ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

COORDINATION AS A WEB SERVICE: UNA MODERNA IMPLEMENTAZIONE DEL MODELLO LINDA

Elaborato in
SISTEMI DISTRIBUITI

Relatore
Prof. ANDREA OMICINI

Co-relatore
Dott. GIOVANNI CIATTO

Presentata da
LORENZO RIZZATO

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

coordinazione

Linda

interoperabilità

servizi web

ReST

A mia sorella Francesca,
come esempio di impegno
e perseveranza

Abstract

Scopo della tesi è lo sviluppo di un moderno middleware distribuito, basato sul modello di coordinazione LINDA, che possa fornire una sostanziale indipendenza di due aspetti fondamentali della coordinazione basata su tuple: il linguaggio di coordinazione e i linguaggi di comunicazione. Questo permetterà alle entità coordinate di interagire tramite gli spazi di tuple avendo a disposizione un insieme eterogeneo di tecnologie di rappresentazione dell'informazione tra cui scegliere. Queste funzionalità saranno rese accessibili tramite un servizio web che esporrà una API RESTful per abilitare l'interazione con gli agenti in rete; ciò sarà realizzato previa uno studio approfondito dei contributi tecnologici passati, le cui idee e concetti contribuiranno alla formulazione del concetto di *Coordination as a Web Service*. Infine, sarà proposta un'architettura software che permetta di realizzare un middleware semplice ma competitivo, in grado di trovare applicazione in molteplici contesti reali.

Indice

Introduzione

Il modello di coordinazione LINDA [7] fu originariamente creato e concepito nell'ambito dei sistemi chiusi, in particolare per il calcolo parallelo dove le entità computazionali facenti parte del sistema erano stabilite in fase di progettazione. In LINDA, i processi comunicano mediante uno *spazio di tuple* condiviso, un tipo di memoria associativa contenente un insieme di *tuple*, costituite da dati di natura possibilmente eterogenea. Le tuple sono rappresentate tramite un determinato *linguaggio di comunicazione*, che ne detta il formato di rappresentazione. A ogni linguaggio di comunicazione corrisponde un *linguaggio di templating*, usato per formulare le *query* sullo spazio di tuple.

Ai tempi dell'ideazione del modello LINDA, i sistemi concorrenti non erano generalmente concepiti per supportare tecnologie di comunicazione eterogenee, in quanto essi assolvevano a compiti ben specifici, dove il formato dei dati trasmessi durante l'elaborazione risultava essere specifico e ben definito. Negli ultimi anni, in particolare grazie all'arrivo sul mercato – sia consumer che industriale – di tecnologie mobili quali gli smartphone, WiFi, Bluetooth e – più recentemente – WiFi Direct, i progettisti di applicazioni distribuite hanno dovuto fare i conti con l'affermazione sempre più imponente dei cosiddetti sistemi aperti, ovvero scenari computazionali concorrenti le cui entità interagenti, altresì note come *agenti*, possono risultare fortemente diverse tra di loro. Le differenze possono essere innumerevoli, tra cui ad esempio la potenza computazionale o la "lingua" parlata dai singoli agenti.

Effettuando un'analisi delle tecnologie di coordinazione degli ultimi vent'anni, dove, sebbene l'interesse della comunità scientifica verso questo mondo stia seguendo una tendenza generalmente positiva, notiamo chiaramente come le implementazioni attualmente disponibili in letteratura generalmente non risultino pronte a un impiego su larga scala, né in contesti industriali, né nelle funzioni di *backbone* per lo sviluppo di applicazioni distribuite; questo perché non risultano di facile impiego, o semplicemente perché molte non sono più mantenute [4]. Oltretutto, è importante puntualizzare che il servizio di coordinazione fornito da tali implementazioni veniva spesso reso fruibile esclusivamente sotto forma di *libreria*; perciò non era possibile usufruire di una API che fornisse una distinzione netta fra interfaccia e implementazione.

Negli ultimo decennio, tuttavia, abbiamo assistito a una rapida crescita della popolarità e della pervasività dei servizi web. Particolare rilevanza ha assunto, nello specifico, lo stile architetturale denominato *Representational State Transfer* (ReST), che nel corso degli anni ha permesso la costruzione di servizi estremamente scalabili dotati di un'interfaccia pubblica ben definita (ReST API) e disaccoppiata dall'implementazione interna delle funzionalità esposte. Risulta quindi ragionevole pensare che una nuova implementazione di un modello di coordinazione come Linda possa trovare vantaggio nell'aderenza a tale stile per poter essere competitivo nel mondo di oggi. Il primo passo verso una tale infrastruttura, dunque, consisterebbe nel concepire gli spazi di tuple come *risorsa*, reificando quindi il concetto di coordinazione come servizio [12].

L'aderenza a un modello di progettazione *web-oriented* presuppone, peraltro, l'adozione di una serie di tecnologie standard di rappresentazione dei dati, di utilizzo comune. Fra le tante, a titolo di esempio, basti citare le più famose, JSON e XML. Un nuovo servizio di coordinazione che adoperi queste tecnologie garantirebbe sicuramente un alto livello di interoperabilità fra gli agenti comunicanti, i quali tendono a essere di natura molto varia in un sistema aperto, come già accennato. Un secondo passo, perciò, sarebbe volto a dare la possibilità di usufruire di spazi di tuple di diverso tipo, a seconda del linguaggio di comunicazione e, quindi, del rispettivo linguaggio di templating.

Il terzo e ultimo aspetto da affrontare è una diretta derivazione del precedente, ed è forse il più insidioso. Una volta messo in essere quanto esposto finora, l'ultimo passo verso la massima apertura e interoperabilità consiste nel superare la divisione "fisica" e linguistica dei diversi tipi di spazi di tuple, collassandoli in un'unica tipologia che riesca a supportare tutti i linguaggi *contemporaneamente*, traducendo le informazioni da un formato all'altro in modo trasparente agli agenti coinvolti nell'interazione; il tutto, possibilmente, sfruttando i meccanisimi di *content-negotiation* messi a disposizione dal protocollo HTTP. Questa funzionalità da una parte permetterebbe agli agenti di poter leggere e scrivere dati sugli spazi di tuple in qualsiasi formato essi desiderino, ma soprattutto fornirebbe la possibilità di interagire tramite la tecnologia a loro più comoda, lasciando così all'infrastruttura l'onere di gestire l'*overhead* computazionale che ne deriva.

Scopo di questa tesi è quindi presentare un nuovo servizio di coordinazione che prende il nome di *Tuple Spaces over the Web* (nel seguito TuSoW) e che cerca di porre delle solide basi per la risoluzione delle problematiche finora descritte. TuSoW si presenta a tutti gli effetti come un'implementazione del modello originale di Linda, del tutto aderente al meta-modello *Coordination as a Service* introdotto in [12] che ne descrive formalmente una possibile semantica. Esso tuttavia include anche alcune primitive aggiuntive rispetto al modello Linda, così come originariamente concepito da Gelernter [7] proposte

a più riprese in letteratura e, prendendo spunto da LoTuS [9], supporta anche più linguaggi di rappresentazione e templating simultaneamente. Verranno definiti, implementati e descritti, oltre che il suo funzionamento interno, la ReST API da utilizzare per interfacciarsi con TuSoW, nonché la sua architettura e il suo comportamento.

Di seguito è possibile trovare un breve riassunto del contenuto dei capitoli costituenti questo volume di tesi.

Capitolo 1 In questo capitolo viene analizzato lo stato dell'arte delle tecnologie che riguardano il supporto a più linguaggi di comunicazione, e di conseguenza dell'espressività dei corrispondenti linguaggi di templating, proponendo una descrizione formale della politica di matching che sia in grado di gestire la coesistenza dei vari formati di rappresentazione, e che garantisca l'ortogonalità fra i linguaggi di comunicazione e il linguaggio di coordinazione.

Capitolo 2 In questo capitolo viene fornito un mapping concettuale con i framework formali esistenti, per poi giungere all'enunciazione dei requisiti del sistema e alla loro analisi.

Capitolo 3 In questo capitolo viene descritta l'architettura del sistema realizzato, con una successiva trattazione specifica sul design dei singoli moduli software che lo compongono.

Conclusioni e sviluppi futuri In quest'ultima parte, vengono tratte le conclusioni sul lavoro svolto, e sono riportati alcuni possibili sviluppi futuri mirati al completamento e all'estensione del sistema realizzato.

Capitolo 1

Stato dell'arte

In questo capitolo sono riportati alcuni contributi passati significativi ai fini di questo elaborato di tesi.

Nella prima sezione verranno affrontati i concetti fondamentali del modello LINDA [7], ossia il meccanismo di comunicazione generativa e il funzionamento di alcune primitive. Nella seconda sezione si parlerà di *Coordination as a Service* [12] e dei benefici che tale framework é in grado di apportare nella progettazione delle moderne applicazioni distribuite. Nella terza e ultima sezione, invece, si parlerà brevemente dell'ortogonalità fra comunicazione e interazione; verranno inoltre accennate le caratteristiche base del modello LOTUS [9].

1.1 Tuple-based coordination: il modello Linda

Il modello di coordinazione LINDA viene presentato a metà degli anni Ottanta ad opera di David Gelernter [7] della Yale University, in collaborazione con Nicholas Carriero - ricercatore della medesima università - e Sudhir Ahuja dei Bell Laboratories.

LINDA nasce nel mondo del calcolo parallelo per fornire agli sviluppatori di applicazioni distribuite la possibilità di gestire la comunicazione fra processi tramite l'ausilio di un vero e proprio linguaggio distribuito, piuttosto che sfruttare opportune chiamate di sistema offerte dai singoli linguaggi di programmazione in cui gli stessi programmi venivano scritti. Era opinione comune, infatti, che programmare la coordinazione fra calcolatori in un linguaggio distribuito permettesse una maggiore espressività.

In quegli anni LINDA non era l'unico linguaggio distribuito presente in letteratura, anzi ne esistevano diversi ognuno con le sue peculiarità.

Questi linguaggi soffrivano, al fine del loro funzionamento, della necessità di un *runtime communication system* che gestisse l'invio e la ricezione dei messaggi scambiati fra i vari processi, poichè questi ultimi lavoravano su spazi di indirizzamento *separati*. Il punto di forza di LINDA consisteva proprio nel proporre un modello di coordinazione che fosse completamente distribuito sia a livello spaziale che a livello temporale, rendendo quindi vana la presenza di un'infrastruttura sottostante che interfacciasse gli spazi di memoria fra di loro.

1.1.1 Comunicazione generativa

Gli spazi di tuple. Il modello di comunicazione di LINDA si fonda sulla presenza di un ambiente computazionale astratto che prende il nome di **spazio di tuple**. Gli spazi di tuple aderiscono al paradigma della memoria associativa, secondo il quale i dati presenti in una memoria condivisa devono essere letti non tramite una qualche nozione di nome o indirizzo, bensì specificando una parte del loro contenuto come chiave di ricerca. Un programma LINDA, quindi, consiste in null'altro che una successione di letture, rimozioni e scritture su uno spazio di tuple.

Generalmente, le tuple possiedono due proprietà fondamentali:

- Le tuple sono **eterogenee**, cioè gli elementi che contengono possono essere di tipi di dato diversi;

- Le tuple sono **ordinate**, ovvero gli elementi che contengono appaiono in un ordine ben preciso.

In LINDA, le tuple godono di una terza proprietà: esse possiedono un'*esistenza indipendente* all'interno dello spazio che le contiene. Il modello di comunicazione di LINDA, infatti, è detto *generativo* proprio perchè una volta create le tuple non sono legate all'esecuzione di alcun processo; l'accesso a esse, al contrario, è permesso ugualmente a tutti coloro che partecipano all'interazione. E' possibile quindi immaginare uno spazio di tuple come una lavagna condivisa, in cui i processi possono condividere messaggi, variabili e codice.

Finora si è voluta fornire un'introduzione concettuale al modello di comunicazione LINDA, esplorando brevemente la visione innovativa alla base di questa tecnologia. Nel seguito verranno mostrate in dettaglio le operazioni base, o meglio, le *primitive di comunicazione* introdotte contestualmente al concepimento stesso del paradigma di comunicazione generativa, usate dai processi per interagire con e per mezzo di uno spazio di tuple.

1.1.2 Primitive base

Le primitive di comunicazione essenziali originariamente introdotte da Gelernter sono le seguenti:

`in(<template>)` Consuma una tupla che corrisponde al template indicato dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta è *sospesa* fino al suo inserimento.

`rd(<template>)` Legge, senza consumarla, una tupla che corrisponde al template indicato dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta è *sospesa* fino al suo inserimento.

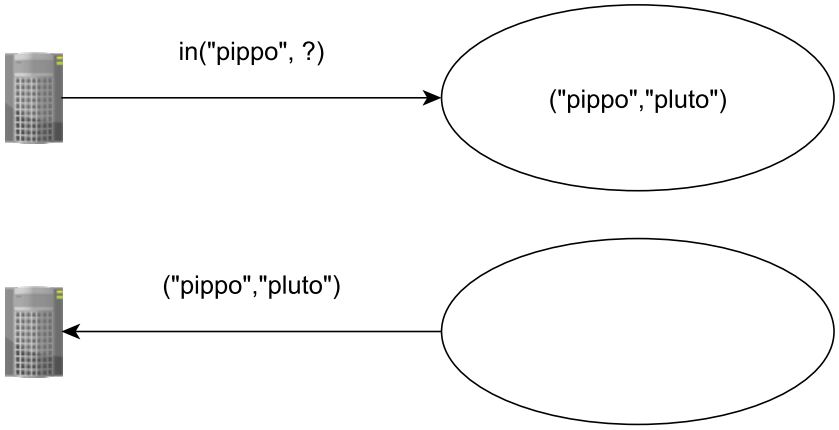
`out(<tuple>)` Inserisce una nuova tupla nello spazio di tuple.

I template – in letteratura spesso chiamati anche *anti-tuple* – rappresentano di fatto lo strumento tramite il quale è possibile selezionare i dati presenti in uno spazio di tuple. Come abbiamo detto in precedenza, gli spazi di tuple sono un meccanismo di memoria associativa: lo scopo dei template è proprio contenere una conoscenza – totale o parziale – del dato che si vuole recuperare. Se le informazioni e la struttura del template fornito a un'operazione di lettura o scrittura combaciano con quelle di una qualche tupla dello spazio, quest'ultima potrà essere restituita come risposta. Un breve approfondimento sui meccanismi di matching verrà fornito nella sezione 1.3.

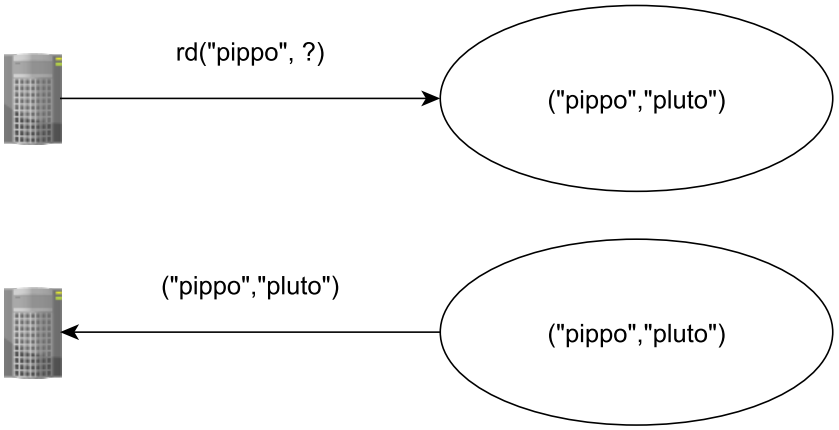
Nelle figure 1.1a, 1.1b e 1.1c é possibile trovare una rappresentazione del comportamento tipico delle primitive sopra esposte.

Semantica sospensiva. Come si può evincere dalle definizioni sopra riportate, le primitive di lettura e di rimozione di tuple seguono la cosiddetta *semantica sospensiva*. Ciò vuol dire che se non esiste alcuna tupla il cui contenuto aderisca al template fornito, la richiesta viene sospesa fino all'inserimento di una tupla aderente al suddetto template. Questo comportamento, naturalmente, non si manifesta durante la creazione di una nuova tupla - primitiva `out` - in quanto si assume che la tupla venga inserita nello spazio immediatamente dopo l'invocazione di tale operazione.

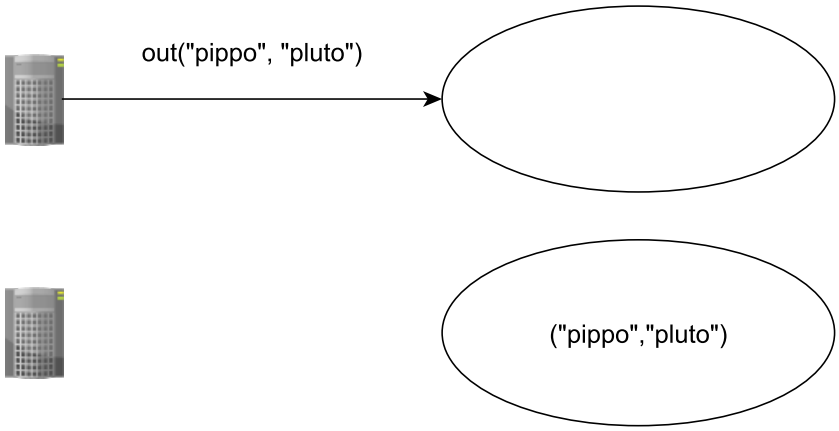
Non-determinismo. È necessario notare che le operazioni descritte possiedono generalmente un comportamento non-deterministico. Questo vuol dire che se consideriamo uno scenario dove due o più processi sono in comunicazione con uno spazio di tuple, non è possibile stabilire in alcun modo a priori se una operazione di `in` o `rd` andrà subito a buon fine o se si bloccherà in attesa della tupla desiderata. Supponiamo infatti di essere in una situazione in cui un certo numero di richieste pendenti di tipo `in` specificano lo stesso template,



(a) Raffigurazione del funzionamento della primitiva `in`.



(b) Raffigurazione del funzionamento della primitiva `rd`.



(c) Raffigurazione del funzionamento della primitiva `out`.

Figura 1.1: Raffigurazione del funzionamento delle primitive `in` (1.1a), `rd` (1.1b), `out` (1.1c). In ogni figura é mostrata sinteticamente l'interazione fra un agente (a sinistra) e uno spazio di tuple (a destra) mediante una sequenza di richieste e risposte.

e che nello spazio di tuple venga ad un certo punto inserita una sola tupla adatta a soddisfare tali richieste per mezzo della primitiva `out`. La semantica non deterministica di LINDA non permette di stabilire quale di esse verrà sbloccata.

Esaminiamo adesso un caso speculare al precedente, nel quale è dato uno spazio di tuple al cui interno sono presenti un certo numero di tuple tutte aderenti al medesimo template. In tale scenario, supponiamo venga invocata da un agente la primitiva `in` sul quel template. Analogamente, non è possibile stabilire quale delle suddette tuple verrà restituita come risultato della richiesta all'agente coinvolto.

Si tenga presente che il non-determinismo, in un modello di coordinazione, implica che l'implementatore del modello abbia **piena libertà** nel proporre una soluzione ai casi sopra esposti. Ogni implementazione di LINDA potrà quindi mostrare un comportamento differente dalle altre.

Semantica della primitiva out(). È possibile effettuare una distinzione fra due possibili semantiche [2] associate alla primitiva `out`:

1. **Semantica sincrona**: la generazione della tupla consiste in un'unica operazione, tramite la quale viene creata una nuova tupla;

2. **Semantica asincrona**: la generazione della tupla avviene tramite la creazione di un agente, il quale renderà disponibile la tupla dopo un lasso di tempo impredicibile.

Originariamente [2], tuttavia, questi due approcci vengono indicati come *ordered* e *unordered*. Questi due termini esprimono il fatto che, nel caso in cui l'operazione di output venga modellata secondo il primo approccio, le tuple generate vengono inserite nello spazio di tuple nello stesso ordine nel quale sono state invocate le rispettive primitive di output. Contrariamente, il secondo approccio presuppone che una volta che lo spazio di tuple riceve la richiesta di inserimento, questa non venga assolta direttamente, ma che sia necessario uno *step* interno aggiuntivo che permetta l'effettiva generazione della nuova tupla; questa operazione, essendo eseguita da un flusso di controllo indipendente, sarà completata dopo un lasso di tempo impredicibile.

I termini *sincrona* e *asincrona* sono qui usati per mostrare il punto di vista dell'agente che invoca la primitiva `out`. Nel primo caso, infatti, l'agente che vuole generare una tupla é in grado, una volta ricevuta una risposta di avvenuto inserimento, di sincronizzare il proprio flusso di controllo con quello dello spazio di tuple, assicurandosi quindi che la nuova tupla sia visibile fin da subito nello spazio. Nel secondo caso, invece, dopo l'invocazione della primitiva – e, eventualmente, dopo la conferma di ricezione della richiesta – l'agente e lo spazio di tuple continueranno ad agire in contesti temporali separati.

Ortogonalità comunicativa. La comunicazione generativa possiede un'importante caratteristica che è doveroso citare ai fini della comprensione di alcune sue proprietà fondamentali. Essa è nota come *communication orthogonality* [7], ed è basata sul fatto che i processi comunicanti fra di loro tramite uno spazio di tuple non si conoscono a vicenda: essi infatti possiedono solamente la percezione dei dati condivisi nello spazio, ma non è dato loro sapere la quantità e la natura dei processi che condividono lo spazio di memoria. Perciò, un processo che inserisca una tupla nello spazio non può sapere da chi verrà consumata: analogamente, un processo che tenti di rimuovere una tupla dallo spazio non è in grado di determinare chi è stato a generarla.

Questa particolarità della comunicazione generativa porta necessariamente alla formulazione di tre semplici ma importanti aspetti del modello di comunicazione:

1. **Disaccoppiamento spaziale**: una tupla generata da un qualsiasi agente può essere ricevuta da uno qualsiasi degli altri nodi in rete. Analogamente, una tupla ricevuta da un agente può essere stata creata all'interno di uno qualsiasi degli altri nodi in rete. Questo rende Linda un modello *completamente distribuito nello spazio*.

2. **Disaccoppiamento temporale**: a causa della comunicazione generativa, una tupla introdotta in uno spazio di tuple vi permane fino a che l'invocazione di una `in` non la rimuove. Ne segue che, concettualmente, una tupla può rimanere nello spazio di tuple per un tempo indefinito, per cui Linda rende possibile la comunicazione fra processi temporalmente disgiunti.[1]

3. **Condivisione distribuita**: quest'ultima proprietà deriva direttamente dalle due precedenti. Il disaccoppiamento spaziale e temporale dei processi fra di loro permette, infatti, di depositare una variabile condivisa nello spazio di tuple senza che un altro processo ne controlli l'atomicità degli accessi, in quanto essa viene automaticamente garantita dalle primitive Linda già discusse.

Mentre in questa in questa parte sono state mostrate le primitive originali del modello Linda, assieme a una concisa trattazione delle principali caratteristiche offerte da questo modello di comunicazione, nella prossima verrà introdotta una serie di primitive non contemplate nel modello originale, ma

[1] Questa è una caratteristica che molti linguaggi distribuiti generalmente non mettono a disposizione. Infatti, la maggior parte sono basati su modelli di tipo *message passing*, dove ogni messaggio viene consumato e letto dal destinatario in modo automatico. L'informazione, pertanto, permane in memoria condivisa per un tempo definito.

che sono state progressivamente introdotte in letteratura nel corso degli anni [10] [3].

1.1.3 Altre primitive

Di seguito verrà fornito un breve *excursus* di un insieme di primitive "derivate" dalle semplici `rd`, `in`, e `out`. Le più degne di nota, nonchè le più comuni, si dividono in due gruppi: le primitive *predicative* e le primitive *bulk*. Esse risultano utili quando si vuole effettuare una lettura o una rimozione *non bloccante*, o quando si ha la necessità di effettuare operazioni su più dati contemporaneamente.

Primitive predicative. Le primitive `rd` e `in`, come abbiamo visto, reificano la semantica sospensiva del modello LINDA: questo implica che il risultato dell'invocazione di tali primitive, in astratto, sia sempre un successo, considerato che una richiesta sospesa verrà, dopo un tempo indefinito, soddisfatta. Tale affermazione, naturalmente, ha senso in un'ottica puramente teorica. E' chiaro che nella pratica esistono infiniti motivi per i quali una richiesta non possa essere soddisfatta: la connessione potrebbe cadere, il processo potrebbe terminare prima del previsto, e così via.

Le primitive predicative prevedono che il risultato della richiesta possa essere un *successo* o un *fallimento* e che l'invocazione della primitiva non sia soggetta alla semantica sospensiva. Se la tupla richiesta è immediatamente presente nello spazio di tuple, essa viene restituita come risultato e l'invocazione della primitiva è risolta con un successo; in caso contrario, non viene restituito alcun risultato e la primitiva *fallisce*. Di seguito viene riportata la definizione delle primitive predicative:

`inp(<template>)` Consuma una tupla che corrisponde al template indicato dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta *fallisce*.

`rdp(<template>)` Legge, senza consumarla, una tupla che corrisponde al template indicato dallo spazio di tuple. Se la tupla richiesta non è immediatamente disponibile, la richiesta *fallisce*.

È naturale osservare come queste primitive non implementino la semantica sospensiva di LINDA. Difatti, l'invocazione di una di queste primitive da parte di un processo non ne blocca mai il flusso di controllo, qualsiasi sia il risultato della richiesta.

Risulta altresì superfluo evidenziare il fatto che la `out` non si veda rappresentata nell'insieme sopra riportato, dal momento che la generazione di una

tupla non produce di per sè altro risultato che non sia la presenza stessa del nuovo dato nello spazio di tuple. In altre parole, la `out` non risente della semantica sospensiva di LINDA, ovvero non è un'operazione bloccante: non è quindi possibile derivarne una versione predicativa in quanto ciò risulterebbe tautologico, almeno in via formale. Tuttavia, sarebbe possibile fare una distinzione fra generazione *sincrona* e *asincrona*, aspetto che al momento viene considerato un dettaglio implementativo e che merita certamente una discussione a parte.

Primitive bulk. Come illustrato precedentemente, le primitive `in` e `rd` permettono di recuperare sempre e solo un dato alla volta; analogamente, la `out` consente di inserire una sola tupla per invocazione.

Le primitive *bulk* estendono le capacità delle primitive originali permettendo ai processi di inserire, leggere o rimuovere atomicamente più tuple alla volta:

`in_all(<template>)` Consuma tutte le tuple corrispondenti al template indicato dallo spazio di tuple. Se nessuna tupla soddisfa il template indicato, viene restituita una lista vuota di tuple.

`rd_all(<template>)` Legge, senza rimuoverle, tutte le tuple corrispondenti al template indicato dallo spazio di tuple. Se nessuna tupla soddisfa il template indicato, viene restituita una lista vuota di tuple.

`out_all(<tuples>)` Inserisce un insieme di tuple nello spazio di tuple.

Sebbene le primitive `in_all` e `rd_all` non vengano di per sè classificate come predicative, neanch'esse risentono della semantica sospensiva di LINDA. Tuttavia, queste operazioni implementano una gestione differente dei fallimenti, ritornando come risultato un'insieme vuoto di tuple nel caso in cui il template specificato non individui alcuna tupla nello spazio.

1.2 Coordination as a service

In questa sezione verranno brevemente esplorati i concetti fondamentali affrontati nel framework *Coordination as a Service* [12] e i vantaggi apportati da una sua adozione nell'ambito dei sistemi aperti.

1.2.1 Framework concettuale

I modelli di coordinazione come LINDA hanno avuto origine nel contesto dei sistemi chiusi, come ad esempio gli scenari applicativi di calcolo parallelo. Storicamente, in tali contesti, la coordinazione è stata definita in funzione del

linguaggio usato per descrivere l'interazione nelle applicazioni concorrenti. È bene notare oltretutto, che le entità coinvolte nella coordinazione erano ben note in fase di progettazione; oltretutto, le entità che fornivano la coordinazione – i *coordination media* – venivano costruite e messe in essere su misura per ogni specifica applicazione. Questi modelli di coordinazione sono stati, nel tempo, oggetto di formalizzazione tramite, per esempio, l'algebra dei processi, fondamentalmente nella stessa maniera in cui viene specificata la semantica dei linguaggi concorrenti. Questo particolare punto di vista ha preso il nome di *coordination as a language*.

In tempi più recenti, tuttavia, si è sentita fortemente l'esigenza di un cambio di approccio. I sistemi moderni hanno acquisito una complessità tale che si è resa necessaria la modellazione non solo del linguaggio di comunicazione in sè, ma anche e soprattutto delle *entità* protagoniste della coordinazione. Questo bisogno, di fatto, si è tradotto nella formulazione di nuove astrazioni che potessero descrivere queste entità in modo formale, in modo da costruire un solido modello di pensiero che potesse essere adottato lungo tutte le fasi del processo ingegneristico volto alla costruzione di sistemi paralleli moderni. La sistematicità di un tale approccio risulta oggi estremamente benefica, in quanto spesso si ha a che fare con situazioni in cui molte applicazioni eterogenee devono comunicare tramite un singolo ambiente comune, il che porta a *fattorizzare* le esigenze comuni di suddette applicazioni, in modo da renderle efficaci e affidabili. In questi termini, è naturale che un'applicazione, una volta scelta l'infrastruttura di coordinazione – il *servizio di coordinazione* – ne accetti e sfrutti le leggi intrinseche dei suoi mezzi di coordinazione.

È tramite queste premesse che nasce *Coordination as a Service* [12], di cui sono stati appena descritti gli obiettivi concettuali. Nella prossima sezione verrà data una breve spiegazione dei componenti architetturali di questo modello: le *coordinated entities*, l'*interaction space* e i *coordination media*.

1.2.2 Ontologia

Un *sistema coordinato* è un sistema che adotta un particolare modello di coordinazione. I sistemi coordinati sono costituiti da tre parti fondamentali separate fra di loro: il *coordinated space*, l'*interaction space* e il *coordination space*. Ognuno di questi sottosistemi comprende una serie di entità dal comportamento ben definito, ognuna caratterizzata da un'astrazione di design ben definita.

In figura 1.2 è possibile trovare una rappresentazione dell'architettura logica dei sistemi coordinati, come visto in [12]:

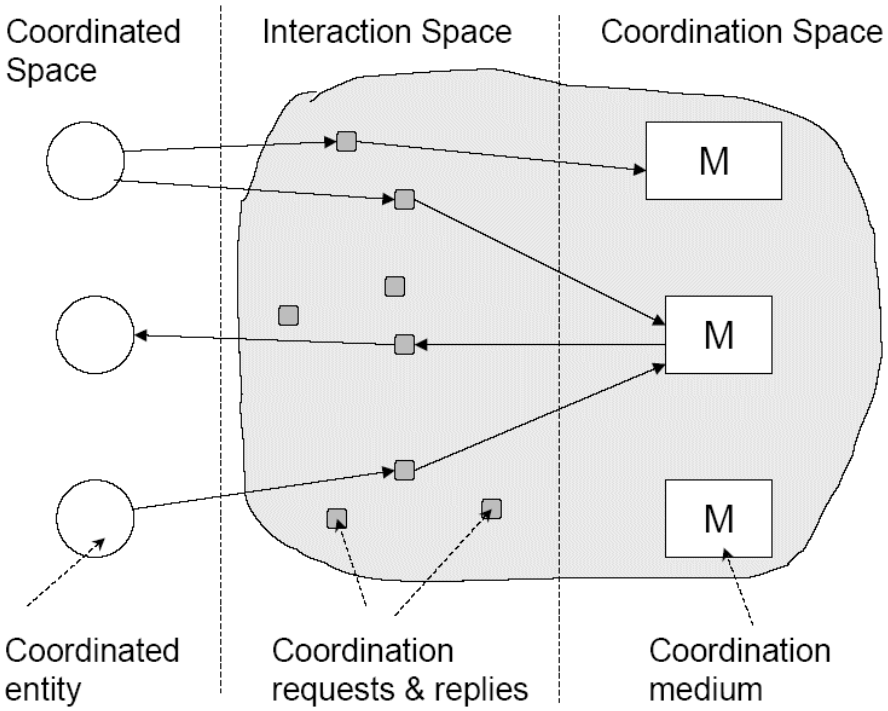


Figura 1.2: Architettura logica dei sistemi coordinati [12].

Coordinated space. Per spazio coordinato si intende quella parte del sistema le cui entità – le *coordinated entities* – usufruiscono dei servizi di coordinazione messi a loro disposizione. Le interazioni fra queste entità vengono regolate dal modello di coordinazione adottato e, in generale, risultano modellate in termini di *coordination requests*, *coordination replies* o *silent actions*, ovvero computazioni interne ai singoli agenti.

La nozione di **agente** verrà, da qui in poi, usata come sinonimo di processo per fare fede alla complessità dei sistemi aperti attuali. Per definizioni più complete, invitiamo a far riferimento all'ampia letteratura a riguardo.

Dal punto di vista di questi ultimi, dunque, l'interazione è vista come un susseguirsi di richieste e risposte scambiate con l'ambiente. È proprio tramite questo "ambiente" che può avvenire la comunicazione fra agenti; è inoltre l'unica cosa che essi possano percepire come entità esterna, il che rende di fatto la comunicazione fra processi diversi un meccanismo *indiretto*. Gli agenti non hanno modo di comunicare fra di loro, ma devono necessariamente ricorrere alla mediazione del servizio di coordinazione.

Oltretutto, è opportuno tenere in considerazione il fatto che l'insieme delle *coordinated entities* **non deve** essere fissato a priori. L'obiettivo di questo framework, ricordiamo, è di definire un modello di servizio di coordinazione che si adatti ai sistemi aperti, nei quali le entità richiedenti il servizio possono

entrare e lasciare lo spazio coordinato in modo dinamico.

Lo spazio delle interazioni. Lo spazio delle interazioni rappresenta l'ambiente presentato precedentemente come l'unico canale di comunicazione disponibile agli agenti. Il suo compito è semplice: tradurre le "azioni" di coordinazione eseguite dalle altre entità del sistema in *eventi* di comunicazione. Come abbiamo già accennato, questi eventi possono essere sostanzialmente di due tipi: *request events* e *reply events*. Se li osserviamo dal punto di vista delle *coordinated entities*, le *request* hanno come destinazione lo spazio delle interazioni, mentre le *reply*, essendo generate in tale spazio, hanno come destinatario gli agenti stessi, e quindi possono essere consumati da essi.

È semplice notare come una *reply* sia diretta a un unico agente, che in precedenza avrà presumibilmente inoltrato una richiesta relativa.

I media di coordinazione. I media di coordinazione sono quelle entità all'interno delle quali avviene l'attività di coordinazione. Il loro compito principale è quello di consumare le *request* provenienti dallo spazio delle interazioni e, conseguentemente, produrre altri eventi in risposta. Essi possono essere delle *reply* che verranno consumate dalle entità coordinate, o ulteriori *reply* diretti agli altri media di coordinazione presenti nel sistema.

Verso la coordinazione sul Web. Il framework finora discusso ci fornisce una semplice ma potente formulazione di un modello di progettazione dei sistemi di coordinazione, seguendo il filo conduttore denominato *Coordination as a Service*.

Una volta riconosciuta la potenza delle astrazioni fornite, appare chiaro [12] come la continuazione naturale di questo lavoro debba necessariamente trovare applicazione nel campo dei **servizi web**. Infatti, se concepiamo la coordinazione come un processo interattivo basato sullo scambio di richieste e risposte, essa risulta immediatamente traducibile in modello infrastrutturale applicabile nell'ambito dei servizi web.

Oggigiorno, peraltro, con un impiego sempre più massivo dei sistemi aperti, é difficile pensare a un servizio di coordinazione che non sfrutti lo spettro tecnologico disponibile in ambito Web. L'impiego dei framework formali esistenti e delle astrazioni che forniscono [12], unito all'interoperabilità resa possibile da tecnologie potenti e consolidate come il linguaggio JavaScript, il formato di rappresentazione JSON e lo stesso protocollo HTTP, permetterebbero infatti di costruire un prodotto software sicuramente altamente competitivo e adottabile in un ampio ventaglio di applicazioni.

1.3 Ortogonalità fra comunicazione e interazione

Nella sezione precedente si é visto come sia possibile suddividere lo spazio di coordinazione per suddividerne e incapsularne le funzioni. In questa sezione, invece, sarà posto l'accento sulle differenze che corrono fra comunicazione e interazione, e verranno evidenziate alcune interessanti intuizioni e proprietà derivate da esse.

Verrà considerata come filo conduttore l'analisi proposta in [9].

1.3.1 Linguaggio di comunicazione

Il linguaggio di comunicazione viene usato dalle entità coordinate per rappresentare l'informazione che viene inviata e ricevuta. Un modello di coordinazione, tipicamente, definisce il linguaggio di comunicazione unicamente in termini di *sintassi*; si assume generalmente che una certa semantica venga definita in modo condiviso dalle entità coordinate, in quanto essa rappresenta di per sé l'aspetto chiave sui cui si basa la comunicazione stessa.

Per esempio, in alcune implementazioni di LINDA [11] si é usato XML come linguaggio di comunicazione, mentre in altre, come ad esempio lo stesso LOTUS [5] [9] si usa il linguaggio della logica del primo ordine.

Il modello LOTUS risulta di grande importanza nella misura in cui esso stabilisce una chiara indipendenza fra le dimensioni di comunicazione e coordinazione. Difatti, la distinzione che si crea fra primitive di coordinazione e linguaggio di comunicazione si reifica proprio nel concetto di predicato, il quale permette di astrarre il concetto di *matching*, rendendolo ben separato dai meccanismi di interazione, e che quindi permette di far variare i linguaggi di comunicazione indipendentemente dai meccanismi di coordinazione.

Naturalmente, il solo fatto di poter dare una definizione autonoma della comunicazione ci permette di separarla dalla coordinazione. Difatti, lo stesso linguaggio di coordinazione può quindi essere usato con diversi linguaggi di comunicazione.

1.3.2 Linguaggio di coordinazione

Il linguaggio di coordinazione, contrariamente al linguaggio di comunicazione, va definito in funzione sia della sintassi che della semantica. Inoltre, siccome esso riguarda l'atto della comunicazione, risulta strettamente legato al linguaggio di comunicazione.

A livello sintattico, il linguaggio di coordinazione associa un insieme di primitive al linguaggio di comunicazione, mentre a livello semantico mette in

relazione ogni primitiva di comunicazione con un insieme di eventi di input/output. Ad esempio, le primitive presentate nelle sottosezioni precedenti costituiscono, per il modello LINDA, il linguaggio di coordinazione.

La semantica del linguaggio di coordinazione assume sicuramente un ruolo fondamentale, in quanto essa generalmente non può essere estratta dalla semplice sintassi di tale linguaggio. Ad esempio, nel modello LINDA la sintassi delle primitive non fornisce alcun indizio sulla semantica sospensiva che le caratterizza.

Se rievochiamo quanto detto nella 1.2.2, dove viene constatato che l'interazione può essere espressa come una sequenza di eventi di *request* e *reply*, ogni primitiva di comunicazione ammissibile può quindi essere definita in termini degli effetti che provoca sull'*interaction space*, piuttosto che nei termini del rispettivo comportamento del *coordination medium*.

Ne segue che, disponendo dell'astrazione dell'evento di comunicazione come "ponte" fra i due linguaggi, l'invocazione di una primitiva generalmente può essere considerata *separata* dalla sua esecuzione, la quale risulta a carico del *coordination medium*.

Deduciamo, quindi, che le *coordinated entities* e i *coordination media* possono essere modellati in modo *indipendente* come **processi asincroni**. Questo permette di semplificare la progettazione dei *run-time* di coordinazione per i sistemi aperti; inoltre, consente un maggiore supporto all'eterogeneità delle entità, poichè ognuna di esse può essere implementata separatamente dalle altre.

1.3.3 LoTuS

In [9] viene introdotto per la prima volta un modello di coordinazione di nome LOTUS (Logic Tuple Spaces). LOTUS é un modello che adotta le tuple Prolog come linguaggio di comunicazione attraverso diversi spazi di tuple. La sua sintassi é basata sulla logica del primo ordine, ma in particolare adotta le convenzioni sintattiche tipiche di Prolog.

In breve, in LOTUS il **linguaggio di tuple** coincide con l'insieme delle *ground atomic formulae* (formule logiche senza variabili), mentre il **linguaggio di template** coincide con l'insieme delle *atomic formulae*. Il *matching* fra tuple e template viene modellato per mezzo di un **predicato** che, in questo modello particolare, coincide con l'operazione di unificazione.

Si noti bene che il predicato rappresenta qui il punto critico del meccanismo di memoria associativa, in quanto solo esso é in grado di mettere in relazione un template con un certo insieme di tuple compatibili. Ogni predicato rappresenta una diversa politica di *matching*, e ognuna di esse può identificare diversi tipi di spazi di tuple.

Ai fini di questa trattazione non sono riportate definizioni formali: per maggiori dettagli, si veda [9].

Un'ulteriore aspetto ritenuto di particolare rilevanza riguarda la cosiddetta *coordination transparency*. Nel modello LoTuS, l'interazione fra entità – siano esse spazi di tuple o entità attive – può avvenire a discapito dell'effettiva esistenza di tali entità. Questo ci porta a fare un paio di considerazioni: i) non sono necessarie primitive di meta-coordinazione per gestire la creazione delle entità, e ii) le entità coordinate non hanno il bisogno di assicurarsi che uno spazio di tuple esista prima di usarlo. Tali proprietà risultano estremamente utili nella costruzione di sistemi aperti, nei quali é necessario che i dispositivi che vogliono accedere al servizio offerto possano godere di una visione il più possibile concisa e fluida delle funzionalità messe a disposizione; questo, naturalmente, é raggiungibile ancor più grazie a un'interfaccia minimale, che riduca al minimo le operazioni disponibili, assicurando però un'interazione efficace e significativa.

Le considerazioni effettuate in questa sezione si sono rivelate particolarmente utili durante il processo di formulazione e costruzione del middleware TuSoW, di cui si parlerà nel capitolo 2.

Capitolo 2

Tuple Spaces over the Web

Nel capitolo precedente si é condotta un'analisi volta a considerare i contributi tecnologici passati nell'ambito della *tuple-based coordination* [3][4][7][9][10][12].

Abbiamo descritto le caratteristiche fondamentali del modello LINDA, descrivendone succintamente la nascita e le astrazioni di base su cui si fonda; abbiamo considerato le primitive di comunicazione messe a disposizione da questo modello, nonché alcune altre introdotte in letteratura nel corso del tempo.

Abbiamo parlato del framework *Coordination as a Service*, di come la transizione da "coordinazione come linguaggio" a "coordinazione come servizio" costituisca un *paradigm shift* essenziale per la progettazione di sistemi distribuiti moderni ed eterogenei. Abbiamo esplorato l'ontologia dei sistemi coordinati, e abbiamo riportato la visione che propone l'integrazione dei servizi di coordinazione nel mondo dei servizi web.

Inoltre, è stato presentato un brevissimo *excursus* sulle differenze presenti tra i linguaggi di comunicazione e di coordinazione. In aggiunta, é stato spiegato come le entità dei sistemi coordinati possano essere espresse come processi asincroni; é stato ribadita l'intuizione di definire il matching fra tuple e template sulla base di un predicato, e della rispettiva relazione con il linguaggio Prolog.

Scopo di questo capitolo, quindi, è presentare un *mapping* concettuale di tutte le intuizioni precedentemente riportate con le funzionalità esposte da questo progetto di tesi. Esso prende il nome di *Tuple Spaces over the Web* (da qui in poi abbreviato in TUSOW), un moderno middleware distribuito basato su LINDA.

Come vedremo, verranno dapprima delineati gli elementi "centrali" del sistema, cioè gli spazi di tuple (i *coordination media*). Verrà sottolineata la loro capacità di supportare più linguaggi di comunicazione contemporaneamente, e nondimeno verrà posta particolare considerazione sul loro comportamento

asincrono. Oltretutto, verranno formulate alcune nuove primitive di comunicazione che, a nostro avviso, potrebbero considerarsi un valore aggiunto al servizio.

Successivamente, saranno definite le caratteristiche di un servizio web che permetta l'interazione fra diversi dispositivi; gli spazi di tuple saranno modellati come risorse web, il che porterà a definire una API ReST, proponendo un *mapping* delle primitive alla suddetta interfaccia e reificando gli intenti originali di *Coordination as a Service*.

Per ultimo, verranno ipotizzate le funzionalità di un client a riga di comando che sia in grado di interfacciarsi con un qualunque spazio di tuple visibile sul web.

2.1 Funzionalità base

In questa sezione saranno formulate le caratteristiche principali del middleware TuSoW. In particolare, verranno dapprima esplorate le proprietà principali degli spazi di tuple, e si cercherà di definire una relazione fra essi, il modello Linda e l'ontologia dei sistemi coordinati; in seguito verranno introdotte nuove primitive di coordinazione; infine verrà affrontato il concetto di poliglottismo, dapprima tramite una panoramica di alcune tecnologie di comunicazione, per poi giungere a una definizione formale di un meccanismo di matching.

2.1.1 Spazi di tuple

Gli spazi di tuple sono, secondo l'ontologia dei sistemi coordinati, dei *coordination media*, ovvero quelle entità che forniscono di fatto il servizio di coordinazione, consumando le richieste delle altre entità e producendo delle risposte.

Il modulo software principale di TuSoW fornisce una semplice astrazione del concetto di spazio di tuple, che da qui in poi chiameremo con il nome di *dataspace*.

Comportamento asincrono. Ogni *dataspace* fornisce la possibilità di interagirvi mettendo a disposizione una semplice API composta dalle primitive di comunicazione.

Naturalmente, siccome un *dataspace* potrebbe essere chiamato a gestire un numero elevato di richieste su una quantità altrettanto sostanziosa di dati, esso fornisce un servizio dalla natura intrinsecamente asincrona, in modo da garantire un'interazione *reattiva* e mai bloccante. Ciò significa che chiunque si

trovi ad invocare una delle primitive di comunicazione si dovrà aspettare una risposta possibilmente ritardata nel tempo.

Questo è un aspetto fondamentale del middleware proposto, poichè reifica la filosofia secondo cui le *coordinated entities* e i *coordination media* vadano modellati come processi separati; l'invocazione di una primitiva da parte di un agente, perciò, è da considerarsi temporalmente disgiunta dalla sua esecuzione all'interno del *medium* di coordinazione.

Non-determinismo. Nella sezione 1.1.2 è stata ricordata il comportamento non-deterministico del modello LINDA. Questo significa che il modello LINDA lascia decidere alle sue implementazioni l'ordine di gestione da parte degli spazi di tuple delle richieste in arrivo.

In TUSOW, come originariamente stabilito in LINDA [7], si é deciso di non seguire un ordine particolare nella gestione delle richieste. Gli agenti connessi al servizio, perciò, non potranno fare alcuna assunzione su quali richieste verranno soddisfatte prima di altre, poichè il servizio stesso selezionerà di volta in volta richieste totalmente casuali. Collocandosi nell'ambito dei sistemi aperti, infatti, TUSOW non rispetta alcun criterio di precedenza, mirando all'obiettivo di fornire un servizio di coordinazione il più possibile trasparente.

Relazione con l'ontologia. Ricordiamo che all'interno dell'ontologia dei sistemi coordinati esistono tre elementi fondamentali: le *coordinated entities*, l'*interaction space* e i *coordination media*.

Siccome l'obiettivo di questo capitolo è definire le funzionalità messe a disposizione dal servizio TUSOW, non andremo a soffermarci sul concetto di *coordinated entity*, tenendo tuttavia bene a mente – ribadiamo – che gli agenti che useranno il servizio potranno essere fortemente eterogenei. Rimandiamo il lettore alla sezione 2.1.3 per maggiori dettagli sulla possibilità dei *dataspace* di supportare più linguaggi.

Per quanto riguarda l'*interaction space*, esso si é fondamentalmente concretizzato – come si vedrà nel capitolo 3 – nelle astrazioni che modellano i concetti di richiesta e risposta all'interno del modulo software relativo al servizio web. Sarà proprio il servizio web, infatti, ad interfacciarsi con gli spazi di tuple, permettendo quindi l'orchestrazione delle richieste e delle risposte scambiate fra agenti e *coordination media*.

2.1.2 Nuove primitive

Nel capitolo 1 é stata effettuata una discussione sul modello LINDA e sul suo funzionamento; in particolare, si é visto come la comunicazione avvenga tramite l'invocazione di un insieme di primitive di comunicazione.

Ricordiamo che non tutte le primitive discusse sono state introdotte contestualmente alla presentazione del modello: molte, infatti, sono comparse in letteratura negli anni a seguire, in risposta a esigenze molto varie.

In TuSoW si é deciso di includere un piccolo insieme di primitive nuove, oltre alle primitive originali, le primitive *predicative* e le primitive *bulk*. Queste primitive sono da considerarsi di *utility*, nel senso che non fanno altro che generalizzare alcuni aspetti delle primitive originali: alcune, ad esempio, permettono di leggere/rimuovere dati specificando più di un template, mentre altre permettono semplicemente di recuperare e/o sostituire tutti i dati di uno spazio di tuple in modo atomico.

Viene fatto presente che il servizio TuSoW adotta una convenzione sui nomi delle primitive differente da quella comunemente adottata in letteratura. Da ora in poi, quindi, é bene fare fede al seguente *mapping*:

$$\texttt{rd} \mapsto \texttt{read}$$
$$\texttt{in} \mapsto \texttt{take}$$
$$\texttt{out} \mapsto \texttt{put}$$

Nell'elenco sottostante é quindi possibile trovare il nome e la descrizione del comportamento di ogni nuova primitiva introdotta in TuSoW.

`readAny(<template1>, <template2>, ..., <templateN>)` Legge, senza consumarla, una tupla che corrisponde a uno *qualunque* dei template indicati dallo spazio di tuple. Se al momento dell'invocazione non esistono tuple che soddisfino *almeno* uno dei template, la richiesta è *sospesa* fino al suo inserimento.

`takeAny(<template1>, <template2>, ..., <templateN>)` Consuma una tupla che corrisponde a uno *qualunque* dei template indicati dallo spazio di tuple. Se al momento dell'invocazione non esistono tuple che soddisfino *almeno* uno dei template, la richiesta è *sospesa* fino al suo inserimento.

`readAll(<template1>, <template2>, ..., <templateN>)` Legge, senza consumarla, una tupla che corrisponde a *tutti* i template indicati dallo spazio di tuple. Se al momento dell'invocazione non esistono tuple che soddisfino *tutti* i template, la richiesta è *sospesa* fino al suo inserimento.

`takeAll(<template1>, <template2>, ..., <templateN>)` Consuma una tupla che corrisponde a *tutti* i template indicati dallo spazio di tuple. Se al momento dell'invocazione non esistono tuple che soddisfino *tutti* i template, la richiesta è *sospesa* fino al suo inserimento.

`set(<tuple1>, <tuple2>, ..., <tupleN>)` Rimpiazza tutte le tuple dello spazio con la lista di tuple indicata.

`get()` Legge, senza consumarle, tutte le tuple dallo spazio di tuple. Nel caso in cui lo spazio di tuple sia vuoto, come risultato viene restituita una lista vuota.

`clear()` Consuma tutte le tuple dallo spazio di tuple. Nel caso in cui lo spazio di tuple sia vuoto, come risultato viene restituita una lista vuota.

Si noti che tutte le primitive sopra elencate implementano la semantica sospensiva di LINDA, fatta eccezione per le `set()`, `get()` e `clear()`.

2.1.3 Poliglottismo

Nel capitolo 1 sono stati riportati alcuni contributi significativi nel campo della *tuple-based coordination*; in particolare, è stato dato risalto alla filosofia *Coordination as a Service* e al modello LoTuS. Nella concezione del servizio di coordinazione TuSoW si é scelto di dare una grande importanza a questi ultimi due aspetti, con l'obiettivo di proporre un'infrastruttura che sia in grado di essere impiegata a supporto di una vasta gamma di applicazioni. Naturalmente, gli scenari applicativi distribuiti moderni prevedono la partecipazione di un grande numero di agenti, spesso implementati con tecnologie molto diverse fra di loro.

Perciò, é ragionevole pensare che un middleware moderno debba offrire, prima di ogni cosa, **interoperabilità**. Nel caso di TuSoW, si é deciso di reificare questa caratteristica facendo in modo che gli agenti coinvolti nell'interazione possano trasmettere – e richiedere – dati nel formato che più preferiscono.

È naturale che la progettazione di una tale funzionalità possa sollevare alcune questioni sulla sua fattibilità. Per esempio, quanti e quali linguaggi di tuple saranno supportati? È possibile renderli intercambiabili? E se sì, é possibile farlo senza causare perdita d'informazione?

Queste sono tutte domande ovviamente legittime, alle quali si cercherà, almeno in parte, di dare una risposta. La discussione seguente sarà limitata all'enumerazione di un insieme ben limitato di tecnologie di comunicazione e *templating* esemplificative (ma non per questo meno significative), e verrà definito formalmente un semplice meccanismo di traduzione e *matching* dotato, a nostro avviso, di un ragionevole grado di applicabilità.

I linguaggi di tuple supportati. Una scelta oculata delle tecnologie di comunicazione non può non tenere conto della popolarità di cui godono attualmente alcuni linguaggi usati da migliaia di applicazioni per la trasmissione

dei dati su rete. Considerata l'ubiquità dei servizi web, naturalmente non può che emergere il linguaggio **JSON**[1], usato non solo per lo scambio di dati nelle comuni applicazioni di tipo client/server, ma anche per la persistenza, come in MongoDB[2].

Oltre a JSON, si é pensato di adottare il linguaggio **Prolog** al fine di implementare il modello di coordinazione LoTuS precedentemente discusso nella sottosezione 1.3.

Per ultimo, si é deciso di usare il semplice **formato testuale**, poiché sono stati ipotizzati scenari in cui la coordinazione consiste nella produzione e nella consumazione di contenuti testuali.

I linguaggi di template supportati. Parallelamente alla selezione dei linguaggi di tuple, é stato necessario individuare, per ognuno di essi, un rispettivo linguaggio di template. Come sappiamo, infatti, la ricerca delle informazioni all'interno degli spazi di tuple avviene specificando una conoscenza parziale del dato interessato come argomento della primitiva (generalmente di tipo `rd()` o `in()`). Questa conoscenza – il *template*, appunto – permette di descrivere una famiglia (o meglio, un *insieme*) di tuple che potranno comparire come risultato della richiesta di lettura/rimozione.

Per il linguaggio Prolog e per il formato testuale, la scelta si é rivelata abbastanza obbligata. Per il primo dei due si é scelto lo stesso **Prolog** come linguaggio di template, con l'*unificazione* come meccanismo di *matching*, attenendosi di fatto alle specifiche del modello LoTuS. Per il secondo, la scelta é ricaduta sulle **espressioni regolari** – abbr. *regex* – quale strumento dichiarativo per il riconoscimento dei linguaggi regolari.

Per il linguaggio JSON, invece, é stato svolto un lavoro di ricerca che ha portato ad individuare, fra le tante, la libreria **JsonPath**[3], ispirata dalla famosa libreria XPath per l'esecuzione di query su dati in formato XML.

Nella tabella 2.1 é possibile trovare un riepilogo delle tecnologie di comunicazione supportate, associate ognuna al rispettivo linguaggio di templating.

Tuple	Template
JSON	JsonPath
Prolog	Prolog
stringhe	regex

Tabella 2.1: Tabella raffigurante le tecnologie di comunicazione supportate in TuSoW.

[1] https://www.json.org/
[2] https://www.mongodb.com/
[3] https://github.com/json-path/JsonPath

Di seguito sono riportati alcuni esempi di rappresentazione di tuple e template tramite le tecnologie sopra citate.

JSON/JsonPath

Esempio di tupla: `{"name":"Lorenzo"}`

Template positivo: `$..[?(@.name == "Lorenzo")]`

Template negativo: `$..[?(@.name == "Giovanni")]`

Prolog

Esempio di tupla: `name(lorenzo)`

Template positivi: `name(lorenzo)`
`name(A)`

Template negativi: `name(giovanni)`
`name(giovanni, giorgio)`

Stringhe/regex

Esempio di tupla: `Lorenzo`

Template positivi: `.*?ore.*`
`.*?nz.*`

Template negativo: `.*?van.*`

Il meccanismo di traduzione. Una volta stabiliti i linguaggi di comunicazione, é necessario definire un meccanismo che ci permetta di tradurre le tuple contenute nel *dataspace* nei vari linguaggi. Come precedentemente esposto, il *focus* principale di TuSoW é quello di fornire interoperabilità alla stragrande maggioranza degli agenti che dovessero connettersi al servizio. Per farlo si é deciso di fare in modo che i suddetti agenti possano comunicare utilizzando il linguaggio – o i linguaggi – che preferiscono, a scelta fra un insieme di tecnologie considerate di uso comune.

Questo si rende possibile nelle seguenti condizioni:

- Gli agenti devono essere in grado di inserire nello spazio di tuple dati scritti in qualsiasi linguaggio di tuple supportato;
- Gli agenti devono essere in grado di richiedere dati allo spazio di tuple in qualsiasi formato supportato.

Inoltre, é necessario che eventuali traduzioni effettuate risultino il più possibile trasparenti agli utenti del servizio; ciò, naturalmente, ha a che fare con la natura aperta dei sistemi a cui TuSoW fa riferimento.

Perciò, si è deciso di operare nel seguente modo: in risposta all'inserimento di tuple non verrà effettuata alcuna traduzione. Questo, naturalmente, implica che le tuple saranno salvate nello spazio nello stesso formato in cui erano originariamente scritte.

In risposta a letture e rimozioni, invece, lo spazio di tuple cercherà di tradurre le tuple in esso contenute nel formato richiesto, ove possibile; dopodichè verrà effettuato normalmente il matching tramite il template fornito, e si cercherà di completare la richiesta.

In entrambi i casi, tuttavia, sarà necessario specificare un formato che indichi la coppia linguaggio di tuple/linguaggio di template da considerare ai fini del matching. Nel caso delle letture, la causa é banale: l'agente dovrà infatti fare presente al servizio il formato in cui desidera il dato, formato al quale dovrà ovviamente corrispondere quello del template fornito. Nel caso degli inserimenti, similmente, bisognerà specificare il formato in cui il dato fornito è rappresentato. Quest'ultimo accorgimento risulta particolarmente importante in quanto permette di preservare la semantica sospensiva delle primitive Linda. Se così non fosse, infatti, l'inserimento non sarebbe in grado di sbloccare alcuna primitiva sospesa, in quanto non si conoscerebbe il formato del dato inserito, e non sarebbe possibile effettuare alcuna traduzione.

Perciò, é necessario definire una funzione di traduzione per ogni linguaggio di partenza (ovvero, il linguaggio in cui é rappresentato il dato originale) e per ogni linguaggio di arrivo (ossia il linguaggio in cui si desidera tradurre tale dato); queste traduzioni, concettualmente, possono essere rappresentate in un semplice formato tabellare/matriciale.

Procediamo quindi a definire una *matrice di traduzione* (tab. 2.2) che sarà utile nell'evidenziare le relazioni fra i linguaggi di tuple attualmente supportati.

↩	testo	JSON	Prolog
testo	sì	sì	sì
JSON	sì, tramite *wrapper*	sì	sì*
Prolog	sì, tramite *wrapper*	sì*	sì

Tabella 2.2

La suddetta tabella va letta nel seguente modo: i linguaggi in alto sono i linguaggi di *partenza*, ossia rappresentano il linguaggio in cui la tupla da tradurre é attualmente rappresentata; i linguaggi a sinistra, invece, sono i linguaggi di *arrivo*, ovvero quei linguaggi in cui si desidera rappresentare il dato. Ogni cella indica se e in che modo la traduzione dal linguaggio corrispondente

alla sua coordinata orizzontale a quello sulla sua coordinata verticale risulti possibile o meno.

A questo punto é necessario fare qualche considerazione. Ovviamente, le traduzioni giacenti sulla diagonale risultano banali, in quanto non viene effettuata alcuna conversione. Le conversioni $testo \rightarrow JSON$ e $testo \rightarrow Prolog$ risultano altrettanto semplici, in quanto basta incapsulare il testo di partenza, nel caso di JSON, in un oggetto composto da un solo campo testuale, oppure in una struttura *ad hoc* nel caso di Prolog.

Ad esempio, nel caso di JSON:

$$\text{“}hello\text{”} \mapsto \{\text{“}str\text{”} : \text{“}hello\text{”}\}$$

Mentre nel caso di Prolog:

$$\text{“}hello\text{”} \mapsto str(\text{“}hello\text{”}).$$

Invece, per quanto riguarda le traduzioni contrassegnate dall'asterisco (*), l'operazione di conversione risulta leggermente meno ovvia, tuttavia assume una complessità tutto sommato non elevata.

Per quanto riguarda la traduzione $Prolog \rightarrow JSON$, é possibile associare ad ogni funtore un singolo oggetto JSON con un nome corrispondente al funtore stesso.

Ad esempio, il termine

$$f(a, b)$$

diventerebbe

```
{
  "f" : {
    "_1" : "a",
    "_2" : "b"
  }
}
```

Le costanti Prolog, naturalmente, per poter essere tradotte in campi vanno associate a un nome fittizio il quale, ad esempio, può essere visto come un indice progressivo all'interno dell'oggetto JSON. Ciò, d'altronde, non compromette il contenuto informativo del dato originale, ma concorre semplicemente ad assicurare una traduzione efficace. All'interno del middleware, tuttavia, é necessario stabilire una convenzione che specifichi la struttura sintattica di tali indici, in modo da evitare ambiguità ed eventuali comportamenti inconsistenti.

Il caso opposto, che consiste nella traduzione del tipo $JSON \rightarrow Prolog$ é analogo – e inverso – al precedente. Infatti, laddove sia presente un campo identificato da un indice progressivo, sarà sufficiente convertire il valore del campo in una costante; mentre, nel caso in cui si trovi un oggetto, basterà convertirlo in un funtore dal nome uguale a quello di tale oggetto.

Ad esempio:

```
{
  "person" : {
    "name" : "john",
    "surname" : "doe"
  }
}
```

diventerebbe

$$person(name(john), surname(doe)).$$

Il meccanismo di matching. Una volta esplorate le caratteristiche fondamentali e le principali problematiche dei meccanismi di traduzione, é giunto il momento di spiegare come TuSoW gestisce l'operazione di *matching*.

Il *matching*, come già detto, é l'operazione che sta alla base delle memorie associative, in particolare di quelle *tuple-based*. Il *matching* in sostanza può essere visto come una funzione che, dati una tupla e un template, ci dica se la tupla corrisponde al template dato. Più precisamente, si può dire che il risultato di tale operazione sia un booleano assumente valore logico `vero` se la tupla appartiene all'insieme delle tuple descritte dal template, o falso in caso contrario.

TuSoW, come abbiamo già ricordato, gestisce il *matching* seguendo il modello LoTuS come linea guida. L'associazione fra tupla e template avviene infatti sulla base di un **predicato**, il quale descrive le caratteristiche della famiglia di tuple descritte dal template dato, determinando conseguentemente la corrispondenza la tupla e il template in esame.

È necessario, quindi, fornire un formalismo che ponga le basi per una possibile implementazione di TuSoW. Naturalmente, sarà necessario tenere conto non solo del poliglottismo dei *dataspace*, ma anche del meccanismo di traduzione precedentemente illustrato e del meccanismo di matching tramite predicati.

Procediamo quindi a dare una serie di definizioni. Siano

$$\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$$

gli **insiemi delle tuple rappresentabili** in ogni linguaggio di tuple. In questa notazione, i pedici rappresentano simbolicamente i singoli linguaggi di tuple: in generale, $\mathcal{L}_i$ sarà l'insieme di tutte le tuple rappresentabili tramite il linguaggio di tuple i.

Siano inoltre

$$\Theta_1, \Theta_2, \ldots, \Theta_n$$

gli **insiemi dei template rappresentabili** in ogni linguaggio di template. I pedici hanno significato analogo alla definizione precedente.

Si definisca quindi

$$\mathcal{D} \subseteq \mathcal{L}_1 \cup \mathcal{L}_2 \cup \ldots \cup \mathcal{L}_n$$

L'insieme $\mathcal{D}$ racchiude le caratteristiche di un *dataspace poliglotta*. Esso, infatti, rappresenta uno spazio di tuple in grado di contenere tuple scritte in qualsiasi dei linguaggi $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_n$ contemporaneamente.

Naturalmente, é necessario dare una definizione del meccanismo di traduzione. Si consideri quindi

$$\tau_{ij}(l_i) : \mathcal{L}_i \to \mathcal{L}_j, \forall i, j$$

τ prende il nome di **funzione di traduzione**: il suo scopo é quello di convertire una tupla scritta nel linguaggio i in una nuova tupla scritta nel linguaggio j. La funzione di traduzione, quindi, associa ogni tupla rappresentabile in un qualche linguaggio i a ogni tupla rappresentabile nel linguaggio j. Ovviamente, tale funzione andrà definita per ogni coppia di linguaggi (i, j), ove possibile.

Infine, definiamo la **funzione predicato**:

$$\pi(l_i, \theta_j) = \begin{cases} 1, & \tau_{ij}(l_i) \text{ definita} \wedge \pi(\tau_{ij}(l_i), \theta_j) \\ 0, & \text{altrimenti} \end{cases}, \ \forall\ (i,j) \mid i \neq j$$

$$\pi(l_i, \theta_j) = \begin{cases} 1, & l_i \in \theta_j \\ 0, & \text{altrimenti} \end{cases}, \ \forall\ (i,j) \mid i = j$$

la quale, data una tupla scritta nel linguaggio i e un template scritto nel linguaggio j, é in grado di stabilire se sussiste una corrispondenza fra la tupla e il template dato.

La funzione predicato calcola il *matching* nel seguente modo: dapprima, verifica se all'interno della matrice di traduzione é definita una traduzione dal linguaggio i al linguaggio j – ovvero, quello associato al template θ_j; successivamente, ricalcola induttivamente il *matching* fra la tupla originaria tradotta nel nuovo linguaggio – $\tau_{ij}(l_i)$ – fino a giungere al caso base, nel quale i linguaggi i e j coincidono, e si può quindi verificare se tupla e template corrispondono.

2.2 Servizio web

Nella sezione precedente sono state esposti i concetti chiave su cui é basato il servizio TuSoW. In questa sezione, invece, saranno esplorate quelle che sono le caratteristiche del lato web di TuSoW: l'attenzione sarà quindi posta sulla definizione di un insieme di API ReST che permettano di invocare primitive su un *dataspace* in rete. Questo, naturalmente, sarà possibile una volta presentata la nozione di spazio di tuple come *risorsa web*, il che permetterà al middleware di aderire al framework *Coordination as a Service*, estendolo di fatto a una sua derivazione denominata *Coordination as a Web Service*.

2.2.1 La risorsa *dataspace*

Nella filosofia ReST é fondamentale il concetto di **risorsa**. Una risorsa é semplicemente un oggetto sul quale é possibile operare tramite *metodi*, in modo similare a quanto avviene nel paradigma *object-oriented*. La principale differenza, tuttavia, é che mentre gli oggetti "classici" possiedono un numero potenzialmente illimitato di metodi, le risorse ReST possono avere al massimo tanti metodi quanti sono i metodi HTTP (es. GET, POST, PUT, DELETE). In figura 2.1 é possibile trovare una rappresentazione concettuale dei principali modi in cui é possibile organizzare le risorse in un servizio ReSTful.

La localizzazione delle risorse ReST avviene tramite **URI** (*Uniform Resource Identifier*), i quali permettono di indirizzare univocamente le risorse e di dare loro una struttura gerarchica.

Nel caso di TuSoW, si é deciso di caratterizzare ogni *dataspace* come risorsa, dando quindi la possibilità di istanziare più spazi su un singolo host. Le risorse *dataspace* si strutturano nel modo seguente:

```
/data_spaces/<dataspace>
```

Stando alla tassonomia mostrata in 2.1, possiamo riconoscere nella risorsa `/data_spaces` una *collection* composta di tante sotto-risorse `dataspace`.

Nella prossima sezione andremo a esplorare quali sono i modi per usufruire del servizio web a livello pratico, il che comporterà principalmente la formulazione di una ReSTful API.

2.2.2 TuSoW ReSTful API

Procediamo quindi a definire una API pubblica che servirà per interfacciarsi con il servizio web di TuSoW.

Verrà esposto, per ogni metodo HTTP supportato dal servizio, il significato della sua invocazione in relazione ai parametri che verranno allegati alla *query* e

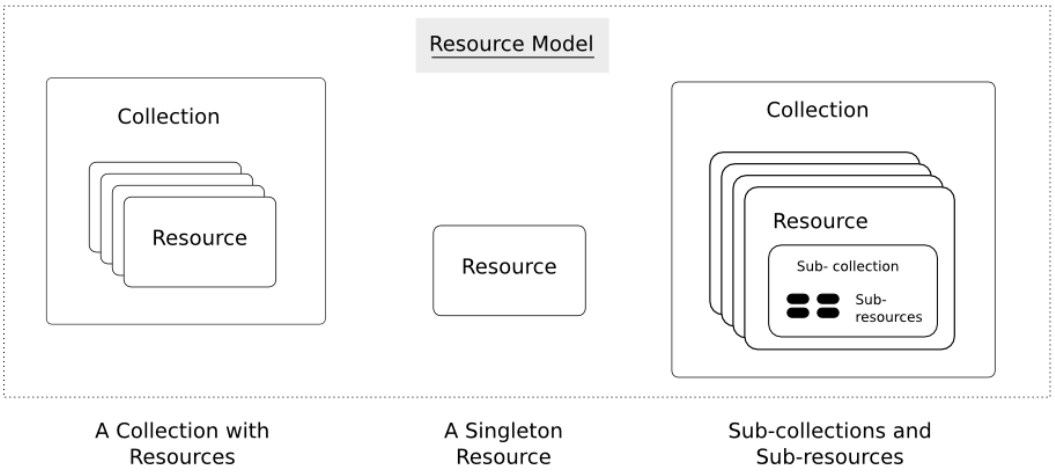


Figura 2.1: Le differenti modalità di organizzazione delle risorse in una ReSTful API [1].

all'eventuale *body* fornito. Verranno inoltre mostrati gli *status code* che saranno inviati a corredo di ogni risposta.

- **GET /data_spaces/⟨dataspace⟩** Cerca nel *dataspace* una tupla che soddisfi il template indicato.

  Parametri:

  template *Obbligatorio.* Specifica il template che verrà usato come chiave di ricerca nel *dataspace*.

  predicative *Valore di default:* `false`. Se `true`, la richiesta fallisce immediatamente se non viene trovato subito il dato desiderato. Se `false` o non specificato, la richiesta viene sospesa fino a che un dato corrispondente non é disponibile nel *dataspace*.

  bulk *Valore di default:* `false`. Se `true`, il risultato conterrà tutti i dati corrispondenti al template indicato presenti nel *dataspace*.

  matching *Valore di default:* `n.s.` Se `matching = any`, il dato richesto dovrà combaciare con *uno qualsiasi* dei template specificati. Se `matching = every`, il dato richiesto dovrà combaciare con *tutti* i template specificati. Se il parametro non é specificato nella query, verrà considerato solo il primo template ai fini del matching.

Codici di stato:

`200 - OK:` La richiesta é andata a buon fine, il risultato é contenuto nel *body*.

`204 - No Content:` La richiesta era predicativa e non é stato trovato alcun dato corrispondente al template indicato.

`400 - Bad Request:` Input non conforme.

- **POST /data_spaces/⟨dataspace⟩** Inserisce una nuova tupla nel *dataspace*.

  Parametri:

  sync *Valore di default:* `false`. Se `true`, la richiesta viene sospesa fino a che il dato fornito non viene effettivamente inserito nel *dataspace*.

  bulk *Valore di default:* `false`. Se `true`, tutti i dati presenti nel *body* verranno inseriti nel *dataspace*. Se `false` o non specificato, verrà inserito solo il primo dato della lista.

  Codici di stato:

  `201 - Created:` La richiesta sincrona (parametro "sync" = `true`) é andata a buon fine, l'inserimento è avvenuto con successo.

  `202 - Accepted:` La richiesta di inserimento é stata presa in carico dal servizio.

  `400 - Bad Request:` Input non conforme.

- **PUT /data_spaces/⟨dataspace⟩** Sostituisce i dati presenti nel *dataspace* con quelli forniti.

  Parametri:

  sync *Valore di default:* `false`. Se `true`, la richiesta viene sospesa fino a che i dati forniti non vengono effettivamente inseriti nel *dataspace*.

  Codici di stato:

  `201 - Created:` La richiesta sincrona (parametro "sync" = `true`) é andata a buon fine, l'inserimento è avvenuto con successo.

`202 - Accepted:` La richiesta di inserimento é stata presa in carico dal servizio.

`400 - Bad Request:` Input non conforme.

- **DELETE /data_spaces/⟨dataspace⟩** Rimuove dal *dataspace* una tupla che soddisfi il template indicato.

  Parametri:

  template *Obbligatorio.* Specifica il template che verrà usato come chiave di ricerca nel *dataspace*.

  predicative *Valore di default:* `false`. Se `true`, la richiesta fallisce immediatamente se non viene trovato subito il dato desiderato. Se `false` o non specificato, la richiesta viene sospesa fino a che un dato corrispondente non é disponibile nel *dataspace*.

  bulk *Valore di default:* `false`. Se `true`, il risultato conterrà tutti i dati corrispondenti al template indicato presenti nel *dataspace*.

  matching *Valore di default:* `n.s.` Se `matching = any`, il dato richesto dovrà combaciare con *uno qualsiasi* dei template specificati. Se `matching = every`, il dato richiesto dovrà combaciare con *tutti* i template specificati. Se il parametro non é specificato nella query, verrà considerato solo il primo template ai fini del matching.

  Codici di stato:

  `200 - OK:` La richiesta é andata a buon fine, il risultato é contenuto nel *body*.

  `204 - No Content:` La richiesta era predicativa e non é stato trovato alcun dato corrispondente al template indicato.

  `400 - Bad Request:` Input non conforme.

Content negotiation. L'agente che voglia utilizzare le suddette API deve corredare la loro invocazione di due header HTTP fondamentali: `Accept` e `Content-Type`. Questi header servono all'agente per comunicare al server, rispettivamente, il formato desiderato della risorsa richiesta e il formato dell'informazione trasportata nella richiesta: questa possibilità di usufruire di una risorsa di un servizio web in diversi formati prende il nome di *content negotiation* ed é alla base dei servizi ReSTful. In particolare, quando é il client a scegliere il formato della risorsa si parla di *agent-driven content negotiation*;

se, invece, un formato appropriato viene selezionato di volta in volta dal server, si parla di *server-driven content negotiation*.

Le API di TuSoW in questo senso sono *agent-driven*, ovvero é l'agente a stabilire il formato nel quale ricevere la tupla – o le tuple – richieste. Nella tabella 2.3 é possibile trovare l'elenco dei MIME supportati dal servizio, associati ognuno al rispettivo tipo di dato o di template.

	tuple	template
testo	`application/string`	`application/regex`
JSON	`application/json`	`application/jsonpath`
Prolog	`application/prolog`	`application/prolog`

Tabella 2.3: Elenco dei MIME supportati, raggruppati per tecnologia di rappresentazione e divisi per tuple e template.

Per usufruire di questo meccanismo, un agente deve seguire le seguenti direttive:

1. Quando si invia una richiesta di tipo GET o DELETE, nell'header `Accept` occorre specificare il formato della tupla richiesta, mentre nell'header `Content-Type` va indicato il formato di rappresentazione del template fornito;

2. Quando si invia una richiesta di tipo POST o PUT, in entrambi gli header va specificato il tipo di tupla che si vuole inserire all'interno del *dataspace*.

Similarità e differenze con ReST. Nel formulare l'API qui proposta, si é cercato di restare aderenti alle caratteristiche base dello stile ReST[6]. Sebbene questo lavoro di tesi sia stato svolto con con l'ottica di mostrare la massima fedeltà a questa filosofia, emergono in realtà alcune differenze che andremo qui a discutere.

Prima di tutto, possiamo affermare che l'API di TuSoW soddisfi i **vincoli** originariamente presentati in [6]:

- **Client-Server**: alla base di questo vincolo sta la "separazione di intenti" che guida la progettazione delle interfacce di client e server. Una volta definiti e separati gli aspetti chiave delle due parti, é possibile farle evolvere in modo separato, garantendo un ottimo livello di disaccoppiamento e scalabilità.

- **Stateless**: il server non tiene alcuna traccia delle informazioni sulle sessioni dei vari client. Ognuno di essi, infatti, deve includere nella richiesta tutte le informazioni necessarie alla sua corretta comprensione.

- **Cache**: le risposte del server devono poter essere rese memorizzate in *cache* in modo tale da garantire una comunicazione più efficiente e, in generale, una migliore *Quality of Service*.

- **Uniform interface**: le implementazioni dei vari componenti devono essere separate dalle interfacce che espongono. Questo permette, ancora una volta, un'evoluzione indipendente dei suddetti componenti, e in generale semplifica l'architettura e amplifica la visibilità delle interazioni, sebbene al costo di una eventuale riduzione delle performance.

- **Layered system**: i componenti non possono "vedere" oltre il primo livello architetturale con il quale stanno interagendo. Fornendo ai componenti una conoscenza ridotta a un singolo livello architetturale, viene promossa l'indipendenza architetturale del servizio e sono rese trasparenti alcune funzionalità interne, come ad esempio eventuali politiche di *load-balancing*.

- **Code-On-Demand (opzionale)**: i client possono estendere le loro funzionalità scaricando dal servizio codice sorgente sotto forma di *applet* o *script*. Questo, naturalmente, permette di semplificare l'implementazione dei client riducendo il numero di funzionalità che viene richiesto loro di implementare; tuttavia, questo causa un abbassamento del livello di visibilità, il che rende *opzionale* l'aderenza a questo vincolo.

Si può affermare che le API di TuSoW soddisfino con successo tutti i vincoli specificati, fatta eccezione per l'ultimo, *Code-On-Demand*, il quale di per sé non ha assunto particolare rilevanza nel concepimento del middleware. Non é stato volutamente rispettato neanche il secondo vincolo, *Cache*, in quanto, data la natura fortemente interattiva delle risorse *dataspace* e vista la grande variabilità nella natura e nella disponibilità dei dati all'interno di tali risorse, il *caching* si rivela inapplicabile, e tutte le risposte sono da considerarsi "implicitamente" *uncacheable*. Per questo motivo, eventuali miglioramenti della *Quality of Service* devono necessariamente essere apportati come modifiche e/o estensioni al servizio di coordinazione stesso.

Un'ulteriore differenza con la filosofia originale ReST riguarda il concetto di risorsa e dell'uso che si fa dei metodi HTTP per interagire con le risorse messe a disposizione da un servizio. Un approccio "ortodosso" richiederebbe infatti che, per esempio, il metodo `POST` venga usato per creare una nuova risorsa, indirizzabile tramite un URI univoco all'interno della gerarchia. Analogamente, il metodo `DELETE` presupporrebbe l'eliminazione di una risorsa e del suo indirizzo dal servizio. In TuSoW, invece, si é preferito adottare una modalità di interazione diversa. La creazione e la cancellazione delle *risorse*

non avvengono come conseguenza dell'invocazione di qualche metodo, bensì – come vedremo – saranno scatenate dal servizio stesso in risposta a particolari condizioni.

Nella tabella 2.4 viene eseguito un confronto tra il funzionamento "classico" dei metodi HTTP in ReST e in TuSoW su un URI di tipo `/collection/resource`.

	ReST	TuSoW
GET	Recupera la risorsa	Recupera un dato contenuto nella risorsa
POST	Crea una risorsa all'interno della *collection*	Crea un dato all'interno della risorsa
PUT	Rimpiazza la *collection* con una nuova *collection*	Rimpiazza il contenuto della risorsa con un nuovo contenuto
DELETE	Elimina l'intera *collection*	Elimina un dato contenuto nella risorsa

Tabella 2.4: Differenze tra la consueta semantica dei metodi HTTP in ReST e in TuSoW, in relazione alla manipolazione di collezioni di risorse.

È facile notare come in TuSoW i metodi HTTP intervengano sul *contenuto* delle risorse piuttosto che sulle risorse stesse. Ciò, naturalmente, é dovuto al fatto che il compito di TuSoW consista nel fornire un servizio di corodinazione basato su spazi di tuple, i quali sono resi accessibili sul web in qualità di risorse. Ne segue che é necessario modificare la semantica dei metodi per permettere un comportamento il più possibile aderente alle specifiche dell'astrazione di *tuple space*.

La filosofia ReST, dunque, é servita prima di tutto per la semplicità e la potenza che é possibile offire tramite un servizio web. In particolare, il vincolo di *Uniform Interface* permette di costruire servizi che abbiano un'interfaccia semplice e ben definita, ma soprattutto che permettano a un grande numero di dispositivi differenti di usufruire del medesimo servizio e della stessa interfaccia, il che semplifica notevolmente l'implementazione interna dei client e incoraggia un impiego pervasivo e stabile del servizio. Tutte queste cose sono ugualmente raggiunte tramite l'aderenza al vincolo *Stateless*, il quale non solo semplifica l'implementazione, ma impedisce interazioni ambigue e rafforza ulteriormente il grado di disaccoppiamento fra i componenti.

Relazione tra API web e primitive. Ogni metodo HTTP usato nell'API assume una semantica ben precisa. Il comportamento di ognuno di essi può essere associato sia a un "gruppo" di primitive di riferimento (come `read`, `put`, ecc...) il quale ne identifica la semantica operazionale, sia a un "tipo"

riguardante, ad esempio, la predicatività, la possibilità di operare su più tuple contemporaneamente, e così via.

Per prima cosa, dunque, é opportuno mostrare le corrispondenze fra i metodi HTTP supportati dall'API e le primitive disponibili in TuSoW, come mostrato in tabella 2.5.

metodo	gruppo di primitive
GET	`read`, `readIfPresent`, `readAll`, `readAny`, `readEvery`, `get`
POST	`put`, `putAll`
PUT	`set`
DELETE	`take`, `takeIfPresent`, `takeAll`, `takeAny`, `takeEvery`, `clear`

Tabella 2.5: Corrispondenze fra i metodi HTTP dell'API di TuSoW e le primitive disponibili.

Più in generale, possiamo associare ai vari metodi, in relazione all'effetto su un *dataspace* causato dalla loro invocazione, i seguenti significati:

- il metodo GET comporta la *lettura* di una o più tuple;
- il metodo POST comporta la *creazione* di una o più tuple;
- il metodo PUT comporta l'*aggiornamento* di tutto il contenuto del *dataspace*;
- il metodo DELETE comporta l'*eliminazione* di una o più tuple.

Come si sarà sicuramente notato nella specifica dell'API, il comportamento dei metodi HTTP supportati può essere modificato grazie alla presenza di alcuni *query parameter*, i quali permettono di invocare primitive diverse all'interno del servizio di coordinazione. Nella tabella 2.6 si é deciso di raffigurare le varie configurazioni di utilizzo dei *query parameter*; non sempre é stato indicato l'uso del parametro `template`, il cui uso si suppone obbligatorio per qualsiasi operazione invocata tramite GET o DELETE. Per quanto riguarda gli altri parametri, se il loro uso non é specificato, é possibile ometterlo nella composizione dell'URI.

Si consideri la suddetta tabella come una "guida all'uso" da conservare come riferimento operativo, sebbene l'API possa certamente subire cambiamenti nel tempo.

primitiva	metodo HTTP	parametri
`read`	GET	–
`readIfPresent`	GET	`predicative = true`
`readAll`	GET	`bulk = true`
`readAny`	GET	`matching = any`
`readEvery`	GET	`matching = every`
`get`	GET	`template` omesso
`put`	POST	–
`putAll`	POST	`bulk = true`
`set`	PUT	–
`take`	DELETE	–
`takeIfPresent`	DELETE	`predicative = true`
`takeAll`	DELETE	`bulk = true`
`takeAny`	DELETE	`matching = any`
`takeEvery`	DELETE	`matching = every`
`clear`	DELETE	`template` omesso

Tabella 2.6: Le possibili configurazioni d'uso dell'API di TuSoW.

Generazione dei dataspace e trasparenza. È necessario ribadire che TuSoW non offre primitive di meta-coordinazione, il che rende impossibile – attualmente – la creazione, eliminazione o replicazione dei *dataspace* in maniera esplicita.

Tuttavia, si é rivelata indispensabile l'introduzione di un meccanismo che permettesse almeno la creazione dei *dataspace*. Questo deve tuttavia avvenire in maniera perfettamente trasparente agli agenti, i quali non devono "accorgersi" della comparsa di nuovi *coordination medium*. Dopotutto, TuSoW deve semplicemente garantire la disponibilità delle primitive di coordinazione come supporto alla scrittura di applicazioni distribuite.

Per questo motivo si é scelto di fare in modo che un *dataspace* venga creato in risposta alla prima operazione invocata su di esso. Come mostrato precedentemente, ogni *dataspace* é anche una risorsa web, il che in questo caso le conferisce un nome univoco; se ne deduce che all'interno di una singola istanza di TuSoW non possono essere creati due *dataspace* con lo stesso nome. Questo approccio, inoltre, permette di fornire agli agenti connessi un servizio trasparente, in quanto é sufficente che l'interazione *avvenga* perché esista un relativo *coordination media* a supporto di essa.

Al momento non sono stati concepiti meccanismi per l'eliminazione o la replicazione dei *dataspace*. La loro ideazione potrà certamente costituire l'argomento di sviluppi futuri.

2.2.3 Coordination as a Web Service

L'analisi concettuale svolta finora in questa sezione ha preso spunto dai suggerimenti forniti in [12] per quanto riguarda la trasposizione del framework *Coordination as a Service* nell'ambito dei servizi web.

È stato difatti suggerito come un tale framework possa essere impiegato con il minimo sforzo in ambito web, poiché l'interazione può essere descritta semplicemente come una serie di richieste e risposte, il che implica una forte somiglianza con il modello di comunicazione fornito dal protocollo HTTP.

Questa proprietà, naturalmente, é stata adottata come nozione fondamentale durante la modellazione del middleware: é proprio attorno al sistema di scambio di messaggi HTTP che é stato costruito il funzionamento del servizio.

Per questi motivi si é deciso di riferirsi all'ambito dei servizi web di coordinazione con l'appellativo di *Coordination as a Web Service* (abbr. CoWS), concentrandovi le proprietà e le problematiche derivate, da una parte, da *Coordination as a Service* e dalle tecnologie di coordinazione in generale, e dall'altra dal mondo dei servizi web.

2.3 Client CLI

L'ultima parte del sistema che si é deciso di realizzare consiste in un semplice client che permetta a un utente fisico di interagire con un servizio TuSoW su rete in tempo reale, sia per questioni strettamente operative, ad esempio, sia per questioni di *monitoring*. A tal proposito si é scelto di mettere in conto la produzione di un piccolo applicativo *no-frills*, consistente in una semplice interfaccia a riga di comando, che possieda le seguenti funzionalità:

1. Connettersi a un servizio TuSoW ospitato su un qualsiasi host in rete;
2. Invocare le primitive supportate dal servizio, avendo la possibilità di specificare tutte le informazioni necessarie al suo corretto funzionamento;
3. Facilità di utilizzo.

Capitolo 3

Architettura e progettazione

Finora, grazie alle conoscenze illustrate nel capitolo 1, é stata condotta un'analisi delle funzionalità che avrebbe dovuto offire TuSoW, un moderno servizio di web di coordinazione basata sulle tuple.

Inizialmente é stato introdotto un modello che permette di costruire servizi di coordinazione poliglotti, ossia servizi in grado di supportare contemporaneamente più linguaggi di tuple e template, che dispongano di funzionalità di traduzione dei dati da una tecnologia di rappresentazione all'altra.

Successivamente, é stata fatta un'associazione tra il comportamento tipico dei *coordination media* come gli spazi di tuple e alcune delle principali tecnologie per lo sviluppo di servizi web, cioé il protocollo HTTP e la filosofia ReST, arrivando a specificare una API ReSTful che serva ad interagire con il servizio su rete.

Infine, sono stati descritti brevemente i requisiti essenziali per costruire un client a riga di comando da utilizzare per interfacciarsi manualmente con il servizio di coordinazione.

Sono stati quindi concepiti tre moduli software, ognuno atto ad assolvere i requisiti funzionali riassunti sopra:

- **Modulo core**: fornisce un'astrazione di spazio di tuple multi-linguaggio, e ne implementa i meccanismi di traduzione;

- **Modulo web**: mette a disposizione un servizio web che permetta di comunicare in rete con uno o più spazi di tuple tramite un'interfaccia ReSTful;

- **Modulo client**: contiene un semplice client a riga di comando che permette a un utente fisico di comunicare in modo interattivo con uno spazio di tuple.

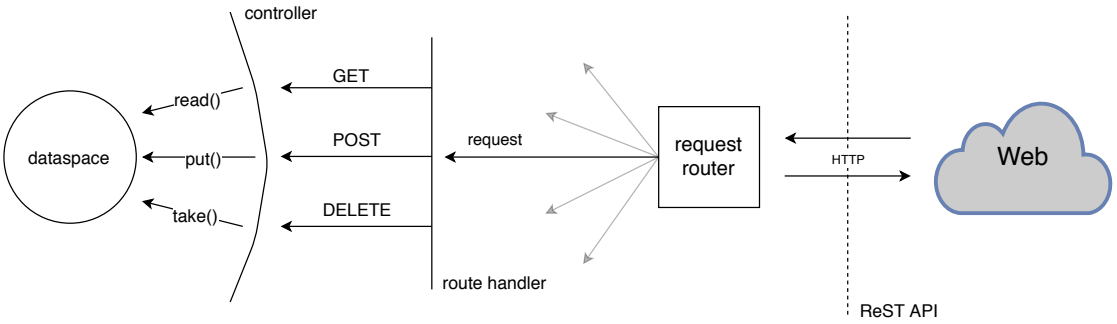


Figura 3.1: Architettura funzionale del servizio web TuSoW.

In questo capitolo, dunque, verrà finalmente proposta una descrizione dell'architettura del servizio TuSoW e verranno documentate le scelte di *design* che hanno caratterizzato lo sviluppo dei vari moduli software. Per ognuno di essi, infatti, saranno dapprima introdotte le caratteristiche architetturali generali: esse saranno corredate, ove possibile, di appositi diagrammi volti a rappresentare in modo sintetico il comportamento e la struttura del sottosistema in esame. Successivamente si procederà a spiegare le scelte di progettazione più rilevanti, ovvero quelle più determinanti ai fini della realizzazione dei requisiti specifici analizzati nel capitolo 2.

3.1 Architettura del servizio

In figura 3.1 é possibile trovare una raffigurazione concettuale della struttura del servizio web.

Tale raffigurazione é puramente informale, e fornisce una rappresentazione concettuale dei componenti software principali del servizio TuSoW e delle loro funzioni.

Di seguito, quindi, verranno descritti brevemente tali concetti, letti dalla figura 3.1 in ordine da destra a sinistra. La natura delle loro funzioni é stata adottata come riferimento durante la progettazione di dettaglio, la quale verrà esposta nelle sezioni a seguire.

Request router. Questo componente si può considerare come il più "esterno" dell'architettura, costituendo di fatto il punto di arrivo/partenza delle richieste/risposte HTTP. Il *router* ha la responsabilità di ricevere le richieste HTTP e di ridirezionarle, a seconda dell'URI specificato, al rispettivo *handler*.

Route handler. Questo componente ha il compito di estrarre tutte le informazioni necessarie dal messaggio HTTP in arrivo e di passarle al *controller*

tramite un'opportuna chiamata di metodo, corrispondente al metodo HTTP usato. Tali informazioni comprendono il metodo HTTP, i *query parameter*, gli header e l'eventuale *payload*. Il *route handler*, inoltre, é responsabile di gestire il processo di *content negotiation* necessario a selezionare correttamente il linguaggio tramite il quale l'agente desidera inviare o ricevere dati. Ogni *route handler* é associato a uno e un solo URI, e può gestire uno o più *controller*.

Controller. Il *controller* ha la responsabilità di interpretare le informazioni ricevute dal rispettivo *handler* al fine di scegliere la corretta primitiva da invocare sul *dataspace*. Ogni *controller* controlla uno e un solo *dataspace*.

Dataspace. Quest'ultimo componente fondamentale del servizio rappresenta il vero e proprio mezzo (*medium*) di coordinazione. Questo componente, il più "profondo" del flusso computazionale che si occupa della gestione delle richieste HTTP, espone un'interfaccia che permette di invocarvi le primitive di coordinazione supportate dal servizio e si occupa di delegare la traduzione delle tuple a un'apposito modulo di *utility* (non elencato in questa lista).

In figura 3.2 é possibile trovare un diagramma delle classi che mostra le relazioni statiche fra le suddette componenti software, mentre in figura 3.3 si può trovare un diagramma di sequenza che descriva le interazioni base fra agenti, interfaccia web e *dataspace*, incluso il servizio di traduzione delle tuple.

3.2 Modulo core

Il modulo *core* si può considerare il "cuore pulsante" del servizio di coordinazione TuSoW. Da esso, infatti, dipende il modulo web (sezione 3.3) che si basa sulle astrazioni descritte all'interno della sezione corrente per fornire agli spazi di tuple un'interfaccia web, e quindi la possibilità di essere visti e manipolati su Internet.

Perciò, il contenuto di questa sezione sarà volto dapprima a descrivere e ad argomentare le scelte di design volte alla realizzazione dei *coordination media*, ossia gli spazi di tuple; in un secondo momento, l'attenzione sarà concentrata sul meccanismo di traduzione delle richieste.

3.2.1 I coordination media

I *coordination media* – ovvero quelle entità di un sistema coordinato che contengono le tuple al loro interno – sono state progettate tramite alcune semplici astrazioni.

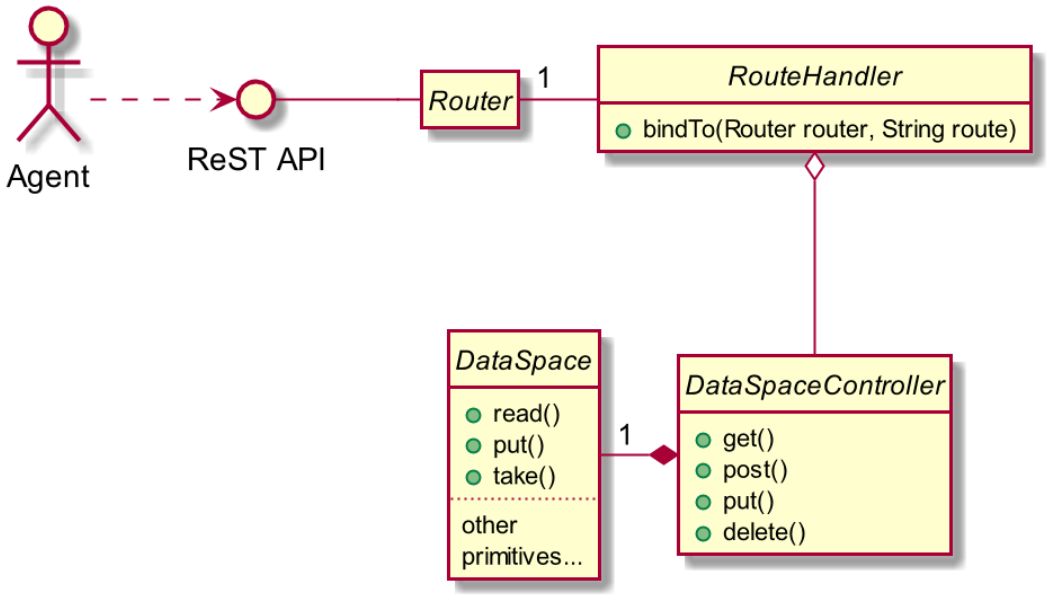


Figura 3.2: Diagramma delle classi che mostra le relazioni statiche fra i principali componenti del servizio.

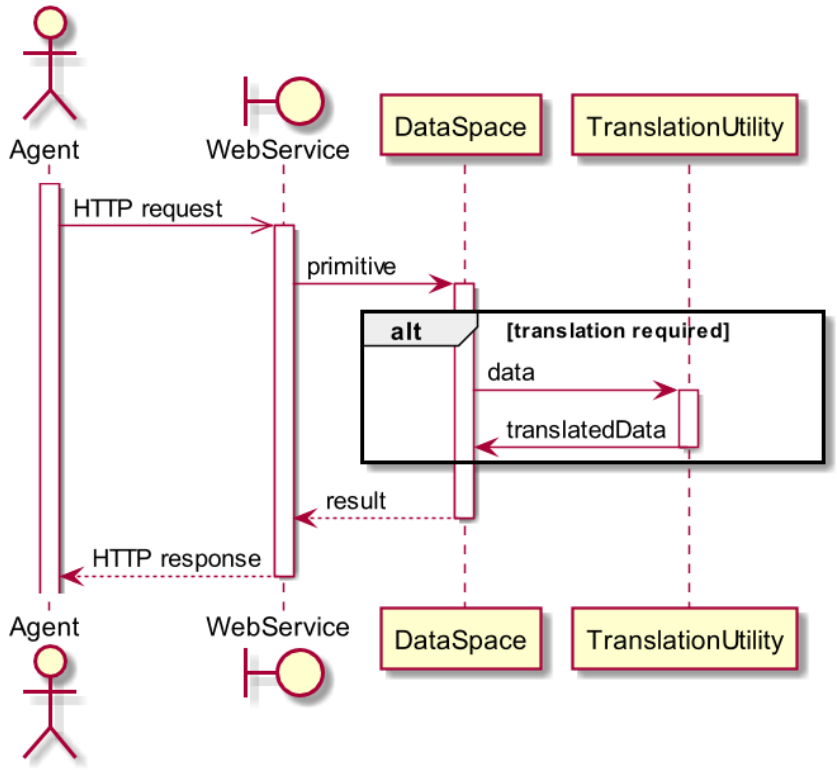


Figura 3.3: Diagramma di sequenza che mostra le interazioni (semplificate) fra i principali componenti del servizio, con l'aggiunta del modulo di traduzione.

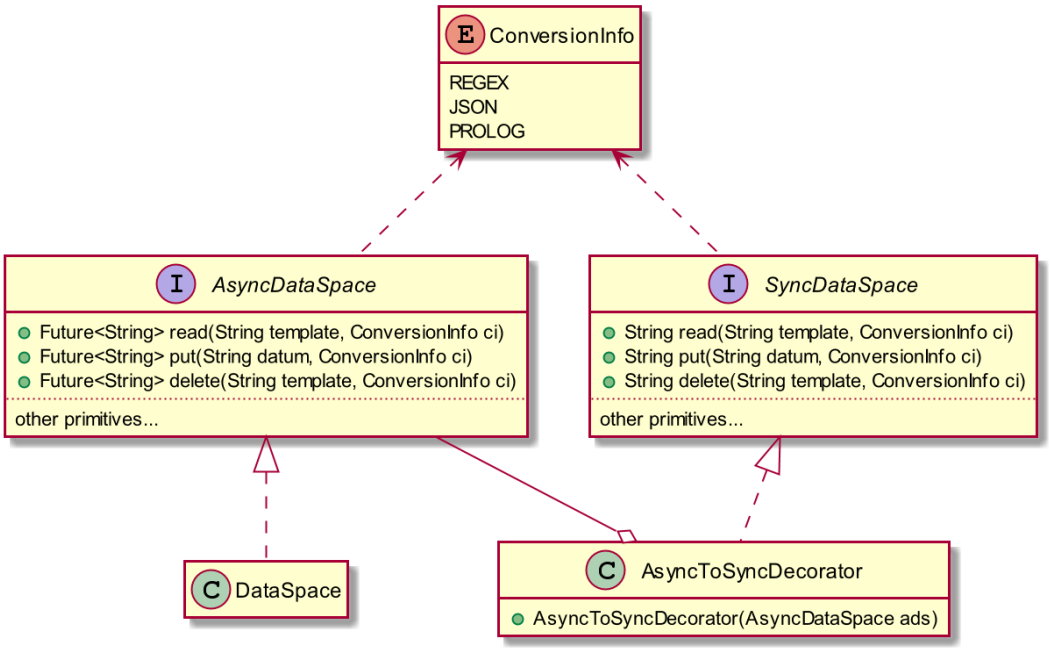


Figura 3.4: Diagramma delle classi che mostra il design degli spazi di tuple in TuSoW.

In figura 3.4 é possibile trovare un diagramma delle classi che rappresenta le principali astrazioni inerenti alle operazioni di coordinazione, come descritte di seguito.

AsyncDataSpace

```
interface AsyncDataSpace {
   Future<String> read(String template, ConversionInfo ci);
   Future<String> take(String template, ConversionInfo ci);
   Future<String> put(String datum, ConversionInfo ci);
   ...
}
```

L'interfaccia *AsyncDataSpace* espone i metodi corrispondenti alle primitive di coordinazione supportate dal servizio.

Ogni metodo riceve in input un oggetto di tipo *String* e un oggetto di tipo *ConversionInfo*. Nel caso delle primitive di inserimento, il testo in input corrisponde alla rappresentazione del dato da inserire, e l'oggetto di tipo *ConversionInfo* ne descrive il tipo. Nel caso delle primitive di lettura, il testo in input corrisponde al template usato come query, mentre l'oggetto di tipo *Con-*

versionInfo descrive il tipo di dato corrispondente al linguaggio di template indicato e, conseguentemente, al tipo di dato richiesto. È proprio tramite il parametro `ci` che é possibile indicare allo spazio di tuple il formato desiderato in cui ricevere il dato; maggiori dettagli verranno forniti parlando della classe *ConversionInfo* e di come avviene la traduzione.

Questa interfaccia prevede che le classi che aderiscono al contratto in essa espresso mostrino un comportamento asincrono, per far sì che il codice cliente, una volta invocato un metodo, non si blocchi in attesa della fine della computazione, ma possa bensì registrare eventuali *callback* da eseguire una volta che il *dataspace* notifica la terminazione dell'operazione. Per questo motivo si é scelto di adottare il costrutto delle *future* come tipo di ritorno dei vari metodi, cosicché fungano da punti di sincronizzazione per il codice cliente.

In programmazione concorrente, ci si riferisce con il nome di *future* a un particolare costrutto che fa da "contenitore" del risultato di un'operazione asincrona, il quale pertanto risulta inizialmente sconosciuto. Generalmente, le *future* sono oggetti accessibili in sola lettura, poichè permettono al codice cliente solamente di leggere il valore restituito dalla computazione asincrona; il loro uso, dunque, é spesso coniugato all'impiego di un ulteriore costrutto, le *promise*, che permettono di assegnare una sola volta un risultato a una data *future*.

DataSpace

Questa classe costituisce l'implementazione principale dell'interfaccia *AsyncDataSpace*. In essa viene descritto e definito il funzionamento specifico di ogni singolo metodo/primitiva del contratto a cui aderisce.

La classe *DataSpace* fornisce un'implementazione *non-deterministica* delle primitive: si rimanda il lettore alla sottosezione 2.1.1 per una descrizione della politica di gestione delle richieste in TuSoW.

SyncDataSpace

```
interface SyncDataSpace {
   String read(String template, ConversionInfo ci);
   String take(String template, ConversionInfo ci);
   String put(String datum, ConversionInfo ci);
   ...
}
```

L'interfaccia *SyncDataSpace* risulta estremamente simile all'interfaccia *AsyncDataSpace* precedentemente descritta: essa offre infatti una serie di metodi,

ognuno corrispondente a una particolare primitiva supportata da TuSoW; inoltre, gli argomenti di ogni metodo risultano anch'essi equivalenti.

L'utilità di questa interfaccia risiede nel fatto di mettere a disposizione del codice cliente un'astrazione che preveda che le primitive di coordinazione espongano un comportamento *sincrono*, ossia bloccante. Tutti i metodi, infatti, espongono come tipo di ritorno non più una *future* contenente il dato richiesto, ma direttamente quest'ultimo. L'invocazione di uno di questi metodi comporta, naturalmente, una sospensione del flusso di controllo del codice chiamante, la cui esecuzione viene ripresa solo in seguito all'esaurimento della computazione associata all'operazione in questione.

AsyncToSyncDecorator

```
class AsyncToSyncDecorator {
   private AsyncDataSpace decorated;

   @Override
   String read(String template, ConversionInfo ci) {
      return decorated.read(template, ci).get();
   }

   ...
}
```

Questa classe fornisce un modo per fare uso di un oggetto di tipo *AsyncDataSpace* tramite chiamate *bloccanti*. Questo permette effettivamente di usufruire di un *dataspace* asincrono come se esso implementasse l'interfaccia *SyncDataSpace*.

Questa classe aderisce, almeno parzialmente, al pattern *Decorator* [8]. Si é rivelato impossibile aderire totalmente alle specifiche del pattern in quanto le interfacce *AsyncDataSpace* e *SyncDataSpace* espongono metodi dalle *signature* differenti, seppur molto simili fra di loro. Ciononostante, questa classe mette a disposizione un modo semplice di interagire con un *dataspace* esistente tramite un'interfaccia sincrona, aumentando molto la qualità del codice cliente qualora la logica applicativa mostri la ripetuta necessità di attendere la terminazione delle primitive di coordinazione.

ConversionInfo

Questa enumerazione contiene un elemento per ciascuna coppia linguaggio di tuple/linguaggio di template supportata dal servizio TuSoW. Al momento,

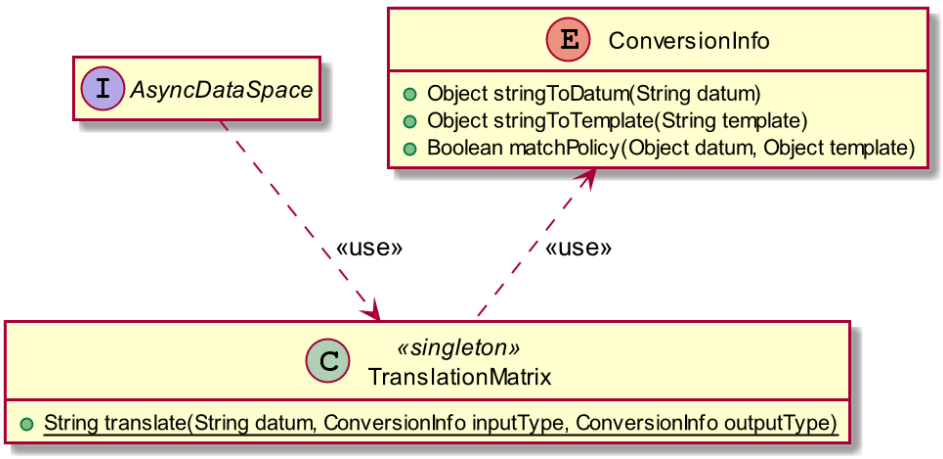


Figura 3.5: Diagramma delle classi che mostra il design del modulo di traduzione.

in particolare, sono disponibili il linguaggio testuale, il linguaggio JSON e il linguaggio Prolog.

Le interfacce *AsyncDataSpace* e *SyncDataSpace*, come si evince dal diagramma in figura 3.4, mostrano una dipendenza verso questo modulo, in quanto esso compare come parametro nelle *signature* di ogni primitiva. Dal punto di vista del codice cliente, infatti, gli oggetti di tipo *ConversionInfo* servono proprio a indicare i vari formati di rappresentazione dei dati: allegandoli a una primitiva, infatti, é possibile indicare al *dataspace* il formato in cui il dato che si desidera inserire é rappresentato, oppure il formato nel quale si vuole leggerlo. In quest'ultimo caso, ovviamente, l'oggetto di tipo *ConversionInfo* rappresenta anche il linguaggio di template usato per rappresentare il template fornito.

3.2.2 Traduzione delle richieste

È stato previsto un modulo che permetta di effettuare la traduzione delle tuple all'interno del sistema.

Il processo di traduzione naturalmente prevede che vengano indicati, oltre al dato da tradurre, anche il tipo di rappresentazione nel quale tale dato é fornito all'inizio della traduzione, e un tipo di rappresentazione nel quale si desidera che il dato venga tradotto.

Questo compito é stato affidato ad un'unica classe denominata *TranslationMatrix*: essa aderisce al pattern *Singleton*, in quanto offre un punto di accesso pubblico al servizio di traduzione e poiché tale servizio viene implementato all'interno di un'unica istanza. La classe *TranslationMatrix* espone

un solo metodo pubblico (statico), ossia il metodo `translate()`, che accetta come input una stringa contenente il dato da tradurre, e due oggetti di tipo *ConversionInfo*: il primo di essi indica il formato del dato in input, mentre il secondo indica il formato desiderato del dato in output. L'output di tale metodo consiste in una nuova stringa contenente il dato tradotto.

All'interno del modulo *core*, al momento, questo servizio viene utilizzato esclusivamente dalla classe *AsyncDataSpace* per verificare il *matching* fra tuple e template corrispondenti a formati di rappresentazione differenti. La classe *TranslationMatrix* deve far affidamento, a sua volta, ai servizi offerti dall'enumerazione *ConversionInfo* la quale offre, oltre al semplice elenco dei formati supportati, una serie di metodi di conversione, implementati per ognuno delle rappresentazioni disponibili.

I metodi sono i seguenti:

`stringToDatum()` prende in input una stringa contenente una tupla rappresentata in uno dei linguaggi disponibili, e lo incapsula in un oggetto del tipo di dato corrispondente;

`stringToTemplate()` prende in input una stringa contenente un template rappresentato in uno dei linguaggi disponibili, e lo incapsula in un oggetto del tipo di template corrispondente;

`matchPolicy()` dati in input due oggetti contenenti rispettivamente una tupla e un template fra di loro compatibili (es. testo/regex, JSON/JsonPath, ecc...), restituisce `true` se la tupla fornita combacia con il template, `false` altrimenti.

In figura 3.5 é possibile trovare un diagramma delle classi contenente gli aspetti principali del meccanismo di traduzione qui discusso.

3.3 Modulo web

Il modulo web di TuSoW, fra quelli progettati e realizzati ai fini di questa tesi, é sicuramente da considerarsi il più importante. Il suo compito principale, infatti, é quello di mettere a disposizione un servizio web che sia in grado di analizzare, gestire e rispondere alle richieste HTTP provenienti da qualsiasi agente in rete.

In questa sezione verranno quindi presentate le scelte di design relative all'architettura e al meccanismo di *routing* e gestione delle richieste.

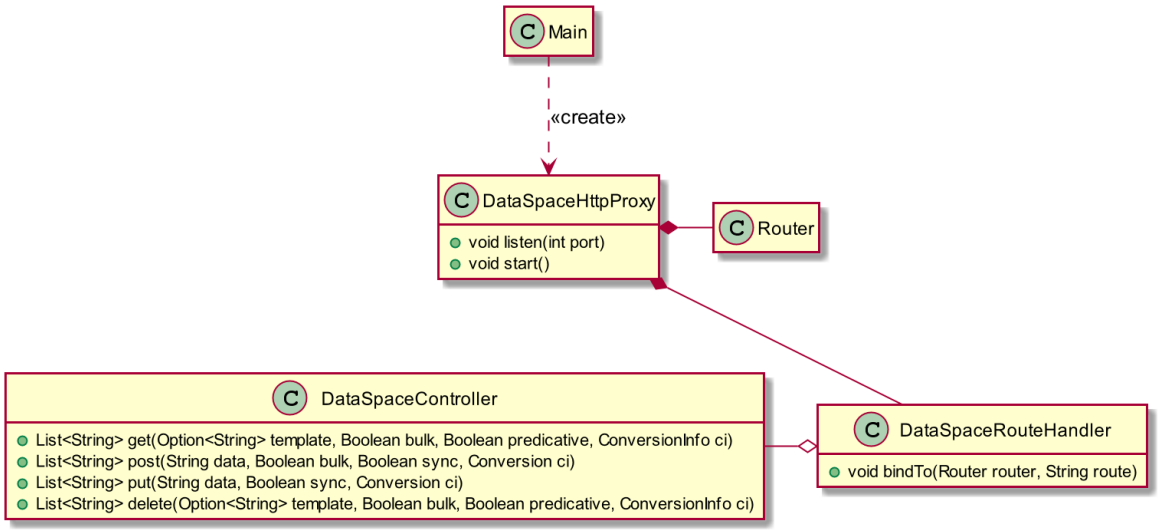


Figura 3.6: Diagramma delle classi che mostra l'architettura software del modulo web.

3.3.1 Architettura

In figura 3.6 é possibile trovare un diagramma contenente le astrazioni principali su cui si basa l'architettura del servizio web, mentre di seguito é riportata una descrizione circa lo scopo e il funzionamento di tali classi.

DataSpaceHttpProxy

```
class DataSpaceHttpProxy {
   Router router;
   List<DataSpaceRouteHandler> handlers;
   void listen(int port) { ... }
   void start() {
      ...
      for (DataSpaceRouteHandler h : handlers) {
         h.bindTo(router, ...);
      }
      ...
   }
}

class Main {
   DataSpaceHttpProxy server = new DataSpaceHttpProxy();
   server.listen(8080);
```

```
    server.start();
  }
```

Questa classe costituisce l'*entry point* del servizio web. Essa infatti consente infatti di effettuare il *deployment* del servizio a livello programmatico.

Sono stati concepiti due metodi. Il metodo `listen()` permette di specificare una porta sulla quale il servizio web deve mettersi in ascolto (tipicamente si userà la porta `8080`). Il metodo `start()`, invece, mette in esecuzione il servizio.

Questa classe deve possedere un oggetto di tipo *Router*, al quale verranno destinate tutte le richieste HTTP per poi essere smistate. Essa contiene anche un insieme di *DataSpaceRouteHandler*, ovvero di gestori di richieste. Sta alla classe qui descritta il compito di "agganciare" gli *handler* al *Router* tramite il metodo `bindTo()` fornito dalla classe *DataSpaceRouteHandler*. Tale associazione deve essere dichiarata all'interno della classe *DataSpaceHttpProxy*

Router

Questa classe é responsabile dell'indirizzamento delle richieste.

Il suo scopo é semplice: in base all'URI specificato nella richiesta HTTP il *Router* passa la richiesta all'*handler* apposito.

DataSpaceRouteHandler

Come introdotto nella sezione 3.1, questa classe ha il compito di estrarre tutte le informazioni utili da una richiesta incombente, individuando il nome del *dataspace* indirizzato dall'agente e passando i dati estratti al *DataSpaceController* corrispondente, in base al metodo HTTP usato.

Questa classe espone solo il metodo `bindTo()`, che permette al codice cliente, ovvero la classe *DataSpaceHttpProxy*, di associare un *handler* a un *Router*.

DataSpaceController

```
class DataSpaceController {
  private AsyncDataSpace ds;
  ...
  public List<String> getFromDataSpace(Option<String> template,
      Boolean bulk, Boolean predicative, ConversionInfo ci) {
        if (template.isDefined()) {
          if (bulk) {
            return ds.readAll(template.get());
```

```
            } else if (predicative) {
               return ds.readIfPresent(template.get()).map(d ->
                  d.toList()).get();
            } else {
               return ds.read(template.get).map(d -> new
                  List<String>(d)).get();
            }
         } else {
            return ds.get().get();
         }
      }
   ...
}
```

Questa classe ha la responsabilità di comandare uno spazio di tuple, invocando su di esso i metodi corrispondenti alle primitive di coordinazione.

La scelta della primitiva da invocare viene effettuata sulla base di due aspetti principali: il metodo HTTP usato nella richiesta e i valori dei parametri specificati dall'agente all'interno della query. Il metodo HTTP viene determinato già dal *DataSpaceRouteHandler*, il quale estrae dalla richiesta tale informazione e sceglie congruentemente il metodo da invocare sul *DataSpaceController*. I valori dei *query parameter*, invece, vengono passati al *controller* contestualmente all'invocazione dei metodi su di esso come parametri.

Oltre alle informazioni contenute nella query, i metodi di questa classe richiedono che venga specificato anche un parametro di tipo *ConversionInfo*, il quale deve essere fornito dal *DataSpaceRouteHandler* in base al contenuto degli header del messaggio.

3.3.2 Routing e gestione delle richieste

La gestione delle richieste viene effettuata dalla classe *DataSpaceRouteHandler*, la cui funzione consiste nell'estrarre i dati necessari da una richiesta HTTP per poi presentarli in modo appropriato a un *DataSpaceController*. Queste informazioni, tuttavia, non vengono estratte direttamente dal *DataSpaceRouteHandler*: esso, anzi, le riceve dal *Router* opportunamente incapsulate in un oggetto di tipo *Request*. Questo tipo di oggetti é generalmente contenuto all'interno di oggetti appartenenti alla classe *Context*, i quali possono essere visti come "contenitori" di tutte le informazioni inerenti a una singola interazione fra agente e servizio. Ogni *Context*, infatti, contiene anche un oggetto *Response* che serve alla costruzione del messaggio di risposta.

In figura 3.7 é possibile trovare un diagramma raffigurante le nuove astrazioni qui introdotte, assieme alle relazioni statiche intercorrenti fra di loro.

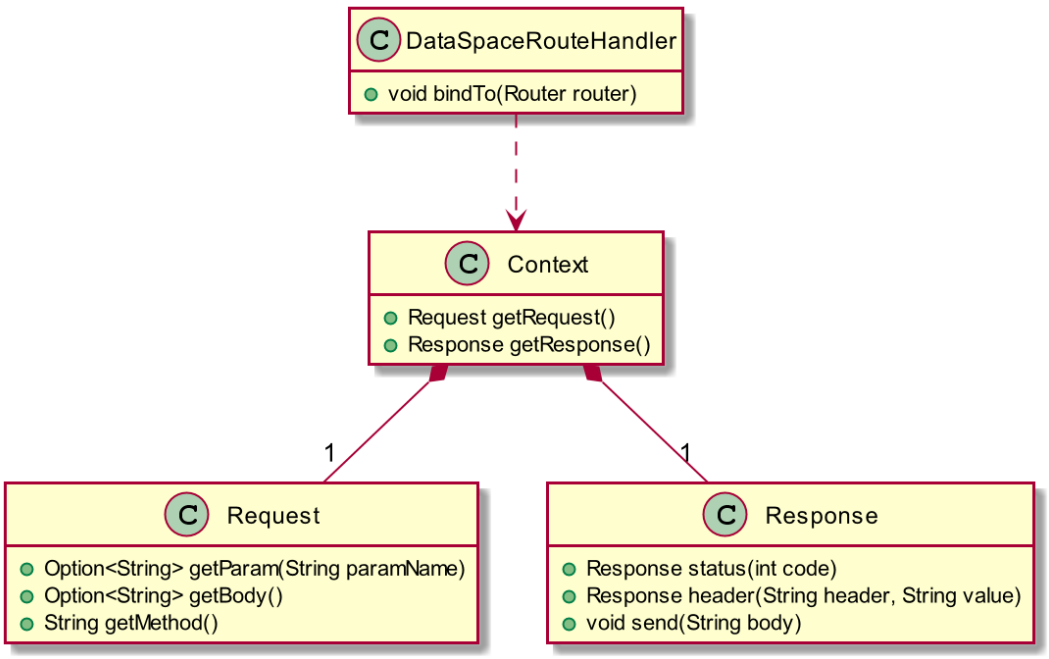


Figura 3.7: Diagramma delle classi che mostra l'architettura software del modulo web.

Context

I *Context* contengono le informazioni relative a un'interazione fra agente e servizio web di tipo richiesta-risposta. Ogni oggetto di tipo *Context* viene costruito da un *Router*, il quale estrapola le informazioni contenute in un messaggio HTTP in arrivo per poi inserirle in un oggetto di tipo *Request*. Ogni *Context*, inoltre, é in grado di costruire e impartire l'invio della risposta tramite una *Response*, i cui dati verranno successiamente utilizzati dal *Router* per l'effettiva redazione del messaggio HTTP da inviare all'agente coinvolto nell'interazione.

Questa classe espone principalmente due metodi *getter* che permettono al *DataSpaceRouteHandler* di accedere e manipolare la richiesta e la risposta relativi al contesto.

In figura 3.8 é possibile trovare un diagramma di sequenza che mostra il flusso informativo osservabile dall'arrivo di una richiesta fino all'invio della risposta.

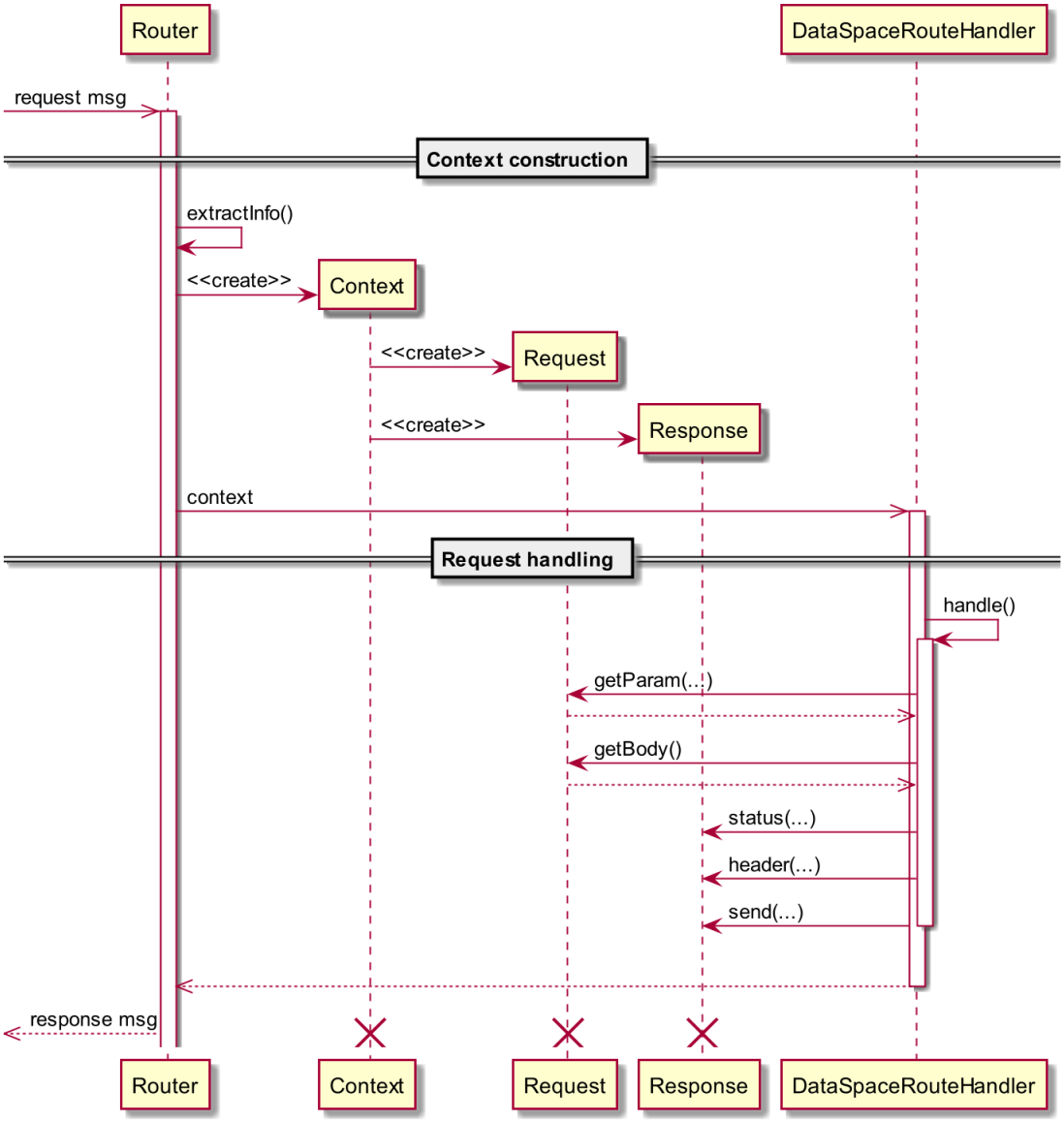


Figura 3.8: Diagramma di sequenza che mostra il processo di manipolazione delle richieste in arrivo e l'invio delle risposte.

Request

Questa classe costituisce un contenitore per le informazioni ricevute dal servizio tramite un messaggio HTTP.

La classe *Request* espone i seguenti metodi:

`getParam()` Dato il nome di un *query parameter*, restituisce un *optional* contenente il valore assunto da tale parametro all'interno della query. Se il parametro non é specificato o assume valore nullo (vuoto), viene restituito un *optional* vuoto;

`getBody()` Restituisce un *optional* con il contenuto del *body* della richiesta. Se il *body* é vuoto, restituisce un *optional* vuoto;

`getMethod()` Restituisce il nome del metodo HTTP usato nella richiesta.

Response

Questa classe permette al *DataSpaceRouteHandler* di creare una risposta da inviare all'agente coinvolto nell'interazione.

Il processo di costruzione della richiesta segue uno stile *fluent*, simile a quello che si può osservare usando il pattern *Builder* [8], del tipo:

```
new Response().status(200).header("Content-Type",
    "application/json").send(""msg":"Hello"");
```

La principale differenza con tale pattern consiste nel fatto che qui non risulta possibile differenziare la logica costruttiva dell'oggetto dalla sua rappresentazione, in quanto non si é reso necessario prevedere la presenza di più rappresentazioni delle *Response*. Tuttavia, l'iter creazionale di questo pattern rimane preservato, il che comporta una semplificazione sostanziale della creazione degli oggetti *Response*; inoltre, con questo approccio risulta possibile effettuare un controllo sull'effettiva integrità informativa dell'oggetto prima che questo diventi effettivamente "funzionante" e, quindi, utilizzabile.

La classe espone i seguenti metodi:

`status()` Imposta lo *status code* del messaggio HTTP, e restituisce l'oggetto *Response* in costruzione;

`header()` Imposta un *header* del messaggio HTTP dati il suo nome e il suo valore, e restituisce l'oggetto in costruzione.

`send()` Dato in input il contenuto del *body* del messaggio, termina la costruzione della risposta impartendone l'invio all'agente in attesa del completamento della richiesta.

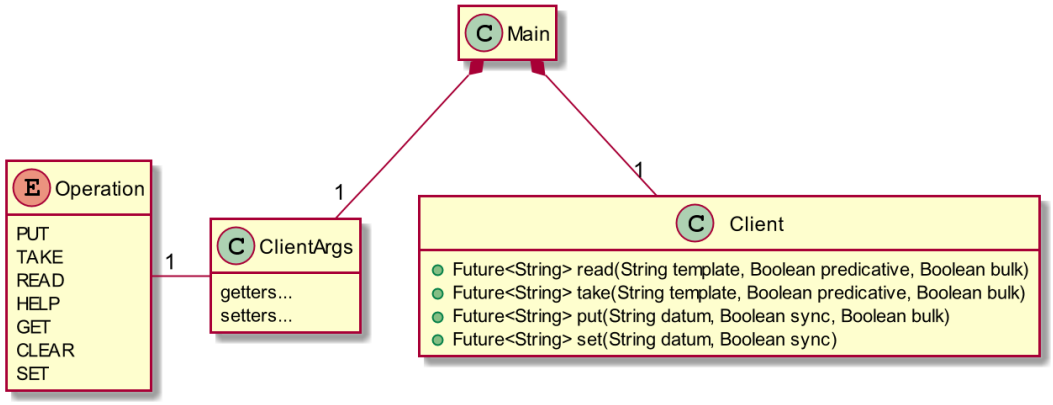


Figura 3.9: Diagramma delle classi che mostra l'architettura del client a riga di comando.

3.4 Modulo client

Il modulo client consta di una semplice applicazione a riga di comando che permette a un utente fisico di comunicare con qualsiasi dataspace in rete.

In questo capitolo verrà dapprima mostrata brevemente l'architettura del sistema; successivamente, saranno descritti i meccanismi tramite i quali l'utente potrà configurare il client per interagire con un *dataspace*.

3.4.1 Architettura

L'architettura di questo modulo risulta estremamente semplice, come è possibile notare dal diagramma delle classi in figura 3.9.

Viene riportata di seguito una breve descrizione delle astrazioni più importanti.

Main

Benché non rappresenti propriamente un'astrazione, la classe `Main` rappresenta l'*entry point* dell'applicazione.

A essa sta il compito di ricevere l'input dell'utente e di mostrare il risultato dell'invocazione dei vari comandi. A essa compete, inoltre, la gestione della logica di controllo; questo può apparentemente risultare un *design flaw*, ma viste le ridotte dimensioni del sistema in esame ciò può essere considerato una scelta condivisibile, data dalla necessità pratica di evitare di complicare inutilmente un'architettura molto semplice.

Questa classe, perciò, costituisce di fatto l'elemento strutturale che scandisce l'interazione con l'utente.

Client

Questa classe ha il compito di instaurare connessioni con i *dataspace* e di comunicare con loro tramite messaggi HTTP.

Essa espone un metodo per ogni "famiglia" di operazioni: ognuno di essi accetta come parametri una serie di informazioni necessarie all'invio del messaggio e, di conseguenza, all'invocazione della corrispondente primitiva a lato server. In virtù del comportamento asincrono delle operazioni sugli spazi di tuple in TuSoW, analogamente alla loro definizione nel modulo *core* – a tal proposito si rimanda il lettore alla sezione 3.2 – i metodi esposti da *Client* mostrano il costrutto *future* come tipo di ritorno. Questo indica che la computazione associata al metodo é potenzialmente *long-running* e/o bloccante.

ClientArgs

La classe *ClientArgs* contiene una serie di informazioni che rappresentano la configurazione dell'applicazione client.

A ogni elemento di configurazione sono associati un *getter* e un *setter* (non inclusi in figura 3.9 a causa di ovvie necessità tipografiche). La configurazione prevede:

`url`	Il percorso del *dataspace* all'interno del servizio (nome escluso);
`port`	La porta a cui si desidera connettersi: di default é `8080`;
`format`	Il formato in cui é rappresentata la tupla/template in input all'operazione;
`dataspace`	Il nome del dataspace al quale si desidera connettersi;
`operation`	L'operazione che si desidera invocare;
`argument`	L'argomento dell'operazione;
`predicative`	Valore booleano che indica se l'operazione é predicativa;
`bulk`	Valore booleano che indica se l'operazione é di tipo *bulk*;
`sync`	Valore booleano che indica se l'operazione é sincrona.

Alla classe *ClientArgs* é inoltre affidata la responsabilità di effettuare il *parsing* dei comandi impartiti dall'utente al sistema. Questa operazione dovrebbe avvenire a livello di costruttore, il quale prenderebbe in input una stringa contenente il comando.

Operation

Questa enumerazione contiene un elemento per ogni tipo di operazione che é possibile eseguire da riga di comando. Comprende tutte le primitive supportate in TuSoW, più un elemento `HELP` che rappresenta l'operazione in cui si chiede al client di stampare lo *usage* dell'applicativo.

3.4.2 Scenario esecutivo

In figura 3.10 si può trovare un diagramma di sequenza che descrive le interazioni fra l'utente e i principali componenti del sistema.

Questo scenario descrive l'invocazione della primitiva `read()`. Dapprima, la classe *Main* (che fa da *boundary*) in risposta all'input dell'utente ordina alla classe *ClientArgs* il *parsing* del comando inserito, tramite il costruttore di quest'ultima. A questo punto le informazioni contenute nel comando sono racchiuse in una struttura astratta, intellegibile al sistema; il *Main* avvia quindi la fase di gestione dell'input, durante la quale viene innanzitutto determinato il tipo di operazione che dovrà essere eseguita.

Nel caso raffigurato, si assume che l'utente abbia scelto una *Operation* di tipo `READ`. Di conseguenza, viene invocato il rispettivo metodo sul *Client*, il quale si occupa di instaurare e condurre una comunicazione HTTP con un servizio TuSoW residente su un qualche dispositivo in rete.

Una volta ricevuta un messaggio di risposta, il risultato viene quindi retropropagato fino ad essere stampato sull'interfaccia utente. A questo punto, il ciclo può ricominciare, dando all'utente la possibilità di impartire al sistema un nuovo comando.

Pur non essendo mostrate in figura, si prevede la possibilità di impartire comandi che non causino necessariamente l'invocazione di operazioni di coordinazione, ma che servano semplicemente a modificare la configurazione dell'applicativo, come, ad esempio, il nome del *dataspace*, il formato e altri parametri.

3.5 Tecnologie

In questa sezione verranno presentate brevemente le tecnologie utilizzate per l'implementazione dei vari moduli software che compongono il middleware TuSoW.

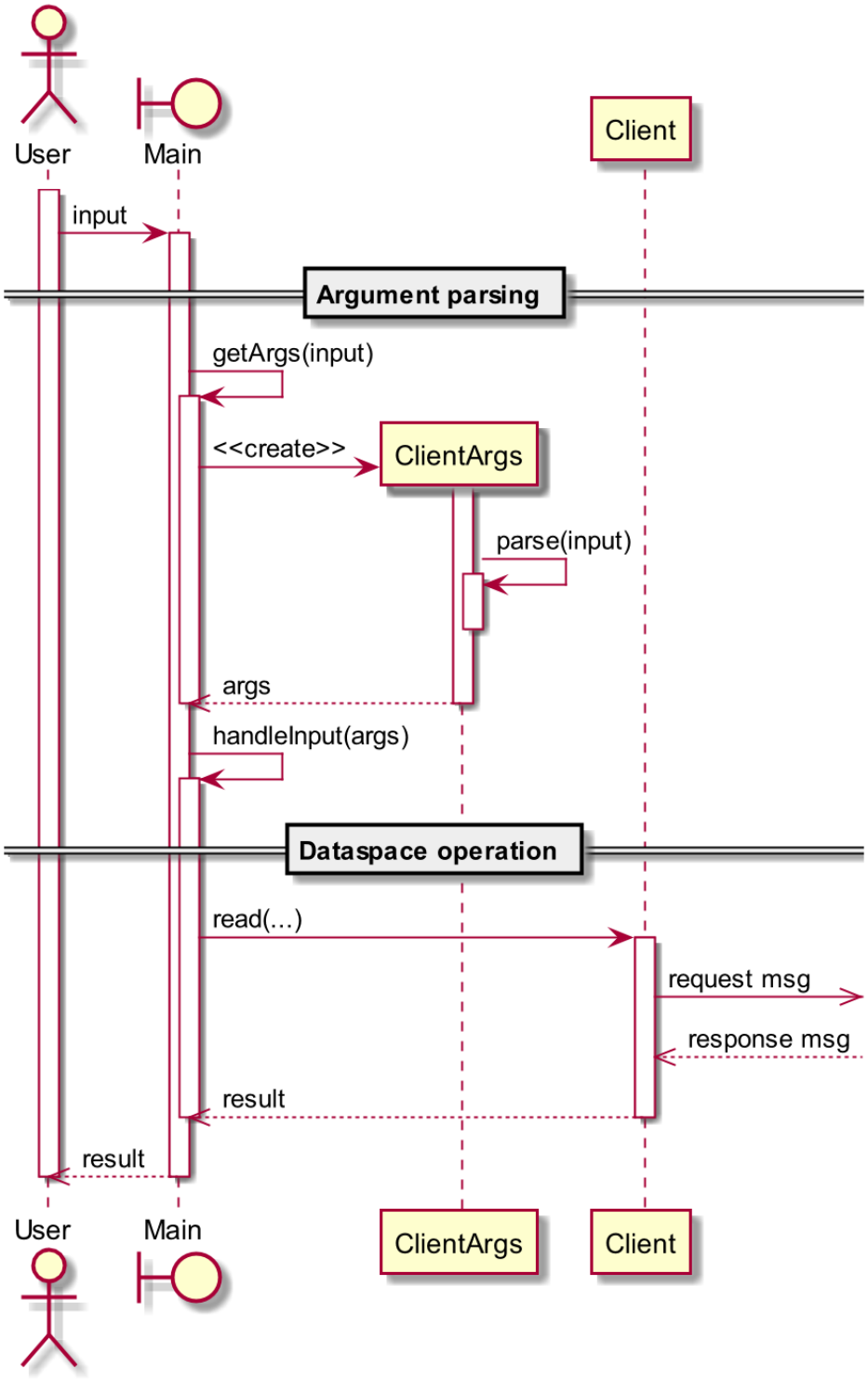


Figura 3.10: Diagramma di sequenza che mostra il processo di invocazione di una primitiva su un *dataspace* in rete.

3.5.1 Scala

Scala[1] é un linguaggio di programmazione multi-paradigma che supporta principalmente la programmazione funzionale e la programmazione ad oggetti, fornendo una tipizzazione statica forte. I programmi scritti in Scala compilano in *bytecode* Java, per cui vengono eseguiti sulla Java Virtual Machine e risultano di fatto compatibili e interoperabili con tutti quei linguaggi ad essa affini, come ad esempio Java e Kotlin.

Si é scelto di scrivere l'intero middleware in tale linguaggio, in quanto risulta essere un linguaggio moderno, interoperabile con tecnologie *mainstream* e basato, di fatto, sul paradigma funzionale, il che ha avuto il pregio di rendere TuSoW un prodotto software dal codice potenzialmente elegante e conciso.

3.5.2 Vert.x

Vert.x[2] é un framework basato sulla Java Virtual Machine dalla politica *event-driven* e *non bloccante* che permette di costruire applicazioni concorrenti reattive. Esso é considerato un framework *unopinionated*, nel senso che permette agli sviluppatori una grande libertà di progettazione, senza imporre alcuna metodologia di utilizzo delle proprie funzionalità, anzi permettendone l'uso nella più totale libertà.

Vert.x é inoltre un framework poliglotta, poichè risulta compatibile con un largo insieme di linguaggi di programmazione come Java, Scala, Kotlin, Javascript, Groovy, Ruby, ecc.

Vert.x possiede, inoltre, il grande pregio di avere un alto grado di modularità e granularità, in quanto espone le sue funzionalità tramite una lunga serie di moduli software (es. core, web, IoT, ecc.) permettendo quindi agli sviluppatori di essere minimamente dipendenti da questa tecnologia, e solo nella misura in cui ciò é strettamente necessario.

Questo framework si é rivelato estremamente utile nell'implementazione del modulo web, grazie alla facilità di definizione dei meccanismi di *content negotiation*, dell'implementazione delle API e di indirizzamento delle richieste.

3.5.3 JsonPath

La libreria JsonPath[3] (implementazione della specifica JSONPath[4]) mette a disposizione un linguaggio di *querying* che permette di effettuare letture su

[1] https://www.scala-lang.org/
[2] https://vertx.io/
[3] https://github.com/json-path/JsonPath
[4] http://goessner.net/articles/JsonPath/

documenti scritti in formato JSON, fortemente ispirato al linguaggio XPath[5], il quale opera invece su documenti XML.

JsonPath é stato usato come linguaggio di rappresentazione dei template nella coppia di linguaggi ⟨JSON, JsonPath⟩.

3.5.4 tuProlog

tuProlog [5] é un'implementazione del linguaggio Prolog leggera e pensata per essere applicata all'interno delle applicazioni e dei sistemi distribuiti. Essa é usufruibile sia sotto forma di libreria, sia tramite un IDE interattivo[6].

Si é scelto tuProlog come motore Prolog per l'implementazione del *matching* nella coppia di linguaggi ⟨Prolog, Prolog⟩.

3.5.5 Scopt

Scopt[7] é una semplice libreria che permette di effettuare il parsing di comandi e argomenti da un'interfaccia CLI. Si é rivelata fondamentale durante l'implementazione del modulo client di TuSoW.

[5]https://www.w3.org/TR/xpath/all/
[6]http://apice.unibo.it/xwiki/bin/view/Tuprolog/WebHome
[7]https://github.com/scopt/scopt

Conclusioni e sviluppi futuri

Nell'introduzione a questo volume si é discusso su come una moderna implementazione del modello LINDA debba seguire, al giorno d'oggi, una serie di passaggi progettuali atti ad assicurare lo sviluppo di una tecnologia di coordinazione competitiva e adoperabile in contesti applicativi reali. Si é ipotizzato il concepimento degli spazi di tuple come risorse web, e sono stati introdotti i benefici della formulazione di una REST API che permetta di interagire con essi tramite un'interfaccia pubblica e ben definita. Conseguentemente, si é visto come la messa a disposizione di spazi di tuple che supportino diverse tecnologie di rappresentazione dei dati permetta un alto grado di interoperabilità e consenta l'interazione fra dispositivi di natura fortemente eterogenea. Infine, é stato formulato l'obiettivo di costruire un sistema di coordinazione in grado di parlare più linguaggi di comunicazione contemporaneamente, implementando di fatto un meccanismo di traduzione trasparente all'interazione.

Si può affermare che questi tre requisiti fondamentali siano stati rispettati. Le scelte progettuali (cap. 3) effettuate dimostrano chiaramente che la costruzione di un servizio di coordinazione moderno e competitivo é senz'altro possibile e risulta traducibile in un'architettura software sufficientemente semplice, nonché ragionevolemente estendibile.

Tuttavia, le soluzioni architetturali proposte non sono di certo le uniche possibili. Infatti, i formalismi introdotti nel capitolo 2 – successivamente reificati a livello progettuale/implementativo – naturalmente possono essere considerati *project-independent*. I modelli matematici proposti potranno certamente essere adottati come riferimento qualora si presenti, in futuro, la necessità di sviluppare nuovi sistemi di coordinazione dalle caratteristiche simili a questo.

È inutile affermare che ci sia ancora molto lavoro da fare prima che TuSoW diventi un servizio di coordinazione veramente completo, sebbene l'introduzione di nuove primitive, le API pubbliche e la semplicità architetturale lo rendano un middleware tutto sommato leggero e mantenibile. È proprio l'alto grado di mantenibilità di questo software che servirà ad incoraggiare sviluppi futuri: una prima estensione potrebbe nascere da un'analisi più approfondita del grado di espressività dei vari linguaggi di comunicazione e delle rispettive tecnologie di *querying* disponibili in letteratura e sul Web, in modo da valutare

la fattibilità di nuovi *mapping* linguistici e, di conseguenza, realizzarli.

Un secondo sviluppo invece potrebbe consistere nell'introduzione di una serie di primitive cosiddette di "meta-coordinazione", fornendo di fatto la possibilità di creare nuovi spazi di tuple sulla macchina che ospita il servizio; altrimenti, si potrebbero inserire primitive che permettano di copiare e/o trasferire tuple da uno spazio all'altro.

Un terzo sviluppo avrebbe come obiettivo l'introduzione di un meccanismo di *federazione distribuita* che permetta di concepire più spazi di tuple come uno unico, indirizzabile tramite un singolo nome.

Un quarto sviluppo sarebbe mirato ad astrarre adeguatamente il concetto di *interaction space*, introducendolo nell'architettura: in questo modo si potrebbero rendere *programmabili* gli spazi i tuple tramite opportuni linguaggi di specifica delle reazioni.

Un quinto e ultimo sviluppo, in conclusione, vedrebbe come scopo l'implementazione di TuSoW nell'ambito dei sistemi mobili – *in primis* su Android – tramite l'adozione di tecnologie di comunicazione standard come Bluetooth, attualmente dominante nel mercato *consumer*, ma anche il più recente Wi-Fi Direct, il quale permetterebbe di godere delle performance tipiche della tecnologia Wi-Fi, eliminando però il bisogno degli *access point*; si ridurrebbero i costi e si aumenterebbe considerevolmente la mobilità dei dispositivi.

Bibliografia

[1] ReSTful API Design. `https://restful-api-design.readthedocs.io/en/latest/resources.html`.

[2] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the expressiveness of linda coordination primitives. *Information and Computation*, 156(1-2):90–121, 2000.

[3] Nicholas John Carriero, Jr. *Implementation of Tuple Space Machines*. PhD thesis, New Haven, CT, USA, 1987. AAI8809192.

[4] Giovanni Ciatto, Stefano Mariani, Maxime Louvel, Andrea Omicini, and Franco Zambonelli. Twenty years of coordination technologies: State-of-the-art and perspectives. In Giovanna Di Marzo Serugendo and Michele Loreti, editors, *Coordination Models and Languages*, pages 51–80, Cham, 2018. Springer International Publishing.

[5] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuprolog: A light-weight prolog for internet applications and infrastructures. In *International Symposium on Practical Aspects of Declarative Languages*, pages 184–198. Springer, 2001.

[6] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[7] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985.

[8] Richard Helm, Ralph E Johnson, Erich Gamma, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Braille Jymico Incorporated, 2000.

[9] Andrea Omicini. On the semantics of tuple-based coordination models. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, SAC '99, pages 175–182, New York, NY, USA, 1999. ACM.

[10] Antony Ian Taylor Rowstron. *Bulk primitives in Linda run-time systems.* University of York, Department of Computer Science, 1996.

[11] Robert Tolksdorf and Dirk Glaubitz. Coordinating web-based systems with documents in xmlspaces. In *International Conference on Cooperative Information Systems*, pages 356–370. Springer, 2001.

[12] Mirko Viroli and Andrea Omicini. Coordination as a service. *Fundam. Inf.*, 73(4):507–534, September 2006.