "RD_REQUEST", "RD_RESPONSE".
sender	Il componente del sistema che ha spedito il messaggio.	Ad esempio "StompService", "Agent1".
receiver	Il componente del sistema al quale il messaggio è diretto.	Ad esempio "StompService", "Agent1".
tupleSpaceName	Lo spazio di tuple al quale l'operazione fa riferimento.	Ad esempio "ts1", "ts2".
tuple	L'eventuale tupla inviata in una RD_RESPONSE o in un OUT_REQUEST.	Ad esempio "recipient: bob, Body della tupla", "Body della tupla".
template	L'eventuale template inviato in una RD_REQUEST o in una IN_REQUEST	Ad esempio ".*?bob.*", ".*?carl.*".
size	Contiene il numero di tuple contenute nel ts, in una GET_SIZE_RESPONSE.	Ad esempio 4, 10.
responseTo	Utilizzato nei messaggi di risposta, contiene l'id della richiesta che l'ha generato.	Valori in formato UUID.

Tabella 1: Tabella che descrive i possibili valori contenuti all'interno del body dei frame scambiati nella rete.

Operazione	Mittente	Descrizione	Campi
RD_REQUEST	Agent	Esprime la volontà di un agent di effettuare una RD all'interno di un ts.	type, sender, receiver, tsName, template
RD_RESPONSE	Service	Completa la RD richiesta dall'agent, includendo la tupla ricercata.	type, sender, receiver, tsName, tuple
IN_REQUEST	Agent	Esprime la volontà di un agent di effettuare una in all'interno di un ts.	type, sender, receiver, tsName, template
IN_RESPONSE	Service	Completa la IN richiesta dall'agent, includendo la tupla ricercata.	type, sender, receiver, tsName, tuple
OUT_REQUEST	Agent	Esprime la volontà di un agent di effettuare una OUT, ponendo una tupla all'interno di un ts.	type, sender, receiver, tsName, tuple
GET_SIZE _REQUEST	Agent	Esprime la volontà di un agent di conoscere il numero di tuple all'interno di un ts.	type, sender, receiver, tsName
GET_SIZE _RESPONSE	Service	Completa la richiesta GET_SIZE _REQUEST dell'agent, includendo il numero di tuple nel ts richiesto.	type, sender, receiver, tsName, size

Tabella 2: Tabella che descrive le operazioni che i frame scambiati nella rete possono rappresentare tramite il loro body, andandone a descrivere i campi utilizzati

3.1.5 Altri package trasversali

Sono presenti poi all'interno del progetto altri componenti "trasversali", ovvero utilizzati da diversi componenti core:

- *Linda*: questo package rappresenta i concetti di base attorno al quale è stato costruito il progetto. Al proprio interno modella le idee di spazio di tuple, tupla, template e le loro estensioni testuali. Essendo la rappresentazione la stessa di quella vista durante le lezioni di laboratorio, questo package viene ereditato dai progetti precedenti. Al suo interno, è anche stata mantenuta l'implementazione *TextualSpaceImpl*, in quanto utilizzata per emulare la memorizzazione delle tuple.

  Nel package è inclusa una classe di utility trasversale, *StompUtils*, che include un wrapper per l'invio di frame *STOMP* all'interno del sistema, seguendo le best practices riguardanti l'inclusione degli header standard *content-type* e *content-length* all'interno dei frame inviati. Di default, *Vertx-stomp* include solamente l'header *destination*.

- *Exceptions*: il package raccoglie delle eccezioni che possono essere sollevate dai vari componenti.

3.2 Comportamento

Come visto nel capitolo precedente, il progetto si articola in tre principali entità, ognuna con il proprio flusso di esecuzione interno. In particolare, broker e service presentano comportamenti ciclici, essendo delle entità 'reattive'. In altre parole, queste avviano i propri comportamenti sulla base di eventi, che li portano ad avviare procedure interne.

- Il broker è un'entità singola e reattiva, che agisce come nodo di smistamento, andando a inoltrare i messaggi in arrivo su un topic a tutte le entità che lo hanno sottoscritto (o, in caso di modello a queue, all'entità corrispondente). Il suo comportamento è quindi periodico: i task devono essere eseguiti periodicamente fin quando il sistema è attivo. Il broker è un'entità passiva, in quanto non può decidere autonomamente quando terminare, ma deve essere richiesta la sua terminazione dall'esterno.

- Il service è anch'esso un'entità singola, reattiva, passiva e con comportamento periodico, ma di più "alto livello": reagisce cioè in risposta a primitive *Linda*, ricevute dai vari agenti, astraendo dal concetto di broker (seppur i messaggi passano per tale nodo). Ad ogni primitiva, il service risponde con i dati richiesti.

- *StompThreadAgent* agisce invece come un'entità attiva, non periodica, in grado di effettuare richieste *Linda* al service e di gestirne le risposte. Anche qua è possibile astrarre dagli aspetti relativi allo smistamento dei messaggi. L'agente in questo caso non è necessariamente un'entità singola: una situazione di esecuzione tipica prevede la presenza di più agenti in contemporanea.

  A differenza delle entità precedentemente descritte, l'agent è un modulo attivo: esso termina quando all'interno del proprio loop di esecuzione viene eseguito il metodo stop().

Nella figura seguente viene mostrato più chiaramente il behaviour interno di ogni componente.

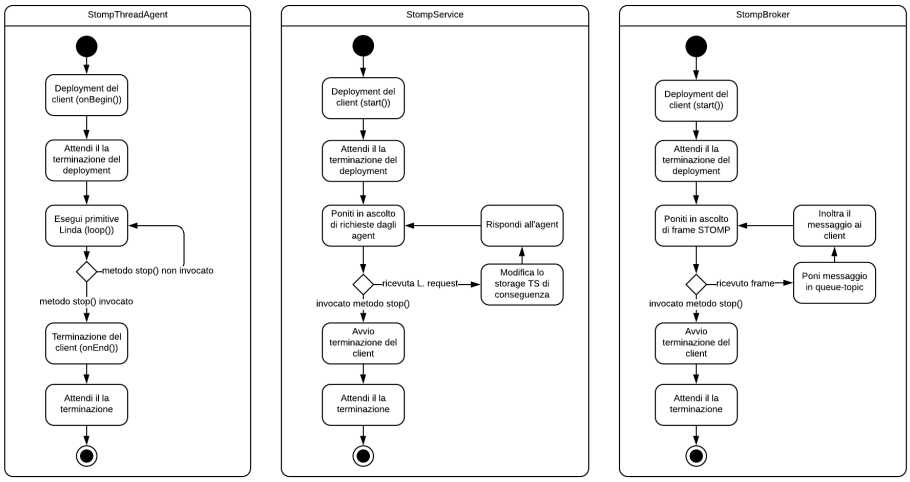


Figura 8: Diagramma delle attività che mostra il comportamento di ogni componente separatamente.

Per una maggiore comprensione del flusso di esecuzione, ritorna però utile avere una visione trasversale dei vari comportamenti che il sistema può assumere. I seguenti diagrammi delle attività consentono di visualizzare il flusso di esecuzione completo per ogni primitiva implementata, dall'invocazione al ritorno del risultato. Dettagli di interazione di più basso livello (relativi al protocollo *STOMP* e ai modelli di messaggistica utilizzati) vengono trattati nella sottosezione successiva.

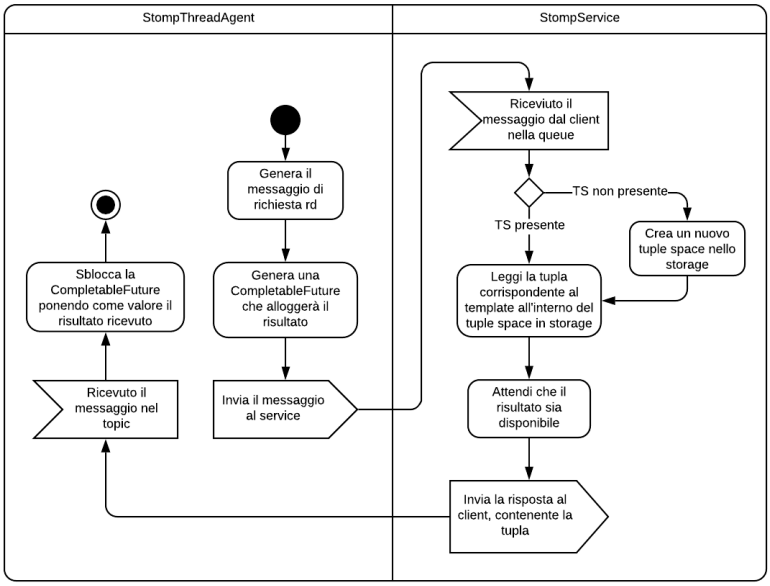


Figura 9: Diagramma delle attività per l'operazione RD.

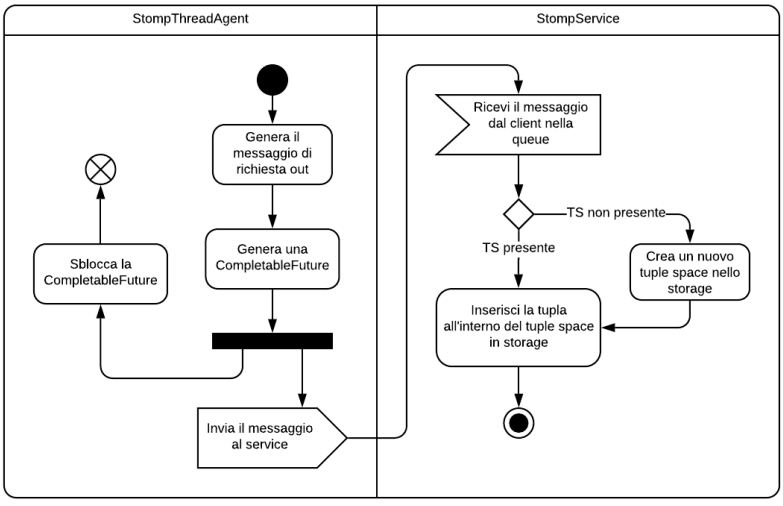


Figura 10: Diagramma delle attività per l'operazione OUT.

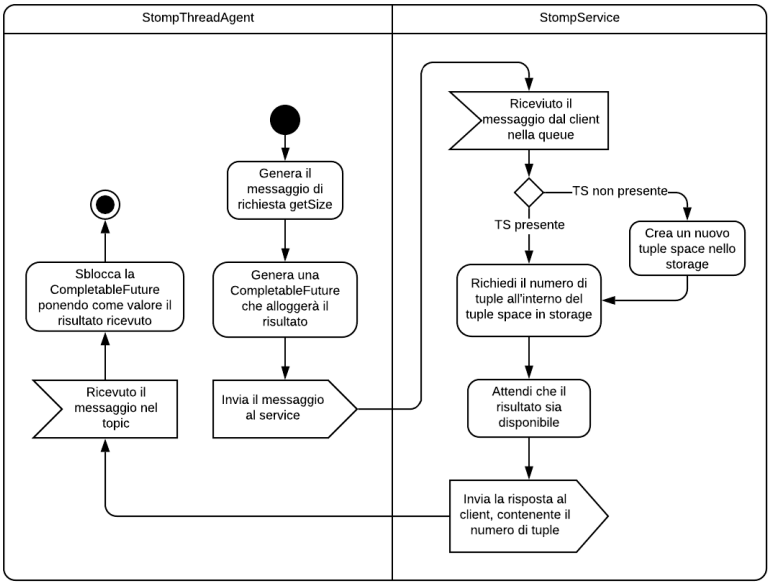


Figura 11: Diagramma delle attività per l'operazione GET SIZE.

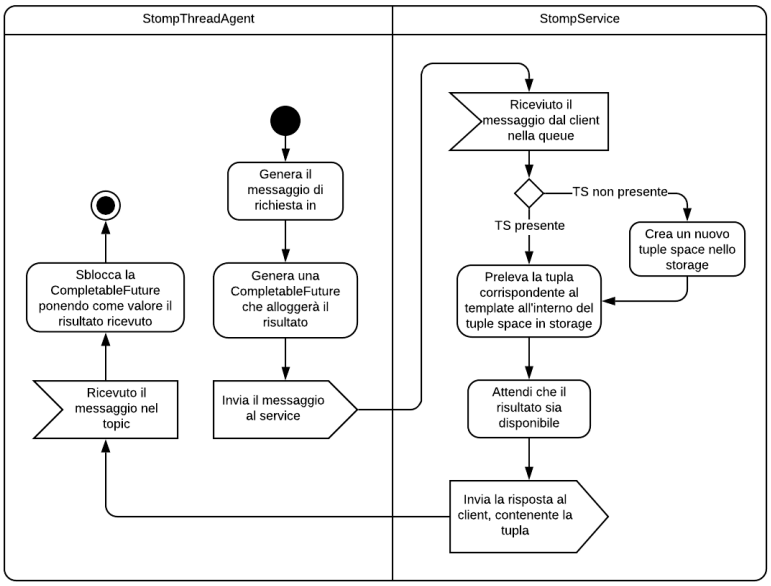


Figura 12: Diagramma delle attività per l'operazione IN.

3.3 Interazione

Il nostro progetto sfrutta per l'interazione tra le varie entità il protocollo *STOMP*, sfruttando la libreria *Vertx-stomp* per la creazione dei messaggi e il loro smistamento.

Ogni client ha la possibilità di connettersi allo *STOMP* Server (nel nostro caso incorporato nel modulo Broker) inviandogli nella sua porta un frame di tipo CONNECT, che può includere varie informazioni per l'autenticazione[2]. Ad autenticazione ottenuta, viene notificata l'avvenuta connessione al client con un frame CONNECTED. Da questo punto in poi le due entità possono comunicare liberamente. Nello specifico, il client può sottoscrivere topic o queue gestiti del server tramite dei frame SUBSCRIBE, oppure inviare dei messaggi in direzione di tali strutture tramite dei frame SEND. Il server provvederà a inoltrare il messaggio a tutti i client che hanno sottoscritto una determinata struttura, inviando loro dei frame MESSAGE, aventi come body quello del messaggio ricevuto. Infine, frame UNSUBSCRIBE e DISCONNECT notificano al server rispettivamente la volontà di non osservare più una struttura o dismettere una connessione.

Osservando la struttura del protocollo, si evince che, essendo tutte le entità di più alto livello basate su dei client *STOMP*, è necessario porre tutte le informazioni di interesse all'interno del body, evitando quando possibile personalizzazioni dei vari campi negli header.

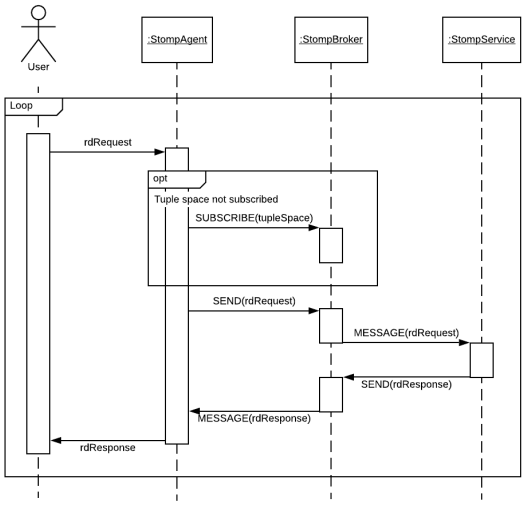


Figura 13: Diagramma di sequenza che descrive il flusso di frame *STOMP* fra i vari componenti per la gestione di una richiesta RD.

[2]Per scelta progettuale, il nostro progetto non implementa meccanismi avanzati di autenticazione. Ciò può essere previsto in un eventuale sviluppo futuro.

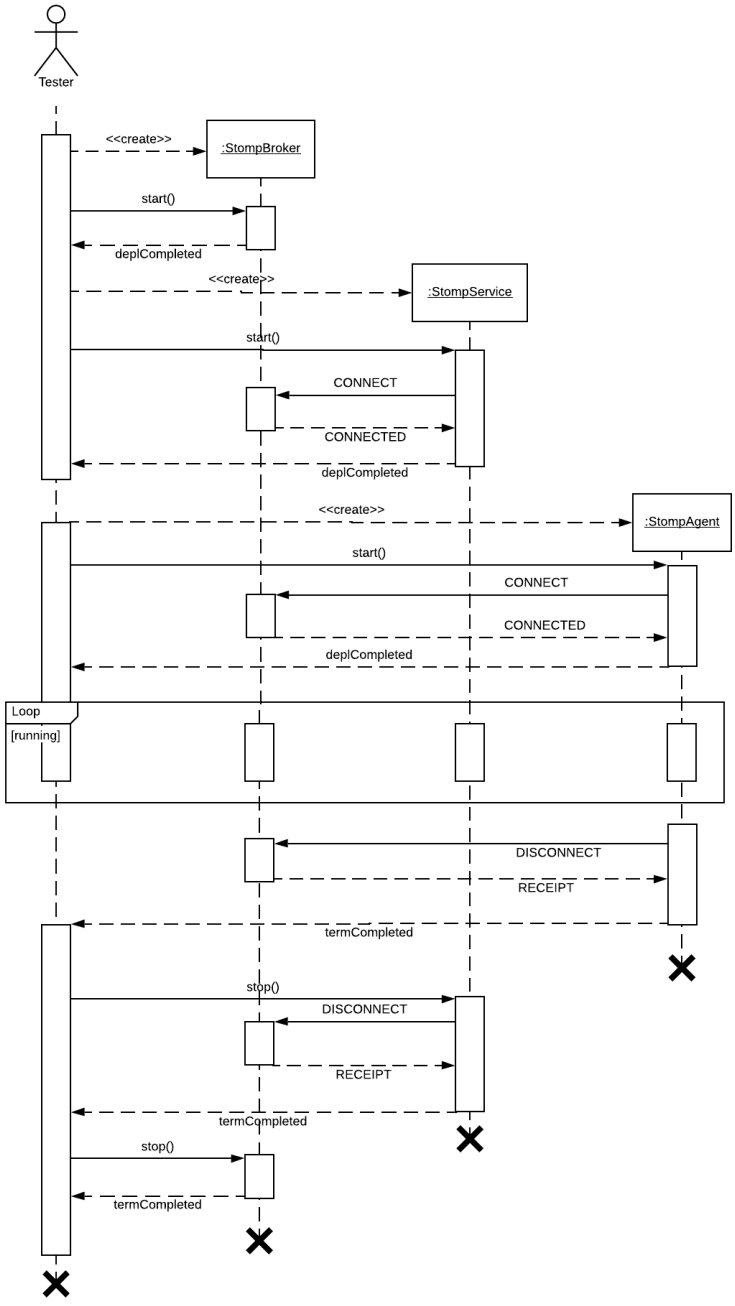


Figura 14: Diagramma di sequenza che descrive ordine e flusso di deployment e termination dei vari componenti, considerando l'intera simulazione controllata da un singolo utente tester.

3.3.1 Queue e topic utilizzati

Per la circolazione dei messaggi, il progetto sfrutta le seguenti strutture dati all'interno del broker:

- una queue denominata *StompService*, utilizzata dall'omonima entità per ricevere richieste *Linda*;
- un topic per ogni spazio di tuple, nei quali i vari agenti si porrano in ascolto delle risposte del Service alle loro richieste.

Ogni Agent andrà quindi a sottoscrivere tanti topic quanti sono gli spazi di tuple di suo interesse (ognuno identificato dal nome del tuple space), mentre il Service andrà a sottoscrivere soltanto una queue. Questo meccanismo consente al Service di ricevere richieste anche da Agent di cui non è a conoscenza. Esso non ha infatti la necessità di rimanere costantemente in contatto con ognuno di essi (e di essere a conoscenza dell'esistenza di ognuno), ma di rispondere solo nel momento in cui esso viene interpellato.

Quindi, una volta elaborata una risposta, il Service la invia sotto forma di frame *STOMP* in direzione del topic corrispondente.

Tutti gli agenti in ascolto riceveranno la risposta, ma questa verrà elaborata solo dall'agente interessato[3].

Nel prototipo iniziale, le strutture dati e i relativi meccanismi per lo scambio di messaggi erano stati concepiti in una maniera ben diversa. Si era pensato di dedicare un topic ad ogni spazio di tuple, ma di porre in ascolto sia il Service che i vari Agent nello stesso topic. In tal modo, sia *Linda* request che response sarebbero circolate nello stesso canale. Si era pensato inoltre di implementare una queue separata, utilizzata dal Service, solo allo scopo di ricevere notifiche dai vari Agent, riguardo alla volontà di sottoscrivere uno specifico tuple space. Tale struttura apportava però una notevole ed inutile difficoltà di gestione dell'intero sistema, ed è stata velocemente scartata in favore di quella precedentemente descritta.

[3] Questa particolarità è dovuta alla struttura stessa dal sistema e del modello di messaggistica publish-subscribe. Ciò potrebbe rappresentare un problema di sicurezza, ma in mancanza di soluzioni alternative, si è deciso di adottare questo modello.

4 Dettagli implementativi

Estensione del modello delle esercitazioni Molte caratteristiche del sistema in esame sono assimilabili a quanto fatto nelle esercitazioni di laboratorio relative allo sviluppo di API REST. Si è quindi deciso per continuità di riutilizzare alcuni moduli, estendendone opportunamente caratteristiche e funzionalità. Ad esempio il package *Linda*, che modella l'idea di tuple space e i concetti ad esso collegati, proviene da quello utilizzato nelle esercitazioni. Oppure la classe ThreadAgent, anch'essa ereditata da lì, ed opportunamente estesa per modellare il concetto di *StompThreadAgent*.

Si sono poi ripresi dai laboratori diversi idee e concetti (come l'idea di emulare storage dei tuple space, o il concetto di dato serializzabile), allo scopo facilitarne la comprensione, rendendo unica la chiave di lettura tra progetto e laboratorio.

Mancata implementazione della primitiva GET Per scelta implementativa, si è deciso di non implementare la primitiva GET, così come pensata all'interno delle esercitazioni di laboratorio. Questo in quanto tale primitiva può prevedere come risposta l'invio di tutte le tuple all'interno del tuple space (nel caso nella richiesta non venisse specificato il template). Modellare un payload tale da dare la possibilità di accorpare molteplici tuple avrebbe apportato un'elevata complessità, che non saremmo riusciti a gestire nei tempi previsti.

Mancata gestione della persistenza Sempre per una scelta implementativa, si è deciso di non porre in atto meccanismi di persistenza sugli spazi di tuple. In un'applicazione reale, la memorizzazione persistente dei dati rappresenta un aspetto rilevante; avendo però il progetto in esame come focus principale l'interazione tra i componenti, si è ritenuto l'aspetto come secondario, e si è deciso di simulare lo storage tramite il modulo *TextualSpaceStorage* visto in laboratorio.

Documentazione e leggibilità Si è cercato di mantenere il quanto più possibile il codice autoesplicativo, tramite l'utilizzo di metodi e attributi con nomi significativi, seguendo i principali pattern di programmazione e modularizzando adeguatamente le entità.

I passaggi più delicati vengono inoltre trattati all'interno del codice sorgente stesso, tramite un sistema di commenti pulito ed efficace.

Design pattern Allo scopo di rendere il codice pulito ed estendibile, sono stati applicati i i principali design pattern della programmazione Java. Ampiamente utilizzati sono ad esempio Factory e Strategy, così da separare implementazione e contratto, e rendere in generale il codice più leggibile e modulare.

5 Autovalutazione/Validazione

Per validare l'operato del progetto, si è operato in due principali step:

- Durante lo sviluppo, si sono sfruttati alcuni unit test per valutare la correttezza di determinate funzioni chiave. Su di queste si è posta particolare attenzione, procedendo con un approccio di tipo TDD.
- Al termine di questo, si è proceduto con l'esecuzione di test di integrazione, definiti precedentemente rispetto alla fase di stesura del codice. Tali test sono stati pensati in maniera tale da verificare le proprietà salienti del sistema, e il rispetto dei requisiti individuati in fase di progettazione.

Si è inoltre verificato formalmente il livello di coverage ottenuto complessivamente dai test, sfruttando lo strumento integrato all'interno di intelliJ. Prendendo come riferimento tutti i test sviluppati, si è ottenuto un coverage delle classi dell'88% (22/25), un coverage dei metodi dell'83% (157/187) e un coverage delle linee di codice dell'81% (531/648).

5.1 Unit test

Come precedentemente accennato, le pratiche TDD sono state applicate solo per particolari funzioni di importanza chiave. I test di unità riportati nell'omonimo package coprono il codice relativo alla classe *StompMessageData* e verificano il corretto funzionamento di determinate funzionalità di *Vertx*.

TestStompMessageData Verifica le caratteristiche critiche della classe *StompMessageData*, quali ad esempio l'univocità degli ID dei messaggi (per una mole ragionevole di questi), dei metodi di interesse, e il corretto funzionamento del processo di serializzazione e deserisalizzazione dei campi dell'oggetto.

TextVertxComponents Dimostra il corretto funzionamento dei componenti *Vertx* utilizzati, quali *StompClient* e *StompServer*. È presente inoltre un test che esegue un semplice ping-pong tra *StompService* e un generico agent, a garanzia della correttezza del meccanismo di scambio, e anche allo scopo di dimostrare in maniera schematica in cosa consiste tale meccanismo.

TestTextualSpace Ripresa dalle esercitazioni tale e quale, allo scopo di dimostrare sia il corretto funzionamento della classe *TextualSpace*, utilizzata per emulare lo storage, sia la sua invarianza rispetto alla soluzione di partenza. Non essendo nel contesto del nostro progetto *TextualSpace* un componente principale, seppur importante, è stato posto insieme agli altri unit test.

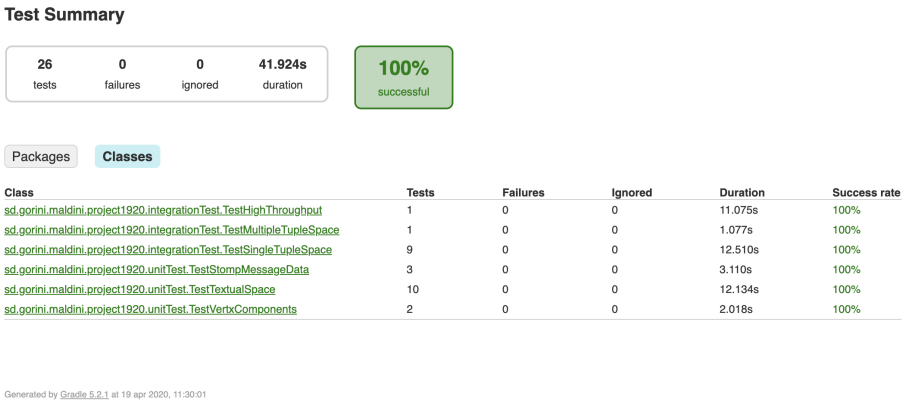
5.2 Integration test

Per verificare il corretto funzionamento dell'intero sistema, sono stati predisposti dei test di integrazione, che coinvolgono tutti i moduli sviluppati. Ogni classe di test ha lo scopo di verificare un differente insieme di caratteristiche di interesse.

TestSingleTupleSpace Riprende i test effettuati all'interno di *AbstractTestTextualSpace* nelle esercitazioni, applicandoli al progetto corrente. Lo scopo è quello di dimostrare la corretta implementazione delle primitive *Linda*, e le loro caratteristiche, oltre ad avere un benchmark in comune con l'API HTTP sviluppata in laboratorio. Unica differenza sta nel metodo di test getAll(), non portato avanti qua, in quanto la primitiva GET non è stata implementata nel progetto.

TestMultipleTupleSpace Dimostra il corretto funzionamento del sistema in presenza di più tuple space in contemporanea. Viene eseguito di fatto un singolo test, che coinvolge però un caso limite per quanto riguarda il numero di tuple space utilizzati allo stesso tempo.

TestHighThroughput Dimostra il throughput elevato che il sistema può garantire avendo una connessione bidirezionale sempre aperta tra le parti, anche senza le ottimizzazioni tipiche dei broker più avanzati. Viene eseguito di fatto un singolo test, che coinvolge però un caso limite per quanto riguarda il numero di messaggi scambiati tra Service e Agent durante la simulazione.



Test Summary

26 tests	0 failures	0 ignored	41.924s duration	100% successful

Packages **Classes**

Class	Tests	Failures	Ignored	Duration	Success rate
sd.gorini.maldini.project1920.integrationTest.TestHighThroughput	1	0	0	11.075s	100%
sd.gorini.maldini.project1920.integrationTest.TestMultipleTupleSpace	1	0	0	1.077s	100%
sd.gorini.maldini.project1920.integrationTest.TestSingleTupleSpace	9	0	0	12.510s	100%
sd.gorini.maldini.project1920.unitTest.TestStompMessageData	3	0	0	3.110s	100%
sd.gorini.maldini.project1920.unitTest.TestTextualSpace	10	0	0	12.134s	100%
sd.gorini.maldini.project1920.unitTest.TestVertxComponents	2	0	0	2.018s	100%

Generated by Gradle 5.2.1 at 19 apr 2020, 11:30:01

Figura 15: Risultati dei test, raccolti e mostrati dallo strumento integrato in IntelliJ.

6 Istruzioni per il deployment

Per facilitare il deployment in una macchina vergine, è stato utilizzato il build tool *Gradle*. Per eseguire il programma occorre seguire i seguenti passi, seguendo l'ordine indicato:

1. Scaricare una JDK con versione $\geq$ 1.8.
2. Clonare la repository GitLab all'indirizzo:

   https://gitlab.com/pika-lab/courses/ds/projects/sd-project-gorini-maldini-1920

3. Posizionarsi all'interno della cartella di progetto.
4. Per compilare il progetto ed eseguire i test, digitare il comando

   `.\gradlew build` (in Windows)

   oppure

   `./gradlew build` (in Linux o MacOS)

5. Per eseguire soltanto i test in un secondo momento, digitare il comando

   `.\gradlew test` (in Windows)

   oppure

   `./gradlew test` (in Linux o MacOS)

7 Esempi di utilizzo

Il sistema può essere utilizzato, sia eseguendo ogni componente all'interno della stessa macchina (nel caso li si voglia testare), sia eseguendo ognuno di questi su macchine diverse (caso di utilizzo reale). In tal caso, sarà necessario indicare propriamente ai componenti agenti come client, l'indirizzo IP e la porta utilizzati dal server, per la fase di connessione iniziale.

In un'implementazione reale, è ragionevole che *StompService* e *StompBroker*, che assieme rappresentano il backend del sistema, risiedano nella stessa macchina, seppur ciò non è mandatorio.

I test di integrazione approfonditi in precedenza, coinvolgendo tutti i componenti principali, rappresentano degli esempi completi di utilizzo, ai quali è possibile far riferimento. Seppur è vero che questi vengono eseguiti in locale su una stessa macchina, mostrano in maniera coincisa come utilizzare i componenti sviluppati.

7.1 Avvio dei componenti backend

Il componente *StompBroker* deve essere avviato prima di qualunque altro nodo client, tramite il metodo *start()*. Questo in quanto tutti gli altri nodi del sistema sono di fatto dei client, interagiscono esclusivamente con il primo (non direttamente tra di loro). Quindi, è necessario avviare il componente *StompService*, che assieme al broker costituisce il backend del sistema.

7.2 Avvio degli agenti

Per implementare agenti, il progetto mette a disposizione il modulo *StompThreadAgent*. Questo è di fatto una classe astratta, che deve essere implementata per poter essere correttamente utilizzata. Implementare tale modulo significa andare a definire un nome per il thread (opzionale) e andare a definire il comportamento del loop di esecuzione. Al suo interno, sarà possibile invocare *Linda* operation tramite i metodi *rd(template)*, *out(tuple)*, *in(template)*, *getSize()*, invocati su dei tuple space, accessibili tramite *getTupleSpaceByName(tsName)*, e decidere quando interrompere il flusso di esecuzione tramite il metodo *stop()*.

Quindi, sarà possibile avviare l'agente tramite il metodo *start()*.

8 Conclusioni

Il progetto finale rispetta a pieno gli obiettivi e i requisiti prefissati inizialmente:

- Si è dimostrato come utilizzare *STOMP* in un sistema ad agenti, e quali sono i vantaggi derivanti dall'utilizzo di tale protocollo.
- Si è lavorato andando a creare vari strati software sopra il protocollo *STOMP*, implementando di fatto le principali direttive *Linda* tramite i componenti *StompThreadAgent*, *RemoteTextualSpaceImpl* e *StompService*.
- Si è lavorato in parte anche a livello più basso, andando a implementare un semplice broker nel modulo *StompBroker*, e adattando le comunicazioni secondo le best practices del protocollo.
- I test di integrazione compongono una garanzia della correttezza dell'elaborato. Si è notato grazie a questi che i moduli che compongono il sistema sono stati creati tutti correttamente e che le invarianti su di loro son state rispettate.

8.1 Lavori futuri

Seppur l'elaborato rispetta gli obiettivi preposti, risulta possibile ampliare il sistema tramite funzionalità aggiuntive, o piene rielaborazioni dello stesso, quali:

- *Autenticazione*: per scelta progettuale, non sono stati predisposti meccanismi per il controllo di accesso da parte degli utenti, sia al broker stesso che ai vari tuple space. Risulterebbe utile implementare un sistema in grado di identificare gli agenti che intendono collegarsi al sistema, autorizzando gli stessi ad interagire con quest'ultimo, o negando loro l'accesso. Ad esempio, sfruttando semplicemente gli header user e passcode previsti dal protocollo nel frame CONNECT, e/o sfruttando la classe AuthProvider prevista da *Vertx-stomp*.
- *Primitiva GET*: per motivi di tempistiche e di eccessiva complessità, si è deciso di non implementare la primitiva GET, così come prevista durante le esercitazioni di laboratorio. Come descritto nel capitolo 4, implementare tale operazione avrebbe indicato una modellazione del payload dei messaggi *STOMP* alquanto complicata e non ci avrebbe permesso di concludere il progetto con il tempo a disposizione.
- *Sicurezza*: altro punto che è possibile approfondire e migliorare, riguarda la sicurezza informatica del sistema. Sarebbe possibile ad esempio utilizzare tecniche di crittografia per rendere più affidabile le comunicazioni tra i vari nodi del sistema.
- *Re-implementazione su web browser*: l'elaborato risulta essere una base teorica e pratica per l'implementazione di *Linda* su web browser, riutilizzando ad esempio la struttura, adattandola ad un ambiente web tramite *StompOverWebsocket*.

8.2 Cosa abbiamo imparato

L'elaborato è stato un importante banco di prova. In primo luogo ci ha permesso di tradurre le conoscenze teoriche di *Linda* e dei sistemi ad agenti in un caso reale di utilizzo. Abbiamo capito quali sono i punti di forza e quali le eventuali criticità di un'implementazione di un ambiente distribuito. Ci ha permesso inoltre di approfondire le conoscenze di *STOMP* e, in generale, dei protocolli che implementano il modello publish-subscribe.

In secondo luogo, provenendo entrambi i componenti del gruppo di lavoro da un'altra facoltà, dove la programmazione ad oggetti non è stata approfondita in dettaglio al livello di questo Corso di Laurea, ci siamo imbattuti in notevoli difficoltà iniziali, dal punto di vista pratico.

Abbiamo approfondito strumenti non presenti nel nostro pregresso bagaglio di conoscenze, quali *jUnit* o *Gradle*, e si è cercato di innalzare la qualità del codice prodotto, rispetto ai nostri standard.

Per la prima volta, abbiamo adottato un modello più rigido per il testing e la validazione, tramite l'utilizzo estensivo di test, e in parte del processo TDD, cogliendone le funzionalità e i vantaggi che esso comporta.

Abbiamo imparato a rispettare le tempistiche per la progettazione di un elaborato, a suddividere la mole di lavoro e ad assegnarne una parte a ciascuno dei componenti.

Possiamo concludere quindi affermando che lavorare su questo progetto è stato importante e formativo per il nostro futuro in questo settore.

Riferimenti bibliografici

[1] A. Omicini and F. Zambonelli,
Coordination for Internet Application Development
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: http://link.springer.com/10.1023/A:1010060322135

[2] Red Hat Consulting team, *Cos'è il middleware*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://www.redhat.com/it/topics/middleware/what-is-middleware

[3] E. Curry, *Message-Oriented Middleware*
[Online] (2004). [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://pdfs.semanticscholar.org/98e1/90fd2ed9e15514f7f5d5ea3dbd8a eb382a9c.pdf

[4] Wikipedia.org, *Message-Oriented Middleware*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://en.wikipedia.org/wiki/Message_broker/

[5] *STOMP* Dev. Team, *STOMP 1.2 specification*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: http://stomp.github.io/stomp-specification-1.2.html

[6] J. Mesnil *stomp-websocket JS module*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://github.com/jmesnil/stomp-websocket

[7] *Vertx* Dev. Team, *Vertx documentation*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://vertx.io/docs/

[8] *Vertx* Dev. Team, *Vertx-stomp documentation*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://vertx.io/docs/vertx-stomp/java/

[9] FasterXML LLC., *Jackson*
[Online]. [Ultimo Accesso: 19 Aprile 2020].
Disponibile: https://github.com/FasterXML/jackson/