Alma Mater Studiorum
Università di Bologna

Campus di Cesena
Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Integrazione sistemi a eventi e multi-agente: Kafka, TuCSoN e JADE

Tesi in
Sistemi Distribuiti

Candidato
Marco Zaccheroni

Relatore
Prof. Andrea Omicini
Correlatore
Stefano Mariani

Anno Accademico 2015/2016 - Sessione I

A mio Nonno.

Indice

Introduzione

Quando si tratta di modellare e progettare i Sistemi Distribuiti bisogna fare i conti con un elevato livello di complessità che nasce dall'interazione distribuita e dalla concorrenza su larga scala dei vari componenti. È di fondamentale importanza quindi che questi sistemi vengano realizzati tramite architetture in grado di fornire astrazioni adeguate per gestire tali problematiche.

Negli ultimi anni, lo stile architetturale più usato per la realizzazione di applicazioni distribuite su larga scala è quello dei *Sistemi ad Eventi* (chiamati brevemente EBS, dal nome inglese Event-Based Systems). Questi sistemi si basano sul concetto di evento, ovvero l'occorrenza di un avvenimento rilevante per il sistema, come costrutto fondamentale nella comunicazione e coordinazione tra i vari componenti del sistema.
Esiste però un altro stile architetturale che apparentemente offre astrazioni più vantaggiose e metodi di progettazione più coerenti per affrontare la complessità dei sistemi distribuiti, quello dei *Sistemi Multi-Agente* (MAS, dal nome inglese Multi-Agent Systems). I MAS forniscono caratteristiche avanzate come mobilità, autonomia, ragionamento simbolico, gestione della conoscenza e riconoscimento delle situazioni, che spesso fanno parte dei requisiti di un sistema distribuito.

L'obiettivo di questa tesi è dimostrare come questi due stili architetturali possano essere integrati per la creazione di una unica architettura capace di sfruttare al meglio le caratteristiche di entrambe le tipologie di sistemi.

Nell'intento di chiarire e approfondire adeguatamente l'argomento, la tesi si articola come segue:

- nel capitolo 1 vengono fornite descrizioni dettagliate sui sistemi finora solo accennati, presentando anche alcuni framework disponibili per la loro realizzazione: JADE e TuCSoN per i Sistemi Multi-Agente, Kafka e RabbitMQ per i Sistemi ad Eventi. Viene inoltre presentato il modello ELDA come esempio già esistente di MAS basato su eventi;
- nel capitolo 2 viene realizzata un'analisi approfondita dei framework presentati e viene descritta l'integrazione dal punto di vista architetturale di JADE e Kafka;
- nel capitolo 3 viene proposto un caso di studio che permette di analizzare le potenzialità dell'integrazione tra JADE, Kafka e TuCSoN.

Capitolo 1

Background

In questo capitolo vengono introdotti i principali argomenti su cui si concentra questa tesi. Si descrivono dapprima i sistemi multi-agente ed i sistemi basati sugli eventi, evidenziandone le caratteristiche peculiari. Per ogni paradigma si introducono inoltre i principali framework di sviluppo, sottolineando i modi in cui essi permettono di gestire la coordinazione all'interno dei sistemi distribuiti. Infine viene presentato anche il modello ELDA come primo riferimento di integrazione tra sistemi a eventi e multi-agente.

1 Sistemi multi-agente

Un sistema multi-agente (MAS) è una collezione di entità computazionali autonome, dette *agenti*, che interagiscono tra di loro e con l'*ambiente* in cui sono situati e svolgono attività in maniera proattiva, compiendo azioni al fine di raggiungere uno o più obiettivi.
Ogni agente ha una propria sfera di influenza, ovvero una porzione di ambiente che è in grado di controllare parzialmente. Nel momento in cui le aree di influenza di più agenti si sovrappongono, essi sono in grado di instaurare rapporti organizzativi per collaborare e raggiungere obiettivi di più alto livello. Gli insiemi organizzati di agenti che collaborano e si coordinano per regolare le rispettive dipendenze, in modo da evitare di interferire tra di loro nel tentativo di raggiungere un obiettivo di sistema, vengono definite *società di agenti*. Si noti che l'ambiente di un MAS ha il compito di catturare l'imprevedibilità del contesto in cui esso opera, modellando le caratteristiche e le

risorse esterne al sistema, insieme ai loro cambiamenti nel tempo [1]. Questo permette agli agenti di avere comportamenti di *situatedness*, ovvero di accorgersi dei cambiamenti tramite la stretta interazione con l'ambiente e di modificare il proprio comportamento di conseguenza.
All'interno di un MAS possiamo anche trovare dei componenti reattivi, denominati *artefatti*, il cui obiettivo è quello di essere usati dagli agenti per aumentare le proprie capacità e svolgere azioni particolari. Tramite artefatti è possibile, ad esempio, modellare e implementare qualsiasi tipo di risorsa ambientale che può essere gestita o controllata dagli agenti, realizzando così gli elementi base per la costruzione di ambienti di lavoro complessi [2].

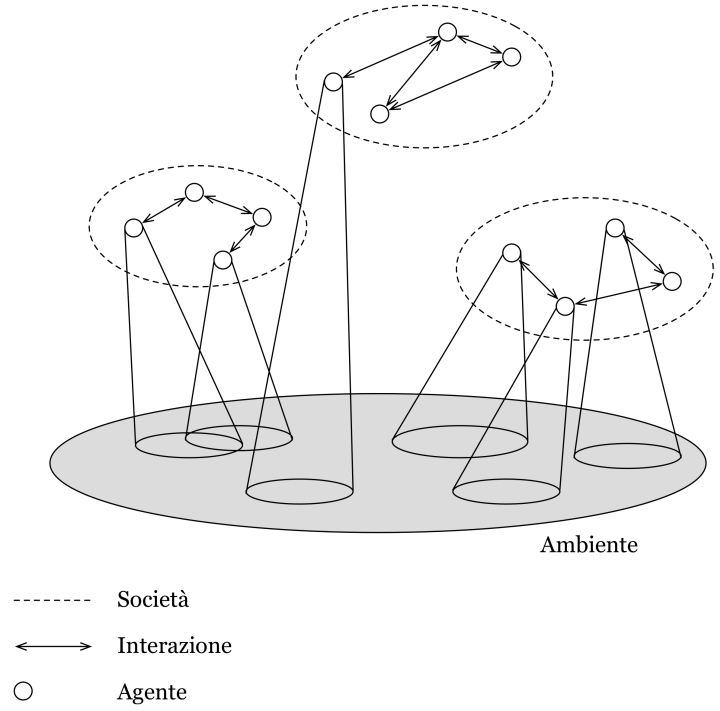

Figura 1.1: Sistema Multi-Agente

1.1 JADE

JADE (Java Agent DEvelopment Framework) [3] è un framework sviluppato in Java che supporta lo sviluppo di applicazioni ad agenti in accordo con gli standard FIPA [1] per la realizzazione di MAS intelligenti e interoperabili. In

[1] sito: http://www.fipa.org

JADE, l'autonomia degli agenti è supportata dall'astrazione dei *behaviour*, mentre la coordinazione tra gli agenti è realizzata tramite un livello di comunicazione a *message-passing asincrono*, chiamato *Agent Communication Channel* (ACC), sopra al quale sono implementati i protocolli FIPA di interazione che vengono forniti agli sviluppatori sottoforma di *framework a callback*.

Un behaviour può essere visto come *un'attività da svolgere con l'obiettivo di completare un task*; esso può essere sia un'attività *proattiva*, svolta dall'agente di propria iniziativa, che un'attività *reattiva*, in risposta all'occorrenza di un particolare evento. Tipicamente, un agente deve compiere attività complesse, composte da numerosi task, che vengono incapsulate in diversi behaviour eseguiti simultaneamente dall'agente. JADE implementa i behaviour come oggetti Java che vengono eseguiti in maniera pseudo-concorrente da uno scheduler round-robin non-preemptive. Ogni behavior è caratterizzato dai metodi `action()` e `done()`: il primo definisce l'insieme di azioni che costituiscono l'attività da svolgere, mentre nel secondo è definita la condizione di completamento del task. Quando un behaviour viene aggiunto all'agente, tipicamente durante la fase di inizializzazione, questo viene inserito nella *ready queue* dello scheduler pronto per essere schedulato. Si noti che quando viene eseguito un behaviour tramite il metodo `action()` *nessun'altro behaviour può essere eseguito* (scheduler non-preemptive); in seguito, quando l'esecuzione di `action()` è terminata, viene eseguito il metodo `done()`; se esso ritorna `true` il behaviour viene rimosso dalla *ready queue*, altrimenti viene reinserito alla fine della coda (scheduler round-robin).

Il servizio middleware ACC si occupa di gestire il message passing asincrono tra gli agenti. Ogni agente è provvisto di una coda di messaggi, una sorta di mailbox, in cui vengono inseriti silenziosamente i messaggi dall'ACC, in attesa che vengano considerati dall'agente in maniera proattiva. L'agente è quindi completamente libero di decidere *come* e *quando* processare i messaggi ricevuti, tramite le primitive `receive()` e `blockingReceive()` che rispettivamente rappresentano la ricezione dei messaggi in maniera asincrona (se non ci sono messaggi l'agente può continuare l'esecuzione) e sincrona (l'agente rimane in attesa fino a che non arriva un messaggio). Si noti che la

comunicazione in JADE è *distribution-transparent* per agente e programmatore, è il middleware che si occupa di scegliere l'indirizzo e il meccanismo di trasporto adeguato in base alle posizioni del mittente e del destinatario.

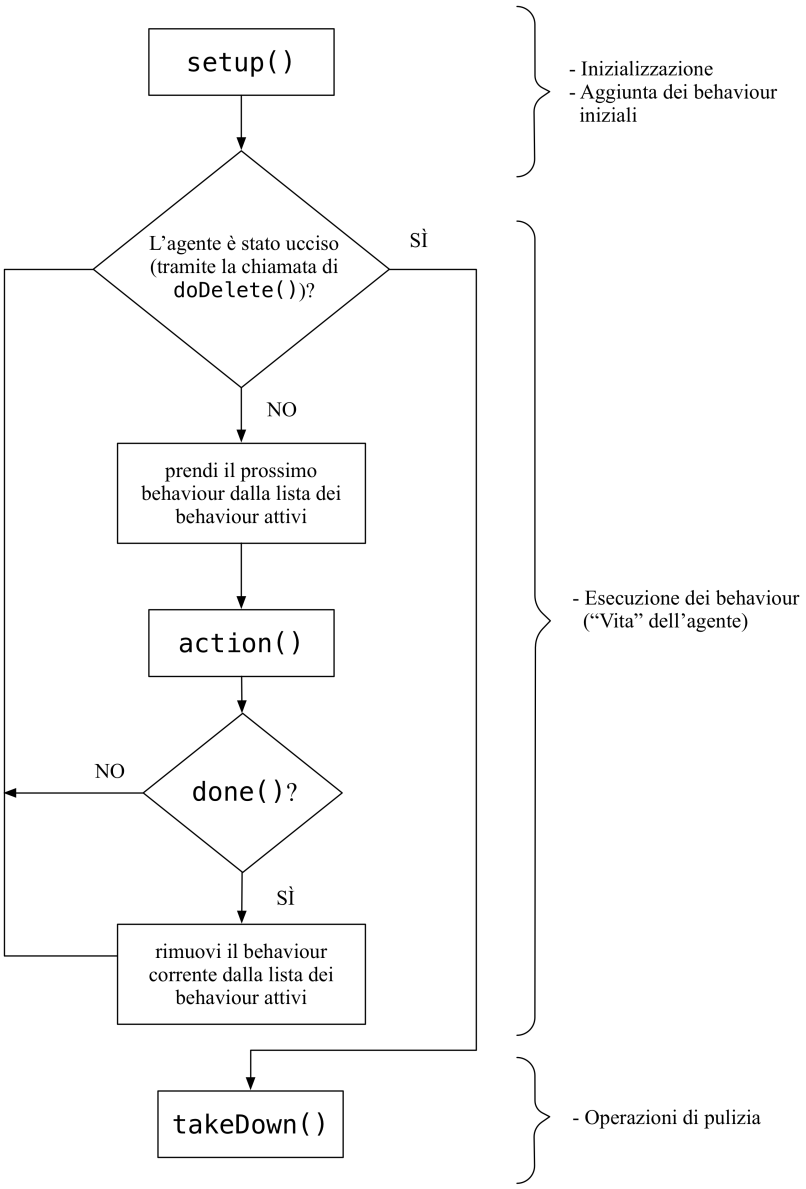

Figura 1.2: Policy non-preemptive di scheduling dei behaviour in JADE

1.2 TuCSoN

TuCSoN (Tuple Centres Spread over the Network)[2] è un modello di coordinazione e una infrastruttura per MAS aperti e distribuiti [4].
In TuCSoN la coordinazione viene realizzata tramite *centri di tuple ReSpecT* [5] distribuiti su una rete di nodi TuCSoN interconnessi.
I centri di tuple sono spazi di tuple logiche potenziati con comportamenti programmabili tramite il linguaggio logico ReSpecT [6] e ricoprono il ruolo di *coordination media* del sistema, infatti l'interazione tra gli agenti e tra gli agenti e le risorse ambientali, avviene tramite lo scambio di tuple attraverso i centri di tuple mediante l'utilizzo delle primitive di coordinazione rese disponibili dal linguaggio di coordinazione TuCSoN.

Ogni volta che un agente accede ad un MAS coordinato da TuCSoN gli viene assegnato un *Agent Coordination Context* (ACC) che ricopre il ruolo di *mediatore* tra l'agente e i centri di tuple. Gli ACC sono artefatti che hanno il compito di mappare le operazioni di coordinazione richieste dagli agenti in eventi ReSpecT, inviare questi ultimi al coordination medium, attendere la risposta ed infine restituirla all'agente. Questo permette di preservare l'autonomia e l'indipendenza dell'agente (sia dal punto di vista esecutivo che di progettazione) tramite il disaccoppiamento del sincronismo dell'invocazione di una operazione di coordinazione dalla semantica bloccante della primitiva di coordinazione usata, tramite l'esecuzione a "due step" di qualsiasi operazione TuCSoN:

1. *Invocation*: la *richiesta* di svolgere un'operazione di coordinazione viene effettuata dall'agente attraverso il suo ACC, il quale poi invia il corrispondente evento dell'operazione al centro di tuple destinatario;
2. *Completion*: la *risposta* all'operazione di coordinazione invocata viene inviata all'agente che ha effettuato la richiesta attraverso il suo ACC, non appena la risposta è pronta.

Questo significa che sostanzialmente è l'ACC dell'agente ad effettuare le operazioni di coordinazione, lasciando decidere l'agente in maniera del tutto autonoma se aspettare che sia pronta la risposta o svolgere altre operazioni nel

[2]sito: http://tucson.apice.unibo.it

frattempo e controllare se questa è pronta in un secondo momento in maniera proattiva.

Le risorse ambientali in TuCSoN vengono chiamate *probes*. Tramite queste astrazioni è possibile modellare qualsiasi tipo di risorsa ambientale, sia quelle che generano informazioni (e.g. sensori) che quelle capaci di svolgere azioni (e.g. attuatori). Analogamente agli agenti, i probes non interagiscono direttamente con il MAS, ma si avvalgono di intermediari chiamati *transducers*. I trasducer sono anch'essi artefatti e svolgono un compito similare a quello degli ACC: traducono i cambiamenti delle proprietà delle risorse in eventi e permettono di svolgere *situated operations* sui Probe. Essi quindi disaccoppiano i probes dai centri di tuple in termini di controllo, riferimento, spazio e tempo.

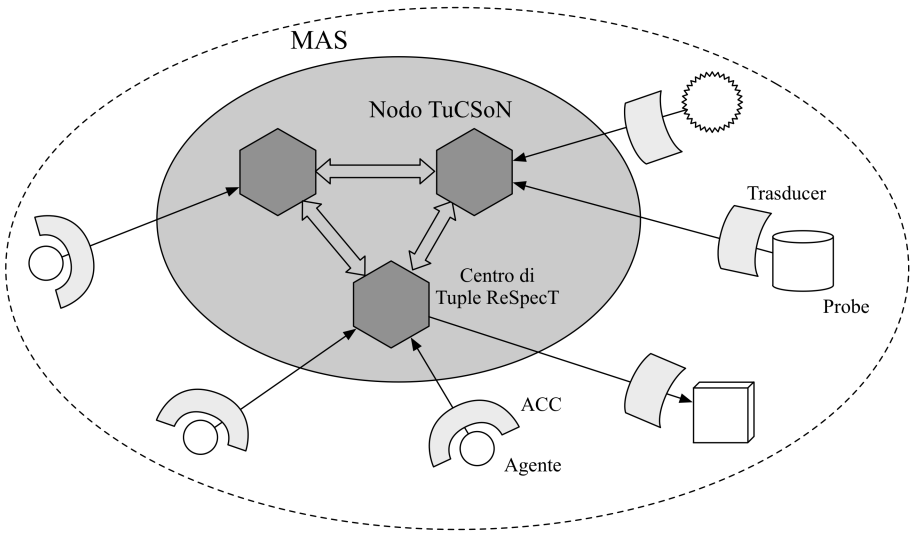

Figura 1.3: Architettura di un nodo TuCSoN

2 Sistemi ad eventi

Un Sistema ad Eventi (EBS) è "un sistema in cui i componenti integrati comunicano generando e ricevendo *notifiche di eventi*" [7], dove un *evento* è l'occorrenza di un avvenimento rilevante per il sistema, e.g. il cambiamento di stato di un qualche componente, e una *notifica* è la reificazione di un

evento all'interno del sistema. I componenti di un EBS si comportano sia come *produttori* che come *consumatori* di notifiche: i produttori *pubblicano* notifiche e forniscono una *interfaccia di output* per la sottoscrizione, mentre i consumatori si *sottoscrivono* alle notifiche come specificato dai produttori (modello Publish/Subscribe). Sostanzialmente i produttori e i consumatori non interagiscono mai direttamente tra di loro: la loro interazione viene mediata dal *bus degli eventi* che permette di astrarre la complessità del *servizio di notifica degli eventi*.

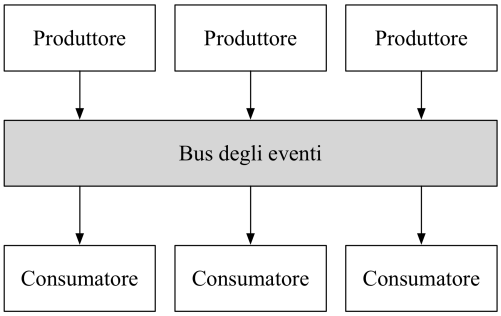

Figura 1.4: Sistema a Eventi

2.1 Kafka

Kafka[3] è un sistema distribuito open-source di scambio di messaggi, realizzato in Scala e Java.
Kafka viene eseguito in un cluster, in cui ogni nodo viene chiamato *broker*. I broker svolgono il ruolo di intermediari tra produttori e consumatori: essi memorizzano in maniera persistente i messaggi che vengono pubblicati dai produttori, in modo che i consumatori possano recuperarli in base alla loro velocità di elaborazione. Si noti che a differenza di numerosi altri sistemi simili, dove i messaggi vengono consegnati ai consumatori non appena sono disponibili (e.g. tramite callback), in Kafka sono i consumatori a richiedere i messaggi al broker quando sono pronti per elaborarli tramite il metodo `poll(`*`timeout`*`)`, in cui il parametro *`timeout`* permette di definire l'intervallo di tempo per il quale il consumatore è disposto ad aspettare nel caso non

[3]sito: http://kafka.apache.org

siano presenti nuovi messaggi.
Kafka utilizza Zookeeper[4] per coordinare tutti i membri del cluster.

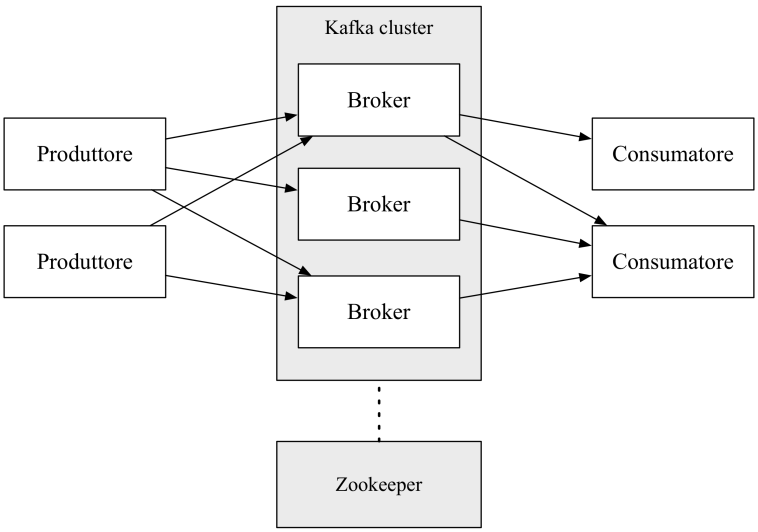

Figura 1.5: Architettura di Kafka

In Kafka tutti i messaggi sono organizzati in *topic*: ogni topic può essere suddiviso in *partizioni*, sequenze ordinate ed immutabili di messaggi, in cui i messaggi vengono accodati esattamente nell'ordine in cui sono ricevuti. I messaggi all'interno di ogni partizione vengono identificati univocamente da un numero sequenziale, chiamato *offset*. L'offset viene anche usato come parametro per indicare in che posizione è arrivato un consumatore a leggere i messaggi all'interno di una particolare partizione, questo parametro è completamente sotto il controllo del consumatore: tipicamente viene incrementato in maniera lineare man mano che legge nuovi messaggi, ma di fatto permette al consumatore di leggere i messaggi nell'ordine che preferisce.
Le partizioni di un topic vengono distribuite tra i vari broker del cluster Kafka e possono essere anche replicate attraverso un numero configurabile di altri broker per garantire fault-tolerance: un broker agisce come *leader* della partizione, gestendo le richieste di lettura e scrittura dei messagi, e zero o più altri broker agiscono come *followers* replicando passivamente il leader. Se il leader cade, un follower automaticamente prende il suo posto. Ogni broker agisce come leader di alcune partizioni e come follower delle altre, in modo

[4]sito: http://zookeeper.apache.org

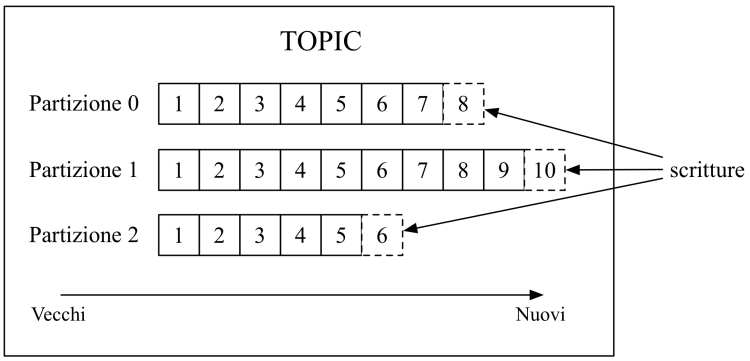

Figura 1.6: Struttura di un topic in Kafka

che il carico sia bilanciato all'interno del cluster.
I produttori pubblicano i messaggi in maniera asincrona indicando per ognuno il topic a cui vogliono indirizzarlo e opzionalmente anche a quale partizione assegnarlo, in mancanza dell'indicazione della partizione questa viene scelta tramite round-robin per bilanciare il traffico oppure tramite una funzione semantica della partizione (e.g. basata su un *chiave* del messaggio).

I sistemi di messaggistica offrono solitamente due tipologie di consegna dei messaggi, tramite *code di messaggi* o *Publish/Subscribe*. In una coda, se più consumatori leggono dallo stesso server, ognuno di loro riceve un messaggio diverso, mentre nel Publish/Subscribe tutti i consumatori sottoscritti ad un topic ricevono tutti i messaggi del topic. Kafka offre una singola astrazione per i consumatori che permette di sfruttare entrambi questi modelli: i *gruppi di consumatori*. Ogni consumatore infatti viene etichettato con il nome del proprio gruppo di consumatori. Se tutti i consumatori condividono lo stesso gruppo, il sistema lavora secondo il modello delle code di messaggi, in cui il carico viene bilanciato tra i consumatori. Se invece tutti i consumatori appartengono a gruppi diversi, il sistema lavora secondo il modello Publish/-Subscribe, in cui tutti ricevono gli stessi messaggi.
Si noti che l'ordine in cui i messaggi vengono ricevuti dai consumatori è garantito solo per i messaggi all'interno di una stessa partizione, mentre non è assicurato all'interno del topic in generale.

2.2 RabbitMQ

RabbitMQ[5] è un middleware a messaggi open-source, scritto in Erlang, che implementa il protocollo *AMQP*[6] (Advanced Message Queuing Protocol). RabbitMQ può essere eseguito in uno scenario distribuito sotto forma di *cluster*, raggruppando diversi server fisici in un unico broker logico, oppure tramite *federation*, realizzando una rete di più broker logici. È possibile anche combinare i due approcci realizzando una federation di cluster. All'interno di un cluster RabbitMQ vengono replicati tutti i dati e gli stati richiesti per le operazioni del broker su tutti i nodi presenti.
Il funzionamento di RabbitMQ seque il modello di AMQP, il quale prevede che i messaggi pubblicati dai produttori vengano inviati a delle entità chiamate *Exchanges* che instradano i messaggi alle *code* dei consumatori, in base a regole dette *binding*.
Supporta lo scambio di messaggi in tutte le sue forme:

- tramite le code di lavoro condivise, in cui più consumatori ricevono i messaggi da una stessa coda in modo che i messaggi vengano bilanciati tra loro;
- tramite il modello Publish/Subscribe, in cui ad ogni consumatore è assegnata una coda dedicata e l'exchange all'interno del broker replica i messaggi e li instrada a tutte o solo ad alcune code, in base ai bindings definiti, che permettono di realizzare *message filtering* in base al tipo o al contenuto del messaggio;
- tramite RPC (Remote Procedure Call), in cui il produttore che pubblica il messaggio di richiesta rimane in attesa che il consumatore riceva il messaggio e che invii la risposta.

La ricezione dei messaggi da parte dei consumatori avviene tramite la callback `handleDelivery(...)` che viene richiamata ogni volta che è disponibile un nuovo messaggio nella coda del consumatore. Una volta che il consumatore riceve correttamente un messaggio, questi notifica il broker tramite un meccanismo di acknoledgement in modo che il messaggio venga rimosso dalla coda.

[5]sito: http://www.rabbitmq.com
[6]sito: http://www.amqp.org

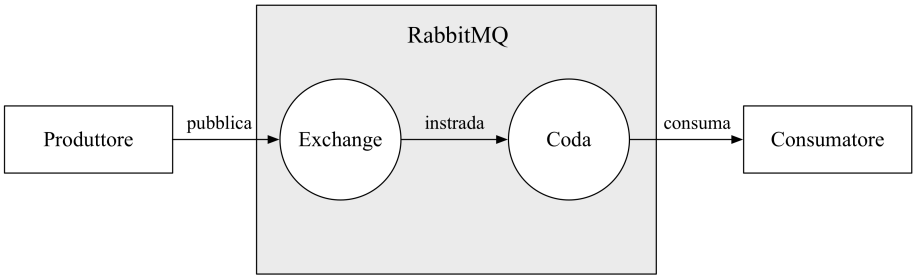

Figura 1.7: Funzionamento di RabbitMQ

3 MAS ad eventi: il caso di ELDA

ELDA[8] (Event-driven Lightweight Distilled Statecharts-based Agent) è un modello basato su eventi il cui obiettivo è quello di modellare MAS all'interno di ambienti di elaborazione aperti e dinamici. È incentrato sul concetto di agenti leggeri guidati dagli eventi, realizzati come entità autonome con singolo thread che interagiscono tramite eventi asincroni, eseguono elaborazioni su reazione agli eventi ricevuti e hanno la capacità di migrare.
È corredato da uno strumento denomitato *ELDATool*, realizzato in Java, che è composto da un framework (ELDAFramework) per la modellazione visuale del comportamento e da un ambiente di simulazione (ELDASim) per la validazione e valutazione del MAS modellato.

Il modello ELDA è basato su tre modelli: *Behavioural*, *Interaction* e *Mobility*. Il modello Behavioural permette la definizione del comportamento dell'agente tramite la definizione di stati, di transizioni tra gli stati e di reazioni. In ELDA una transizione da uno stato ad un altro si verifica all'occorrenza di un particolare evento e comporta una reazione dell'agente, ovvero un'attività atomica in cui l'agente può svolgere computazioni, generare altri eventi o effettuare una migrazione. Sostanzialmente, il comportamento di un agente definisce in che modo egli reagisce ad un determinato insieme di eventi.
Il modello Interaction permette la multi-coordinazione tra gli agenti e tra agenti e gli altri componenti del sistema tramite lo sfruttamento di diversi modelli di coordinazione: Message Passing, Publish/Subscribe, Tuple-Based, RPC e RMI. È basato su eventi asincroni, che formalizzano sia le iniziative dell'agente, tramite gli *Internal Events*, sia le richieste da e verso gli altri

agenti, tramite i *Management, Coordination e Exception Events*. Tutti i tipi di eventi sono ulteriormente classificati in `OUT-events` che sono gli eventi generati dall'agente poi inviati ad altri agenti o componenti e in `IN-events` che invece sono gli eventi ricevuti dall'agente.
Infine il modello Mobility è basato su una "mobilità a grana grossa", ovvero una mobilità forte capace di conservare lo stato di esecuzione dell'agente che può verificarsi sulla base di una *azione* (i.e. un insieme di istruzioni eseguite in maniera atomica) dell'agente. In pratica, i punti di migrazione degli agenti combaciano con la fine dell'esecuzione di una azione atomica dell'agente, nel momento in cui egli può elaborare gli eventi di tipo `Move`. La migrazione degli agenti ELDA può essere sia *autonoma*, ovvero iniziata dall'agente stesso, che *passiva*, ovvero imposta dal sistema o indotta da altri agenti.

Questi tre modelli sono basati sul formalismo DSC[9] (Distilled StateCharts) che deriva direttamente dagli StateCharts[10] e presenta, tra le altre, le seguenti caratteristiche chiave:

- le transizioni tra stati sono basate su regole *ECA* definite come E[C]/A: quando un evento E occorre e la condizione C è valida si esegue la transizione e si svolge l'azione A corrispondente;
- gli stati non devono includere attività o azioni di entrata/uscita: l'attività è portata avanti esclusivamente dalle azioni atomiche legate alle transizioni;
- la semantica d'esecuzione è di tipo *Run-To-Completion* (RTC): un evento può essere processato solo se l'elaborazione dell'evento precedente è stata interamente completata. La sequenza delle operazioni che parte dal fetching di un evento dalla coda degli eventi fino al completo processamento è definito *Run-To-Completion Step* e corrisponde ad un'unica azione atomica per il sistema;
- gli eventi vengono ricevuti implicitamente ed in maniera asincrona tramite la coda degli eventi;
- per permettere esplicitamente di emettere eventi in maniera asincrona si utilizza la primitiva `generate(<E>(<PARAMS>))`, dove `E` è l'istanza dell'evento e `PARAMS` sono i parametri che lo caratterizzano.

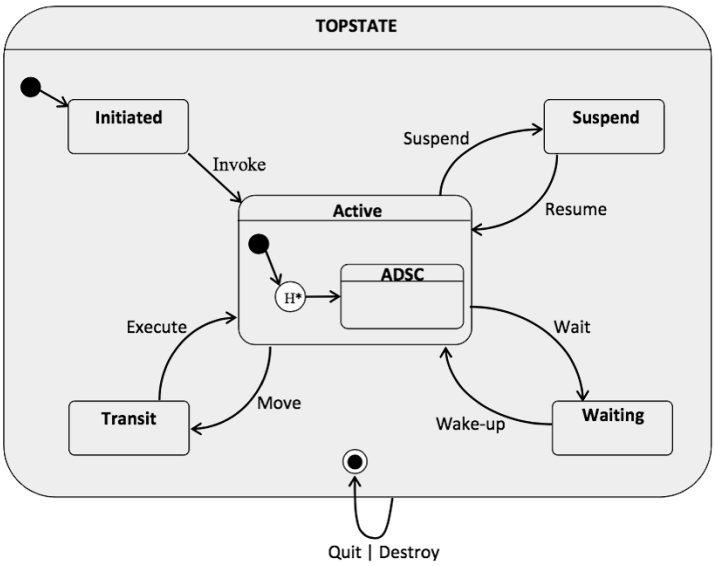

Figura 1.8: Comportamento di un agente ELDA basato sul template FIPA.

Ogni comportamento di un agente ELDA è modellato secondo una versione estesa del *ciclo di vita dell'agente FIPA*[11] nel quale lo stato *ACTIVE* è sempre acceduto tramite uno pseudostato *DHS* (Deep History pseudoState) per ripristinare l'esecuzione dello stato dell'agente dopo una migrazione e, in generale, dopo una sospensione dell'agente. In particolare, lo stato ACTIVE contiene il DSC attivo (ADSC), stato composito verso il quale punta la Default Entrance del DHS; durante la modellazione del comportamento dell'agente, si dovrà quindi solo ridefinire lo stato ADSC, in modo che l'intero behaviour dell'agente rimanga comunque coerente con il ciclo di vita proposto dal modello FIPA.

Le interazioni tra gli agenti e tra gli agenti e gli altri componenti del sistema sono basate completamente su eventi, i quali possono essere di 4 tipologie: *Internal Events*, *Management Events*, *Coordination Events* e *Exception Events*.

Gli Internal Events rappresentano le iniziative prese dall'agente, sono gli eventi che l'agente invia a se stesso per guidare proattivamente il proprio comportamento. Nel momento in cui è generato, l'evento viene inserito nella coda degli eventi dell'agente generante, quindi gli Internal Events possono essere considerati sia `OUT-Event` che `IN-Event`.

I Managements Events includono richieste da e verso altri agenti che ri-

guardano la gestione del ciclo di vita degli agenti, ovvero le operazioni di creazione, clonazione, migrazione, sospensione e distruzione, la gestione delle attività temporizzate, tramite l'utilizzo di timer e l'accesso alle risorse, e.g. file, console, database e sensori/attuatori.
I Coordination Events consentono la coordinazione tra gli agenti e tra gli agenti e gli altri componenti del sistema in base a diversi modelli di coordinazione. Per l'interazione inter-agenti ELDA offre tre modelli di coordinazione: (i) il modello *Direct* per la comunicazione diretta tra gli agenti secondo i paradigmi del Message-Passing asincrono e del Remote Procedure Call (RPC) sincrono, (ii) il modello *Publish/Subscribe* per la pubblicazione e sottoscrizione a determinati eventi o topic e (iii) il modello *Tuple-based* basato sulle primitive Linda. Per la coordinazione tra agenti e altri componenti del sistema invece sono disponibili i modelli *RMI Object* e *Web Services*.
Infine, gli Exception Events sono modellati come `IN-Events` e vengono ricevuti dagli agenti quando risulta impossibile l'esecuzione di un servizio richiesto.

Capitolo 2

Analisi e integrazione

In questo capitolo viene inizialmente svolta un'analisi dei framework introdotti nel capitolo precedente, nell'ottica dell'integrazione tra i sistemi multi-agente ed i sistemi ad eventi. Si esamina la presenza del concetto di evento all'interno di JADE e TuCSoN, si studiano i concetti di autonomia dei componenti e di supporto alla distribuzione del sistema in Kafka e RabbitMQ e si analizzano le caratteristiche più importanti del modello ELDA. In seguito viene mostrato un primo esempio di integrazione tra MAS ed EBS, grazie all'implementazione di uno scenario in cui si realizza l'interazione tra gli agenti tramite Kafka e JADE.

1 Analisi Framework

1.1 Il concetto di evento nei MAS

Sebbene non sia generalmente formalizzato in modo esplicito, il concetto di evento è largamente presente all'interno dei Sistemi Multi-Agente. Considerando quanto scritto nel capitolo precedente, sul fatto che gli agenti sono in grado di interagire tra di loro e con l'ambiente e sono sensibili ai cambiamenti ambientali, è facile notare come le interazioni e i cambiamenti ambientali rappresentino in sostanza il verificarsi di avvenimenti rilevanti per il sistema, ovvero di eventi. Agenti e ambiente risultano quindi essere le sorgenti degli eventi all'interno dei MAS.

In JADE non è presente alcuna astrazione specifica che permette di modellare l'ambiente: l'unica astrazione disponibile capace di svolgere attività sono gli agenti, di conseguenza è necessario realizzare degli agenti anche per catturare i cambiamenti ambientali. Questo significa che in JADE gli "eventi" vengono gestiti interamente dal livello della comunicazione tramite message passing asincrono, quindi dall'ACC, che può quindi essere considerato l'*event mediator* di un sistema JADE.

In TuCSoN il concetto di evento è presente in modo esplicito all'interno dei centri di tuple tramite il linguaggio ReSpecT e rappresenta la reificazione di una attività di un agente o del cambiamento di stato di una proprietà di una risorsa ambientale. I componenti che si occupano di effettuare questa reificazione sono gli artefatti assegnati agli agenti e alle risorse ambientali, ovvero gli ACC e i trasducer. In base all'origine degli eventi è possibile anche distinguerli in eventi esterni ed eventi interni: gli eventi esterni sono quelli che hanno origine al di fuori dello spazio di coordinazione TuCSoN e che vengono reificati dagli ACC o dai trasducer e inviati verso i centri di tuple, mentre gli eventi interni sono quelli che nascono e vengono scambiati all'interno dei centri di tuple stessi e che eventualmente escono verso agenti e risorse per motivi di coordinazione.

1.2 Gli EBS dal punto di vista dei MAS

I Sistemi ad Eventi offrono tre astrazioni principali: produttori, consumatori e bus degli eventi. Osservando la natura e il ruolo di tali astrazioni dal punto di vista dei MAS è possibile riconoscere delle forti similarità tra le due architetture. I componenti di un EBS, i produttori e i consumatori, possono essere visti come agenti o artefatti di un MAS. Essi infatti causano le dinamiche del sistema (i produttori generano gli eventi) e gestiscono le interazioni tra i componenti (i consumatori definiscono le regole di comunicazione e coordinazione tramite le sottoscrizioni); inoltre il bus degli eventi può essere visto come il coordination media di un MAS, essendo l'intermediario di tutte le interazioni dei componenti. Egli cattura gli eventi generati dai produttori e li consegna ai consumatori rispettando le regole definite dalle sottoscrizioni.

Sempre secondo la prospettiva dei MAS, di seguito viene effettuata l'analisi di Kafka e RabbitMQ in relazione all'autonomia concessa ai componenti del sistema (ovvero il controllo che i componenti hanno del proprio flusso di esecuzione) e alla capacità di un sistema realizzato con tali framework di operare in uno scenario fortemente distribuito.

In Kafka un consumatore lavora emettendo richieste di "fetch" ai broker leader delle partizioni che vuole consumare: ogni richiesta si basa sulla posizione di offset che il consumatore ha raggiunto fino a quel momento. Si noti che il consumatore ha il completo controllo sulla sua posizione di offset e all'occorrenza può decidere di tornare indietro e ri-consumare i messaggi (che vengono conservati dal broker a prescindere che siano stati consumati o meno). Questo stile di funzionamento permette al componente consumatore di mantenere il controllo del proprio flusso di esecuzione, così da effettuare le richieste di consumo quando lo ritiene più opportuno. Egli può così ottimizzare le proprie prestazioni ed eventualmente sospendere il consumo dei messaggi al verificarsi di determinate condizioni, ad esempio quando deve svolgere un'attività critica.
Kafka è progettato come sistema distribuito e viene eseguito di default in un cluster. Un cluster Kafka risulta essere notevolmente scalabile, dato che le partizioni che compongono i topic vengono distribuite automaticamente attraverso i nodi disponibili per bilanciare il carico di lavoro, e presenta un grado elevato di fault tolerance, grazie al fatto che le partizioni possono essere replicate su un numero configurabile di nodi secondo il meccanismo di *leader-followers*.

In RabbitMQ è invece presente un meccanismo più tradizionale per la ricezione dei messaggi da parte dei consumatori. Ogni volta che un messaggio viene indirizzato dall'exchange alla coda di un cosumatore, il broker richiama la callback per la gestione del messaggio definita dal consumatore. Il consumatore quindi durante il suo ciclo di vita rimane in *idle*, aspettando passivamente che il broker lo risvegli all'arrivo di un nuovo messaggio. Questo porta all'*inversione del controllo* all'interno del componente, è infatti il broker che definisce quando i consumatori devono agire e in che modo lo

devono fare.
RabbitMQ offre un buon supporto per la realizzazione di un sistema distribuito, infatti può essere configurato sia come cluster che come federation. Configurando RabbitMQ come un cluster, gli utenti del sistema interagiscono tra loro tramite un unico broker logico, che ha accesso a tutte le code dei consumatori presenti in ogni nodo. Il sistema risulta così facilmente scalabile e presenta fault tolerance, dato che tutti i dati e gli stati del broker logico vengono replicati su tutti i nodi del cluster. Questa soluzione presenta però un vincolo sulle possibilità di distribuzione dei nodi del cluster: la comunicazione realizzata tramite message-passing di Erlang quindi il collegamento di rete tra i nodi deve essere molto affidabile, rendendo il cluster un'opzione praticabile solo all'interno di una LAN. Inoltre tutti i nodi del cluster devono condividere i cookie Erlang ed eseguire tutti la stessa versione di RabbitMQ ed Erlang. Configurando invece RabbitMQ come una federation, il sistema si compone di più broker, logicamente separati, che comunicano tra loro tramite AMQP. Questo permette al sistema di essere maggiormente distribuito anche attraverso nodi geograficamente distanti tra loro e collegati tramite WAN. Non è presente però fault tolerance a livello di sistema, questa deve essere realizzata a livello di nodo, ad esempio tramite la creazione di un cluster come descritto in precedenza. Si noti che all'interno di una federation, un clienti che si collega ad un particolare broker logico può vedere solamente le code dei consumatori presenti in quel broker e non può quindi comunicare con i consumatori presenti su altri nodi.

1.3 L'interpretazione di ELDA

Il fatto che in ELDA il comportamento di un agente venga definito tramite la modellazione di un DSC, permette di facilitare notevolmente la sua creazione, ma allo stesso tempo pone forti limiti alle capacità dell'agente. Il comportamento di un agente può essere descritto come un insieme strettamente sequenziale di reazioni ad eventi. Se si aggiunge inoltre che la ricezione degli eventi è completamente implicita, si può facilmente dedurre che gli agenti ELDA soffrono di *inversione del controllo*: il flusso di controllo degli agenti viene comandato dal flusso degli eventi, che quindi decide quando e come gli

agenti debbano agire. Si verifica così una violazione della caratteristica di autonomia dell'agente, che risulta incapace di comportamenti proattivi, se non simulandoli tramite la generazione di Internal Events (eventi che un agente invia a se stesso). È importante però notare che questi eventi non hanno alcun tipo di priorità sulle altre tipologie di evento, ma vengono semplicemente inseriti nella coda degli eventi. Non esiste dunque possibilità per l'agente di comandare liberamente il proprio flusso di controllo, ad esempio scegliendo di 'sospendere" la ricezione degli eventi per svolgere un'attività critica.

ELDA fornisce un modello di coordinazione multi-paradigma, grazie al supporto per la comunicazione inter-agente tramite il modello Direct (che realizza il Message Passing ed RPC), il modello Publish/Subscribe e il modello Tuple-based.
Nel modelli Direct e Publish/Subscribe viene realizzata una coordinazione di tipo soggettivo, dove è l'agente che si deve occupare della gestione di tutte le questioni che riguardano la coordinazione. In questo modo l'agente può determinare quale sia la migliore linea di (inter-)azione per cercare di raggiungere i suoi obiettivi. Il peso della coordinazione viene quindi suddiviso tra tutti i componenti coordinabili. Il modello Direct è supportato dai seguenti Coordination Events:

- gli eventi `OUT-MSGRequest` e `IN-Msg` permettono lo scambio di messaggi asincroni con altri agenti.;
- gli eventi `OUT-RPCRequest` e `IN-RPCResult` permettono la gestione dei messaggi sincroni con altri agenti, in particolare l'evento `RPCRequest` consiste in un'operazione bloccante per l'agente che la effettua, il quale è costretto a rimanere in attesa del correlativo evento `RPCResult`.

Si noti che, a causa del funzionamento dell'agente ELDA su un singolo thread e della totale assenza di supporto alle operazioni concorrenti, un'operazione bloccante come quella relativa all'evento `OUT-RPCRequest`, blocca interamente l'agente impedendogli di compiere qualsiasi altro tipo di azione mentre è in attesa di una risposta che, data la natura distribuita e eterogenea del sistema, potrebbe anche non arrivare mai.
I Coordination Events del modello Publish/Subscribe sono `OUT-Subscribe`,

`OUT-UnSubscribe`, `OUT-Publish` e `IN-EVTNotification` e sono tutti di tipo asincrono. Gli eventi `Subscribe` e `UnSubscribe` permettono di effettuare sottoscrizioni e de-sottoscrizioni ad eventi e topic, `Publish` consente di pubblicare in un particolare topic ed infine l'evento `EVTNotification` consiste in una notifica riguardante un topic precedentemente sottoscritto.
Infine, il modello Tuple-based di ELDA è basato su TuCSoN, il quale, come descritto nel capitolo precedente, estende il modello Linda tramite centri di tuple ReSpecT che incapsulano le politiche e i meccanismi di coordinazione, permettendo così di disaccoppiare il sincronismo di un'operazione di coordinazione dalla semantica del Coordination Event usato dall'agente per richiederla. I centri di tuple TuCSoN offrono quindi coordinazione di tipo oggettiva come servizio. I Coordination Events disponibili per utilizzare il modello Tuple-based sono `OUT- In`, `OUT-Out` e `OUT-Rd` e `IN-ReturnTuple`. `In`, `Out` e `Rd` corrispondono alle primitive Linda rispettivamente per l'inserimento, l'estrazione e la lettura di tuple. `In` e `Rd` possono essere sia sincroni che asincroni, mentre `Out` è solamente asincrono. `ReturnTuple` incorpora le tuple associate ad un evento `In` o `Rd` precedentemente generato.

2 Integrazione tra Kafka e JADE

Alla luce delle analisi svolte, Kafka risulta essere il candidato ideale, tra i framework ad eventi, per realizzare l'integrazione tra MAS ed EBS, soprattutto per la gestione del consumo dei messaggi basata su richieste a polling del consumatore, fondamentale per consentire di rispettare la caratteristica fondamentale di autonomia di un agente.

2.1 Le API di Kafka

Di seguito vengono descritte ed analizzate le principali API di Kafka, facendo riferimento alla versione 0.10.0 attualmente disponibile.

Il primo passo per poter pubblicare i messaggi su Kafka è definire le proprietà che verranno poi passate al produttore. Ci sono 3 proprietà obbligatorie:

- `bootstrap.servers`: è la lista dei broker Kafka. Non deve includere necessariamente tutti i broker presenti nel cluster, il produttore interrogherà i broker di questa lista per ottenere informazioni riguardo gli altri broker.
- `key.serializer` e `value.serializer`: i broker Kafka si aspettano un array di byte come chiave e valore del messaggio, tuttavia l'interfaccia KafkaProducer, tramite l'uso di tipi parametrizzati, permette di inviare oggetti Java come chiave e valore. Questo permette di scrivere codice più comprensibile ma vuole dire anche che il produttore deve sapere come convertire questi oggetti in array di byte. Il package di Kafka offre 3 serializer built-in: ByteArraySerializer (che sostanzialmente non fa molto), StringSerializer e IntegerSerializer per poter usare i tipi comuni per i messaggi. Si noti che è necessario definire il serializer per la chiave del messaggio anche se non si ha intenzione di usarla nell'invio.

Quindi il produttore viene creato come indicato dal seguente snippet di codice:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
KafkaProducer<String, String> producer = new KafkaProducer<String, String>(
    props);
```

In seguito, il metodo più semplice per inviare un messaggio è il seguente:

```
ProducerRecord<String, String> record = new ProducerRecord<>("topicName", "
    messageValue");
try {
  producer.send(record);
} catch (Exception e) {
  e.printStackTrace();
}
```

Il metodo `send()` del produttore che realizza l'invio asincrono del messaggio accetta oggetti di tipo `ProducerRecord` che rappresentano il messaggio che si vuole pubblicare. La classe `ProducerRecord` dispone di costruttori multipli dato che per ogni messaggio ci son due parametri obbligatori e due opzionali: quelli obbligatori sono il nome del topic dove deve essere pubblicato il

messaggio e il contenuto del messaggio stesso, quelli opzionali sono la chiave del messaggio e la partizione di destinazione all'interno del topic.
Quando il produttore ha terminato il suo ciclo di vita è necessario chiamare il metodo `close()` per fermare il produttore non appena tutte le richieste di invio asincrone sono state completate.

L'inizializzazione di un consumatore Kafka avviene in maniera del tutto similare al produttore:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "groupName");
props.put("key.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
    StringSerializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,String>(
    props);
```

L'unica proprietà obbligatoria in più da definire, rispetto al produttore, è `group.id` che indica il nome del gruppo di consumatori di cui il consumatore farà parte.
Affinché un consumatore possa ricevere i messaggi dai broker, è necessario che venga effettuata la sottoscrizione ai topic di interesse tramite il metodo:

```
consumer.subscribe(Arrays.asList("topicName1", "topicName2"));
```

In seguito la ricezione dei messaggi viene effettuata tramite un loop di polling, in modo che il consumatore interroghi continuamente il broker per sapere se ci sono nuovi messaggi disponibili.

```
while (true) {
  ConsumerRecords<String, String> records = consumer.poll(timeout);
  for (ConsumerRecord<String, String> record : records){
      ...
  }
}
```

Nel caso non ci siano nuovi messaggi all'interno del broker, il consumatore rimane in attesa che arrivino per l'intervallo di tempo indicato dal parametro *`timeout`* passato al metodo `poll()`. Si noti che tale parametro può essere anche pari a 0, in modo che la richiesta non sia bloccante. È importante sottolineare che, nell'ottica dell'integrazione tra Kafka e JADE, il fatto

che il metodo `poll(timeout)` *possa* essere bloccante comporta un notevole rischio per l'autonomia dell'agente. Infatti, nel caso in cui venga definito un *timeout* non nullo, verrebbe sospesa l'intera esecuzione dell'agente, non solamente quella del comportamento in cui il metodo viene richiamato, impedendo quindi all'agente di svolgere altri compiti durante l'attesa.
Infine, quando il consumatore ha terminato il suo ciclo di vita, è necessario richiamare il metodo `close()` così da chiudere la connessione col cluster e indicare la terminazione agli altri eventuali componenti del gruppo di consumatori di cui faceva parte.

2.2 Kafka4JADE

Alla luce dei rischi che il metodo `poll(timeout)` può comportare per l'autonomia degli agenti, per integrare Kafka e JADE è necessario implementare un componente che agisca come intermediario tra l'agente JADE ed il consumatore Kafka. In questo modo è possibile disaccoppiare la semantica bloccante dell'invocazione del metodo, usato per effettuare la richiesta di fetch, dall'effettivo sincronismo dell'operazione di coordinazione, analogamente a quanto succede in TuCSoN con l'Agent Coordination Context.
Tale componente è realizzato tramite la classe `KafkaConsumerAssistant` ed offre il metodo `consume(timeout)` per effettuare le richieste al cluster Kafka. Nel momento in cui viene effettuata una richiesta con timeout non nullo, ad essere sospeso sarà solamente il behaviour JADE che ha invocato il metodo e non l'intero agente JADE, così come mostrato nella figura 2.1.
Si noti che questo componente è concepito più per essere utilizzato come *supporto* ai singoli behaviour che all'agente nel suo complesso. Creando un `KafkaConsumerAssistant` per ogni comportamento che vuole sfruttare il paradigma del publish/subscribe è possibile realizzare, all'interno di un unico agente JADE, comportamenti differenti che consumano messaggi da topic differenti in maniera del tutto indipendente tra loro.

Di seguito viene presentato un semplice esempio dell'utilizzo del componente appena introdotto, tramite l'implementazione di uno scenario in cui avviene un dibattito tra due agenti che discutono tra loro riguardo ad argomenti

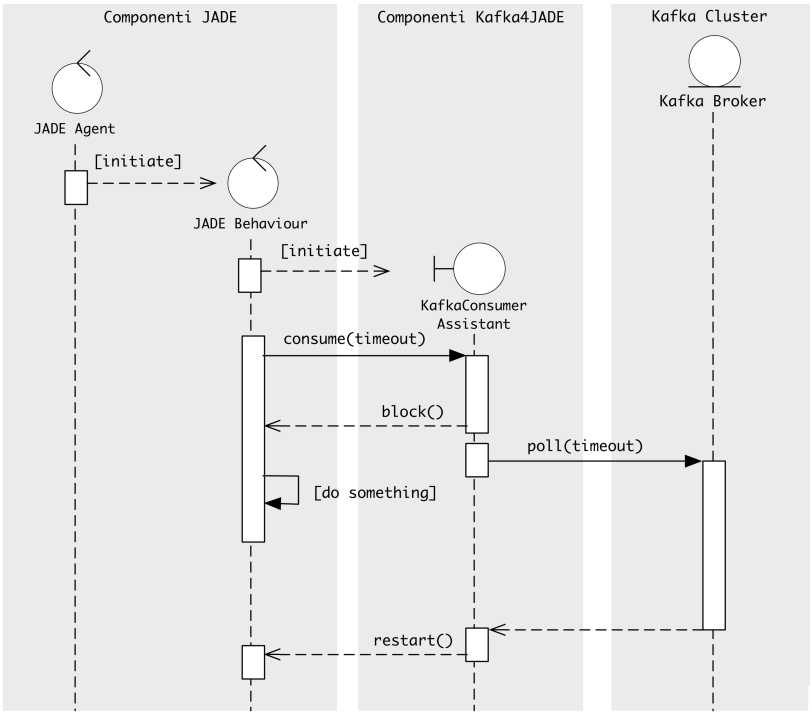

Figura 2.1: Integrazione Kafka4JADE che permette di preservare il modello di autonomia di JADE nelle richieste di fetch dei messaggi Kafka con timeout non nullo

proposti da un terzo agente, moderatore del dibattito.
Tecnicamente, gli argomenti di discussione vengono pubblicati dal moderatore all'interno di un topic Kafka e vengono consumati dai partecipanti al dibattito, i quali discutono tra di loro scambiandosi messaggi tramite l'Agent Communication Channel di JADE.

Nella figura 2.2 sono mostrate le interazioni degli agenti che compongono il sistema. Il mediatore, realizzato tramite la classe `DebateModeratorAgent`, non appena viene inizializzato, controlla che il topic "*debateTopic*" sia presente all'interno del cluster Kafka ed eventualmente ne richiede la creazione, in seguito inizializza il produttore Kafka ed infine aggiunge il comportamento `ChooseArgumentBehaviour` (che estende `TickerBehaviour`) impostando un intervallo di 10 secondi. Ad ogni *tick* del behaviour avviene la pubblicazione di un nuovo argomento nel broker Kafka tramite il produttore creato in sede

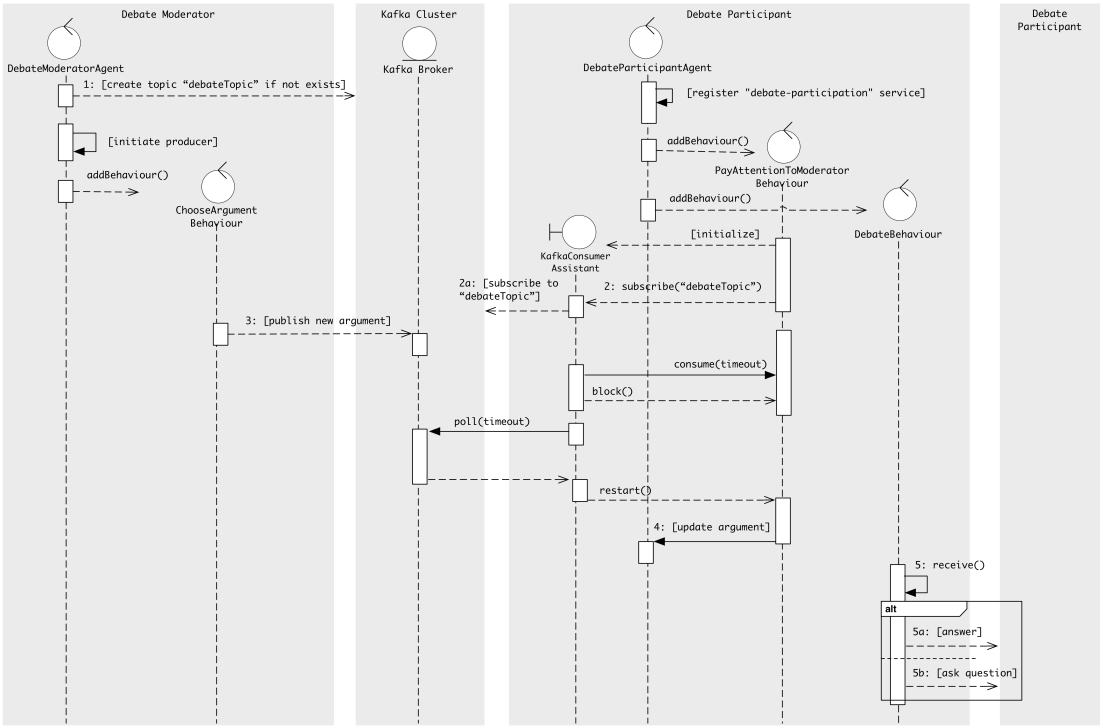

Figura 2.2: Diagramma di interazione degli agenti all'interno del sistema durante un dibattito

di `setup()` dell'agente.

Gli agenti che partecipano al dibattito sono creati tramite la classe `DebateParticipantAgent`. In fase di `setup()` vengono effettuate le seguenti operazoni:

- si registra il servizio "*debate-participation*" all'interno delle *pagine gialle* di JADE;
- si aggiungono i comportamenti `PayAttentionToModeratorBehaviour` e `DebateBehaviour`.

`PayAttentionToModeratorBehaviour`, che estende la classe `CyclicBehaviour`, si avvale del `KafkaConsumerAssistant` per realizzare il comportamento del consumatore Kafka in modo da preservare l'autonomia dell'agente. Quando il comportamento viene inizializzato, si effettua la sottoscrizione al topic "*debateTopic*". In seguito, ad ogni iterazione viene eseguito il metodo `consume(`*`timeout`*`)` per recuperare gli argomenti che vengono proposti dal

mediatore quando vengono pubblicati. Ogni volta che viene recuperato un messaggio, viene aggiornato l'argomento di riferimento che viene utilizzato nel dialogo con gli altri agenti.
Il comportamento `DebateBehaviour` estende invece la classe `TickerBehaviour` e viene eseguito dall'agente ogni 2 secondi. Ad ogni tick l'agente controlla nella propria mailbox se ha ricevuto dei messaggi dagli altri partecipanti al dibattito: nel caso ce ne siano, l'agente prepara la risposta da restituire al mittente; se invece la mailbox è vuota, l'agente controlla le *pagine gialle* di JADE per ottenere l'elenco di tutti gli agenti che offrono il servizio *debate-participation*, quindi ne sceglie casualmente uno e gli invia una domanda riguardo l'argomento del dibatto valido in quel preciso momento.

Si noti che il `DebateParticipantAgent` rappresenta l'esempio chiave dell'integrazione tra Kafka e JADE. Infatti esso può sfruttare concorrentemente entrambi i paradigmi di coordinazione che caratterizzano i due framework, senza intaccare l'autonomia dell'agente, tramite all'utilizzo del componente `KafkaConsumerAssistant`.

Capitolo 3

Caso di studio: The Basketball Playground Scenario

Lo scopo di questo capitolo è quello di fornire un esempio pratico di come sia possibile effettuare l'integrazione tra Kafka, TuCSoN e JADE. Il caso di studio preso in esame permette di realizzare un Sistema Multi-Agente che sfrutta i tre modelli di coordinazione messi a disposizione dai framework sopracitati: il modello Publish/Subscribe, il modello Tuple-based e il modello di Message-passing asincrono.

1 Descrizione

Lo scenario che caratterizza il caso di studio simula un playground di città in cui si svolge una partita di pallacanestro. All'interno del playground sono presenti un agente *custode*, che si occupa di rendere disponibile il campo da gioco, e un numero n di agenti che, in base all'ordine di arrivo al campo, partecipano o meno alla partita.
Durante lo svolgimento della partita, gli agenti *giocatori* si passano la palla tra compagni di squadra e tutti gli avvenimenti che accadono (passaggi effettuati, tiri realizzati o sbagliati e rimbalzi presi) vengono osservati, oltre che dai giocatori stessi, anche da tutti gli altri agenti che decidono di seguire la partita.

2 Analisi dei requisiti

Analizzando le caratteristiche dello scenario è possibile individuare tre situazioni chiave in cui, per il corretto funzionamento del sistema, è importante che l'interazione tra gli agenti sia realizzata tramite il giusto modello di coordinazione: *(i)* il momento in cui gli agenti si presentano al campo da gioco e deve iniziare la partita, *(ii)* il passaggio del pallone tra gli agenti giocatori che compongono una squadra e *(iii)* la propagazione degli eventi di gioco a tutti gli agenti che sono interessati.

Quando gli agenti si presentano al playground è necessario che questi interagiscano in modo che venga deciso senza alcuna ambiguità quali siano i dieci agenti giocatori che posso prendere parte alla partita e quali invece ne sono esclusi. Deve essere anche garantito che la partita abbia inizio solamente quando sono stati trovati tutti e dieci gli agenti partecipanti.

Per quanto riguarda il passaggio del pallone tra gli agenti che appartengono ad una stessa squadra, è importante notare che, affinché questo succeda, deve esserci un accoppiamento diretto tra il giocatore che effettua il passaggio e quello che lo riceve, rendendo quindi necessario che ogni agente che partecipi alla partita conosca (almeno) l'identità di tutti i suoi compagni di squadra.

Infine, in merito agli eventi di gioco (passaggi effettuati, tiri realizzati o sbagliati e rimbalzi presi) che vengono generati dai giocatori durante la partita, è necessario che questi siano catturabili da qualunque agente presente al playground, a prescindere che i giocatori in campo sappiano o meno della sua presenza. È fondamentale quindi che ci sia un totale disaccoppiamento tra gli agenti che generano gli eventi e coloro che li ricevono.

3 Progettazione del MAS

Dall'analisi dei requisiti risulta quindi che il sistema debba essere caratterizzato da una coordinazione multi-paradigma in modo che a seconda della situazione, l'interazione tra gli agenti possa essere gestita in maniera ottima-

le. Questo può essere possibile tramite l'integrazione tra Kafka, TuCSoN e JADE.

La coordinazione iniziale per l'organizzazione della partita viene realizzata tramite il modello tuple-based offerto da TuCSoN. Nel momento in cui l'agente custode "apre i cancelli" del playground, inserisce all'interno di un nodo TuCSoN dieci tuple rappresentanti i giocatori che possono partecipare alla partita. Ognuna di queste tuple è caratterizzata dal nome di una delle due squadre e saranno ovviamente suddivise in numero equo. Il custode inserisce anche una tupla che rappresenta il pallone di gioco. Quando i giocatori si presentano al campo, dichiarano la loro intenzione di giocare cercando di estrarre una "tupla squadra" dal centro di tuple; se l'estrazione ha successo allora l'agente potrà partecipare alla partita, altrimenti rimarrà semplicemente in attesa come spettatore. In seguito all'estrazione, i giocatori inseriscono a loro volta una tupla che indica il loro nome e la squadra di appartenenza. Questo permette loro di sapere quanti siano gli agenti in attesa di giocare e quale sia la loro identità e la squadra di appartenenza. Nel momento in cui il numero delle "tuple giocatore" è pari a quello dei giocatori necessari, la partita può avere inizio. Infine, appena prima di iniziare a giocare, ogni giocatore cerca di estrarre la "tupla pallone", inserita inizialmente dal custode, e l'agente che avrà successo sarà il primo a poter svolgere un'azione di gioco.

Grazie alla coordinazione iniziale con TuCSoN, ogni agente giocatore è a conoscenza delle identità di tutti gli altri partecipanti alla partita. Questo permette di realizzare l'interazione per lo scambio del pallone tramite il modello del message-passing, sfruttando l'Agent Communication Channel di JADE. Quando il giocatore che detiene il possesso del pallone decide di effettuare un passaggio, egli dovrà quindi selezionare casualmente il nome di un compagno di squadra ed inviargli un messaggio che sostanzialmente rappresenta il pallone stesso.

Infine, gli eventi durante la partita vengono gestiti tramite il modello di coordinazione a Publish/Subscribe con Kafka. I giocatori che svolgono le azioni di gioco rilevanti, pubblicano i relativi messaggi all'interno di un topic dedicato

al playground nel Kafka cluster. Ogni agente interessato, che sia un giocatore o un semplice spettatore, può quindi sottoscriversi al topic e ricevere i suddetti messaggi in maniera del tutto indipendente.

Si noti che in base alle tipologie di coordinazione che si svolgono all'interno del sistema nelle varie situazioni, è possibile individuare due *sotto-sistemi MAS* idealmente differenti: il primo è quello che riguarda l'organizzazione della partita, che si può considerare maggiormente *event-driven*, in cui gli agenti seguono il flusso degli eventi per scoprire se possono o meno prendere parte alla partita; mentre il secondo riguarda la partita stessa, in cui sono più forti le caratteristiche tipiche dei MAS nel modello dell'interazione tra gli agenti.

4 Implementazione

L'integrazione tra Kafka, TuCSoN e JADE è stata realizzata tramite la libreria TuCSoN4JADE[1] e tramite l'utilizzo delle API Kafka (per l'implementazione dei produttori) e componente `KafkaConsumerAssistant` (per i consumatori) come descritto nel capitolo precedente.

Nella figura 3.1 è possibile osservare le interazioni degli agenti che compongono il sistema. Per facilitarne la comprensione sono state evidenziate in arancione le interazioni relative all'organizzazione di una partita svolte tramite TuCSoN, in azzurro quelle relative al passaggio del pallone tramite JADE ed infine in verde le interazioni riguardanti gli eventi della partita svolte mediante Kafka. Si noti che in tale diagramma delle interazioni è mostrato il caso in cui l'agente giocatore riesce ad ottenere la "tupla squadra" per poter partecipare alla partita.
Di seguito vengono descritte in maniera approfondita le classi ed i behaviour che caratterizzano le due tipologie di agenti del sistema: l'agente custode e l'agente giocatore.

[1] sito: http://tucson4jade.apice.unibo.it

4.1 Agente custode

L'agente custode è implementato all'interno della classe `KeeperAgent`. Durante la fase di `setup()`, svolge le seguenti operazioni:

- avvia il nodo TuCSoN, nel caso questo non sia già correttamente attivo, e vi inserisce in maniera asincrona cinque tuple `team('WHITE')` e cinque tuple `team('BLACK')`, oltre alla tupla `ball(1)` che rappresenta il pallone di gioco;
- controlla che nel cluster Kafka sia presente il topic `playgroundTopic` relativo agli eventi di gioco ed in caso contrario ne richiede la creazione;
- aggiunge il comportamento `ObservePlaygroundBehaviour`.

```
protected void setup() {
  ...
  if (!this.helper.isActive("localhost", 20504, 10000)) {
    this.helper.startTucsonNode(20504);
  }
  ...
    for (int i = 0; i < this.players_per_team; i++) {
    LogicTuple intention = LogicTuple.parse("team('WHITE')");
    final Out out = new Out(this.tcid, intention);
    this.bridge.asynchronousInvocation(out);
  }
    for (int i = 0; i < this.players_per_team; i++) {
    LogicTuple intention = LogicTuple.parse("team('BLACK')");
    final Out out = new Out(this.tcid, intention);
    this.bridge.asynchronousInvocation(out);
  }
    LogicTuple intention = LogicTuple.parse("ball(1)");
  final Out out = new Out(this.tcid, intention);
  this.bridge.asynchronousInvocation(out);
    ...
    if (!AdminUtils.topicExists(zkUtils, this.playgroundTopic)){
    AdminUtils.createTopic(zkUtils, this.playgroundTopic, partitions,
        replication, topicConfiguration, null);
  }
  this.addBehaviour(new ObserveGameBehaviour());
}
```

`ObserveGameBehaviour` estende la classe `CyclicBehaviour`. Quando questo comportamento viene inizializzato, crea un'istanza del componente `KafkaConsumerAssistant` ed effettua la sottoscrizione al topic del playground. In seguito, ad ogni esecuzione del metodo `action()`, si eseguono le richieste a polling al broker Kafka per ottenere i messaggi presenti nel topic sottoscritto.

Si noti come, grazie al disaccoppiamento dall'effettiva richiesta al broker, sia possibile indicare un tempo di timeout molto elevato senza condizionare l'autonomia dell'agente, riducendo di conseguenza anche il numero di richieste di rete che vengono effettuate verso il cluster Kafka.

```
private class ObservePlaygroundBehaviour extends CyclicBehaviour {
  ...
  private KafkaConsumerAssistant kca;
  ...
  public ObservePlaygroundBehaviour() {
    ...
      kca = new KafkaConsumerAssistant(this, consumer_group);
      kca.subscribe(KeeperAgent.this.playgroundTopic);
  }

  public void action() {
    ConsumerRecords<String, String> records = kca.consume(Long.MAX_VALUE);
        ...
  }
}
```

4.2 Agente giocatore

L'agente giocatore è implementato nella classe `PlayerAgent`. All'inizio della sua vita, l'agente ha un unico comportamento, di tipo `ArrivedToPlaygroundBehaviour`.
All'interno di questo comportamento viene eseguita, tramite il bridge di TuCSoN4JADE, l'operazione `inp`, in maniera asincrona, per estrarre una tupla del tipo `team(S)` dal centro di tuple. Si noti che l'utilizzo della primitiva `inp` permette all'agente di coordinarsi rispettando la propria autonomia. Essa infatti consente di eseguire un'operazione di estrazione non bloccante: se non esistono tuple che corrispondono al template indicato, viene restituito immediatamente all'agente un messaggio di fallimento. L'utilizzo inoltre dell'invocazione asincrona permette di disaccoppiare ulteriormente il comportamento dell'agente dall'esecuzione dell'operazione.

```
private class ArrivedToPlaygroundBehaviour extends WakerBehaviour {

  public ArrivedToPlaygroundBehaviour(Agent a, long timeout) {
    super(a, timeout);
  }

  private static final long serialVersionUID = 1L;
```

```
  @Override
  protected void onWake() {
      ...
    LogicTuple team = LogicTuple.parse("team(S)");
    final Inp inp = new Inp(PlayerAgent.this.tcid, team);
    PlayerAgent.this.bridge.asynchronousInvocation(inp, new DraftBehaviour(
        PlayerAgent.this, 1000), PlayerAgent.this);
    ...
}
```

Il risultato della richiesta viene poi gestito all'interno del comportamento `DraftBehaviour` grazie alla callback `setTucsonOpCompletionEvent(...)` definita da TuCSoN4JADE. Se la tupla è stata estratta con successo, significa che l'agente può partecipare alla partita. Egli esegue quindi una `out` asincrona per inserire la tupla `player(name(nomeAgente), team(squadraAgente))` nel centro di tuple e passa al comportamento `WaitToPlayBehaviour`. Nel caso invece in cui la `inp` non abbia avuto successo, l'agente passa immediatamente al comportamento `ObserveGameBehaviour` tramite il quale osserva la partita da spettatore.

```
public class DraftBehaviour extends TickerBehaviour implements
    IAsynchCompletionBehaviour {
  ...
  public void onTick(){
    ....
    if (this.res.resultOperationSucceeded()){
      ...
      player = LogicTuple.parse("player(name('" + PlayerAgent.this.getAID().
          getName() + "'),team(" + PlayerAgent.this.team + "))");
      final Out out = new Out(PlayerAgent.this.tcid, player);
      PlayerAgent.this.bridge.asynchronousInvocation(out);
      ...
      PlayerAgent.this.addBehaviour(new WaitToPlayBehaviour(PlayerAgent.this
          , 1000));
    } else {
      PlayerAgent.this.addBehaviour(new ObserveGameBehaviour(PlayerAgent.
          this, 1000));
    }
    this.stop();
  }

  public void setTucsonOpCompletionEvent(TucsonOpCompletionEvent ev) {
    this.res = ev;
  }
}
```

Il comportamento `WaitToPlayBehaviour` permette all'agente di attendere che si raggiunga il numero corretto di giocatori per poter iniziare la partita e, nel frattempo, di conoscere l'identità di tutti i giocatori. Questo viene realizzato tramite tre `rdAll`: le prime due (una per squadra) servono per ottenere l'elenco delle tuple che corrispondono a `player(name(S), team(S))` e conoscere l'identità dei giocatori che parteciperanno alla partita, mentre la terza è relativa alle tuple di tipo `team(S)` ancora presenti nel centro di tuple che indicano quanti giocatori mancano ancora all'appello. Nel momento in cui sono arrivati tutti i giocatori ed è nota l'identità di compagni di squadra e avversari, l'agente passa al comportamento `TryToGetTheBall`.

Si noti che, anche se in questo caso la semantica di invocazione dell'operazione `rdAll` è di tipo sincrono, grazie alla presenza di TuCSoN4JADE l'autonomia dell'agente viene preservata ed è solamente il comportamento ad essere sospeso in attesa che vengano ricevute le risposte.

```
private class WaitToPlayBehaviour extends TickerBehaviour {
  protected void onTick(){
    ...
    players_white = LogicTuple.parse("player(name(S), team('WHITE'))");
    players_black = LogicTuple.parse("player(name(S), team('BLACK'))");
    missing_players = LogicTuple.parse("team(S)");
    final RdAll rdall_white = new RdAll(PlayerAgent.this.tcid, players_white
        );
    final RdAll rdall_black = new RdAll(PlayerAgent.this.tcid, players_black
        );
    final RdAll rdall_missing = new RdAll(PlayerAgent.this.tcid,
        missing_players);
    res_white = PlayerAgent.this.bridge.synchronousInvocation(rdall_white,
        null, this);
    res_black = PlayerAgent.this.bridge.synchronousInvocation(rdall_black,
        null, this);
    res_missing = PlayerAgent.this.bridge.synchronousInvocation(
        rdall_missing, null, this);
    if (res_white != null && res_black != null && res_missing != null) {
      ...
      PlayerAgent.this.addBehaviour(new TryToGetTheBallBehaviour());
      this.stop();
    }
    ...
  }
}
```

Il comportamento `TryToGetTheBallBehaviour` funziona in maniera del tutto

similare al comportamento `ArrivedToPlaygroundBehaviour`: quando viene eseguito, effettua una `inp` asincrona per cercare di ottenere la tupla `ball(1)` e affida il risultato al comportamento `BallResultBehaviour`.
All'interno di `BallResultBehaviour` l'agente attende il risultato della `inp` ed in seguito si passa al comportamento `PlayGameBehaviour` con l'indicazione del fatto che il giocatore abbia ottenuto il possesso del pallone o meno.

```
private class TryToGetTheBall extends OneShotBehaviour {
  ...
  public void action() {
    ...
    LogicTuple ball = LogicTuple.parse("ball(1)");
    final Inp inp = new Inp(PlayerAgent.this.tcid, ball);
    PlayerAgent.this.bridge.asynchronousInvocation(inp, new
        BallResultBehaviour(PlayerAgent.this, 1000), PlayerAgent.this);
    ...
  }
}
public class BallResultBehaviour extends TickerBehaviour implements
    IAsynchCompletionBehaviour{
  ...
  public void onTick(){
    ...
    boolean ball = false;
    if (this.res.resultOperationSucceeded()){
      ball = true;
    }
    PlayerAgent.this.addBehaviour(new PlayGameBehaviour(PlayerAgent.this,
        5000, ball));
    this.stop();
  }
  public void setTucsonOpCompletionEvent(TucsonOpCompletionEvent ev) {
    this.res = ev;
  }
}
```

Il comportamento `PlayGameBehaviour` estende `TickerBehaviour` e viene eseguito dall'agente ogni 5 secondi. Esso realizza tutta la logica delle azioni del giocatore durante la partita. Durante l'inizializzazione del comportamento, viene creato il produttore Kafka necessario per poter pubblicare gli eventi di gioco generati dall'agente e viene aggiunto il comportamento `ObserveGameBehaviour`, che è del tutto similare al comportamento omonimo descritto per l'agente custode.
Quando l'agente detiene il possesso del pallone, durante l'esecuzione del comportamento `PlayGameBehaviour`, effettua la scelta dell'azione da compiere

tramite la selezione casuale di un agente tra tutti i componenti della propria squadra (compreso l'agente stesso che deve effettuare l'azione). Se la scelta ricade su un compagno, l'agente si limita a passargli il pallone inviandogli un messaggio tramite ACC di JADE, altrimenti, se viene estratto l'agente stesso, l'azione risulta essere un tiro a canestro, che in base ad un numero casuale, può essere un tiro realizzato o sbagliato. In seguito il possesso del pallone passa poi all'altra squadra: l'agente seleziona un giocatore avversario e gli consegna il pallone inviandogli un messaggio. Non appena viene selezionato il tipo di azione che l'agente deve compiere, questi effettua una pubblicazione del relativo messaggio all'interno del topic Kafka del playground.
Si noti che nel caso in cui il giocatore non è in possesso del pallone, egli si limita semplicemente a controllare la sua mailbox tramite il metodo `receive()` per verificare se qualche giocatore gli ha passato la palla.

```
private class PlayGameBehaviour extends TickerBehaviour {
  ...
  public PlayGameBehaviour(Agent a, long period, boolean ball) {
    ...
    this.producer = new KafkaProducer<String, String>(producer_props);
    PlayerAgent.this.addBehaviour(new ObserveGameBehaviour(PlayerAgent.this,
        1000));
  }
  private void shoot() {
    ...
    int shot_p = new Random().nextInt(100);
    if (shot_p + mood > 40){
      shot = "shot_missed";
    } else {
      shot = "shot_made";
    }
    Json msg = new Json();
    msg.add("type", shot);
    msg.add("player", PlayerAgent.this.getAID().getName());
    msg.add("team", PlayerAgent.this.team);
    ProducerRecord<String, String> record = new ProducerRecord<String,
        String>(PlayerAgent.this.playgroundTopic, msg.toString());
    this.producer.send(record);
    String opponent = PlayerAgent.this.opponents.get(new Random().nextInt(
        PlayerAgent.this.opponents.size()));
    ACLMessage ball = new ACLMessage(ACLMessage.PROPOSE);
    ball.addReceiver(new AID(opponent, AID.ISGUID));
    ball.setOntology("ball");
    ball.setContent(ball_to_opponent);
    send(ball);
  }
  private void pass(String target_agent) {
```

```
    Json msg = new Json();
    msg.add("type", "pass");
    msg.add("player", PlayerAgent.this.getAID().getName());
    msg.add("to", target_agent);
    ProducerRecord<String, String> record = new ProducerRecord<String,
        String>(PlayerAgent.this.playgroundTopic, msg.toString());
    this.producer.send(record);
    ACLMessage ball = new ACLMessage(ACLMessage.PROPOSE);
    ball.addReceiver(new AID(target_agent, AID.ISGUID));
    ball.setOntology("ball");
    ball.setContent(pass_quality);
    send(ball);
  }
  protected void onTick() {
    if (this.ball) {
      ...
      int index = new Random().nextInt(PlayerAgent.this.teammates.size());
      String target_agent = PlayerAgent.this.teammates.get(index);
      if (target_agent.equals(PlayerAgent.this.getAID().getName())){
        this.shoot();
      } else {
        this.pass(target_agent);
      }
      this.ball = false;
    } else {
      final ACLMessage msg = this.myAgent.receive(MessageTemplate.
          MatchOntology("ball"));
      if (msg != null) {
        ...
        this.ball = true;
      }
    }
  }
}
```

L'esecuzione concorrente dei comportamenti `PlayGameBehaviour` e `ObserveGameBehaviour` rappresenta il fulcro dell'integrazione tra Kafka e JADE. Il fatto che `ObserveGameBehaviour` non sia bloccante, grazie al `KafkaConsumerAssistant` (o eventualemente alla chiamata diretta del metodo `poll()` con timeout nullo), permette all'agente di sfruttare message passing e publish/subscribe senza che questi interferiscano tra di loro. Si noti inoltre che, utilizzando il metodo `consume(`*`timeout`*`)` di `KafkaConsumerAssistant` con un *`timeout`* molto elevato, è possibile realizzare la gestione dinamica della concorrenza tra i due comportamenti. Facendo in modo che `ObserveGameBehaviour` rimanga in attesa nella coda dei comportamenti sospesi fino a che la sua esecuzione non diventi significativa, ovvero quando ci sono dei messaggi

Kafka pronti per essere ricevuti.
È importante anche sottolineare che tali comportamenti permettono di modellare una situazione in cui coesiste nell'agente una condotta sia proattiva che reattiva, mantendo intatta la sua autonomia. L'agente è infatti capace di eseguire azioni proattive quando ha il pallone "tra le mani" e agire in maniera reattiva nella gestione dei messaggi (sia quelli JADE per il possesso del pallone che quelli Kafka per gli eventi di gioco), senza che sia uno dei due atteggiamenti a comandare il corso delle sue azioni. Si noti che tale situazione non sarebbe facilmente modellabile tramite ELDA, a causa del problema dell'inversione del controllo che non garantisce alcuna separazione tra il volere dell'agente e il flusso degli eventi.

Il codice completo del caso di studio e dell'esempio mostrato in calce al capitolo 2 è disponibile nel repository `https://github.com/marcozaccheroni2/EbsMas2016_Kafka_TuCSoN_JADE.git`

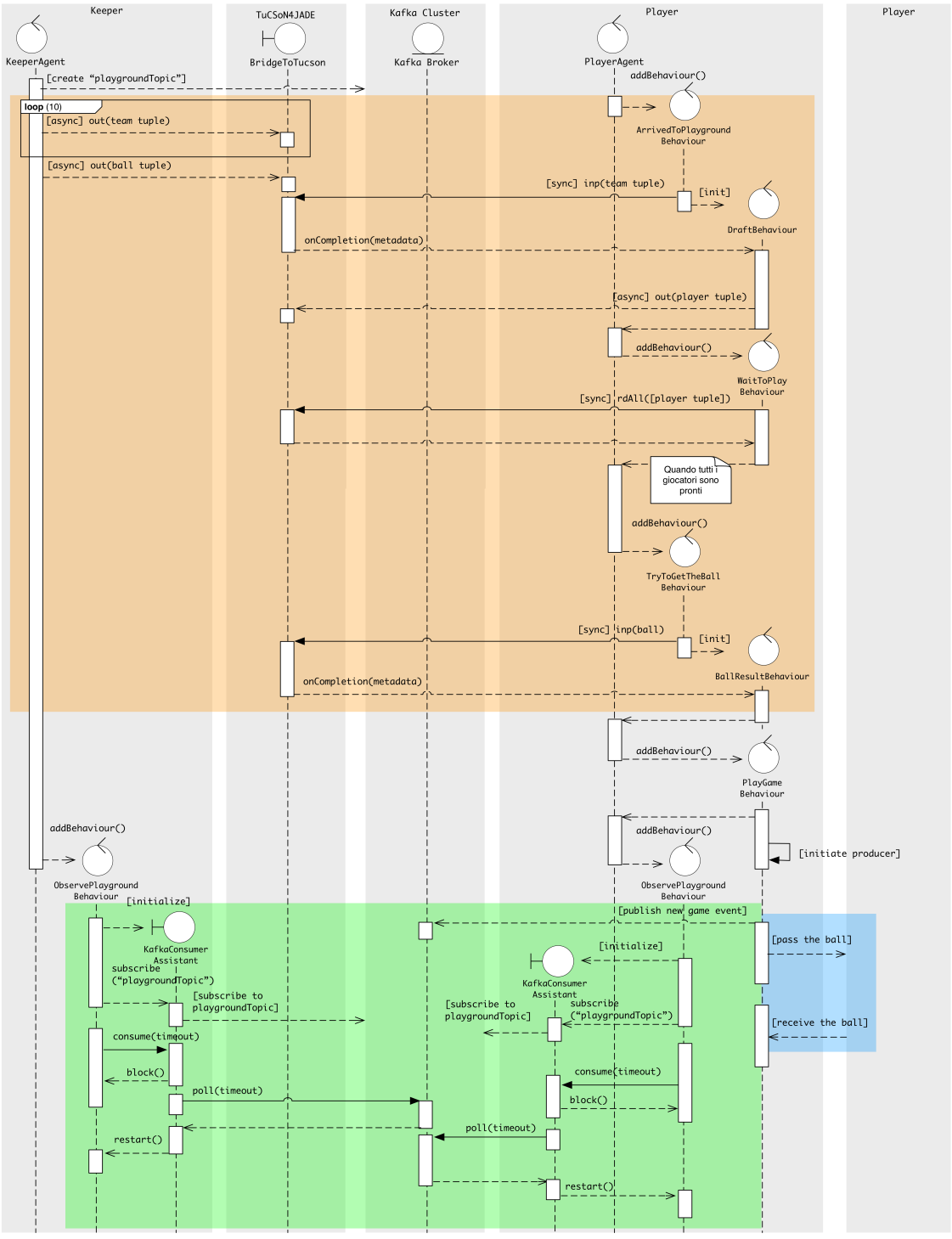

Figura 3.1: Diagramma di interazione degli agenti all’interno del Playground in cui sono evidenziate le interazioni tramite TuCSon (in arancione), JADE (in azzurro) e Kafka (in verde)

Conclusioni e sviluppi futuri

Giunti al termine è opportuno analizzare il lavoro svolto, cercando di individuare i risultati raggiunti e quelli mancati: in modo da offrire interessanti spunti per delle future argomentazioni che integrino ulteriormente il presente trattato confermandone o confutandone il contenuto.

Innanzitutto è bene ricordare l'obiettivo della tesi, al fine di poter valutare correttamente se esso sia stato raggiunto o meno: come è stato già detto, il proposito principale era quello di mostrare come i due stili architetturali apparentemente molto diversi, che caratterizzano i Sistemi Multi-Agente e i Sistemi ad Eventi, presentino in realtà numerosi punti in comune che rendono possibile la loro integrazione al fine di realizzare un'unica architettura capace di sfruttare al meglio le potenzialità di entrambe le tipologie di sistemi.

Si confida nel raggiungimento di tali obiettivi, ad opera dei seguenti capitoli:

- il capitolo 2 dovrebbe aver descritto in maniera esaustiva quali siano i concetti comuni ad entrambi gli stili architetturali e come questi possano essere integrati rispettando i principi fondamentali di entrambe le tipologie di sistemi;
- il capitolo 3 dovrebbe aver mostrato un utile scenario in cui è sfruttata l'integrazione di JADE, TuCSoN e Kafka, mettendo in evidenza come sia possibile integrare questi framework per utilizzare nel miglior modo possibile i loro diversi modelli di coordinazione all'interno di un unico sistema.

Analizzando il risultato ultimo della tesi, ovvero l'integrazione di JADE, TuCSoN e Kafka, è opportuno effettuare alcune osservazioni.

La progettazione e l'implementazione del componente che garantisce il disaccoppiamento di agente e coordination media, nell'integrazione tra JADE e Kafka, ha permesso di dimostrare come sia fondamentale che vengano gestite adeguatamente le dipendenze all'interno di un sistema molto complesso, garantendo l'autonomia dei suoi componenti, in modo che l'interazione possa essere utilizzata in maniera efficace per gestire la complessità del sistema.

A tale proposito, un possibile sviluppo futuro potrebbe essere l'estensione del componente `KafkaConsumerAssistant` in modo da poterlo inserire all'interno dell'infrastruttura JADE così da gettare le basi, includendo anche TuCSoN4JADE, per la realizzazione di un unico framework di sviluppo per i Sistemi Multi-Agente basati su eventi.

Bibliografia

[1] Danny Weyns, Andrea Omicini, and James J. Odell. Environment as a first class abstraction in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, February 2007. Special Issue on Environments for Multi-agent Systems.

[2] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, December 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

[3] Fabio Bellifemine, A Poggi, and Giovanni Rimassa. *JADE - A FIPA-compliant agent framework*, pages 97–108. The Practical Application Company Ltd., 1999.

[4] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special Issue: Coordination Mechanisms for Web Agents.

[5] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[6] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[7] LUDGER FIEGE, GERO MÜHL, and FELIX C. GÄRTNER. Modular event-based systems. *The Knowledge Engineering Review*, 17:359–388, 12 2002.

[8] Giancarlo Fortino, Alfredo Garro, Samuele Mascillaro, and Wilma Russo. Modeling multi-agent systems through event-driven lightweight dsc-based agents. In *Proceedings of 6th International Workshop on "From Agent Theory to Agent Implementation"*, 2008.

[9] Fortino G, Frattolillo F, Russo W, and ZIMEO E. Mobile active objects for highly dynamic distributed computing. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, pages –, April 15-19 - 2002.

[10] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[11] Foundation for Intelligent Physical Agents (FIPA). Agent management support for mobility specification, 2002.

Ringraziamenti

Era il lontano 2006, quando per la prima volta misi piede all'università, in una cupa e uggiosa giornata di fine settembre (in realtà non ricordo assolutamente come fosse il tempo, non sono nemmeno sicuro che fosse settembre, però nei miei ricordi era freddo, umido e anche tutto in bianco e nero). Ora dopo la bellezza di 10 anni (mi vergogno un po' a dirlo, ma tant'è) è giunto il momento che il mio percorso ad Ingegneria arrivi finalmente al termine. Con tutta probabilità non sarei mai riuscito ad arrivare qui, oggi, se non avessi avuto la fortuna di avere un sacco di persone accanto, che mi hanno spronato, supportato e sopportato per tutto questo tempo. Ringraziare tutti credo sia praticamente impossibile, ma cercherò di fare del mio meglio.

Grazie alla Cate, per essere capace di darmi sempre forza e serenità, in qualunque occasione, anche semplicemente standomi accanto.

Grazie a mio babbo e a mia mamma, per non avermi mai fatto mancare niente ed aver fatto sempre il massimo per aiutarmi a superare tutte le difficoltà in modo che potessi arrivare fino a qui.

Grazie ai mio nonno, per "aver fatto i bambini insieme" e per avermi insegnato come affrontare la vita con serenità, accettando quello che capita con un sorriso, cercando sempre di trovare il lato positivo. Vorrei tanto che fosse qui per poter condividere anche con lui questo momento.

Grazie a mia nonna per essere sempre così forte e per essersi ogni volta fatta in quattro per supportarmi in tutti i modi possibili e non farmi mai sentire solo.

Grazie a Ste, per l'inesauribile pazienza, la disponibilità e per tutto l'aiuto che mi ha dato per scrivere questa tesi, senza il quale non sarei mai riuscito a completarla.

Grazie a Piero, Busca e Richard per tutti i momenti passati assieme in questi anni di magistrale tra robot, parcheggi, pianeti e fantasmi.
Grazie a Piero anche per Whale TRUE, TV Files e tutti i progetti fighissimi, mai realizzati, per cui adesso (forse) possiamo trovare il tempo.

Grazie ad Ale, Cianca, Gaia, Giova, Harry, Mike e Pira. Prima di conoscerli ero quello che si può definire *un branco con un lupo solo*, ora invece siamo in tanti nel branco.

Grazie a tutto il Wuber: Sacco, Zatto, Mela, Dulo, Riccio, Manina, Masi, Edo, Greggione, Caffi, Bando, Johnny, Ross e Capitan Cangini, per tutte le serate passate in palestra, perché certe volte giocare assieme a voi era l'unico modo per riuscire a distrarmi.

Grazie al Prof. Omicini, per la Sua inesauribile disponibilità e per la Sua simpatia.

Grazie alla mia Birbina, per tutte le bellissime chiacchierate che facciamo e per avermi insegnato che l'affetto incondizionato è una cosa meravigliosa, anche se è un animalino a dartelo.

Grazie infine a Wikipedia e StackOverflow per l'inestimabile aiuto che mi hanno offerto in questi anni di università, se sono riuscito a laurearmi è anche merito loro.