ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE
IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Sistemi Distribuiti

Agenti, programmazione logica e sistemi distribuiti: esperimenti in JADE e tuProlog

CANDIDATO
Alberto Sita

RELATORE:
Chiar.mo Prof. Andrea Omicini

CORRELATORI
Chiar.mo Prof. Enrico Denti
Ing. Roberta Calegari

Anno Accademico 2016/17

Sessione I

Alla mia famiglia che mi ha sempre sostenuto,
a Francesca che mi ha sempre ascoltato
e a tutte le persone che hanno reso possibile questo lavoro.

Indice

1. Introduzione

Negli ultimi anni il settore dell'*Internet of Things* (IoT) è stato in forte espansione sia in ambito di ricerca e sviluppo sia in ambito commerciale[IOTG]. In passato le potenzialità di Internet erano prerogativa dei soli computer. Con la progressiva miniaturizzazione dei componenti elettronici ed il miglioramento dei consumi energetici, oggigiorno non si hanno solamente dispositivi portatili connessi ad Internet (laptop, smartphone e wearables) ma anche dispositivi embedded come sensori, controllori ed attuatori, presenti ad esempio in lampadine, termostati, dispositivi medici e di sorveglianza.

Connettere ad Internet oggetti che circondano l'uomo nelle sue attività quotidiane crea sistemi distribuiti complessi in cui la tecnologia non si limita ad essere una commodity volta a migliorare la vita dell'individuo, ma diventa una componente attiva, sempre presente, con la quale l'uomo può interagire e collaborare. L'IoT rende pervasiva la tecnologia e consente di avere informazioni sul contesto e sulle abitudini del singolo individuo che possono essere utilizzate per realizzare sistemi intelligenti (*Internet of Intelligent Things,* IoIT)[IOIT]. Scenari concreti sono ad esempio reti per la domotica, sistemi di assistenza alla guida, sistemi per il controllo del traffico all'interno di un ambiente smart city, ecc. Da un punto di vista ingegneristico, lo sviluppo di applicazioni per sistemi IoT richiede la presenza di un supporto software che astragga dalle problematiche riguardanti la distribuzione delle risorse computazionali, che doti di intelligenza il sistema, che sia configurabile in modo dichiarativo e che permetta allo sviluppatore di concentrarsi soltanto sulla progettazione della logica applicativa. Tale supporto middleware si può parzialmente identificare in una piattaforma ad agenti che consente lo sviluppo di software basato sull'astrazione dell'agente, un'entità autonoma, socievole, reattiva e situata nello spazio.

In questo contesto si colloca la tesi: partendo dall'esigenza dei sistemi distribuiti di avere a disposizione intelligenza situata, ma anche interoperabilità, configurabilità e coordinazione, scopo del lavoro è cercare di unire la programmazione ad agenti e la programmazione logica per sviluppare efficacemente applicazioni IoT intelligenti e distribuite. Inoltre si vuole sperimentare questo approccio come primo passo verso la definizione di un'architettura middleware che consenta la gestione della conoscenza e dell'intelligenza di un sistema distribuito.

Nella trattazione degli argomenti si procederà con una breve presentazione della programmazione ad agenti in Java, della programmazione logica e della sua re-interpretazione come servizio, cioè della *Logic Programming as a Service* (LPaaS, capitolo 2)[LPaaS]. Poi si approfondirà in dettaglio l'obiettivo del lavoro e si tratterà di come sia necessario utilizzare la programmazione multi-paradigma nella

realizzazione di applicativi IoT intelligenti (capitolo 3). Successivamente si presenterà una soluzione per integrare la programmazione ad agenti Java alla programmazione logica e se ne discuterà l'implementazione (capitoli 4, 5). L'attenzione sarà posta principalmente sul supporto software da sviluppare per consentire l'utilizzo di LPaaS all'interno di un sistema distribuito multi-agente. Inoltre si parlerà delle estensioni apportate al motore inferenziale *tuProlog*[2PL] per poterlo utilizzare in un contesto distribuito (capitolo 6)[2PARCH]. Si presenterà poi un caso di studio applicativo che utilizza il software sviluppato. Il caso di studio ha come soggetto una cucina intelligente in cui un frigorifero, un frullatore, un forno ed una dispensa (dispositivi IoT) forniscono informazioni ad una applicazione client via LPaaS sulle scorte di cibo e sulle abitudini alimentari degli utenti. L'applicazione client, sfruttando la programmazione logica, è in grado di fare inferenze sui dati raccolti dai dispositivi IoT (capitolo 7). Si concluderà discutendo i pregi, le mancanze e le possibili estensioni future del supporto software sviluppato (capitolo 8).

2. Background

2.1 Programmazione ad agenti

Il concetto di *agente software* è nato all'inizio degli anni '80 come punto d'incontro tra il campo di ricerca dell'intelligenza artificiale e quello dell'informatica. Attualmente non esiste una definizione universalmente accettata di cosa si intenda effettivamente con il termine *agente software*. Volendone comunque dare una definizione intuitiva, esso può essere identificato come software in grado di agire autonomamente, non dotato di un'interfaccia e non invocabile direttamente da altro codice. Un agente è quindi un'entità attiva, situata, che incapsula un flusso di controllo ed è capace di agire sull'ambiente che lo circonda.

2.1.1 Proprietà di un agente software

Un agente software si caratterizza principalmente per le seguenti proprietà:

Situatedness: indica che un agente è collocato in un determinato ambiente e che è sensibile ai suoi cambiamenti.

Autonomia: caratterizza il grado di libertà che possiede l'agente nel prendere decisioni riguardo alle azioni che deve compiere. È anche un sinonimo di intelligenza.

Proattività: è la capacità di un agente di prendere decisioni autonome a prescindere dagli stimoli che provengono dall'ambiente che lo circonda. Un agente può quindi avere un comportamento *goal-oriented*.

Reattività: è la capacità di un agente di reagire ai cambiamenti dell'ambiente in cui esegue.

Abilità sociali: gli agenti sono in grado di interagire tra loro ed eventualmente anche con gli esseri umani. La comunicazione deve avvenire attraverso linguaggi con semantiche concordate in precedenza.

Mobilità: gli agenti sono in grado di migrare da un nodo computazionale ad un altro.

Persistenza: nel caso di sospensione dell'esecuzione, l'agente è in grado di salvare il proprio stato all'interno dell'ambiente in cui è situato. In questo modo è in grado di continuare la propria esecuzione quando viene riattivato.

Adattività: indica la capacità di un agente di apprendere sia dalle proprie azioni che dall'ambiente che lo circonda. Un agente è quindi in grado di migliorare nel tempo la propria base di conoscenza.

Benevolenza: un agente prova sempre ad eseguire il compito che gli è stato affidato controllando che i goal assegnati non siano in contrasto tra loro.

Fidatezza: un agente si definisce fidato se fornisce la garanzia di non comunicare a terzi informazioni riservate.

Cooperazione: un agente interagisce con altri agenti per raggiungere uno scopo comune. In questo modo un agente può ottimizzare le proprie operazioni ed offrire servizi ad altri agenti presenti nell'ambiente in cui esegue. È possibile anche che vi sia *competitività* tra agenti nel caso in cui essi tentino di accedere contemporaneamente alle stesse risorse.

È possibile classificare gli agenti in due categorie: *deboli*, se possiedono solo le prime sette proprietà, altrimenti si definiscono *forti*. In seguito, con il termine agente si intenderà sempre un agente *forte*.

2.1.2 Lo standard FIPA

Lo standard de facto per l'interoperabilità tra agenti è il *Foundation for Intelligent and Physical Agents* (FIPA)[FIPA]. Esso è stato elaborato dall'omonimo consorzio, di cui fanno parte numerose multinazionali che operano nel campo dell'informatica e dell'elettronica e dal 2005 è entrato a far parte dell'IEEE come Standards Committe. Come FIPA stesso dichiara, lo scopo dello standard è di definire un'interfaccia comune per le diverse entità che popolano l'ambiente in cui gli agenti software operano, in modo da permettere la comunicazione tra agenti di diverse piattaforme, ed anche tra agenti e software basato su altre astrazioni.

In particolare, lo standard FIPA coinvolge i seguenti ambiti:

Applications: è costituito da un insieme di esempi di applicazioni, per ognuna delle quali FIPA propone un insieme minimo di servizi che l'agente dovrebbe essere in grado di fornire.

Abstract Architecture: definisce, a livello astratto, gli elementi essenziali per la realizzazione di una piattaforma ad agenti software, le relazioni tra essi ed indica alcune linee guida per la loro implementazione.

Agent Communication: riguarda un insieme di specifiche per la comunicazione tra agenti. Su questa tematica FIPA ha elaborato l'*Agent Communication Language* (ACL), un linguaggio standard che prevede protocolli e schemi di interazione per lo scambio di informazioni tra agenti[ACL].

Agent Management: è una serie di linee guida per la gestione e il controllo degli agenti e del loro comportamento, sia quando essi si trovano all'interno di una piattaforma sia durante il transito da una piattaforma all'altra.

Agent Message Transport: definisce le modalità standard per il trasferimento e la rappresentazione delle informazioni attraverso reti disomogenee per quanto riguarda i protocolli di trasporto adottati dalle piattaforme ad agenti.

2.1.3 Piattaforme ad agenti Java

Le principali piattaforme ad agenti sviluppate in Java sono:

Aglets: è una piattaforma per agenti mobili sviluppata dai laboratori di ricerca IBM[AGL]. È open-source e non soddisfa le specifiche dello standard FIPA.

Cougaar: sviluppata dal *Defence Advanced Research Projects Agency* (DARPA)[DARPA], è una piattaforma ad agenti mobili dotata di una buona documentazione, numerosi add-on e di una comunità open-souce molto attiva[COUG]. Non aderisce allo standard FIPA.

Semoa: è una piattaforma per agenti mobili realizzata dall'azienda tedesca *Fraunhofer IGD*[SEM] ed è attualmente disponibile per il download pubblico sotto licenza LGPL. Ha fatto della sicurezza degli agenti e delle loro comunicazioni il suo punto di forza: sono disponibili meccanismi avanzati di crittografia e di gestione di certificati per la rilevazione di *malicious host*, così come per la difesa da attacchi di tipo *denial-of-service*, ed anche per garantire l'integrità delle comunicazioni tra gli agenti. Semoa è conforme allo standard FIPA.

Agent Factory: è stata sviluppata dall'Università di Dublino nell'ambito di ricerca sulle possibili applicazioni degli agenti mobili in settori come il mobile computing, la robotica e reti intelligenti di sensori[AG.FACT]. La caratteristica principale della piattaforma è quella di poter sviluppare rapidamente sistemi ad agenti; essa è fornita di numerosi modelli predefiniti di reti ad agenti mobili che possono poi essere composti per crearne di nuovi e di maggiore complessità. È resa accessibile al pubblico tramite licenza LGPL ed è conforme allo standard FIPA.

JADE: è la piattaforma utilizzata per lo sviluppo del software di questa tesi. Aderisce pienamente allo standard FIPA, la sua architettura verrà presentata al paragrafo 2.1.4.

2.1.3.1 Confronto tra le piattaforme

Nella scelta della piattaforma ad agenti inizialmente si sono preferite quelle conformi allo standard FIPA in quanto permettono lo sviluppo di sistemi ad agenti facilmente interoperabili con altre entità. Si sono quindi scartate *Aglets* e *Cougaar* in quanto non conformi allo standard. Si è infine preferito JADE alle altre piattaforme principalmente perché è il sistema che ha maggiore diffusione, dispone di una buona documentazione e di una comunità di sviluppatori molto attiva.

2.1.4 JADE

Java Agent DEvelopment Framework (JADE) è una piattaforma ad agenti mobili sviluppata dai laboratori di ricerca Telecom di Torino[JADE]. Il progetto, nato alla fine del 1998, è stato reso disponibile per il download all'inizio del 2000. Poco dopo, il team di sviluppo decise di aderire allo standard FIPA e rese open-source il software, rilasciandolo sotto licenza LGPL. Da quel momento si è formata una numerosa comunità di programmatori che ha contribuito allo sviluppo di JADE ed ha creato numerose estensioni della piattaforma. Il rilascio sotto licenza LGPL non ha comportato l'abbandono del progetto da parte dei laboratori Telecom che, ancora oggi, contribuiscono attivamente al suo sviluppo.

2.1.4.1 Elementi di base dell'architettura

JADE si compone degli elementi architetturali mostrati in *figura 1*:

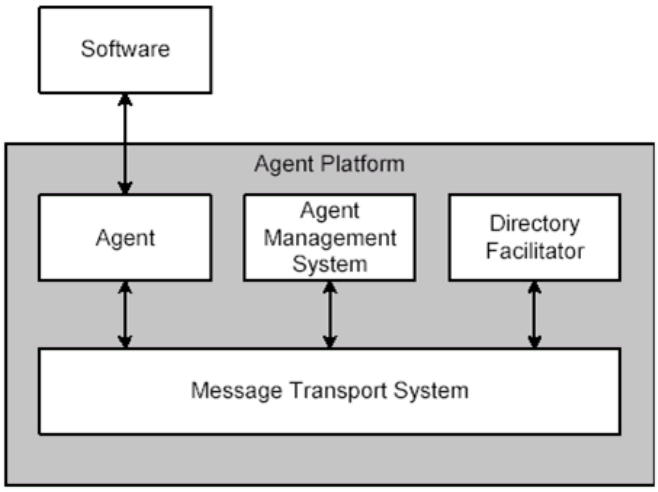

figura 1 – componenti della piattaforma JADE.

Agente: rappresenta l'entità principale che opera all'interno della piattaforma, ed è in grado di interagire con altri agenti, con la piattaforma o con software esterno ad essa. Ogni agente ha un proprietario (*owner*), che può essere un utente umano oppure un'entità software. Ciascun agente possiede anche un *Agent IDentifier* (AID), un identificatore univoco che permette di individuarlo all'interno della piattaforma in cui opera; esso è in genere costituito da una stringa, da un numero o da una combinazione di questi due elementi. Inoltre, poiché un agente può spostarsi da un nodo ad un altro, possiede anche un indirizzo di trasporto, che lo identifica univocamente all'interno della rete in cui esegue. Infine un agente JADE può essere considerato *forte* in quanto può garantire tutte le proprietà elencate al paragrafo 2.1.1.

Agent Management System (AMS): è un componente che supervisiona gli accessi alla piattaforma da parte di entità esterne e monitora le azioni che esse compiono. Ogni agente che vuole operare in una piattaforma JADE deve ricevere il consenso da parte dell'AMS. Secondo le specifiche FIPA all'interno di una piattaforma deve essere presente un solo AMS.

Directory Facilitator (DF): come per l'AMS, anche questo componente è presente all'interno di una piattaforma JADE, ma ve ne possono essere molteplici. Il suo compito principale è quello di fornire il servizio di pagine gialle (*rendez-vous*): mantiene cioè al suo interno un elenco dei servizi offerti da ogni agente presente nella piattaforma. Un agente che offre un servizio deve registrarsi al DF che provvederà a memorizzare tale informazione e a renderla disponibile a terzi. Simmetricamente, un altro agente che necessita di un servizio può effettuare una query di discovery al DF per conoscere l'identità e la locazione dell'agente che lo rende disponibile.

Message Transport System (MTS): ha il compito di gestire e coordinare lo scambio di informazioni tra agenti; questi ultimi possono essere situati nella stessa piattaforma o anche in piattaforme diverse.

Software Esterno: indica tutto il software non basato sulla tecnologia ad agenti, ma che offre servizi a cui gli agenti possono accedere.

Agent Platform: è la piattaforma per gli agenti mobili, ovvero un supporto middleware che fornisce un ambiente uniforme in cui gli agenti possono svolgere i propri compiti, comunicare e spostarsi.

Una piattaforma JADE si compone di uno o più *container*, ovvero di entità in grado di ospitare agenti al proprio interno. Tra di essi il *Main Container* ha il compito di controllare e coordinare il funzionamento degli altri contenitori, detti anche contenitori periferici. È importante sottolineare che esiste una corrispondenza univoca tra un contenitore principale ed una piattaforma JADE (*figura 2*); se si avvia un altro *Main Container*, esso costituisce un'altra piattaforma JADE, indipendente ed esterna alla precedente. Ciò che rende il contenitore principale indispensabile per il funzionamento della piattaforma è la presenza obbligatoria al suo interno dell'AMS e di almeno un DF: essi sono stati implementati in JADE come agenti software e vengono istanziati automaticamente all'avvio della piattaforma.

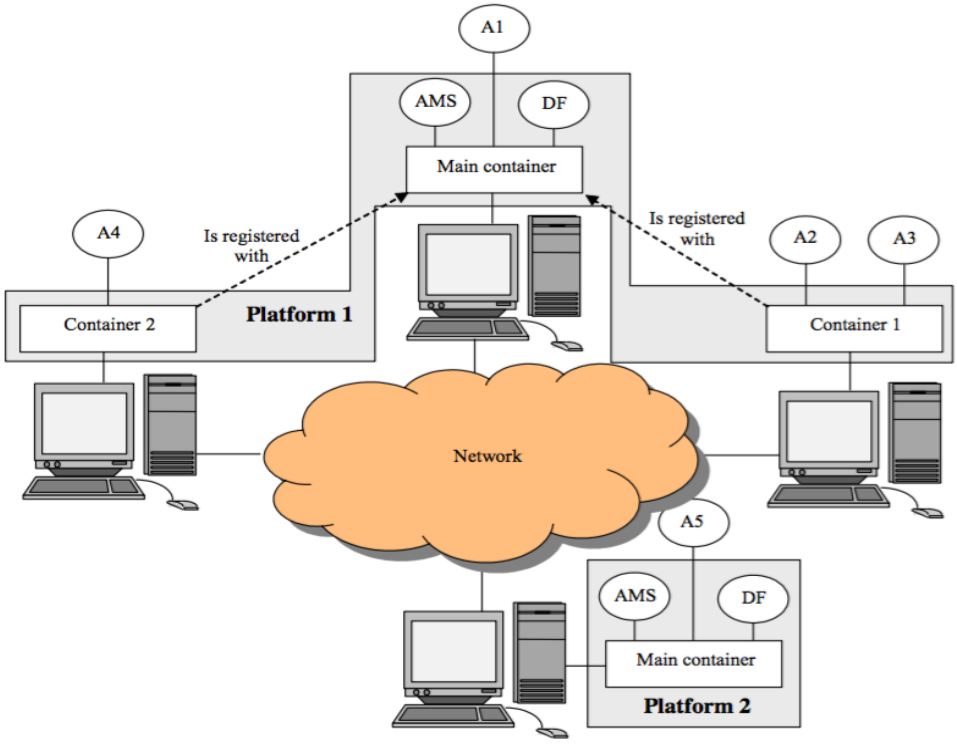

figura 2 – esempio di deployment di un sistema ad agenti JADE con due piattaforme.

2.1.4.2 Comunicazione a scambio di messaggi (ACL)

Un'abilità fondamentale di cui sono dotati gli agenti è quella di poter comunicare tra loro: in JADE questo avviene tramite unità informative discrete chiamate messaggi. In particolare, JADE ha implementato il formato ACL definito dallo standard FIPA[ACL] e supporta i protocolli di trasporto da esso definiti: quale venga utilizzato per la trasmissione dei messaggi dipende da dove sono situati i due agenti che vogliono comunicare. Si hanno tre casi distinti a seconda che i due agenti che comunicano si trovino nello stesso contenitore, in due contenitori diversi all'interno della stessa piattaforma, o in due piattaforme diverse.

Nel caso di comunicazioni interne ad un contenitore, JADE utilizza il protocollo *Internal Message Transport Protocol* (IMTP); esso è utilizzato anche per il trasporto di comandi interni per la gestione della piattaforma e per il monitoraggio dello stato dei container periferici. Nel caso invece di comunicazioni tra due agenti situati in due contenitori distinti, sia all'interno della stessa piattaforma sia in piattaforme diverse, viene utilizzata la *Remote Method Invocation* (RMI) di Java. Grazie a RMI, è possibile ottenere un riferimento ad un oggetto in modo trasparente rispetto al fatto che si trovi in un altro contenitore: in questo caso, il contenitore che ospita l'agente destinatario del messaggio è un server RMI, mentre quello che contiene il mittente svolge il ruolo di client RMI.

Con riferimento alla *figura 3*, si può notare come un messaggio ACL sia diviso in due parti: la prima, chiamata *Envelope*, contiene informazioni utili per la consegna del messaggio, come ad esempio mittente, destinatario, codifica utilizzata, etc. La seconda, chiamata *Payload*, contiene invece il vero e proprio messaggio (*Message parameters* e *Message content*) che si vuole inviare.

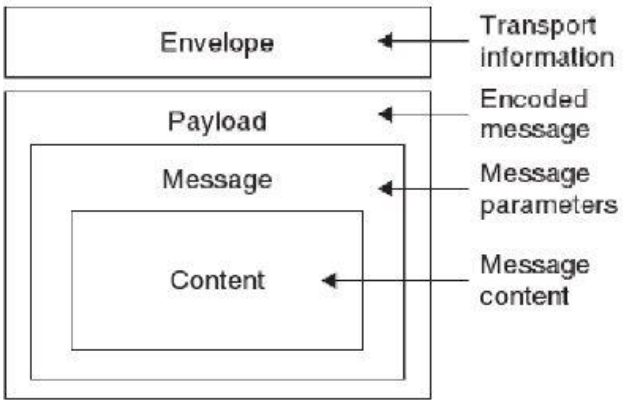

figura 3 – struttura di un messaggio ACL.

2.1.4.3 Ciclo di vita degli agenti

La piattaforma JADE adotta un modello di programmazione a *componente-container*: l'agente è un componente che vive all'interno di un container che ne gestisce il ciclo di vita. Il programmatore deve inserire la propria logica applicativa all'interno di determinati metodi di callback che vengono chiamati dal container JADE quando l'agente cambia il proprio stato di esecuzione. La *figura 4* schematizza il ciclo di vita di un agente JADE: il sistema crea l'oggetto agente e gli assegna un unico thread di esecuzione. Successivamente la piattaforma invoca il metodo *setup()*: qui è possibile inserire la configurazione iniziale dell'agente. A meno di errori avvenuti durante la configurazione, il container procede ad estrarre il primo *behaviour* e ad eseguirlo. Il sistema controlla poi se è stato invocato il metodo *doDelete()* dal codice applicativo: in caso affermativo il container invoca il metodo *takeDown()* per terminare l'esecuzione dell'agente, altrimenti esegue il *behaviour* successivo se presente.

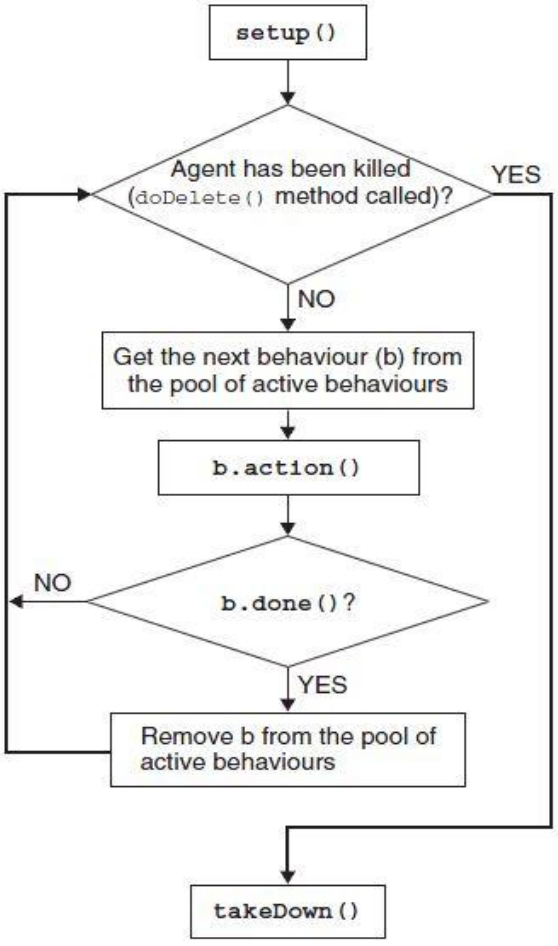

figura 4 – ciclo di vita di un agente JADE.

2.1.4.3.1 Behaviour

Per permettere l'esecuzione di uno o più compiti agli agenti, JADE fornisce l'astrazione *behaviour*, che si riferisce al comportamento dell'agente. Un behaviour è implementato concretamente dalla classe *jade.core.behaviours.Behaviour* e corrisponde ad un particolare compito che un agente deve svolgere durante il suo ciclo di vita. Un behaviour definisce due metodi fondamentali: il metodo *action()*, che contiene il codice relativo alle operazioni che l'agente deve compiere, ed il metodo *done()* che indica al sistema se l'agente ha terminato il behaviour o meno.

Le API di JADE forniscono nativamente, oltre alla classe *Behaviour*, anche numerose altre classi che la estendono e che permettono di creare logiche di comportamento complesse. I principali behaviour sono i seguenti:

OneShotBehaviour: è un behaviour che viene eseguito una volta soltanto. Il sistema garantisce quindi che il metodo *action()* sia invocato una volta sola e che il metodo *done()* restituisca sempre il valore *true* dopo l'unica esecuzione di *action()*.

CyclicBehaviour: è un behaviour che viene eseguito ciclicamente. Il rispettivo metodo *done()* restituisce sempre il valore *false*, causando l'esecuzione continua del metodo *action()*. Il behaviour si conclude soltanto alla terminazione dell'agente mediante la chiamata al metodo *doDelete()*.

SequentialBehaviour: è un behaviour che permette di aggregare più behaviour tra loro ed eseguirli in sequenza secondo l'ordine indicato dal programmatore.

ParallelBehaviour: è un behaviour che permette di aggregare più behaviour tra loro ed eseguirli in parallelo.

TickerBehaviour: è un behaviour che permette di eseguire un'azione allo scadere di un intervallo temporale prefissato.

FSMBehaviour: è un behaviour che permette la creazione di automi a stati finiti in cui ogni stato corrisponde ad un behaviour. L'esecuzione dei behaviour inizia sempre da quello che costituisce lo stato iniziale dell'automa e termina quando finisce il metodo *action()* di un behaviour considerato come uno stato finale.

2.1.4.4 Mobilità agenti

JADE fornisce un supporto alla mobilità degli agenti di tipo *not-so-weak* implementato tramite il meccanismo della serializzazione Java. Un agente è quindi in grado di migrare soltanto il proprio stato serializzabile.

La migrazione di un agente può avvenire sia tra container che appartengono alla stessa piattaforma JADE sia tra container di piattaforme differenti. Nel primo caso la migrazione avviene nei seguenti quattro passaggi:

1. Durante l'esecuzione di un behaviour, l'agente esegue il metodo *doMove()* che scatena il processo di migrazione verso il contenitore avente identificativo uguale a quello passato al metodo.
2. L'agente, una volta scelto il container di destinazione, procede con la copia del proprio codice e del proprio heap; subito dopo, il runtime JADE crea una nuova istanza dello stesso agente all'interno del container di destinazione.
3. Nel caso la copia effettuata dall'agente sia andata a buon fine, un attimo prima dell'attivazione di una nuova istanza di se stesso nel contenitore di destinazione, l'agente esegue il metodo *beforeMove()*: in tale metodo devono essere inserite le azioni che l'agente deve compiere prima della terminazione del suo thread, come ad esempio il rilascio delle risorse utilizzate.
4. A questo punto, l'AMS provvede a terminare definitivamente il thread dell'agente nel contenitore di partenza e ad avviarne uno nuovo in quello di destinazione. Prima di effettuare quest'ultima azione, viene eseguito il metodo *afterMove()* che deve contenere le azioni che l'agente deve compiere alla riattivazione.

Nel caso i due contenitori non appartengano alla stessa piattaforma, la procedura rimane la stessa, tranne che per due dettagli:

1. Al metodo *doMove()* non viene più passato come parametro l'identificativo del contenitore di destinazione, ma quello della piattaforma di arrivo, che corrisponde al suo contenitore principale. Come conseguenza di ciò, l'agente viene in realtà trasferito prima al contenitore principale della piattaforma di arrivo, e da qui successivamente al contenitore di destinazione vero e proprio: l'intera operazione è coordinata dall'MTS.
2. L'heap e il codice dell'agente non vengono trasportati tramite serializzazione, ma all'interno di opportuni messaggi ACL.

2.1.4.5 Sicurezza: JADE Security

JADE Security (JADE-S) è un add-on JADE che ha lo scopo di rendere sicura la piattaforma ad agenti[JADE-S]. JADE-S offre meccanismi per l'autenticazione degli utenti che usano la piattaforma, per esplicitare i permessi sulle azioni che un utente può compiere e per garantire l'integrità e la confidenzialità dello scambio di messaggi.

2.1.4.5.1 Autenticazione

Per quanto riguarda l'autenticazione, JADE-S si basa su *Java Authentication and Authorization Service* (JAAS)[JAAS]. JAAS mette a disposizione diversi metodi per riconoscere un utente: JADE ha implementato le modalità *Unix*, *NT* e *Kerberos*. Le modalità *Unix* ed *NT* sono progettate in modo da estrarre l'identità dell'utente dalla sessione corrente di utilizzo del sistema operativo. La modalità *Kerberos* richiede che nel sistema ad agenti vi sia l'omonimo server di autenticazione in quanto JADE-S interagisce direttamente con quest'ultimo. Esiste anche un'altra modalità di autenticazione chiamata *Simple* che utilizza un file non cifrato contenente gli identificativi degli utenti e le loro password. Questa modalità è consigliata soltanto per scopi di test in quanto è la meno sicura.

2.1.4.5.2 Gestione dei permessi

Grazie al meccanismo di autenticazione, la piattaforma JADE può essere considerata multiutente. Tutte le azioni che gli utenti desiderano compiere sul sistema possono essere consentite o negate in base alla loro identità. La *figura 5* mostra come sono strutturate le policy di sicurezza.

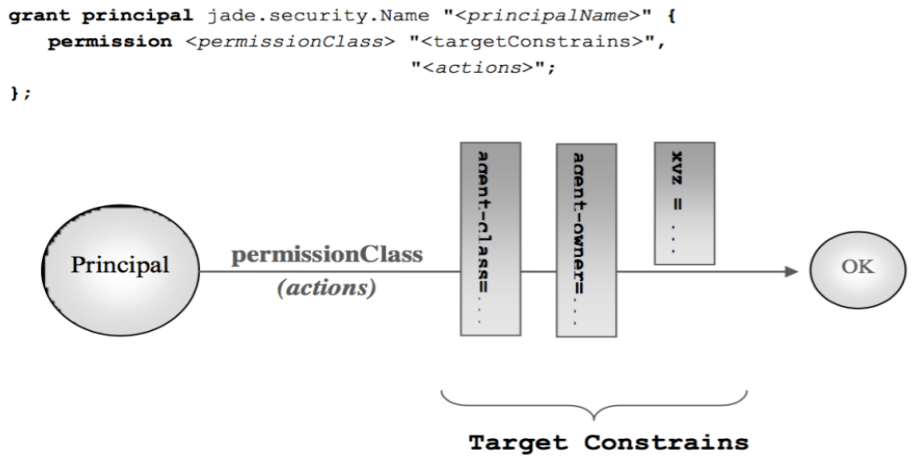

figura 5 – policy di sicurezza per un utente del sistema JADE.

Una policy prevede che ad un *principal*, ovvero ad un utente del sistema, siano concesse delle *permission:* l'attributo *permissionClass* identifica il tipo di permesso concesso, mentre gli attributi *targetConstraint* ed *actions* servono a limitare una permission ad un determinato scopo. Ad esempio la permission:

```
permission jade.security.ContainerPermission "container-owner=bob", "create, kill"
```

specifica che l'utente "*Bob"* può creare e distruggere (*create, kill* actions) uno o più container JADE periferici di cui ha l'ownership.

2.1.4.5.3 Integrità e confidenzialità dei messaggi

JADE-S permette anche di rendere sicuro lo scambio di messaggi tra due agenti mediante firma digitale e cifratura. I principali servizi offerti dalla piattaforma per questo scopo sono:

jade.core.security.SecurityService: ha il compito di gestire la generazione delle chiavi per la firma e/o cifratura dei messaggi.

jade.core.security.signature.SignatureService: gestisce il meccanismo di firma digitale.

jade.core.security.encryption.EncryptionService: gestisce la cifratura dei messaggi.

2.1.4.5.4 Limitazioni

JADE-S presenta alcune limitazioni: permessi sulla mobilità degli agenti non sono supportati e quindi non è garantito il trasferimento sicuro di un agente da un container ad un altro. Inoltre i canali SSL-TLS utilizzati dalla piattaforma per scambiare messaggi di controllo non sono firmati quindi è possibile che un agente malevolo o un hacker riesca a mandare messaggi non autorizzati ad un container JADE ed a modificarne il comportamento.

2.2 Programmazione logica

2.2.1 Le basi

In informatica la *programmazione logica* è un paradigma di programmazione che adotta la logica del primo ordine per rappresentare ed elaborare l'informazione. La programmazione logica differisce dalla programmazione tradizionale in quanto consente al programmatore di descrivere la struttura logica del problema piuttosto che il modo di risolverlo. Adotta quindi uno stile di programmazione dichiarativo e non imperativo.

Un programma logico è composto da una serie di fatti, che descrivono situazioni sempre vere, ed una serie di regole, che permettono di dedurre nuove situazioni vere sulla base dei fatti a disposizione. Tipicamente un programma logico è eseguito da un motore inferenziale che, data una determinata base di conoscenza, è in grado di asserire se un goal richiesto dall'utente è vero oppure è falso.

Il più noto linguaggio per la programmazione logica è il Prolog il quale si basa sulle clausole di Horn. Ad esempio la riga di codice:

h:- $b_1, b_2, \ldots, b_n$.

è una clausola Prolog di cui h è la testa e $b_1, b_2, \ldots, b_n$ sono i letterali che ne compongono il corpo. h è vera se è solo se il corpo è vero, ovvero se sono veri tutti i letterali nel corpo.

Un programma Prolog è costituito da un insieme di termini (letterali) ognuno dei quali è costituito da un insieme di caratteri. Un oggetto può essere semplice, come una costante o una variabile, oppure composto, ossia formato da altri oggetti detti strutture (*figura 6*).

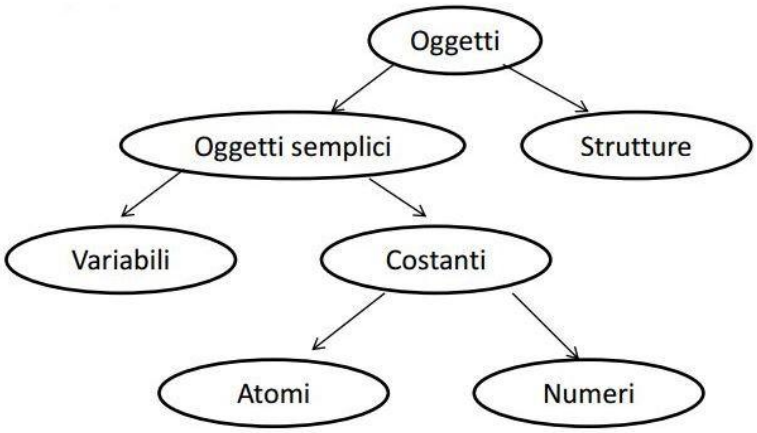

figura 6 – tipologie di oggetti (termini) Prolog.

Il motore Prolog possiede un meccanismo di inferenza che tenta di rispondere alle domande ponendole in relazione con i fatti e le regole della base di conoscenza e tentando di eseguire delle deduzioni.

Il meccanismo di inferenza si basa sul concetto di unificazione: dati due termini, essi unificano se vale una delle seguenti condizioni:

- Sono identici.
- Le variabili in essi possono essere istanziate con oggetti in modo che, dopo la sostituzione delle variabili, i termini diventano identici.

L'unificazione è un processo che considera due termini e verifica se unificano. Se non unificano, il processo fallisce, se unificano, il processo di unificazione ha successo e istanzia le variabili nel modo piú generale (*most general unification*), cioè meno vincolante.

È possibile anche dare una definizione operativa (ricorsiva) di unificazione:

- Se S e T sono costanti, S e T unificano solo se sono lo stesso oggetto.
- Se S è una variabile e T è qualsiasi cosa, allora S e T unificano e S viene istanziata a T (se sia S che T sono variabili, una viene rinominata con l'altra).
- Se S e T sono strutture, allora S e T unificano solo se: S e T hanno lo stesso funtore principale e tutte le componenti corrispondenti unificano. L'istanziazione risultante è determinata dall'unificazione delle componenti.

2.2.2 tuProlog

tuProlog[2PL], sviluppato presso il dipartimento DISI dell'Università di Bologna[DISI], è un motore inferenziale Prolog leggero per infrastrutture e applicazioni distribuite composto da un core minimale, configurabile sia staticamente sia dinamicamente tramite il caricamento di librerie di predicati. tuProlog supporta nativamente la programmazione multi-paradigma, realizzando un'integrazione tra Prolog ed i più comuni linguaggi di programmazione object-oriented come Java e C#[2PMP][2PU].

tuProlog è stato progettato per essere uno dei componenti base delle infrastrutture e applicazioni internet: questo scopo ha determinato le sue principali caratteristiche come la semplicità di deploy, la leggerezza, la configurabilità dinamica e l'interoperabilità con Java[2P].

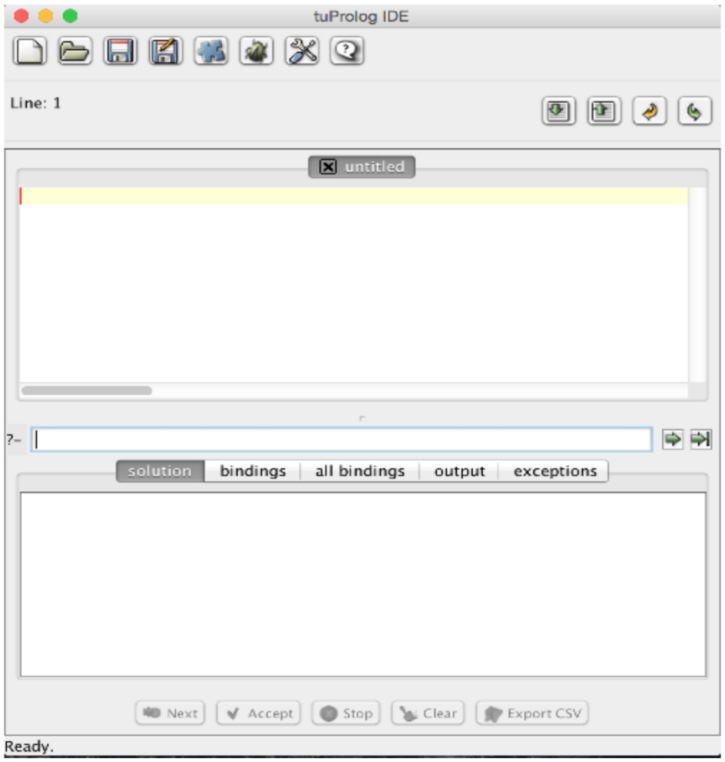

figura 7 - tuProlog IDE.

tuProlog è scaricabile al seguente link:

https://bitbucket.org/tuprologteam/tuprolog/downloads

oppure è disponibile su Maven:

https://mvnrepository.com/artifact/it.unibo.alice.tuprolog/tuprolog/3.2.1

Il motore è utilizzabile come programma stand-alone (*figura 7*), oppure è possibile integrare la sua libreria all'interno di progetti Java ed usarla senza l'ausilio di un'interfaccia grafica.

L'integrazione con Java, la leggerezza e la possibilità di configurare dinamicamente il motore sono le principali motivazioni che hanno portato a scegliere tuProlog come motore inferenziale per il supporto alla programmazione logica nel codice sviluppato in questa tesi.

2.2.3 Logic Programming as a Service (LPaaS)

Attualmente, dato l'aumento del numero dei dispositivi connessi in rete, la computazione distribuita si sta evolvendo nella direzione di offrire servizi *anytime, anywhere*, ovvero disponibili sempre e ovunque. Tali servizi arricchiscono la user-experience degli applicativi distribuiti, ma richiedono anche coordinazione ed intelligenza.

La programmazione logica costituisce un buon candidato per facilitare l'integrazione di sistemi distribuiti e per arricchirli di intelligenza in quanto possiede le seguenti caratteristiche:

- esprime la computazione come deduzione logica;
- consente il non determinismo nella computazione, ovvero maggiore flessibilità nella computazione distribuita;
- permette di ragionare su ciò che è vero e non su quale azione bisogna compiere;
- permette di computare anche con informazioni parziali;
- consente di usare l'unificazione come un meccanismo non direzionato per lo scambio di messaggi tra entità multiple.

Le tecniche di programmazione logica tradizionale, tuttavia, sono in contrasto con gli scenari tipici delle applicazioni distribuite (ad agenti e non), dominate da modelli basati sulla mobilità degli elementi, distribuzione dei nodi, funzionalità e risorse offerte *as a service*. Per questi motivi nasce la necessità di declinare la programmazione logica classica nella prospettiva di uno scenario distribuito e orientato ai servizi[LPaaS].

LPaaS rappresenta proprio questa evoluzione, e si propone di usare i principi dei sistemi service-oriented per distribuire intelligenza situata al fine di abilitare scenari di intelligenza collettiva che partono dalla conoscenza locale di ogni nodo.

LPaaS fornisce una visione astratta del motore inferenziale e lo espone come servizio. Il motore Prolog è acceduto da remoto attraverso due interfacce standardizzate: *client* e *configurator* (paragrafo 4.1.1.1). La distinzione permette di abilitare un cliente a fare query Prolog che modificano o meno la base di conoscenza del servizio: ad esempio un utente che ha accesso all'interfaccia *configurator* può aggiungere teorie al motore Prolog mente un utente che ha i permessi per accedere all'interfaccia *client* può soltanto interrogare il servizio, cioè può richiedere la dimostrazione di uno dei goal che il servizio espone.

La *figura 8* mostra un'interazione tra un client e diversi servizi LPaaS: la stessa query Prolog *q* può fornire risultati differenti in quanto ciascun servizio LPaaS è sensibile ai cambiamenti dell'ambiente che lo circonda, ovvero possiede una base di conoscenza situata e dipendente dal contesto di esecuzione.

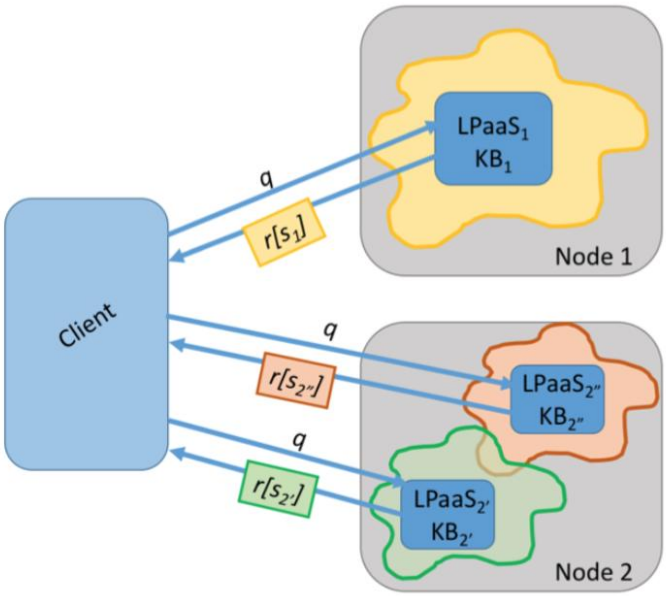

figura 8 – interazioni tra un cliente e diversi servizi LPaaS.

3. Obiettivi ed analisi dei requisiti

La tesi ha come principale obiettivo quello di studiare una possibile integrazione tra la programmazione ad agenti e la programmazione logica per realizzare un prototipo di supporto software che faciliti lo sviluppo di applicativi Java IoT distribuiti ed *intelligenti*. Con il termine *intelligenti* si intende una categoria di applicativi autonomi nelle decisioni, reattivi e consapevoli dell'ambiente in cui eseguono.

Per la realizzazione di applicativi IoT dotati di intelligenza situata si avverte la necessità di individuare un modello di programmazione flessibile (paragrafo 3.1), che coniughi le proprietà di un agente (paragrafo 2.1.1) e le potenzialità offerte dalla programmazione logica (paragrafo 2.2). Nei paragrafi successivi si tratterà anche della necessità di utilizzare un approccio multi-paradigma per arricchire di intelligenza un sistema IoT (paragrafo 3.2), della possibilità di offrire un servizio LPaaS attraverso un agente (paragrafo 3.3) e della necessità di poter configurare dichiarativamente il sistema ad agenti in modo da poter facilmente modificare il deployment di un applicativo IoT (paragrafo 3.4).

3.1 Modello di programmazione flessibile

L'utilizzo della programmazione logica in un sistema distribuito ad agenti richiede che almeno su un nodo del sistema vi sia un motore inferenziale che interpreti le richieste di clienti remoti (ed eventualmente anche locali).

Al fine di semplificare lo sviluppo di applicativi intelligenti e distribuiti, si ha la necessità di individuare un modello di programmazione che permetta agli sviluppatori di configurare dichiarativamente il motore inferenziale e di affiancargli metodi che contengono logica applicativa. Inoltre si desidera poter scrivere codice facilmente riutilizzabile e resistente ai cambiamenti: lo scopo è incapsulare il motore Prolog in un oggetto di più alto livello, *general purpose*, indipendente dagli altri componenti del sistema (principi di *encapsulation* ed *information hiding)*, con una struttura di base definita e facilmente personalizzabile da chi progetta il sistema. L'oggetto che contiene il motore deve fornire ad una entità esterna informazioni su come gestire le proprie istanze, su come configurare il motore inferenziale e su quali metodi invocare automaticamente.

Da queste necessità si deduce che un modello di programmazione a *componente-container* è adatto per semplificare la configurazione e l'utilizzo di un motore inferenziale in un sistema distribuito. Nel modello *componente-container* il componente è rappresentato dall'oggetto che incapsula l'engine inferenziale e la relativa logica di business mentre il container è un middleware di livello applicativo

che gestisce il numero delle istanze, la configurazione ed il ciclo di vita dei componenti.

Il modello presenta diversi vantaggi in quanto:

- disaccoppia la logica applicativa dalla logica di gestione;
- uniforma il codice imponendo un formato base standard ai componenti;
- rende flessibile gli applicativi che lo adottano in quanto un componente può essere facilmente spostato da un container all'altro in fase di deployment;
- è estremamente *general purpose* in quanto il container può essere integrato in qualsiasi tipo sistema, distribuito e non.

Siccome si desidera utilizzare il modello per la progettazione di sistemi ad agenti in ambito IoT, si deve anche tenere conto del requisito di leggerezza che caratterizza questi sistemi: il container deve avere un footprint minimale e non deve appesantire l'esecuzione degli applicativi.

3.2 Programmazione multi-paradigma

L'utilizzo della programmazione ad agenti assieme alla programmazione logica per lo sviluppo di applicativi IoT implica coniugare paradigmi di programmazione differenti.

JADE e tuProlog utilizzano la programmazione ad oggetti per fornire astrazioni di alto livello alle applicazioni: nel primo caso si ha la possibilità di utilizzare l'astrazione di agente, nel secondo si ha a disposizione un interprete Prolog. Anche il modello di programmazione proposto al paragrafo precedente si basa sui principi della programmazione ad oggetti per incapsulare comportamento, nascondere stato applicativo e fornire un'astrazione di alto livello.

Il punto di incontro tra programmazione ad agenti e programmazione imperativa ad oggetti sta nel fatto che la prima è realizzabile utilizzando le proprietà della seconda: un agente JADE è anche un oggetto Java standard. Ciò che differenzia un agente da un oggetto è la semantica associata a ciascuna astrazione: un agente è pensato per essere autonomo, reattivo, non invocabile in modo procedurale e situato nel tempo e nello spazio. Un oggetto Java, contrariamente, è pensato per interazioni passive mediante invocazione di metodo, non è autonomo né reattivo e non possiede la proprietà di situatedness. Inoltre gli agenti sono pensati per interazioni lasche basate su scambio di messaggi non bloccante, mentre le interazioni tra gli oggetti prevedono trasferimento di controllo bloccante. A prima vista può quindi sembrare un controsenso dire che un agente JADE è un oggetto Java: ciò e vero se si considerano

le semantiche di utilizzo delle due astrazioni, ma se si considera l'implementazione del sistema, si nota come le proprietà tipiche di un agente siano garantite dal runtime in cui esegue attraverso l'utilizzo della programmazione ad oggetti. Nel caso di JADE la programmazione ad oggetti ha la funzione di *enabler* per la realizzazione degli agenti.

tuProlog si basa sui principi della programmazione ad oggetti per realizzare una macchina virtuale Prolog. Il motore costituisce un esempio particolare di unione tra programmazione imperativa e programmazione logica: esso non si limita ad utilizzare la prima per implementare le astrazioni della seconda, ma consente un approccio ibrido tra le due, permettendo di scrivere codice Prolog che invoca codice Java e viceversa[2PMP]. Ciò consente di arricchire il codice Java con proprietà tipiche della programmazione logica quali la dichiarativité ed il non determinismo. In tuProlog è possibile scrivere teorie logiche che durante la loro dimostrazione prevedono l'invocazione di codice Java: in questo modo, in caso di backtracking, è possibile esplorare soluzioni alternative ed invocare in modo non deterministico codice Java. Va detto però che questo approccio si allontana dai principi della programmazione logica classica, i quali prevedono che la computazione sia stateless e che esecuzioni multiple di una stessa query Prolog siano tra loro idempotenti se non si modifica la base di conoscenza del motore inferenziale. L'approccio multi-paradigma di tuProlog prevede che la dimostrazione di una query possa alterare l'ambiente di esecuzione e quindi non essere idempotente: questa caratteristica risulta particolarmente utile all'interno di sistemi distribuiti in quanto permette di sfruttare la programmazione logica per agire sull'ambiente in cui un nodo esegue e sul suo comportamento.

Ciò che si desidera sperimentare è l'unione del paradigma ad agenti con il paradigma logico per poter arricchire le capacità espressive e di conseguenza dotare di intelligenza sistemi distribuiti IoT. La programmazione multi-paradigma permette di utilizzare l'astrazione più opportuna in base alle necessità applicative: non si vuole una netta separazione tra oggetti, agenti e programmazione logica, ma si desidera impiegare un modello ibrido in cui i diversi paradigmi sono in grado di interoperare all'interno della stessa applicazione.

3.3 Agenti e servizi LPaaS

Una caratteristica fondamentale degli agenti è di poter cooperare tra loro per la risoluzione di goal comuni. Si desidera sfruttare la socialità degli agenti per fornire servizi LPaaS all'interno di un sistema multi-agente.

L'idea è di fornire un servizio LPaaS attraverso un agente dedicato e non attraverso l'utilizzo di Web Services RESTful[WS]. Quest'ultimi utilizzano lo schema architetturale *Representational State Transfer* (REST) che prevede che si acceda a una risorsa attraverso richieste HTTP[REST]. L'approccio REST modella interazioni client-server *stateless* e facilmente *cachable* poiché l'invocazione remota di metodi avviene attraverso lo scambio di messaggi, tipicamente sopra il protocollo HTTP. In un Web services RESTful l'engine Prolog LPaaS espone le interfacce *client* e *configurator* a specifici URL che identificano il servizio; i clienti mandano normali richieste HTTP agli URL delle interfacce ed ottengono, se autorizzati, il risultato dell'esecuzione del servizio.

L'approccio REST presenta i seguenti pregi: favorisce il disaccoppiamento di cliente e servitore, prevede la standardizzazione di messaggi per l'invocazione di servizi ed è *implementation agnostic*. Per quanto riguarda i difetti, REST su HTTP risulta verboso ed inefficiente nello scambio di messaggi, inoltre impone che ciascun dispositivo LPaaS supporti un HTTP server. Infine in REST le interazioni sono sempre client-server e non peer-to-peer: i ruoli di cliente e servitore sono fissi nel tempo e non possono essere adattati dinamicamente a runtime in base alle esigenze applicative.

Ciò che si desidera sperimentare è offrire un servizio LPaaS a livello applicativo, attraverso un agente dedicato e non secondo i principi REST.

Si hanno i seguenti requisiti:

- l'agente deve gestire uno o più motori Prolog che offrono uno o più servizi LPaaS;
- l'agente deve registrarsi all'agente DF del sistema JADE per consentire la discovery dei servizi LPaaS;
- l'agente si deve occupare di controllare se il cliente che invoca il servizio ha i permessi necessari per accedervi.

Questo tipo di approccio nello sviluppo di servizi LPaaS presenta i seguenti vantaggi: si basa su invocazione di metodi attraverso scambio di messaggi standardizzati, non impone una rigida divisione tra i ruoli cliente e servitore, favorisce la mobilità del servizio e la sua situatedness. Per quanto riguarda il formato dei messaggi, lo standard utilizzato è FIPA ACL che ha come scopo quello di favorire la comunicazione tra agenti (e qualsiasi tipologia di software conforme allo standard).

Generalmente le interazioni tra agenti sono peer-to-peer: non si ha una netta divisione in ruoli ma un agente può comportarsi sia come cliente che come servitore verso gli agenti con cui interagisce. Questa modalità di interazione permette ad un agente LPaaS di offrire un servizio ma anche di essere cliente verso altri servizi. Un altro pregio nell'utilizzo degli agenti come mezzo per realizzare servizi LPaaS è la possibilità di sfruttare la propria mobilità per rilocare un servizio in caso di cambiamenti nell'ambiente di esecuzione o di fault a runtime. Infine un agente è per sua natura situato nello spazio e nel tempo e di conseguenza è adatto ad offrire un servizio di LPaaS.

Rispetto a REST, l'approccio ad agenti presenta come vantaggi quello di non vincolare i ruoli tra le entità e di favorire la mobilità e la situatedness dei servizi. Lo svantaggio principale risiede nella limitata adozione delle piattaforme ad agenti nello sviluppo di applicativi distribuiti.

3.4 Configurazioni dichiarative e programmazione logica

Una necessità che si avverte nello sviluppo di applicazioni IoT è di poter configurare dichiarativamente il deployment del software che deve eseguire su un nodo. Tale necessità deriva dal fatto che i sistemi IoT si caratterizzano per una elevata eterogeneità dei nodi. Poter configurare dichiarativamente il software non solo ne semplifica il deployment, ma rende flessibile il sistema e permette di configurarlo per sfruttare al meglio le risorse del singolo nodo su cui esegue.

A tal proposito la programmazione logica costituisce un buon candidato per risolvere il problema in quanto consente di programmare dichiarativamente teorie logiche che rappresentano la configurazione del sistema. In questo modo la configurazione di un nodo può avvenire mediante la dimostrazione di un goal che si occupa di consultare una teoria che contiene tutti i dati di configurazione. Questo approccio consente di impiegare il non determinismo, tipico della programmazione logica, nelle configurazioni: in caso di fallimento di una configurazione, è possibile esplorarne un'altra e tentare comunque di avviare il sistema.

Questo approccio è applicabile anche per la configurazione di sistemi distribuiti multi-agente: ciò che si desidera sviluppare è un supporto software leggero e minimale che permetta di dichiarare teorie logiche Prolog per la configurazione degli agenti che eseguono su un nodo. A tale scopo è utile l'approccio multi-paradigma offerto dal motore inferenziale tuProlog, il quale permette di fare interoperare codice Prolog e codice Java: è possibile gestire la configurazione degli agenti dal Prolog e contemporaneamente invocare il codice Java che avvia la piattaforma JADE.

Inoltre si desidera utilizzare il sistema di configurazione anche per gestire la migrazione di un agente verso un altro nodo computazionale: in questo modo si ha a disposizione la possibilità di programmare dichiarativamente teorie logiche che descrivono le condizioni che devono verificarsi affinché un agente possa spostarsi da un nodo del sistema ad un altro.

Per la realizzazione del sistema di configurazione si hanno quindi i seguenti requisiti:

- si deve disporre di un motore inferenziale tuProlog che si occupi soltanto della configurazione del sistema e della migrazione degli agenti;
- il motore inferenziale deve interfacciarsi con la piattaforma ad agenti JADE per consentirne la configurazione;
- il motore inferenziale deve essere in grado di scatenare la migrazione di un agente verso un altro container della piattaforma;
- la base di conoscenza del motore deve rispecchiare in ogni istante la configurazione degli agenti in esecuzione sul nodo;
- il supporto alla configurazione deve essere minimale e leggero.

4. Progettazione

La progettazione del software è avvenuta in diverse fasi ed ha affrontato le seguenti problematiche:

- come progettare il modello a componente-container per semplificare la configurazione e l'utilizzo di un motore inferenziale in un sistema distribuito (paragrafo 4.1);
- come integrare la programmazione logica all'interno del sistema ad agenti JADE e fornire supporto ad LPaaS (paragrafo 4.2);
- come realizzare la configurazione del sistema ad agenti in modo dichiarativo per poter modificare semplicemente il deployment di un applicativo IoT distribuito (paragrafo 4.3).

4.1 Modello componente-container

4.1.1 Componente applicativo

Si è scelto di modellare un componente applicativo come una classe Java composta da tre elementi principali (*figura 9*):

- ***Motore Inferenziale (tuProlog):*** consente l'utilizzo della programmazione logica all'interno della logica di business del componente.
- ***Metadati:*** danno informazioni al container su come deve gestire un componente. Si dividono in:
    - ***Metadati per la configurazione del motore inferenziale***: permettono di specificare quali teorie logiche caricare nel motore inferenziale, quali direttive eseguire in fase di configurazione, quali flag abilitare e come configurare il sistema di management del motore (paragrafo 6.3.4).
    - ***Metadati sulla gestione del numero di istanze dei componenti***: permettono di specificare al container se un componente applicativo deve essere considerato come *singleton* o meno.
    - ***Metadati per l'invocazione trasparente di metodi***: permettono al container di automatizzare l'invocazione di metodi di un componente quando quest'ultimo viene istanziato, deve migrare o sta per essere distrutto.
- ***Logica di Business:*** è a carico del programmatore ed è il codice del componente che utilizza il motore inferenziale.

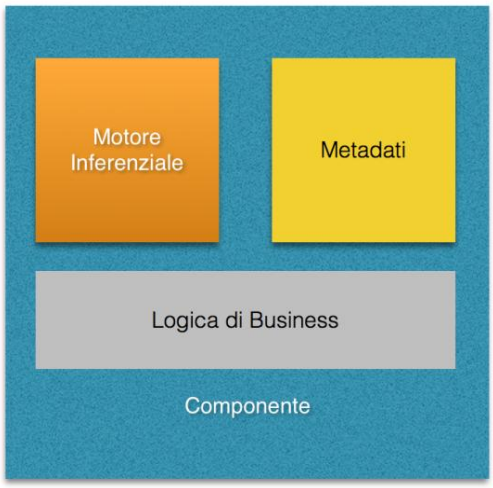

figura 9 - struttura di un componente applicativo.

Per rendere flessibile lo sviluppo di componenti custom, si è deciso di permettere la dichiarazione e l'utilizzo di altre risorse oltre al motore inferenziale; la gestione di tali risorse è a carico dello sviluppatore.

4.1.1.1 Componente LPaaS

Un componente LPaaS possiede le stesse caratteristiche dei componenti applicativi, ma in più offre la possibilità di specificare quali goal sono disponibili via LPaaS. Un componente di questa tipologia rappresenta il core di un servizio LPaaS ed implementa le interfacce *client* e *configurator*. Un utente *configurator* può modificare la base di conoscenza del motore e cambiare la lista di goal esposti dal componente; un utente *client* può soltanto interrogare il componente.

La *tabella 1* riporta i metodi delle interfacce *client* e *configurator*.

Client interface	
Metodo	**Descrizione**
String solve(String goal)	Restituisce la soluzione del goal passato come argomento.
String solve(String goal, long time)	Restituisce la soluzione del goal passato come argomento in un intervallo temporale di al più *time* millisecondi.
String solveNext()	Restituisce la soluzione successiva (se presente) dell'ultimo goal richiesto.

String solveNext(long time)	Restituisce la soluzione successiva (se presente) dell'ultimo goal richiesto in al più *time* millisecondi.
String solveN(String goal, int num)	Restituisce al più le prime *num* soluzioni del goal passato come argomento.
String solveN(String goal, int num, long time)	Restituisce al più le prime *num* soluzioni del goal passato come argomento. L'operazione deve essere completata in al più *num*time* millisecondi.
String solveAll(String goal)	Restituisce tutte le soluzioni del goal passato come argomento.
String solveAll(String goal, long time)	Restituisce tutte le soluzioni del goal passato come argomento. L'operazione deve essere completata in al più *numSoluzioni*time* millisecondi.
String isGoal(String goal)	Controlla se la soluzione di un goal può essere richiesta o meno.
Configurator interface	
Metodo	**Descrizione**
String addGoal(String goal)	Aggiunge il goal passato come argomento alla lista di goal ammessi.
String getGoalList()	Ottiene la lista dei goal esposti dal componente.
String removeGoal(String goal)	Rimuove il goal passato come argomento dalla lista dei goal esposti dal componente.
String addTheory(String theory)	Aggiunge la teoria passata come argomento alla base di conoscenza del motore inferenziale.

tabella 1 – LPaaS client e configurator interface.

4.1.2 Il container

Il *container* è il supporto software che gestisce il ciclo di vita dei componenti: si occupa della loro creazione, configurazione e distruzione.

Si sono individuati quattro stati che caratterizzano il ciclo di vita del container (*figura 10*): *Bootstrap*, *Container operativo*, *Migrazione* e *Garbage*. Il primo stato modella la configurazione del container che consiste nella lettura dei metadati dei componenti. Lo stato di *Container operativo* è lo stato principale in cui il container permette al software che lo utilizza di accedere ai componenti. La configurazione ed istanziazione dei componenti può avvenire o nello stato di *Boostrap,* qualora si necessiti di una politica di inizializzazione *eager*, oppure nello stato di *Container operativo*, in modo *lazy*, quando un cliente richiede di utilizzare un componente. La scelta è lasciata alle specifiche implementazioni del container, in base alle ottimizzazioni che si vogliono realizzare (paragrafo 5.2.1).

Lo stato di *Migrazione* è lo stato in cui transita il container quando l'applicazione che lo usa desidera cambiare nodo di esecuzione. Si è deciso che il container si occupi di serializzare/de-serializzare lo stato dei motori inferenziali dei componenti; il trasferimento dello stato è demandato al software che gestisce la migrazione del container (paragrafo 4.2.1). Ogni altra risorsa utilizzata dai componenti deve essere rilasciata prima della migrazione e riacquisita dopo il loro ripristino.

Lo stato di *Garbage* è uno stato transitorio in cui il container si pone quando l'applicazione che lo usa non ne ha più necessità o sta per terminare. In questo stato il container provvede alla finalizzazione dei componenti, ovvero invoca in callback i metodi che il programmatore ha indicato nei metadati.

In caso di qualsiasi tipo di errore, il container si porta in uno stato degenere di *Error* (omesso in *figura 10*).

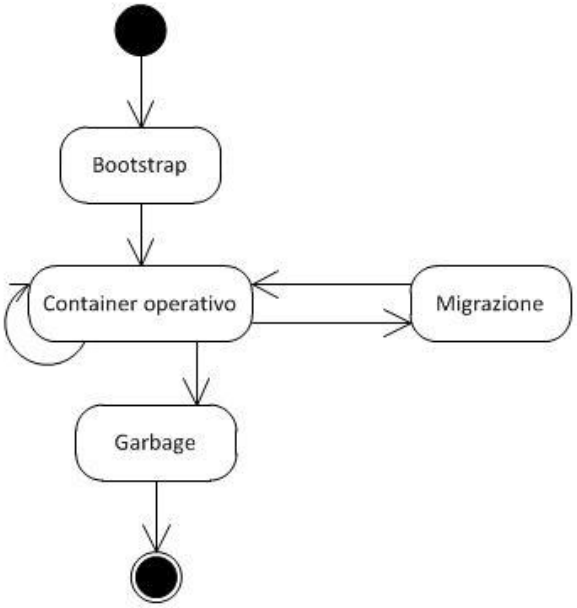

figura 10 – diagramma a stati del ciclo di vita del container.

La *figura 11* mostra le operazioni svolte dal container per la creazione e la configurazione di un componente: il container, dopo aver letto i metadati del componente, decide in base a questi come creare il componente (se in modo *eager* o *lazy*), come configurarne il motore inferenziale ed infine quale metodi eseguire automaticamente.

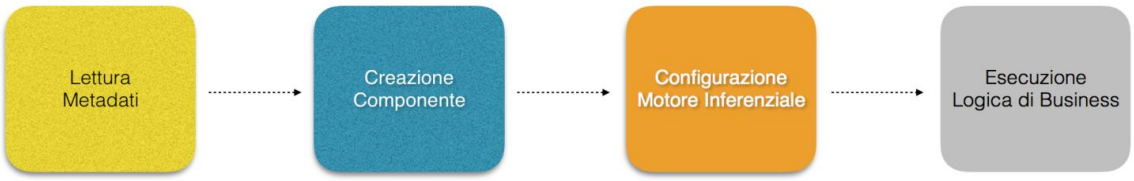

figura 11 - operazioni compiute dal container per la configurazione di un componente.

La *figura 12* mostra l'evoluzione dello stato del container durante il processo di migrazione: si è scelto di dividere lo stato di *Migrazione* in due sotto-stati che modellano rispettivamente le azioni compiute dal container prima di migrare e dopo la propria riattivazione. Lo stato di *Pre-Migration Setup* prevede che, per ogni componente, il container provveda prima a rilasciare le risorse, invocando in callback i metodi che il programmatore ha indicato nei metadati, poi a serializzare lo stato del motore inferenziale. Rispettando questa sequenza di azioni, si garantisce che non vi sia alcuna risorsa che utilizza il motore quando il container lo serializza.

Dopo la migrazione, il container si porta nello stato di *Post-Migration Setup* in cui ripristina i componenti, riattivando i motori inferenziali, ed invoca in callback i metodi che il programmatore ha indicato nei metadati come metodi da invocare dopo la migrazione. In questo modo è possibile riacquisire le risorse necessarie per l'esecuzione dei componenti.

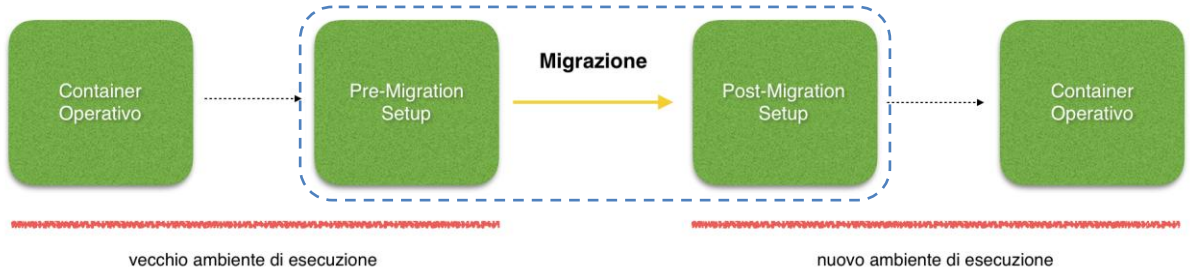

figura 12 - evoluzione dello stato del container durante il processo di migrazione.

Si è scelto di stabilire un'interfaccia di alto livello per descrivere i metodi che una qualsiasi implementazione del container deve fornire (*tabella 2*). Un'interfaccia più dettagliata è presentata al paragrafo 5.2 in cui si approfondisce l'implementazione del container.

Container interface	
Metodo	**Descrizione**
configure	Effettua il bootstrap del container configurando i componenti.
isConfigured	Testa se è avvenuto il bootstrap del container.
getComponent	Restituisce un componente a partire dal suo identificatore.
getComponentMetaData	Restituisce i metadati di un componente a partire dal suo identificatore.
removeComponent	Elimina un componente in base al suo identificatore.
preMigrationSetup	Effettua le operazioni necessarie per effettuare la migrazione del container e dei suoi componenti.
postMigrationSetup	Riconfigura il container ed i suoi componenti dopo la migrazione.
destroy	Elimina tutti i componenti contenuti nel container.

tabella 2 – Container interface.

4.2 Agenti JADE e programmazione logica

4.2.1 Agente per la gestione di componenti applicativi

Per quanto riguarda l'integrazione tra sistema ad agenti JADE ed il supporto alla programmazione logica, si è scelto di progettare un agente astratto che all'interno del proprio ciclo di vita gestisca il ciclo di vita di un container per componenti applicativi. In questo modo la programmazione logica può essere utilizzata indirettamente dall'agente il quale può invocare metodi dei componenti applicativi che utilizzano il motore inferenziale. L'agente non si occupa soltanto di gestire la configurazione e la terminazione del container, ma coordina anche le operazioni necessarie alla migrazione del container nel caso in cui l'agente decida di cambiare nodo di esecuzione.

Si è deciso che il bootstrap del container vada effettuato quando l'agente esegue il metodo *setup()* e che la distruzione del container venga eseguita quando l'agente esegue il metodo *takeDown()*. In questo modo si garantisce che il container sia operativo quando l'agente inizia ad eseguire i propri compiti e che sia distrutto quando l'agente sta per terminare. Per quanto riguarda la gestione della migrazione, nel metodo *beforeMove()* dell'agente si serializza il contenuto del container, mentre nel metodo *afterMove()* si ripristina il container ed il suo contenuto. In questo modo la migrazione del container è allineata alla migrazione dell'agente.

La *tabella 3* riporta le corrispondenze tra i metodi del ciclo di vita dell'agente astratto e le operazioni da compiere sul container.

Metodo agente astratto	**Effetto sul container**
setup()	Creazione e configurazione del container.
beforeMove()	Setup del container per l'operazione di migrazione dei motori inferenziali dei componenti.
afterMove()	Ripristino del container e del suo contenuto.
takeDown()	Distruzione del container e del suo contenuto.

tabella 3 – metodi agente astratto per la gestione del ciclo di vita del container.

4.2.2 Agente LPaaS

Un agente LPaaS si caratterizza per fornire servizi logici ad altri agenti. Per quanto riguarda la progettazione, si è deciso di basarsi sul modello presentato al paragrafo precedente e di specializzarlo per supportare servizi LPaaS.

Un agente LPaaS possiede un container che gestisce soltanto componenti LPaaS (paragrafo 4.1.1.1): quest'ultimi hanno ciascuno il compito di gestire il core di un servizio, mentre l'agente si occupa di invocare il servizio richiesto da un cliente sul componente LPaaS opportuno e di rispondere con l'esito dell'esecuzione. Un agente LPaaS si caratterizza per avere un comportamento ciclico in cui, ad ogni iterazione, riceve una richiesta di servizio da un cliente, la esegue e risponde comunicando l'esito dell'operazione richiesta.

L'agente LPaaS si deve registrare presso l'agente *Directory Facilitator* (DF) della piattaforma JADE fornendo un identificativo logico ed indicando che la tipologia del servizio offerto è LPaaS: in questo modo gli agenti clienti possono effettuare una operazione di discovery ed individuare quali agenti LPaaS sono presenti nel sistema. La registrazione al DF deve essere fatta nel metodo *setup()*, ovvero prima che l'agente sia completamente operativo. Inoltre l'agente deve de-registrarsi dal DF quando esegue il metodo *takeDown()*.

4.2.2.1 Formato messaggi LPaaS

La comunicazione tra gli agenti JADE avviene mediante lo scambio di messaggi FIPA ACL (paragrafo 2.1.4.2).

Il messaggio ACL per la richiesta di un servizio LPaaS deve avere la *performative* FIPA *Request*[ACL], in quanto l'agente che riceve il messaggio deve compiere una azione, cioè invocare il servizio LPaaS sul componente corrispondente. Il messaggio deve contenere sia l'identificativo logico del componente LPaaS sia l'identificativo dell'operazione che si vuole eseguire (una stringa che identifica univocamente un metodo di quelli presentati al paragrafo 4.1.1.1). Inoltre il messaggio di richiesta deve contenere il goal Prolog che si vuole dimostrare ed eventualmente il numero di soluzioni da esplorare ed un'indicazione temporale che specifica il tempo massimo di esecuzione del servizio.

Il messaggio ACL di risposta ha come *performative* FIPA *Inform*[ACL], in quanto notifica al cliente il risultato dell'esecuzione di un servizio. Il messaggio di risposta contiene la/le soluzione/i del goal richiesto oppure un messaggio di errore nel caso in cui non sia stato possibile eseguire il servizio.

4.2.2.2 Interazioni tra agenti clienti ed agenti LPaaS

Un agente cliente che desidera utilizzare un servizio LPaaS deve, come prima azione, effettuare un'operazione di discovery presso l'agente DF; il cliente può richiedere l'elenco di tutti gli agenti che offrono servizi LPaaS, oppure, se ne conosce l'identificativo logico, può richiedere uno specifico servizio. Un esempio di interazione tra agente cliente e agente LPaaS è mostrata in *figura 13*.

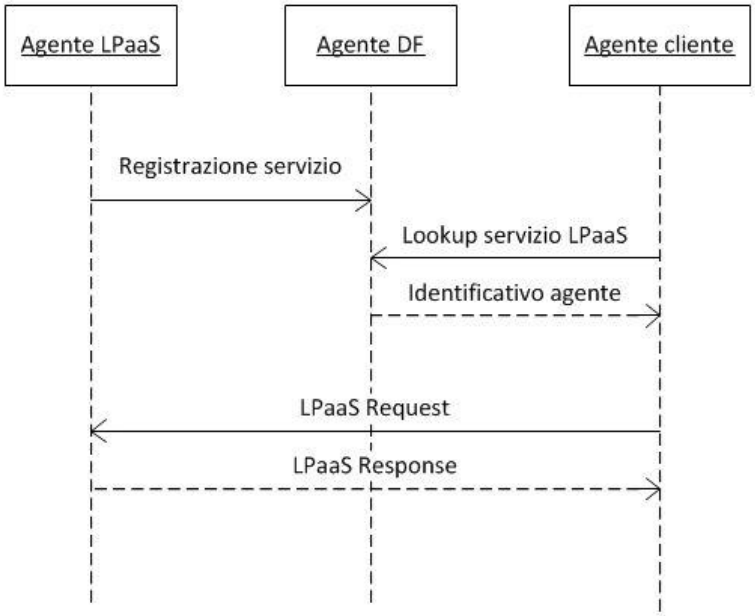

figura 13 – interazione tra agente cliente, agente DF ed agente LPaaS.

L'agente cliente, dopo aver ottenuto l'identificativo (AID) di uno o più agenti LPaaS manda un messaggio ACL di *LPaaS Request* all'agente selezionato. L'agente LPaaS risponde con un messaggio di *LPaaS Response* contenente l'esito dell'invocazione del servizio. L'interazione è sempre *request-response*: anche nel caso di errori, l'agente LPaaS comunica il fallimento dell'esecuzione del servizio al cliente.

La ricezione di un messaggio, sia per l'agente LPaaS sia per il cliente, può avvenire in maniera bloccante o meno; per l'agente LPaaS si preferisce la modalità bloccante in modo da poter sospendere l'esecuzione dell'agente qualora non ci siano richieste di servizio, mentre per il cliente la scelta dipende dalla logica applicativa che si desidera sviluppare.

4.2.2.3 Agente LPaaS e sicurezza

Il behaviour di un agente LPaaS si deve occupare anche di gestire la sicurezza del servizio. Si sono individuati due livelli di sicurezza:

1. autenticità e confidenzialità della comunicazione tra un agente cliente ed un agente LPaaS;
2. identificazione dei permessi che un cliente possiede nei confronti di un servizio LPaaS, ovvero la possibilità di accedere all'interfaccia *configurator* o soltanto all'interfaccia *client* del servizio.

Il primo livello di sicurezza può essere garantito utilizzando JADE-S: l'add-on mette a disposizione meccanismi di firma digitale e cifratura di messaggi. JADE-S permette anche di autenticare un utente che accede alla piattaforma ad agenti e di specificare quali azioni può compiere (paragrafo 2.1.4.5). In questo modo è possibile associare l'identità di un utente ed i relativi permessi ad un agente.

Per quanto riguarda la sicurezza della comunicazione, si è deciso che ogni messaggio LPaaS sia firmato e cifrato. Questa scelta comporta un overhead computazionale ad ogni invio di messaggio, ma garantisce la riservatezza della comunicazione e l'autenticità dei messaggi. Siccome un servizio LPaaS può influire direttamente sul comportamento di un dispositivo fisico, è importante rendere sicuro lo scambio di messaggi per evitare attacchi da parte di agenti malintenzionati.

Il secondo livello di sicurezza richiede che l'agente LPaaS sia in grado di distinguere quali agenti hanno il ruolo di *configurator* e quali no. Gli agenti che hanno il ruolo di *configurator* possono modificare la base di conoscenza dei componenti LPaaS e modificare l'elenco dei goal esposti dal componente. Un agente LPaaS può invocare i metodi dell'interfaccia *configurator* dei componenti LPaaS soltanto se il cliente che ha richiesto tale operazione ha i diritti per farlo. In questo modo si evita che il servizio LPaaS possa essere modificato arbitrariamente da qualsiasi agente.

Infine va ricordato che JADE-S non garantisce sicurezza al servizio di mobilità degli agenti: durante la migrazione un agente LPaaS è quindi vulnerabile ad attacchi di entità malintenzionate quindi si consiglia di disattivare il servizio di mobilità degli agenti quando si reputa probabile un attacco malevolo al sistema.

4.3 Configurazione agenti JADE e programmazione logica

Come esposto al paragrafo 3.4, si desidera sviluppare un supporto software per poter configurare dichiarativamente il sistema ad agenti. Tale supporto si deve basare sulla programmazione logica e deve consentire di progettare teorie logiche che rappresentino la configurazione del sistema.

Per la progettazione del sistema di configurazione, si è scelto di utilizzare il modello *componente-container* in quanto permette di sviluppare una soluzione flessibile e disaccoppiata. La soluzione proposta (*figura 14*) prevede che vi sia una classe di supporto che possiede un container il quale gestisce un componente che si occupa di configurare il sistema ad agenti. Il componente, denominato *Agent Configurator*, dichiara un motore tuProlog che si occupa di gestire la configurazione e l'attivazione del sistema ad agenti mediante la programmazione logica.

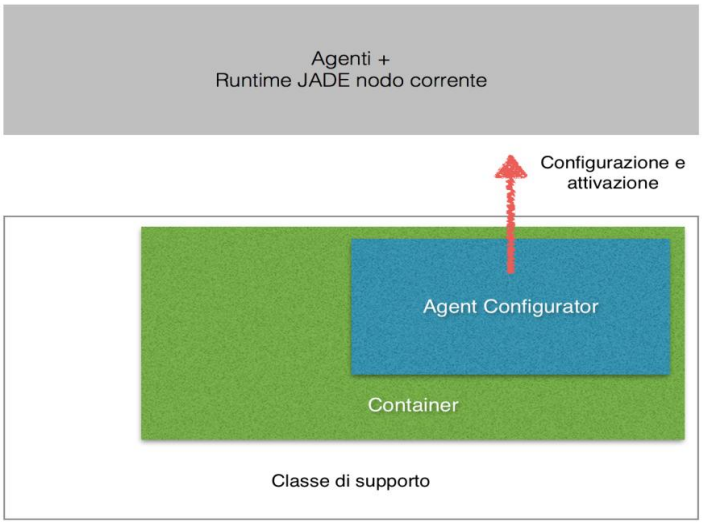

figura 14 – struttura del sistema di configurazione.

Per non vincolare la classe di supporto ad una specifica implementazione del componente *Agent Configurator*, si è deciso di progettare un'interfaccia standard che definisce due metodi:

- ***void configureAgents():*** chiede al motore inferenziale di risolvere il goal di configurazione del sistema ad agenti;
- ***void solveGoal(String goal):*** consente di accedere al motore inferenziale del componente e di richiedere la dimostrazione di un goal.

Per permettere al motore tuProlog, contenuto in *Agent Configurator*, di configurare il sistema ad agenti, si è progettata un'opportuna libreria di predicati Prolog (*Agent Library*). Questa libreria costituisce il punto di incontro tra la programmazione

logica, per la dichiarazione delle configurazioni, e la programmazione imperativa, per la manipolazione del sistema ad agenti.

Agent Library definisce i seguenti predicati Prolog:

- ***configureAgent(+PropertiesFile, +IsMain):*** configura e lancia un agente in base a quanto è specificato nel file di proprietà identificato dalla variabile *PropertiesFile.* Se la variabile *IsMain* ha valore *yes* lancia l'agente in un container principale, altrimenti in un container periferico;
- ***findallAgents(-List):*** ottiene una lista con tutti gli identificativi logici degli agenti correntemente in esecuzione sul nodo in cui risiede il motore inferenziale;
- ***createAgentContainer(+AgentId, +Name):*** crea un contenitore periferico all'interno della piattaforma JADE identificato dal valore della variabile *Name* e lo associa all'agente identificato dal valore della variabile *AgentId*;
- ***destroyAgentWrapper(+Name):*** rimuove dal sistema il container periferico identificato dal valore della variabile *Name*;
- ***migrateAgent(+AgentId, +Destination):*** avvia la migrazione dell'agente identificato dal valore della variabile *AgentId.* La destinazione dell'agente è identificata dal valore della variabile *Destination*.

Le interazioni tra la classe di supporto, il container ed il componente *Agent Configurator* sono mostrate in *figura 15:* la classe di supporto accede al componente *Agent Configurator* attraverso il *Container*. Una volta ottenuto il riferimento a questo componente, la classe di supporto invoca il metodo *configureAgents* sul componente; il metodo accede al motore inferenziale e gli richiede di dimostrare il goal *configureAgents*, scatenando il processo di configurazione e attivazione degli agenti.

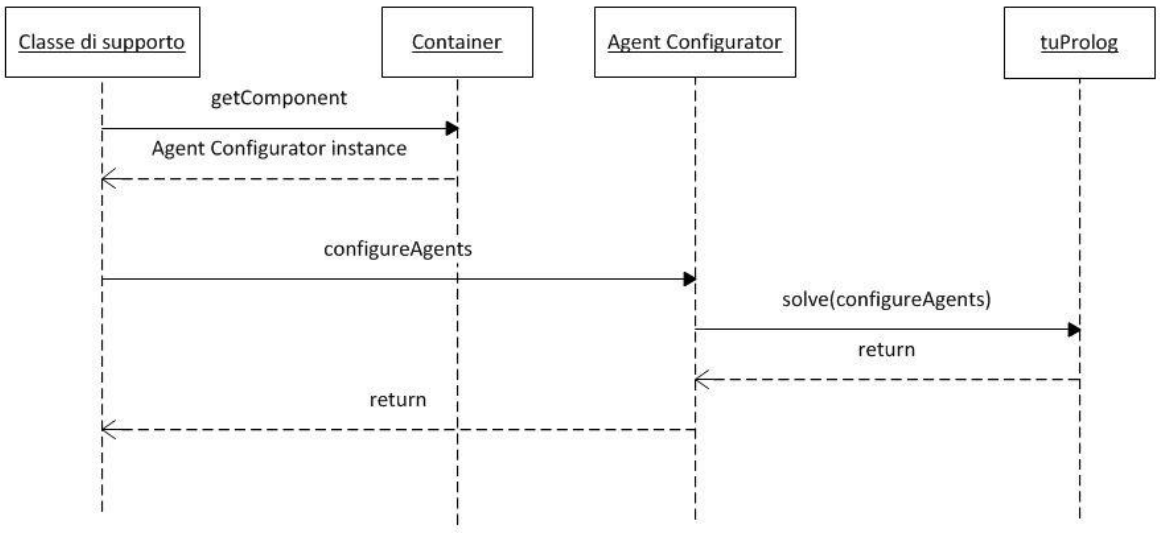

figura 15 – interazioni tra classe di supporto, container e componente Agent Configurator.

5. Implementazione

Si è scelto di sviluppare il codice come un'estensione di tuProlog (package: *alice.tuprologx.middleware*) in quanto ha l'obiettivo di fornire una libreria middleware per facilitare l'uso della programmazione logica in applicazioni distribuite basate su agenti JADE.

Per maggiori dettagli sulle estensioni apportate al core del motore inferenziale si veda il capitolo 6.

5.1 Componenti applicativi

Un componente è una classe Java che dichiara un motore tuProlog e fornisce al container dei metadati su come configurare a runtime l'engine. Un componente può dichiarare anche altre risorse, la cui gestione è a carico del programmatore, e definire metodi di business logic; non deve sovrascrivere il costruttore di default in quanto è utilizzato dal container per istanziarlo.

Per il formato dei metadati si è deciso di utilizzare le annotazioni Java, in quanto permettono di associare meta-informazioni al codice in modo semplice e diretto. Se ne sono sviluppate tre tipologie che ricalcano quelle definite in progettazione (paragrafo 4.1.1) e che servono per la gestione delle istanze del componente, per la configurazione di tuProlog ed per l'invocazione trasparente di metodi.

Tutte le annotazioni sviluppate hanno retention policy di tipo runtime, poiché il container deve poterle leggere a tempo di esecuzione e configurare di conseguenza un componente.

5.1.1 Annotazioni per la gestione delle istanze di un componente

Sono annotazioni marker, senza attributi, che possono essere applicate alla definizione di una classe.

@AsSingleton: marca un componente *singleton*. A runtime, il container ne crea soltanto un'istanza. L'unicità dell'istanza è garantita a livello di container.

@AsPrototype: opposto di *@AsSingleton*. A runtime, il container crea più istanze di un componente. Il numero delle istanze dipende da quante volte un applicativo invoca il metodo *getComponent* del container.

5.1.2 Annotazioni per la configurazione di tuProlog

Specificano come configurare ed amministrare il motore inferenziale.

@PrologConfiguration: specifica le configurazioni del motore. La *tabella 4* elenca gli attributi opzionali dell'annotazione.

Attributo	**Default**	**Descrizione**
prologTheory	stringa vuota	Teoria logica da caricare nel motore.
directives	elenco vuoto	Array di direttive da eseguire alla configurazione del motore.
fromFiles	elenco vuoto	Elenco di file *.pl* contenenti teorie Prolog da caricare nel motore.

tabella 4 – proprietà annotazione @PrologConfiguration.

@PrologManagement: specifica come attivare il sistema di management JMX (*Java Management Extensions*) di tuProlog (paragrafo 6.3.4). La *tabella 5* elenca gli attributi opzionali dell'annotazione.

Attributo	**Default**	**Descrizione**
host	localhost	Indirizzo del server JMX.
port	45000	Porta del server JMX.
lazyBoot	true	Parametro che indica se attivare il server su richiesta esplicita del programmatore (*true*) o in automatico alla creazione del motore (*false*).
adaptor	HTTP	Adattatore di protocollo per interagire con il server JMX.
credentialFile	stringa vuota	File *.txt* contenente le credenziali degli utenti amministratori.
SSLconfigFile	stringa vuota	File .properties contenente le configurazioni di SSL-TLS.

tabella 5 – proprietà annotazione @PrologManagement.

@GoalsToMatch: annotazione specifica per l'utilizzo del motore inferenziale in componenti LPaaS (paragrafo 5.1.5). Permette di specificare un insieme di stringhe che rappresentano i goal che un cliente può chiedere di dimostrare all'engine LPaaS.

5.1.3 Annotazioni per l'invocazione trasparente di metodi

Questo tipo di annotazioni permette al programmatore di marcare metodi affinché il container li invochi trasparentemente.

@PostConstructCall: marca un metodo che deve essere invocato dal container dopo la configurazione del motore inferenziale.

@PreMigrationSetup: marca un metodo che deve essere invocato prima che avvenga la migrazione del motore inferenziale del componente.

@PostMigrationSetup: marca un metodo che deve essere invocato dopo il ripristino del motore inferenziale, ovvero al termine del processo di migrazione.

@OnDispose: marca un metodo che deve essere invocato dal container prima di eliminare un componente applicativo.

5.1.4 Esempio di componente applicativo generico

La *figura 16* mostra un esempio di componente: la classe *SensorComponent* modella un componente applicativo che gestisce un sensore di un dispositivo IoT. Questa classe è annotata *@AsSingleton* in quanto si vuole che il container ne crei soltanto un'istanza. *SensorComponent* dichiara un motore tuProlog e lo marca con le annotazioni *@PrologManagement* e *@PrologConfiguration* per specificare al container come configurare l'istanza del motore.

In *@PrologManagement* si specifica che il server JMX di management (paragrafo 6.3.4) è fruibile attraverso richieste HTTP (*adaptor*) all'indirizzo IP 172.20.10.2 (*host*) e porta 45001 (*port*). Il server è attivato alla creazione del motore in quanto l'attributo *lazyBoot* è impostato a *false*. Le credenziali degli utenti amministratori del motore e le configurazioni di SSL-TLS per l'utilizzo di HTTPS sono rispettivamente indicate nei file credential.txt e ssl.properties (attributi *credentialFile* e *SSLconfigFile*).

L'annotazione *@PrologConfiguration* specifica che l'occurs check deve essere disabilitato (*directives*) e che il file *mySensorTheory.pl* contiene una o più teorie logiche da caricare nel motore.

```
@AsSingleton
public class SensorComponent {

    @PrologManagement(host = "172.20.10.2",
            port = 45001,
            lazyBoot = false,
            adaptor = PrologMXBeanServer.HTTP_ADAPTOR,
            credentialFile ="./credential.txt",
            SSLconfigFile = "./ssl.properties")
    @PrologConfiguration(directives = {"set_prolog_flag(occursCheck, off)."},
            fromFiles = {"mySensorTheory.pl"})
    private Prolog prolog;

    @PostConstructCall
    public void configureSensor(){
        manageSensorConfiguration(prolog);
    }

    @PreMigrationSetup
    public void lastWishes(){
        unlinkSensor();
    }

    @PostMigrationSetup
    public void reload(){
        manageSensorConfiguration(prolog);
    }

    @OnDispose
    public void dispose(){
        unlinkSensor();
    }

    //business logic

    private void unlinkSensor() {}
    private void manageSensorConfiguration(Prolog prolog) {}

}
```

figura 16 – classe SensorComponent.

Il metodo *configureSensor()* è marcato con l'annotazione *@PostConstructCall:* è invocato dal container dopo che il motore Prolog è stato configurato e si occupa di ultimare la configurazione del componente. Nell'esempio si simula la registrazione al gestore di un sensore.

Il metodo *lastWishes()* è marcato con l'annotazione *@PreMigrationSetup*: questo metodo è invocato dal container prima della migrazione del motore. Nell'esempio il metodo simula il rilascio delle risorse, ovvero la de-registrazione dal gestore di un sensore.

Il metodo *reload()* è marcato con l'annotazione *@PostMigrationSetup* poiché si desidera che il container lo invochi dopo che il motore è stato migrato e prima che il componente torni ad essere operativo. Nell'esempio si simula che il componente riacquisisca le risorse necessarie per l'esecuzione, ovvero la registrazione al gestore di un sensore.

Il metodo *dispose()* è marcato con l'annotazione *@OnDispose* in quanto si desidera che il container lo invochi subito prima della rimozione del componente dal sistema. Nell'esempio si ipotizza che il metodo compia la de-registrazione dal gestore di un sensore.

Possono essere presenti più metodi con la stessa annotazione: il container li invoca in ordine di definizione. Il container non gestisce la consistenza delle annotazioni: è compito del programmatore annotare in modo corretto i metodi dei componenti.

5.1.5 Componente LPaaS

Un componente LPaaS è un componente applicativo che implementa due interfacce standard di LPaaS: *client* e *configurator*.

Per semplificare lo sviluppo di componenti LPaaS si è realizzata la classe astratta *alice.tuprologx.middleware.LPaaS.LPaaSComponent* che implementa le interfacce *client* e *configurator* presentate nel paragrafo 4.1.1.1. Tutti i metodi delle interfacce restituiscono una rappresentazione testuale del risultato della computazione. Nell'implementazione fornita si è scelto di utilizzare il formato JSON per la rappresentazione dei risultati[JSON]. Questa scelta semplifica la comunicazione del risultato ad un cliente remoto.

Per realizzare componenti LPaaS custom è necessario estendere la classe *LPaaSComponent* e dichiarare opportunamente un motore tuProlog in base al servizio che si vuole rendere disponibile (un esempio concreto è presentato nel capitolo 7).

5.1.5.1 Note sull'utilizzo

I componenti LPaaS sono pensati per implementare il core di un servizio LPaaS. Si è scelto di affidare al componente soltanto la gestione della lista di goal che possono essere richiesti da un cliente. La comunicazione tra entità remote, la sicurezza ed il meccanismo per l'invocazione dei metodi dei componenti dipendono da come vengono integrati i componenti LPaaS con il software che si occupa della distribuzione. Nella soluzione proposta è l'agente LPaaS (ed il relativo behaviour) ad occuparsi di tutti questi aspetti. Per maggiori dettagli si veda il paragrafo 5.3.2.

5.2 Il container

Il container è lo strumento che solleva lo sviluppatore dalla necessità di configurare ed istanziare manualmente i componenti applicativi. Per garantire l'indipendenza da una specifica implementazione, si è definita un'interfaccia standard (*tabella 6*) che ogni oggetto container deve implementare. È possibile fornire varie implementazioni che ottimizzino fattori diversi, come la gestione delle istanze dei componenti e/o le strategie di serializzazione dei motori inferenziali.

alice.tuprologx.middleware.interfaces.IComponentConfigurator	
Metodo	**Descrizione**
void setConfigFileLocation(String fileName)	Imposta il path del file di configurazione del container.
String getConfigFileLocation()	Restituisce il path del file di configurazione del container.
void configure() throws ComponentConfiguratorException	Effettua il bootstrap del container configurando i componenti.
boolean isConfigured()	Restituisce *true* se il container è stato configurato correttamente, altrimenti *false*.
<T> T getComponent(String id)	Restituisce un componente di tipo *T* a partire dal suo identificatore. Se il componente richiesto è annotato *@AsSingleton*, il metodo restituisce l'unica istanza del componente, altrimenti restituisce una nuova istanza del componente occupandosi della configurazione del motore inferenziale.
<T> T getComponentMetaData(String id)	Restituisce i metadati del componente identificato dalla stringa *id*.
void removeComponent(String id)	Rimuove dal container il componente identificato dalla stringa *id*.
void preMigrationSetup()	Effettua le operazioni necessarie per effettuare la migrazione del container e dei suoi componenti.
void postMigrationSetup() throws ComponentConfiguratorException	Riconfigura il container ed i suoi componenti dopo la migrazione.
String toJSON()	Ottiene una rappresentazione JSON dello stato dei motori inferenziali dei componenti presenti nel container.

void fromJSON(String jsonString)	Ottiene lo stato dei motori inferenziali dei componenti a partire dalla loro rappresentazione JSON.
void destroy()	Elimina tutti i componenti contenuti nel container.

tabella 6 – interfaccia del Container.

Il container è in grado di gestire sia componenti LPaaS che non, ma si consiglia di non unire diverse tipologie di componenti in quanto tale scelta può complicare la logica applicativa del sistema che si desidera sviluppare.

5.2.1 Configurazione dei componenti

L'implementazione del container prevede che il programmatore fornisca un file di configurazione in cui, per ogni componente da istanziare e configurare, sia indicato l'identificativo logico del componente ed il nome completo della classe che lo implementa (*figura 17*). Queste informazioni sono necessarie in quanto il container deve identificare un componente e conoscere quale classe istanziare. Il file di configurazione ha estensione *.properties* e rispetta la sintassi dei file di proprietà Java, ovvero contiene una coppia *proprietà = valore* per ogni riga del file.

```
componentId.1=comp1
componentClass.1=it.unibo.test.MyComponent1
componentId.2=comp2
componentClass.2=it.unibo.test.MyComponent2
```

figura 17 – esempio di file di configurazione del container.

La proprietà *componentId.N* specifica l'identificativo logico dell'ennesimo componente e la proprietà *componentClass.N* indica la classe che lo implementa. In *figura 17*, si definiscono due componenti *comp1* e *comp2* rispettivamente di classe *it.unibo.test.MyComponent1* e *MyComponent2*.

Nello stato di *Bootstrap* (metodo *configure*) il container configura i componenti seguendo il diagramma degli stati mostrato in *figura 18.*

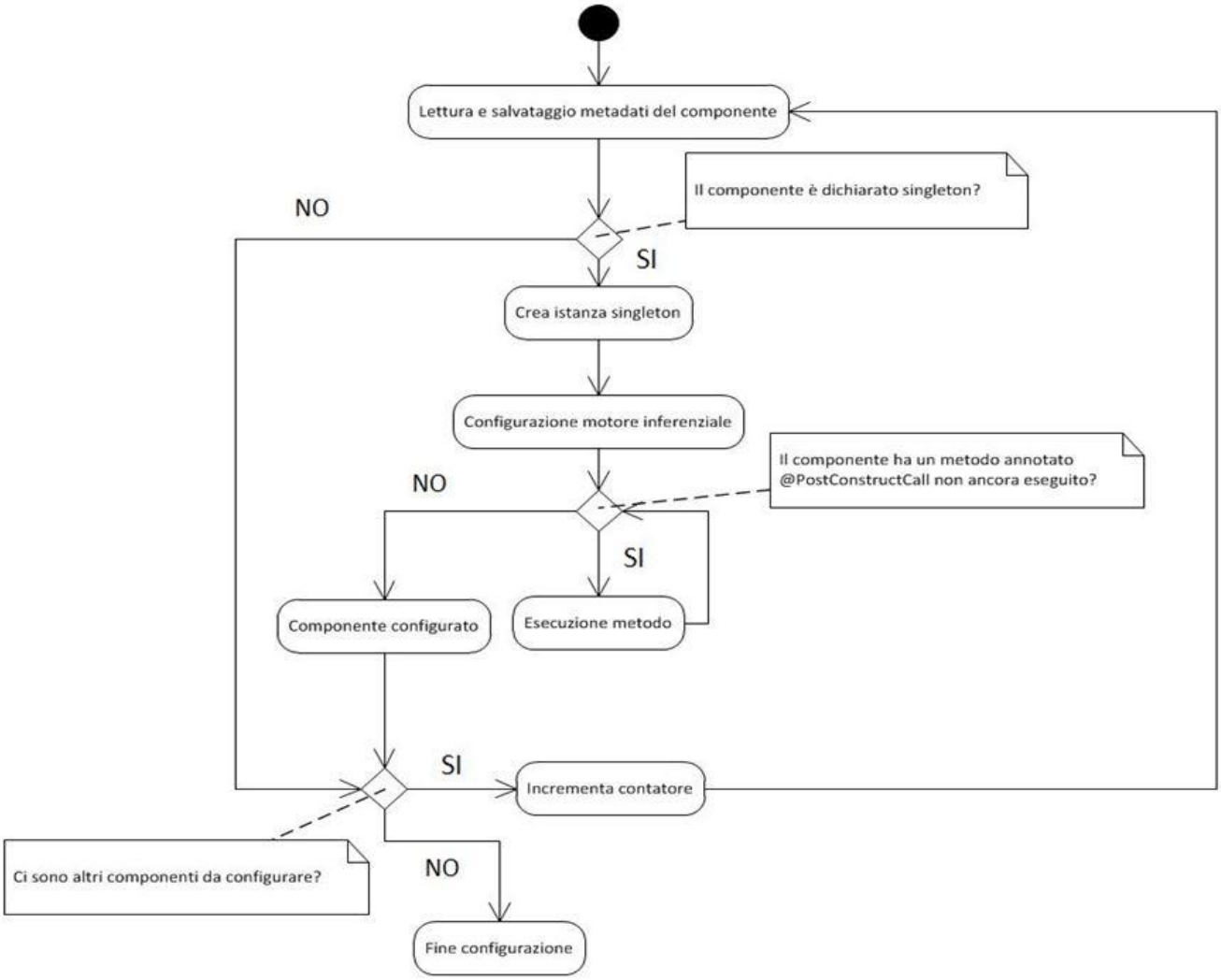

figura 18 – azioni compiute dal container nello stato di Bootstrap.

Il container ciclicamente legge e memorizza la coppia di proprietà di un componente e tramite reflection controlla se la classe del componente è marcata *@AsSingleton* o *@AsPrototype.* Nel primo caso istanzia il componente e procede nella configurazione, nel secondo prosegue con l'analisi dei componenti da istanziare. I componenti marcati *@AsPrototype* vengono creati e configurati solo quando un'applicazione client li richiede al container. Si ha quindi una configurazione *eager* per i componenti *@AsSingleton* ed una configurazione *lazy* per i componenti *@AsPrototype.* Si è scelto questo approccio per ridurre il tempo di accesso ad oggetti *@AsSingleton* e per pagare l'overhead di creazione di oggetti *@AsPrototype* solo se

effettivamente utilizzati. Nel caso in cui la classe di un componente non fosse annotata, il container gestisce il componente come un singleton.

La configurazione di un componente prevede che il container controlli, tramite reflection, se il componente dichiara un motore tuProlog: in caso affermativo si procede alla configurazione del motore, altrimenti si ha un'eccezione a runtime.

Il container controlla se sono presenti le annotazioni *@PrologConfiguration* e *@PrologManagement*:

- Se entrambe le annotazioni sono presenti, prima legge il contenuto dell'annotazione *@PrologConfiguration* per configurare l'istanza del motore, poi legge il contenuto di *@PrologManagement* per configurare il sistema di management del motore. Se l'attributo *lazyBoot* ha valore *false*, il container attiva il server JMX associato al motore.
- Se è presente solo *@PrologConfiguration* il container configura il motore ed ignora la configurazione del sistema di management.
- Se è presente solo *@PrologManagement* il container non compie alcuna azione, in quanto non è possibile avviare il sistema di management senza aver configurato il motore. Quest'ultimo caso costituisce un errore di programmazione.

Conclusa la configurazione del motore, il container controlla se il componente dichiara metodi marcati con l'annotazione *@PostConstructCall*: in caso affermativo li invoca in ordine di definizione, altrimenti termina la configurazione del componente e, se il componente è marcato @AsSingleton, lo inserisce in una mappa che mantiene le corrispondenze tra gli oggetti singleton ed il loro nome logico.

5.2.2 Serializzazione e riattivazione dei componenti

Il container supporta la serializzazione automatica dei motori inferenziali dei componenti *@AsSingleton*. Per quanto riguarda i componenti *@AsPrototype*, si è scelto di affidare la gestione della serializzazione al programmatore in quanto l'implementazione realizzata del container non tiene traccia di questa tipologia di componenti dopo la loro configurazione.

La serializzazione è utilizzata nel processo di migrazione del container. Lo schema in *figura 19* mostra le azioni compiute dal container nello stato di *Pre-Migration Setup*. Per ciascun componente il container controlla se questo possiede dei metodi marcati con l'annotazione *@PreMigrationSetup*: in caso affermativo li invoca in ordine di definizione, altrimenti procede con la serializzazione dello stato del motore inferenziale. Al termine del ciclo di serializzazione, il container è pronto per la migrazione.

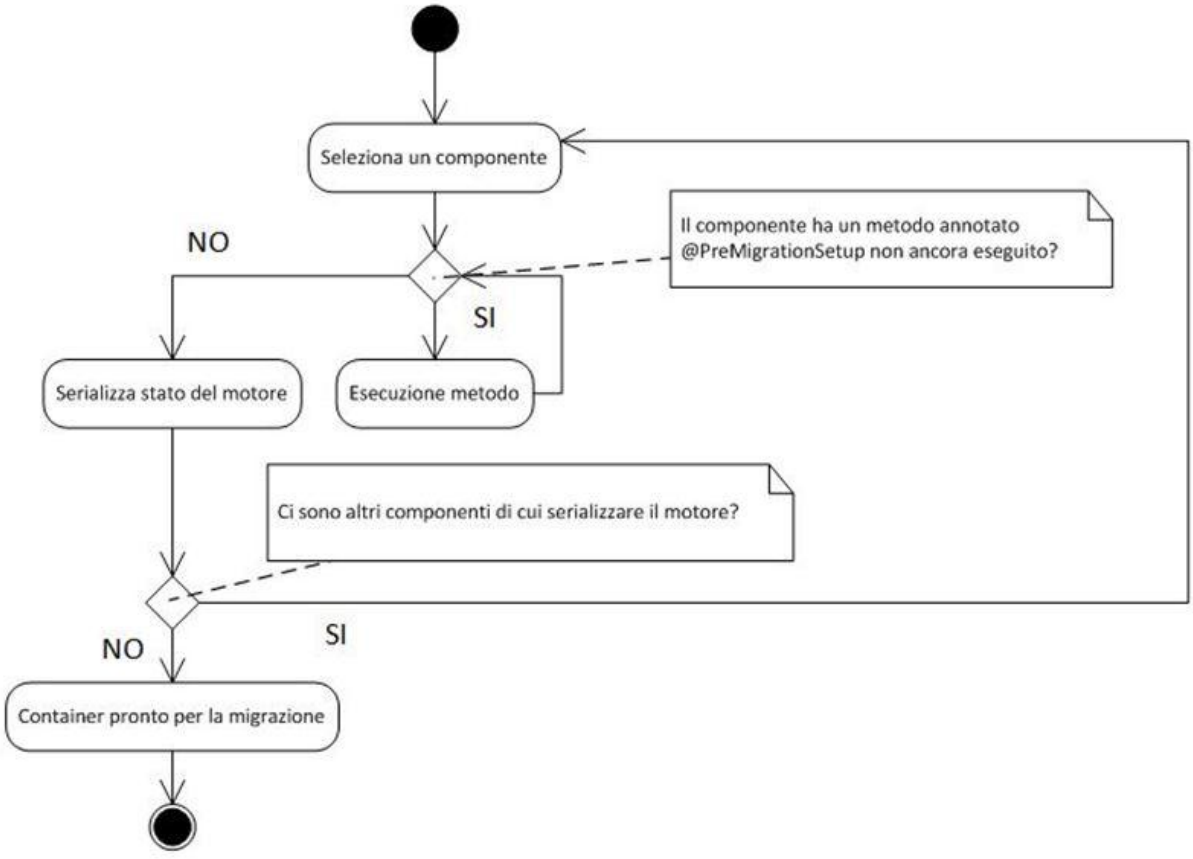

figura 19 – azioni compiute dal container nello stato di Pre-Migration Setup.

Le azioni compiute dal container nello stato di *Post-Migration Setup* sono mostrate in *figura 20*: il container ciclicamente seleziona un componente da riattivare e ne ripristina il motore inferenziale a partire dallo stato serializzato. Dopo il ripristino del motore, il container invoca, in ordine di definizione, tutti i metodi marcati *@PostMigrationSetup:* questo meccanismo permette al componente di riacquisire le risorse di cui necessita per l'esecuzione

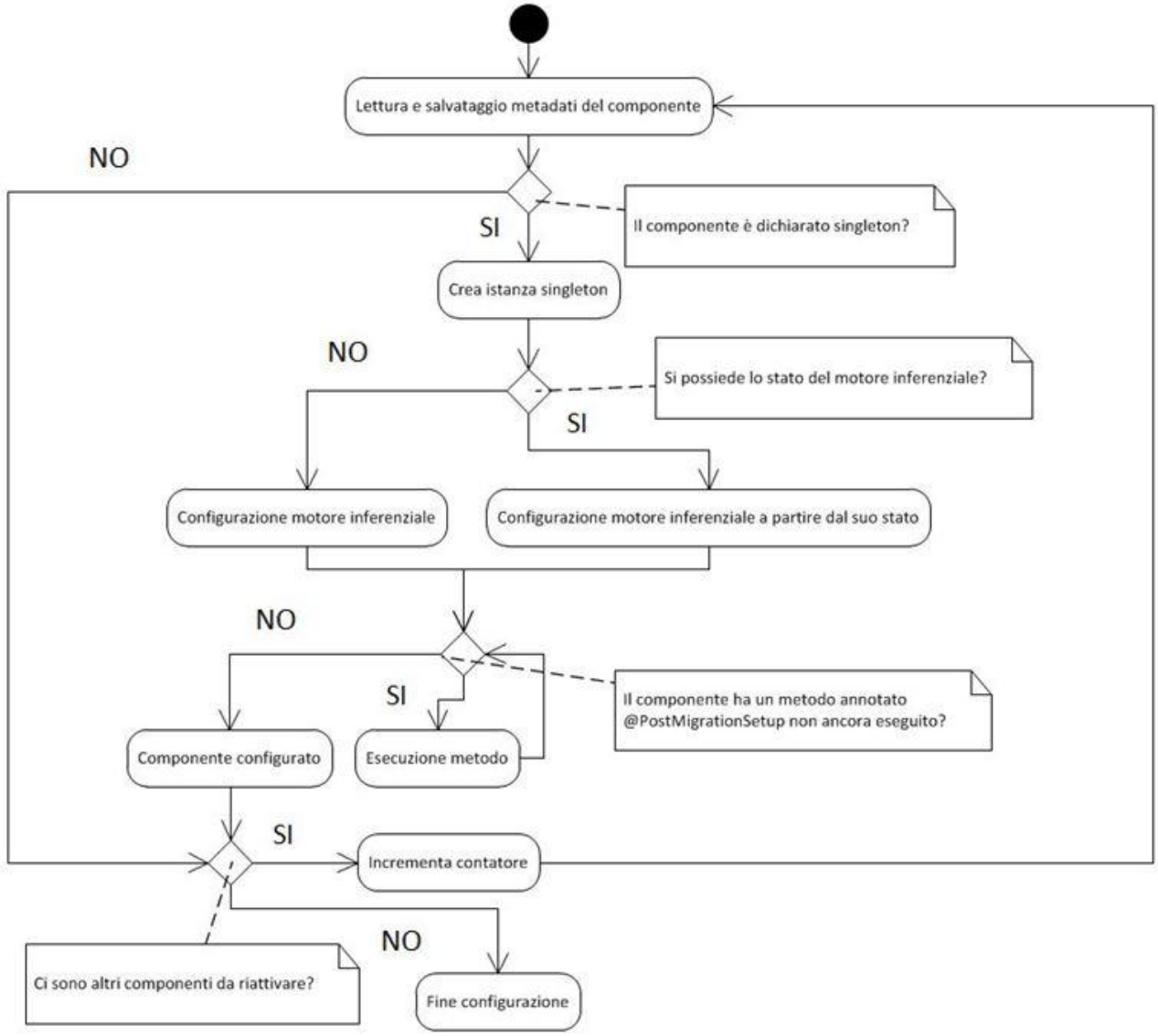

figura 20 – azioni compiute dal container nello stato di Post-Migration Setup.

5.2.3 Note sull'utilizzo

L'implementazione fornita del container non è thread-safe: è compito del programmatore gestire accessi concorrenti a tale risorsa. Si sconsiglia un utilizzo multithreaded in quanto può portare ad errori a runtime se i thread applicativi non hanno una visione consistente dello stato del container.

5.3 Agenti JADE e programmazione logica

5.3.1 Agente per la gestione di componenti applicativi

La classe *alice.tuprologx.middleware.agents.jade.java.ContainerAgent* implementa l'agente astratto che si occupa di gestire il ciclo di vita di un container per componenti applicativi. Come mostrato in *figura 21*, la classe *ContainerAgent* estende la classe *jade.core.Agent* e possiede un riferimento ad un container di tipo *IComponentConfigurator*.

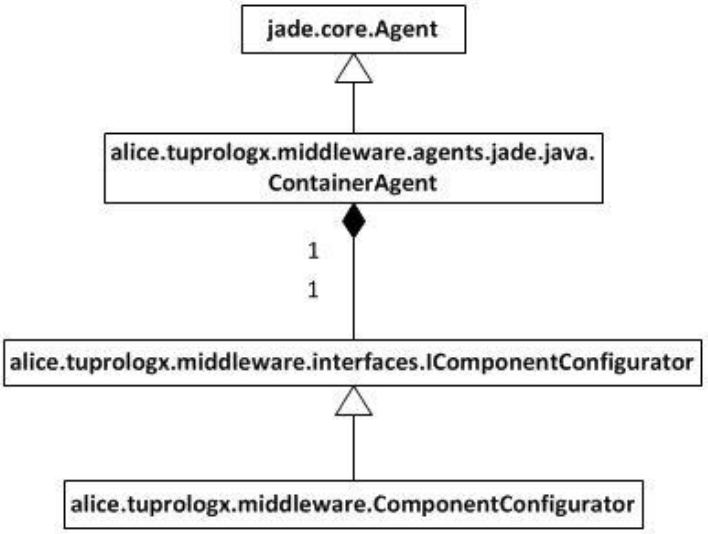

figura 21 – implementazione agente astratto per la gestione del ciclo di vita del container.

Si riporta la descrizione delle implementazioni dei metodi che permettono di coordinare il ciclo di vita del container con quello dell'agente.

Nel metodo *setup()* della classe *ContainerAgent* si effettuano le seguenti operazioni:

- si accede alle proprietà dell'agente mediante il metodo *getBootProperties()* e si legge il valore della proprietà *component-configurator-file*, la quale deve contenere il nome assoluto del file di configurazione del container;
- si ottiene un'istanza del container attraverso la classe di supporto *Middleware* (contenuta nel package *alice.tuprologx.middleware*);
- si effettua il bootstrap del container passandogli il nome assoluto del file di configurazione (invocazione del metodo *setConfigFileLocation* e del metodo *configure*).

Nel metodo *beforeMove()* si invoca sull'istanza del container il metodo *preMigrationSetup()* che fa transitare il container nell'omonimo stato.

In modo duale, nel metodo *afterMove()*, si invoca sull'istanza del container il metodo *postMigrationSetup()*.

Nel metodo *takeDown()* si invoca il metodo *destroy()* del container che provvede ad eliminare tutti i componenti gestiti dal container.

5.3.2 Agente LPaaS

L'implementazione di un agente LPaaS estende e concretizza la classe *ContainerAgent* (*figura 22*).

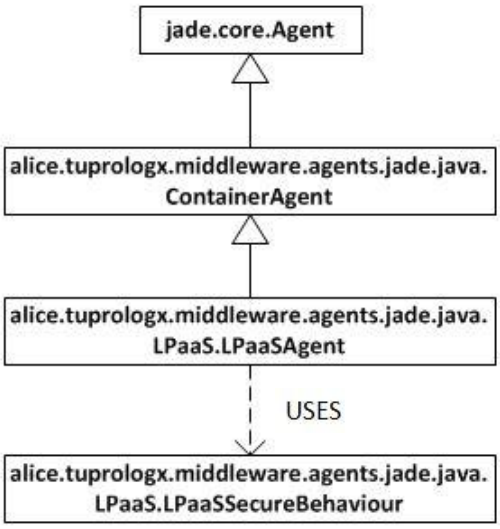

figura 22 – implementazione agente LPaaS.

La classe *LPaaSAgent* ridefinisce ed estende i metodi *setup()* e *takeDown()* per gestire la registrazione dell'agente presso il DF. Questa classe è contenuta nel package *alice.tuprologx.middleware.agents.jade.java.LPaaS.*

Il metodo *setup()* crea un oggetto di classe *DFAgentDesctiption*, che rappresenta una descrizione dell'agente, e gli associa un oggetto di tipo *ServiceDescription* in cui è salvato il nome logico del servizio e la sua tipologia (LPaaS). Il metodo procede poi alla registrazione dell'agente presso il DF invocando il metodo *register*, della classe statica *DFService*, il quale richiede un riferimento all'agente e all'oggetto di tipo *DFAgentDesctiption*. Infine il metodo crea un behaviour di classe *LPaaSSecureBehaviour* e lo aggiunge alla lista dei behaviour che l'agente deve eseguire (paragrafo 5.3.2.2).

Il metodo *takeDown()*, prima di cedere il controllo alla superclasse, si occupa di invocare il metodo *deregister* della classe *DFService*, per cancellare la registrazione dell'agente presso il DF.

5.3.2.1 LPaaS Request/Response DTO

Per quanto riguarda la comunicazione tra un agente cliente ed un agente LPaaS, si è deciso di implementare due classi conformi al pattern *Data Transfer Object* (DTO): la classe *alice.tuprologx.middleware.LPaaS.LPaaSRequest* modella una richiesta di servizio LPaaS, mentre la classe *alice.tuprologx.middleware.LPaaS.LPaaSResponse* modella il risultato dell'esecuzione di un servizio LPaaS.

LPaaSRequest dichiara i seguenti campi:

- ***componentName:*** permette di specificare il nome logico del componente LPaaS a cui richiedere un servizio;
- ***componentAction:*** è una stringa che identifica univocamente un metodo di un componente LPaaS (*figura 23*);
- ***content:*** in base a quanto specificato in *componentAction*, può rappresentare un goal da dimostrare, un goal da aggiungere/rimuovere a/da quelli esposti da un componente LPaaS o una teoria da aggiungere alla base di conoscenza di un componente LPaaS.
- ***timeout:*** consente di specificare la durata massima (in millisecondi) di un servizio LPaaS (opzionale);
- ***numSol:*** permette di specificare il numero di soluzioni da esplorare (opzionale).

LPaaSRequest definisce anche una serie di costanti stringa (*figura 23*) che identificano univocamente i metodi delle interfacce *client* e *configurator* implementate da un componente LPaaS (paragrafo 4.1.1.1).

```
public static final String SOLVE = "solve";
public static final String SOLVE_N = "solveN";
public static final String SOLVE_N_TIMED = "solveNTimed";
public static final String SOLVE_ALL = "solveAll";
public static final String SOLVE_ALL_TIMED = "solveAllTimed";
public static final String SOLVE_TIMED = "solveTimed";
public static final String ADD_GOAL = "addGoal";
public static final String REMOVE_GOAL = "removeGoal";
public static final String ADD_THEORY = "addTheory";
public static final String IS_GOAL = "isGoal";
public static final String GET_GOAL_LIST = "getGoalList";
public static final String SOLVE_NEXT = "solveNext";
public static final String SOLVE_NEXT_TIMED = "solveNextTimed";
```

figura 23 – costanti stringa che identificano i metodi delle interfacce LPaaS client e configurator.

LPaaSResponse dichiara i seguenti campi:

- ***message:*** permette di specificare un messaggio di errore;
- ***response:*** è il risultato dell'esecuzione del servizio LPaaS.

I DTO *LPaaSRequest* e *LPaaSResponse* sono inclusi nei rispettivi messaggi ACL attraverso il metodo *setContentObject* della classe *ACLMessage*.

5.3.2.2 Behaviour sicuro

La classe *LPaaSSecureBehaviour*, contenuta nel package *alice.tuprologx.middleware.agents.jade.java.LPaaS*, implementa il comportamento di un agente LPaaS. *LPaaSSecureBehaviour* estende *CyclicBehaviour* ed utilizza i servizi offerti dall'add-on JADE-S per autenticare e cifrare/decifrare la comunicazione con i clienti LPaaS.

Il behaviour esegue ciclicamente le seguenti operazioni:

1. Attende in modo bloccante l'arrivo di un messaggio LPaaS.
2. Alla ricezione di un messaggio di richiesta di servizio, controlla, attraverso il *Security Helper* di JADE-S, se il messaggio è firmato da un agente conosciuto. In caso affermativo procede con la decifrazione del messaggio, altrimenti risponde con un messaggio di errore. Si ha un errore anche nel caso in cui il messaggio ricevuto sia in chiaro o non cifrato correttamente.
3. Successivamente il behaviour estrae dal messaggio l'oggetto DTO di classe *LPaaSRequest*. Se il servizio richiesto appartiene all'interfaccia *configurator* di un componente LPaaS, il behaviour controlla che l'identità del mittente sia nell'elenco dei clienti che hanno accesso a tale interfaccia: in caso affermativo invoca il servizio sul componente LPaaS specificato nella richiesta, altrimenti risponde con un messaggio di errore. Se il servizio richiesto appartiene all'interfaccia *client*, il behaviour non esegue alcun controllo sull'identità del cliente.
4. Infine, al termine dell'esecuzione del servizio, il behaviour risponde al mittente con un messaggio contenente un oggetto DTO di classe *LPaaSResponse*. Il messaggio di risposta è firmato e cifrato.

Per l'autenticazione e la cifratura dei messaggi, si è utilizzata la configurazione di default di JADE-S che prevede di utilizzare l'algoritmo RSA a 512 bit per la firma digitale e l'algoritmo AES a 128 bit per la cifratura.

5.4 Configurazione dichiarativa del sistema ad agenti

Come esposto al paragrafo 4.3, il supporto per la configurazione dichiarativa del sistema ad agenti si suddivide in tre elementi architetturali: una classe di supporto, un container per componenti applicativi ed un componente ad hoc. La classe di supporto è implementata dalla *classe alice.tuprologx.middleware.launcher.Launcher* (in blu in *figura 24*): questa classe utilizza un'istanza del container (in verde) ed invoca il metodo *configureAgents()* sul componente *Agent Configurator* (in rosso). Il componente *Agent Configurator* dichiara un motore tuProlog che utilizza principalmente due librerie: *OOLibrary* ed *AgentLibrary*. La prima è fornita dal motore inferenziale e permette al codice Prolog di interfacciarsi con il codice Java, la seconda utilizza la prima ed è stata sviluppata per poter manipolare la configurazione del sistema ad agenti direttamente dal Prolog.

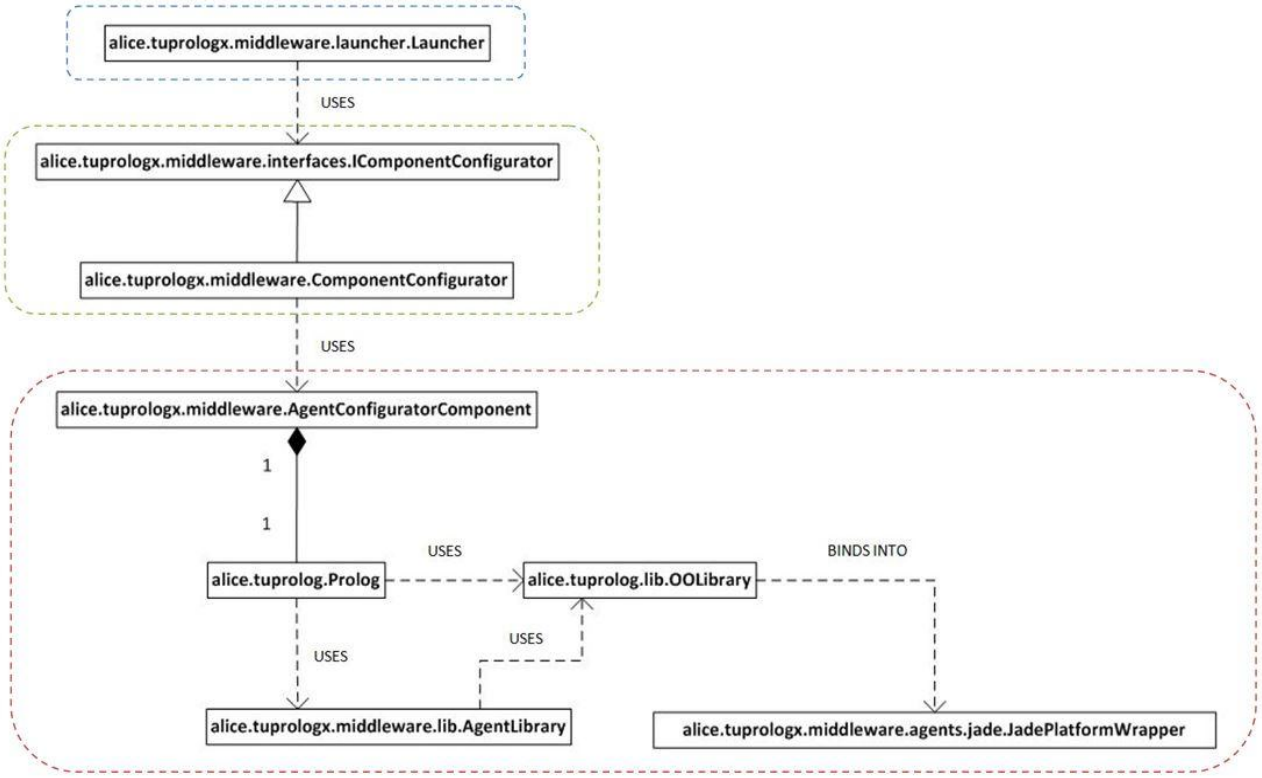

figura 24 – implementazione del supporto per la configurazione dichiarativa del sistema ad agenti.

Agent Library definisce la seguente teoria Prolog:

```
configureAgents :-
    findall(agent(PropertiesFile, IsMain), agent(PropertiesFile, IsMain), List),
    configureAgentsFromList(List).
configureAgentsFromList([]).
configureAgentsFromList([agent(PropertiesFile, IsMain)|T]) :-
    configureAgent(PropertiesFile, IsMain), configureAgentsFromList(T).
```

La teoria Prolog definita da *Agent Library* viene consultata quando la classe *Launcher* invoca il metodo *configureAgents()* sul componente *Agent Configurator*. La teoria prevede di elencare tutte le clausole *agent(PropertiesFile, IsMain)* che contengono le informazioni sul file di configurazione dell'agente da attivare (*PropertiesFile*) e su dove l'agente deve essere avviato: in un container principale o periferico (*IsMain*). Dopo aver elencato le clausole di configurazione, il predicato *ConfigureAgentsFromList* scorre la lista di clausole e chiama il predicato *configureAgent* che si occupa di configurare ed attivare un agente JADE. Quest'ultimo predicato utilizza le funzionalità di tuProlog che consentono a codice Prolog di invocare codice Java; all'interno della sua definizione, il predicato invoca un metodo Java che istanzia un oggetto di classe *alice.tuprologx.middleware.agents.jade.JadePlatformWrapper*, il quale è un wrapper per la piattaforma JADE. L'oggetto wrapper permette di creare un container JADE (principale o periferico) e di avviarvi un agente. Gli oggetti wrapper sono registrati come oggetti statici in *OOLibrary*: in questo modo è possibile accedervi da Prolog in qualsiasi momento.

Gli altri predicati Prolog di *Agent Library* che beneficiano della natura multi-paradigma di tuProlog sono:

- ***createAgentContainer(+AgentId, +Name):*** crea un nuovo container JADE periferico attraverso un oggetto di tipo *JadePlatformWrapper*. Il wrapper è memorizzato in *OOLibrary* ed è associato all'identificativo univoco *Name*.
- ***destroyAgentWrapper(+Name):*** interrompe l'esecuzione del container JADE identificato da *Name*. Il predicato provvede anche a deregistrare da *OOLibrary* l'oggetto *JadePlatformWrapper* associato al container di cui si è fermata l'esecuzione.
- ***migrateAgent(+AgentId, +Destination):*** scatena la migrazione dell'agente identificato da *AgentId* verso il container JADE identificato da *Destination*. Il predicato utilizza il *JadePlatformWrapper* dell'agente identificato da *AgentId* per attivare il processo di migrazione.

L'implementazione del supporto alla configurazione dichiarativa del sistema ad agenti fornisce i meccanismi base per configurare una piattaforma JADE. Il supporto non è reattivo ai cambiamenti del contesto d'esecuzione dei singoli agenti. Il programmatore può aggiungere tale comportamento agendo direttamente sulla classe *Launcher* e sul componente *Agent Configurator*.

6. tuProlog: modifiche per l'utilizzo in un sistema distribuito ad agenti

In questo capitolo si discuteranno le caratteristiche principali di tuProlog (paragrafi 6.1 e 6.2) e le modifiche che si sono rese necessarie al fine di poter utilizzare efficacemente il motore in un sistema distribuito IoT ad agenti (paragrafo 6.3). Tali modifiche riguardano l'introduzione del concetto di tempo massimo di dimostrazione di un goal Prolog (paragrafo 6.3.2), la serializzazione dello stato del motore (paragrafo 6.3.3) ed i meccanismi per monitorare, controllare e gestire da remoto i componenti del core di tuProlog (paragrafo 6.3.4).

6.1 Architettura del motore inferenziale

tuProlog possiede un core inferenziale malleabile, ovvero facilmente modificabile ed estendibile dal punto di vista architetturale. La malleabilità non è una proprietà atomica e comprende l'estensibilità, la manutenibilità e la possibilità di personalizzare il software secondo le proprie esigenze. Per estensibilità si intende la possibilità di aggiungere facilmente funzionalità al motore senza pregiudicarne l'architettura. Questa proprietà è ottenuta grazie al fatto che i componenti del core sono tra loro separati. Per manutenibilità si intente la capacità di fare interventi correttivi al software senza alterarne la struttura. Infine la facilità di personalizzazione di tuProlog deriva dalle precedenti proprietà e consente di adattare l'engine a diversi contesti applicativi.

La malleabilità del core è stata ottenuta scomponendo quest'ultimo in diversi sottosistemi, ciascuno dei quali è gestito secondo il pattern singleton ed implementa una specifica funzionalità. tuProlog non espone direttamente i sottosistemi core, ma li rende accessibili attraverso un oggetto *façade.*

Il core si compone di:

Library manager: si occupa di gestire le librerie di predicati Prolog con cui è possibile arricchire le funzionalità dell'engine.

Theory manager: implementa i meccanismi che gestiscono la base di conoscenza del motore. Si occupa di memorizzare le teorie Prolog statiche e dinamiche.

Primitive manager: identifica quali predicati Prolog si mappano su metodi Java e ne mantiene informazioni utili per l'esecuzione a runtime.

Flag manager: gestisce i flag Prolog presenti nel sistema. I flag sono utilizzati dal motore per testare condizioni durante la dimostrazione di un goal.

Operator manager: gestisce gli operatori utilizzati nelle teorie logiche.

Engine manager si occupa di interagire con l'automa a stati finiti utilizzato nell'algoritmo di dimostrazione di un goal Prolog. Fornisce i metodi per risolvere un goal ed ottenerne la soluzione.

6.2 Automa a stati finiti

L'algoritmo di dimostrazione di un goal Prolog, fulcro del core del motore, utilizza l'automa a stati finiti (AST) mostrato in *figura 25*.

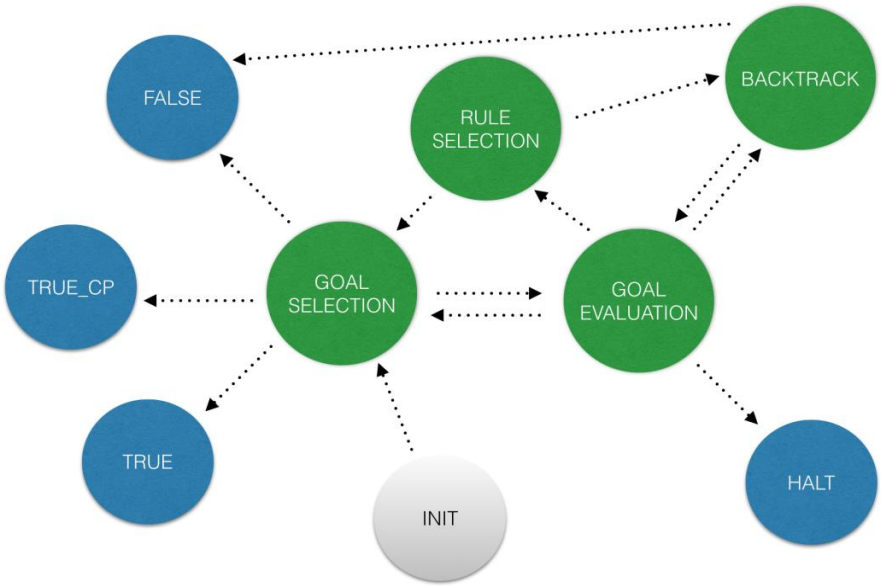

figura 25 - automa a stati finiti per la risoluzione di un goal Prolog.

L'automa è composto da uno stato iniziale (in grigio), utilizzato dal motore all'inizio della dimostrazione di un goal, da quattro stati principali (in verde) che rappresentano le azioni compiute dal motore durante la risoluzione di un goal e da quattro stati finali (in blu) che rappresentano i diversi modi in cui una dimostrazione può terminare[2PARCH].

Lo stato *INIT* dell'AST è il punto di partenza della dimostrazione di un goal e si occupa di inizializzare il core del motore estraendo i sub-goal da valutare e configurando il contesto di esecuzione del primo sub-goal. Quest'ultimo è scelto

dallo stato *GOAL SELECTION*, a cui *INIT* cede il controllo non appena ha completato le azioni di inizializzazione del core. Lo stato di *GOAL SELECTION* si occupa di estrarre dalla lista dei sub-goal il primo goal da dimostrare: se tale goal non esiste in quanto la lista dei sub-goal è vuota ed il processo di dimostrazione è terminato, controlla l'esistenza di un altro punto di scelta nella dimostrazione appena conclusa. Se tale punto di scelta esiste, l'AST passa nello stato finale di *TRUE_CP*, da cui può esplorare l'ulteriore soluzione del goal dimostrato, altrimenti l'AST passa nello stato *TRUE* e la dimostrazione del goal termina. Se la lista dei sub-goal non è vuota, lo stato di *GOAL SELECTION* estrae il goal da valutare e ne verifica la dimostrabilità. Se tale controllo fallisce, l'AST passa nello stato di *FALSE* e la dimostrazione termina con esito negativo, altrimenti l'automa passa nello stato di *GOAL EVALUATION*. In questo stato si possono verificare due casi: il funtore del goal in analisi è legato ad una primitiva del sistema oppure non lo è. Nel primo caso si esegue la primitiva e, se l'esito è positivo, l'AST transita nello stato di *GOAL SELECTION*, altrimenti in quello di *BACKTRACK*. Se il funtore non è legato ad alcuna primitiva, l'automa passa nello stato *RULE SELECTION* per selezionare la clausola Prolog da utilizzare nella dimostrazione del goal corrente. Se durante il processo di valutazione del goal si ha un errore, l'AST transita nello stato di *HALT*.

Lo stato di *RULE SELECTION* si occupa di estrarre dalla base di conoscenza del motore le clausole compatibili con il goal che si vuole dimostrare. Se non identifica alcuna clausola, l'automa passa nello stato di *BACKTRACK*, altrimenti crea un nuovo contesto di esecuzione, transita nello stato di *GOAL SELECTION* e procede alla dimostrazione del goal corrente. Nello stato di *BACKTRACK* l'automa controlla sempre se sono disponibili clausole compatibili con l'ultimo punto di scelta lasciato aperto nella dimostrazione. Se non dovessero esserci clausole compatibili, dallo stato di *BACKTRACK* l'automa transita nello stato *FALSE* terminando con fallimento la dimostrazione del goal.

6.3 Motore inferenziale e sistema ad agenti JADE

6.3.1 Requisiti

Affinché tuProlog possa essere utilizzato efficacemente all'interno di un sistema distribuito IoT ad agenti JADE, deve soddisfare il seguente elenco di requisiti.

Leggerezza: il motore non deve consumare troppe risorse del nodo su cui esegue.

Estendibilità: il motore deve essere facilmente estendibile per potersi integrare al meglio con future evoluzioni del sistema ad agenti.

Configurabilità dinamica: il motore deve permettere di configurare la base di conoscenza in modo da adattarla sia al contesto applicativo sia all'ambiente in cui un agente esegue.

Supporto alla programmazione multi-paradigma: il motore deve coniugare l'utilizzo della programmazione logica e di quella imperativa per potersi interfacciare con il sistema ad agenti.

Gestione del tempo nella dimostrazione di un goal: il motore deve poter arrestare la dimostrazione di una query che richiede un tempo infinito di esecuzione.

Supporto alla serializzazione dello stato del motore: il motore deve essere serializzabile e supportare la migrazione a runtime della propria base di conoscenza.

Supporto al management del motore da remoto: il motore deve disporre di meccanismi che consentano di fare monitoraggio, controllo e gestione da remoto dei propri sottosistemi.

La *tabella 7* mostra quali requisiti sono rispettati dalla versione attuale di tuProlog (3.2.1) e quali invece devono essere implementati.

Requisito	
Leggerezza	**Si**
Estendibilità	**Si**
Configurabilità dinamica	**Si**
Supporto alla programmazione multi-paradigma	**Si**
Gestione del tempo nella dimostrazione di un goal	**No**
Supporto alla serializzazione dello stato del motore	**No**
Supporto al management del motore da remoto	**No**

tabella 7 – requisiti per l'utilizzo di tuProlog in un sistema ad agenti JADE.

tuProlog manca di meccanismi opportuni per gestire il tempo di dimostrazione di una query Prolog, per serializzare lo stato del motore e per fare management del core dell'engine da remoto. Queste funzionalità sono necessarie al fine di consentire un utilizzo efficiente della programmazione logica in un sistema IoT ad agenti. Nei paragrafi successivi si tratterà delle modifiche apportate a tuProlog per renderlo conforme ai requisiti elencati.

6.3.2 Gestione del tempo e dimostrazione di una query Prolog

6.3.2.1 Analisi del problema e requisiti

Generalmente l'operazione di *solve* di una query Prolog richiede un tempo *t* che dipende dalla teoria logica che si consulta. Per questo motivo il parametro *t* può assumere sia valori finiti, per teorie finite, che tendere all'infinito nel caso in cui la teoria Prolog sia ciclica.

In un contesto distribuito, un cliente che richiede l'operazione di *solve* su un motore inferenziale remoto, percepisce un tempo di esecuzione del comando pari alla somma del tempo necessario per l'invio della richiesta, l'esecuzione del comando e la ricezione della risposta. Nel caso in cui la query richiedesse un tempo di esecuzione infinito, ovvero causasse un loop nel motore Prolog, il cliente remoto non riceverebbe alcuna risposta. Per evitare scenari di questo tipo, un cliente remoto deve poter specificare il tempo massimo di esecuzione di una query, allo scadere del quale la dimostrazione deve essere interrotta e fallire. In tuProlog ciò comporta la modifica del processo di risoluzione di una query in quanto il motore deve essere sensibile allo scorrere del tempo. Nel paragrafo successivo si presenteranno le modifiche fatte all'automa a stati finiti per gestire vincoli di tempo nel processo di dimostrazione di una query.

6.3.2.2 Progettazione: nuovo automa a stati finiti

Osservando l'automa descritto al paragrafo 6.2, si nota la possibilità di avere cicli tra gli stati di *GOAL SELECTION*, *GOAL EVALUATION*, *RULE SELECTION* e *BACKTRACK*.

Per evitare possibili loop e terminare la computazione in un tempo finito bisogna modificare l'AST ed aggiungere opportuni controlli temporali quando l'automa transita da/verso uno stato di quelli appena menzionati.

In *figura 26* è mostrato l'AST di tuProlog modificato: si è aggiunto un nuovo stato finale chiamato *TIME ELAPSED* (in rosso). Ad ogni transizione di stato tra gli stati *GOAL SELECTION*, *GOAL EVALUATION*, *RULE SELECTION* e *BACKTRACK* si controlla se la computazione ha consumato tutto il tempo a disposizione ed in caso affermativo, si termina la dimostrazione con fallimento transitando nello stato di *TIME ELAPSED*. L'informazione sul tempo a disposizione per la dimostrazione del goal viene fornita all'automa in fase di inizializzazione. Se non presente, il comportamento dell'AST rimane inalterato e lo stato *TIME ELAPSED* diviene irraggiungibile.

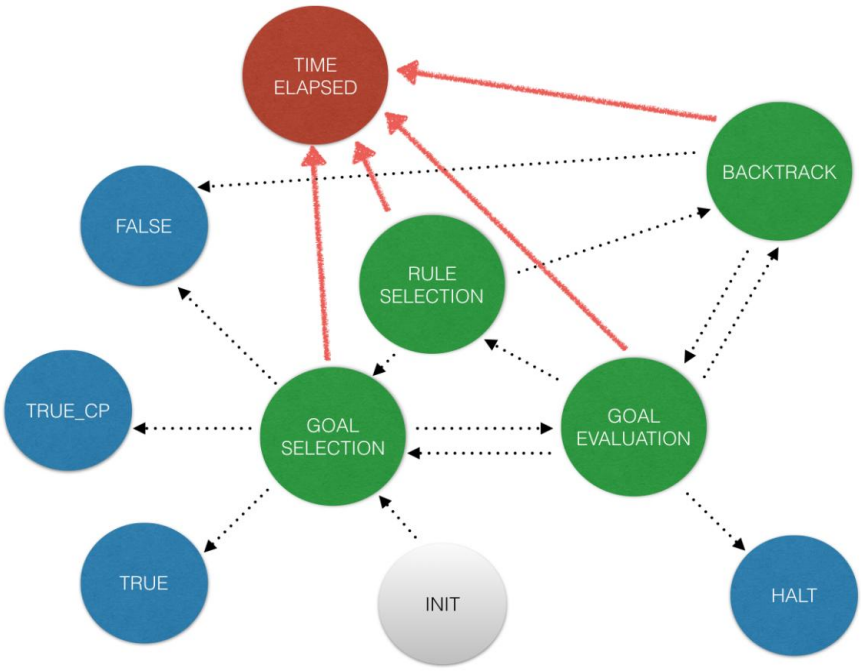

figura 26 - automa a stati finiti per la soluzione di una query con vincoli di tempo.

6.3.2.3 Implementazione

L'oggetto *façade* di classe *alice.tuprolog.Prolog* consente di accedere ai sottosistemi del core di tuProlog. *Prolog* possiede un riferimento ad un oggetto di classe *alice.tuprolog.EngineManager* che gestisce il sottosistema che implementa l'algoritmo di dimostrazione di una query Prolog. *EngineManager* riferisce un oggetto di classe *alice.tuprolog.EngineRunner* che gestisce la dimostrazione di un goal. Per supportare operazioni di *solve* e *solveNext* soggette a vincoli di tempo si sono aggiunti tre metodi alla classe *Prolog* (elencati sotto) i quali invocano gli omonimi metodi delle classi *EngineManager* ed *EngineRunner* (*figura 27*).

public SolveInfo solve(String str, long maxTime) throws MalformedGoalException

Risolve la query Prolog indicata come stringa in al più *maxTime* millisecondi. Se la query passata come stringa contiene errori di sintassi lancia l'eccezione *MalformedGoalException*.

public SolveInfo solve(Term term, long maxTime)

Risolve la query Prolog passata come oggetto di classe *alice.tuprolog.Term* in al più *maxTime* millisecondi.

public SolveInfo solveNext(long maxTime) throws NoMoreSolutionException

Dà come risultato l'ulteriore soluzione della query appena risolta in al più *maxTime* millisecondi. Se non ci sono altre soluzioni disponibili lancia l'eccezione *NoMoreSolutionException*.

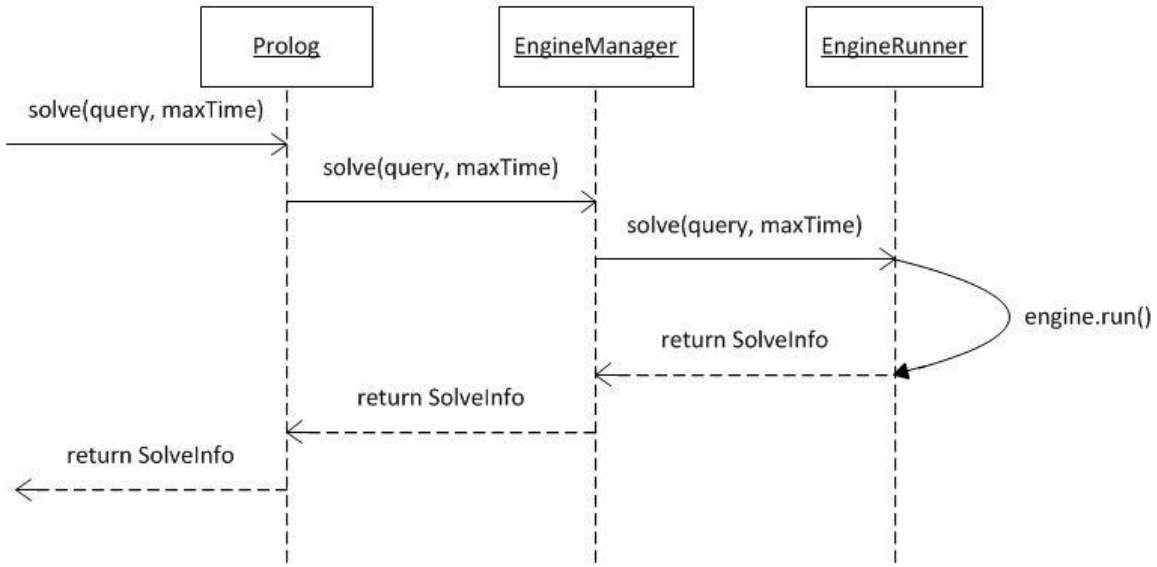

figura 27 – invocazioni metodo solve con vincoli temporali.

In *figura 27* è presentato il diagramma delle chiamate per l'esecuzione di una operazione di solve con vincoli temporali. Un cliente invoca il metodo *solve* dell'oggetto *Prolog* passando la query da risolvere (come stringa o come oggetto di classe *alice.tuprolog.Term*) e la durata massima della computazione in millisecondi. L'oggetto *Prolog* chiama l'omonimo metodo dell'oggetto *EngineManager* che a sua volta delega l'esecuzione ad un oggetto di classe *EngineRunner*. Quest'ultimo ha il ruolo di *esecutore* e si occupa di dimostrare la query. Ad ogni operazione di solve, *EngineRunner* crea un nuovo oggetto di classe *alice.tuprolog.Engine*, che implementa l'automa a stati finiti, e imposta il vincolo temporale. La chiamata al metodo *run* dell'oggetto *Engine* attiva il processo di risoluzione della query. Al termine della dimostrazione, il risultato di tipo *alice.tuprolog.SolveInfo* è restituito all'oggetto *Prolog* che lo rende disponibile al chiamante.

L'invocazione del metodo *solveNext* dell'oggetto *Prolog* è gestita analogamente a quella del metodo *solve* appena discusso.

Il metodo *run* dell'oggetto *Engine* può potenzialmente richiedere un tempo di esecuzione infinito nel caso in cui si formi un loop tra gli stati *GOAL SELECTION*, *GOAL EVALUATION*, *RULE SELECTION* e *BACKTRACK* dell'automa. Il controllo sul tempo impiegato nella dimostrazione della query avviene nel metodo *run* dell'oggetto *Engine*: se è terminato il tempo a disposizione e non è ancora terminata la dimostrazione, l'automa a stati finiti è forzato a transitare nello stato di *TIME ELAPSED* ed a concludere la dimostrazione con fallimento. In questo modo si evita la possibilità di avere cicli infiniti nella percorrenza degli stati dell'automa e si garantisce che la computazione termini al più nel tempo indicato.

```
StateEnd run() {
    String action;
    long startTime = System.currentTimeMillis();
    do {
        if(this.maxTime > 0 && checktime(startTime)){
            nextState = manager.END_TIMED_OUT;
            break;
        }
        if (mustStop) {
            nextState = manager.END_FALSE;
            break;
        }
        action = nextState.toString();
        nextState.doJob(this);
        manager.spy(action, this);


    } while (!(nextState instanceof StateEnd));
    nextState.doJob(this);

    return (StateEnd)(nextState);
}
```

figura 28 - metodo run della classe alice.tuprolog.Engine.

In *figura 28* è riportata l'implementazione del metodo *run* della classe *alice.tuprolog.Engine*. Si può notare come lo stato dell'automa (*nextState*) sia forzato a *TIME ELAPSED* (costante *manager.END_TIMED_OUT*) nel caso in cui il tempo a disposizione per la dimostrazione sia scaduto.

6.3.3 Serializzazione dello stato e migrazione del motore

6.3.3.1 Analisi del problema e requisiti

In un sistema distribuito ad agenti può essere necessario far migrare un agente da un nodo computazionale ad un altro. Tipicamente tale operazione è facilitata dalla piattaforma ad agenti che automatizza il processo. Per fare in modo che anche il motore inferenziale sia una risorsa automaticamente rilocabile, occorre poterlo serializzare e riattivare senza perdere informazioni sul suo stato interno. Si rende quindi necessario individuare lo stato del motore e quali strutture dati ne fanno parte.

Si assume che sul nodo sorgente e sul nodo destinatario della migrazione ci sia la stessa versione di tuProlog con lo stesso set di librerie disponibili. Si suppone anche che ogni altra risorsa associata al motore, ma esterna ad esso (ad esempio: file, socket, oggetti java in *OOLibrary*), sia rilasciata dal programmatore prima della serializzazione e riacquisita dopo la migrazione. Inoltre si assume che la serializzazione del motore avvenga quando questo non sta computando.

6.3.3.2 Progettazione

Ispirandosi alla struttura modulare del core del motore, si è deciso di progettare un sottosistema dedicato alla serializzazione dello stato. In questo modo si aggiungono nuove funzionalità al core senza minarne l'architettura.

Si sono individuati due tipi di stato del motore: uno ridotto, che rappresenta l'esito di esecuzione di una query, ed uno completo (estensione del primo), che racchiude tutte le strutture dati che caratterizzano un'istanza del motore. Il primo tipo di stato è utilizzabile per inviare l'esito di una query ad un cliente remoto, il secondo è impiegato per la migrazione del motore. In questo secondo caso, per poter serializzare e riattivare il motore senza perderne lo stato, occorre analizzare ogni suo sottosistema e serializzare tutte le informazioni utili a runtime.

Library manager: contiene l'elenco delle librerie di predicati caricate nel motore. È sufficiente salvare il loro nome in quanto si suppone di disporre dello stesso insieme di librerie alla riattivazione del motore.

Theory manager: si devono serializzare le teorie dinamiche del motore. Non è necessario serializzare anche quelle statiche in quanto si è memorizzato il nome delle librerie che le contengono.

Primitive manager: questo sottosistema non contiene informazioni da serializzare in quanto i dati sulle primitive di sistema vengono rigenerati automaticamente quando si riattiva il motore.

Flag manager: è necessario serializzare tutti i flag definiti nel motore.

Operator manager: è necessario serializzare l'insieme degli operatori definiti nel motore.

Engine manager: si deve tenere traccia dell'ultima query Prolog eseguita in quanto la serializzazione può avvenire a metà di una dimostrazione che ammette soluzioni multiple. Oltre all'ultima query bisogna anche tenere traccia di quante soluzioni sono state esplorate e se ci sono ancora dei punti di scelta aperti nella dimostrazione. Queste informazioni sono utilizzate alla riattivazione del motore per poter ricomputare l'ultimo goal ed eventualmente restituire le soluzioni non ancora esplorate.

Si memorizza anche il timestamp dell'instante in cui avviene la serializzazione: questa informazione può essere utilizzata per distinguere la freschezza dei dati nel caso di serializzazioni multiple.

La *figura 29* schematizza il processo di serializzazione del motore: tramite un parametro booleano è possibile scegliere la tipologia di stato da serializzare, ridotto o completo. Dopo la scelta, il sottosistema di serializzazione si occupa di produrre la stringa JSON che rappresenta lo stato del motore.

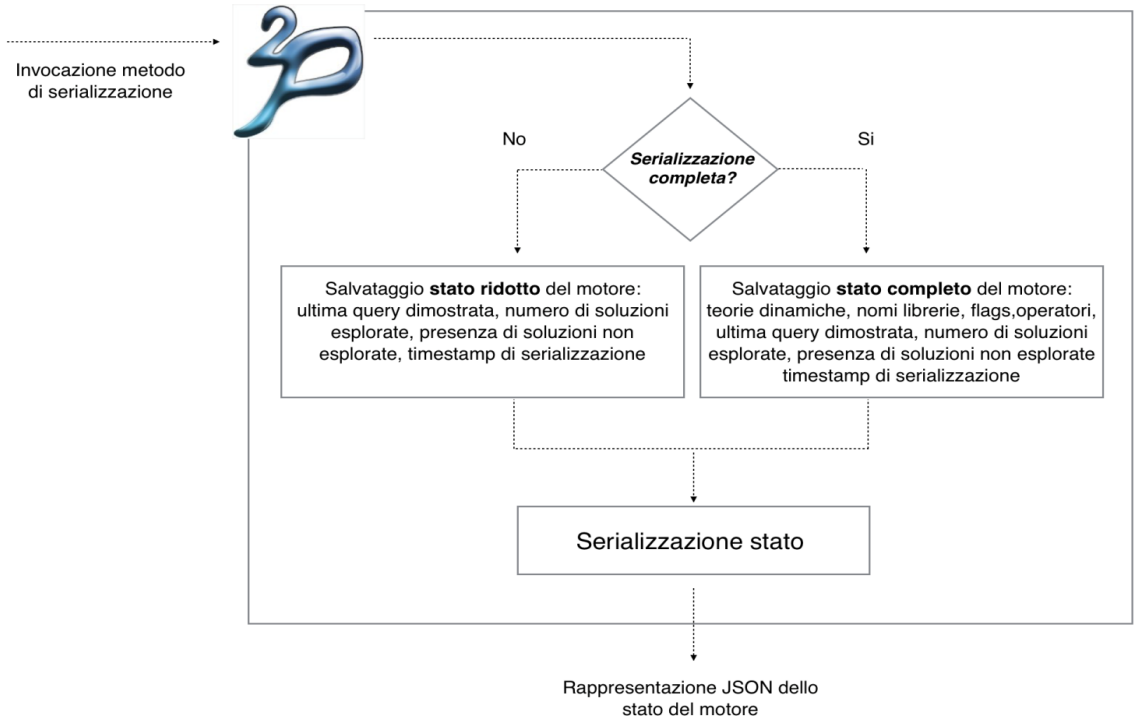

figura 29 – serializzazione stato del motore inferenziale.

Si è scelto di serializzare lo stato in formato JSON in quanto è un formato standard, *human-readable* e compatibile con il Web[JSON].

La *figura 30* schematizza la riattivazione del motore: il metodo di ripristino verifica se lo stato ricevuto in formato JSON è uno stato completo ed in caso affermativo procede alla de-serializzazione e alla riattivazione del motore Prolog. Al contrario, il metodo ritorna un puntatore nullo.

L'operazione di serializzazione, migrazione e riattivazione è un'operazione costosa in termini di risorse computazionali in quanto query con soluzioni multiple vengono sempre rieseguite dall'inizio quando si ricrea il motore.

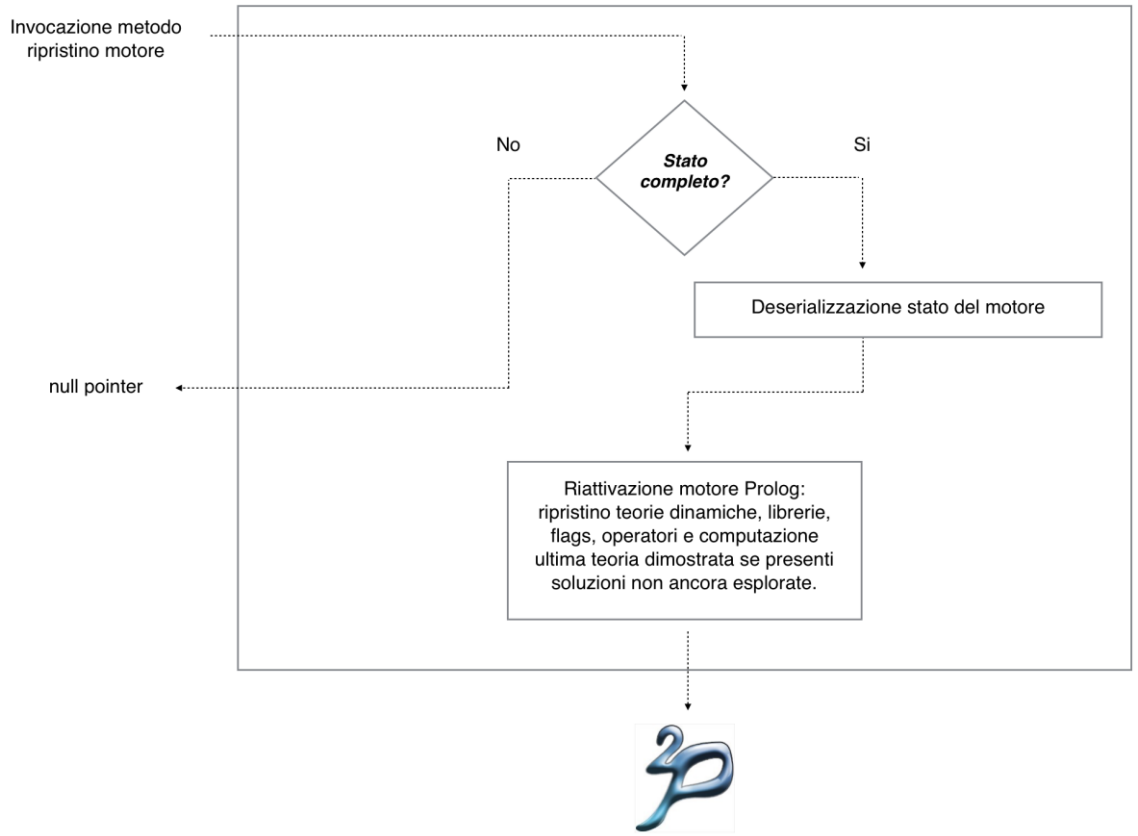

figura 30 – riattivazione del motore dopo la migrazione.

6.3.3.3 Implementazione

Nel package *alice.tuprolog.json* le classi *JSONSerializerManager* e *JSONMarshaller* implementano il sottosistema di serializzazione. Entrambe le classi si basano su GSON, una libreria open-source di Google, che automatizza la serializzazione/de-serializzazione di oggetti Java[GSON]. *JSONSerializerManager* gestisce un'istanza singleton di GSON mentre la classe *JSONMarshaller* ha il compito di configurare GSON per serializzare le strutture dati che compongono lo stato di tuProlog.

Per semplificare lo scambio di informazioni sullo stato, il sottosistema utilizza una gerarchia di oggetti Java che secondo il pattern DTO, incapsula lo stato del motore e ne facilita la serializzazione.

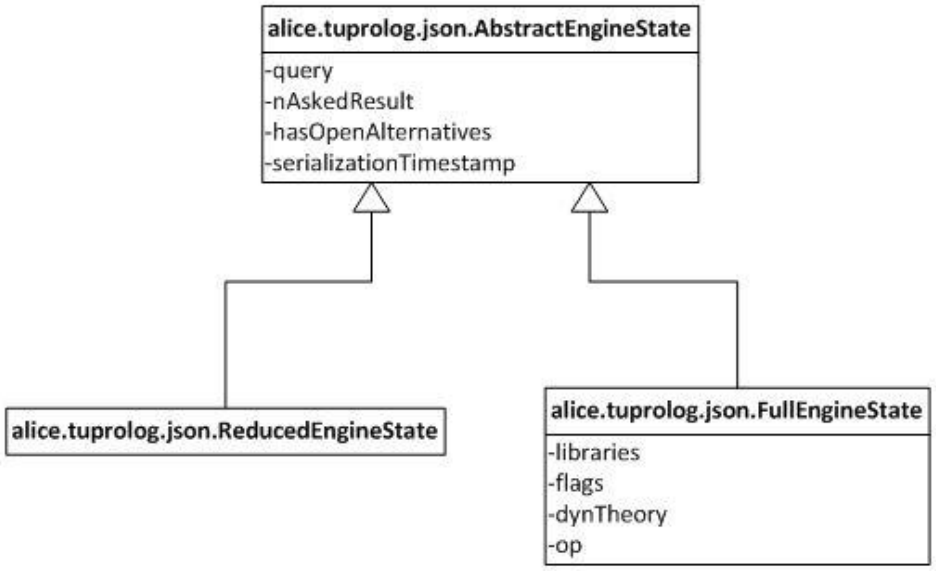

figura 31 - gerarchia di classi DTO per la serializzazione dello stato del motore.

La gerarchia, mostrata in *figura 31*, è composta da una classe astratta, *AbstractEngineState*, e da due classi concrete, *ReducedEngineState* e *FullEngineState*. *AbstractEngineState* fattorizza le informazioni minime sullo stato del motore composte da: l'ultima query richiesta (*query*), il numero di risultati esplorati (*nAskedResult*), la presenza di punti di scelta non esplorati nel processo di dimostrazione della query (*hasOpenAlternatives*) ed il timestamp della serializzazione (*serializationTimestamp*). La classe *ReducedEngineState* si limita a concretizzare *AbstractEngineState* e rappresenta lo stato ridotto del motore. La classe concreta *FullEngineState* aggiunge ad *AbstractEngineState* i nomi delle librerie caricate nel motore (*libraries*), i flag Prolog (*flags*), le teorie dinamiche (*dynTheory*) e

gli operatori Prolog (*op*). *FullEngineState* rappresenta lo stato completo del motore ed è l'oggetto DTO che viene serializzato e trasferito quando si migra l'engine.

La serializzazione del motore avviene invocando il metodo *toJSON(boolean fullState)* della classe *alice.tuprolog.Prolog*. Questo metodo si occupa di reperire le informazioni sullo stato dai sottosistemi di tuProlog e di serializzarle. Come mostrato in *figura 32*, se il parametro passato ha valore *true* il metodo utilizza il DTO che rappresenta lo stato completo del motore (*FullEngineState*), altrimenti utilizza il DTO per la rappresentazione ridotta dello stato (*ReducedEngineState*). In questo modo si ha la possibilità di scegliere tra una serializzazione completa, per la migrazione dell'engine, ed una serializzazione ridotta, per avere informazioni sull'esito dell'ultima query eseguita. Il metodo termina restituendo la rappresentazione JSON del DTO scelto.

```
public String toJSON(boolean fullState){
    AbstractEngineState brain = null;
    if(fullState)
    {
        brain = new FullEngineState();
        this.theoryManager.serializeLibraries((FullEngineState)brain);
        this.theoryManager.serializeDynDataBase((FullEngineState)brain);
        ((FullEngineState)brain).setOp((LinkedList<Operator>)this.opManager.getOperators());
    }
    else
        brain = new ReducedEngineState();

    this.theoryManager.serializeTimestamp(brain);
    this.engineManager.serializeQueryState(brain);
    this.flagManager.serializeFlags(brain);

    return JSONSerializerManager.toJSON(brain);
}
```

figura 32 - metodo toJSON.

Il metodo statico *fromJSON(String jsonString)* della classe *alice.tuprolog.Prolog* crea un motore Prolog a partire dalla stringa JSON passata come parametro. L'implementazione del metodo (*figura 33)*, si articola in tre fasi: inizialmente il metodo testa se la stringa JSON contiene la stringa *"FullEngineState"* ed in caso affermativo crea un oggetto Java DTO di classe *FullEngineState* che contiene lo stato da caricare nella nuova istanza del motore. In caso negativo, il metodo restituisce *null* in quanto la stringa passata non contiene le informazioni necessarie per riattivare il motore. Se *fromJSON* ha un'istanza di stato completo, crea un nuovo oggetto *Prolog* e gli carica lo stato ottenuto dall'oggetto DTO. Qualora si verificasse un'eccezione, il metodo restituisce *null*. Infine *fromJSON* controlla se la query serializzata ha ancora punti di scelta aperti nella sua dimostrazione: in caso negativo il metodo termina restituendo l'istanza del motore; in caso affermativo, ricalcola la query (che

necessariamente deve essere idempotente) e poi restituisce l'istanza del motore. In questo modo alla riattivazione del motore è possibile esplorare soluzioni non ancora dimostrate.

```
public static Prolog fromJSON(String jsonString) {
    AbstractEngineState brain = null;
    Prolog p = null;
    if(jsonString.contains("FullEngineState")){
        brain = JSONSerializerManager.fromJSON(jsonString, FullEngineState.class);
    }
    else
        return p;
    try {
        p = new Prolog(((FullEngineState) brain).getLibraries());
        p.setTheory(new Theory(((FullEngineState) brain).getDynTheory()));
        p.opManager = new OperatorManager();
        LinkedList<Operator> l = ((FullEngineState) brain).getOp();
        for(Operator o : l)
            p.opManager.opNew(o.name, o.type, o.prio);
        } catch (InvalidLibraryException e) {
            e.printStackTrace();
            return null;
        } catch (InvalidTheoryException e) {
            e.printStackTrace();
            return null;
        }
        p.flagManager.reloadFlags((FullEngineState) brain);
        int i = 0;
        int n = brain.getNumberAskedResults();
        if(brain.hasOpenAlternatives()){
            p.solve(brain.getQuery());
            while(i<n){
                try {
                    p.solveNext();
                } catch (NoMoreSolutionException e) {}
                i++;
            }
        }
        return p;
    }
```

figura 33 - metodo fromJSON.

Per una maggiore flessibilità del motore si sono aggiunti metodi analoghi a quelli presentati alle classi *Theory*, *Term* e *SolveInfo* del package *alice.tuprolog.* In ciascuna di queste classi il metodo *public String toJSON()* restituisce una stringa JSON che rappresenta l'oggetto su cui è invocato. Il metodo *public static <Object type> fromJSON(String jsonString)* è il duale del precedente e restituisce un oggetto di tipo *<Object type>* a partire dalla sua rappresentazione JSON. Il valore *<Object type>* dipende dalla classe in cui è definito il metodo ed è un tipo compreso tra *Theory*, *Term* e *SolveInfo.*

6.3.4 Monitoraggio, controllo e gestione del core del motore

6.3.4.1 Analisi del problema e requisiti

L'utilizzo di motori inferenziali su nodi distribuiti in rete fa nascere la necessità di poterli controllare da remoto a runtime. L'amministratore deve poter accedere sia al centro di comando del sistema ad agenti, per gestirne la creazione e l'attivazione, sia al core dei motori Prolog presenti sui vari nodi. Una gestione ottimale del motore Prolog è importante in quanto è l'engine che amministra l'intelligenza ed il comportamento del nodo su cui esegue.

Utilizzando tuProlog come motore inferenziale, le operazioni di monitoraggio, controllo e gestione si traducono nella necessità di gestire da remoto ciascun sottosistema del core. Consentendo l'accesso da remoto ai sottosistemi del motore si permette all'amministratore di sistema di monitorarne lo stato e di modificarlo agendo sulla base di conoscenza dell'engine. Permettere l'accesso da remoto al core di tuProlog introduce anche problemi legati alla sicurezza: soltanto l'amministratore di sistema deve poter accedere al core ed in caso di scambio di dati sensibili, è necessario che la connessione tra client di management e nodo remoto sia cifrata. Un approfondimento sulla sicurezza è trattato al paragrafo 6.3.4.5.

Per l'implementazione del sistema di management di tuProlog si è scelto di non realizzare una soluzione custom, ma di adottare la tecnologia middleware *Java Management Extensions* (JMX), di riferimento per il mondo Java, che offre gli strumenti per arricchire codice Java con metodi di management invocabili da remoto[JMX]. Si è scelto JMX in quanto standardizza i meccanismi di management ed accelera il processo di sviluppo del sistema. Inoltre JMX risolve gran parte delle problematiche che si sarebbero dovute affrontare nella implementazione di una soluzione ad hoc come ad esempio l'interazione con i componenti monitorati e la remotizzazione delle chiamate di metodi di management.

Nei paragrafi successivi si presenta la tecnologia JMX e si descrivono i metodi aggiunti ai sottosistemi del core di tuProlog al fine di supportare operazioni di management.

6.3.4.2 JMX: presentazione ed analisi della tecnologia

JMX è un insieme di specifiche standard che permettono di inserire all'interno di una applicazione Java dei componenti per il monitoraggio della stessa. L'architettura di JMX si suddivide in tre livelli: *Instrumentation*, *Agent* e *Distributed Services level*.

1. Instrumentation level

Comprende le risorse da monitorare rappresentate da oggetti Java *MBean* (*Managed Bean*). Gli MBean sono normali oggetti Java che, oltre ai metodi di business logic, forniscono metodi di management logic. I metodi di management vengono dichiarati dal programmatore in un'opportuna interfaccia di nome *<nome-bean>MXBean*. Esistono quattro tipologie di MBean che si distinguono in base al tipo di interfaccia di management che implementano.

***Standard MBean*:** implementano un'interfaccia di management statica (è un'interfaccia Java standard) che non cambia nel tempo.

***Dynamic MBean*:** forniscono un set di metadati che descrivono un'interfaccia di management dinamica. Questa interfaccia può cambiare nel tempo.

***Open MBean*:** come *Standard MBean* ma i tipi di ritorno e dei parametri dei metodi di management appartengono ad un set limitato di tipi primitivi. Sono usati per motivi di compatibilità con sistemi non Java based.

***Model MBean*:** come *Dynamic MBean* ma offrono un set di metadati più ampio. Sono adatti a descrivere risorse complesse e dinamiche.

Soltanto i metodi dichiarati nell'interfaccia di management di un MBean sono invocabili da remoto.

2. Agent level

Il livello di agente è composto da un *MBeanServer*, ovvero da un server che ha il compito di gestire gli MBean e di fare da intermediario con le applicazioni di management. Quest'ultime non interagiscono direttamente con gli MBean ma chiedono al server di eseguire operazioni sugli MBean. Come mostrato in *figura 34*, l'applicazione client chiede al server di invocare un metodo su di un MBean (*invoke*) indicandone il nome, il nome e la firma del metodo da eseguire ed eventuali parametri. Per identificare gli MBean in gestione, *MBeanServer* mantiene una tabella (*MBean Repository*) in cui a ciascun nome logico di MBean fa corrispondere il

riferimento all'oggetto Java che lo implementa. Dopo aver identificato l'oggetto MBean, il server invoca su questo il metodo richiesto usando la reflection. Inoltre va sottolineato che *MBeanServer* non gestisce gli MBean in maniera thread-safe. Eventuali accessi concorrenti alla stessa risorsa devono essere gestiti dal programmatore.

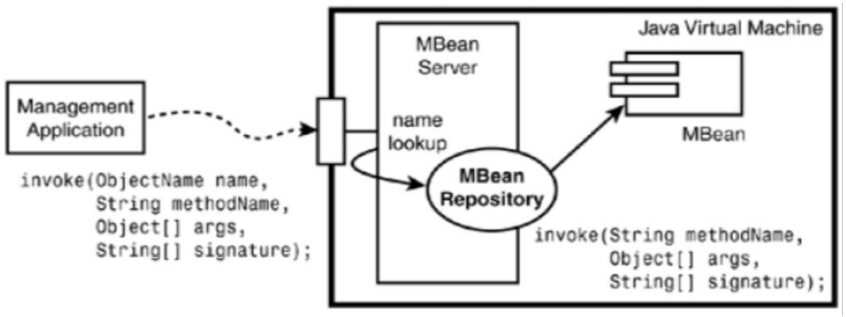

figura 34 – invocazione remota di un metodo di un MBean.

Instrumentation level ed *Agent level* sono locali alla stessa Java Virtual Machine: questa scelta semplifica l'implementazione del server in quanto quest'ultimo può interagire direttamente con gli oggetti managed. Da ciò segue che in un sistema distribuito che usa JMX come tecnologia di management si ha almeno un *MBeanServer* su ogni nodo. È compito dell'applicazione di management decidere quale *MBeanServer* interrogare per ottenere le informazioni di management desiderate.

3. Distributed Services level

Questo livello si occupa di rendere il server JMX accessibile a clienti remoti e comprende due tipologie di componenti: i *connector* ed i *protocol adapter*. Per connector JMX si intende il supporto software JMX lato cliente e servitore che rende trasparente la comunicazione di rete. I connector vengono utilizzati quando si sviluppano applicazioni di management ad hoc. I protocol adapter JMX invece sono supporti software che permettono di interagire con *MBeanServer* attraverso protocolli applicativi general purpose, come ad esempio HTTP. Generalmente si utilizzano i protocol adapter quando le operazioni di management sono invocate da browser web. La *figura 35* dà una visione di insieme dell'architettura JMX.

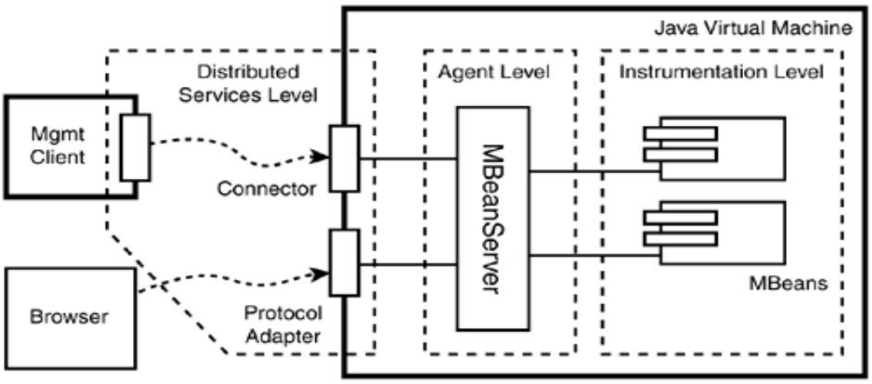

figura 35 - architettura JMX.

6.3.4.3 Progetto

L'utilizzo di JMX per il management dei sottosistemi del core di tuProlog impone che ciascun sottosistema sia trasformato in un MBean. Tale operazione, grazie alla malleabilità del core di tuProlog, risulta essere poco invasiva: ciascuna classe che gestisce un sottosistema del core deve implementare un'opportuna interfaccia di management conforme alle specifiche JMX. Si è scelto di utilizzare soltanto *Standard MBean* in quanto si ritiene che gli attuali sottosistemi del core di tuProlog non siano soggetti a significativi cambiamenti nel tempo.

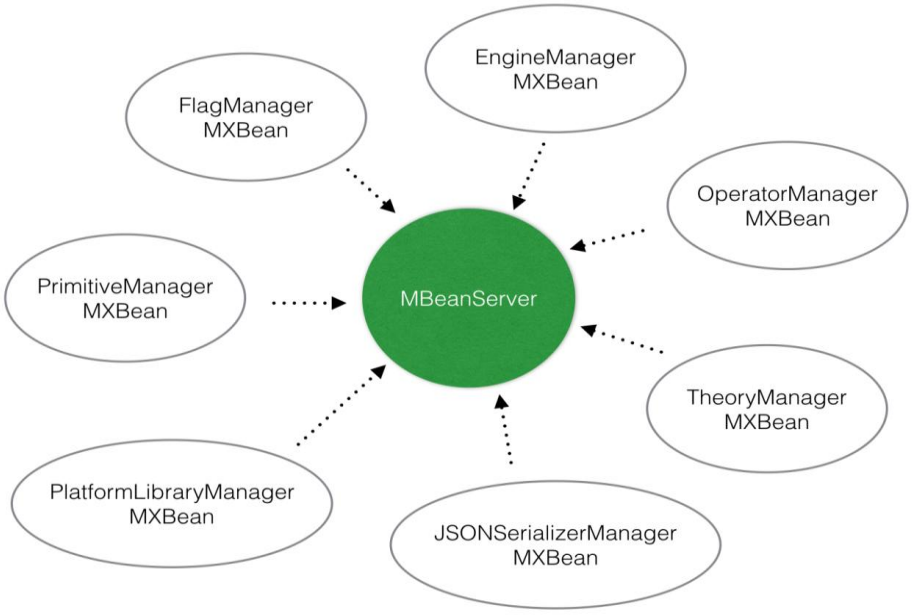

figura 36 – sistema di management di tuProlog.

La *figura 36* mostra la struttura del sistema di management di tuProlog: ogni istanza del motore, se opportunamente configurata dal programmatore, possiede un

MBeanServer che gestisce ciascun componente del core come standard MBean. *MBeanServer* esegue in un thread dedicato e compete per l'utilizzo delle risorse del core: è compito del programmatore evitare situazioni di corse critiche e garantire che tutti i metodi di management accedano in modo mutuamente esclusivo alle strutture dati che mantengono lo stato del motore.

Si è deciso di utilizzare il protocol adapter HTTP/HTTPS per amministrare tuProlog direttamente da browser web. Questa scelta semplifica il client di management in quanto non si ha la necessità di svilupparlo ad hoc, ma è possibile utilizzare un browser qualsiasi. L'adapter aggiunge a *MBeanServer* un server HTTP che riceve dal web le richieste di invocazione di metodo e le inoltra a *MBeanServer*.

L'URL per l'invocazione di metodo ha la seguente struttura:

```
http[s]://host:port/invoke?objectname=X&operation=Y&type0=Z&value0=W…
```

dove *host* e *port* indicano l'indirizzo IP e la porta del server HTTP dell'adapter, *objectname* identifica l'oggetto MBean su cui si vuole invocare l'operazione specificata nel parametro *operation.* I parametri da passare al metodo da invocare sono inclusi nell'URL: *typeN* indica il tipo dell'ennesimo parametro nella segnature del metodo, *valueN* il valore di tale parametro. Qualora non siano necessari parametri, l'URL termina dopo l'attributo *operation.*

La *figura 37* mostra l'architettura JMX adattata al management del core di tuProlog.

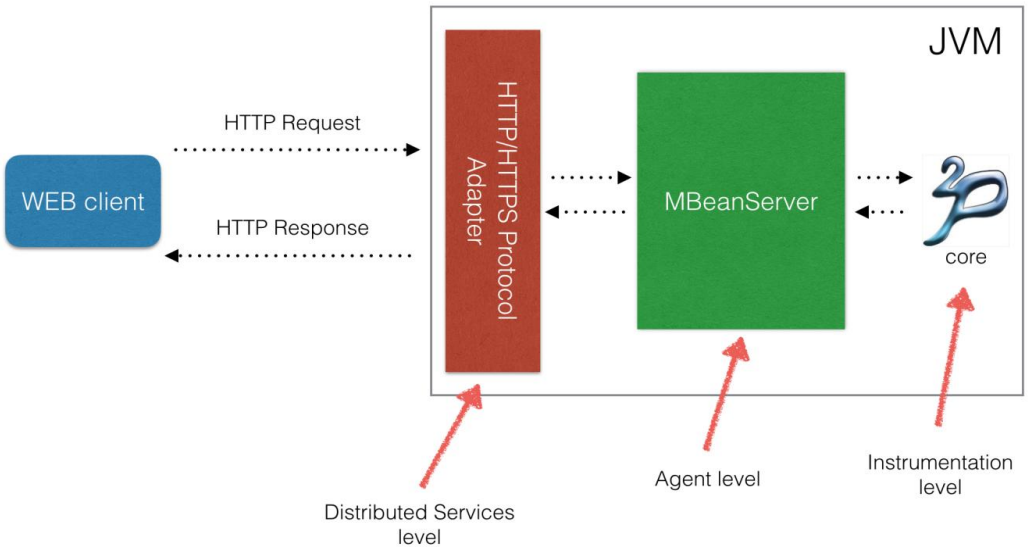

figura 37 – architettura JMX per il management di tuProlog.

6.3.4.4 Implementazione

Per l'implementazione del sistema di management si è impiegata la libreria MX4J in quanto è una distribuzione JMX open-source utilizzata da una cospicua comunità di sviluppatori[MX4J]. Questa libreria fornisce anche una implementazione del protocol adapter HTTP. Per quanto riguarda la sicurezza, l'adapter HTTP consente l'utilizzo di HTTPS e/o dell'autenticazione utente tramite username e password (paragrafo 6.3.4.5).

La classe *alice.tuprolog.management.PrologMXBeanServer* si occupa di istanziare e configurare un *MBeanServer*. Il metodo *startMXBeanServer* ha come parametri di ingresso: un'istanza di tuProlog, un indirizzo IP, un numero di porta, una costante che indica quale protocol adapter JMX attivare, il nome del file che contiene le credenziali per l'autenticazione HTTP ed il nome del file che contiene le configurazioni per abilitare SSL-TLS ed HTTPS. I parametri sulla configurazione della sicurezza sono opzionali: qualora valgano *null*, la sicurezza è disabilitata. *startMXBeanServer* si occupa di creare un'istanza di *MBeanServer*, registrarci i vari sottosistemi del core di tuProlog, ed attivare il protocol adapter all'indirizzo IP indicato. Attualmente si è scelto di supportare solamente il protocol adapter HTTP in quanto si vuole che le operazioni di management avvengano via browser web. Sempre per questo motivo, tutti i metodi di management che prevedono un risultato lo restituiscono al chiamante in formato JSON[JSON].

La *tabella 8* riporta le interfacce di management implementate dai sottosistemi del core di tuProlog. Tutte le interfacce elencate sono contenute nel package *alice.tuprolog.management.interfaces*.

Interfaccia	EngineManagerMXBean
Classe del core che la implementa	alice.tuprolog.EngineManager
Descrizione	I metodi definiti in questa interfaccia consentono di risolvere query per scopi di management. Tutti i metodi, in caso di errore, restituiscono una stringa che descrive l'errore avvenuto a runtime.
Metodi	
String solve_untimed(String goal)	Risolve senza vincoli di tempo il goal Prolog passato come stringa.
String solveNext_untimed()	Dimostra senza vincoli di tempo la soluzione successiva del goal appena risolto.
String solve_timed(String goal, long maxTime)	Risolve il goal Prolog passato come stringa in al più *maxTime* millisecondi.

String solveNext_timed(long maxTime)	Dimostra in al più *maxTime* millisecondi la soluzione successiva del goal appena risolto.
String solveN_untimed(String goal, int n)	Dimostra senza vincoli di tempo le prime n soluzioni del goal passato come stringa.
String solveN_timed(String goal, int n, long maxTime)	Dimostra in al più *n*maxTime* millisecondi le prime n soluzioni del goal passato come stringa.
String solveAll_timed(String goal, long maxTime)	Dimostra tutte le soluzioni del goal passato come stringa. Il metodo esegue per al più *tot*maxTime* millisecondi con *tot* numero non noto a priori di soluzioni del goal.
String solveAll_untimed(String goal)	Dimostra senza vincoli di tempo tutte le soluzioni del goal passato come stringa.
Interfaccia	FlagManagerMXBean
Classe del core che la implementa	alice.tuprolog.FlagManager
Descrizione	L'interfaccia definisce metodi per il management dei flag Prolog.
Metodi	
void reset()	Cancella tutti i flag.
boolean occursCheckIsEnabled()	Controlla se il flag *occursCheck* ha valore *on*: in caso affermativo restituisce *true*, altrimenti *false*.
boolean modifiable(String name)	Testa se il flag di nome *name* può essere modificato dall'utente.
boolean configurePrologFlag (String name, String term)	Assegna il termine rappresentato dalla stringa term al flag identificato da *name*.
boolean validValue(String name, String term)	Testa se il termine rappresentato dalla stringa *term* è un valore ammissibile per il flag identificato da *name*.
String fetchAllPrologFlags()	Restituisce in formato JSON l'elenco di tutti i flag Prolog del motore.
String fetchPrologFlag(String name)	Restituisce in formato JSON il flag identificato da *name*.

Interfaccia	JSONSerializerManagerMXBean
Classe del core che la implementa	alice.tuprolog.json.JSONSerializerManager
Descrizione	L'interfaccia definisce metodi per il management del sistema di serializzazione dello stato del motore.
Metodi	
void reset()	Resetta il sistema di serializzazione.
String fetchCurrentAdapters()	Restituisce una stringa JSON contenente il nome delle classi di sistema che necessitano di un adattatore GSON per la serializzazione.
boolean addAdapter(String className)	Aggiunge un adattatore GSON per la serializzazione della classe identificata dal parametro *className*.
Interfaccia	OperatorManagerMXBean
Classe del core che la implementa	alice.tuprolog.OperatorManager
Descrizione	L'interfaccia definisce metodi per la gestione degli operatori utilizzati nelle teorie Prolog.
Metodi	
void reset()	Cancella tutti gli operatori definiti nel motore.
String fetchAllOperators()	Restituisce l'elenco in formato JSON di tutti gli operatori definiti nel motore.
void opNew(String name, String type, int prio)	Crea un nuovo operatore di nome *name*, tipo *type* e priorità *prio*.
int opPrio(String name, String type)	Restituisce la priorità dell'operatore di nome *name* e di tipo *type.*
int opNext(int prio)	Restituisce la priorità più bassa e più vicina alla priorità indicata dal parametro *prio*.
Interfaccia	PlatformLibraryManagerMXBean
Classe del core che la implementa	alice.tuprolog.AbstractPlatformLibraryManager
Descrizione	L'interfaccia definisce metodi per la gestione delle librerie di predicati che possono essere caricate dinamicamente nel motore.
Metodi	
String fetchCurrentPlatform()	Restituisce l'identificativo della piattaforma corrente (Java, Android o .NET).
String fetchCurrentLibraries()	Restituisce una lista in formato JSON contenente il nome delle librerie caricate nel motore.

String fetchCurrentExternalLibraries()	Restituisce una lista in formato JSON contenente il nome delle librerie esterne caricate nel motore.
boolean loadLibraryIntoEngine(String libraryClass)	Carica nel motore la libreria di predicati identificata dal parametro *libraryClass.*
boolean unloadLibraryFromEngine(String libraryClass)	Rimuove dal motore la libreria di predicati identificata dal parametro *libraryClass.*
Interfaccia	PrimitiveManagerMXBean
Classe del core che la implementa	alice.tuprolog.PrimitiveManager
Descrizione	L'interfaccia definisce metodi per ottenere informazioni riguardo le primitive definite nel motore Prolog.
Metodi	
String fetchDirectiveInfo(String directive)	Restituisce una stringa JSON che descrive la direttiva identificata dal parametro directive.
String fetchFunctorInfo(String functor)	Restituisce una stringa JSON che descrive il funtore Prolog identificato dal parametro functor.
String fetchPredicateInfo(String predicate)	Restituisce una stringa JSON che descrive il predicato Prolog identificato dal parametro predicate.
Interfaccia	TheoryManagerMXBean
Classe del core che la implementa	alice.tuprolog.TheoryManager
Descrizione	L'interfaccia ha lo scopo di offrire metodi di management per la gestione della base di conoscenza del motore.
Metodi	
String fetchKnowledgeBase(boolean all)	Restituisce in formato JSON la base di conoscenza del motore. Se il parametro *all* vale false il metodo restituisce soltanto la base di conoscenza statica, altrimenti include nel risultato anche le teorie dinamiche.
String fetchMostRecentConsultedTheory()	Restituisce l'ultima teoria consultata.
void clearKnowledgeBase()	Cancella la base di conoscenza del motore.

void consultTheory(String theory, boolean dyn, String lib) throws InvalidTheoryException	Aggiunge la teoria *theory* alla base di conoscenza del motore. Il parametro *dyn* indica se la teoria è dinamica o meno ed il parametro *lib* indica quale libreria definisce la teoria. Qualora la teoria contenesse errori di sintassi, il metodo lancia l'eccezione *InvalidTheoryException*.
void assertA(String clause, boolean dyn, String lib, boolean backtrackable)	Aggiunge la clausola *clause* in testa al gruppo di clausole della stessa tipologia. Il parametro *dyn* indica se la clausola è dinamica o meno, il parametro *lib* indica quale libreria definisce la clausola ed il parametro *backtrackable* indica se è possibile effettuare backtrack sulla clausola specificata.
void assertZ(String clause, boolean dyn, String lib, boolean backtrackable)	Aggiunge la clausola *clause* in coda al gruppo di clausole della stessa tipologia. Il parametro *dyn* indica se la clausola è dinamica o meno, il parametro *lib* indica quale libreria definisce la clausola ed il parametro *backtrackable* indica se è possibile effettuare backtrack sulla clausola specificata.
void retract(String clause)	Esegue l'operazione di retract per la clausola identificata dal parametro clause.
void removeLibraryTheory(String libName)	Rimuove la teoria contenuta nella libreria di nome *libName*.

tabella 8 – interfacce di management del core di tuProlog.

6.3.4.5 Note sulla sicurezza e sul multithreading

6.3.4.5.1 Sicurezza protocol adapter HTTP

Il protocol adapter HTTP fornito da MX4J offre due livelli di sicurezza: uno prevede soltanto l'autenticazione dell'utente via HTTP, l'altro prevede l'autenticazione HTTP e l'utilizzo di SSL-TLS per una comunicazione sicura. È possibile anche non configurare l'autenticazione dell'utente ed utilizzare il sistema in chiaro o con SSL-TLS: si sono trascurate quest'ultime opzioni in quanto si devono autenticare gli amministratori di sistema per verificare se posseggono i privilegi per gestire da remoto l'engine Prolog.

L'autenticazione semplice via HTTP ha il difetto di codificare lo username e la password in formato *Base64*: questa codifica è facilmente invertibile da parte di malintenzionati che intercettano la comunicazione. Inoltre questo meccanismo si occupa solo dell'autenticazione dell'utente, non protegge la conversazione dell'utente con il server. L'utilizzo del protocollo SSL-TLS, associato all'autenticazione, rende privata la comunicazione tra client e server: in questo modo operazioni sensibili di management non possono essere spiate e/o alterate da attaccanti malintenzionati.

6.3.4.5.2 Management e multithreading

Per quanto riguarda il multithreading va detto che il motore Prolog non lo gestisce automaticamente: ciò potrebbe causare inconsistenze se si attuano operazioni di management mentre il motore esegue in un thread diverso da quello di *MBeanServer*. È compito dello sviluppatore prevedere uno stato di *manutenzione* del sistema nel quale le operazioni di monitoraggio, controllo e gestione possono avvenire senza causare corse critiche per l'accesso ai sottosistemi del motore inferenziale.

7. Caso di studio: Smart Kitchen

7.1 Lo scenario

Il caso di studio prevede di sviluppare una applicazione per la gestione intelligente di una *Smart Kitchen*[SK]. Con il termine *Smart Kitchen* si intende una cucina di nuova generazione in cui i componenti sono dispositivi IoT. Si suppone che la cucina sia dotata almeno di un frigorifero, un frullatore, un forno e una dispensa; ciascuno di questi elementi integra al suo interno un calcolatore a capacità limitate come ad esempio un *Raspberry Pi 2*[RASP]. Inoltre si suppone che i dispositivi IoT siano dotati di opportuni sensori che permettano di rilevare sia le abitudini alimentari degli utenti sia lo stato dei dispositivi. Ad esempio il frigorifero e la dispensa sono in grado di rilevare quali beni sono in esaurimento mentre il frullatore ed il forno sono in grado di tracciare quali alimenti sono stati consumati dagli utenti. I dispositivi non sono solo aggregatori di dati ma sono anche entità attive in grado di interagire con gli altri elementi smart. Nella cucina è presente anche un dispositivo di controllo (mobile o fisso) che deve essere in grado di comunicare con i dispositivi IoT e fare inferenze sulle informazioni che questi forniscono.

7.2 Analisi e requisiti

Lo scenario presentato al paragrafo precedente richiede che ciascun nodo IoT non fornisca semplicemente dati grezzi, ma che sia un dispositivo intelligente, in grado di produrre dati di alto livello e capace di coordinarsi e collaborare con gli altri elementi presenti nel sistema.

La presenza di un dispositivo di controllo non vincola a realizzare il sistema in modo centralizzato, ma implica soltanto che vi sia la possibilità di sviluppare software che si comporti come un cliente nei confronti dei dispositivi IoT.

Si sono identificati i seguenti requisiti:

- ciascun nodo deve possedere una base di conoscenza locale che mantenga informazioni di alto livello sul suo stato;
- ciascun nodo deve essere a conoscenza dell'ambiente che lo circonda (*situatedness*);
- i dispositivi IoT devono poter comunicare con il dispositivo di controllo per condividere informazioni sullo stato della cucina;
- la comunicazione tra i dispositivi deve essere sicura;

- deve essere possibile migrare il software in esecuzione su un dispositivo qualora quest'ultimo necessiti di essere sostituito;
- deve essere possibile aggiungere/rimuovere dispositivi IoT al sistema senza doverlo fermare.

7.3 Progetto

Per quanto riguarda la progettazione del prototipo del sistema, si è scelto di utilizzare un'unica piattaforma JADE in quanto si suppone che tutti i dispositivi IoT ed il dispositivo di controllo siano connessi ad un unica sottorete domestica. Inoltre si è deciso che il *Main Container* JADE risieda sul nodo del dispositivo di controllo in quanto è quest'ultimo che ha il compito di interagire ed ottenere informazioni dai dispositivi della cucina smart. Sul dispositivo di controllo si è scelto di sviluppare un agente JADE ad hoc che si occupi di interrogare i dispositivi IoT e di elaborare le informazioni ottenute.

Riguardo i singoli dispositivi IoT, si è scelto di progettare un componente LPaaS per ciascuno di essi e di dotarli di un agente LPaaS che esegue in un container JADE periferico. Il componente LPaaS ha il compito di gestire la base di conoscenza del nodo e di esporre la lista di goal Prolog che il dispositivo IoT fornisce come servizi. La *tabella 9* riporta i goal Prolog che si sono progettati per esporre mediante LPaaS le informazioni mantenute da ciascun nodo IoT.

Frigorifero	
Goal	**Descrizione**
fridgeData(-List).	Ottiene tutti i dati presenti nella base di conoscenza del frigorifero.
fridgeModelInfo(-Info).	Ottiene le informazioni sul modello del frigorifero.
fridgeGroceryList(+Type, +Threshold, +UnitOfMeasure, -Supply).	Ottiene l'elenco degli alimenti in esaurimento. La variabile *Type* consente di specificare la tipologia di alimento, *Threshold* indica la soglia sotto la quale un alimento è da considerarsi in esaurimento e *UnitOfMeasure* indica l'unità di misura di *Threshold.*

fridgeGroceryList(+Threshold, +UnitOfMeasure, -Supply).	Ottiene l'elenco degli alimenti in esaurimento. La variabile *Threshold* indica la soglia sotto la quale un alimento è da considerarsi in esaurimento. *UnitOfMeasure* indica l'unità di misura di *Threshold.*
fridgeGroceryList(+Threshold, -Supply).	Ottiene l'elenco degli alimenti in esaurimento. La variabile *Threshold* indica la soglia sotto la quale un alimento è da considerarsi in esaurimento.
getFridgeTemperature(-Temperature, -UnitOfMeasure).	Ottiene la temperatura interna del frigorifero.
fridgeAntiIceSystemStatus(-Status).	Ottiene lo stato del sistema anti ghiaccio del frigorifero.
fridgeCoolingSystemStatus(-Status).	Ottiene lo stato del sistema di raffreddamento del frigorifero.
fridgeStatus(-Status).	Ottiene un'informazione generale sullo stato attuale del frigorifero.
Frullatore	
Goal	**Descrizione**
mixerData(-List).	Ottiene tutti i dati presenti nella base di conoscenza del frullatore.
mixerModelInfo(-Info).	Ottiene le informazioni sul modello del frullatore.
mixLiked(-Liked).	Ottiene l'elenco dei frullati che sono piaciuti all'utente.
mixerStatus(-Status).	Ottiene un'informazione generale sullo stato attuale del frullatore.
Forno	
Goal	**Descrizione**
ovenData(-List).	Ottiene tutti i dati presenti nella base di conoscenza del forno.
ovenModelInfo(-Info).	Ottiene le informazioni sul modello del forno.
getOvenTemperature(-Temperature, -UnitOfMeasure).	Ottiene la temperatura interna del forno.
ovenCoolingSystemStatus(-Status).	Ottiene lo stato del sistema di raffreddamento del forno.
ovenStatus(-Status).	Ottiene un'informazione generale sullo stato attuale del forno.

getRecentCookedMeals(-List).	Ottiene la lista dei pasti cucinati di recente.
Dispensa	
Goal	**Descrizione**
pantryData(-List).	Ottiene tutti i dati presenti nella base di conoscenza della dispensa.
pantryModelInfo(-Info).	Ottiene le informazioni sul modello della dispensa.
pantryGroceryList(+Type, +Threshold, +UnitOfMeasure, -Supply).	Ottiene l'elenco dei beni in esaurimento. La variabile *Type* consente di specificare la tipologia di alimento, *Threshold* indica la soglia sotto la quale un bene è da considerarsi in esaurimento e *UnitOfMeasure* indica l'unità di misura di *Threshold.*
pantryGroceryList(+Threshold, +UnitOfMeasure, -Supply).	Ottiene l'elenco dei beni in esaurimento. La variabile *Threshold* indica la soglia sotto la quale un bene è da considerarsi in esaurimento. *UnitOfMeasure* indica l'unità di misura di *Threshold.*
pantryGroceryList(+Threshold, -Supply).	Ottiene l'elenco dei beni in esaurimento. La variabile *Threshold* indica la soglia sotto la quale un bene è da considerarsi in esaurimento.
pantryStatus(-Status).	Ottiene un'informazione generale sullo stato attuale della dispensa.
getPantryTemperature(-Temperature, -UnitOfMeasure).	Ottiene la temperatura interna della dispensa.
pantryAeringSystemStatus(-Status).	Ottiene lo stato del sistema di aerazione della dispensa.

tabella 9 – goal Prolog esposti dai servizi LPaaS.

Al fine di permettere il management remoto dei motori inferenziali dei componenti LPaaS si è scelto di attivare su ciascuno di essi il sistema di monitoraggio, controllo e gestione JMX (paragrafo 6.3.4). Si è fatta questa scelta in quanto si vuole permettere all'amministratore del sistema *Smart Kitchen* di poter intervenire direttamente sui motori inferenziali, bypassando i meccanismi di LPaaS.

L'agente LPaaS in esecuzione su ciascun nodo IoT gestisce il componente LPaaS e garantisce che la comunicazione tra le entità del sistema avvenga in modo sicuro mediante autenticazione e cifratura dei messaggi (paragrafi 4.2.2.3 e 5.3.2.2).

Si è deciso inoltre di non disabilitare la mobilità degli agenti anche se il supporto alla sicurezza JADE non garantisce protezione da attacchi malevoli durante la migrazione di un agente. Ciò non costituisce un rischio per la sicurezza del sistema in quanto si ipotizza di dover compiere un numero limitato di migrazioni (ad esempio soltanto quando un dispositivo necessita di manutenzione o deve essere sostituito).

Per quanto riguarda la possibilità di aggiungere dispositivi al sistema senza doverne interrompere l'esecuzione, si è deciso di permettere all'amministratore della *Smart Kitchen* di poter creare container JADE periferici e di poter attivare su questi nuovi agenti.

La *figura 38* dà una visione d'insieme della piattaforma JADE progettata: lo *Smart Kitchen Agent* è l'agente che ha il ruolo di controllore del sistema. L'interazione tra *Smart Kitchen Agent* ed agenti LPaaS avviene tramite lo scambio di messaggi ACL *LPaaS Request/Response:* in questo modo lo *Smart Kitchen Agent* può ottenere informazioni di alto livello dai dispositivi IoT, elaborarle e decidere quale azione compiere. Ad esempio, se un alimento presente nel frigorifero è in esaurimento, lo *Smart Kitchen Agent* potrebbe procedere ad effettuare un ordine online.

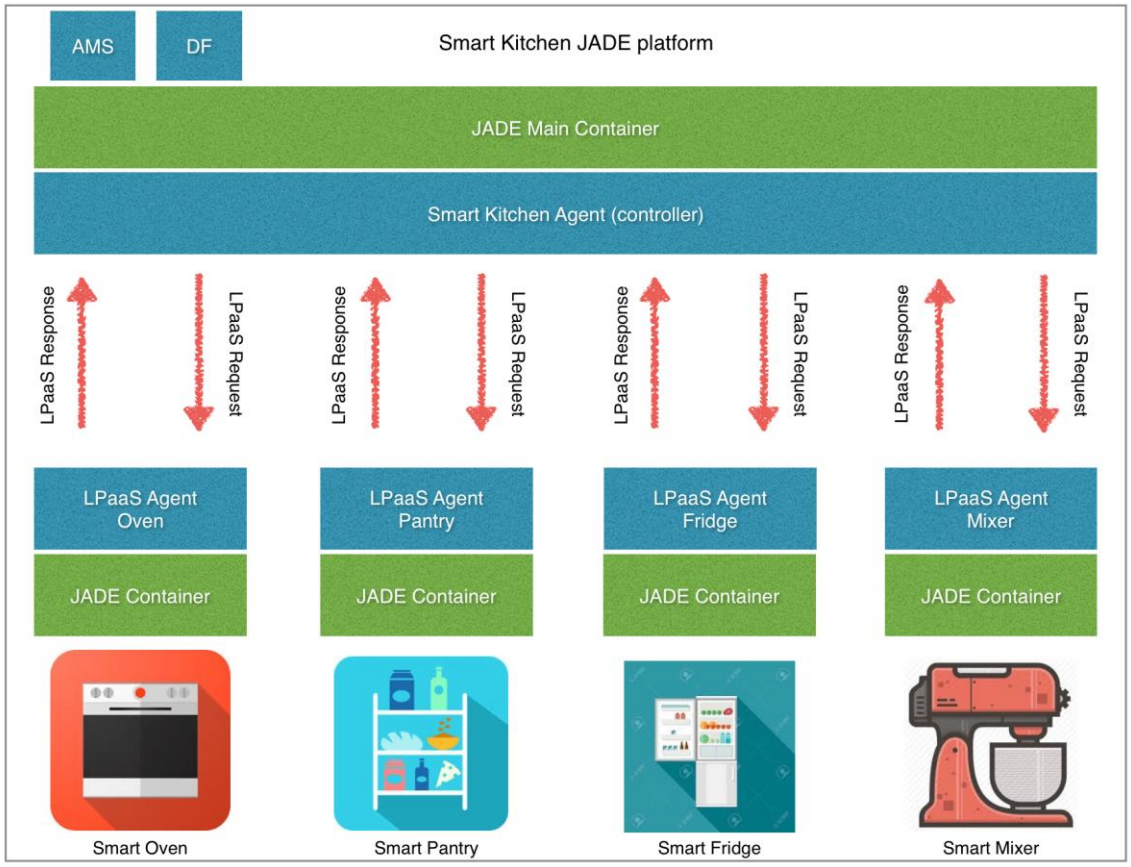

figura 38 – architettura del sistema Smart Kitchen.

7.4 Implementazione prototipo

7.4.1 Componenti LPaaS

Per ogni dispositivo IoT si è implementato soltanto un componente LPaaS ad hoc in quanto l'agente LPaaS è fornito dal supporto software sviluppato. Nella descrizione dell'implementazione dei componenti LPaaS si prenderà come esempio il componente che modella i servizi offerti dal frigorifero (*FridgeLPaaS*). Tale componente, mostrato in *figura 39*, estende la classe di supporto *LPaaSComponent*: in questo modo eredita le implementazioni dei metodi definiti nelle interfacce *client* e *configurator* di LPaaS (paragrafo 4.1.1.1).

```
@AsSingleton
public class FridgeLPaaS extends LPaaSComponent {

    @PrologManagement(host = "localhost", port = 45001, lazyBoot = false,
            adaptor = PrologMXBeanServer.HTTP_ADAPTOR,
            credentialFile = "./credential.txt",
            SSLconfigFile = "./ssl.properties")
    @PrologConfiguration(directives = {"set_prolog_flag(occursCheck, off)."},
            fromFiles = {"fridgeTheory.pl"})
    @GoalsToMatch(toMatch = {"fridgeData(List).", "fridgeModelInfo(Info).",
            "fridgeGroceryList(Type, Threshold, UnitOfMesure, Supply).",
            "fridgeGroceryList(Threshold, UnitOfMesure, Supply).",
            "fridgeGroceryList(Threshold, Supply).",
            "getFridgeTemperature(Temperature, UnitOfMeasure).", "fridgeAntiIceSystemStatus(Status).",
            "fridgeCoolingSystemStatus(Status).", "fridgeStatus(Status)."})
    private Prolog prolog;

    private JFrame frame = null;

    @PostConstructCall
    @PostMigrationSetup
    public void setupGUI(){
        LPaaSComponent component = this;
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    frame = new LPaaSComponentGUI("Fridge", component);
                    frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

    @OnDispose
    @PreMigrationSetup
    public void disposeGUI(){
        if(frame!=null){
            frame.setVisible(false);
            frame.dispose();
        }
    }

}
```

figura 39 – implementazione componente FridgeLPaaS.

Il componente è marcato con l'annotazione *@AsSingleton* in quanto si desidera che il container, in esecuzione sull'agente LPaaS, ne crei un'unica istanza.

Il motore inferenziale del componente è annotato con tre annotazioni: *@PrologManagement* e *@PrologConfiguration* configurano il motore inferenziale (paragrafo 5.1.2) mentre *@GoalsToMatch* configura i goal che il componente espone attraverso LPaaS. Nell'annotazione sono riportati tutti i goal elencati in *tabella 9* per il frigorifero. L'implementazione delle teorie logiche associate ai goal elencati in *@GoalsToMatch* è nel file *fridgeTheory.pl* specificato in *@PrologConfiguration.* Questo file contiene anche teorie logiche private del componente che consentono la dimostrazione dei goal esposti come servizio.

Il componente definisce anche due metodi *setupGUI()* e *disposeGUI():* essi gestiscono un'interfaccia grafica di testing che permette di interagire con il componente *FridgeLPaaS*. Si è scelto di utilizzare un'interfaccia grafica per simulare le informazioni prodotte dal frigorifero in quanto non si dispone dell'hardware dotato di opportuna sensoristica. Attraverso questa interfaccia, mostrata in *figura 40*, è possibile aggiungere una nuova teoria al componente, aggiungere/rimuovere un goal del servizio e risolvere con o senza vincoli temporali un goal Prolog.

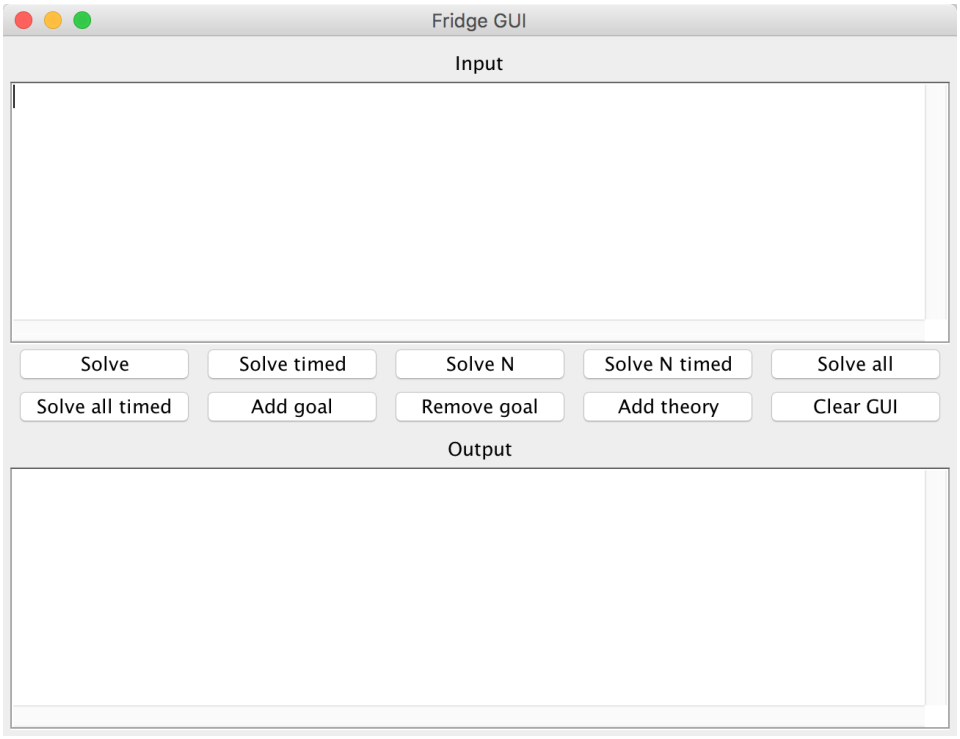

figura 40 – interfaccia grafica del componente FridgeLPaaS.

È importante osservare come siano annotati i metodi che gestiscono l'interfaccia grafica: *setupGUI()* è annotato con *@PostConstructCall* e *@PostMigrationSetup* che garantiscono che l'interfaccia grafica sia opportunamente configurata ed attivata dopo la creazione del componente e dopo la riattivazione del componente in caso di

migrazione. *disposeGUI()* è annotato con *@OnDispose* e *@PreMigrationSetup* che garantiscono che l'interfaccia grafica sia eliminata alla terminazione dell'applicazione o perché si sta migrando l'agente che gestisce il componente. In una implementazione non prototipale del sistema, queste annotazioni possono essere utilizzate per marcare opportuni metodi che collegano/scollegano il componente ed il motore inferenziale all'entità software che gestisce i sensori dei dispositivi IoT.

Per quanto riguarda gli altri componenti del sistema si è realizzata una implementazione analoga a quella mostrata per il frigorifero.

7.4.2 Smart Kitchen Agent

Per quanto riguarda l'agente che interagisce con i dispositivi IoT, si è scelto di implementare un agente JADE con il seguente behaviour ad hoc:

1. L'agente attende un goal Prolog dall'utente.
2. Quando un utente richiede di dimostrare un goal, l'agente compie una operazione di discovery: cerca sul DF un agente LPaaS in grado di risolvere il goal. Per individuare l'agente che offre il servizio logico, lo *Smart Kitchen Agent* invia a tutti gli agenti che offrono servizi LPaaS un messaggio di *LPaaS Request* contenente il goal *isGoal(goalUtente)* che testa se il servizio LPaaS espone il *goalUtente* richiesto.
3. Se l'agente ottiene una risposta affermativa all'operazione di discovery, procede a richiedere all'agente LPaaS individuato il servizio richiesto dall'utente, altrimenti segnala errore. Nel caso siano presenti più servizi che offrono lo stesso goal Prolog, si è scelto di interrogare il primo che risponde alla operazione di discovery.

Il behaviour dello *Smart Kitchen Agent* non è ciclico, ma viene schedulato ogni volta che l'utente richiede la dimostrazione di un goal. Il behaviour è sicuro: autentica e cifra tutti i messaggi che scambia con gli altri componenti del sistema. In caso di errori dovuti alla sicurezza, l'agente mostra opportuni messaggi d'errore.

Inoltre l'agente, come tutti gli altri agenti del sistema, supporta la mobilità.

Nell'implementazione prototipale del sistema, lo *Smart Kitchen Agent* gestisce un'interfaccia grafica (mostrata nel paragrafo 7.5) che permette ad un utente di interagire con i dispositivi IoT mediante servizi LPaaS: in una implementazione reale del sistema l'agente potrebbe svolgere compiti più complessi come quello di fare un ordine online se alcuni prodotti sono in esaurimento, contattare la ditta di manutenzione del frigorifero o del forno se lo stato di questi dispositivi indica che è necessario fare manutenzione, oppure suggerire all'utente che cosa potrebbe cucinare in base ai suoi gusti e alle sue abitudini.

7.4.3 Configurazione del sistema

Per quanto riguarda la configurazione del sistema *Smart Kitchen*, ciascun agente esegue su un nodo diverso e deve essere opportunamente configurato per svolgere i propri compiti. Su tutti i nodi, assieme all'eseguibile dell'applicazione, sono presenti i seguenti file di configurazione:

- *agent_configurator.pl*: contiene una clausola nel formato *agent(PropertiesFile, IsMain)* per la configurazione dichiarativa dell'agente in esecuzione sul nodo.
- Due file *.properties*: uno obbligatorio, che contiene le proprietà che configurano l'agente in esecuzione sul nodo (il nome del file deve corrispondere con il valore della variabile *PropertiesFile* della clausola *agent*), l'altro dipende dalla natura dell'agente e contiene le proprietà per configurare il container per componenti applicativi (paragrafo 5.2.1). I nomi di questi file sono a scelta del programmatore. Per quanto riguarda gli agenti LPaaS, il file che contiene le proprietà dell'agente comprende una proprietà di nome *configurator* che indica l'identità dell'utente del sistema JADE che può configurare i servizi LPaaS.
- *credential.txt*: contiene username e password dell'amministratore del sistema di management JMX dei motori Prolog dei componenti LPaaS.
- *ssl.properties:* contiene i parametri di configurazione del protocollo SSL-TLS utilizzato dal sistema di management JMX dei motori Prolog dei componenti LPaaS.
- *jaas.txt:* contiene la configurazione di *Java Authentication and Authorization Service*. Si è scelto di utilizzare la modalità *Simple* (paragrafo 2.1.4.5.1) in quanto il prototipo del sistema non necessita di un meccanismo di autenticazione che garantisca un livello di sicurezza maggiore.
- *passwords.txt:* contiene username e password degli utenti che possono autenticarsi presso la piattaforma JADE.
- *policy.txt:* contiene le policy di accesso alla piattaforma JADE. I permessi hanno il formato presentato nel paragrafo 2.1.4.5.2 (*figura 41*).

```
grant principal jade.security.Name "alberto" {
       permission jade.security.PlatformPermission "", "create,kill";
       permission jade.security.ContainerPermission "", "create,kill";
       permission jade.security.AgentPermission "", "create,kill";
       permission jade.security.AgentPermission "", "suspend,resume";
       permission jade.security.AMSPermission   "", "register,deregister,modify";
       permission jade.security.MessagePermission "", "send-to";
};
```

figura 41 – esempio di permessi concessi ad un utente del sistema JADE Smart Kitchen.

L’esempio di policy di sicurezza mostrato in *figura 41* concede un insieme di permessi all’utente che si autentica alla piattaforma JADE con identità “*alberto”*. L’utente può creare e distruggere un container JADE principale o periferico (*PlatformPermission* e *ContainerPermission* con azioni concesse di *create* e *kill*) ed ha pieno controllo degli agenti che può attivare nel sistema (*AgentPermission* con azioni concesse di *create*, *kill*, *suspend* e *resume*). Inoltre l’utente ha i permessi di gestire la registrazione dei propri agenti all’AMS (*AMSPermission* con azioni concesse di *register*, *deregister*, e *modify)* e di consentire ai propri agenti di comunicare mediante lo scambio di messaggi (*MessagePermission* con azione *send-to*).

7.5 Deployment e testing

Il prototipo del sistema è stato testato con il seguente deployment:

- ***Smart Kitchen Agent:*** esegue nel *Main container* JADE attivato su un *Macbook Pro* (versione *late 2016*).
- ***Agente LPaaS frigorifero e forno:*** eseguono ciascuno in un container periferico JADE attivato su un *Macbook Pro* (versione *late 2016*).
- ***Agente LPaaS frullatore e dispensa:*** eseguono ciascuno in un container periferico JADE attivato su un *Raspberry Pi 2*[RASP]. I relativi componenti LPaaS non sono dotati di interfaccia grafica (hanno una struttura analoga a quella mostrata in *figura 39*, paragrafo 7.4.1, ma non dichiarano i metodi *setupGUI()* e *disposeGUI()*).

Il primo test effettuato sul sistema prevede che l'utente richieda allo *Smart Kitchen Agent* di scoprire se il frigorifero sta per terminare il proprio rifornimento di frutta. Si è scelto di testare la query Prolog *fridgeGroceryList(frutta, 2, unit, Supply)* imponendo anche un vincolo temporale sull'esecuzione del servizio LPaaS (si veda il paragrafo 6.3.2 per quanto riguarda la gestione di vincoli di tempo in tuProlog): nel primo test si impone che il servizio non esegua per più di due millisecondi, nel secondo si impone un tempo massimo di esecuzione pari a cento millisecondi. L'indicazione temporale è da considerarsi come tempo massimo di esecuzione del servizio localmente al componente LPaaS che lo offre.

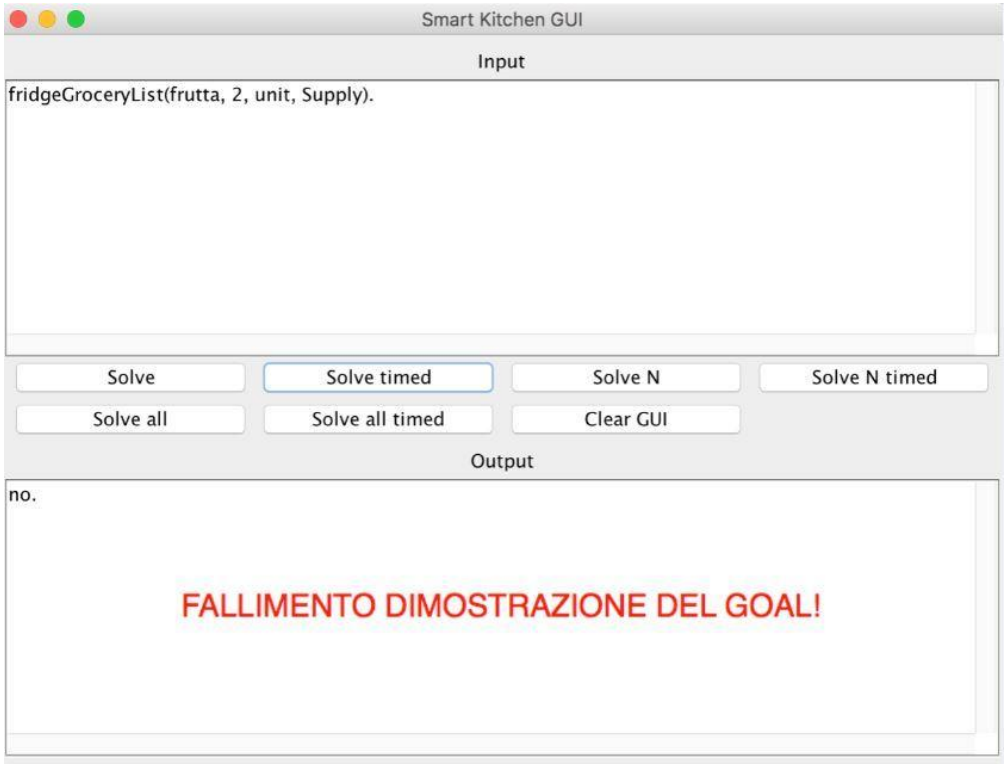

figura 42 – esecuzione del goal fridgeGroceryList con vincolo temporale di due millisecondi.

La *figura 42* mostra il risultato dell'esecuzione del primo test: la query fallisce in quanto il motore inferenziale del componente LPaaS del frigorifero non riesce a dimostrare la query entro il limite temporale di due millisecondi (il vincolo temporale è inserito in una GUI ad hoc, non mostrata in figura, resa visibile quando si preme il bottone *solve timed*). L'esito negativo della query è dovuto al fatto che il motore inferenziale del componente LPaaS del frigorifero è transitato nello stato di *TIME ELAPSED* (paragrafo 6.3.2.2).

La *figura 43* mostra il risultato del secondo test: in questo caso la query va a buon fine in quanto il motore inferenziale ha tempo a sufficienza per dimostrarla.

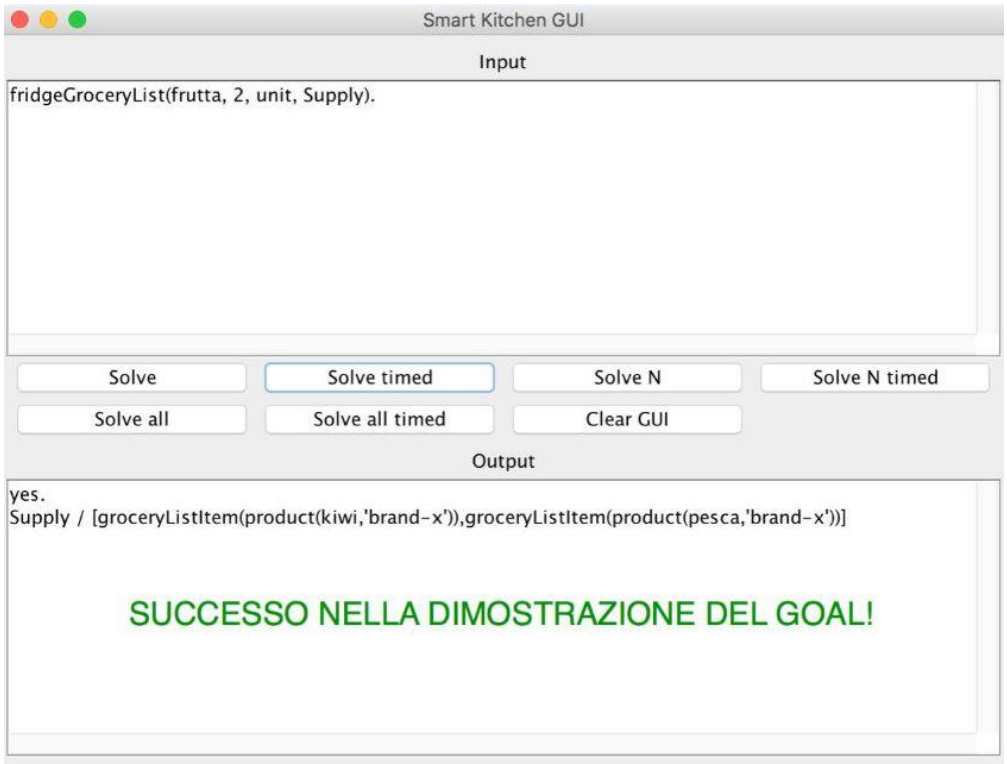

figura 43 – esecuzione del goal fridgeGroceryList con vincolo temporale di cento millisecondi.

Il risultato della query informa lo *Smart Kitchen Agent* che i kiwi e le pesche di marca "*brand-x"* sono in esaurimento in quando sono sotto le due unità.

Successivamente si sono testati (con e senza vincoli di tempo più o meno stringenti) tutti i goal elencati in *tabella 9* (paragrafo 7.3).

Questa tipologia di test dimostra sia che il sistema ha il comportamento atteso sia che è possibile limitare l'esecuzione temporale di una query Prolog LPaaS: così facendo si garantisce che la computazione termini entro il vincolo temporale indicato nella richiesta di servizio.

Il secondo tipo test effettuato sul sistema verifica la capacità di un agente di poter migrare da un nodo computazionale ad un altro. Il test prevede di migrare l'agente LPaaS del frullatore (la scelta è arbitraria e può coinvolgere qualsiasi agente del sistema), in esecuzione su un container JADE periferico attivato sul *Raspberry Pi 2*, su un nuovo container periferico attivato sul *Macbook Pro*.

Come primo passo, il test prevede di dimostrare il goal *createAgentContainer*: la risoluzione di questo goal è demandata al componente *Agent Configurator* (in esecuzione sul *Macbook Pro*) che utilizza *Agent Library* per manipolare la configurazione del sistema ad agenti. La dimostrazione del goal crea un nuovo container periferico nella piattaforma JADE. Successivamente, si richiede al componente *Agent Configurator*, in esecuzione sul *Raspberry*, di dimostrare il goal *migrateAgent(lpaasAgent_mixer, tempMixer)*: la risoluzione di questo goal scatena la migrazione dell'agente LPaaS del frullatore (*lpaasAgent_mixer*) verso il container di nome *tempMixer* creato al passaggio precedente sul *Macbook Pro*. Infine si richiede al componente *Agent Configurator* in esecuzione sul *Raspberry* di dimostrare il goal *destroyAgentWrapper(lpaasAgent_mixer)* che interrompe l'esecuzione del container sorgente della migrazione (paragrafi 4.3 e 5.4).

La *figura 44* mostra il risultato della soluzione del goal *createAgentContainer*, mentre la *figura 45* mostra il risultato della dimostrazione dei goal *migrateAgent* e *destroyAgentWrapper*.

```
User 'mixer' Successfully Authenticated.

giu 27, 2017 4:03:33 PM jade.core.PlatformManagerImpl localAddNode
INFORMAZIONI: Adding node <tempMixer> to the platform
giu 27, 2017 4:03:33 PM jade.core.PlatformManagerImpl$1 nodeAdded
INFORMAZIONI: --- Node <tempMixer> ALIVE ---
giu 27, 2017 4:03:33 PM jade.core.AgentContainerImpl joinPlatform
INFORMAZIONI: ----------------------------------------
Agent container tempMixer@192.168.137.1 is ready.
------------------------------------------------
yes.
```

figura 44 – risultato della soluzione del goal createAgentContainer.

```
INFO: ----------------------------------------
Agent container mixer_agent@169.254.25.76 is ready.
------------------------------------------------
LPaaS Agent: registered on Jade DFService as Name: mixer, Type: lpaas.
LPaaS Agent: started.
migrateAgent(lpaasAgent_mixer, tempMixer).
yes.

destroyAgentWrapper(lpaasAgent_mixer).
Mar 28, 2016 12:55:13 AM jade.core.messaging.MessageManager shutdown
INFO: MessageManager shutting down ...
yes.

Mar 28, 2016 12:55:13 AM jade.core.Runtime$1 run
INFO: JADE is closing down now.
```

figura 45 – risultato della dimostrazione dei goal migrateAgent e destroyAgentWrapper.

Al termine del processo di migrazione è possibile testare il successo dell'operazione interagendo con lo *Smart Kitchen Agent*: ad esempio si è scelto di testare la risoluzione del goal *mixLiked(Liked)* che ottiene informazioni sui frullati che sono piaciuti all'utente. La *figura 46* mostra il risultato del goal: l'utente ha gradito due frullati. Dato l'esito positivo della richiesta LPaaS, si assume che la migrazione dell'agente e del componente LPaaS del frullatore sia andata a buon fine.

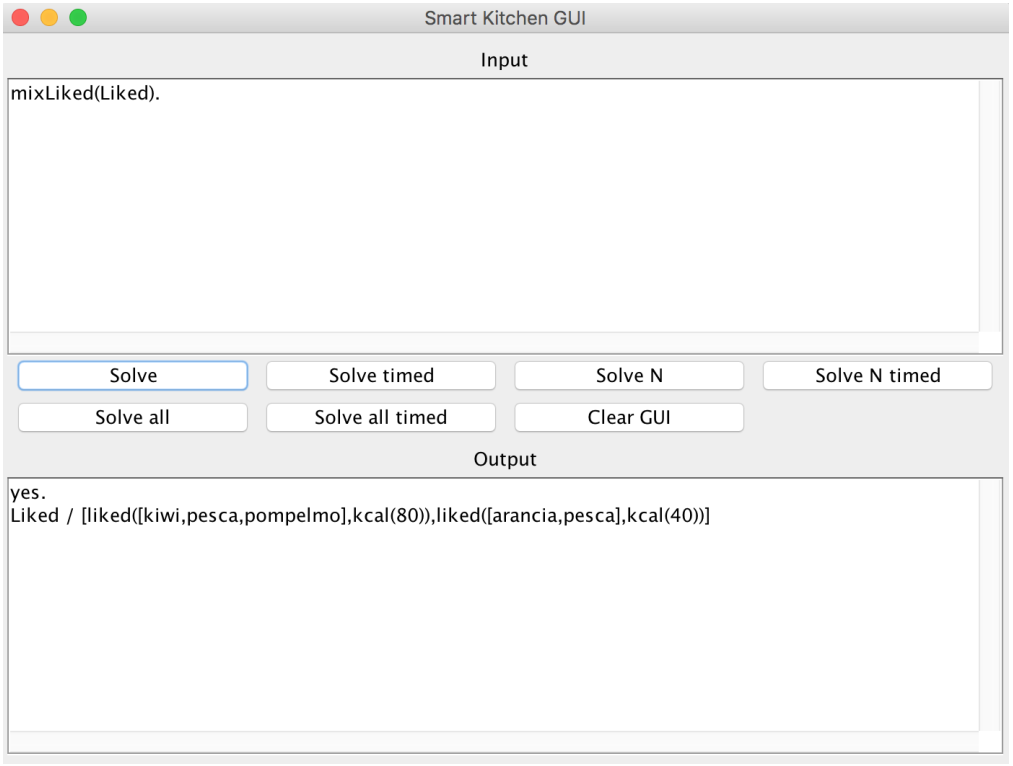

figura 46 – risultato della dimostrazione del goal mixLiked.

L'ultima tipologia di test eseguita sul prototipo ha l'obiettivo di verificare il corretto funzionamento del sistema di management JMX dei motori inferenziali dei componenti. Si è scelto di testare il sistema di management JMX del motore del componente LPaaS del frigorifero per ottenere l'elenco delle librerie caricate nel motore e per ottenere le teorie dinamiche della base di conoscenza del motore. Con riferimento al paragrafo 6.3.4, da browser web (nel test si è utilizzato *Safari* su *MacOS El Capitan*) si sono invocati i seguenti URL:

1. https://localhost:45001/invoke?objectname=service:prolog=libraryManager&operation=fetchCurrentLibraries
2. https://localhost:45001/invoke?objectname=service:prolog=theoryManager&operation=fetchKnowledgeBase&type0=boolean&value0=true

Entrambi gli URL prevedono che si utilizzi il supporto SSL-TLS (*https*): in questo modo le operazioni di management possono essere effettuate da browser in modo sicuro. La porzione di URL *localhost:45001/invoke?* è costante in entrambi ed indica che si vuole invocare un'operazione di management (*invoke*) sul server JMX in esecuzione all'indirizzo e porta specificati (*localhost, 45001*).

Il primo URL invoca sull'oggetto MBean associato al *LibraryManager* (*service:prolog=libraryManager*) di tuProlog il metodo *fetchCurrentLibraries* (*operartion*): in questo modo è possibile sapere quali librerie sono attualmente caricate nel motore.

Il secondo URL invoca sull'oggetto MBean associato al *TheoryManager* (*service:prolog=theoryManager*) di tuProlog il metodo *fetchKnowledgeBase* (*operartion*) con parametro di ingresso *true*: in questo modo si richiedono al motore Prolog tutte le teorie che costituiscono la base di conoscenza dinamica.

La *figura 47* mostra in ordine i risultati dell'invocazione degli URL descritti alla pagina precedente (il risultato del secondo URL è riportato parzialmente).

Il risultato delle operazioni è in formato HTML e contiene due tag:

- ***MBeanOperation:*** racchiude il risultato dell'operazione di management.
- ***Operation:*** contiene le informazioni sull'operazione richiesta. Si caratterizza per le seguenti proprietà:
    - ***objectname:*** è l'identificativo dell'oggetto MBean coinvolto nell'operazione di management.
    - ***operation:*** è l'operazione richiesta. Tipicamente ha valore *invoke* in quanto si richiede l'esecuzione di un metodo.
    - ***result:*** è il risultato dell'operazione.
    - ***return:*** è il valore del risultato (nel caso mostrato in *figura 47* è una stringa JSON)
    - ***returnclass:*** è il tipo Java del risultato dell'operazione.

Infine si sono testate tutte le funzionalità esposte in *tabella 8* (paragrafo 6.3.4.4): i risultati positivi dei test permettono di concludere che il sistema di management ha il comportamento desiderato.

```
<MBeanOperation>
<Operation objectname="service:prolog=libraryManager" operation="invoke" result="success"
return="["alice.tuprolog.lib.BasicLibrary","alice.tuprolog.lib.ISOLibrary","alice.tuprolog.lib.IOLibrary","alice.tu
prolog.lib.OOLibrary"]" returnclass="java.lang.String"/>
</MBeanOperation>

<MBeanOperation>
<Operation objectname="service:prolog=theoryManager" operation="invoke" result="success"
return=""fridgeModel(samsung,xvcd567yuhGo).\n\nfridgeAntiIce(on,timestamp(0)).\n
\nfridgeData([fridgeSupply(mela,frutta,\u0027brand-z\u0027,2,unit,timestamp(0)),fridgeSupply(pera,frutta,
\u0027brand-y\u0027,3,unit,timestamp(0)),fridgeSupply(kiwi,frutta,\u0027brand-x
\u0027,1,unit,timestamp(0)),fridgeSupply(banana,frutta,\u0027brand-x
\u0027,3,unit,timestamp(0)),fridgeSupply(arancia,frutta,\u0027brand-y
\u0027,10,unit,timestamp(0)),fridgeSupply(caco,frutta,\u0027brand-x
\u0027,4,unit,timestamp(0)),fridgeSupply(anguria,frutta,\u0027brand-y
\u0027,4,unit,timestamp(0)),fridgeSupply(pesca,frutta,\u0027brand-x
\u0027,0,unit,timestamp(0)),fridgeSupply(pompelmo,frutta,\u0027brand-x\u0027,2,unit,timestamp(0))]).\n
\nfridgeData([fridgeSupply(insalata,verdura,\u0027brand-z
\u0027,2,unit,timestamp(0)),fridgeSupply(pomodori,verdura,\u0027brand-y\u0027,3,unit,timestamp(0))]).\n
\nfridgeStatus(on,timestamp(0)).\n\nfridgeStatus(S) :-\n\tgetTime(O),\n\tfridgeStatus(A,timestamp(O)),\n\tS
\u003d A,\n\tfindall(fridgeStatus(A,B),fridgeStatus(A,B),L),\n\tretractDataList(L),\n
\tassert(fridgeStatus(A,timestamp(O))),\n\t!.\n\nretractDataList(\u0027[]\u0027).\n\nretractDataList([H|
T]) :-\n\tretract(H),\n\tretractDataList(T).\n\nfridgeCooling(on,timestamp(0)).
...
...
returnclass="java.lang.String"/>
</MBeanOperation>
```


figura 47 – risultati delle invocazioni delle operazioni di management JMX.

7.5 Osservazioni e sviluppi futuri

L'utilizzo del supporto software sviluppato ha ridotto il tempo di realizzazione del sistema *Smart Kitchen* in quanto ha permesso di progettare soltanto i componenti LPaaS dei singoli dispositivi IoT, lo *Smart Kitchen Agent* e le interfacce grafiche poiché lo sviluppatore deve progettare solamente ciò che è strettamente *application-specific*. Il supporto middleware ha anche permesso di dotare i singoli componenti del sistema di intelligenza situata: ciò è stato reso possibile inserendo un motore Prolog, contenuto nei componenti applicativi, all'interno di agenti LPaaS. Inoltre il sistema permette di gestire trasparentemente la migrazione dei motori inferenziali dei componenti: ciò semplifica la gestione degli agenti e rende più facile e naturale l'utilizzo di un motore inferenziale in un contesto distribuito.

Il prototipo non è da considerarsi completo: ad esempio manca della gestione della persistenza della base di conoscenza dei motori inferenziali e non è tollerante ad eventuali guasti dei nodi su cui eseguono gli agenti. Futuri sviluppi del sistema possono prevedere sia l'utilizzo di un sistema di persistenza che memorizzi in un database le basi di conoscenza dei componenti applicativi sia lo scambio di messaggi di *heartbeat* tra gli agenti per testare se un elemento del sistema è attivo o ha subito un fault. In quest'ultimo caso è possibile prevedere un componente del sistema che tenti di riattivare il software che ha subito un malfunzionamento.

Il prototipo possiede gli strumenti per gestire la piattaforma ad agenti mediante la programmazione logica (*AgentLibrary*) ma non è reattivo ai cambiamenti del contesto di esecuzione dei nodi: futuri sviluppi del sistema possono prevedere entità software che si occupino di rimanere in ascolto dei cambiamenti dell'ambiente di esecuzione e, in base a questi, configurare e gestire dichiarativamente la piattaforma ad agenti.

8. Conclusioni

Il supporto software sviluppato nella tesi è una possibile soluzione per integrare la programmazione distribuita ad agenti alla programmazione logica. Tale unione è stata resa possibile attraverso lo sviluppo di un supporto middleware di livello applicativo: il container per componenti logici. Il container si caratterizza principalmente per essere leggero, flessibile e facilmente utilizzabile in altri sistemi; per quanto riguarda l'integrazione con il sistema ad agenti JADE, è stato possibile realizzare un mapping uno a uno con il ciclo di vita di un agente ed il ciclo di vita del container. L'utilizzo del modello *componente-container* non si limita soltanto a velocizzare lo sviluppo di un applicativo distribuito ed a rendere disaccoppiato il supporto alla gestione dei componenti dagli altri elementi di infrastruttura del sistema (come nel caso di studio *Smart Kitchen*, capitolo 7), ma facilita l'utilizzo della programmazione logica all'interno di un sistema distribuito, ad agenti e non. La programmazione logica è dunque utilizzabile sia in forma di LPaaS, attraverso la definizione di opportuni componenti, sia in modo indiretto e non standardizzato, definendo implementazioni ad hoc dei componenti che la utilizzano.

Per quanto riguarda il supporto a LPaaS, l'utilizzo di un agente per offrire un servizio logico favorisce la flessibilità del sistema in quanto consente di rilocare un servizio a seconda delle necessità applicative. L'implementazione realizzata dell'agente LPaaS gestisce un loop di attesa di richieste di servizio: come futuro miglioramento del supporto è possibile progettare un agente LPaaS che svolga sia il ruolo di *fornitore di un servizio* sia il ruolo di *cliente di un servizio*. In questo modo si otterrebbe la composizione ed il coordinamento di servizi LPaaS.

Unire la programmazione ad agenti a quella logica permette di sviluppare sistemi distribuiti IoIT complessi, caratterizzati principalmente per disporre di intelligenza situata e per essere interoperabili, flessibili e socievoli. Tali caratteristiche sono arricchite dalla possibilità di utilizzare la programmazione logica sia nella sua forma classica sia in modo ibrido, ovvero consentendo a quest'ultima di interoperare con altri paradigmi di programmazione, come quello imperativo ad oggetti. La programmazione multi-paradigma ha permesso di utilizzare la programmazione logica per agire direttamente sul sistema ad agenti: tale approccio è stato utilizzato nel sistema per la configurazione dichiarativa della piattaforma JADE ed è utilizzabile anche a livello applicativo.

In conclusione, il supporto alla configurazione dichiarativa può essere visto come il punto di partenza per individuare l'architettura di un middleware per la gestione di un sistema distribuito basato sulla programmazione logica. Tale middleware dovrebbe caratterizzarsi per possedere una struttura leggera, disaccoppiata e facilmente configurabile. Inoltre potrebbe sfruttare la programmazione multi-paradigma per

permettere di utilizzare il Prolog all'interno del middleware stesso; questo approccio potrebbe avere diversi vantaggi in quanto consentirebbe di utilizzare il non determinismo, tipico della programmazione logica, per gestire in maniera flessibile casi di errore a runtime. Inoltre la programmazione logica permetterebbe di impiegare l'unificazione come meccanismo non direzionato per lo scambio di messaggi tra entità multiple, favorendone la comunicazione e la cooperazione.

9. Bibliografia

[IOTG] Daugherty Paul; Negm Walid; Banerjee Prith; Alter Allan. "Driving Unconventional Growth through the Industrial Internet of Things".

https://www.accenture.com/mz-en/_acnmedia/Accenture/next-gen/reassembling-industry/pdf/Accenture-Driving-Unconventional-Growth-through-IIoT.pdf

[IOIT] Hugo Serra; Rui Francisco; Artur Arsénio. (2014) "Internet of Intelligent Things: Bringing Artificial Intelligence into Things and Communication Networks".

https://www.researchgate.net/publication/265248986_Internet_of_Intelligent_Things_Bringing_Artificial_Intelligence_into_Things_and_Communication_Networks

[LPaaS] Calegari R.; Denti E.; Mariani S.; Omicini A. (2017) "Logic Programming as a Service (LPaaS): Intelligence for the IoT".

http://apice.unibo.it/xwiki/bin/download/Publications/LpaasIcnsc2017/150.pdf

Calegari R.; Denti E.; Mariani S.; Omicini A. (2016) "Towards Logic Programming as a Service: Experiments in tuProlog".

http://ceur-ws.org/Vol-1664/w14.pdf

[FIPA] The Foundation for Intelligent Physical Agents.

http://www.fipa.org/

[AGL] Aglets platform web site.

http://aglets.sourceforge.net/

[DARPA] Defence Advanced Research Projects Agency official site.

https://www.darpa.mil/

[COUG] Cougaar Software.

http://www.cougaarsoftware.com/

A. Helsinger; T. Wright (2005) "Cougaar: a robust configurable multi-agent platform".

http://ieeexplore.ieee.org/document/1559614/

[AG.FACT] Agent Factory Main Page.

http://agentfactory.ucd.ie/index.php/Main_Page

[SEM] Secure Mobile Agents (SEMOA).

http://semoa.sourceforge.net/about/details.html

[JADE] Java Agent DEvelopment Framework.

http://jade.tilab.com/

[JADE-S] JADE Security.

http://jade.tilab.com/download/add-ons/

[2P] Denti E.; Omicini A.; Ricci A.; (2001) "tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures".

http://www-lia.deis.unibo.it/~ao/pubs/pdf/2001/padl.pdf

[2PMP] Denti E.; Omicini A.; Ricci A.; (2005) "Multi-paradigm Java-Prolog integration in tuProlog".

http://www-lia.deis.unibo.it/~ao/pubs/pdf/2005/scico-dor.pdf

[2PU] Denti E.; Omicini A.; Calegari R.; (2013) "tuProlog: Making Prolog Ubiquitous".

https://www.cs.nmsu.edu/ALP/2013/10/tuprolog-making-prolog-ubiquitous/

[2PARCH] Piancastelli G.; Benini A.; Omicini A.; Ricci A. "The Architecture and Design of a Malleable Object-Oriented Prolog Engine".

http://www-lia.deis.unibo.it/~ao/pubs/pdf/inpress/sac-pbor.pdf

[2PL] Home-page progetto tuProlog.

http://tuprolog.unibo.it

[DISI] Dipartimento Informatica – Scienza e Ingegneria, homepage.

http://www.informatica.unibo.it/it

[WS] "Web Services Architecture", W3C. 11-02-2004.

https://www.w3.org/TR/ws-arch/

[REST] Fielding, Roy Thomas (2000). "Chapter 5: Representational State Transfer (REST)". Architectural Styles and the Design of Network-based Software Architectures (Ph.D.). University of California, Irvine.

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[ACL] FIPA ACL Message Structure Specification.

http://www.fipa.org/specs/fipa00061/SC00061G.html

[JAAS] Java Authentication and Authorization Service Reference Guide.

https://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html

[GSON] Google GSON Github.

https://github.com/google/gson

[JMX] Java Management Extensions Technology.

http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html

[MX4J] Open-source JMX for enterprise computing.

http://mx4j.sourceforge.net/

[JSON] Final draft Standard ECMA-262 edition 5.1, 03-2011 (Rev. 6) - Ecma 262.

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[SK] Calegari R.; Denti E.; (2017) "Context Reasoning and Predictions in Smart Environments: The Home Manager Case".

https://link.springer.com/chapter/10.1007%2F978-3-319-59480-4_45

[RASP] Raspberry Pi official site.

https://www.raspberrypi.org/