

Linda on CoAP

Michael Camporesi

`michael.camporesi@studio.unibo.it`

Davide Bondi

`davide.bondi3@studio.unibo.it`

20 Settembre 2021

L'obiettivo di questo progetto è la realizzazione di un'interfaccia che modelli il linguaggio di coordinazione Linda tramite il protocollo di comunicazione CoAP¹. Quest'ultimo realizza un sottoinsieme di operazioni ReST² tramite HTTP, ottimizzate per applicazioni M2M³, con le quali poter operare con diverse tipologie di device IoT⁴ aventi varia natura e capacità di prestazioni per lo più ridotte. CoAP possiede un'architettura centralizzata di tipo client-server, i quali comunicano in modo request-response tramite scambio di messaggi asincrono via UDP. Lo stato di un'applicazione CoAP risiede nelle resource, ed ognuna di esse viene identificata da un proprio URI.

Un client di CoAP può eseguire le seguenti operazioni (o metodi), inviando un messaggio verso una destination rappresentata da un URI di una resource.

- GET: richiede di ottenere l'informazione rappresentata da una resource.
- POST: richiede che venga elaborato il contenuto racchiuso nel messaggio. L'operazione effettiva che viene eseguita, è determinata dalla resource e dal suo origin server. Di solito causa la creazione di una nuova resource o l'aggiornamento della destination resource.
- PUT: richiede che la destination resource venga aggiornata o creata in base al contenuto racchiuso nel messaggio.
- DELETE: richiede di eliminare una resource.

Per quanto riguarda Linda, è tuple-based e ricade nella categoria dei modelli architetturali blackboard. Nello specifico, in Linda vi sono agenti che interagiscono con uno o più shared data space, chiamati tuple space.

¹Constrained Application Protocol

²Representational State Transfer

³Machine to Machine, termine che indica comunicazioni dirette (wired o wireless) tra 2 o più dispositivi

⁴Internet of Things

Un agente in Linda può eseguire le seguenti primitive di base.

- Read (rd): permette di leggere una tupla, che rispetti il match con il template inviato, da uno spazio di tuple.
- Take (in): permette di leggere e rimuovere una tupla, che rispetti il match con il template inviato, da uno spazio di tuple.
- Write (out): permette di scrivere una tupla in uno spazio di tuple.

Considerando queste specifiche, il sistema sarà composto dai seguenti elementi architetturali (figura 1).

- Components: client e server.
- Connectors: Linda e CoAP.
- Data: tuple e template.

Il progetto racchiuderà molteplici spazi di tuple dentro al server, invierà le operazioni di Linda nel payload della POST e ne gestirà la semantica sospensiva, infine introdurrà una semantica per il multicast.

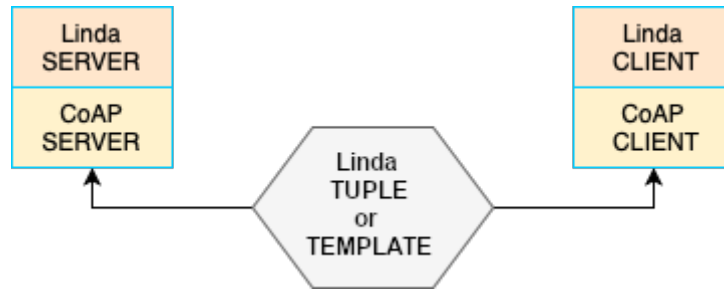


Figura 1: architettura di progetto: layer di Linda e layer di CoAP.

1 Obiettivi e Requisiti

In questa sezione vengono definiti gli obiettivi e i requisiti, stabiliti prima di cominciare la realizzazione del progetto.

1.1 Obiettivi

- Implementazione di un'interfaccia o libreria, con la quale poter programmare utilizzando il linguaggio di coordinazione Linda, mediante il protocollo di comunicazione CoAP.
- Realizzazione del LindaServer, che dovrà gestire gli spazi di tuple e le richieste che arriveranno dai diversi LindaClient. Tale server dovrà basarsi sul CoAP server ed utilizzare le resource per la comunicazione con i client.
- Realizzazione del LindaClient, che dovrà eseguire le operazioni di base di Linda sugli spazi di tuple gestiti da uno o più LindaServer. Tale client dovrà basarsi e sfruttare le operazioni disponibili al CoAP client.
- Realizzazione di un'interpretazione dei messaggi scambiati tra client-server.
- Nel caso la quantità di lavoro dovesse risultare insufficiente, si prevede anche l'implementazione di altre funzionalità.
Per quanto riguarda CoAP, si prevede di sfruttare le seguenti feature:
 - mandare messaggi in Multicast
 - Observation e Discovery delle resource (non implementato)
 - una forma di Content Negotiation (non implementato)Per quanto riguarda Linda, si prevede di implementare le operazioni:
 - predicative
 - bulk (non implementato)
 - duali (non implementato)
- Realizzazione di un'applicazione che permetta di dimostrare un esempio di utilizzo reale della nostra libreria. Tale applicazione dovrà fare uso di più server e client che dovranno scambiarsi messaggi tra loro dimostrando tutte le potenzialità della libreria che andremo a realizzare.

1.2 Requisiti

L'obiettivo è realizzare un sistema che permetta lo scambio di tuple e template tra più server e più client remoti. Tale sistema dovrà essere basato sul protocollo CoAP, per questo è stata scelta un'implementazione già esistente di CoAP, ovvero una libreria chiamata Californium, scritta in linguaggio Java.

I diversi server dovranno salvare al loro interno gli spazi di tuple, che saranno accessibili ai client remoti, i quali potranno eseguire su di essi le operazioni di Linda.

Vorremmo garantire al programmatore che utilizzerà la nostra libreria, la possibilità di scegliere tra classi prestabilite, lasciando però la completa libertà di personalizzare e configurare in autonomia i diversi aspetti della comunicazione CoAP tra client e server.

1.3 Risultati Attesi

Come principale obiettivo abbiamo quello di realizzare l'interfaccia in maniera completa e chiara. La libreria dovrà essere facilmente utilizzabile da altre persone, ci aspettiamo quindi che sia ben strutturata ed organizzata.

Dovremo realizzare in maniera completa e funzionante sia client che server, insieme ad un'interfaccia per tuple e template, accompagnata da un paio di elementi di esempio, cosicché il programmatore possa creare le proprie implementazioni.

Un altro obiettivo è la realizzazione di un esempio di utilizzo della libreria per un ipotetico caso reale. Questo dovrà far uso sia di client diversi sia di vari server, possibilmente con tanti spazi di tuple e tante tuple diverse.

Per quanto riguarda le feature opzionali, procederemo ad implementare per primo il multicast, il quale permette lo scambio di messaggi a tutti i server in ascolto contemporaneamente. Cercheremo quindi di integrare tale logica con le operazioni di Linda.

Poi si passerà alla Observation e Discovery delle resource, in particolare degli spazi di tuple presenti all'interno dei vari server.

L'ultima funzionalità opzionale che pensiamo di integrare è quella di permettere le operazioni di Linda: predicative, bulk e duali.

1.4 Scenari

L'utente avrà a disposizione 2 prospettive diverse per interagire con la libreria.

Nel primo caso, potrà creare un'applicazione Linda con approccio client-server tramite le classi di default.

Nel secondo caso, potrà utilizzare le componenti client-server della libreria integrandole a piacere con uno scambio di messaggi del protocollo CoAP configurato da lui stesso, in maniera da garantire trasparenza e un'elevata flessibilità di personalizzazione dell'implementazione.

1.5 Politiche di Autovalutazione

Cercheremo di scrivere codice di buona qualità servendoci delle nostre esperienze di studi e di lavoro. La priorità sarà sempre l'organizzazione e la pulizia del codice, uniti ad una buona documentazione che renda tutto il più fruibile possibile.

La libreria dovrà quindi essere semplice da utilizzare e da testare. A questo proposito, i test dovranno essere chiari e leggibili così che si capisca subito il loro scopo e da essi si possano trarre esempi di implementazione.

Anche l'applicazione finale dovrà essere semplice da usare, ed allo stesso tempo sufficientemente chiara e dettagliata in modo tale che anch'essa possa essere utilizzata da terzi per capire il funzionamento delle nostre interfacce e della libreria in generale.

Le operazioni di Linda dovranno essere opportunamente incapsulate.

Ci serviremo anche di test JUnit per validare il nostro codice. Cercheremo di garantire la correttezza di tali test poiché essi determineranno poi la correttezza globale dell'intera libreria. I test finali verranno eseguiti servendosi anche dell'applicazione finale.

2 Requirements Analysis

2.1 Requisiti Funzionali, Modelli e Paradigmi

Il formalismo di Linda può essere mappato facilmente con l'architettura client-server di CoAP: un client corrisponde ad un agente che interagisce con gli spazi di tuple contenuti in un server.

Deve essere precisato che, nell'interfaccia che si andrà a sviluppare, non verrà considerato il concetto di agente, poiché questo è posto ad un livello di astrazione maggiore rispetto all'obiettivo del progetto, in cui si propone di definire una semantica delle operazioni di Linda tramite CoAP.

Nella sezione sottostante verranno analizzati i problemi di coesione tra la semantica di Linda e il protocollo CoAP, e in seguito verranno descritti i requisiti concreti che l'applicazione dovrà rispettare per risolvere i problemi di coesione.

2.1.1 Dominio del Problema: Problemi di Coesione, Linda VS CoAP

1. Il primo problema che emerge è trovare un modo significativo per rappresentare e gestire i tuple space all'interno del server CoAP.
2. Osservando la figura 1, si intuisce che le operazioni di Linda devono viaggiare nei messaggi CoAP. Un altro dettaglio che emerge dalla figura è che: la soluzione ideale del sistema dovrebbe fare in modo di separare quanto più possibile, il layer di Linda dal layer di CoAP.
3. Le operazioni di Linda read e take non ammettono il fallimento, di conseguenza hanno semantica sospensiva, cioè devono bloccarsi fintanto che non è possibile leggere una tupla. CoAP invia i messaggi in modo asincrono, ma nonostante ciò, impone una scadenza nella sessione o exchange di messaggi per questioni di sicurezza. C'è quindi una forte incompatibilità con la sospensione a lungo tempo del risultato di read o take, e si deve gestire questo problema.
4. C'è un'altra incompatibilità molto importante che riguarda la semantica bloccante di read e take, e coinvolge il multicast.

Innanzitutto si assume che la read o la take effettuate in multicast possano ricevere come risultato una sola tupla, poiché per la loro definizione formale esse leggono una tupla, pertanto per coerenza si vuole mantenere questa regola.

Non sarebbe sbagliato aspettare di ricevere i risultati da tutti i server, tuttavia lo si ritiene piuttosto controintuitivo, inoltre è difficile avere la certezza di quanti risultati attendere perché un server potrebbe aggiungersi o rimuoversi (o disconnettersi) dal gruppo multicast in qualsiasi momento. Pertanto il coinvolgimento di molteplici server rende questa situazione facilmente soggetta ad errori, la cui gestione è tutt'altro che banale.

Nonostante l'aver imposto alla read e take di ricevere come risultato una sola tupla, l'eventuale sospensione del risultato di una read o take effettuata in multicast, genera una situazione estremamente caotica. Quando un'operazione bloccante inviata in multicast non ottiene risultato immediato, richiede di essere salvata all'interno di tutti i server appartenenti al gruppo multicast fino a quando non viene soddisfatta. Questo potrebbe causare l'invio dei risultati sospesi mentre il client ha già concluso l'operazione, e nel caso peggiore, il client potrebbe aver iniziato un'altra operazione identica alla precedente, la quale viene soddisfatta da un risultato richiesto dall'operazione precedente. Ciò è ritenuto sbagliato e particolarmente difficile da gestire.

Un'altra situazione confusionale che potrebbe verificarsi in ambito di operazioni sospensive effettuate in multicast, è descritta nell'esempio sottostante, in cui si assume che i template e le tuple utilizzate nelle operazioni siano correlate e garantiscano il match tra loro.

- a) Un client A inoltra un'operazione take verso il gruppo multicast che include i server X e Y. Il server X contiene una tupla e la invia al client A, quindi A conclude l'operazione, tuttavia il server Y non contiene alcuna tupla, quindi Y memorizza l'operazione take richiesta dal client A.
- b) Un client B inoltra un'operazione take verso il gruppo multicast che include i server X e Y. Entrambi i server non contengono alcuna tupla, quindi memorizzano l'operazione take richiesta dal client B.
- c) Un client C inoltra un'operazione write in unicast verso il server Y, ora quest'ultimo può soddisfare un'operazione take, e soddisferà la take richiesta del client A, perché è avvenuta prima di quella del client B. Così facendo tuttavia, invierà inutilmente il risultato verso A perché esso ha già concluso l'operazione, e B rimarrà senza risultato.

In seguito a queste considerazioni, si è stabilito che il multicast non deve eseguire operazioni bloccanti, bensì solamente operazioni che portano un risultato immediato, come la write. Sempre per questo motivo, si è deciso di inserire nel progetto le operazioni read e take in versione predicative, ed eseguirle in multicast perché queste modellino il successo (cioè il risultato) ed il fallimento immediati.

2.1.2 Dominio della Soluzione: Requisiti Concreti

1.
 - Chiaramente un server CoAP deve poter contenere molteplici spazi di tuple, e l'unico modo per farlo è mappare lo spazio di tuple come un tipo specifico di resource.
 - Si dovrà elaborare un sistema di naming efficace che possa permettere anche la presenza di altri tipi di resource, poiché avere un sistema IoT che dispone solo di tuple space può essere limitativo.

2.
 - Ovviamente, le operazioni di Linda devono essere inviate nei messaggi, ma il punto fondamentale è che il layer di astrazione di Linda deve essere separato quanto più possibile dal layer di astrazione di CoAP.

Per applicare questa separazione, si è deciso che le operazioni di Linda devono viaggiare solo nel payload. Si dovrà rigorosamente evitare di utilizzare le opzioni e i metadati dei messaggi di CoAP per rappresentare il formalismo di Linda, in questo modo non vi sarà alcuna intersezione tra i due layer, dal momento che saranno separati e ben distinti. Colloquialmente si può affermare che Linda sarà "appoggiato" sopra CoAP, pertanto quest'ultimo può essere configurato in maniera totalmente indipendentemente da Linda.

Di conseguenza, per mappare le operazioni di Linda si dovranno scegliere metodi CoAP che ammettano un payload nel loro messaggio (ad esempio, la GET non ammette payload), e per semplicità, si è deciso di inviare tutte le operazioni di Linda tramite un unico metodo: POST.

Gli unici metodi che apparentemente potrebbero sembrare idonei sono POST e PUT, tuttavia PUT è nettamente sbagliato. Il problema fondamentale di coesione tra PUT e le operazioni di Linda è il seguente: le operazioni read e take leggono una tupla, la quale viene fornita in modo non deterministico dallo spazio di tuple (cioè dal server), e questo risultato non deterministico coinvolge una "computazione" effettuata dal server, che per quanto possa essere semplice, è comunque più coesa con la semantica di POST rispetto alla semantica di PUT. POST richiede un'elaborazione server side, mentre PUT richiede un'elaborazione server side in base al contenuto racchiuso nel suo messaggio. Formalmente questo comporta che PUT sia idempotente, mentre POST no, allo stesso modo il risultato non deterministico di read e take le rende operazioni non idempotenti.

- Dato che le operazioni di Linda devono viaggiare nel payload, servirà un meccanismo per interpretarlo, sia nel client sia nel server.
3.
 - Per gestire l'incompatibilità tra la semantica bloccante di Linda e la scadenza della sessione di scambio messaggi di CoAP, si è giunti alla soluzione di inserire un CoAP server dentro un `LindaClient`, in modo da poter aspettare di ricevere l'eventuale risultato sospeso per un tempo illimitato.
(Disporre di un client e server su uno stesso nodo, è una situazione consueta in scenari IoT implementati con CoAP⁵.)
Allo stesso modo, il `LindaServer` memorizzerà le operazioni di Linda sospese, e non appena riceverà una tupla per soddisfarle, utilizzerà un CoAP client contenuto al suo interno per inviare i risultati verso il client bloccati.

⁵The interaction model of CoAP is similar to the client/server model of HTTP. However, machine-to-machine interactions typically result in a CoAP implementation acting in both client and server roles [2]

(Alternativamente a questa soluzione, si è giunti ad un'altra conclusione, che poi è stata scartata: qualora si fosse verificata la scadenza della sessione di scambio messaggi CoAP, si sarebbe potuto fare in modo di riattivare una nuova sessione dal Client, ma questo avrebbe comportato un traffico di messaggi superiore e totalmente inutile.)

Anche in questo caso, il layer di astrazione di Linda sarà ben distinto da quello di CoAP: i client e server di CoAP saranno incapsulati dentro al `LindaClient` e al `LindaServer`, e anche questa volta, gli elementi di CoAP saranno configurabili indipendentemente dagli elementi Linda.

- Per gestire questa situazione, il payload dei messaggi dovrà essere più complesso di quanto stabilito fino ad ora. Non è più sufficiente inviare solamente le operazioni di Linda. Vi è l'esigenza di inserire un ulteriore metadato, e cioè l'URI di destinazione posto nel `LindaClient`, sul quale ricevere il risultato bloccato.

4. • Implementare le seguenti operazioni in multicast: `write`; `try read`; `try take` .

2.2 Requisiti Non Funzionali

Il primo dei requisiti è quello di realizzare una documentazione completa e chiara, in modo da facilitare l'utilizzo della libreria da parte di altre persone.

L'interfaccia da implementare dovrà nascondere il funzionamento interno di CoAP all'utente finale, in modo da poter utilizzare Linda anche senza conoscere a fondo CoAP.

Il sistema dovrà garantire quanto più possibile la personalizzazione del protocollo CoAP, come ad esempio, poter garantire l'uso di diversi tipi di messaggi, in modo tale che si possa scegliere quello che più si preferisce, in termini di prestazioni o affidabilità forniti dalle svariate opzioni di CoAP.

Il design del progetto dovrà produrre un'architettura, la quale dovrà essere implementabile sopra altre librerie che implementano il protocollo CoAP.

2.3 Requisiti Tecnologici

La comunicazione tra client e server dovrà essere costruita utilizzando il protocollo di comunicazione CoAP.

Per garantire sia la portabilità del codice e che la possibilità di utilizzo della nostra libreria dal maggior numero di dispositivi possibili, abbiamo scelto di realizzare il progetto utilizzando il linguaggio Java.

Abbiamo scelto inoltre di utilizzare un'implementazione esistente di CoAP per il linguaggio Java, chiamata Californium.

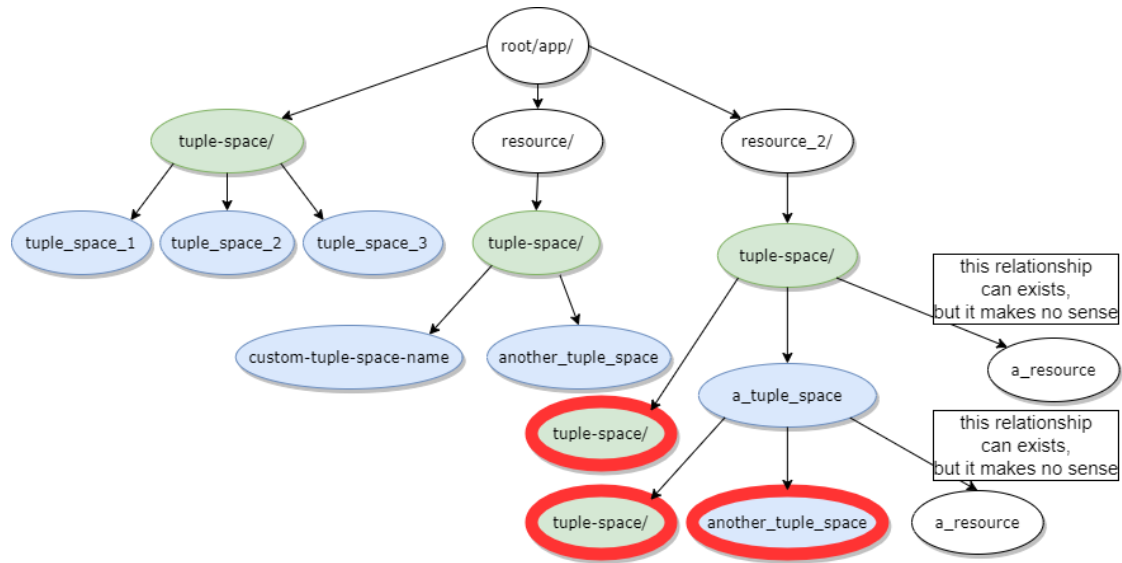


Figura 2: il naming graph del namespace delle resource all'interno del CoAP server.
 Le resource bianche sono resource di CoAP programmabili in qualsiasi modo.
 Le resource verdi identificano le loro child resource come spazi di tuple.
 Le resource azzurre sono spazi di tuple.
 Le resource con il bordo rosso non sono ammesse nel sistema.

3 Design

La figura 2 indica la soluzione al requisito numero 1 nell'elenco della sezione precedente. Uno spazio di tuple può avere un qualsiasi nome personalizzato e può essere inserito in un qualsiasi livello di annidazione di resource, con l'unica condizione di avere la risorsa genitore chiamata *tuple-space*, la quale deve essere inserita prima di uno spazio di tuple per identificarlo nel sistema. In questo modo, gli spazi di tuple sono ben distinti e incapsulati: possono coesistere con altri tipi di CoAP resource.

Il progetto LindaOnCoAP è costituito da 3 progetti: Client, Server e Communication.

3.1 Communication

L'esigenza di questo progetto nasce dal fatto che client e server devono disporre degli stessi elementi per comunicare, pertanto questo progetto deve essere incluso come dipendenza negli altri due.

3.1.1 Structure

Gli elementi condivisi tra i componenti client e server sono i seguenti.

1. Il formalismo riguardo il namespace delle resource, rappresentato in figura 2.

Per farlo rispettare, è stata creata una classe di utility apposita: **TSPathVerifier**. (TS è l'abbreviazione di *tuple space*.)

2. Il meccanismo per generare e tradurre il payload dei messaggi.

PayloadConverter è l'interfaccia che rappresenta la logica per farlo, mentre **PayloadConverterImpl** è una sua implementazione concreta.

Se è necessario gestire un contesto in cui vi sono vincoli riguardo il payload, ad esempio, se la lunghezza del payload deve essere particolarmente ridotta (questa situazione è tipica degli scenari IoT in cui il consumo di energia deve essere basso), allora si può aggiungere un'implementazione personalizzata del **PayloadConverter** che utilizza una serializzazione diversa e adatta a soddisfare i vincoli a cui è soggetto il payload.

La classe concreta che implementa **PayloadConverter** deve essere identica nei client e nei server che devono comunicare fra loro.

3. Il package *response* racchiude **LindaResponse** e il package *request* racchiude **LindaRequest**, questi sono gli oggetti inviati rispettivamente nelle fasi del modello di interazione request-response definito da CoAP.

Si può osservare in figura 3 che la **LindaRequest** contiene i field **uri** e **expirationTime**. Si è già discusso di **uri** durante l'analisi dei requisiti, questo è il destination URI dove verrà inviato l'eventuale risultato sospeso. Non è mai stato citato **expirationTime**, perché questo dettaglio è stato aggiunto durante il design del client sincrono. In pratica, l'esecuzione sincrona di un'operazione di Linda, come tale, richiede di fermare l'esecuzione del programma fintanto che non è soddisfatta; per cui, in questo caso, si è ritenuto necessario inserire **expirationTime** in modo da sbloccare l'esecuzione sincrona subito dopo che è trascorsa la soglia di tempo **expirationTime** (questo aspetto viene trattato con maggiore dettaglio nella sezione dedicata al comportamento del client 3.2.2, e nella sezione di interazione client-server 3.5).

4. Gli elementi all'interno del package *purelinda* modellano il dominio di Linda.

3.1.2 Behaviour

I comportamenti degli elementi di questo progetto sono i seguenti.

1. **TSPathVerifier** verifica la validità di un path appartenente ad un URI secondo quanto riportato in figura 2, inoltre include **TUPLE_SPACE** cioè il nome *tuple-space* che deve avere una resource genitore per "dichiarare" le sue child resource di tipo spazio di tuple.

TSPathVerifier fornisce anche altre utility secondarie.

Il motivo per cui questo elemento deve essere contenuto in client e server è che: nel **LindaServer** non si devono poter creare resource di tipo spazio di tuple che non rispettino il naming graph in figura 2, e allo stesso tempo il **LindaClient** non deve poter inoltrare operazioni Linda verso una resource che non rappresenta uno spazio di tuple.

2. **PayloadConverter** racchiude la logica per convertire un oggetto **LindaRequest** o un oggetto **LindaResponse** in payload pronto per essere inviato, e viceversa per convertire il payload ricevuto in un oggetto **LindaRequest** o **LindaResponse** pronto per essere utilizzato nel sistema.

PayloadConverterImpl implementa **PayloadConverter** utilizzando la Java object serialization.

3. **LindaRequest** e **LindaResponse** sono strutture dati in cui racchiudere i meta-dati dell'astrazione del progetto LindaOnCoAP (**uri** e **expirationTime**) insieme agli elementi del dominio di Linda: **LindaOperation**; **LindaOperationID**; **LindaOperationResultID**; **Tuple**; **Template**.
4. Gli oggetti **Tuple** e **Template** interagiscono secondo la regola di match definita nel formalismo di Linda.
5. Gli oggetti di tipo **LindaOperation**, **LindaRequest** e **LindaResponse** devono avere un contenuto preciso in relazione all'identificativo (enumerazioni **LindaOperationID** e **LindaOperationResultID**) che indica l'operazione o il risultato rappresentato.

Ad esempio: un oggetto **LindaOperation** che contiene **LindaOperationID.WRITE**, deve contenere un oggetto di tipo **Tuple**, mentre il suo field **template** deve essere inizializzato a **null**. Se si vuole trasmettere questa **LindaOperation**, è necessario inserirla in una **LindaRequest**, e quest'ultima deve avere **uri** e **expirationTime** impostati a **null**, perché l'operazione write di Linda non è sospensiva.

Un altro esempio: un oggetto **LindaOperation** che contiene **LindaOperationID.TAKE**, deve contenere un oggetto di tipo **Template**, mentre il suo field **tuple** deve essere inizializzato a **null**. Se si vuole trasmettere questa **LindaOperation**, è necessario inserirla in una **LindaRequest**, e quest'ultima deve contenere l'URI sul quale ricevere l'eventuale risultato sospeso, opzionalmente può avere anche impostato **expirationTime** (solo il client sincrono inizializza questo valore).

Quindi per fare rispettare la corretta creazione di questi oggetti, il costruttore delle loro classi è stato definito **protected**, e la responsabilità di creazione è stata affidata totalmente alle loro rispettive **Factory**, le quali implementano il pattern **Factory Method**.

3.1.3 Interaction

Questo progetto deve essere incluso come dipendenza negli altri due, quindi le sue utility e i suoi elementi vengono utilizzati dai componenti client e server.

3.2 Client

La classe **CoapClient** di Californium implementa la trasmissione sincrona e asincrona, di conseguenza si è deciso di sviluppare un Linda client per ciascuna trasmissione.

Successivamente, si deve considerare ciò che è emerso dall'analisi dei requisiti. Dato che solo le operazioni bloccanti di Linda devono disporre di un CoAP server interno per ricevere il risultato sospeso, è bene separare le operazioni bloccanti da quelle non bloccanti in due classi distinte.

Quindi sono state definite 4 classi di Linda client: **LindaSyncClient**; **LindaAsyncClient**; **LindaSyncNBClient**; **LindaAsyncNBClient**. Le lettere NB sono l'abbreviazione di *Non-Blocking*.

LindaSyncClient e **LindaAsyncClient** implementano tutte le operazioni di Linda, bloccanti e non bloccanti.

3.2.1 Structure

Prima di iniziare a descrivere la struttura vera e propria, sono necessarie un paio di considerazioni riguardo la suddivisione dei Linda client in 4 classi. Questa scelta è dovuta principalmente allo studio della classe di Californium **CoapClient** che si occupa di rappresentare appunto il client dell'architettura di CoAP. **CoapClient** offre davvero tantissimi metodi per iniziare la comunicazione, sincroni e asincroni, ed infatti è lunga 1665 righe di codice. Sicuramente è comodo racchiudere tutti i metodi di comunicazione in un'unica classe client, tuttavia non è stato semplice scegliere il metodo più adatto per farlo nel Linda client. Il problema di racchiudere tutti i metodi di comunicazione in un'unica classe è che rende piuttosto difficile la sua comprensione a chi la deve utilizzare per la prima volta. Quindi per rendere più comprensibile il Linda client, è stato diviso in 4 classi.

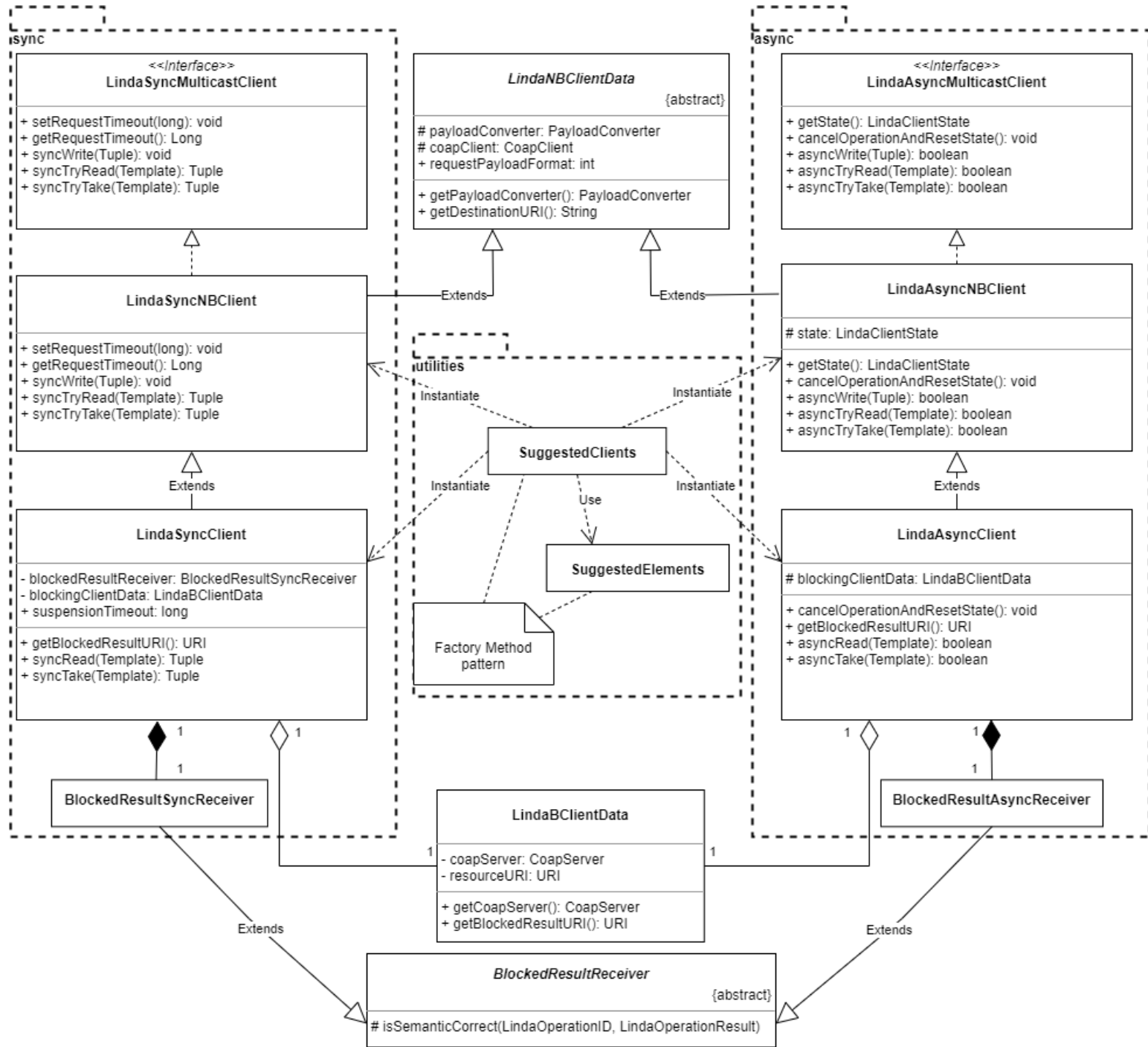


Figura 4: UML class diagram del progetto Client.

Un'altra osservazione riguardo il `CoapClient`, è che, forse, se ci fosse modo di separare i metodi di comunicazione sincroni e asincroni in due classi distinte (e osservando il codice del `CoapClient`, sembrerebbe fattibile), la compresione del `CoapClient` ne gioverebbe, così come il suo utilizzo. In conclusione, si ritiene che dato un oggetto, se con esso si vuole effettuare un'operazione sincrona o asincrona, tendenzialmente si continuerà ad utilizzarlo in modo sincrono o asincrono per tutto il suo ciclo di vita, e anche questa considerazione consolida la suddivisione dei Linda client in 4 classi.

Intuitivamente, i Linda client condividono vari elementi tra loro. Si inizia a descrivere la loro struttura a partire dai client non bloccanti, perché sono i più semplici.

Innanzitutto, si deve constatare che la sola differenza tra `LindaSyncNBClient` e `LindaAsyncNBClient` consiste nell'essere uno sincrono e l'altro asincrono. Essi devono condividere gli stessi oggetti per iniziare la comunicazione. Come rappresentato in figura 4, `LindaNBClientData` è la struttura dati che racchiude gli elementi per comunicare (in modo non sospensivo) con il `LindaServer`, ed è stata impostata come classe `abstract` perché non ha alcun senso istanziarla fuori da un contesto che non sia un Linda client. Quindi entrambe le classi dei Linda client non bloccanti estendono `LindaNBClientData`. Prima di passare ai client bloccanti, si deve discutere delle due interfacce `LindaSyncMulticastClient` e `LindaAsyncMulticastClient`. Come descritto nell'analisi dei requisiti, si è stabilito che si devono effettuare in multicast solo operazioni non bloccanti, quindi i client non bloccanti implementano queste interfacce. Tuttavia inserire queste due interfacce nel sistema non è particolarmente funzionale nella pratica, infatti non hanno un vero e proprio utilizzo concreto. Le classi dei client non bloccanti implementerebbero comunque tutti i loro metodi senza disporre di queste interfacce, ad ogni modo, queste interfacce sono necessarie perché rappresentano all'interno del codice l'intento di effettuare in multicast solo operazioni non bloccanti.

Osservando la figura 4, così come i client non bloccanti estendono `LindaNBClientData`, i client bloccanti contengono `LindaBClientData`. Sarebbe più idoneo fare estendere `LindaBClientData` dalle classi dei client bloccanti, anche perché non si dovrebbe poter istanziare un `LindaBClientData` fuori da un contesto che non sia un client bloccante, perché non ha alcun senso. Purtroppo però in Java non è disponibile l'ereditarietà multipla, e i client bloccanti devono assolutamente estendere quelli non bloccanti, perché i client bloccanti sono client completi di tutte le funzionalità.

Un altro paio di osservazioni sulla struttura di `LindaSyncClient` e `LindaAsyncClient`. Entrambi usano la classe di Californium `CoapServer` per utilizzare la resource che riceve gli eventuali risultati sospesi, ma utilizzano due implementazioni diverse di resource. La classe di Californium `CoapResource` viene estesa da `BlockedResultReceiver`, quest'ultima è una classe `abstract` che viene estesa a sua volta dalle due concrete resource che collaborano con il loro relativo Linda client per ricevere i risultati sospesi, e completare le operazioni con comportamento sincrono o asincrono.

3.2.2 Behaviour

Le differenze tra i Linda client sincroni e asincroni, sono quelle tipiche del codice sincrono e asincrono.

Il client sincrono pone un paio di timeout.

1. Il primo timeout è `CoapClient.timeout`.
 - a) È configurabile tramite `LindaSyncNBClient.setRequestTimeout(long)`.
 - b) Inizia dopo l'invio della `LindaRequest`.
 - c) Deve essere un timeout di breve durata (pochi secondi) perché il `LindaServer` risponde sempre appena possibile inviando una `LindaResponse`.
 - d) Se `LindaSyncNBClient` non riceve nessuna risposta prima dello scadere di questo timeout, viene lanciata una `RequestTimeoutException`
2. Il secondo timeout è `LindaSyncClient.suspensionTimeout`.
 - a) È modellato come `public field`.
 - b) Inizia dopo la ricezione della `LindaResponse` che rappresenta la semantica bloccante e dopo l'avvio del `CoapServer` interno in cui verrà ricevuto il risultato sospeso.
 - c) Deve essere un timeout di lunga durata (molti minuti) perché deve modellare la semantica sospensiva di Linda.
 - d) Se `LindaSyncClient` non riceve nessuna `LindaResponse` contenente la tupla (risultato) sospesa prima dello scadere di questo timeout, viene lanciata una `SuspensionTimeoutException`.
 - e) Questo timeout è utilizzato per configurare `LindaRequest.expirationTime`.

Il client asincrono ha lo stesso comportamento del client sincrono, ma non impone alcun timeout, non restituisce il risultato sincronicamente, e possiede uno stato rappresentato dalla classe `LindaClientState`.

Il `LindaClientState` si ottiene eseguendo `LindaAsyncNBClient.getState()`, e mantiene salvate le informazioni sottostanti.

- L'identificativo dell'ultima operazione di Linda che ha iniziato il client.
- Il progresso dell'ultima operazione di Linda che ha iniziato il client. Per rappresentarlo è stata realizzata una enumerazione apposita: `LindaComputationState`.
- Il risultato dell'ultima operazione di Linda che ha iniziato il client (se è presente).

Si può fermare l'esecuzione di un operazione Linda in qualsiasi momento eseguendo `LindaAsyncNBClient.cancelOperationAndResetState()`. Si deve prestare molta attenzione quando si utilizza questo metodo, perché se il server ha già ricevuto la `LindaRequest` continuerà a gestirla come dovuto.

Per mettere a confronto le analogie e differenze tra il comportamento dei due client, si possono osservare i diagrammi di stato in figura 5 e 6, i quali hanno molte transizioni in comune.

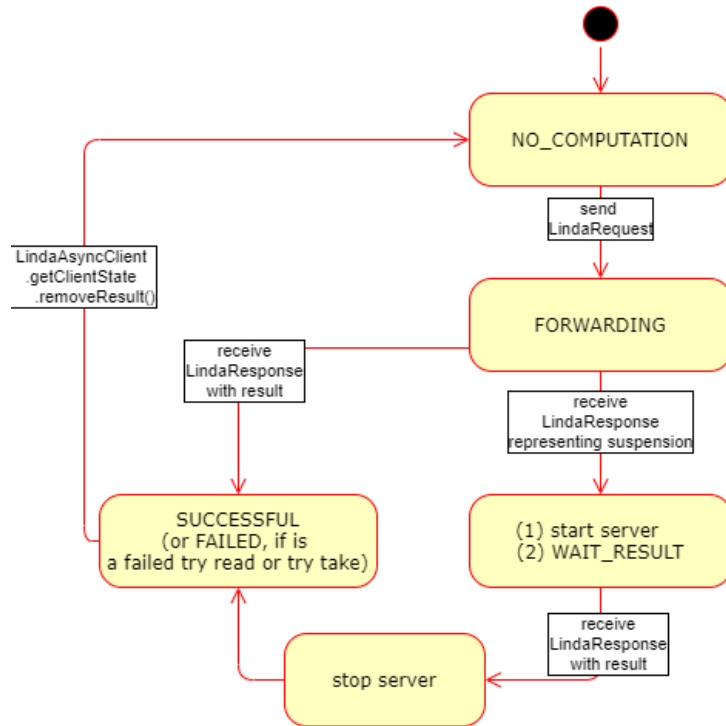


Figura 5: UML state diagram del comportamento di `LindaAsyncClient`.
 Non sono rappresentati gli stati di errore.
 Le scritte in maiuscolo sono il valore assunto da `LindaComputationState`.

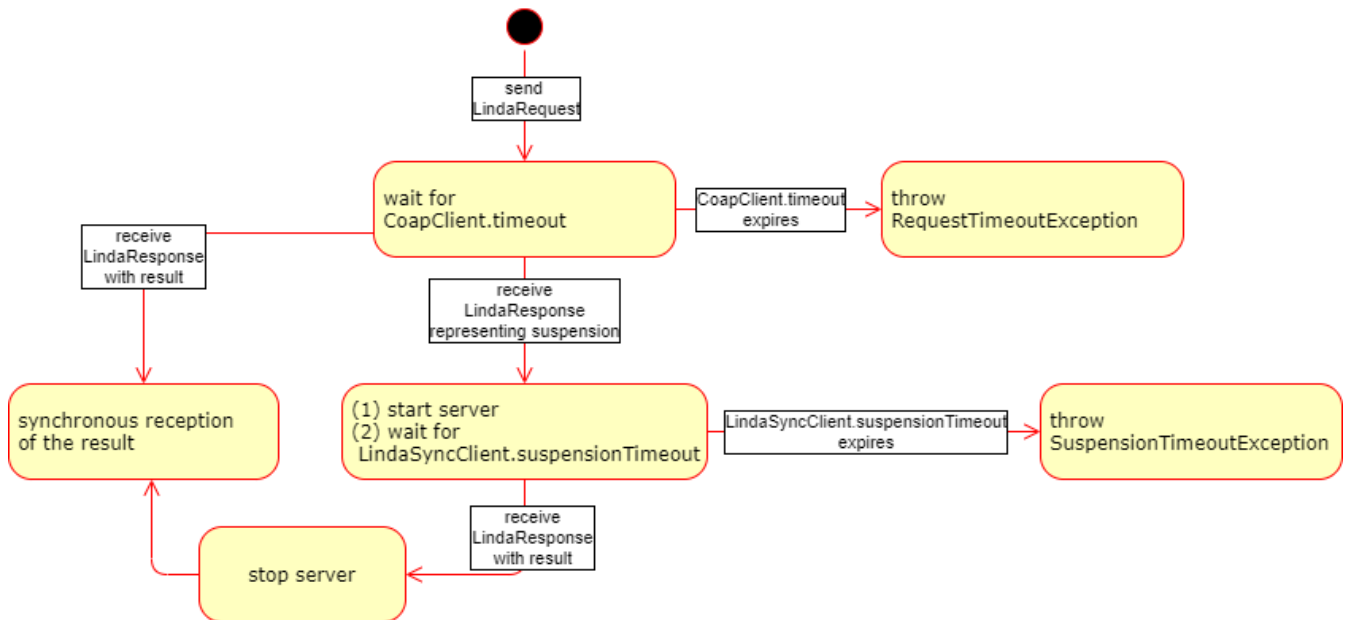


Figura 6: UML state diagram del comportamento di `LindaSyncClient`.
 Sono stati omessi molti lanci di exception.

Ci sono due approcci per creare gli oggetti Linda client.

- La classe **SuggestedClients** implementa il pattern Factory Method per creare oggetti Linda client configurati con l'impostazione di default proposta in questo progetto. Questi metodi di creazione sono consigliati nel caso in cui si voglia utilizzare gli elementi LindaOnCoAP senza configurare personalmente gli elementi di Californium.

SuggestedClients utilizza la classe **SuggestedElements**, anche quest'ultima implementa il pattern Factory Method per creare gli elementi di Californium configurati con l'impostazione di default proposta in questo progetto.

- Alternativamente, i Linda client possono essere istanziati tramite i costruttori delle loro classi, fornendo come input gli elementi di Californium personalizzati ad-hoc.

Un'ultima osservazione molto importante, riguardo il comportamento dei client: un oggetto Linda client può effettuare solo un'operazione di Linda alla volta. Per essere più precisi, ecco una lista riguardo i motivi per cui i Linda client non possono svolgere molteplici operazioni di Linda in modo concorrente.

- In realtà, un oggetto di tipo **LindaSyncNBClient** può essere utilizzato per svolgere molteplici richieste contemporaneamente, ma il suo **CoapClient** interno deve essere configurato correttamente, cioè per coerenza con gli altri client, l'impostazione di default proposta in **SuggestedClients** può gestire solo una richiesta alla volta.
- **LindaAsyncNBClient** e **LindaAsyncClient** possono svolgere una sola richiesta alla volta, perché il suo **LindaClientState** mantiene solo un risultato.
- **LindaSyncClient** e **LindaAsyncClient** possono svolgere una sola richiesta alla volta, perché le classi concrete di tipo **CoapResource** sono programmate per ricevere un solo risultato, e gestire la ricezione di molteplici risultati in modo concorrente coordinandoli con il **CoapClient** che le invia, non sembra affatto semplice.

3.2.3 Interaction

Si rimanda alla sezione 3.5.

3.3 Riflessioni sui Client e Client con "Richieste in Coda"

3.3.1 5 Metodi per 5 Operazioni

Considerazioni sugli argomenti dei metodi.

I client illustrati in precedenza possiedono 5 metodi distinti per svolgere 5 operazioni differenti di Linda. Questo dettaglio offre il vantaggio di programmare tramite il formalismo di Linda, ignorando completamente il layer di CoAP, infatti questi metodi per operare richiedono come argomento solamente una **Tuple** oppure un **Template**.

Tuttavia esponendo un piccolo dettaglio della soluzione LindaOnCoAP, è possibile ridurre i metodi ad uno solo. Il modello di comunicazione request-response di CoAP permette di iniziare la comunicazione tramite l'invio di una request, quest'ultima in LindaOnCoAP richiede di effettuare le operazioni di Linda inviando un oggetto **LindaRequest**, quindi è il contenuto di **LindaRequest** che determina quale operazione di Linda svolgere, pertanto l'interfaccia di un Linda client può essere ridotta ad un solo metodo che accetta come argomento una **LindaRequest**.

Considerazioni sul tipo di ritorno dei metodi.

Unificare i metodi in uno solo, implica unificare il tipo di dato restituito.

Client sincrono. Nel client sincrono, è più corretto utilizzare 5 metodi distinti, perché in alcuni casi si deve avere sempre e solo un risultato (N.B: restituire un solo risultato è un vincolo maggiore rispetto a restituire un solo tipo di ritorno): la write restituisce sempre **void**; la read e la take restituiscono sempre una **Tuple**. Le operazioni try read e try take restituiscono una **Tuple** in caso di successo, oppure **null** in caso di fallimento. (Nei casi di errore vengono lanciate eccezioni.)

Unificare il tipo di ritorno è giusto nei casi di try read e try take, poiché utilizzare **null** potrebbe essere considerato una cattiva pratica di programmazione. Supponiamo quindi di unire try read e try take, facendo restituire un nuovo tipo di dato che chiamiamo **TupleOrFail**. A questo punto se continuiamo l'unificazione con read e take, oscuriamo il loro risultato, il quale deve essere sempre di tipo **Tuple**, per non parlare della write che restituisce sempre **void**. In conclusione, unificare i metodi sincroni oscura il loro funzionamento, e non risulta affatto chiaro raggrupparli nei seguenti sottoinsiemi caratterizzati dallo stesso tipo di ritorno: try read e try take; read e take; write.

Client asincrono. Nel client asincrono, tutte le operazioni scrivono il loro risultato nello stato dell'oggetto client che le esegue, e facendo in questo modo non viene oscurato nulla, perché il client asincrono possiede il **LindaClientState**, il quale memorizza i dati dell'ultima operazione di Linda invocata. Quindi in questa situazione è ragionevole unificare i metodi asincroni, poiché non vi è alcun conflitto fra loro.

3.3.2 Client asincrono: responsabilità del chiamato e del chiamante, stato vs future

C'è un forte accoppiamento tra un client e il risultato di un'operazione, perché un client può svolgere una sola operazione alla volta.

Nel client asincrono illustrato in precedenza, è responsabilità del chiamato occuparsi di ricevere e mantenere il risultato nel suo stato interno. Di conseguenza il chiamante deve preoccuparsi di effettuare un'operazione alla volta e prima di svolgere la prossima operazione deve ottenere il risultato precedente interrogando lo stato del client chiamato. Questo dettaglio impatta in maniera rilevante sull'usabilità del sistema.

Per migliorare questa situazione, si deve rimuovere lo stato interno dal client asincrono e modellarlo tramite una future esterna all'oggetto client. In questo modo, il risultato è mantenuto in un oggetto dedicato, ed è possibile ottenerlo in un momento qualsiasi, anche dopo aver effettuato altre operazioni con lo stesso client.

Inoltre utilizzando una future dedicata, è possibile estendere maggiormente le funzionalità del client, in particolare è possibile aggiungere un meccanismo per accodare le richieste e risolverle sequenzialmente, aggiornando indipendentemente i loro risultati mantenuti nelle future.

3.3.3 `LindaAsyncClientWithQueue`

`LindaAsyncClientWithQueue` è un terzo client (asincrono) che possiede tutte le miglirie introdotte nelle precedenti sezioni appena trattate.

- La sua interfaccia possiede un solo metodo per eseguire le operazioni.
- Il singolo metodo con cui esegue operazioni, accetta una richiesta, e se è possibile, la esegue immediatamente, altrimenti se è occupato ad eseguire un'altra richiesta, inserisce la nuova richiesta in coda e la eseguirà seguendo l'ordine della coda FIFO.
- `LindaAsyncClientWithQueue` utilizza un proprio oggetto `LindaSyncClient`, quindi ogni richiesta eseguita è caratterizzata internamente dagli stessi timeout.
- Ad ogni operazione restituisce un oggetto `CompletableFuture<LindaAsyncResult>`, e come facilmente intuibile, la classe `LindaAsyncResult` rappresenta "solamente" il risultato di un'operazione svolta da questo client.
- Il design veramente ottimo dovrebbe essere un client asincrono, fortemente accoppiato con una classe future personalizzata ad-hoc, la quale rappresenti non il risultato dell'operazione di Linda, ma l'operazione stessa, da cui è possibile osservare lo stato dell'operazione (proprio come previsto nel primo client asincrono avente stato interno).

Questa migliria sembra piuttosto impegnativa e non introduce un contributo significativo, quindi per mancanza di tempo si rimanda ad eventuali lavori futuri.

- Attenzione però, si potrebbe pensare che il design ottimo debba includere anche un meccanismo per interrompere l'operazione rappresentata dalla future (solo se è attualmente in corso) in modo da sbloccare il client asincrono (proprio come previsto nel primo client asincrono avente stato interno) e procedere con le prossime richieste in coda. In questo contesto, tale meccanismo induce a compiere errori.

Nel caso del primo client asincrono, dato che questo può accettare e svolgere solo un'operazione alla volta, fermare l'esecuzione di un'operazione può avere senso, ma è comunque "pericoloso" perché come già spiegato in precedenza, l'operazione viene fermata solo a client side, tuttavia si può continuare ad usare questo client con cognizione di causa ed aggirare tranquillamente i vari problemi che potrebbero derivare dal fermare l'operazione.

Nel caso di `LindaAsyncClientWithQueue`, le operazioni devono essere coordinate tramite i timeout, perché è cruciale che client e server siano coordinati tra loro (preventivamente, mediante la scadenza della `LindaRequest`).

Fermare arbitrariamente l'esecuzione di un'operazione in un client di questo tipo, causa comportamenti imprevedibili nelle operazioni in coda svolte a seguire (il problema è: in quale stato si trovava l'operazione interrotta? Questa situazione potrebbe creare scenari simili a quelli descritti nella sezione in cui si trattano i problemi del multicast, sezione 2.1.1 punto 4), pertanto in questo caso, l'interruzione arbitraria di un'operazione richiede l'introduzione di un apposito meccanismo per comunicare l'interruzione anche a server side.

Alternativamente si potrebbe aggiungere facilmente un meccanismo che interrompe l'operazione attualmente in corso (solamente a client side) e allo stesso tempo elimina tutte le altre richieste in coda.

3.4 Server

E' stata definita un'unica classe `LindaServer`, in grado di far fronte alle richieste provenienti da qualsiasi tipo di `LindaClient`, sia sincrono che asincrono.

3.4.1 Structure

La classe principale è quella del `LindaServer`.

Il `LindaServer` contiene al suo interno un `CoapServer`, che serve a gestire le richieste provenienti dai client. Al `CoapServer` vengono assegnate delle `TupleSpaceResources`, che estendono le `CoapResources` proprie di Coap.

Queste resources contengono ciascuna al proprio interno un `TupleSpace`. Il `LindaServer` mette a disposizione metodi per gestire gli spazi di tuple e per aggiungere eventuali endpoints al `CoapServer`. La gestione delle tuple è totalmente demandata alla classe `TupleSpace`.

Questa classe rappresenta uno spazio di tuple. E' identificata da un path ed al suo interno contiene le tuple ed i metodi per interagire con esse. Questa classe offre inoltre i metodi per gestire le richieste di Linda (Read, Take e Write) eseguite sullo spazio di tuple in questione. Infine, nella classe `TupleSpace` sono contenute anche le `PendingRequests` relative allo spazio di tuple in questione.

La classe `PendingRequests` gestisce le richieste che non sono ancora state soddisfatte relative ad uno spazio di tuple. Quando una tupla richiesta non è contenuta nello spazio di tuple selezionato, la richiesta viene salvata in questa classe e, una volta che può essere soddisfatta, è questa classe che si occupa di rispondere al client con la tupla desiderata. Infine, all'interno di un `LindaServer` è contenuta la classe `LindaServerSettings`. Questa classe contiene il `PayloadConverter`, che deve essere lo stesso utilizzato dal client, un `Logger`, che serve per esportare i log del server, dai quali è possibile analizzare il comportamento del suddetto e riscontrare eventuali problemi o anomalie, ed infine l'indirizzo del server.

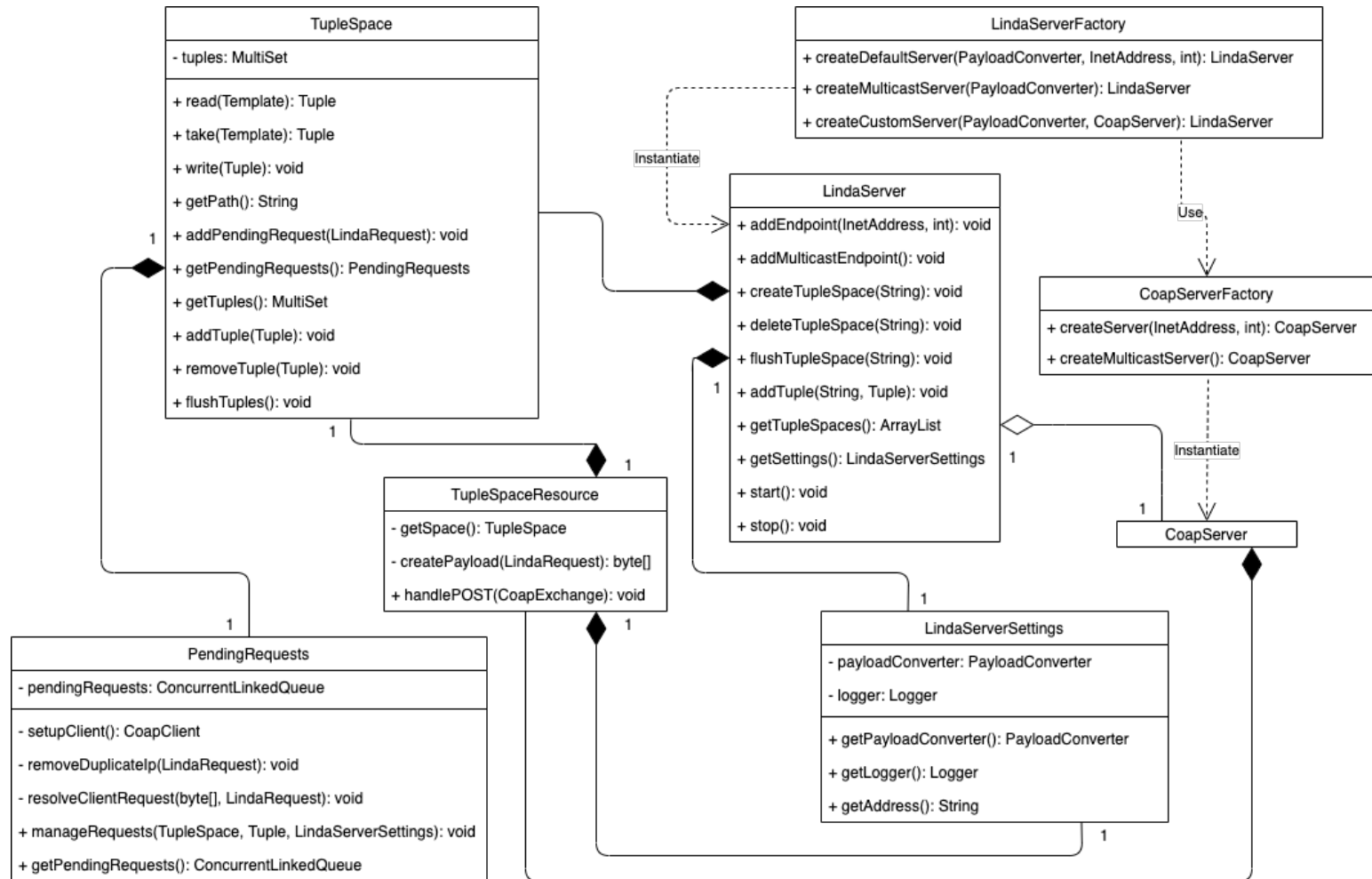


Figura 7: UML class diagram del progetto Server.

3.4.2 Behaviour

All'arrivo di una richiesta da parte di un client, la classe che si occupa di analizzarla è la **TupleSpaceResource**. La richiesta arriva come POST, e la suddetta classe analizza il payload ed esegue la richiesta di Linda che è arrivata dal client sul suo **TupleSpace**. La classe **TupleSpace** gestisce la richiesta e, se questa viene soddisfatta, restituisce la tupla richiesta (o un codice di successo in caso dell'operazione di Write) alla **TupleSpaceResource**, che si occupa di creare la risposta con opportuno payload ed inviarla al client.

Se la richiesta non viene soddisfatta, ed è una richiesta di tipo Read o Take, questa viene inserita nella classe **PendingRequests**. Viene inoltre restituito un messaggio alla **TupleSpaceResource**, che crea una risposta per il client che contiene uno specifico pay-

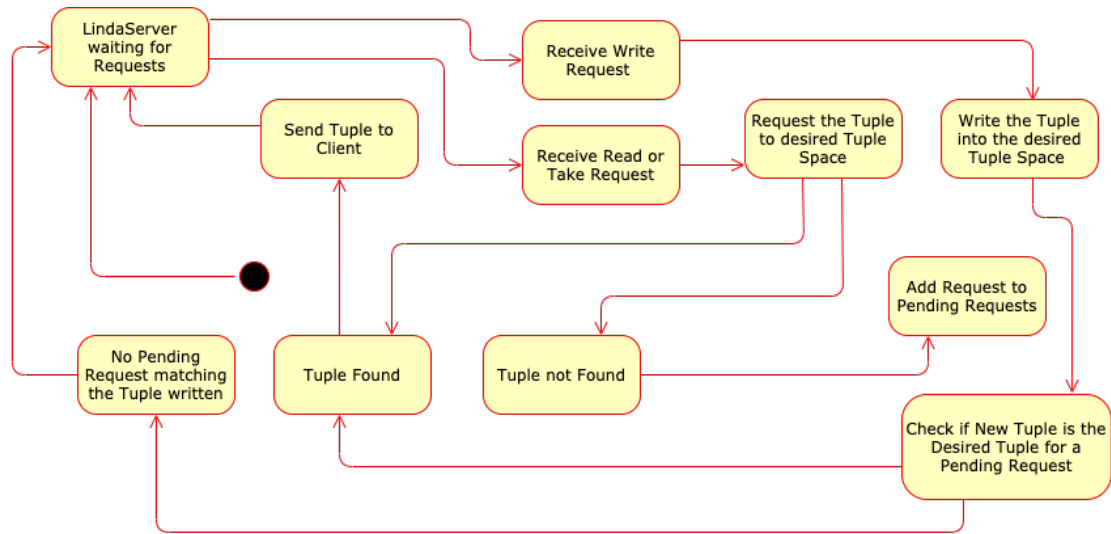


Figura 8: Diagramma di Stato che rappresenta il comportamento del LindaServer.

load il quale lo informa che la richiesta è sospesa.

Quando viene eseguita un'operazione di Write su uno spazio di tuple al cui interno sono presenti delle **PendingRequests**, queste vengono controllate e, se la tupla che è stata aggiunta soddisfa una richiesta, dopo aver controllato che questa non abbia tempo di scadenza o che il tempo di scadenza della richiesta non sia trascorso, la classe **PendingRequests** si occupa di creare un **CoapClient**. Questo **CoapClient** esegue una richiesta, al cui interno risiede un payload con la tupla desiderata, verso il **CoapServer** che è attivo all'interno del **LindaClient** che aveva originariamente eseguito la richiesta.

3.4.3 Interaction

Si rimanda alla sezione 3.5.

3.5 Interazione Client-Server

Arrivati a questo punto serve un'altra precisazione: abbiamo usato i messaggi non confermabili di CoAP dato che i test sono JUnit e sono in locale, quindi i messaggi non possono andare persi.

Nella Figura 9 si evince lo scambio di dati tra client e server. Sono rappresentate tutte le operazioni, sia bloccanti che non.

Per quanto riguarda la componente Multicast, lo schema delle interazioni è rappresentato allo stesso modo del primo blocco mostrato nella Figura 9, se nonché tale operazione a lato server avviene in tutti i server che hanno al loro interno un **CoapServer** che appartiene al gruppo multicast.

Infine, si precisa che l'interazione mostrata nella Figura 9 coinvolge dei Client Asincroni.

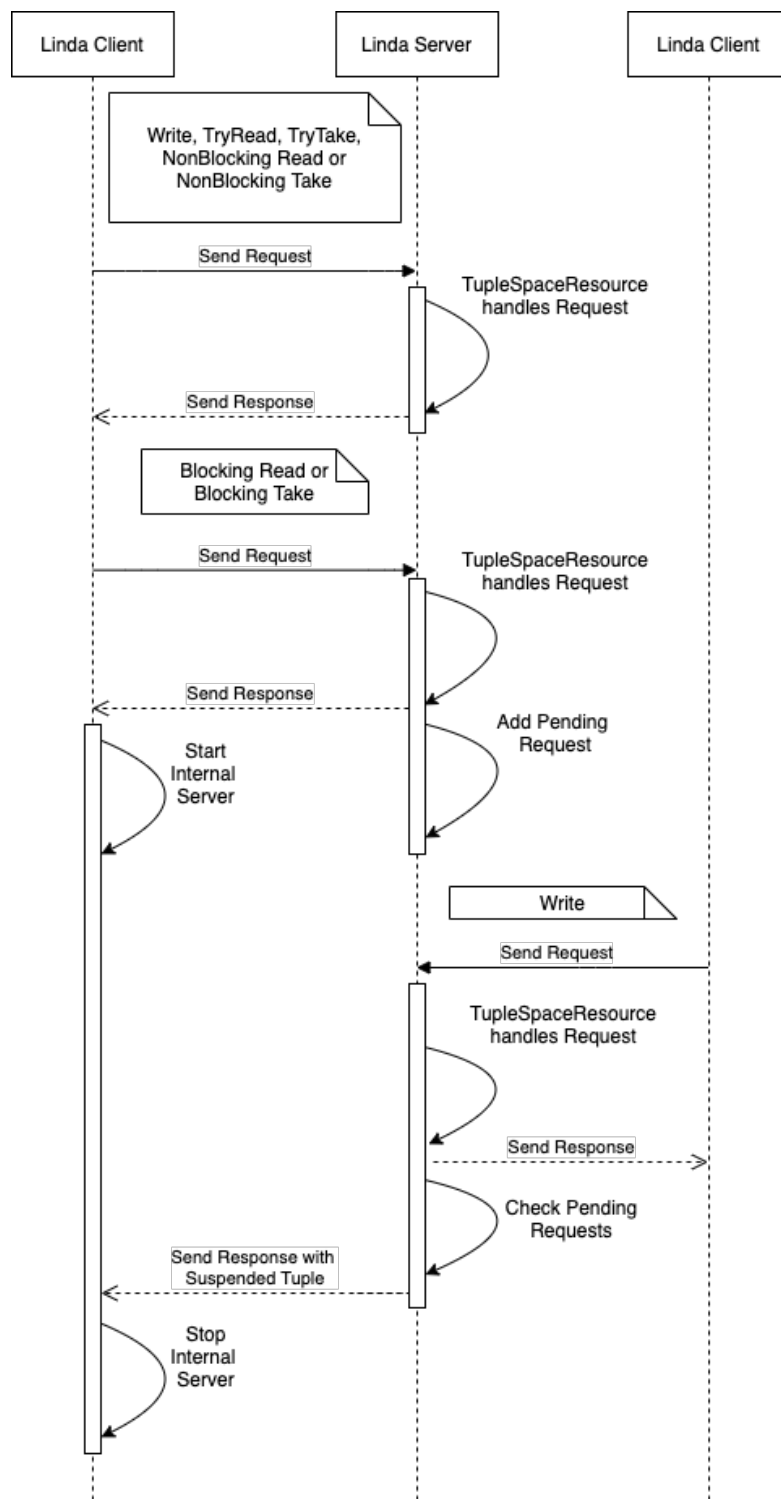


Figura 9: Diagramma di Sequenza delle interazioni tra Client e Server.

4 Implementation Details

4.1 Client Sincroni

I metodi che eseguono operazioni di Linda, dichiarano una quantità notevole di eccezioni. In Java purtroppo le eccezioni sono checked, quindi l'implementazione deve essere lasciata in questo modo.

```
public Tuple syncRead(final Template template) throws  
    IOException, ConnectorException, ClassNotFoundException, ExecutionException,  
    InterruptedException, RequestTimeoutException, SuspensionTimeoutException {
```

Si potrebbe pensare di racchiudere qualche eccezione in una sola, ma così facendo si nascondono gli errori del sistema e si rende più difficile il debug in caso di problemi. Ogni eccezione dichiarata rappresenta un errore ben preciso, e per ognuno di essi è possibile disporre un'apposita gestione dell'errore.

5 Self-Assessment / Validation

Per validare il nostro progetto, abbiamo scelto di utilizzare i test di JUnit (versione 4), insieme ad alcune prove pratiche utilizzando l'applicazione reale.

L'applicazione reale è fondamentale per testare il funzionamento del multicast, soprattutto nelle operazioni di try read e try take, è necessario monitorare globalmente quali tuple vengono rimosse dai server e quale tupla viene letta dal client. Non è ragionevole testare le operazioni in multicast solo con un test dedicato, poiché il risultato letto da try read e try take in multicast non è deterministico.

Per quanto riguarda invece il code coverage, per calcolarlo si è utilizzato il tool integrato in IntelliJ, e questo è quanto emerso dai test.

- Nel progetto Communication, le classi da testare sono molto semplici, si tratta di `TSPathVerifier` e `PayloadConverterImpl`. I test unità sono perfetti per verificarne il funzionamento, infatti in queste classi si è raggiunto il coverage al 100%. Anche `PathUtilities` ha raggiunto il 100%, mentre invece gli altri elementi del package `payload` hanno raggiunto all'incirca il 35%, tuttavia questo è irrilevante poiché la loro implementazione è piuttosto banale, ed il loro funzionamento corretto è implicito perché sono classi che vengono utilizzate in tutto il sistema.
- Il progetto Client ha raggiunto una copertura del 75%, calcolata considerando:
 - i test di interazione client-server nel package `lindaoncoap.localcommunication`
 - il test `lindaoncoap.client.async.ForwardingStateTest`

Eventualmente si può ulteriormente migliorare aggiungendo un paio di test per innescare le eccezioni del client sincrono.

- Il progetto Server ha raggiunto una copertura del 80%, calcolata considerando:
 - i test di interazione client-server nel package `lindaoncoap.localcommunication`
 - test offline creati per testare i meccanismi interni del server

6 Deployment Instructions

Di seguito verranno elencate le istruzioni e i passi necessari per l'installazione degli strumenti richiesti per il corretto avvio dell'applicazione e dei test.

1. Installare Java JDK 14.
2. Installare git, ed eseguire il comando `git clone` sul repository del progetto <https://gitlab.com/pika-lab/courses/ds/projects/sd-project-bondi-camporesi-aa1920>
3. Build: a questo punto bisogna eseguire la build di Gradle del progetto. Quindi, bisogna spostarsi nella cartella del progetto con il terminale ed eseguire il seguente comando:

```
.\gradlew build
```

4. Test (eseguiti già durante la Build): per eseguire i test del progetto, spostarsi con il terminale nella cartella contenente il progetto ed eseguire la seguente istruzione:

```
gradle test
```

5. Applicazione: infine, per lanciare la demo di applicazione reale che abbiamo realizzato, lanciare il seguente comando da un terminale posizionandosi nella cartella di progetto:

```
.\gradlew ApplicationExample
```

7 Usage Examples

In questa sezione verrà spiegato il funzionamento e l'implementazione delle componenti personalizzabili dell'applicazione. A tale scopo verrà usato come esempio il codice che è stato scritto per la realizzazione dell'esempio di applicazione reale.

7.1 Tuple e Template

Per realizzare una tupla personalizzata, basta creare una classe che implementi l'interfaccia `Tuple`. In questo modo si andrà a creare la propria tipologia personalizzata di tuple con i metodi necessari al funzionamento della nostra libreria.

Allo stesso modo, per creare il proprio tipo di template, basta creare una classe che implementi l'interfaccia `Template`.

7.2 PayloadConverter

Si può cambiare il metodo di serializzazione delle `LindaRequest` e `LindaResponse`, implementando una classe personalizzata che estenda l'interfaccia `PayloadConverter`. Questa classe personalizzata dovrà essere passata durante la creazione ad-hoc dei propri `LindaServer` e `Linda` client personalizzati.

7.3 Server

Di seguito verranno illustrati i passi necessari alla creazione di un `LindaServer` di default.

```
final LindaServer server;  
server = LindaServerFactory.createDefaultServer(new  
    PayloadConverterImpl(), ipAddress, port);
```

Come si può vedere, è disponibile una factory che permette di creare un server di default passando come argomenti un `payloadConverter`, l'indirizzo ip e la porta del server.

Per aumentare il grado di personalizzazione della nostra libreria, al `LindaServer` va passato come argomento un `PayloadConverter`, che può essere quello di default o personalizzato dall'utente per gestire i messaggi tra client e server in maniera diversa da quella pensata da noi.

Per aggiungere il server appena creato al gruppo multicast, basta eseguire la seguente istruzione.

```
server.addMulticastEndpoint();
```

Infine, le seguenti righe di codice permettono di creare sul server uno spazio di tuple, sfruttando anche lo strumento di verifica del path degli spazi di tuple, e l'ultima riga avvia il server.

```
String tupleSpacePathElliptical = TSPathVerifier.TS_PARENT_RESOURCE +  
    "/" + "elliptical";  
server.createTupleSpace(tupleSpacePathElliptical);  
server.start();
```

7.4 Client

Come esempio di client è stato scelto un client asincrono. Il codice seguente mostra come creare un client asincrono opportunamente settato per eseguire una richiesta su un determinato spazio di tuple di un determinato server.

Innanzitutto, viene definita la stringa che rappresenta lo spazio di tuple su cui eseguire le operazioni e da essa l'URI del server e dello spazio di tuple.

```
String tupleSpacePathElliptical = TSPathVerifier.TS_PARENT_RESOURCE +  
    "/" + "elliptical";  
InetAddress serverAddress1 = InetAddress.getByName("224.0.1.191");  
URI tupleSpaceUri1Elliptical = new URI(CoAP.COAP_URI_SCHEME + "://" +  
    serverAddress1 + ":5690/" + tupleSpacePathElliptical);
```

Quindi, viene creato il client utilizzando uno dei client predefiniti della nostra libreria. A sua discrezione, l'utente potrebbe utilizzare anche un altro tipo di client. Anche al client viene passato il PayloadConverter. Ovviamente deve essere lo stesso del server per far sì che entrambi interpretino i messaggi allo stesso modo.

```
private final LindaAsyncClient client1Elliptical;  
InetAddress localhost = InetAddress.getLocalHost();  
client1Elliptical = SuggestedClients.newAsync(  
    new PayloadConverterImpl(), tupleSpaceUri1Elliptical,  
    new InetSocketAddress(localhost, 5000),  
    new InetSocketAddress(localhost, 5001),  
    "deliverResource1");
```

Per il setup del client viene anche passato l'URI del server e dello spazio di tuple a cui connettersi, l'indirizzo da assegnare al client e l'indirizzo da assegnare al CoAPServer interno al client in caso questo si debba bloccare in attesa di un messaggio proveniente dal server.

A questo punto, per eseguire un'operazione di out di una tupla basta scrivere il seguente codice.

```
ProductTuple tuple = new ProductTuple("Ellittica Kettler", "001",  
    ProductType.ELLIPTICAL);  
client1Elliptical.asyncWrite(tuple);
```

Per eseguire un'operazione di take, invece, il codice è il seguente.

```
ProductTemplate template = new ProductTemplate(ProductType.ELLIPTICAL);
client1Elliptical.asyncTake(template);
```

7.5 Multicast

Per quanto riguarda i messaggi inviati in multicast, il setup del client è identico a quanto mostrato in precedenza. L'unica differenza è che l'indirizzo del server a cui connettersi dovrà essere il seguente.

Inoltre, il server dovrà essere dotato di un endpoint multicast.

```
InetAddress serverAddress = CoAP.MULTICAST_IPV4;
```

Nelle richieste multicast, l'operazione di out va eseguita nello stesso modo che negli altri messaggi. La differenza è nelle operazioni di read e take.

Relativamente a queste operazioni, nel caso del multicast vanno eseguite invece le try-Read e tryTake. Queste operazioni non sono bloccanti e come risultato il client si troverà la prima tupla corrispondente al template trovata in uno dei server a cui la richiesta è arrivata, oppure nessuna tupla se nessuno dei server a cui è arrivato il messaggio possiede una tupla che corrisponde al template nello spazio di tuple desiderato.

Queste operazioni non sono bloccanti poiché non sarebbe stato possibile poi gestire più server bloccati sulla stessa richiesta di un solo client, in quanto quando un server avrebbe risposto alla richiesta, gli altri non sarebbero stati in grado di trovare il client in attesa e si sarebbero bloccati.

Da notare che un'operazione di tryTake eseguita in multicast restituisce al client la prima tupla trovata nel primo server che risponde, ma rimuove la prima tupla trovata corrispondente al template in tutti i server in cui questa viene appunto trovata.

7.6 Applicazione Esempio

Una volta lanciata l'applicazione di esempio, ci si troverà davanti a tre GUI. La prima è quella dei Producer. Ogni pulsante di questa attiva un client che esegue un'operazione di out su uno o più server. Ci sono cinque diversi oggetti che possono essere consegnati, ed ognuno di questi può essere consegnato a tutti e tre i server contemporaneamente o ad uno singolo. Ci sono tre categorie diverse di prodotto, che corrispondono a tre diversi template.

La seconda GUI è quella dei Consumer. Questa permette di richiedere un determinato template ad un server singolo (operazione di take), o a tutti contemporaneamente (operazione di tryTake). Nelle operazioni di take, se un client non ha risposta, si blocca in attesa di ricevere la tupla. Quando il server corrispondente riceverà tale tupla, questa verrà inviata al client in attesa.

L'ultima interfaccia è quella Server. Questa mostra i tre server con le rispettive tuple contenute al loro interno, con la corrispondente quantità. In basso è anche presente un campo di testo in cui vedere i log dei client e dei server. I log dei server arriveranno appena questi riceveranno un'istruzione e agiranno su uno spazio di tuple. I log dei client arriveranno alla pressione del tasto "REFRESH".

Alcune prove che si possono fare con l'applicazione reale sono:

- eseguire un'operazione di write multicast di un oggetto su tutti i server e vedere se l'oggetto è stato aggiunto a tutti i server;
- eseguire un'operazione di tryTake multicast dell'oggetto precedente e vedere se è stato rimosso da tutti i server;
- eseguire un'operazione di write unicast di un oggetto su uno specifico server e vedere se l'oggetto viene aggiunto con successo;
- eseguire un'operazione di take unicast dell'oggetto precedente sul server a cui era stato aggiunto per vedere se viene rimosso e consegnato al client;
- eseguire un'operazione di take unicast su un server in cui non è presente l'oggetto desiderato e notare se il client non riceve nulla;
- eseguire un'operazione di write dell'oggetto desiderato sul server a cui era stato chiesto per vedere se dopo essere stato aggiunto, l'oggetto viene rimosso e consegnato al client che lo stava aspettando.

8 Conclusions

Il progetto da noi realizzato è in linea con gli obiettivi che ci eravamo proposti di realizzare. Anche se alcune delle funzionalità opzionali non sono state sviluppate, il sistema risulta completo nelle sue parti fondamentali, con l'aggiunta di qualche feature opzionale che siamo riusciti ad includere.

Abbiamo realizzato un'architettura client-server, basata su protocollo CoAP, che permetta tramite lo scambio di messaggi tra questi, di realizzare le operazioni di Linda sulle tuple contenute in diversi spazi di tuple remoti. I principali componenti realizzati sono:

- **LindaServer**: il server di Linda che al suo interno gestisce uno o più spazi di tuple sfruttando le resources di CoAP;
- **LindaClient**: il client di Linda, che esegue le operazioni di Linda di read, take, out, tryRead e tryTake su uno o più spazi di tuple.
I client possono essere sincroni e asincroni. I client sincroni bloccano la loro esecuzione se rimangono in attesa di una tupla fino al raggiungimento di un tempo di timeout impostabile. I client asincroni eseguono le operazioni in background, il che permette di monitorare il loro stato, visualizzare l'ultima operazione eseguita o bloccare la richiesta in corso.
- **PayloadConverter**: permette di gestire il payload contenuto nei messaggi scambiati tra client e server, garantendo una corretta interpretazione dei messaggi da ambo le parti;
- **TSPathVerifier**: una classe che permette di verificare il path degli spazi di tuple (che vengono gestiti dalle resources di CoAP), in modo tale che non ci siano spazi di tuple mal formattati.

8.1 Future Works

Di seguito un elenco dei possibili sviluppi futuri che abbiamo pensato di implementare:

- realizzazione di una libreria client e server che integri CoAP anche in altri linguaggi;
- implementazione del meccanismo di discovery delle risorse ad opera dei client verso i server in modo che i client possano in autonomia scoprire quali spazi di tuple sono raggiungibili per ogni server;
- implementazione di un meccanismo di content negotiation che sfrutti le possibilità offerte da CoAP.

8.2 What Did We Learned

Con questo progetto abbiamo imparato innanzitutto a conoscere più a fondo Linda, la sua semantica ed i suoi vincoli.

In seguito ci siamo approcciati a CoAP, del quale abbiamo studiato funzionalità e possibilità. Una volta approfondito questo protocollo, abbiamo studiato Californium, l'implementazione di CoAP già esistente realizzata in linguaggio Java.

Abbiamo lavorato in team, scoprendo sempre nuove difficoltà e trovando nuove soluzioni per ovviare alla distanza imposta dal periodo e dagli impegni lavorativi.

Abbiamo compreso più a fondo l'importanza dei test e quanto questi siano fondamentali per scoprire nuove criticità e di aiuto per comprendere meglio le possibili soluzioni e riuscire ad applicarle nel più breve tempo possibile.

Infine, abbiamo imparato ad usare Gradle e abbiamo esplorato le possibilità che offre, in particolare i vantaggi nel gestire una build multi-progetto.

Riferimenti bibliografici

- [1] Eclipse Foundation. Eclipse californiumTM.
- [2] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014.