Lambdas-as-a-Service

Matteo Castellucci
matteo.castellucci3@studio.unibo.it

Ottobre 2023

Questo progetto vuole realizzare il prototipo di un *web service* per la computazione distribuita, attraverso il quale un utente può mettere in esecuzione il proprio codice da remoto.

Un utente sottopone il proprio codice al servizio sotto forma di eseguibile, attraverso un *client* accessibile mediante il proprio *web browser*. Sarà quindi il sistema a decidere dove allocarlo e a rendere accessibile la sua esecuzione sempre mediante la stessa interfaccia utente. Il servizio offre delle funzionalità di base per quanto riguarda il controllo d'accesso degli utenti.

Esistono due tipologie di componente che costituiscono il sistema, quello detto "principale" e quello detto "secondario", che comunicano mediante uno spazio di tuple. Il sistema è pensato per funzionare con una sola copia del componente principale. Il numero di componenti secondari invece è libero: ne è presente uno per macchina, fisica o virtuale, sulla quale si desidera allocare e mettere in esecuzione i *file* eseguibili.

La *UI* dell'utente è offerta mediante il componente principale, il quale riceve anche tutte le richieste che l'utente fa grazie alla stessa. Questo significa che ad ogni nuova richiesta di allocazione di un eseguibile il componente principale dovrà farsi carico di diramare la richiesta ai componenti secondari. Questi si candideranno per ricevere il *deployment* dell'eseguibile e sarà compito di quello principale scegliere il più adatto ed assegnargli il compito. Una volta deciso quale componente secondario dovrà portare a termine l'allocazione, questo genererà al suo interno un agente dedicato solamente all'esecuzione di quell'eseguibile. Una volta fatto, sulla *UI* sarà visibile il nuovo eseguibile.

Anche per quanto riguarda la messa in esecuzione, il principio è lo stesso: viene contattato dapprima il componente principale, il quale inoltra la richiesta al giusto componente secondario e il risultato è pubblicato sulla *UI*.

1 Obiettivi e requisiti

1.1 Obiettivi

Questo progetto si dà come obiettivo quello di realizzare il prototipo di un *web service* per la computazione distribuita. Questo significa che il sistema permetterà ai suoi utenti di mettere in esecuzione il proprio codice indipendentemente dal fatto che accedano al sistema dalla macchina in cui esso si trova, dalla rete locale che ospita questa macchina o dalla rete Internet.

L'utente del servizio ha a disposizione due funzionalità principali. La prima è quella che permette all'utente di sottoporre il proprio *file* eseguibile al sistema perché quest'ultimo lo possa allocare su una qualche macchina su cui opera. Oltre al *file*, all'atto della richiesta si vuole che l'utente inserisca un nome per poter distinguere i diversi eseguibili tra loro. Inoltre, sarà sempre possibile cambiare il *file* e il nome scelti fino a che non si deciderà di inviarli. La seconda funzionalità è sempre disponibile per tutti gli eseguibili già allocati e diventa disponibile per un eseguibile una volta effettuato il suo *deployment*. Questa permette di mettere in esecuzione il *file*, fornendo i parametri che devono essergli passati in input. I parametri sono opzionali, in quanto è possibile che l'eseguibile non li debba ricevere. Una volta completata l'esecuzione, l'*output* della stessa verrà mostrato all'utente, il quale comprenderà l'"*exit code*" del processo dell'eseguibile, nonché tutto ciò che avrà stampato su "*standard output*" e "*standard error*".

Per i componenti che si occuperanno di ospitare i *file*, nonché di metterli in esecuzione, dovrà essere possibile specificare i tipi di eseguibile supportati, nonché il numero di "slot" disponibili. Un tipo di eseguibile è rappresentato dal linguaggio di programmazione in cui questo è scritto, il che significa che al momento questo può essere solamente il linguaggio Java. Uno slot rappresenta invece uno "spazio libero" per un eseguibile che il componente ha a disposizione, come approssimazione delle metriche di norma utilizzate per determinare la possibilità di allocazione.

1.2 Scenari d'uso

Lo scenario d'uso principale, nonché unico, previsto per l'utente del sistema è il seguente. Un utente ha necessità di mettere in esecuzione un certo *file* eseguibile per portare a termine un suo compito. La macchina che ha correntemente a disposizione non possiede i requisiti per poterlo fare perché non ha a disposizione una *toolchain* completa per l'esecuzione del *file* oppure non ha le specifiche tecniche. Oppure ancora, l'utente non ha i permessi di sicurezza per l'esecuzione o le risorse necessarie non sono disponibili dalla rete locale. Per uno qualsiasi di questi motivi, decide di ricorrere al servizio offerto da questo progetto.

A questo punto, collegandosi grazie al proprio *web browser*, l'utente avrà nella prima pagina della *web app* implementata la possibilità di registrarsi al sistema o di effettuare il *login*. L'utente potrà scegliere ciò che è più adatto alla sua situazione, effettuando la prima operazione se non si è mai registrato in precedenza, oppure la seconda se l'ha già fatto. Le credenziali richieste sono un nome utente e una *password*, nome utente che dovrà essere unico per ogni utente nel sistema. Effettuata una qualsiasi delle due

operazioni, comparirà la *home* dell'utente, che mostrerà tutti gli eseguibili già caricati. Se si è appena registrato, la schermata necessariamente apparirà vuota.

Nella schermata *home* l'utente avrà la possibilità di effettuare il *logout* e quindi fare in modo che il sistema si "dimentichi" del suo accesso. L'utente è libero di ricaricare la pagina o di chiudere e riaprire il *browser*: l'applicazione ricorderà il suo accesso e chiederà di effettuarlo nuovamente dopo un giorno, sia che sia stata utilizzata, sia che non lo sia stata. La schermata *home* offrirà anche la possibilità di caricare un nuovo eseguibile. Se l'utente decidesse di farlo, comparirà una maschera che richiederà l'inserimento del nome che l'utente vuole assegnare al *file* e la scelta dell'eseguibile dal computer dell'utente, con possibilità di cambiarlo anche dopo averlo già selezionato. Infine, tramite un apposito pulsante, l'utente potrà richiedere l'allocazione del *file* sul sistema. Qualora si concludesse con successo, il nuovo eseguibile comparirà nella schermata *home* assieme agli altri già caricati.

La terza e ultima possibilità per l'utente nella schermata *home* è quella di mettere in esecuzione un *file* già allocato. Sarà possibile per ciascun eseguibile aprire una maschera che permette di inserire gli argomenti da passare al *file* una volta messo in esecuzione. Una volta fatto, operazione comunque opzionale, tramite un pulsante sarà possibile avviare la messa in esecuzione e, al termine della stessa, comparirà una nuova maschera. Questa riporterà i risultati dell'esecuzione, ovvero l'"*exit code*" del processo che ha messo in esecuzione il *file*, nonché tutto ciò che il processo ha stampato su "*standard output*" e su "*standard error*".

Questo scenario è esemplificato dal diagramma dei casi d'uso in figura 1.

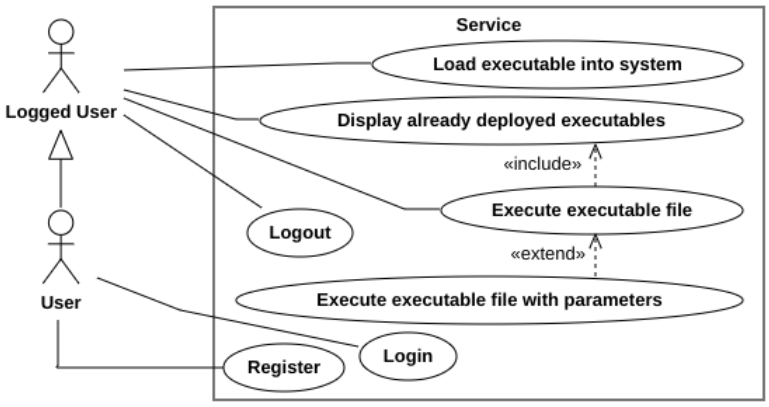

Figura 1: Diagramma dei casi d'uso dal punto di vista di un utente del servizio

1.3 Requisiti funzionali

1. Deve essere presente un meccanismo di autenticazione dell'utente nel sistema;
   1.1. L'utente deve potersi registrare al sistema;

1.1.1. La procedura di registrazione richiede all'utente di inserire un nome utente e una *password*;

1.1.2. Il nome utente deve essere unico tra tutti gli utenti del sistema, qualora l'utente scegliesse un nome già utilizzato, viene visualizzato un messaggio di errore;

1.1.3. Il nome utente non deve superare i 40 caratteri.

1.2. L'utente deve poter effettuare il *login* al sistema;

1.2.1. La procedura di *login* richiede all'utente di inserire il nome utente e la *password* scelti in fase di registrazione;

1.2.2. In caso ci sia corrispondenza con i dati inseriti da un utente registrato in precedenza, l'utente viene autenticato nel sistema;

1.2.3. In caso non ci sia nessuna corrispondenza, viene visualizzato un messaggio di errore e richiesto all'utente di riprovare di nuovo.

1.3. L'utente deve poter effettuare il *logout* dal sistema.

2. Una volta effettuato il *login* o la registrazione con successo, deve essere presentata all'utente la sua pagina *home*;

2.1. Questa mostrerà tutti i *file* eseguibili che l'utente ha caricato fino a quel momento;

2.2. Per ognuno di essi verrà mostrato il suo nome, oltre che un pulsante per mettere in esecuzione il *file* aprendo un'apposita maschera;

2.2.1. La maschera permetterà di passare i parametri necessari all'eseguibile, anche nessuno in caso non fosse necessario;

2.2.2. La maschera permetterà anche di visualizzare i risultati prodotti dall'eseguibile, qualora non si siano verificati errori durante l'esecuzione.

2.2.2.1. Deve essere visualizzato l'"*exit code*" del processo di esecuzione;

2.2.2.2. Deve essere visualizzato lo "*standard output*" del processo di esecuzione;

2.2.2.3. Deve essere visualizzato lo "*standard error*" del processo di esecuzione.

2.2.3. In caso di errore durante l'esecuzione, verrà mostrato il messaggio corrispondente all'errore verificatosi.

2.3. La *home* permetterà anche di caricare un nuovo eseguibile nel sistema e di allocare le risorse per la sua esecuzione attraverso una seconda maschera;

2.3.1. L'utente dovrà fornire il nome del *file* che intende trasferire e il *file* stesso;

2.3.2. Il nome del *file* non deve superare i 40 caratteri;

2.3.3. In caso l'utente decida di cambiare il *file* da caricare, deve essergli permesso di farlo mediante un apposito pulsante;

2.3.4. Se l'allocazione dell'eseguibile avrà successo, questo apparirà nella *home* dell'utente assieme agli altri già caricati;

2.3.5. In caso di errore durante il *deployment*, verrà mostrato il messaggio corrispondente all'errore verificatosi.

3. Ogni componente responsabile dell'allocazione di eseguibili sulla propria macchina deve poter essere adattato alle caratteristiche della stessa.

3.1. Deve essere possibile specificare quali tipi di eseguibili una certa macchina è capace di mettere in esecuzione;

3.1.1. Le tipologie di eseguibili sono rappresentate dal linguaggio di programmazione in cui il sorgente degli stessi è stato scritto;

3.1.2. È richiesto di supportare solamente il linguaggio Java, quindi gli eseguibili che possono essere forniti dall'utente sono *file* JAR funzionanti;

3.2. Deve essere possibile specificare la capacità di allocazione di una macchina.

3.2.1. La capacità di allocazione è rappresentata da un certo numero di "slot";

3.2.2. Ogni slot rappresenta la possibilità di mettere in esecuzione un *file*.

1.4 Politica di autovalutazione

Essendo questa una *proof of concept*, l'interesse è rivolto verso la fattibilità della realizzazione di un servizio che soddisfi i requisiti indicati. Questo significa che la qualità della soluzione verrà valutata nei termini della correttezza del suo codice e, conseguentemente, nei termini della sua capacità di soddisfare i requisiti. La correttezza del codice verrà valutata attraverso l'uso di *unit test* capaci di garantire sufficiente *code coverage* della *code base*. Questo ci garantirà un grado di certezza sufficiente nell'affermare che la soluzione non presenta difetti bloccanti e non bloccanti inerenti agli scenari d'uso previsti. Inoltre, saranno realizzati degli *integration test* per poter testare anche i moduli *software* nel loro complesso. In questo modo, le verifiche saranno ancora più aderenti alla realtà, andando a simulare un ambiente di *deployment* del servizio quanto più possibile reale.

Mettere al centro la fattibilità della soluzione implica anche che non c'è alcun interesse nel valutare tutta una serie di questioni pratiche che sarebbero invece importanti qualora questo progetto volesse realizzare una soluzione completa. Non verrà perciò valutata l'accessibilità, né tantomeno la *user experience*, della *UI* che verrà realizzata, senza che però questo rappresenti una scusa per produrre qualcosa di inammissibile in un contesto reale. Ci si limiterà a valutare manualmente l'usabilità dell'interfaccia utente, ovvero la sua capacità di soddisfare dei requisiti e l'assenza di difetti. Questo significa anche che non verranno nemmeno valutate le prestazioni del sistema realizzato e non si cercherà in nessun modo di ottimizzarle. Ci si vincola semplicemente a non realizzare soluzioni che apertamente violino i principi di efficienza, scalabilità e disponibilità.

2 Analisi dei requisiti

2.1 Requisiti non funzionali

1. La soluzione implementata deve permettere la corretta valutazione della fattibilità dei requisiti definiti;
   1.1. Non devono apparire errori bloccanti o non bloccanti durante l'utilizzo del servizio;
   1.2. L'interfaccia utente non deve presentare fenomeni di *stuttering* o *freezing*, ma il suo uso deve risultare fluido all'utente;
   1.3. La *code coverage* deve essere superiore al 65% sull'intero progetto.
2. La soluzione implementata deve essere distribuita, ovvero non è possibile realizzarla nella sua interezza come sistema nel concentrato;
   2.1. I dati presentati all'utente devono sempre essere consistenti con lo stato attuale del sistema, sia in caso che si verifichi un partizionamento di rete, sia nel caso non si verifichi. Si vuole preferire la correttezza dei dati alla disponibilità del servizio in quanto il progetto è volto alla realizzazione di un prototipo;
   2.2. Deve essere garantita la tolleranza ai partizionamenti di rete: il sistema riesce a notificare correttamente l'utente in caso di errore e gli permette di riprovare in seguito, ad errore risolto, per rispettare il requisito non funzionale 1.1;
3. Non verrà data nessuna garanzia sulla sicurezza del sistema implementato, eccetto che per alcuni accorgimenti basilari:
   3.1. La compartimentazione tra gli utenti dovrà essere totale: nessun utente deve poter capire quanti e quali eseguibili un altro utente ha caricato, men che meno dovrà poterli mettere in esecuzione;
   3.2. Le *password* verranno sempre salvate in maniera cifrata dopo un'operazione di *salting*.
4. Il sistema sarà trasparente nei confronti della "*data locality*", perciò non permetterà di suddividere il carico di lavoro tra componenti sulla base del luogo geografico in cui si trovano. Non sarà quindi permesso all'utente di scegliere quello più vicino a lui in caso di *deployment* di un eseguibile.

2.2 Requisiti implementativi

1. Ogni componente del sistema dovrà essere modellato come un "*agent*", inteso come un'entità autonoma, reattiva e proattiva, capace di incapsulare il proprio flusso di controllo. Questo permetterà una rigida separazione tra le parti "attive" e "passive" del sistema, favorendo i principi di "*separation of concern*" e "*single responsibility*". Oltre alla modularizzazione del sistema, l'utilizzo del paradigma ad agenti permette più facilmente di realizzare una soluzione distribuita, che oltre

ad essere un requisito, permette di ottenere un sistema più flessibile, scalabile, manutenibile e "*fault tolerant*";

1.1. L'implementazione degli *agent* impiegherà il *framework* "Akka", che si basa sul paradigma ad attori. La proattività degli attori, normalmente mancante, sarà simulata mediante il meccanismo di "*message self-sending*". La scelta degli attori è particolarmente adeguata perché permette di modellare le parti attive del sistema come semplici *state machine*, fornendo già gli strumenti per supportare lo scambio di messaggi tra componenti locali, astraendo dalle questioni di basso livello.

2. La comunicazione e il coordinamento tra componenti differenti, e quindi tra *agent* su macchine differenti, dovranno avvenire mediante un *coordination medium*. L'incarnazione scelta per questo è quella che si basa sulla nozione di "*tuple space*" e sul linguaggio di coordinamento "Linda" [4]. Questo garantirà la possibilità agli agenti non solo di comunicare tra di loro, ma anche di coordinare le proprie azioni e costruire quindi attività complesse per soddisfare i requisiti dati. La scelta del *tuple space* è particolarmente adeguata perché permette un maggiore disaccoppiamento tra i componenti del sistema: non devono sapere nulla l'uno dell'altro e possono eseguire i propri compiti senza simultaneità spaziale o temporale;

   2.1. L'implementazione del *tuple space* non utilizzerà nessun *framework* esterno, ma sarà parte del progetto. Questo permetterà di avere a disposizione un *middleware* che contemporaneamente utilizzi gli standard e le tecnologie più recenti nella sua implementazione e sia costruito sui bisogni di questo progetto;

   2.2. Il *tuple space* dovrà necessariamente essere reattivo, per notificare i *client* dell'avvenuto completamento delle primitive sospensive. Per questo motivo, dovrà essere impiegato il protocollo "Websocket" per mettere in comunicazione le due parti. Questo protocollo è particolarmente adeguato perché ampiamente supportato e particolarmente efficace ed efficiente nel contesto di architetture "client-server" dove entrambe le parti desiderano iniziare la comunicazione;

   2.3. Le macchine che ospiteranno il sistema devono essere tutte connesse via rete alla macchina che ospita lo spazio di tuple.

3. Il coordinamento tra gli *agent* dovrà essere regolato attraverso gli standard del consorzio "FIPA". L'uso di uno standard permette di rendere l'implementazione più chiara a chi non ha partecipato direttamente all'implementazione, ma anche più facile da manutenere ed evolvere in futuro;

4. La *User Interface* con cui interagirà l'utente sarà una *web application* implementata nel linguaggio JavaScript, lo standard *de facto* per questo tipo di applicazioni.

   4.1. La comunicazione tra sistema e *UI* avverrà attraverso l'uso del protocollo "Websocket" perché possa aggiornare l'utente in tempo reale sul completa-

mento delle richieste fatte. I vantaggi di questo protocollo sono già stati precedentemente elencati.

2.3 Assunzioni

Si assume che l'utente abbia a disposizione un computer per accedere al servizio dotato di un *web browser*, che sarà impiegato per utilizzare il sistema. Questo necessariamente implica che l'utente dovrà avere una connessione di rete stabile verso la macchina che ospita il componente principale del sistema o comunque capace di supportare l'*upload* di *file* eseguibili in tempo ragionevole. Inoltre, il *web browser* dovrà essere sufficientemente moderno da supportare il protocollo "Websocket".

Si assume inoltre che gli eseguibili forniti dall'utente al sistema debbano necessariamente terminare e non proseguire la loro computazione "per sempre". Questo deriva dal fatto che questo progetto si ispira a servizi *cloud* di tipo "Function-as-a-Service", i cui casi d'uso tipici non prevedono l'esecuzione di programmi che non terminano. È infatti consigliato l'utilizzo di questi servizi qualora si volesse costruire un sistema *event driven* a partire da funzioni *stateless* e *single purpose* di trasformazione dei dati. Un servizio implementato tramite FaaS che rimane perennemente in attività, non appena ne venisse sospesa l'esecuzione da parte del *service provider* in caso di mancato uso, perderebbe il suo stato, rendendo inutile la sua implementazione. Inoltre, questo tipo di servizi danno all'utente la possibilità di ridurre fortemente i costi quando il servizio non viene usato, costi ad esempio presenti in caso di *deployment* tramite servizi "PaaS", grazie all'esecuzione delle funzioni *on demand*. Questo implica che un eseguibile che non termina implementato tramite FaaS avrebbe costi monetari proibitivi.

3 Design

3.1 Struttura

L'architettura del sistema è composta da tre diverse componenti. Il primo, il più semplice, è il "*tuple space*". Questo è realizzato come un servizio che si compone di una "Java Virtual Machine", la quale mette in esecuzione l'"Actor System" che permette all'attore che ne implementa il comportamento di funzionare. Poiché l'attore rappresenta tutte le funzionalità dell'*API* fornita dallo spazio di tuple, a questo è stato dato un nome corrispondente. Questo attore si serve di un "*controller*" che fa da interfaccia verso i *client* di questo servizio, *controller* che quindi espone le diverse *route* disponibili. Non essendo però questa una ReST *API*, è esposta una sola *route* che permette di aprire Websocket da parte dei *client*.

I *client* del *tuple space* sono la "componente principale" e le "componenti secondarie". Le componenti secondarie possono essere più di una, dato che deve esserne presente una per ogni macchina che è parte dell'infrastruttura del sistema e queste possono essere diverse. L'architettura è simile a quella dello spazio di tuple, dove la differenza principale risiede nel fatto che le funzionalità sono diverse, perciò diversi sono gli attori che popolano il suo "Actor System". Sono presenti infatti due tipi di attori: il "Worker Agent",

che si occupa di implementare tutti i comportamenti necessari per il componente, e il "Runner Agent", che ha il compito di mettere in esecuzione un *file* eseguibile allocato sotto richiesta del WorkerAgent. In questo modo si replica in locale e più in piccolo l'intero sistema distribuito, scaricando il peso dell'esecuzione ad altri attori pensati solo per questo scopo.

Un'altra differenza con lo spazio di tuple è che una componente secondaria non è un *service*, perciò non ha *controller*, non espone *route* e quindi le sue funzionalità non sono esternamente accessibili. L'unico modo per poterlo contattare è passare appunto attraverso lo spazio di tuple, per questo motivo il componente deve dipendere da un modulo che faccia da *client* verso quest'ultimo. È dotato anche di un *client* HTTP generico per poter contattare direttamente la componente principale. Questo serve nel momento nel quale il componente è chiamato ad allocare un *file* perchè, per portare a termine il compito, deve necessariamente trasferirlo verso di sé. L'utilizzo del protocollo HTTP per supplire a questa necessità è particolarmente adeguato perché pensato per il trasferimento di *file* in maniera efficiente e allo stesso tempo ampiamente diffuso nel contesto di sistemi distribuiti sul web.

Da ultimo, l'architettura del sistema è composta dalla "componente principale". Anche in questo caso l'architettura della singola componente è simile a quella delle due già viste in precedenza. La similarità con il *tuple space* è però più evidente, dato che anche questo è un *web service* e ne segue quindi la struttura. È presente un solo attore all'interno della componente, che gestisce la sua *API*, il quale necessita di un *controller* per esporre le sue *route* verso l'esterno. Sicuramente tra queste c'è quella per il *download* dei *file* eseguibili da parte delle componenti secondarie. Deve però essere presente anche una *route* per permettere l'accesso alla *web app* che l'utente utilizzerà per accedere al sistema e che sarà in esecuzione sul suo *browser*. La comunicazione tra *web app* ed API avverrà attraverso l'uso di Websocket, quindi anche in questo caso non ci troviamo di fronte ad una ReST API. La *route* per accedere a tutte le funzionalità dell'*API* è quindi unica, esattamente come per lo spazio di tuple, utile per aprire Websocket di comunicazione. Anche l'attore che implementa il comportamento della componente principale si serve di un *client* per il *tuple space*, così come di un *client* per il servizio di *storage*, per poter interagire con essi. É infatti necessario anche l'impiego di una tecnologia di memorizzazione, nella fattispecie un *database*, per registrare le informazioni inerenti agli utenti del sistema. La comunicazione con questo avviene tramite protocollo "JDBC", tipicamente utilizzato nelle applicazioni Java per comunicare con i *database*.

La scelta di un'architettura basata sulla suddivisione tra componenti principali e secondari è stata profondamente ispirata da quella di altri *software* distribuiti, come quelli parte del *framework* "Hadoop". Se si osservano le architetture di "Hadoop Distributed File System" [7], di "Yet Another Resource Negotiator" [5], ma anche di *framework* a loro vicini come "Apache Spark" [6], si potrà osservare come siano sempre simili a quella proposta. La motivazione dietro ad una scelta simile è semplice. Dà la possibilità di mantenere il controllo dell'intero sistema in un'unica componente, demandando però i compiti più onerosi che rallenterebbero la sua capacità di servire le richieste a componenti secondari e locali alle macchine su cui devono essere portati a termine. In figura 2 si può osservare l'intera architettura appena descritta.

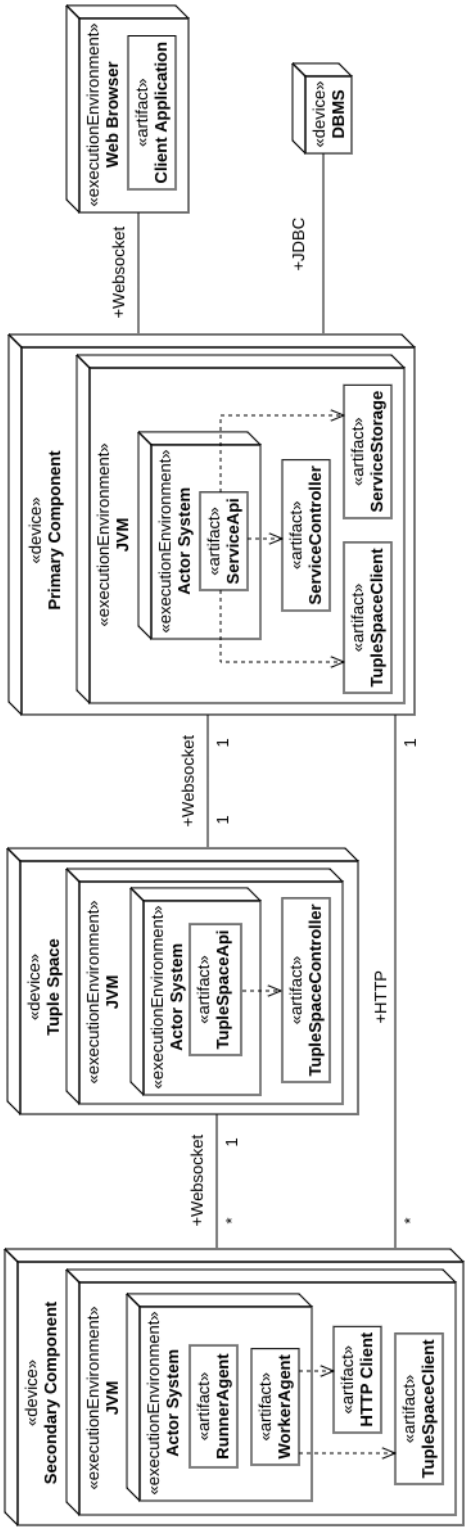

Figura 2: Diagramma di *deployment* che mostra i componenti principali del sistema

Il modello dei dati nel *database* è molto semplice, in quanto per un utente si trattiene solamente lo *username* e la *password*. Lo *username* funziona da *primary key*, in quanto questo valore non può essere lo stesso di un altro utente. Ad ogni utente possono essere associati zero o più eseguibili, a seconda di quanti ha deciso di allocare, ed ogni eseguibile è di proprietà di un solo utente. Ogni eseguibile è rappresentato da un identificatore e da un nome che l'utente ha scelto per esso, dove il primo, identificando appunto l'eseguibile, è unico per entità e rappresenta quindi la sua *primary key*. Questo modello può essere osservato in figura 3.

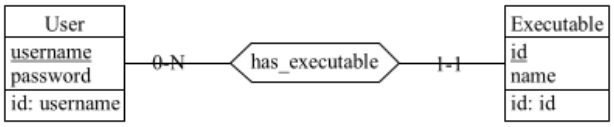

Figura 3: Diagramma "Entity-Relationship" per il modello di dati del *database*

La logica conseguenza è che questo venga tradotto in un modello relazionale composto da due relazioni identiche alle entità descritte in precedenza. L'associazione tra le due è modellata attraverso una *foreign key* in "Executable" che fa riferimento allo "User" a cui l'eseguibile appartiene. La trasformazione del diagramma E-R in quello relazionale è visibile in figura 4.

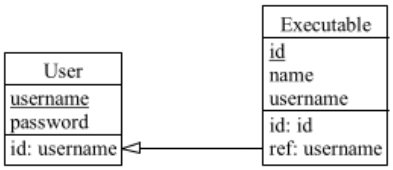

Figura 4: Diagramma relazionale per il modello di dati del *database*

Il modello delle componenti principale e secondaria non è particolarmente interessante. Tutti gli identificativi necessari, sia che fossero legati ad un *file eseguibile*, sia che fossero legati ad una richiesta di esecuzione o di allocazione, sono stati rappresentati da "*Universally Unique IDentifiers*". Questo ha fatto sì che la probabilità di collisioni tra questi fosse trascurabile, pur non essendo generati da un'unica sorgente. I restanti concetti sono stati modellati come *union* o *intersection type* necessari unicamente a trattenere i dati necessari. Questo è l'esempio di "ExecutionOutput", che trattiene l'*output* dell'esecuzione di un *file*. Questo è costituito solamente dall'*exit code*, dallo *standard output* e dallo *standard error* del processo di esecuzione, nulla più e nulla meno. Allo stesso modo "DeployedExecutable", che rappresenta un eseguibile che è stato allocato in passato, è un tipo composto dall'identificatore dell'eseguibile e dallo *username* dell'utente che ha richiesto il *deployment*. Tutte queste entità possono poi essere racchiuse in oggetti che rappresentano le richieste che la *web app* può rivolgere alla componente principale o in risposte che da quest'ultima vengono fatte pervenire alla prima.

Più interessante è invece il modello che sta dietro allo spazio di tuple. L'elemento centrale è appunto la tupla, vista come sequenza ordinata ed immutabile di suoi elementi, ognuno dei quali è rappresentato da un indice, che può essere anche vuota. Ogni elemento può essere contenuto in più tuple allo stesso tempo. Una tupla è rappresentata dall'entità "JsonTuple" perché si è voluto limitare gli elementi che possono essere contenuti al suo interno solamente a quelli il cui tipo è presente nella specifica JSON [1]. Sono perciò ammessi interi, interi lunghi, numeri in virgola mobile a singola e doppia precisione, stringhe, booleani e il valore "*null*". Per permettere maggiore flessibilità, una tupla può essere contenuta in un'altra tupla e così via ricorsivamente. Ognuno di questi tipi è quindi rappresentato dalla sua controparte nella specifica e una tupla è rappresentata da un "JSON Array".

L'altro concetto fondamentale è quello di "*template*", ovvero di scheletro da utilizzare per identificare se una tupla vi si può associare oppure no, se vi fa "*match*" oppure no. La definizione dei diversi tipi di *template* si basa sullo standard "JSON Schema" [8]. Questo perché è lo standard più diffuso e riconosciuto per la definizione di *schema*, quindi di *template*, per oggetti JSON. Poiché ogni tupla ed ogni elemento che essa contiene è un elemento JSON, questo significa che un *template* che segue tale standard sarà naturalmente in grado di fare *match* con questi, per definizione. I tipi di *template* definiti sono i seguenti:

- "JsonAnyTemplate", capace di fare *match* con qualsiasi elemento;
- "JsonNullTemplate", capace di fare *match* con il solo valore *null*;
- "JsonStringTemplate", capace di fare *match* con una stringa. È possibile specificare un insieme di valori ammissibili per la stringa, una lunghezza minima per essa, una lunghezza massima oppure una "regular expression" che rappresenti l'insieme delle stringhe accettate;
- "JsonBooleanTemplate", capace di fare *match* con un booleano. È possibile specificare il valore con cui fare *match*;
- "JsonTupleTemplate", capace di fare *match* con una JsonTuple. È possibile specificare sia i *template* per gli elementi che compongono la tupla, da zero a più di uno, sia se il *template* è "incompleto", ovverosia che sono stati specificati solamente i *template* per gli elementi iniziali e non tutti;
- "JsonNotTemplate", capace di fare *match* dove non fa *match* un *template* dato e viceversa;
- "JsonMultipleTemplate", il supertipo che raccoglie tutti i *template* che fanno *match* a partire da più template alla volta, ovvero:
  - "JsonAllOfTemplate", capace di fare *match* quando tutti i *template* dati fanno *match*;
  - "JsonAnyOfTemplate", capace di fare *match* quando almeno un *template* tra quelli dati fa *match*;

  - "JsonOneOfTemplate", capace di fare *match* quando esattamente un *template* tra quelli dati fa *match*.

- "JsonNumericTemplate", il supertipo che raccoglie tutti i *template* che fanno *match* con un valore numerico. Questo significa che permette di specificare un valore esatto per l'elemento numerico, oppure un valore minimo o uno massimo, i quali possono essere inclusivi o esclusivi. I *template* che da questo derivano sono:
  - "JsonFloatTemplate", capace di fare *match* con un numero in virgola mobile a precisione singola;
  - "JsonDoubleTemplate", capace di fare *match* con un numero in virgola mobile a precisione doppia;
  - "JsonIntegralTemplate", il supertipo che raccoglie tutti i *template* che fanno *match* con un numero intero. Rispetto a JsonNumericTemplate, questo permette anche di specificare il vincolo per il quale l'elemento intero deve essere il multiplo di un valore dato. I *template* che da questo derivano sono:
    * "JsonIntTemplate", capace di fare *match* con un numero intero;
    * "JsonLongTemplate", capace di fare *match* con un numero intero lungo.

In figura 5 si possono osservare le classi descritte e le relazioni tra queste.

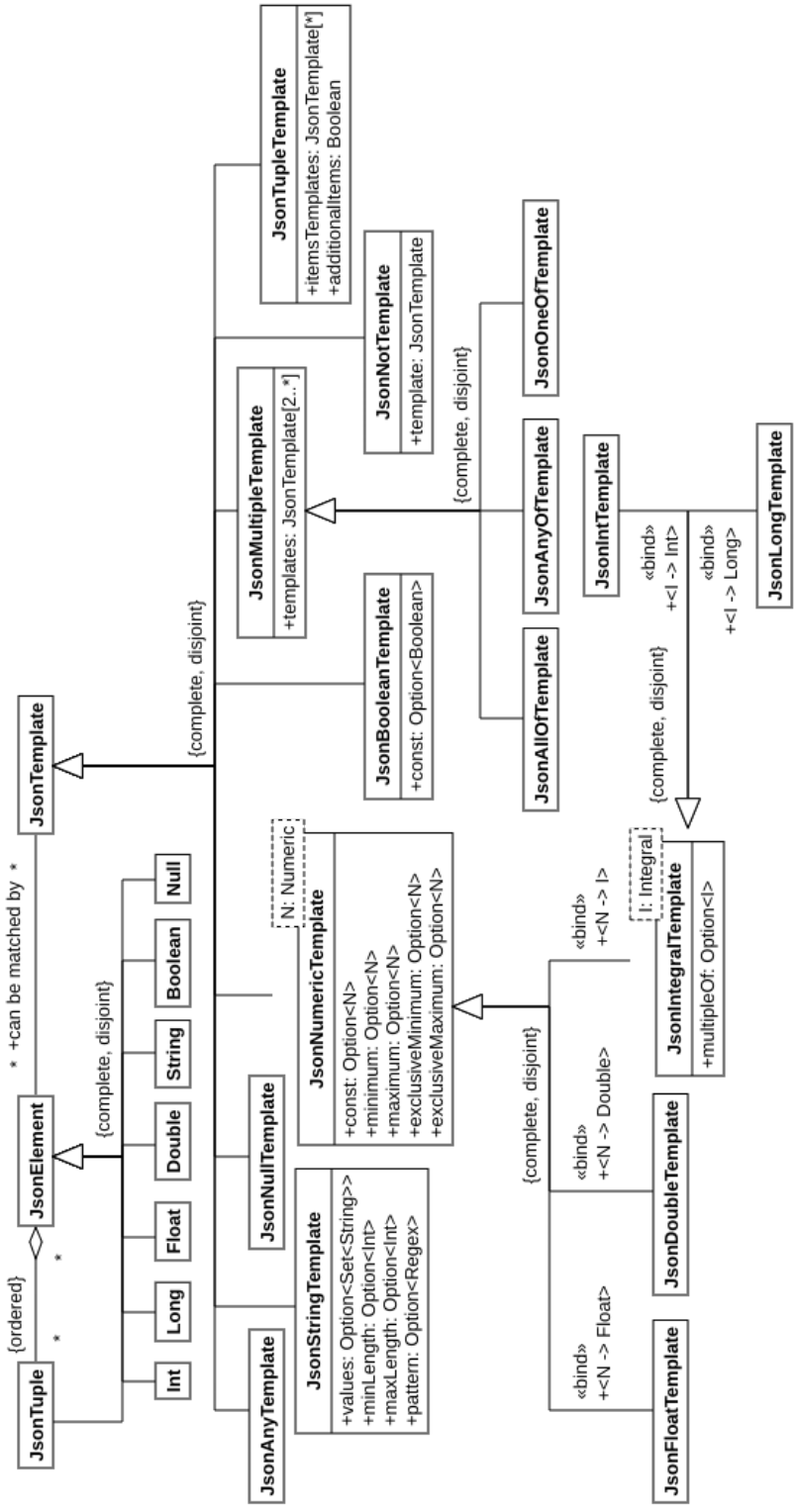

Figura 5: Diagramma delle classi che mostra i concetti propri del *tuple space*

Le tuple, così come i *template*, vengono poi racchiusi in degli oggetti che costituiscono le richieste e le risposte che *client* e spazio di tuple si scambiano. Tra queste, sicuramente la "meta-richiesta" più interessante è quella di unione di identificatori. Poiché potrebbe avvenire una disconnessione improvvisa tra *client* e *server*, non si vuole che lo stato che il primo mantiene sul secondo, ovvero le operazioni sospensive come "in" e "rd", vengano perse. Per questo motivo, all'apertura di ogni nuova connessione il *server* invia un identificatore univoco al *client*, che quest'ultimo memorizza. In caso di disconnessione involontaria, ad una successiva connessione il *client* ha la facoltà di inviare una richiesta speciale dove indica che il suo nuovo identificatore e quello vecchio sono in realtà lo stesso e vanno perciò "fusi" assieme. Il *server*, che in caso di disconnessione per errore non elimina le richieste pendenti, mentre in caso di disconnessione volontaria sì, sposterà tutte le richieste associate al vecchio identificatore sul nuovo. Il *client* è quindi dotato anche della facoltà di riconnettersi con "*exponential backoff*" in caso di disconnessione, per non sovraccaricare il *server* e generare un attacco di tipo "Denial-of-Service" involontario. L'identificatore è utilizzato dallo spazio di tuple anche per distinguere le richieste dei diversi *client* e fornire risposta a quello da cui la richiesta è pervenuta.

3.2 Comportamento

Il comportamento del *client* del *tuple space* e del *tuple space* vero e proprio non sono particolarmente rilevanti e non saranno quindi approfonditi. Il *client* si limita a inoltrare le richieste che gli vengono fatte e ad attendere le risposte correlate. Lo spazio di tuple è in perenne attesa di nuove richieste, che serve diversamente in funzione dell'operazione che esse contengono, inviando una risposta conclusiva. Unica particolarità è l'operazione di fusione degli identificatori, già discussa in precedenza, così come l'apertura e la chiusura di una connessione.

Maggiormente rilevanti sono i comportamenti che rappresentano l'implementazione dei diversi attori. Consideriamo inizialmente il RunnerAgent, l'agente il cui compito è mettere in esecuzione i singoli *file eseguibili*. Come si può osservare dalla figura 6, all'avvio l'agente invia un messaggio al WorkerAgent che l'ha generato per notificare la sua disponibilità ad accettare richieste. Specifica l'identificatore e il tipo dell'eseguibile assegnatogli per distinguersi dagli altri RunnerAgent che il WorkerAgent potrebbe aver avviato. Una volta entrato nello stato principale, attende dei messaggi di "esecuzione", costituiti da identificatore dell'operazione di esecuzione e dagli argomenti da passare all'eseguibile. Una volta ricevutone uno, l'agente lancia il processo di esecuzione del *file* con gli argomenti che gli sono passati. Una volta terminato il processo, l'agente raccoglierà gli *output* dello stesso e invierà un messaggio di completamento dell'esecuzione al suo WorkerAgent, messaggio costituito dall'identificatore dell'operazione e dagli *output* ottenuti.

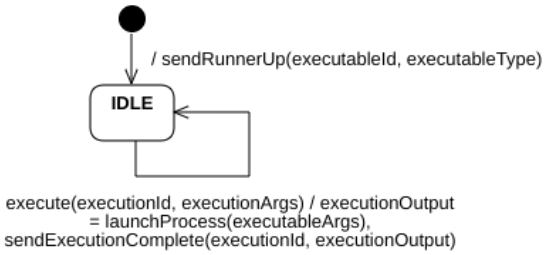

Figura 6: Diagramma degli stati dell'agente "RunnerAgent"

Passiamo quindi a considerare il comportamento del WorkerAgent, visibile in figura 7. La procedura d'avvio del WorkerAgent consiste nell'ottenere tutti gli eseguibili che già sono stati allocati sulla macchina in cui si trova. Questo viene fatto perché l'agente potrebbe essersi spento a causa di un malfunzionamento e, al suo riavvio, deve ripristinare il suo stato precedente allo spegnimento. Nel caso in cui non avesse allocato nessun eseguibile, passa direttamente allo stato principale. Se c'era almeno un eseguibile già allocato, viene avviato un RunnerAgent per ogni eseguibile e si attende il corrispondente messaggio di avvio. Ad ogni nuovo messaggio, si controlla se sono stati avviati tutti gli agenti. Se no, si attende incrementando un contatore apposito, se sì, si passa nello stato principale, non prima però di avere richiesto allo spazio di tuple la ricezione del prossimo messaggio di "request" di esecuzione per ogni RunnerAgent. Ogni messaggio "request" viene richiesto passando il corrispondente identificatore dell'eseguibile per cui quel messaggio è stato generato. Il nome scelto per esso non è casuale, in quanto le operazioni che verranno seguite dalle parti coinvolte ricalcano quelle del protocollo "request" [3], uno standard emanato dal consorzio "Foundation for Intelligent Physical Agents".

Una volta nello stato principale, l'agente richiede allo spazio di tuple di ricevere la prossima "call-for-proposal" per l'allocazione di un eseguibile, passando l'insieme di tipologie di eseguibili accettati. Anche in questo caso, il fatto che sia denominata "call-for-proposal" non è una casualità: la richiesta di allocazione di un eseguibile segue il protocollo "Contract Net" [2] standardizzato dal consorzio "FIPA". Dopodichè, il WorkerAgent si mette in attesa delle richieste da parte della componente principale mediante il *tuple space*. Nel caso in cui ricevesse una richiesta di esecuzione da parte dello spazio di tuple, l'agente inoltra la richiesta al RunnerAgent competente, passando l'identificatore dell'esecuzione e gli argomenti dell'esecuzione. In seguito, richiede allo spazio di tuple la prossima richiesta di esecuzione per lo stesso eseguibile, in modo da poterla gestire. Una volta ricevuto il messaggio di completamento dell'esecuzione, si controlla qualora l'esecuzione abbia avuto successo oppure no. In caso l'abbia avuto, dall'*output* dell'esecuzione si estraggono *exit code*, *standard output* e *standard error* e si utilizzano assieme all'identificatore dell'operazione per inviare il messaggio di completamento con successo nel *tuple space*. Questo messaggio, per seguire lo standard "request", avrà una *performative* di tipo "inform-result". In caso di completamento con fallimento, si invia un messaggio nello spazio di tuple composto da identificatore dell'operazione di esecuzione

e un messaggio d'errore, il quale avrà performativa "failure".

Nel caso in cui ricevesse una "call-for-proposal", il WorkerAgent controlla se il numero di *slot* disponibili, quindi di spazi per allocare eseguibili, è superiore a zero. Se sì, invia la sua candidatura riutilizzando l'identificatore dell'asta, assieme al suo identificatore di agente e al numero di *slot* disponibili, inviando un messaggio la cui performativa è "propose". Altrimenti, invia un messaggio nello spazio di tuple con performativa "refuse", che contiene l'identificatore dell'asta. Nel caso in cui l'agente venisse scelto come candidato per l'allocazione, riceverà un messaggio dallo spazio di tuple con performativa "proposal-accepted" contenente ancora una volta l'identificatore dell'asta, assieme all'identificatore e al tipo dell'eseguibile. Poiché l'agente potrebbe partecipare a più di un'asta concorrentemente, un'operazione potrebbe avere ridotto gli *slot* disponibili, quindi vengono nuovamente controllati. Se sono ancora superiori a zero, l'agente tenta la creazione del RunnerAgent che si occuperà di mettere in esecuzione l'eseguibile. Nel fare questo, gli fornirà l'identificatore e il tipo dell'eseguibile: così potrà ricostruire il nome del *file* come salvato sulla macchina locale. Da ultimo, si decrementano gli *slot* perché un altro è stato quindi occupato. Se invece non è più presente alcuna disponibilità, viene inviato un messaggio sullo spazio di tuple con performativa "failure" contenente l'identificativo dell'asta. Quando il WorkerAgent riceve il messaggio di completamento dell'avvio del RunnerAgent appena lanciato, contenente identificatore e tipo dell'eseguibile associato, fa partire il *download* del *file* dalla componente principale, utilizzando le stesse informazioni. Se la copia dell'eseguibile da remoto fallisse, viene fermato il RunnerAgent, incrementato il numero degli *slot*, perché ora un nuovo eseguibile può essere allocato, e infine mandato un messaggio di "failure" con l'identificatore dell'asta. Se invece il *download* ha successo, si invia allo spazio di tuple un messaggio con performativa "inform-done" per notificare dell'avvenuto successo, così come un messaggio di attesa di nuove richieste di esecuzione per il nuovo eseguibile.

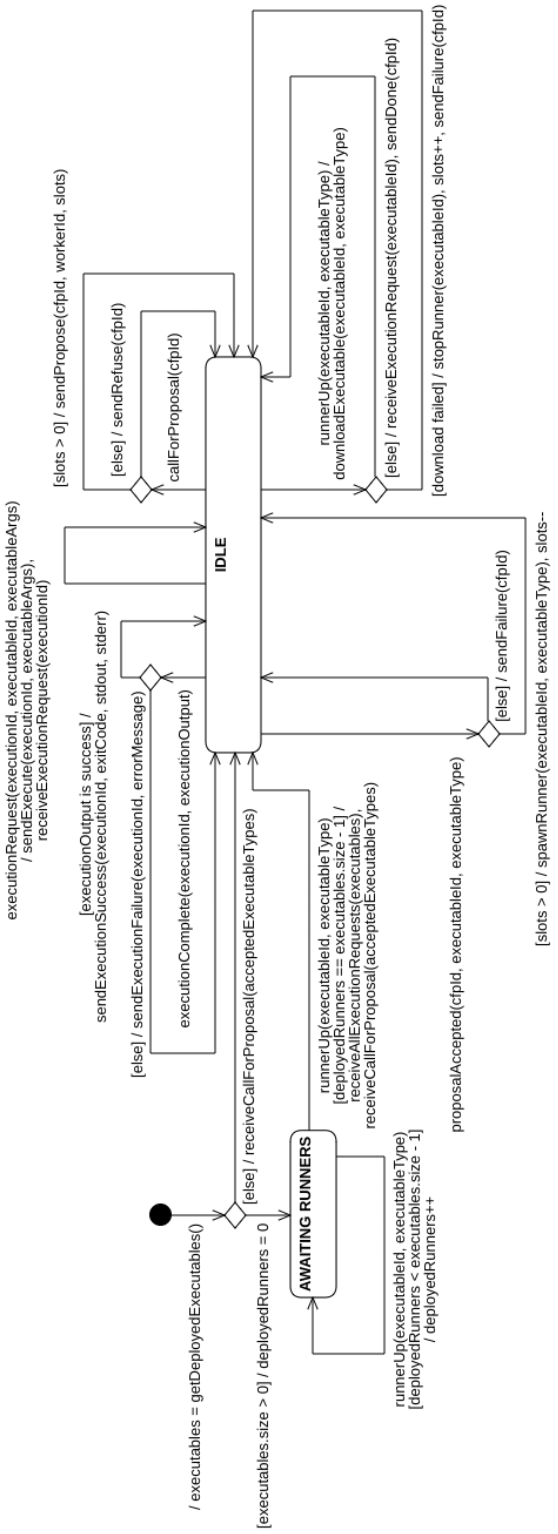

Figura 7: Diagramma degli stati dell'agente "WorkerAgent"

Il comportamento dell'agente "Service", quello che gestisce il servizio vero e proprio all'interno della componente principale, è leggermente più complesso.

Quando riceve una richiesta di apertura di connessione da parte di un *web client*, crea una nuova sessione generando un identificatore specifico. In questo modo, saprà sempre da chi proviene un determinato messaggio. Una volta che la connessione sarà chiusa, anche la sessione verrà eliminata cancellando l'identificatore associato. Ogni richiesta che verrà fatta da parte del *web client* fornirà quindi anche l'identificatore di sessione oltre alle informazioni che porta con sé.

Nel caso in cui venga richiesta la registrazione di un utente, verranno forniti dalla *web app username* e *password*. L'agente farà quindi in modo che questi dati vengano salvati sul *database*, venga inviato un messaggio di risposta alla richiesta contenente lo stato attuale dei dati dell'utente e venga fatto partire un *timer* di sessione. Questo *timer* ha una durata di un giorno e ha come scopo quello di permettere all'utente di ricaricare la pagina, o comunque di chiudere e riaprire il *browser*, e vedere la propria sessione mantenuta attiva. Per ragioni di sicurezza però, al termine della durata del *timer*, i dati in memoria dell'utente vengono rimossi e occorre fare nuovamente *login* per accedere all'applicazione.

Nel caso in cui la richiesta che perviene dal *web client* sia di *login*, i dati che la accompagneranno saranno di nuovo *username* e *password*. Nel caso in cui i dati non siano validi, la risposta conterrà un messaggio d'errore. Se invece corrispondono ad un utente precedentemente registrato, la risposta conterrà i dati dell'utente allo stato attuale e verrà riavviato il *timer* di sessione. Nel caso in cui non fosse mai stato avviato, semplicemente viene fatto partire.

Se la richiesta che proviene dalla *web app* è per ottenere lo stato dell'utente, vuol dire che nella *cache* del *browser* è stato salvato un vecchio identificatore di sessione legato ad una connessione precedente. In questo caso, questo viene passato al ServiceApi e, nel caso in cui non fosse associato a nessuna sessione, l'agente invia come risposta un messaggio di errore. Se invece l'identificatore corrisponde ad una sessione in cui un utente aveva effettuato il *login*, vengono restituiti i suoi dati allo stato corrente. Il meccanismo per il mantenimento della sessione nel *browser* si basa perciò proprio su questa richiesta per evitare il *login* continuo.

L'ultima richiesta che l'utente può inviare inerente ai meccanismi di accesso del servizio è quella di *logout*. A fronte di questa richiesta, semplicemente viene fermato il *timer* associato alla sessione dell'utente, in quanto non ha più accesso all'applicazione.

Nel caso in cui venisse richiesta l'esecuzione di un *file* eseguibile, viene prima controllato se l'utente abbia fatto il *login* in precedenza e se l'identificatore dell'eseguibile è tra quelli a lui associati. In caso negativo, la risposta consisterà in un messaggio d'errore assieme all'identificatore del *file*. In caso positivo, verrà inoltrata una richiesta di esecuzione al RunnerAgent responsabile per quel *file* passando, oltre all'identificatore, anche l'identificatore dell'operazione e gli argomenti con cui lanciare l'esecuzione. In caso di successo dell'operazione, viene risposto alla richiesta con un messaggio che è fatto dall'identificatore del *file*, l'*exit code*, lo *standard output* e lo *standard error* del processo di esecuzione. Se invece è fallita, viene di nuovo inviato un messaggio di errore.

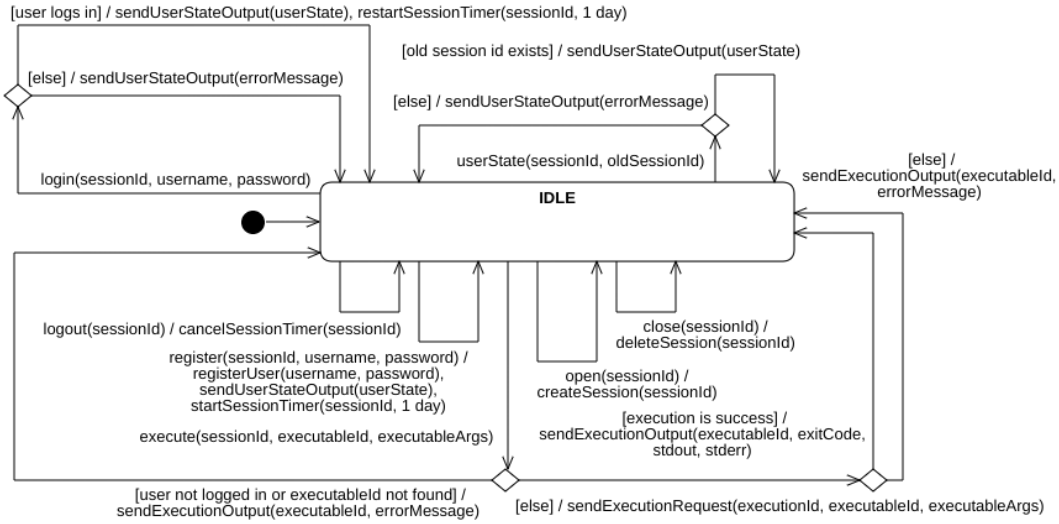

Figura 8: Diagramma degli stati che rappresenta il comportamento dell'agente "ServiceApi" senza quello relativo all'allocazione di eseguibili

Il comportamento che un agente ServiceApi mette in atto nel momento nel quale la *web app* richiede l'allocazione di un nuovo *file* è invece più complesso. Tutto inizia con una richiesta di allocazione, costituita dall'identificatore dell'eseguibile, il suo tipo e il nome che l'utente ha scelto per esso. La richiesta può essere portata avanti solamente se l'utente ha precedentemente fatto *login*, in tal caso viene inviato nel *tuple space* un messaggio con performativa "call-for-proposal", un identificatore generato sul momento e il tipo dell'eseguibile. Il tipo dell'eseguibile è necessario perché possano candidarsi solamente i WorkerAgent capaci di supportarlo. Dopodiché, viene avviato un *timer* di cinque secondi per la richiesta. Nel caso in cui non fosse stato effettuato nessun *login* preventivo, la richiesta viene completata con un messaggio d'errore.

Allo scadere del *timer*, viene rimosso dallo spazio di tuple il messaggio che bandiva la nuova asta, in quanto non si accetteranno da ora in poi nuove proposte di partecipazione. Inoltre, vengono ottenute tutte le proposte che sono state inviate dagli agenti che si sono candidati per l'allocazione e tra queste viene scelta la migliore. Nel caso in cui venga trovata una proposta ottima, viene inviato un messaggio con performativa "accept-proposal" contenente l'identificatore dell'asta, dell'agente scelto cosicché la possa ricevere solo lui, l'identificatore dell'eseguibile e infine il tipo dell'eseguibile. Questi ultimi due parametri permettono di identificare il *file* di cui il WorkerAgent necessita. Nel caso in cui venisse ricevuto dal ServiceApi un messaggio di risposta di fallimento dell'operazione di allocazione, elimina l'agente scelto dalla rosa dei potenziali candidati e sceglie il secondo ottimo, ripetendo il procedimento precedente. Nel caso in cui una soluzione ottima non venisse trovata, ad esempio perché non ci sono più candidati rimanenti, viene completata la richiesta inviando al *web client* un messaggio d'errore. Nel caso in cui

si ricevesse invece dal WorkerAgent candidato un messaggio di successo dell'operazione di allocazione, con performativa "inform-done", vengono innanzitutto aggiornati i dati dell'utente aggiungendogli il nuovo eseguibile, includendo identificatore e nome del *file*. A tutti gli altri agenti rimanenti viene inviato un messaggio con performativa "reject-proposal", perché le loro proposte non sono più necessarie. Infine, viene completata la richiesta presso la *web app* con l'identificatore del nuovo eseguibile allocato.

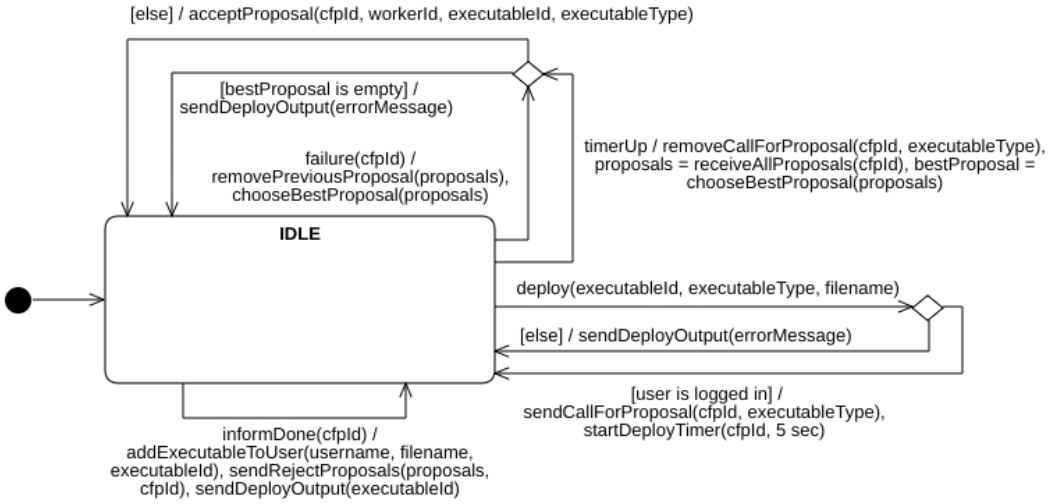

Figura 9: Diagramma degli stati che rappresenta il comportamento dell'agente "ServiceApi" con solamente quello relativo all'operazione di allocazione

3.3 Interazioni

Le interazioni più significative sono quelle che riguardano le operazioni di esecuzione e di *deployment* di un eseguibile, in quanto sono le più complesse.

Per quanto riguarda l'esecuzione, tutto inizia con l'avvio del WorkerAgent, che controlla sul *file system* locale se sono presenti dei *file* eseguibili. In tal caso, per ognuno di essi deve lanciare un nuovo RunnerAgent fornendo identificatore dell'eseguibile e tipo dell'eseguibile ricevuti dal *file system*. Inoltre, per ognuno di essi deve attendere la conferma del loro avvio, decrementando di volta in volta il numero di agenti che sta aspettando. Infine, per ogni identificatore di eseguibile chiede allo spazio di tuple di ricevere la prossima richiesta di esecuzione corrispondente. Qualora non avesse trovato alcun eseguibile e non avesse quindi lanciato nessun agente, le operazioni precedenti vengono saltate.

Nel momento nel quale l'utente attraverso l'interfaccia della *web app* effettua una richiesta di esecuzione, specificando identificatore del *file* e argomenti, quello che fa l'*API* è controllare presso lo *storage* se il primo è associato all'utente oppure no. In caso negativo, la richiesta viene completata con un messaggio d'errore. In caso positivo, la

richiesta viene inoltrata allo spazio di tuple, che permette al WorkerAgent di poterla quindi ricevere. Attraverso delle strutture dati interne, l'agente risale all'identificatore del RunnerAgent che si occupa di quell'eseguibile, cioè l'"ActorRef" dell'agente, che può usare per inoltrare a lui la richiesta. Il RunnerAgent lancia quindi il processo per l'esecuzione del *file* con gli argomenti dati e ottiene da questo un *output*, che viene inviato tramite un messaggio apposito al WorkerAgent. Nel caso in cui l'*output* rappresenti il fallimento del processo, questo conterrà un messaggio di errore, che l'agente invierà allo spazio di tuple, cosicché possa essere ricevuto dal ServiceApi e inviato poi alla *web app*. Nel caso in cui il risultato rappresenti il successo dell'operazione, la procedura seguita è la stessa, ma vengono inviati *exit code*, *standard output* e *standard error* del processo di esecuzione.

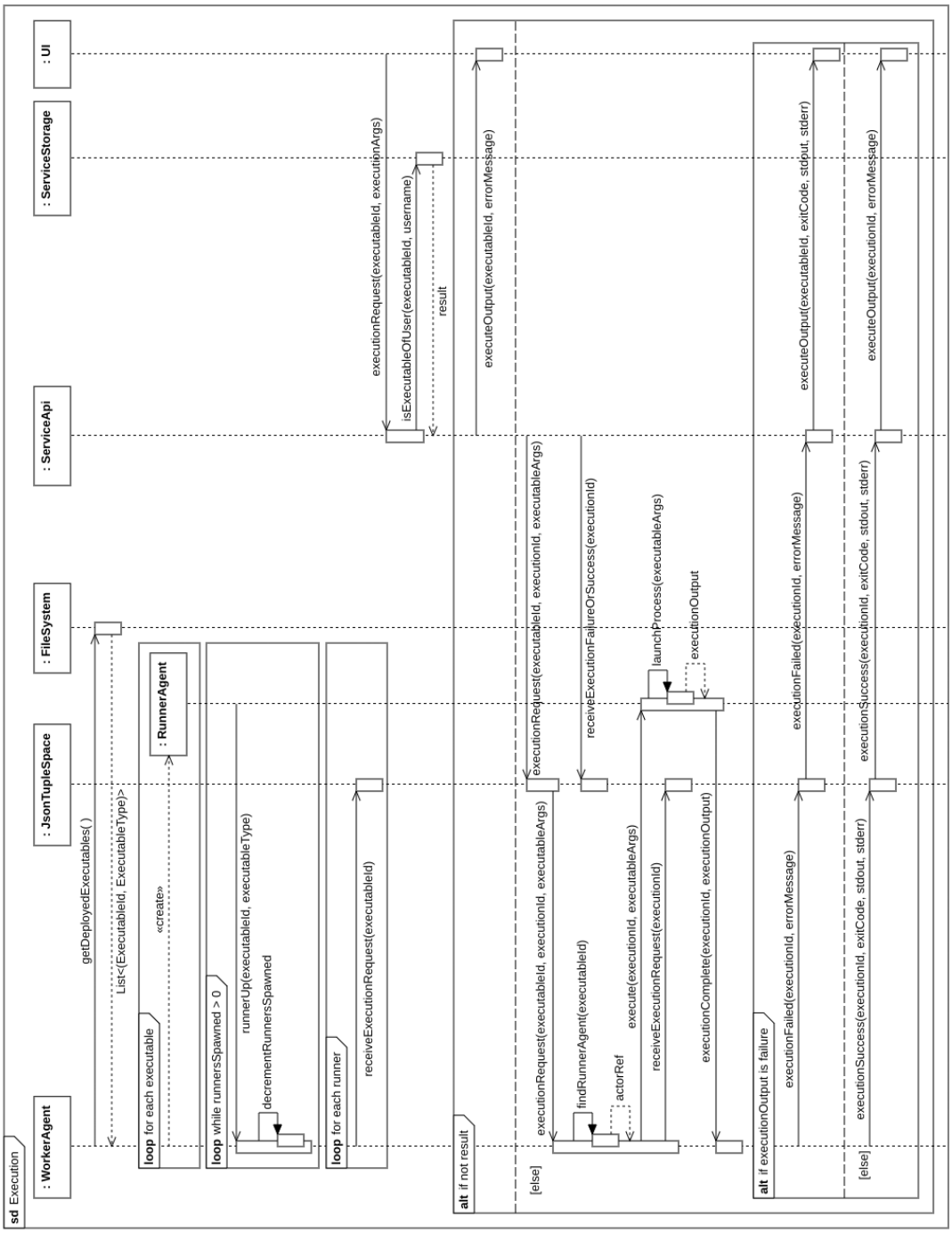

Figura 10: Diagramma di interazione che descrive l'operazione di esecuzione

Per quanto riguarda l'allocazione di un nuovo eseguibile, osserviamo inizialmente il solo punto di vista di un componente secondario. L'agente che ne implementa il comportamento invia innanzitutto una richiesta per ricevere la prossima "call-for-proposal" dal *tuple space*, indicando i tipi di eseguibili che è disposto ad accettare. In questo modo, riceverà solamente dallo spazio di tuple le informazioni sulle aste a cui ha possibilità di partecipare. A questo punto, riceverà una copia della prossima "call-for-proposal" adatta a lui, assieme all'identificatore che la contraddistingue. Se non ha più *slot* per l'allocazione di eseguibili a disposizione, risponderà con un messaggio "refuse", se invece ne ha ancora disponibili si candiderà per l'asta inviando un messaggio "propose". Questo messaggio, oltre a contenere l'identificatore dell'agente stesso perché possa ricevere risposta, conterrà anche gli *slot* disponibili così che la componente principale possa giudicare quale tra gli agenti candidati è il più adatto. Fatto questo, invia nuovamente una richiesta per ottenere messaggi con performativa "call-for-proposal", così da poter ricevere anche la successiva e così via, finché l'agente è in vita. In seguito, il WorkerAgent invierà una richiesta allo spazio di tuple per ricevere la risposta alla sua candidatura, che potrebbe essere una "accept-proposal" o una "reject-proposal". Nel caso in cui la sua proposta sia stata rifiutata, non rimane altro da fare che ricominciare il processo da capo, non appena una nuova asta verrà bandita. Nel caso in cui venisse accettata, gli verranno forniti l'identificatore e il tipo dell'eseguibile per poter ottenere il *file* dalla componente principale. A questo punto viene nuovamente controllato se il numero di *slot* è pari o superiore a zero, perché essendo la ricezione dei messaggi un'operazione asincrona, potrebbero essere stati ricevuti altri messaggi concorrentemente. Tra questi, potrebbero esserci stati messaggi inerenti ad altre aste e perciò il numero di *slot* potrebbe essere stato decrementato tra l'invio della proposta e la ricezione dell'accettazione. Nel caso in cui il numero di posti sia zero, viene inviato un messaggio di "failure". In caso contrario, viene creato un nuovo RunnerAgent, responsabile per l'eseguibile, e per poter proseguire si attende che invii il suo messaggio di conferma di avvio, il quale contiene le informazioni che garantiscono che sia lui ad essersi avviato. Dopodiché, il WorkerAgent procede ad effettuare il *download* del *file* eseguibile dalla componente principale, unica interazione diretta tra i due. Nel caso in cui il trasferimento dovesse fallire, viene inviato un messaggio allo spazio di tuple di tipo "failure". Altrimenti, viene inviato il messaggio per ricevere la prossima richiesta di esecuzione per l'eseguibile appena allocato, visto che è stato possibile farlo con successo. La sequenza di operazioni termina con un messaggio con performativa "inform-done" inviato allo spazio di tuple.

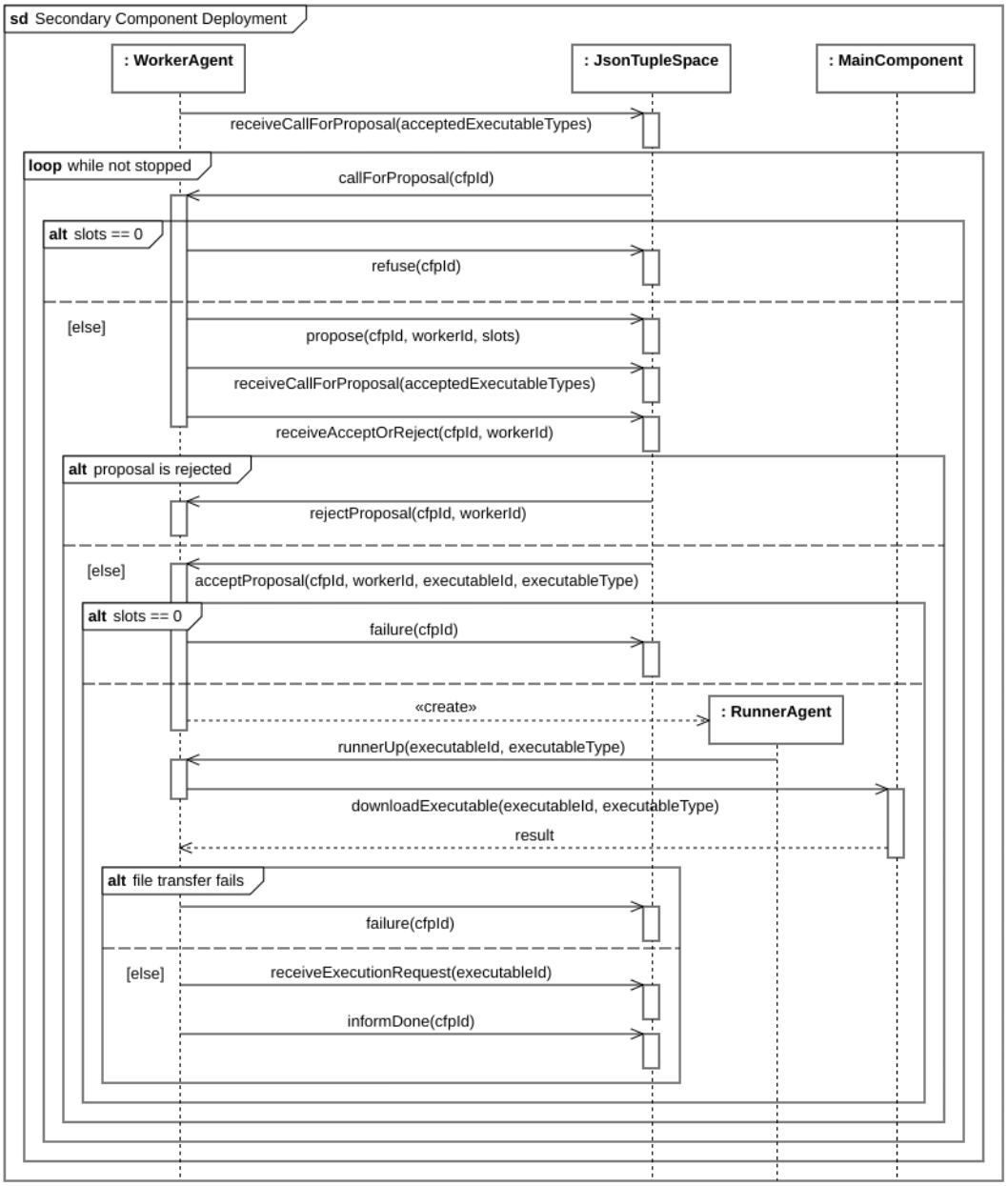

Figura 11: Diagramma di interazione che descrive l'operazione di *deployment* dal punto di vista di un componente secondario

Per quanto riguarda la componente principale, tutto comincia con una richiesta da parte della *UI* di effettuare il *deployment* di un eseguibile, fornendo identificatore, tipo e nome dello stesso. A questo punto l'*API* del servizio, ovvero l'agente ServiceApi,

genererà un nuovo identificatore per l'asta, diverso da qualsiasi altro sia già stato utilizzato in passato. Una volta fatto, invia un messaggio di tipo "call-for-proposal" con l'identificatore generato e il tipo di eseguibile che deve venire allocato, in modo che tutti i WorkerAgent interessati a rispondere e capaci di farlo si candidino. A questo punto, l'agente attende cinque secondi, in modo da lasciare il tempo necessario ai WorkerAgent di vedere la richiesta e di inviare le proprie proposte. Passato il tempo stabilito, l'agente chiede al *tuple space* di ricevere tutte le risposte relative al bando dell'asta, il quale prontamente risponde. Vengono rimosse tutte le risposte che rappresentano un rifiuto a partecipare, dopodiché viene applicato un algoritmo per determinare la migliore proposta tra tutte. Se non c'è una proposta migliore, ad esempio perché nessun WorkerAgent si è candidato, viene restituita una risposta contenente un messaggio di errore alla *UI*. Altrimenti, viene inviato un messaggio con performativa "accept-proposal" all'agente che ha fatto la proposta migliore, passando anche identificatore e tipo dell'eseguibile da allocare. A questo punto, all'*API* non rimane che rimanere in attesa del completamento del *deployment* notificato dal WorkerAgent tramite apposito messaggio.

Come primo *step* dell'allocazione, il WorkerAgent chiederà al ServiceApi di poter fare il *download* del *file* eseguibile mediante i riferimenti che ha ricevuto, poi continuerà con la procedura già descritta in precedenza. Nel caso in cui dovesse fallire, l'agente invierà un messaggio di tipo "failure" al ServiceApi e quest'ultimo si vedrà costretto a ripetere la procedura di scelta della proposta migliore eliminando quella già esplorata. Nel caso in cui il WorkerAgent completi con successo, invierà un messaggio di tipo "inform-done". Il ServiceApi dovrà quindi associare l'eseguibile all'utente nel sistema di *storage* sapendo lo *username* e l'identificatore e il nome del *file*. Una volta fatto, invia un messaggio con performativa "reject-proposal" a tutti gli altri agenti candidati, in quanto il loro servizio non è più necessario e notifica la *web app* che l'allocazione è avvenuta con successo indicando l'identificatore dell'eseguibile.

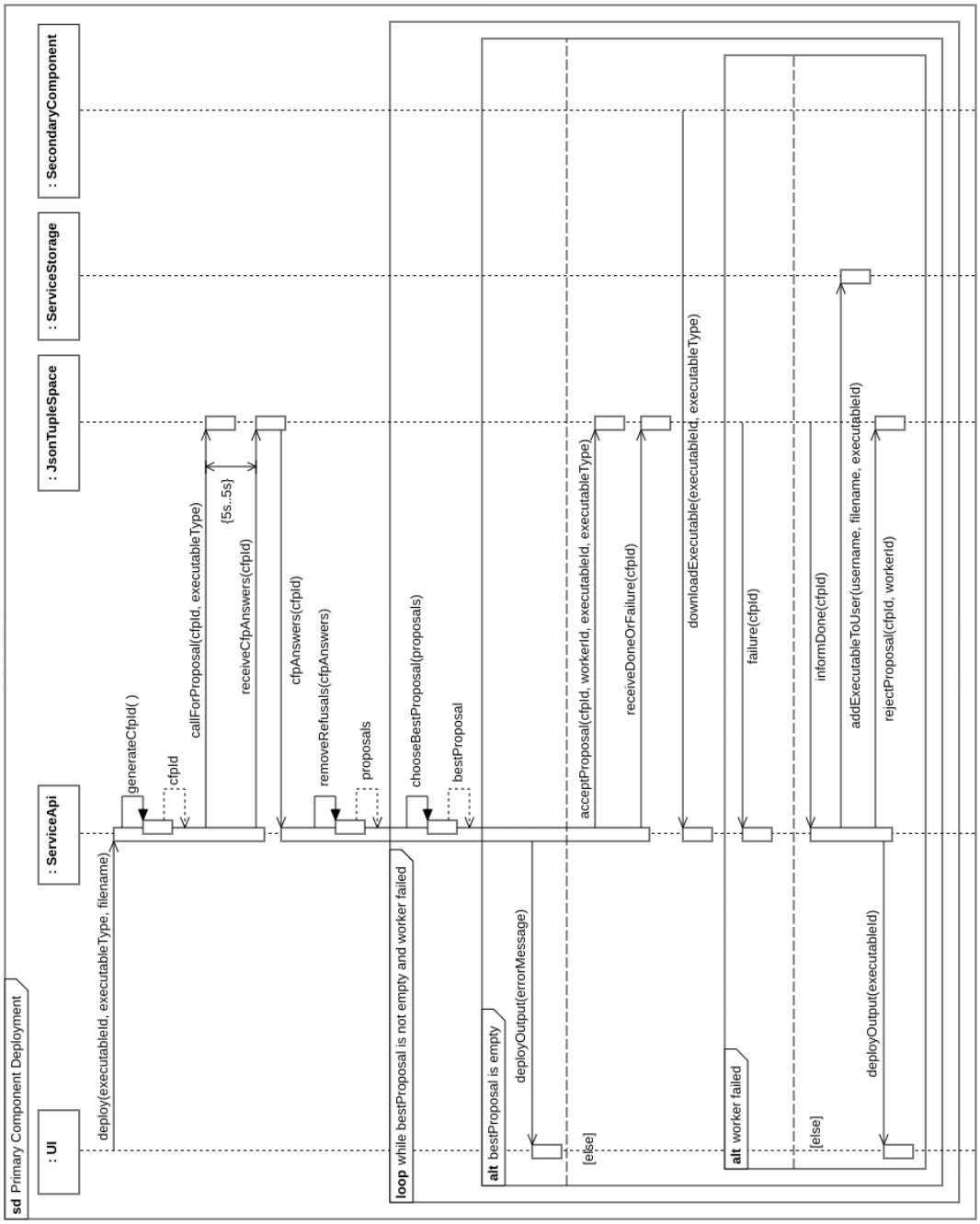

Figura 12: Diagramma di interazione che descrive l'operazione di *deployment* dal punto di vista di un componente principale

4 Dettagli implementativi

Il progetto è stato suddiviso in sei *submodule*, in modo tale che venisse applicato il principio di *separation of concerns* anche a livello di *repository*. In questo modo, il codice rimane ben separato sulla base delle aree funzionali di appartenenza, limitando la possibile confusione e migliorando la sua comprensione e quindi la sua manutenzione ed estensibilità. Chiaramente, questo tipo di approccio porta inevitabilmente a delle ridondanze in quel codice che viene utilizzato da più funzionalità, ma si è giudicata questa ridondanza minimale, tanto da poter essere trascurata. I *submodule* nel progetto sono sei e sono i seguenti:

- "core", contiene le entità di dominio per quanto riguarda lo spazio di tuple. Quindi mantiene solamente l'implementazione di tuple, *template* e del *domain specific language* pensato per un loro più facile utilizzo;
- "client", contiene il *client* per interagire con lo spazio di tuple, quindi un'entità che si occupa di fare da tramite tra il *tuple space* remoto, cioè quello vero e proprio, e ciò che sta usando il *client*. Oltre a questo sono presenti le implementazioni per le richieste che il *client* può rivolgere e le risposte che può ricevere;
- "server", contiene l'implementazione del *tuple space* remoto, l'entità che lo realizza propriamente e al quale si accede tramite *client* apposito. Come per il *submodule* precedente, contiene anche l'implementazione delle richieste che possono essergli rivolte e delle risposte corrispondenti;
- "worker", contiene l'implementazione della componente secondaria, quindi dell'agente che si occupa di gestire le singole macchine su cui il sistema è distribuito, permettendo l'allocazione e l'esecuzione dei *file*. L'implementazione coinvolge sia l'attore che rappresenta la componente principale vera e propria, sia l'attore che viene lanciato per ogni eseguibile da mettere in esecuzione, nonché tutti gli elementi di dominio;
- "master", contiene l'implementazione della componente principale, quindi dell'agente che si occupa di inoltrare le richieste alle diverse componenti secondarie sulla base delle proprie scelte. Contiene sia l'attore che implementa l'agente, che gli elementi di dominio utili per i suoi compiti, gli stessi del modulo precedente;
- "ui", contiene l'implementazione della *web app* che fa da interfaccia verso l'utente e che permette di accedere alle funzionalità del sistema.

Lo *storage* utilizzato dalla componente principale per trattenere i dati d'utente è il *DBMS* "PostgreSQL". Il suo utilizzo è stato giudicato particolarmente adatto data la sua natura di *software* "*open source*", la sua ricca documentazione e la disponibilità di una sua versione containerizzata rilasciata dai manutentori ufficiali del progetto.

Lo sviluppo è avvenuto seguendo i principi del "trunk-based development", una tecnica di gestione delle *repository* pensata per supportare al meglio le tecniche di *CI/CD*. Questo

modello infatti prevede il mantenimento di un *branch* "main" su cui sono presenti le versioni rilasciate del sistema, non diversamente da quanto previsto nel più classico "Git Workflow". La differenza si ha nel fatto che tutti gli altri *branch* scompaiono, sostituiti da *branch* pensati per singole *feature* e che perciò necessariamente sono di breve durata. La variante applicata in questo progetto ha previsto l'introduzione di un *branch* "beta" per ospitare i rilasci di tutte le versioni precedenti alla "1.0.0", che altrimenti non seguirebbero rigidamente lo standard di "semantic versioning" usato per numerarle. Il *branch* "beta" è stato utilizzato nello stesso modo di "main" e una volta ottenuta la prima versione ufficiale, si è provveduto al suo *merge* con "main", continuando lo sviluppo su quest'ultimo. Questa tecnica ha effettivamente permesso una più semplice gestione degli aspetti di *continuous integration*, permettendo di automatizzare i test sui "*feature branch*" e di automatizzare le *release* quando questi venivano riuniti con il flusso di sviluppo principale.

La *build automation* è stata permessa dall'uso di "Scala Build Tool", già usato per la suddivisione in *submodule* del progetto, assieme ai "GitHub Workflow" e le loro "GitHub Action". Il *workflow* di test prevede una fase iniziale di *linting* del codice grazie ai *tool* "scalafmt", "scalafix" e "wartremover", quest'ultimo già applicato in fase di compilazione. Dopodiché vengono lanciati i test con registrazione della *coverage*, che viene poi caricata sul sito "codecov" per l'analisi, oltre che su "sonarcloud" per ottenere una valutazione sulla qualità del codice. Il *workflow* di *release*, dopo avere lanciato quello di test ed essersi assicurato che completi con successo, si preoccupa di utilizzare il *tool* "semantic release" per analizzare i *commit*, il cui messaggio è sempre in formato "Angular conventional commit". Questo permette di generare le *release notes*, il *file* di *changelog*, calcolare automaticamente la nuova versione da rilasciare e sostituirla dove serve. Una volta fatto, viene utilizzato il *plugin* per SBT "sbt-assembly" per generare i "fat JAR" che saranno utilizzati per creare le immagini Docker dei componenti del sistema, operazione che verrà fatta attraverso il *plugin* per SBT "sbt-docker". I JAR e le corrispondenti immagini che vengono create sono tre: una per mettere in esecuzione lo spazio di tuple, una per la componente principale e una per la componente secondaria. La generazione dell'immagine per il componente principale richiede anche la compilazione della *web app* realizzata tramite il *framework* "React". Dopo aver caricato le immagini su "DockerHub", viene creata la *release* anche su GitHub contenente i tre JAR già citati. Il *workflow* si conclude con la generazione della scaladoc e la sua pubblicazione come sito statico tramite "GitHub Pages". La sua ultima versione è sempre disponibile al seguente link.

Le immagini Docker caricate su DockerHub serviranno poi a realizzare la demo, che viene lanciata tramite un *file* di specifica "docker compose". Questo crea un sistema prototipale costituito da una componente principale, un *tuple space* e due componenti secondarie, che pur essendo sulla stessa macchina, si trovano in *container* Docker differenti e perciò si comportano come se il sistema fosse distribuito. In aggiunta, viene anche lanciato un *container* contenente il *database* PostgreSQL, così che possa essere utilizzato dalla componente principale, le cui tabelle vengono create tramite apposito *script* SQL. Per il *container* dello spazio di tuple è stato definito un "*healthcheck*" in modo tale che gli altri servizi vengano fatti partire solamente quando questo ha successo,

ovvero quando la porta sulla quale avvengono le comunicazioni è stata aperta. Il *file* inoltre specifica per ogni servizio le corrette variabili di ambiente di cui ha bisogno e per le componenti secondarie definisce anche un *volume* per ciascuna. In questo modo, in caso di riavvio del *container*, i *file* che sono già stati allocati non vengono persi. Da ultimo è bene osservare che solamente per la componente principale viene aperta una porta verso l'*host*, quella che sarà utilizzata per accedere dal *browser* al *web service*.

5 Validazione e testing

Si è adottato un paradigma di sviluppo "*test-driven*" che integrasse al suo interno la verifica dell'aderenza ai requisiti prefissati, nonché della correttezza del sistema sviluppato. In questo modo, le specifiche informali sono state codificate formalmente attraverso test che potessero essere messi in esecuzione autonomamente tramite un'apposita piattaforma. L'integrazione di questo paradigma con tecniche e tecnologie di "continuous integration" ha permesso di venire informati in tempo reale di eventuali problemi riscontrati nell'implementazione, così come della copertura dei test sull'intera *codebase*.

La piattaforma utilizzata per tutti i test è stata "scalatest", assieme a "testcontainers-scala" per alcuni *integration test*. Il *tool* per la valutazione della *coverage* utilizzato è stato "scoverage". La *coverage* complessiva sul progetto è del 67,33% e tutti i test hanno successo, altrimenti non sarebbe stato possibile procedere con il *deployment* del sistema, a causa dei vincoli imposti sulla *CI*.

5.1 Modulo core

Il modulo "core" è composto unicamente da elementi del dominio dello spazio di tuple implementato. Questo significa che contiene unicamente elementi passivi, che possono essere verificati mediante *unit test*. Sono stati quindi testati i *trait* JsonTuple, JsonTemplate, nonché la loro corretta serializzazione e deserializzazione. Infine, è stato testato il *domain specific language* utilizzato per creare tuple e *template*, in modo da verificare che creasse gli oggetti richiesti.

Sono stati realizzati complessivamente 100 test, suddivisi su 5 *suite* e la *coverage* sul singolo modulo è del 97,23%, pressoché totale.

5.2 Modulo client

Il modulo "client" contiene l'implementazione del *client* dello spazio di tuple, quindi una componente attiva, assieme alle richieste che questo può rivolgere e alle risposte che può ricevere, che sono invece componenti passive. Poiché testare il *client* significa dover costruire richieste e risposte, non è stato ritenuto utile testare separatamente queste ultime tramite *unit test*. È stato implementato solamente un *integration test* per il *client* vero e proprio, che viene effettuato simulando la presenza dello spazio di tuple tramite un *server web* locale creato con la libreria Akka HTTP.

Sono stati realizzati complessivamente 22 test in un'unica *suite* e la *coverage* sul singolo modulo è del 63,82%. La *coverage* più bassa rispetto al modulo precedente è da ascri-

vere all'incapacità di scoverage di valutarla sulle "given instances" e su alcune "lambda expression" presenti nel codice.

5.3 Modulo server

Il modulo "server" contiene l'implementazione del *tuple space*, che è un *web service*, quindi si compone di un *controller* che accetta le richieste in entrata e inoltra le risposte in uscita, effettuando le corrette serializzazione e de-serializzazione. Oltre al *controller*, nel modulo è contenuto un agente che si preoccupa di gestire le richieste così da generare le risposte adeguate. Infine, sono presenti le implementazioni di tutte le richieste che il *web service* riceve in entrata e di tutte le risposte che invia in uscita. Come per il modulo "client", non è stato necessario l'impiego di *unit test* per testare le richieste e le risposte, in quanto già coperte dagli *integration test*. I test di integrazione si sono quindi concentrati sul testare l'agente che implementava il comportamento dello spazio di tuple, utilizzando il *test kit* fornito da Akka per verificare il comportamento di un attore. In questo modo si è riusciti a simulare la presenza del *controller* e più importantemente di un *client* dietro di esso. Analogamente, i test per verificare il comportamento del *controller* hanno simulato la presenza di un attore a cui il primo avrebbe dovuto inoltrare le richieste ricevute e da cui avrebbe dovuto ricevere le risposte, mediante l'uso del *test kit* di Akka. Inoltre, è stato simulato anche il *client* che effettua le richieste e attende le risposte, utilizzando questa volta il *test kit* di Akka HTTP per la verifica delle *route*.

Sono stati realizzati complessivamente 51 test, suddivisi su 2 *suite* e la *coverage* sul singolo modulo è del 68,73%. La *coverage* ridotta, in maniera analoga al modulo precedente, è da ascrivere all'incapacità di scoverage di valutarla sulle "given instances" e su alcune "lambda expression" presenti nel codice.

5.4 Modulo worker

Il modulo "worker" si compone di alcune entità passive che fungono da strutture dati per contenere informazioni sul dominio, come l'identificatore e il tipo di un eseguibile o l'identificatore, gli argomenti e l'*output* di un'operazione di esecuzione. Sono presenti però anche le implementazioni di due agenti, "RunnerAgent" e "WorkerAgent", che rispettivamente si occupano di mettere in esecuzione gli eseguibili e di gestire una singola macchina su cui il sistema può allocare dei *file* per una successiva esecuzione. Anche in questo caso, come in precedenza, non sono stati previsti *unit test*, ma solamente *integration test* per gli agenti, in quanto capaci di coprire anche il corretto comportamento delle entità di dominio. Entrambi i test si sono svolti con l'aiuto del *test kit* di Akka, nel caso del primo attore per poter simulare il WorkerAgent che l'ha creato e che gli inoltra le richieste di esecuzione. Nel caso del secondo attore, invece, il *test kit* è stato utilizzato solamente per poterlo mettere in esecuzione. La simulazione del *tuple space* è stata fatta avviando un *container* costruito a partire dall'immagine Docker dello spazio di tuple precedentemente rilasciata su DockerHub utilizzando il *framework* testcontainers-scala. Durante il test le richieste allo spazio di tuple vengono effettuate creando un *client* e utilizzandolo, così come fa il WorkerAgent.

Sono stati realizzati complessivamente 12 test, suddivisi su 2 *suite* e la *coverage* sul singolo modulo è del 27,89%. La *coverage* è molto bassa per questo modulo perché ampie parti del WorkerAgent risultano non coperte secondo il *tool*. Questo però non è vero, in quanto è stata verificata manualmente la copertura delle linee di codice. Il risultato è quindi segno di un errore da parte dello strumento nel rilevare le linee effettivamente messe in esecuzione.

5.5 Modulo master

Il modulo "master" si compone, in maniera del tutto simile ai precedenti tre, di una parte di componenti attivi, fatta da agenti, e di una parte di componenti passivi, che riguardano invece il dominio e le richieste e le risposte scambiate. Anche questo modulo contiene l'implementazione di un *web service*, in maniera del tutto analoga al modulo "server". Troviamo infatti al suo interno sia l'agente che implementa il comportamento della componente principale, sia il *controller* che fa da mediatore e si preoccupa di tradurre adeguatamente le richieste in entrata e le risposte in uscita. In questo caso, è presente anche un componente che fa da *storage* per il *web service*, cioè che permette l'interfacciamento con il *database* che trattiene le informazioni. Anche in questo caso, non sono stati previsti *unit test* perché gli *integration test* realizzati sono stati capaci di garantire *coverage* per le entità di dominio e per quelle che rappresentano le richieste e le annesse risposte. Il test per verificare la correttezza del *controller* si è servito del *test kit* di Akka per simulare l'attore a cui inoltrare le richieste, nonché da cui ricevere le risposte, e del *test kit* di Akka HTTP per simulare il *client* che effettua le richieste. Il test per lo *storage*, invece, ha utilizzato testcontainers-scala per avviare un *container* ottenuto dall'immagine di PostgreSQL così da avere un *database* con cui poter interagire. Da ultimo, il test per l'agente che realizza il comportamento del servizio utilizza il *test kit* di Akka solamente per poter avviare l'attore che lo implementa. Utilizza testcontainers-scala per avviare un *container* di PostgreSQL, in quanto per poter funzionare l'attore necessita di uno *storage* completo, e un *container* contenente il servizio dello spazio di tuple. Quest'ultimo serve perché per poter comunicare l'agente usa il *tuple space* e la sua mancanza impedirebbe al test di poter essere svolto correttamente.

Sono stati realizzati complessivamente 24 test, suddivisi su 3 *suite* e la *coverage* sul singolo modulo è del 51,14%. Anche in questo caso, la *coverage* particolarmente bassa è principalmente dovuta allo strumento che non è capace di registrare correttamente le linee di codice attraverso il quale il test è passato.

5.6 Modulo ui

Questo modulo contiene solamente l'implementazione della *UI*, cioè la *web app* che si interfaccia con l'utente. Per questo motivo, benché esistano strumenti per testare il corretto funzionamento delle interfacce utente, si è preferito l'approccio più semplice del *testing* manuale. Si è perciò verificato senza l'ausilio di ulteriori *tool* che l'interfaccia offrisse tutte le funzionalità richieste e che queste fossero correttamente visualizzate.

6 Istruzioni di deployment

Per provare il sistema sul proprio computer, sarà necessario avere installato localmente una versione del *software* Docker sufficientemente avanzata da essere compatibile con la versione "3.9" del formato di *file* di docker compose. Questo è l'unico requisito necessario, nessun altro software ha bisogno di essere presente per poter avviare la demo.

Una volta installato Docker, basterà lanciare il comando `docker compose up` nella cartella radice che contiene il progetto. A questo punto, il sistema avrà bisogno di avviarsi e l'operazione sarà completata quando i tre messaggi illustrati nel *listing* 1 saranno comparsi a video. Questo non vuol dire che saranno gli unici messaggi a comparire, anzi, ma quando questi saranno comparsi il sistema sarà pronto per essere utilizzato. Per potervi accedere, basterà navigare grazie al proprio *browser* verso l'*URL* `http://localhost:8080`. Il sistema partirà completamente vuoto e senza nessun dato pregresso, così da simulare un vero primo avvio dello stesso.

Per terminare la demo, basterà inviare la combinazione di tasti `CTRL+C` nella *console* in cui il sistema è stato avviato e sta mostrando il suo *log*. Un'alternativa è lanciare il comando `docker compose down` nella stessa cartella in cui prima si è lanciato il comando di avvio della demo.

Listing 1: Listato che mostra i tre messaggi che devono comparire a video perché il sistema possa considerarsi avviato

```
lambdas-as-a-service-first_worker-1 | Worker bdd00fa6-db48-4b08-8b3c-488
    ↪ a1795f1c5 up
lambdas-as-a-service-second_worker-1 | Worker 17439dad-d9ed-4ee6-af05-7
    ↪ eca427ae025 up
lambdas-as-a-service-master-1 | Master up
```

7 Esempio d'uso

Una volta connessi all'indirizzo a cui si trova la *web app* sul proprio *browser*, ci si troverà davanti alla schermata che si può vedere in figura 13. Questa schermata dà la possibilità di registrarsi se non lo si è già fatto tramite la maschera di sinistra. Se si usa la maschera di destra, invece, è possibile effettuare il *login* con credenziali già registrate in precedenza. Le credenziali da utilizzare in entrambi i casi sono le stesse, ovvero *username* e *password*.

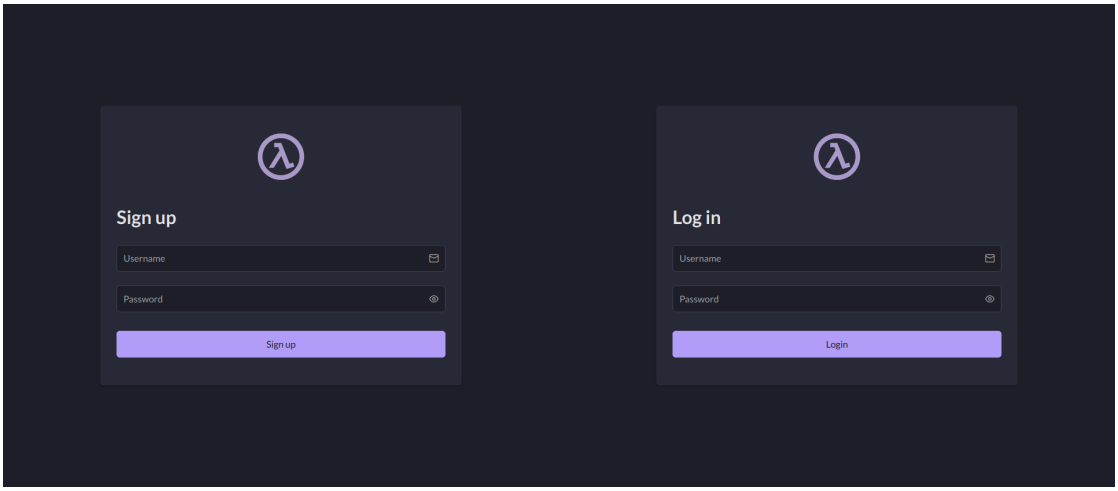

Figura 13: Schermata iniziale dell'applicazione

Se è la prima volta che si accede all'applicazione, ci si troverà davanti ad una schermata come quella in figura 14. Questa schermata permette, grazie ai pulsanti in alto a destra, di effettuare il *deployment* di un nuovo file o di effettuare il *logout* dall'applicazione. Se si decide di effettuare il *logout*, si verrà riportati alla schermata iniziale.

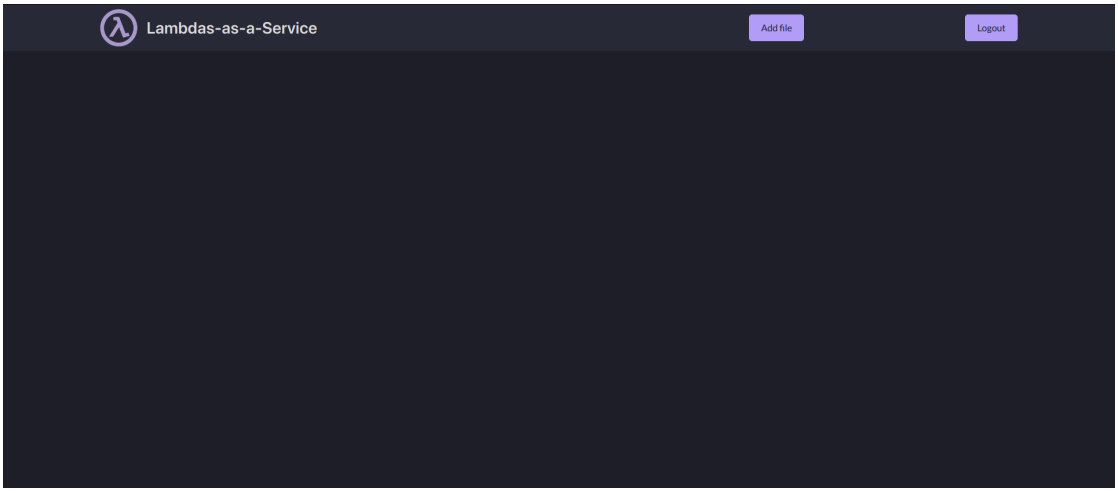

Figura 14: Schermata della *home* dell'utente visibile dopo l'accesso, inizialmente vuota

Se si decide invece di voler effettuare l'allocazione di un nuovo eseguibile, sarà mostrata una maschera come quella in figura 15. La maschera offre la possibilità di scegliere il *file* da caricare dal proprio computer, di assegnargli un nome e, una volta soddisfatti della scelta fatta, avviare l'allocazione.

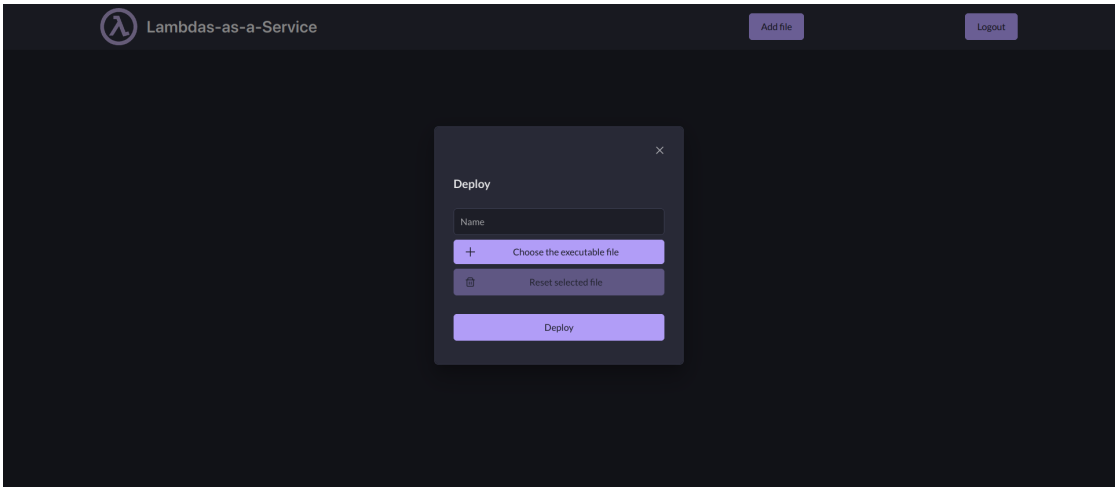

Figura 15: Schermata che mostra la maschera per effettuare il *deployment* di un nuovo eseguibile

Una volta scelti l'eseguibile e il nome, la maschera apparirà come in figura 16. Il pulsante per rimuovere il *file* caricato sarà attivo, il che significa che sarà possibile premerlo per poterne sceglierne uno diverso, qualora si abbia cambiato idea.

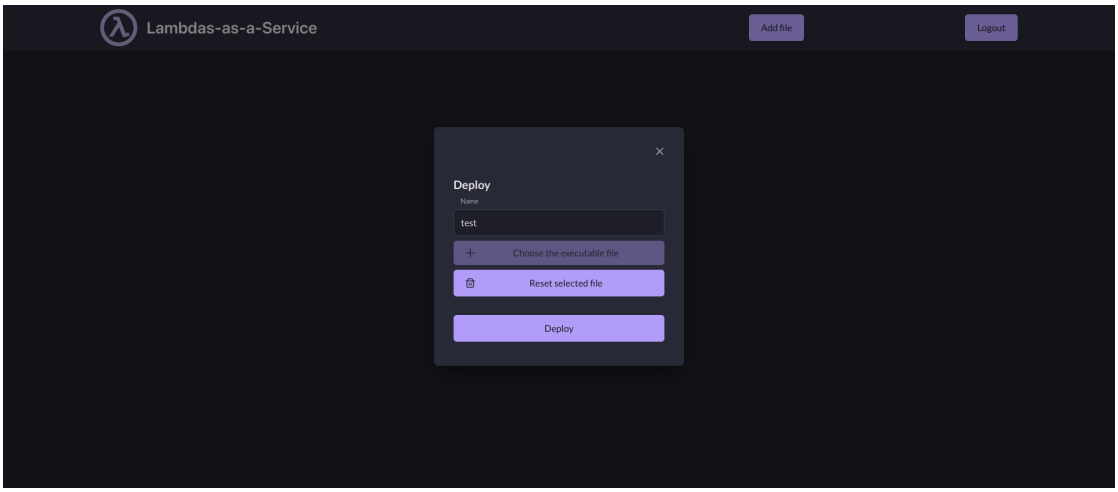

Figura 16: Schermata che mostra la maschera per effettuare il *deployment* con nome e *file* inseriti

Una volta premuto il pulsante per avviare il *deployment*, la maschera apparirà come in figura 17. Bisognerà attendere finché la schermata non tornerà in uno stato "attivo" prima di poter effettuare una nuova allocazione.

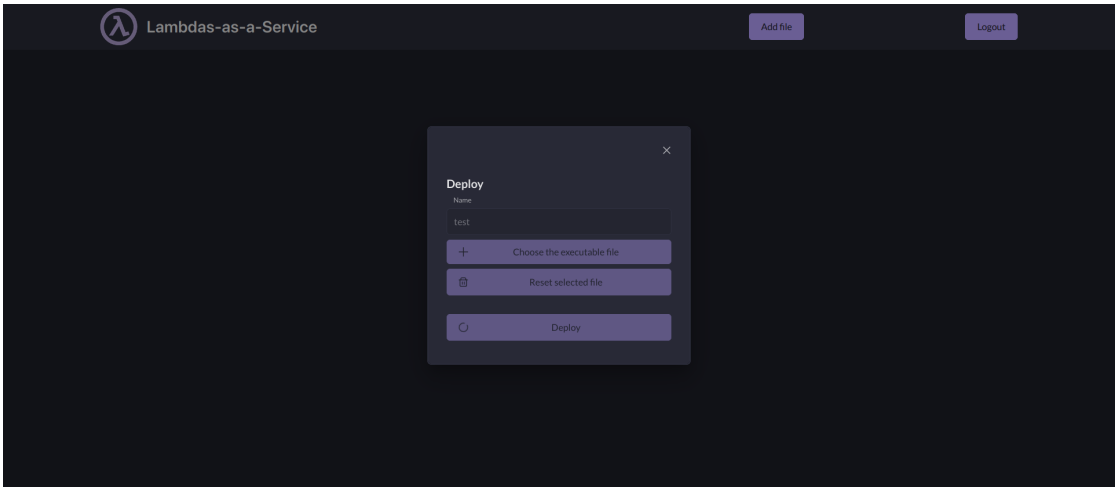

Figura 17: Schermata che mostra la maschera per effettuare il *deployment* mentre questo è in corso

La terminazione dell'allocazione sarà segnalata, oltre al cambiamento di stato della maschera apposita, dalla comparsa di un nuovo elemento nella *home* dell'utente. Questo rappresenta l'eseguibile appena allocato e infatti ne possiede il nome. Subito sotto, è presente un pulsante che permette di aprire la maschera per metterlo in esecuzione, come si vede in figura 18.

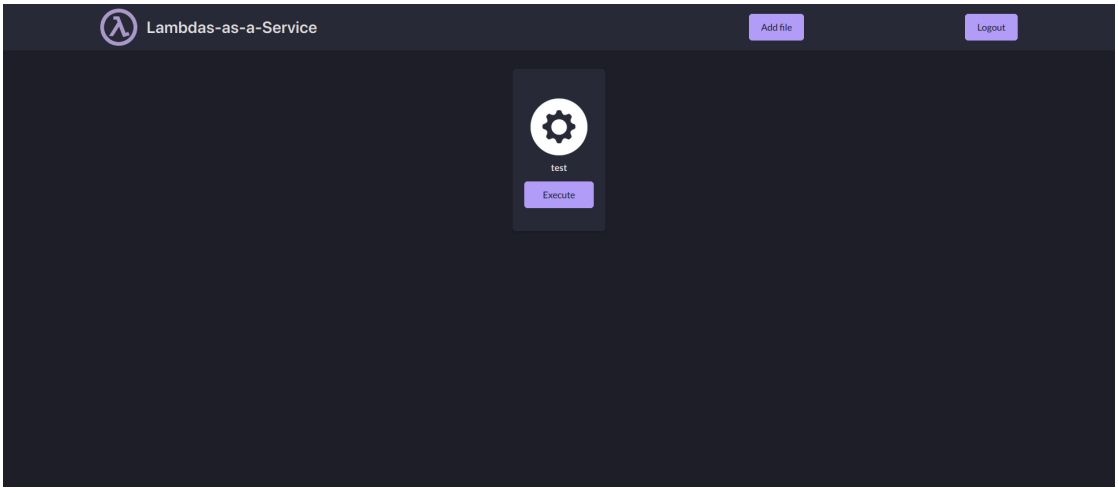

Figura 18: Schermata della *home* dell'utente visibile dopo avere effettuato il *deployment* di un eseguibile

Una volta premuto il pulsante, la maschera che appare è quella in figura 19. Questa presenta un campo di testo in cui è possibile inserire, uno per riga, gli argomenti da

passare al processo che metterà in esecuzione il *file* eseguibile. Il pulsante sottostante permette di lanciare l'esecuzione.

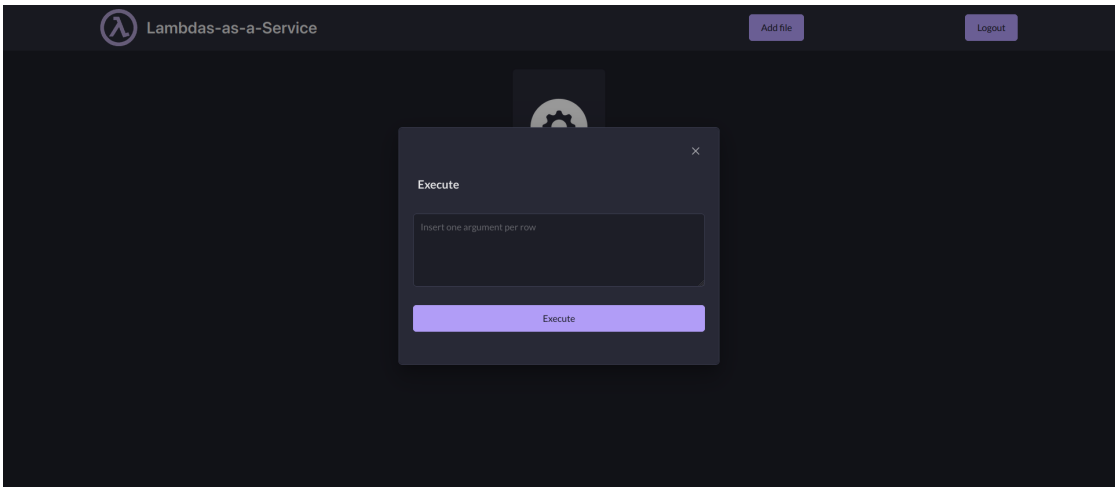

Figura 19: Schermata che mostra la maschera per la messa in esecuzione del *file* eseguibile selezionato

Una volta terminata l'esecuzione, apparirà invece la maschera visibile in figura 20. Questa presenterà tre campi di testo, che non possono essere modificati, contenenti rispettivamente l'*exit code* del processo che ha messo in esecuzione l'eseguibile, il suo *standard output* e il suo *standard error*.

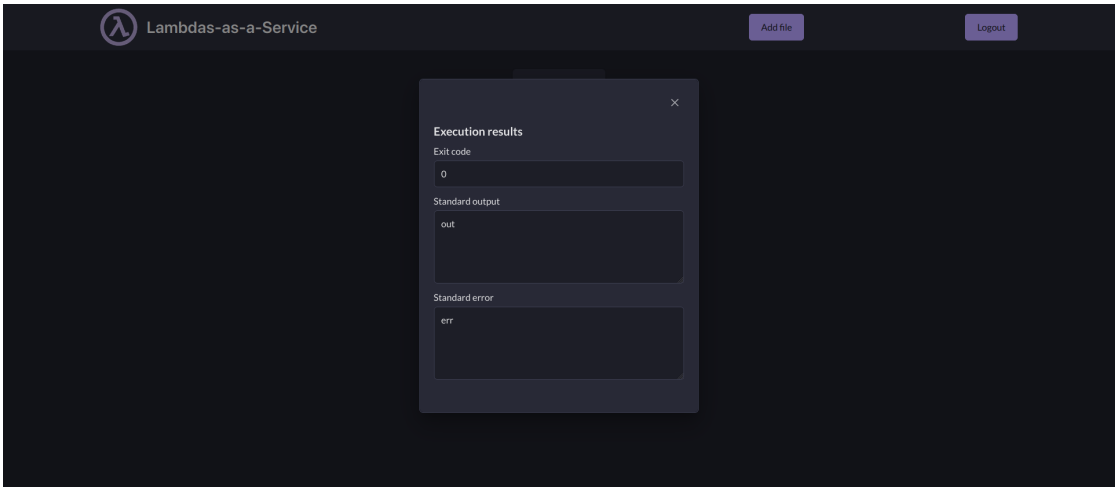

Figura 20: Schermata che mostra la maschera contenente i risultati dell'esecuzione lanciata poco prima

8 Conclusioni

In questo progetto, si è realizzato il prototipo di un *web service* per la computazione distribuita, capace di allocare e mettere in esecuzione *file* eseguibili da remoto. Questo significa che il sistema dispone di una *UI* accessibile via *web browser* per accedere alle sue funzionalità. Queste sono esclusivo appannaggio di un utente registrato al sistema, per questo motivo esso offre anche un meccanismo di base per il controllo d'accesso degli utenti.

L'architettura del sistema è costituita da due tipi di componenti: uno principale e uno secondario. Quello principale funge da mediatore tra utente e sistema nel suo complesso, orchestrando tutte le operazioni necessarie, e ne è presente una singola istanza. Quello secondario può essere invece presente in più istanze, una per macchina utilizzata, rendendo il sistema effettivamente distribuito, ma soprattutto scalabile, flessibile e *fault tolerant*. Per facilitare lo sviluppo di un sistema con tali caratteristiche, si è deciso di impiegare il paradigma ad agenti nella sua progettazione.

Per comunicare, tutte le componenti si servono di un *tuple space* condiviso con cui interagiscono sfruttando le primitive definite dal linguaggio di coordinazione "Linda". Questo permette una comunicazione tra componenti che sono tra loro disaccoppiati e che perciò non sanno nulla l'uno dell'altro ed eseguono senza simultaneità spaziale e temporale.

8.1 Sviluppi futuri

Il sistema attualmente supporta solamente la messa in esecuzione di *file* JAR, che possono essere stati implementati in qualsiasi linguaggio, purché capace di essere messo in esecuzione su di una "Java Virtual Machine" alla versione "19.0.2". Questo lascia aperte le porte ad un miglioramento del sistema volto al supporto di altre versioni della JVM e di altri linguaggi "*portable*", quindi linguaggi interpretati o compilati per una *virtual machine* che ha un supporto multi-piattaforma. È già stato fatto uno sforzo per supportare il linguaggio "Python" a livello di codice, come dimostrazione dell'estensibilità del sistema, ma non a livello di infrastruttura. Questo perché i processi di esecuzione degli eseguibili vengono lanciati come processi della macchina in cui il componente secondario è correntemente in esecuzione. Il supporto ai linguaggi compatibili con la JVM è garantito dal fatto che, se non ci fosse supporto, nemmeno il componente secondario potrebbe essere in esecuzione, in quanto implementato nel linguaggio "scala".

Un ulteriore miglioramento, quindi, potrebbe essere quello per cui i processi di esecuzione vengono lanciati come, o dentro a, *container* Docker. In questo modo si avrebbe un miglioramento nel supporto ai linguaggi che potrebbe essere virtualmente infinito, ma si avrebbe anche un miglioramento in termini di sicurezza e di gestione delle risorse. Lanciare l'esecuzione all'interno di un *container*, infatti, permetterebbe di isolare il processo dal resto della macchina ed evitare quindi che codice malevolo vada a manipolarla per i propri scopi. Inoltre, Docker offre molte possibilità per quanto riguarda la limitazione delle risorse che sono offerte ai suoi *container*. È possibile specificare il numero di CPU assegnate, il tempo per cui possono esserlo, la quantità di RAM, di memoria

secondaria, eccetera. Tutti questi elementi potrebbero essere poi presi in considerazione nella valutazione delle risorse rimanenti per una data macchina.

Un terzo elemento che può essere migliorato è infatti l'algoritmo di scelta della migliore componente secondaria su cui effettuare il *deployment*. L'uso degli *slot* è un'approssimazione a grana molto grossa dei fattori che vengono realmente presi in considerazione, alcuni dei quali sono stati citati poc'anzi, dai sistemi che devono effettuare allocazione di risorse.

Un ulteriore miglioramento nella direzione della sicurezza riguarda il sistema di accesso dell'applicazione. Si potrebbe pensare di passare ad un modello "*role based*", o più semplicemente offrire più controllo sui permessi che riguardano i singoli eseguibili caricati nel sistema.

Un miglioramento si può avere nel campo delle funzionalità: attualmente un utente non può de-registrarsi dal servizio, come non può de-allocare un eseguibile allocato in precedenza, e si potrebbe volergli offrire queste possibilità.

Sempre parlando di funzionalità, si può migliorare la cattura dell'*output* per gli eseguibili *long-running*. Attualmente, tutto l'*output* viene raccolto alla terminazione dell'esecuzione e restituito all'utente che l'ha richiesta. Se un *file* impiega molto tempo ad eseguire, l'operazione di raccolta a sua volta impiega molto tempo. Il miglioramento necessario al sistema è minimale, poiché coinvolge semplicemente l'invio al *tuplespace* di una tupla per ogni riga stampata su *standard output* e su *standard error*, mano a mano che queste vengono prodotte. Una volta terminata l'esecuzione, il RunnerAgent invierà una tupla speciale contenente l'*exit code* segnalando questo evento. La ServiceApi dovrà perciò intercettare tutte queste tuple e inviare il loro contenuto sulla *websocket* corrispondente all'utente corretto, fino a che non inoltrerà il messaggio di terminazione dell'esecuzione, come già viene fatto.

D'altro canto, il sistema è già capace di supportare eseguibili che invocano il sistema stesso per mettere in esecuzione altri *file*. L'unico vincolo a questo meccanismo è che l'eseguibile, una volta caricato sulla macchina che ospita il RunnerAgent, deve avere accesso all'eseguibile che vuole lanciare, ad esempio copiandolo attraverso la rete. Dopodiché, basterà aprire una *websocket* verso la ServiceApi bypassando il *web client*, ovvero sostituendosi ad esso, e inviando la corretta sequenza di messaggi per richiedere l'esecuzione di questo secondo eseguibile. Questo implica una vulnerabilità di sicurezza non indifferente: un *file* può sempre mettere in esecuzione se stesso ricorsivamente sul servizio, andando a compiere un attacco mirato ad esaurire le risorse disponibili.

8.2 Cosa abbiamo imparato

In questo progetto è stato possibile approfondire ulteriormente i concetti propri del linguaggio Linda, come le primitive che supporta, e la nozione di "spazio di tuple". È stato inoltre possibile studiare in maniera pratica come sviluppare *web service* nel linguaggio scala, adottando le giuste tecniche di programmazione ed approfondendo le tecnologie che questo linguaggio offre come "Akka HTTP". Un'altra tecnologia che è stata studiata nel corso di questo progetto è stata il protocollo "Websocket", utilizzato per tutte le comunicazioni tra le diverse parti. Seppur in maniera minore, è stata approfondita

anche l'architettura "ReST", per valutare se il progetto avrebbe potuto consistere nello sviluppo di un "*ReSTful web service*", così come per lo stesso motivo è stato approfondito il generatore di specifiche "SwaggerHub". Inoltre, è stato possibile mettere in pratica l'uso del paradigma ad agenti nell'ambito dell'implementazione di un sistema distribuito complesso, implementazione che è quindi modellata come un "*Multi-Agent System*". Sono stati approfonditi gli aspetti del corso inerenti alla *build automation* e in generale all'automazione dei compiti inerenti ad un progetto *software* o ad una *repository* grazie allo strumento "SBT". Da ultimo, ma non per questo meno importante, è stata approfondita la tecnologia "Docker" per la containerizzazione di applicazioni, senza la quale non sarebbe stato possibile testare e mettere in funzione l'intero sistema.

Riferimenti bibliografici

[1] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. Ultimo accesso: 2023-10-15.

[2] Foundation for Intelligent Physical Agents. FIPA Contract Net Interaction Protocol Specification. `http://www.fipa.org/specs/fipa00029/SC00029H.html`. Ultimo accesso: 2023-10-15.

[3] Foundation for Intelligent Physical Agents. FIPA Request Interaction Protocol Specification. `http://www.fipa.org/specs/fipa00026/SC00026H.html`. Ultimo accesso: 2023-10-15.

[4] David Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, gen 1985.

[5] The Apache Software Foundation. Apache Hadoop YARN. `https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html`. Ultimo accesso: 2023-10-15.

[6] The Apache Software Foundation. Cluster Mode Overview. `https://spark.apache.org/docs/latest/cluster-overview.html`. Ultimo accesso: 2023-10-15.

[7] The Apache Software Foundation. HDFS Architecture. `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`. Ultimo accesso: 2023-10-15.

[8] A. Wright, H. Andrews, B. Hutton, and G. Dennis. JSON Schema: A Media Type for Describing JSON Documents. `https://json-schema.org/draft/2020-12/draft-bhutton-json-schema-01`. Ultimo accesso: 2023-10-15.