Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Definizione di algoritmi per la gestione di eventi provenienti da Smart Object in ambito Cloud

Andrea Giordano,
Giandomenico Spezzano,
Andrea Vinci

RT-ICAR-CS-14-03 **Ottobre 2014**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
– Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.icar.cnr.it*

Sommario

Questo documento descrive un approccio alla parallelizzazione dell'esecuzione di query CEP (Complex Event Processing) nel contesto dei cyber-physical system, utilizzando la proprietà di elasticità che è possibile ottenere con le architetture cloud. La piattaforma Rainbow è stata utilizzata per implementare l'approccio, poiché offre alcuni vantaggi che permettono di migliorare ulteriormente le performance dell'esecuzione parallela.

Indice

Elenco delle figure

Elenco dei codici

1 Introduzione

Recentemente, i progressi nel campo delle comunicazioni elettriche e dei sistemi embedded hanno dato il via ad una nuova rivoluzione tecnologica nota sotto il nome di Cyber-Physical. Tale nuovo scenario applicativo investe sia il mondo industriale e manifatturiero che quello della ricerca. La complessità e l'eterogeneità dei sistemi cyber-physical offrono nuove ed interessanti sfide al mondo scientifico. L'eterogeneità può essere affrontata in diversi modi, ad esempio, rifacendosi a standard più o meno noti, o cercando di identificare un comportamento comune a tutte le entità coinvolte nel cyber-physical approntando middleware che tengano conto di tali specificità. Per quel che riguarda la complessità, essa può avere diverse accezioni. Difatti, è possibile interpretare il termine complessità, in questo ambito, nel senso di difficoltà nello gestire compiutamente scenari applicativi piuttosto intricati che richiedono grande esperienza e lunghi tempi di sviluppo e test. La ricerca, da questo punto di vista, si muove verso la definizione di formalismi appropriati che possano semplificare sempre più il processo di sviluppo e manutenzione nel contesto cyber-physical. In questo ambito possono essere usati i formalismi pensati per il Complex Event Processing (CEP). Ve ne sono svariati, in questo documento si farà riferimento esplicitamente a quei formalismi CEP che prevedono codice scritto similmente alle query nei database. Un altra accezione del termine complessità può riguardare la mole di dati coinvolti. Da questo punto di vista, oggigiorno, vi è un grande sforzo di ricerca nel campo dei big data e nella parallelizzazione dell'esecuzione. Negli ultimissimi anni vi è stata una vera e propria esplosione nelle dimensioni dei data center che ha dato il via alle architetture di cloud-computing. Queste ultime possono essere la risposta anche alle problematiche legate alla alta dimensionalità dei dati nel CEP.
Questo documento descrive un approccio alla parallelizzazione dell'esecuzione di query CEP nel contesto cyber-physical utilizzando la proprietà di elasticità che è possibile ottenere con le architetture cloud. La piattaforma Rainbow è stata utilizzata per implementare l'approccio anche perché offre alcuni vantaggi che permettono di migliorare ulteriormente le performance dell'esecuzione parallela.

2 Complex Event Processing

Alcune delle principali innovazioni tecnologiche degli ultimi cinquant'anni dipendono da varie forme di event processing. Tra queste ci sono senza dubbio la simulazione ad eventi discreti, le reti di computer, i database e le tecnologie di middleware. Il concetto di event processing nasce con la simulazione ad eventi discreti negli anni Cinquanta. L'idea principale era che il comportamento di un sistema, sia esso un hardware, un sistema di controllo, una catena di produzione

di una fabbrica o un fenomeno naturale come il meteo, potesse essere simulato tramite un programma per computer. Presi dei dati in input, il programma avrebbe dovuto generare eventi che riproducevano le interazioni tra i componenti del sistema. Ogni evento doveva registrarsi ad un orario fissato da un orologio, e, ovviamente, alcuni eventi potevano verificarsi nello stesso istante. Il simulatore, quindi, doveva registrare il flusso di eventi tra i componenti, l'esecuzione dei componenti, e lo scorrere del tempo dell'orologio. Le simulazioni ad eventi discreti di quel tempo stavano certamente svolgendo event processing e i modelli, infatti, erano architetture event-driven. Un altro tipo di event processing fu coinvolto nello sviluppo delle reti di computer, a partire dagli ultimi anni Sessanta con la rete ARPANET. L'obiettivo era realizzare una comunicazione affidabile tra computer attraverso reti, tramite l'uso di eventi contenenti sequenze di dati binari, i cosiddetti pacchetti. Trasmettere o ricevere un pacchetto corrispondeva ad un evento. La parte più difficile fu senza dubbio lo sviluppo dei protocolli per poter trasmettere una sequenza di pacchetti in maniera affidabile, anche se la rete stessa era totalmente insicura e soggetta ad errori. Dagli sforzi fatti per mettere in piedi tale rete si deve la definizione di standard per i protocolli di rete che utilizzano il concetto di livello di eventi. Due esempi sono il protocollo TCP/IP e il modello ISO/OSI a sette livelli. Lo sviluppo della tecnologia inerente ai cosiddetti active database ha inizio nei tardi anni Ottanta, come un miglioramento dei database tradizionali per poter venire incontro alla sempre più crescente domanda di elaborazione a real time. Essi furono estesi con capacità di event processing che permettevano loro di invocare applicazioni in risposta ad eventi. Per fare ciò, un livello di event processing fu implementato al di sopra di ogni database tradizionale: questo permetteva la definizione di eventi e di semplici tipi di schemi di eventi che potevano essere utilizzati come trigger per le regole reattive, anche chiamate regole ECA (Event - Condition - Action).

2.1 Concetto di evento

Prima di procedere con una vera e propria definizione di che cosa sia esattamente il Complex Event Processing, bisogna chiarire cosa si intende esattamente con il termine 'evento'. Un evento è un'occorrenza all'interno di un particolare sistema o dominio: esso è qualcosa che è avvenuto, o che si prevede che possa avvenire in quel dominio. La parola 'evento', inoltre, è usata per rappresentare, in un sistema computazionale, l'occorrenza di una specifica entità di programmazione, come, ad esempio, la pressione di un pulsante all'interno di un'applicazione può portare allo scatenarsi dell'evento 'mouse clicked'. Esistono eventi che si potrebbero definire come 'rumore di fondo' e che, quindi, non richiedono alcuna reazione. Gli eventi che, invece, ne richiedono vengono spesso chiamati 'situazioni'. Si definisce una situazione come l'occorrenza di un evento che potrebbe

richiedere una reazione. Una delle tematiche principali legate all'event processing è l'identificazione e il riscontro di situazioni, così che possano essere garantire le eventuali reazioni. Una reazione potrebbe essere, semplicemente, rispondere al telefono o aggiungere un oggetto alla lista della spesa, oppure potrebbe consistere in qualcosa di più complesso: nel caso in cui una persona perda l'aereo esistono svariate alternative di reazione a seconda dell'ora e del giorno, dell'aeroporto in cui si trova, delle regole della compagnia e del numero di altri passeggeri nelle sue stesse condizioni.

2.2 Event processing

L'event processing è un processo di computazione che compie operazioni su eventi. Le più comuni sono l'analisi, la creazione, la trasformazione e la cancellazione. Esistono due tematiche principali all'interno di quest'area:

- Il design, la struttura del codice e il funzionamento delle applicazioni che utilizzano gli eventi, sia direttamente che indirettamente. Ci si riferisce a ciò come programmazione event-based, sebbene più spesso viene utilizzata la dicitura architettura event-driven.

- Le operazioni di elaborazione che è possibile eseguire su eventi come parti di una certa applicazione. Questo include il filtraggio di certi tipi di eventi, il cambiare l'istanza di un evento da un tipo ad un altro ed esaminare un insieme di eventi al fine di riscontrare un modello specifico. Queste operazioni spesso fanno sì che nuove istanze di eventi vengano generate.

E' possibile, ovviamente, scrivere programmi event-based senza ricorrere esplicitamente ad operazioni di event processing. Le tre principali differenze che distinguono l'event processing dalla semplice programmazione event-based, e che aprono ad un ricco insieme di possibilità, sono:

- Astrazione - Le operazioni che compongono la logica dell'event processing possono essere separate dalla logica applicativa, permettendone così la modifica senza toccare le applicazioni che producono e consumano gli eventi.

- Disaccoppiamento - Gli eventi riscontrati e prodotti da una specifica applicazione possono essere utilizzati e consumati da applicazioni totalmente differenti. Non c'è alcun bisogno per le applicazioni che producono e quelle che consumano eventi di essere a conoscenza ognuna dell'esistenza dell'altra; non a caso esse possono essere dislocate in qualunque parte del mondo.

Un evento emesso da una singola applicazione produttrice può essere utilizzato da più macchine consumatrici di eventi e, viceversa, si può far sì che una applicazione consumi eventi prodotti da più applicazioni generatrici di eventi.

- Obiettivo incentrato sul mondo reale - L'event processing ha spesso a che fare con eventi che si verificano, o che dovrebbero verificarsi, nel mondo reale.

2.3 Principi fondamentali del Complex Event Processing

Rispetto all'interazione request-response, gli eventi differiscono nella seguente maniera: un evento è l'indicazione di qualcosa che è già accaduto, mentre una richiesta, come suggerisce il nome, esprime la volontà del richiedente che qualcosa di specifico si verifichi nel futuro. Come risultato si usa una terminologia leggermente diversa per i partecipanti ad una interazione che coinvolge eventi. Invece di parlare di richiedente di servizi e di service provider, si parlerà di event producer, o generatore di eventi, e event consumer, o consumatore di eventi. Guardando a due semplici esempi, se si consulta un servizio per la prenotazione online di un volo aereo per cercare quali voli sono disponibili in data specificata, si sta inviando al servizio una richiesta, ma quando l'aereo effettivamente decolla, allora si parla di evento. In alcuni casi, è possibile far sì che un evento mostri le occorrenze sottostanti in maniera completa. Si consideri, ad esempio, un sensore di temperatura che invia un evento ogni minuto. L'evento potrebbe contenere tutto ciò che c'è da sapere: la posizione del sensore, l'ora del giorno e il valore della temperatura. Tuttavia, in altri casi, esistono molte variabili di una occorrenza che si vorrebbero osservare, ma che l'evento in questione non è in grado di mostrare. Si pensi ad un sistema di monitoraggio di un paziente in un ospedale che deve essere in grado di effettuare molteplici misurazioni sulle condizioni del paziente, ma che, quando viene interrogato, riporta solo le misurazioni che sono rilevanti per il trattamento del paziente. Un'altra possibilità è che una singola occorrenza possa essere rappresentata da più di una istanza di evento. Ciò si verifica quando più applicazioni sono interessate a diversi aspetti della medesima occorrenza. Si consideri, a tale scopo, l'occorrenza di un evento come l'assunzione di un nuovo impiegato. L'applicazione che gestisce il libro paga dovrebbe essere interessata a cose come il nome dell'impiegato, il suo identificativo, il livello gerarchico all'interno dell'impresa e il salario di partenza, mentre l'applicazione che si occupa delle polizze assicurative è più interessata alla sua anamnesi familiare e al suo stato di salute corrente. Gli eventi vengono talvolta utilizzati per indicare cambiamenti di stato, spesso all'interno di sistemi di monitoraggio. Un sistema monitorato viene rappresentato come un insieme

di risorse, che possono essere fisiche, come sensori o altri tipi di apparecchiature di rilevazione dati, o logiche, quali processi di business, e ad ognuno dei quali è associata una informazione di stato. Solitamente esiste un modo per interrogare ogni risorsa direttamente, così da poter leggere il suo stato, ma è anche possibile che le risorse agiscano da generatori di eventi e ne inviino uno ogniqualvolta uno o più dei loro stati interni subisce un cambiamento. Questo permette alle applicazioni di monitoring di essere immediatamente avvisate se succede qualcosa, senza così dover effettuare costantemente polling sulla risorsa desiderata. Un evento è per definizione totalmente indipendente dal suo generatore e dal suo consumatore, e questo permette che gli event producer e gli event consumer siano totalmente disaccoppiati gli uni dagli altri. L'idea di utilizzare l'evento stesso come mezzo per disaccoppiare l'event producer e l'event consumer è una sostanziale differenza tra la programmazione event-based e il design applicativo basato su interazioni di request-response. Esiste chiaramente un certo grado di disaccoppiamento anche nelle interazioni request-response; tuttavia, il richiedente del servizio dipende sempre dal service provider che garantisce la funzionalità concordata. Nell'event processing, invece, si può avere mutuo disaccoppiamento del produttore e del consumatore.

2.4 Architettura di event processing

Non tutte le applicazioni di event processing sono uguali tra di loro, ovviamente, ma la maggior parte di queste hanno una struttura riconducibile a quella in Figura 1.

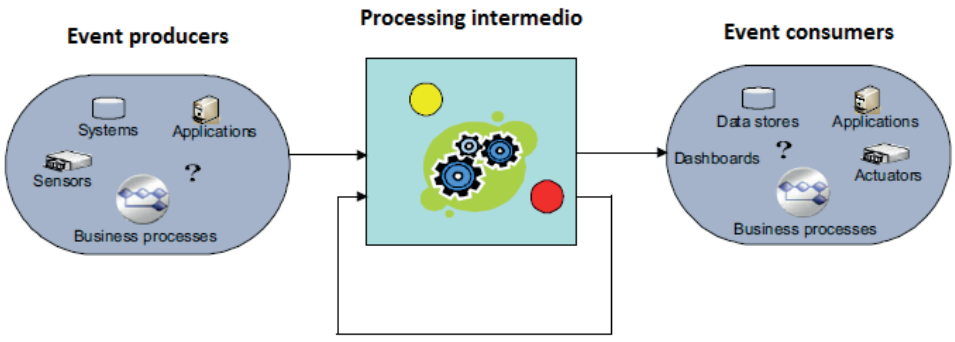

Figura 1: Architettura di un event processing

Un'applicazione può contenere uno o più generatori di eventi, anche chiamati, in linea con quanto detto fino ad ora, event producer. Gli event producer possono apparire in una vasta gamma di forme e dimensioni: per esempio, sensori hardware che producono eventi quando rilevano un certo tipo di occorrenza,

ma anche semplici bit di strumentazione software che produce eventi quando alcune condizioni di errore vengono segnalate o, ancora, bit di controllo di una applicazione.
La controparte degli event producer sono gli event consumer. Questi componenti sono coloro che ricevono gli eventi e, tipicamente, agiscono su di essi. Le loro funzionalità possono variare: essi sono in grado, ad esempio, di immagazzinare gli eventi per un uso futuro, mostrarli in una interfaccia utente, o decidere eventuali comportamenti reattivi dopo averli ricevuti. Gli event producer e gli event consumer sono collegati a meccanismi di distribuzione degli eventi, identificati dalle frecce in Figura 1 e, solitamente, qualche meccanismo di event processing intermedio viene posto tra i suddetti. La distribuzione è garantita mediante l'uso di una tecnologia di instradamento dei messaggi asincrona; per questo motivo, spesso con un leggero abuso di notazione, si parla di un producer che invia un evento o di un consumer che riceve un evento. Ciò nonostante, si possono utilizzare altri meccanismi, come far sì che l'event producer scriva semplicemente le varie occorrenze degli eventi su un file di log, il quale viene poi letto successivamente dagli event consumer. Il meccanismo di distribuzione degli eventi è solitamente one-to-many per far sì che ogni evento, dopo essere stato inviato, possa essere ricevuto da più event consumer. Per quel che riguarda i sistemi di event processing intermedi, nei casi più semplici il processing può semplicemente essere ricondotto ad una attività di routing con o senza filtraggio. Una delle caratteristiche più importanti nelle architetture event-driven è il fatto che i sistemi intermedi sono in grado di generare eventi addizionali. Tali eventi possono essere distribuiti ai vari event consumer, ma possono anche essere anche soggetti ad ulteriori sistemi di event processing, così come è suggerito dalla freccia di feedback nella Figura 1. Il diagramma mostra il sistema intermedio come un componente monolitico. Nella pratica comune, invece, i sistemi intermedi sono composti da una serie di sotto-componenti. E' importante notare il disaccoppiamento, di cui si è già parlato, tra gli event producer e gli event consumer. Gli eventi sono il centro dell'attenzione di tutto il sistema. L'event producer ha una relazione con ogni evento che produce, invece di una relazione con gli event consumer. Non è importante per esso sapere quanti event consumer ci sono per i suoi eventi, e non gli interessano nemmeno le azioni che questi ultimi potrebbero prendere una volta ricevuti gli eventi. Allo stesso modo gli event consumer reagiscono all'evento in sè, e non all'atto della ricezione di un evento da uno specifico event producer, anche se in alcuni casi tale informazione può rivelarsi fondamentale per la corretta reazione del componente. E' possibile distinguere anche due tipologie di eventi: eventi grezzi e eventi derivati. Si definisce evento grezzo un evento introdotto in un sistema di event processing da un event producer. Tale definizione si basa solamente sulla sorgente di tale evento e non sulla sua struttura: un evento grezzo può o meno essere composto da altri eventi. Un evento deriva-

to, invece, si definisce come un evento generato dal sistema di event processing come risultato di una computazione. Un oggetto evento può essere generato in un sistema di event processing, e perciò esso sarà un evento derivato in quel sistema, dopodiché può essere passato ad un altro sistema di event processing dal quale viene considerato un evento grezzo.

3 CEP su Cloud

Dalla crescita delle dimensioni delle applicazioni di Complex Event Processing scaturisce la convenienza nella parallelizzazione dell'esecuzione in modo da garantire scalabilità e buone performance. Tale scenario è supportato dall'interesse verso i cosiddetti 'big-data' e dall'avanzamento delle tecnologie inerenti larghi data-center utilizzabili tramite diversi middleware in modo standardizzato, il che viene spesso riferito complessivamente come 'cloud computing'.
Vi è quindi forte interesse verso la tematica dell'esecuzione del complex event processing su architettura cloud. Anche nello scenario cyber-physical e degli Smart Object è vera l'assunzione precedente. Inoltre, come dettagliato in sezione 4, l'esecuzione può essere condivisa tra cloud e i nodi sensori aumentando l'efficacia della parallelizzazione in questi contesti.
Parlando di 'crescita delle dimensioni' nelle applicazioni di CEP occorre specificare meglio che vi sono diverse "dimensioni" da tenere in considerazioni e, la scalabilità del sistema va valutata in base a queste ultime. In generale possiamo affermare che un sistema CEP è scalabile se riesce a:

- Gestire un grande numero di query
- Gestire query che necessitano di una grande quantità di memoria
- Gestire query molto complesse
- gestire un grande numero di eventi

Consideriamo ogni "dimensione" separatamente

- *Gestire un grande numero di query*

  Questo è il caso più semplice da parallelizzare. Infatti è sufficiente suddividere le query tra i diversi motori CEP paralleli. L'occorrenza degli eventi può essere notificata a tutti nodi paralleli o, in modo più ottimizzato, si può adottare un semplice meccanismo di publish/subscribe in modo da notificare gli eventi solo ai nodi CEP di interesse.

- *Gestire query che necessitano di una grande quantità di memoria*

  Può avvenire, ad esempio, quando si sceglie di utilizzare una finestra temporale molto estesa per la quale è necessario memorizzare molti eventi. In tali casi meccanismi di cache distribuita possono aiutare allo scopo

- *Gestire query molto complesse & gestire un grande numero di eventi* Questi casi sono quelli più difficili da gestire. Infatti in entrambi si tratta di eseguire una singola query su diversi nodi CEP paralleli. La soluzione può risiedere, infatti, nello "spezzettamento" della query come vedremo nel prossimo paragrafo

Passando dal CEP in generale al nostro contesto del cyber-physical e degli SO, vi è un altro motivo per "spezzettare" una query legato alla 'località' degli eventi. Infatti gli eventi fisici avvengono in punti dello spazio ben definiti e vengono catturati da sensori ben definiti che li trasmettono a ben definiti nodi del sistema. In tal senso può essere opportuno eseguire le porzioni di query legate all'occorrenza di determinati eventi sui nodi del sistema dove questi eventi vengono catturati.

3.1 Algoritmo elastico per CEP

CEP può essere inquadrato nel campo del Data Stream Processing, ovvero, è caratterizzato dal fatto che piuttosto che usare i tradizionali database dove memorizzare i dati per future elaborazioni, si processano flussi di dati on-the-fly. Tale scelta nasce dall'esigenza di dover elaborare una quantità enorme di dati che non può essere fisicamente memorizzata oppure dal voler realizzare applicazioni in cui la velocità di reazione al verificarsi di una data situazione sia un parametro importante e prioritario.
Un flusso di dati può essere visto come una infinita sequenza di tuple definite dallo stesso insieme di campi: $(A_1, A_2...., A_n)$. Ci riferiremo all'attributo A_i della tupla t con $t.A_i$. Ogni tupla è etichettata con un attributo *timestamp* che stabilisce il tempo logico con cui essa è entrata nel sistema. Le query in questo ambito sono definite come 'continue', infatti esse sono eseguite continuamente sul flusso di dati e ogni volta che i predicati di una query sono soddisfatti l'evento è notificato lato 'utente'. In generale possiamo vedere una query come un grafo aciclico dove i nodi sono operatori di algebra relazionale e gli archi definiscono flussi di dati.
Per gli scopi dell'algoritmo che segue, gli operatori possono essere suddivisi in 2 tipi:

- operatori **stateless**: ovvero che non mantengono uno stato ma si applicano semplicemente di volta in volta ad ogni nuova tupla entrante. Es.: Map, Union e Filter.

- operatori **stateful**: ovvero che si applicano su sequenze di tuple. Considerata la natura "infinita" degli stream, le query stateful tipicamente si applicano su finestre scorrevoli definite su un certo periodo temporale o legate al numero di eventi. Es.: predicati "aggregati" (media, varianza etc.), prodotto cartesiano e Join.

3.1.1 Query Parallelization Strategies

Quando si tenta di eseguire una query in parallelo bisogna fare particolare attenzione alla cosiddetta *correttezza semantica*, ovvero, il risultato della query eseguita in parallelo deve essere uguale a quando la stessa query è eseguita in modo sequenziale e centralizzato. La difficoltà nell'esecuzione di query in distribuito sta nella gestione degli operatori stateful. Infatti, per ottenere un risultato coerente, tutte le tuple coinvolte in una operazione di aggregazione/join dovrebbero essere analizzate dallo stesso nodo. Possono essere adottate diverse strategie per la parallelizzazione, esse possono essere classificate in base alla granularità dell'"unità di parallelizzazione" adottata. Tale unità di parallelizzazione può essere l'intera query oppure il singolo operatore o tutte le possibilità tra questi 2 estremi. La strategia di parallelizzazione si valuta in base al numero di "hop" necessari (ovvero il numero di passaggi da un nodo all'altro necessari per l'esecuzione della query) e al "fan-out" (il numero massimo di nodi con cui un certo nodo deve comunicare per garantire la correttezza semantica). Per illustrare le varie strategie di parallelizzazione utilizziamo una query d'esempio (Figura 2) che verrà eseguita su 90 nodi. La query è composta da 2 operatori stateful: Join (J) e una aggregazione (Ag); e quattro stateless: due Map (M) e due operazioni di filtraggio (F).

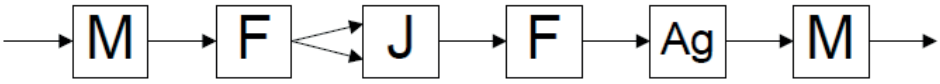

Figura 2: Query d'esempio.

- **Query-cloud strategy** - **QC** (Fig. 3). Quando l'unità di parallelizzazione è l'intera query, questa viene istanziata su ognuno dei 90 nodi a disposizione. La correttezza semantica è garantita dal fatto che le tuple sono 'ridistribuite' tra i nodi immediatamente prima di ogni operatore stateful. In questo modo le tuple che devono essere aggregate o unite tra loro saranno ricevute dallo stesso nodo. Ogni nodo riceverà in ingresso un novantesimo dello stream iniziale, mentre la comunicazione tra i nodi (a

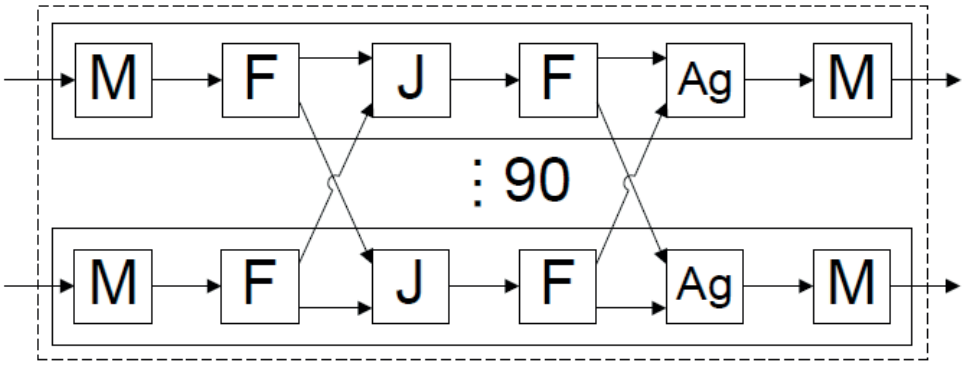

Figura 3: Query parallela con Query Cloud strategy.

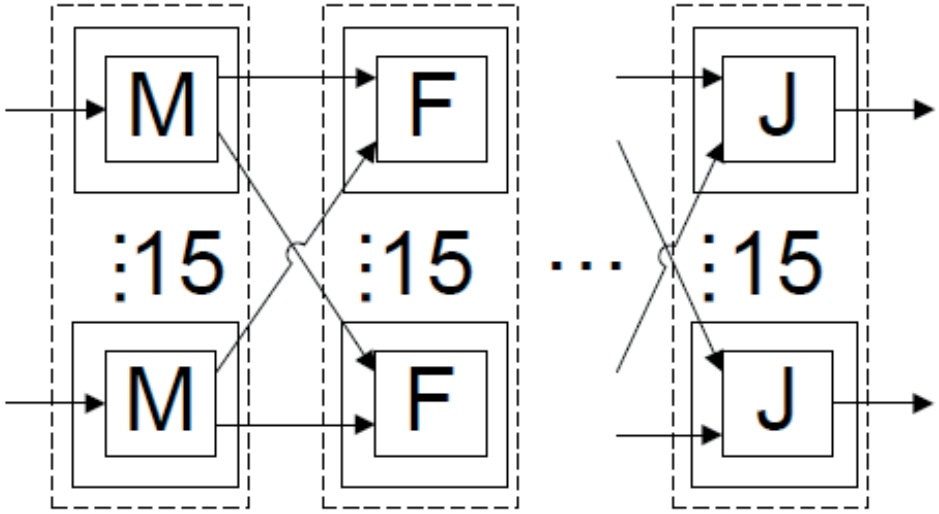

Figura 4: Query parallela con Operator Cloud strategy.

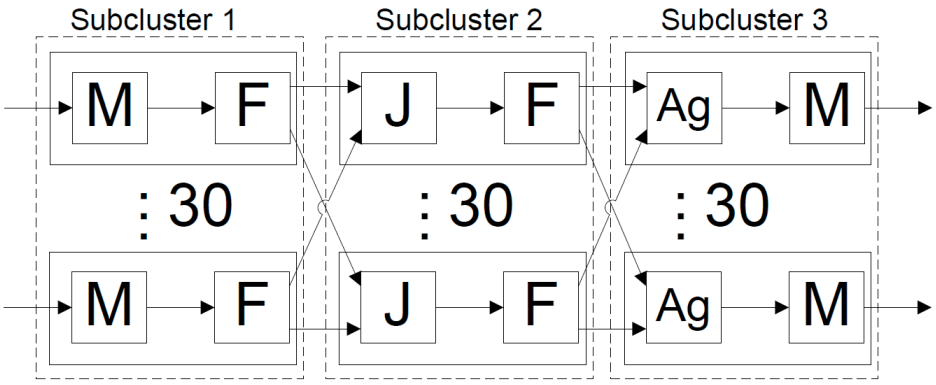

Figura 5: Query parallela con Operator Set Cloud strategy.

causa degli operatori stateful) avverrà tra ogni nodo e tutti gli altri. In definitiva, il numero di hop necessari è uguale al numero di operatori stateful (in questo caso 2) mentre il fan-out, in base a quando evidenziato, sarà 89 (numero di nodi meno uno).

- **Operator-cloud strategy** - **OC** (Fig. 4). In questa strategia l'unità di parallelizzazione è il singolo operatore. Ad ognuno di essi è riservato un sottoinsieme di nodi (subcluster). Nel nostro esempio, i subcluster sono composti da 15 nodi e le comunicazioni avvengono tra ogni subcluster e tutti i nodi del successivo subcluster. Da queste considerazioni si evince che il numero di hop è pari al numero di operatori meno uno (5 nell'esempio), mentre in fan-out è pari al numero di nodi di cui è composto un subcluster (nel nostro caso 15).

- **Operator-set-cloud strategy** - **SC** (Fig. 5).

  questa strategia rappresenta un giusto compromesso tra le precedenti 2 e permette di ottenere buoni risultati per ciò che riguarda numero di hop e fan-out. L'idea sottostante è quella di operare le comunicazioni inter-nodo solo 'prima' di ogni operatore steteful. Ogni query sarà quindi spezzettata in sottoquery per quanti operatori stateful ci sono, più, eventualmente, una sottoquery che contiene tutti i predicati stateless all'inizio della query. Ogni sottoquery è composta da un operatore staeful seguito da tutti gli operatori stateless che precedono l'operatore stateful susseguente (che farà parte della 'successiva' sottoquery). La prima sottoquery conterrà invece tutti gli operatori stateless prima del primo operatore stateful. Le sottoquery così identificate rappresentano l'unità di parallelizzazione della strategia in esame.

  La query d'esempio 2 ha 2 operatori stateful più un "prefisso" stateless, quindi in totale sarà splittata in 3 sottoquery. Ogni sottoquery, come si può vedere in Figura 5 è assegnata ad un *sottocluster* di 30 nodi. Si evince che il numero di hop è pari al numero di operatori stateful (meno uno se non vi è il prefisso stateless), nel nostro caso è quindi 2, mentre il fan-out è pari al numero di nodi dei sottocluster (nel nostro caso 30).

3.1.2 Query parallelization

In questo paragrafo vedremo come parallelizzare una query introducendo due entità utili allo scopo: il **Load Balancer** e l'**Input Merger**. Come query esemplificativa sarà usata quella riportata in Figura 6 formata, come si può vedere, da un operatore di mapping (M), una aggregazione (Ag) e una operazione di filtraggio (F). Dalle considerazioni del paragrafo precedente risulta chiaro che tale

query sarà spezzettata in 2 sottoquery assegnate a 2 sottocluster come visibile in figura 6. Dato un certo sottocluster i nodi facente parte del sottocluster 'precedente' saranno chiamati anche nodi *upstream* mentre i nodi del sottocluster 'successivo' si chiameranno nodi *downstream*.

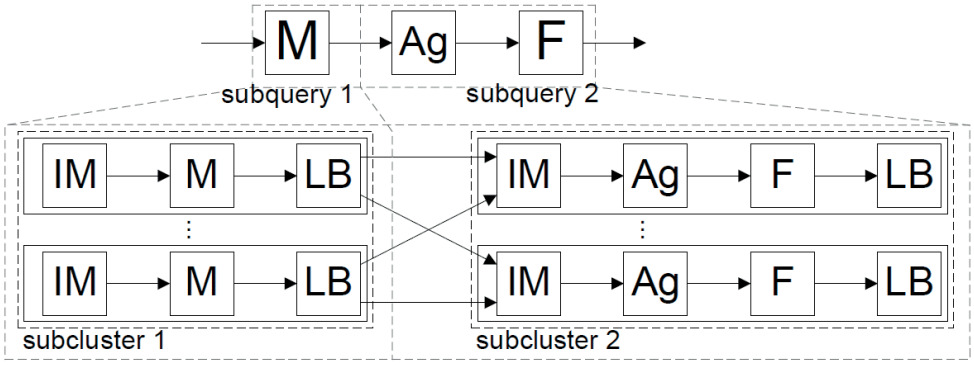

Figura 6: Parallelizzazione di una query.

Per garantire la correttezza nella distribuzione dei dati tra un sottocluster e i successivi, si assegna ogni tupla in uscita da un sottocluster a un *bucket*. l'assegnazione è basata sugli attributi della tupla. In particolare, data una tupla $t = (A_1, A_2, ..., A_n)$, il corrispondente bucket viene calcolato applicando una funzione hash, es: $b = H(A_i, A_j) \bmod B$ dove B è il numero di bucket. Tutte le tuple mappate allo stesso bucket saranno inviate allo stesso nodo downstream. L'associazione tra i bucket e gli N nodi downstream è realizzata dalla funzione BIM (bucket-instance map, *instance* è usato qui con il significato di nodo). Formalmente $BIM[b].dest = A$ significa che il nodo A è responsabile dell'esecuzione delle tuple legate al bucket b. Le operazioni correlate alle relazioni tra tuple e bucket e tra bucket e nodi downstream sono realizzate da speciali operatori: il **Load Balancer** (**LB**) e l'**Input Merger** (**IM**).
Gli LB sono logicamente posizionati su ogni 'arco uscente' dei nodi di un sottocluster e servono a distribuire le tuple uscenti dal nodo tra i nodi del successivo sottocluster. Gli IM, invece, sono posti sugli archi entranti ai nodi di un sottocluster e servono per unire i flussi di tuple provenienti dai LB dei sottocluster precedenti in un unico flusso che sarà dato in pasto al nodo. Di seguito il dettaglio di queste due entità.

3.1.3 Load Balancers

I Load Balancer sono responsabili di distribuire le tuple che risultano da una data sottoquery a tutti i nodi downstream. Per far si che le tuple che devono essere aggregate/unite insieme siano ricevute dallo stesso nodo downstream gli LB devono

essere 'arricchiti' di qualche conoscenza semantica aggiuntiva. In particolare essi devono conoscere la natura dell'operatore stateful con cui parte la sottoquery amministrata dai nodi downstream. Ecco come gli LB si comporteranno nei vari casi:

- **operatore di Join** (equijoin): consideriamo gli operatori di join nei quali vi è almeno una clausola di eguaglianza. Data una tupla t Gli attributi che sono coinvolti nella clausola d'eguaglianza saranno usati per determinare il bucket e il nodo ricevente. Formalmente se A_i è l'attributo di t coinvolto nella clausola d'eguaglianza allora $BIM[H(t.A_i) \bmod B].dest$ sarà il nodo ricevitore. Se la clausola d'eguaglianza utilizza più attributi la funzione hash sarà calcolata su tutti gli attributi. Come conseguenza, tuple che sono uguali sugli attributi coinvolti nell'uguaglianza saranno inviati agli stessi nodi downstream.

- **Prodotto Cartesiano**:

  Il Prodotto Cartesiano differisce dall'operazione di Join in quanto non vi sono clausole d'uguaglianza. Gli LB upstream sono 2: il sinistro e il destro. Essi partizionano i dati in B_l e B_r bucket rispettivamente. Il BIM dell'LB associa un nodo ricevitore per ogni $(b_l, b_l) \in B_l X B_r$. Considerando un predicato che agisce sugli attributi A_i, A_j, ogni tupla t in ingresso all'LB upstream di sinistra sarà inviata a tutti i nodi $BIM[b_l, b_r].dest$ per le quali $b_l = H(A_i, A_j) \bmod B_l$. Similmente, ogni tupla t_0 in ingresso all'LB upstream di destra è inviata a tutti i nodi $BIM[b_l, b_r].dest$ per le quali $b_r = H(A_i, A_j) \bmod B_r$

  Figura 7 mostra una query d'esempio costituita da due mapping (M_l e M_r) e un prodotto cartesiano (CP). M_l divide per 2 gli interi che gli arrivano in ingresso. M_r rende maiuscoli le lettere che gli arrivano come ingresso. L'operatore prodotto cartesiano ha il predicato: $left.numero \bmod 2 = 1 \wedge right.lettera \neq A$ e una finestra temporale di 2 secondi. La parte superiore di Figura 7 mostra anche un possibile input alla query e il corrispondente output. Il valore del *timestamp* (ts) delle tuple è indicato all'alto dello stream.

  La parte di sotto di Figura 7 mostra la versione parallela. Seguendo la strategia Operator-set-cloud, la query è suddivisa in 2 parti: una contiene l'operatore di mapping e l'altra il prodotto cartesiano. La prima parte eseguirà su un sottocluster di 2 nodi e la seconda su uno di 4 (N=4). Gli stream d'ingresso destri e sinistri del prodotto cartesiano hanno valori 0-3 e A-D rispettivamente. Entrambi gli stream sono suddivisi in 2 bucket. Il BIM per l'LB di sinistra mappa le tuple con interi n $0, 3$ ai nodi 0 e 1,

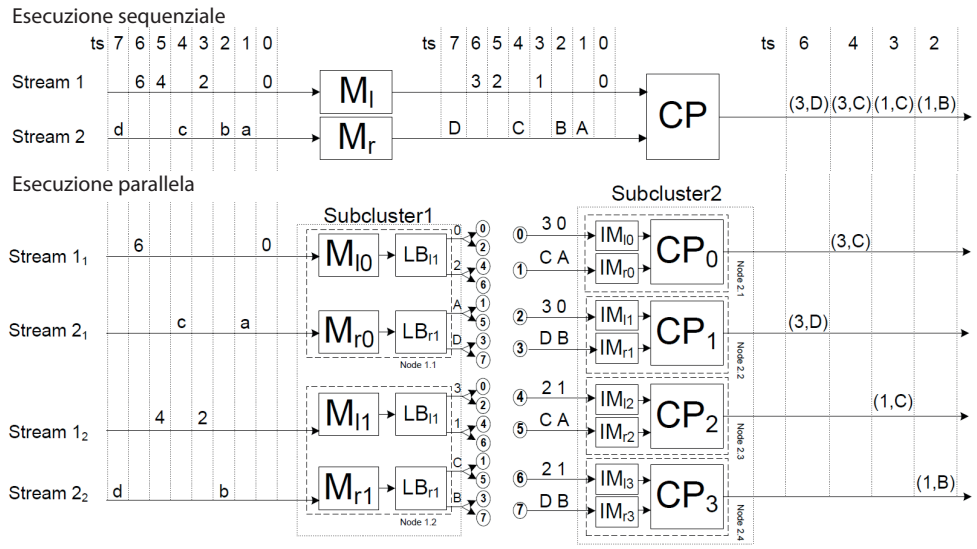

Figura 7: Esempio d'esecuzione di una query in presenza di prodotto cartesiano.

mentre le tuple con interi in $1, 2$ ai nodi 2 e 3. Il BIM per l'LB di destra mappa le tuple con lettere in A, C ai nodi 0 e 2, mentre le tuple con lettere in B, D ai nodi 1 e 3. Di conseguenza ogni nodo ricevitore eseguirà un quarto dell'intero prodotto cartesiano.

- **operatore Aggregazione**:

  Quando si parallelizza una aggregazione è necessario che le tuple che condividono gli stessi attributi per i campi di 'aggregazione' (ovvero quelli specificati nella clausola Group By) siano processati dagli stessi nodi. Per questo motivo è abbastanza semplice capire che il comportamento dell'LB è del tutto analogo al caso di operatore di Join.

3.1.4 Input Mergers

Gli Input Merger (IM), come i Load Balancer, servono a garantire la correttezza semantica. Senza IM potrebbero nascere problemi legati all'ordine d'arrivo delle tuple. Ad esempio ipotizziamo una finestra temporale di 2 eventi e supponiamo che 2 tuple A, B, C abbiano rispettivamente i timestamp 1, 2, 3. Supponiamo inoltre che le tuple A e B siano coinvolte in un prodotto cartesiano. A questo punto occorre ricordare che nella comunicazione tra i nodi l'ordine di trasmissione non è preservato, può quindi succedere che le tre tuple arrivino nel seguente ordine: A, C, B. Il problema è che considerato che C ha timpestamp 3 la tupla A (che ha timestamp 1) sarebbe cancellata per effetto della finestra temporale prima che sia ricevuta la tupla B.

Per ovviare a questa eventualità, gli IM aspettano sempre di ricevere le tuple da entrambi i LB upstream e immettono la tupla con il timestamp più piccolo. In questo modo l'ordine delle tuple basato su timestamp è preservato. Per evitare che il sistema rimanga bloccato in attesa di avere entrambi i valori per ogni upstream, Gli LB manderanno delle tuple 'dummy'.

3.1.5 Elasticità

Il vantaggio di utilizzare il cloud sta nella possibilità di adattare velocemente le risorse alla computazione, ovvero di allocare/deallocare nodi di calcolo dinamicamente in base ai cambiamenti nel carico computazionale. Questa proprietà è spesso chiamata *elasticità*. Le architetture dei moderni data center sono pensate proprio per garantire questa conveniente proprietà, ciononostante, il sistema globale può risultare più o meno 'elastico' sulla base dell'algoritmo utilizzato. Infatti molti problemi sono stati "riscritti" secondo il noto paradigma *map-reduce* proprio per la facilità di esecuzione elastica sulle odierne architetture di cloud. Ovviamente il map-reduce non è la panacea per ogni problematica e quindi la ricerca è molto attiva nel trovare soluzioni per la parallelizzazione ed esecuzione elastica di specifici problemi o categorie di problemi. L'algoritmo descritto in questo documento si inserisce in questa categoria. Dal punto di vista dell'esecuzione cloud esso prevede 3 entità aggiuntive: un Local Manager (LM) per ogni nodo di calcolo, un Resource Manager (RM) e un Elastic Manager (EM) come è possibile vedere in Figura 8.
Gli LM monitorano il carico computazionale su ogni nodo di calcolo e lo riportano all'EM che potrà dinamicamente decidere quando allocare o deallocare nodi di calcolo. Per far questo dialoga con il RM che mantiene il pool di nodi di calcolo disponibili. E' importante notare che l'elasticità è garantita dalla struttura stessa dell'algoritmo, infatti quando si decide di allocare/deallocare nuovi nodi, dal punto di vista dell'algoritmo, è necessario solamente aggiornare le funzioni BIM mentre il resto continua a funzionare esattamente come prima.

4 Integrazione con Rainbow

In questa sezione introdurremmo la piattaforma Rainbow delineandone le principali caratteristiche e possibilità. Tale piattaforma è stata già efficacemente utilizzata in alcuni scenari applicativi collocabili nei nuovi scenari del cyber-physical. Qui viene descritto come integrare/usare la piattaforma Rainbow nell'ambito dell'esecuzione parallela di query CEP. In particolare viene mostrato come l'architettura è stata rivista per poter gestire in modo appropriati flussi di dati. Inoltre è mostrato come incrementare le performance dell'algoritmo di esecuzione ela-

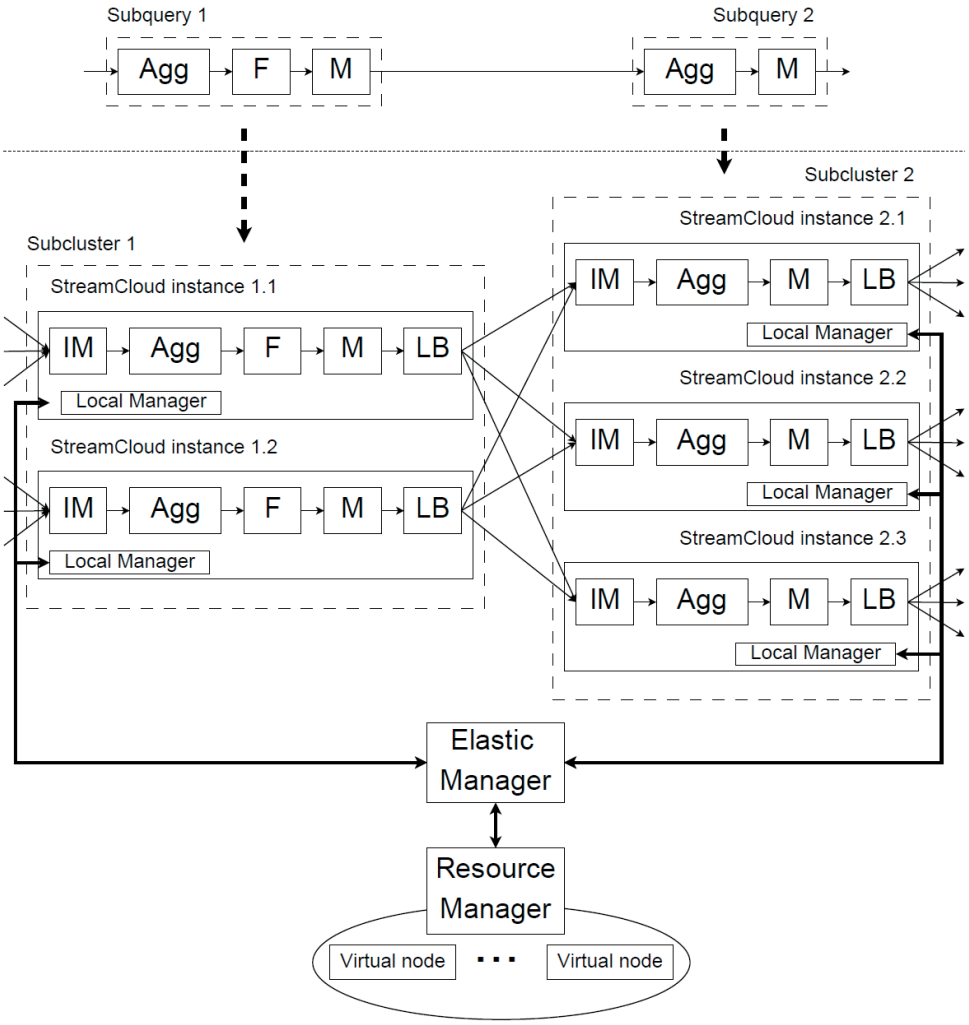

Figura 8: Architettura per l'esecuzione elastica.

stica descritto in 3 eseguendo alcune porzioni delle query direttamente sui nodi sensore.

4.1 Rainbow middleware

Rainbow è una architettura distribuita a tre livelli pensata per portare la computazione (ovvero la parte di controllo) più vicina possibile al livello fisico.
Rainbow include un livello multi-agente distribuito che permette di realizzare applicazioni che godano delle proprietà di *Autonomia*, *Località* e *Decentralizzazione* Sulla base di queste basilari caratteristiche, un sistema multi-agente è in grado di esibire comportamenti anche complessi (detti *emergenti*) attraverso l'interazione tra agenti che invece singolarmente esibiscono comportamenti anche molto semplici. Esempi di comportamenti emergenti possono avere a che fare con le proprietà di *adattatività*, *tolleranza ai guasti*, *riconfigurazione automatica* etc. In generale possiamo parlare di *swarm-intelligence* quando un comportamento "intelligente" emerge dall'interazione tra semplici entità. In letteratura sono noti numerosi algoritmi di swarm intelligence bio-ispirati che potrebbero essere usati efficacemente nell'ambito del CEP.

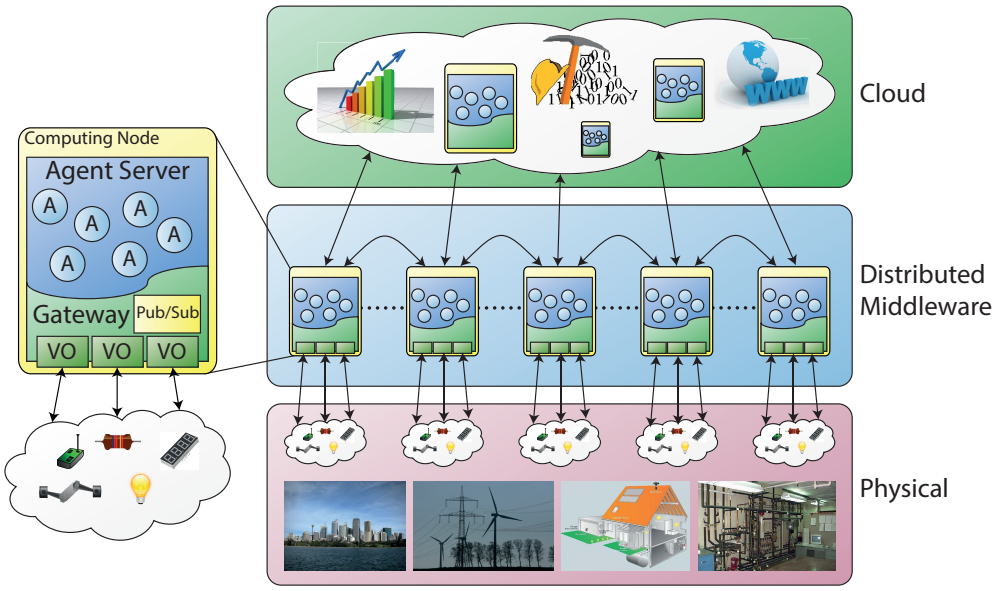

Figura 9: Architettura di Rainbow.

In figura 9 è mostrata l'architettura Rainbow, evidenziandone i diversi livelli.
Il livello più basso rappresenta la parte fisica. E' costituito dai vari sensori e attuatori, oltre che da moduli con capacità computazionali direttamente disponibili nell'ambiente fisico.

Nel livello intermedio, i sensori e gli attuatori sono rappresentati in modo software dai *(Virtual Objects)* (VO). I VO espongono tramite API un'interfaccia ben definita, in modo da offrire agli agenti un accesso trasparente e standardizzato alla parte fisica. Difatti, tramite i VO, gli agenti possono accedere direttamente alla parte fisica senza che questo significhi avere a che fare con driver proprietari o altri dettagli di specifiche tecnologie. Le informazioni disponibili al livello fisico sono tradotte in funzionalità software, che possono essere combinate, anche in modo sofisticato, o usate per definire regole complesse la cui occorrenza influenza l'applicazione utente.
In pratica, come sarà dettagliato ulteriormente nella sezione 4.1.1, tutti i dispositivi fisici sono mappati opportunamente in dei VO che, a loro volta, sono amministrati all'interno di contenitori distribuiti detti *gateway*. Gli eventi e le regole ad essi associati sono gestite dalla componente *publish/subscribe* interna al gateway dettagliata in sezione 4.1.2. I nodi computazionali che contengono i gateway rappresentano il livello intermedio dell'architettura Rainbow. In questi nodi sono presenti anche degli agent server che permettono l'esecuzione corretta degli agenti. Gateway e Agent Server sono entrambi all'interno di ciascun nodo computazionale in modo da consentire agli agenti un accesso diretto alla parte fisica, attraverso l'astrazione fornita dai VO. In pratica, piuttosto che trasferire i dati ad una unità di elaborazione centralizzata, si preferisce trasferire i processi (ovvero gli agenti in esecuzione) verso la sorgente dei dati. In questo modo meno informazioni dovranno essere trasferite su lunghe distanze (ovvero verso host remoti), privilegiando quindi l'accesso locale alle risorse, impattando positivamente su performance e scalabilità dell'intero sistema. Nel caso di esecuzione di query CEP, la query può essere spezzettata come descritto in 3. L'esecuzione della query, così come descritto, può usufruire dell'elasticità offerte dalle architetture cloud.
Difatti, il livello più alto di Rainbow è quello relativo al cloud. Questo livello, oltre ad offrire supporto per l'esecuzione elastica di query CEP, può essere efficacemente utilizzato per tutte quelle attività che non possono essere eseguite compiutamente nel livello intermedio, quali, ad esempio, l'esecuzione di algoritmi che necessitano di conoscenza globale o di capacità computazionali molto onerose o, ancora, che hanno a che fare con dati di tipo storico (data warehouse). Tutte quelle attività che, viceversa, necessitano di accedere alla parte fisica in real-time potranno essere eseguite compiutamente nel livello intermedio. Infatti, come dettagliato in 4.3, alcuni porzioni delle query CEP possono essere eseguite compiutamente nei nodi sensori del livello intermedio col risultato di ottenere performance migliori.

4.1.1 Virtual Object

Il Virtual Object, come già detto, può incapsulare un semplice sensore, che fornisce una grandezza fisica, oppure una struttura molto più complessa (es. nel caso di smart room, smart building etc.). In generale, un Virtual Object può fornire diverse informazioni, siano esse *puntuali* (es. la temperatura in un dato punto nello spazio) che *aggregate* (es.l'umidità media nell'arco delle ultime 24 ore). Le informazioni del Virtual Object possono scaturire da semplici misurazioni fisiche (es. il valore di un certo sensore) così come possono essere frutto di elaborazioni complesse o che coinvolgano più entità insieme (es. la temperatura in un dato punto geografico stimata interpolando le misurazioni fornite da sensori disseminati nel territorio).

Il Virtual Object, inoltre, può avere funzionalità di *attuazione*, modificando lo *smart environment* che lo circonda in base a stimoli esterni o propri reasoning.

I Virtual Object possono anche essere organizzati in modo gerarchico: una *applicazione* che utilizza diversi Virtual Object (anche non relativi allo stesso nodo di calcolo) può implementare a sua volta la Virtual Object Interface, divenendo anch'esso un Virtual Object. Ciò può essere utile per fornire risorse aggregate e/o frutto di reasoning.

Tale generalità di comportamenti deve riflettersi nel modo in cui si interagisce con i Virtual Object. Per tali motivazioni, si è pensato di considerare il Virtual Object come un oggetto complesso che può fornire, o attuare su, diverse risorse fisiche elementari.

In tale contesto, si è previsto di utilizzare la metafora dei *servizi*, qui intesi più propriamente come *funzionalità*. Ogni Virtual Object espone dei servizi (es. una Virtual Room può offrire la temperatura di una stanza, la sua umidità, il numero medio di persone che vi entra ogni giorno etc.).

Per generalità, si pensa che ogni servizio/funzionalità, sia esso di *sensing* o *acting*, possa essere configurabile, nel senso che è possibile passare dei parametri per specificare dinamicamente il suo comportamento.

Da queste considerazioni si prevede che la risorsa elementare utilizzabile sia identificata dalla seguente tripletta:

$$[VOId, VOServiceId, VOServiceParams]$$

dove *VOId* identifica il Virtual Object, *VOServiceId* identifica il servizio/funzionalità e *VOServiceParams* è, in generale, una lista di parametri per la configurazione del servizio scelto.

Esempio. Si supponga di disporre di un Virtual Room che incapsula delle grandezze fisiche all'interno di una stanza. Tale Virtual Object è realizzato con diversi sensori disseminati nella stanza. Se dall'esterno vogliamo leggere la temperatura in un preciso punto della stanza, la tripletta potrebbe essere del tipo: $[VirtualRoom, temperature, [x, y, z]]$.

4.1.2 Publish/subscribe di eventi

La componente di *publish/subscribe* prevede la pubblicazione e la sottoscrizione di eventi tramite delle regole che ne definiscono l'occorrenza. Ogni regola può interessare più servizi dello stesso Virtual Object o di differenti Virtual Object. Una regola è definita come una *proposizione logica* i cui *atomi* sono del tipo:

$risorsa\ fisica < soglia$ (es. $temperatura < 300$),

$risorsa\ fisica > soglia$,

$risorsa\ booleana$ (es. $porta\_aperta$).

Un esempio di regola:

$$(temperatura < 100 \vee luminosità > 500) \wedge num\_persone > 3 \wedge porta\_aperta$$

La regola, comunque complessa, verrà parserizzata in modo da generare un albero binario costruito nel seguente modo: ad ogni nodo dell'albero è associata una proposizione logica; i figli f_L e f_R di un nodo n sono associati a proposizioni logiche $F_{Ln}()$ e $F_{Rn}()$, in modo tale che la proposizione $N()$, associata al nodo n, sia pari a $N() = F_{Ln}() \wedge F_{Rn}()$ oppure $N() = F_{Ln}() \vee F_{Rn}()$. La radice dell'albero è associata alla regola totale, mentre le foglie sono associate alle proposizioni atomiche passate ai Virtual Object. Essi sono incaricati di stabilire, istante per istante, se una data proposizione è soddisfatta o meno (Figura 10).

Le proposizioni dei nodi vengono valutate in base al valore delle proposizioni dei figli. Quando la proposizione associata alla radice è soddisfatta viene notificata ai sottoscrittori.

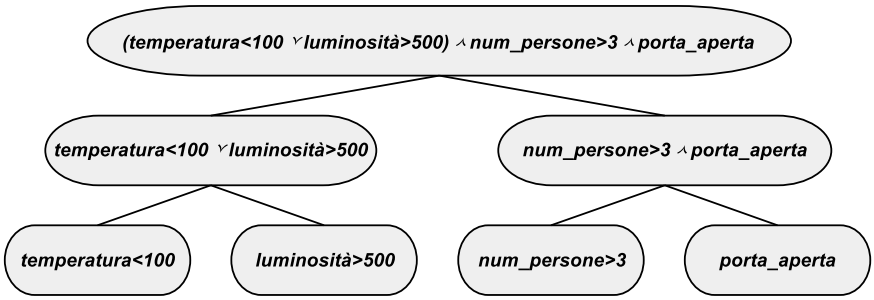

Figura 10: Esempio di risoluzione di una regola.

4.1.3 Interfaccia per i Virtual Object

Dalle considerazioni precedenti, l'interfaccia base dei Virtual Object utile per interagire con l'architettura descritta è mostrato nel Codice 1, dove: *checkValue* è il metodo per leggere una data risorsa fisica; *acting* permette di eseguire una operazione di attuazione che produce una modifica nello smart environment; mentre i due metodi *setRule* servono ad impostare le regole atomiche (vedi sezione 4.1.2).

Codice 1: Interfaccia per Virtual Object.

```
interface VirtualObjectInterface {
    VOResult checkValue(VOServiceId serviceId, VOServiceParams
        params);
    VOResult acting(VOServiceId serviceId, VOServiceParams
        params);
    void setRule(VOServiceId serviceId, VOServiceParams params,
        Operator operator, VOValue threshold, RuleMatchedListener
         listener);
    void setRule(VOServiceId serviceId, VOServiceParams params,
        RuleMatchedListener listener);
}
```

VOServiceId e *VOServiceParams*, dato un Virtual Object, identificano univocamente una risorsa fisica, come spiegato precedentemente. *Operator* in *setRule* può valere banalmente $>$ oppure $<$ (maggiore o minore), mentre *threshold* è il valore di soglia.

Lultimo parametro di *setRule* è un oggetto *ascoltatore* che viene notificato quando la regola è soddisfatta o meno, invocando uno dei metodi della sua interfaccia,

descritta nel Codice 2.

Codice 2: Interfaccia RuleMatchedListener.

```
interface RuleMatchedListener {
    void ruleMatched();
    void ruleNotMatched();
}
```

nel seguito mostreremo una estensione di tale interfaccia per permettere di gestire flussi di dati e query CEP

4.2 Primitive per flussi di dati e CEP

Nella precedente sezione è stata introdotta l'architettura Rainbow. La struttura e le caratteristiche principali della suddetta architettura sono stati dettagliati. In particolare, è stato introdotto il concetto di VO e le sue funzionalità sono state descritte mostrando la sua interfaccia software. Tali funzionalità riguardavano la lettura/scrittura delle risorse fisiche e la possibilità di poter definire, e sfruttare in modo publish/subscribe, degli eventi tramite delle proposizioni logiche. E' importante sottolineare che l'interfaccia sinora presentata (e dettagliata nei precedenti deliverable) non è idonea per gestire flussi e per usare query CEP come quelle descritte nelle prime sezioni. Per poter ospitare queste nuove funzionalità è opportuno estendere l'interfaccia VO arrivando a quella mostrata nel Codice 1.

Codice 3: Interfaccia per Virtual Object.

```
interface VirtualObjectInterface {
    VOResult checkValue(VOServiceId serviceId, VOServiceParams
       params);
    VOResult acting(VOServiceId serviceId, VOServiceParams
       params);
    void setRule(VOServiceId serviceId, VOServiceParams params,
       Operator operator, VOValue threshold, RuleMatchedListener
        listener);
    void setRule(VOServiceId serviceId, VOServiceParams params,
       RuleMatchedListener listener);
    void streamQueryValue(String query, VONewValueListener
       listener, List<VOResource> resources);
}
```