POLITECNICO DI MILANO

V Facoltà di Ingegneria

Corso di laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Implementazione e analisi di protocolli per il rilevamento distribuito di eventi complessi

Relatore: Prof. Gianpaolo Cugola

Tesi di laurea di:
Alberto Negrello Matr. 675281

Anno Accademico 2010 - 2011

Indice

Elenco delle figure **iv**

1 Introduzione **1**

1.1 DSP systems 4

1.2 CEP systems 6

1.3 Obiettivi di questo lavoro 9

1.4 Organizzazione dell'esposizione 10

2 TESLA **12**

2.1 TRIO: breve descrizione 16

2.2 Struttura delle regole di TESLA 18

2.3 Semantica delle regole 18

2.4 Operatori di TESLA 20

2.4.1 Occorrenza di un evento 20

2.4.2 Policy di sezione 21

2.4.3 Parametrizzazione 22

2.4.4 Negazione 23

2.4.5 Aggregazione 24

2.4.6 Riuso degli eventi 24

2.4.7 Gerarchia di eventi 26

2.4.8 Ripetizioni 26

2.4.9 Timers 27

3 T-REX: Algoritmo **28**

3.1 Prima implementazione: *Automa* 30

3.2 Seconda implementazione: *Stacks* 37
3.2.1 Window e policy di selezione e consumo 43
3.2.2 Parametrizzazione 44
3.2.3 Negazione 45
3.2.4 Aggregazione 46

4 T-REX: Distribuzione 47
4.1 Distribuzione del procollo 49
4.2 Distribuzione dell'algoritmo 53
4.2.1 Rule Deployer 53
4.2.2 Rule Handler 56
4.3 Fasi del protocollo 57
4.3.1 Start-up 57
4.3.2 Advertisements 58
4.3.3 Gestione delle regole 58
4.3.4 Subscriptions 60
4.3.5 Publications 60
4.3.6 Notifications 61
4.4 Modalità alternativa: Push-Pull 62
4.5 Protocolli implementati 67

5 Ambiente di simulazione: OMNet++ 68
5.1 OMNet++ 68
5.2 Simulatori ad eventi discreti 68
5.3 Punti di forza di OMNet++ 70
5.4 Caratteristiche Principali 71
Moduli e loro gerarchia 71
Messaggi, interfacce e link 72
Modello di trasmissione dei pacchetti 73
Parametri 74
Topologia del modello 74
Modello di programmazione 76
Esecuzione della simulazione 77

6 Dettagli implementativi **78**
6.1 Architettura della simulazione 79
6.1.1 Diagrammi di struttura 81
6.1.2 Diagrammi d'interazione 92
6.2 Architettura della logica applicativa 96
6.2.1 Diagrammi di struttura 98
6.2.2 Diagrammi d'interazione 102
6.3 Casi d'uso 104

7 Simulazioni effettuate **106**
7.1 Scenario di riferimento 106
7.2 Confronto con gli automi 108
7.3 Esecuzione delle simulazioni 112
7.4 Parametri simulati e risultati 113
7.4.1 Numero di nodi 113
7.4.2 Numero delle regole 115
7.4.3 Numero di eventi primitivi 116
7.4.4 Frequenza delle pubblicazioni 117
7.4.5 Numero di valori validi per ogni attributo 118
7.4.6 Dimensione delle regole 119
7.4.7 Numero di sottoscrizioni 120
7.4.8 Località degli eventi 121
7.4.9 Altri parametri 122
7.5 Analisi generale dei risultati 124

8 Lavori correlati **126**

9 Conclusioni **135**

Bibliografia **I**

Elenco delle figure

1.1 Differenza tra DBMS e DSP 4

2.1 Esempio di ricezione di eventi primitivi 15

3.1 Grafo d'ordine della regola precedente 30
3.2 Automa per la regola *R1* 32
3.3 Un esempio di elaborazione per la regola *R2* 33
3.4 Architettura del sistema implementato tramite Automa 36
3.5 Stacks per la regola R4 38
3.6 Un esempio di elaborazione utilizzando gli Stacks 40
3.7 Architettura del sistema implementato tramite Stacks 43
3.8 Tipologie di negazioni 45

4.1 Confronto tra protcollo ad albero singolo e multiplo 52
4.2 Situazione di esempio per la regola R5 54
4.3 Esempio di finestre aperte tra sotto-regole 64
4.4 Esempio di impiego del protocollo Push-Pull 66
4.5 Stati possibili per un evento 67

5.1 Gerarchia dei moduli di OMNet++ 72
5.2 Tipologie di connessioni tra moduli 73
5.3 Esempio di file `omnet.ini` 74
5.4 Esempio di *NED file* 75

6.1 Architettura di un nodo del modello di simulazione 80
6.2 Classe Publisher 82

6.3 Classe Subscriber 83
6.4 Classe RuleManager 83
6.5 Classe TopologyManager 84
6.6 Classe Processor 86
6.7 Classe TrafficMonitor 87
6.8 Classe WorkloadHandler 88
6.9 Classi Encoder e Decoder 89
6.10 Pacchetti principali impiegati dal simulatore 91
6.11 Altri pacchetti impiegati dal simulatore 92
6.12 Generazione di una regola 94
6.13 Pubblicazione di un evento 94
6.14 Ricezione di un terminatore 95
6.15 Ricezione di un evento composito 95
6.16 Architettura della logica applicativa 97
6.17 Classe Stack 98
6.18 Classe IndexingTable 99
6.19 Classe DistributedRuleHandler 99
6.20 Classe StacksRule 101
6.21 Classe DupRemover 102
6.22 Classe RuleDeployer 102
6.23 Elaborazione di una regola 103
6.24 Elaborazione di un evento 104
6.25 Installazione di una regola 105
6.26 Gestione di una pubblicazione 105

7.1 Througput a confronto nel caso *each-within* 109
7.2 Tempo di elaborazione a confronto nel caso *each-within* 109
7.3 Througput a confronto nel caso *last-within* 110
7.4 Tempo di elaborazione a confronto nel caso *last-within* 110
7.5 Througput a confronto nel caso *first-within* 111
7.6 Tempo di elaborazione a confronto nel caso *first-within* 111
7.7 Prestazioni al variare del numero di nodi 114
7.8 Prestazioni al variare del numero delle regole 115

7.9 Prestazioni al variare del numero di eventi primitivi 116
7.10 Prestazioni al variare del *rate* di pubblicazione 117
7.11 Prestazioni al variare del numero di valori per attributo 118
7.12 Prestazioni al variare della lunghezza delle sequenze 119
7.13 Prestazioni al variare del numero di sottoscrizioni 120
7.14 Prestazioni al variare della località degli eventi 121
7.15 Prestazioni al variare del numero di negazioni 122
7.16 Prestazioni al variare del numero di aggregati 123
7.17 Prestazioni al variare della dimensione delle finestre 123
7.18 Prestazioni al variare della percentuale di each 123
7.19 Prestazioni al variare della percentuale di each 125

Capitolo 1

Introduzione

Molti ambiti applicativi richiedono l'osservazione di eventi, una loro elaborazione ed una reazione agli stessi. Queste caratteristiche portano direttamente alla necessità di un *Engine* che si occupa di interpretare, filtrare e combinare *eventi primitivi* che accadono nell'ambiente esterno, con lo scopo di identificare *eventi compositi*, sulla base di una serie di regole.
Obiettivo finale dell'engine è quello di inviare notifica degli eventi compositi riconosciuti, ai componenti che sono adibiti a reagire a questi ultimi.
Esempi di sistemi che operano in questo modo sono le reti di sensori per il monitoraggio ambientale [10, 17], le applicazioni finanziarie che necessitano di un continuo controllo sull'andamento delle azioni per determinarne i trend [15], i sistemi di individuazione delle frodi che analizzano le transizioni dei circuiti delle carte di credito, per individuare sequenze di operazioni sospette [31], i sistemi di gestione dei magazzini basati su RFID che effettuano controlli continui sui dati registrati per tracciare i percorsi validi e scoprire irregolarità nel processo di spedizione [32]. Più in generale il sistema informativo di ogni sistema complesso può essere organizzato sulla base di un *event-based core*, che realizza una sorta di sistema nervoso e determina le operazioni effettuate da ogni altro sottosistema.
L'architettura generica di queste applicazioni basate su eventi si compone su tre diverse entità:

sources: osservano gli eventi primitivi che si verificano nell'ambiente di funzionamento;

sinks: pubblicizzano il loro interesse nei confronti degli eventi compositi e ne ricevono notifica dal sistema qualora si verifichino;

engine: è il componente fondamentale del sistema ed è normalmente parte di un *middeleware* che include anche le librerie lato client per accedere ai servizi forniti.

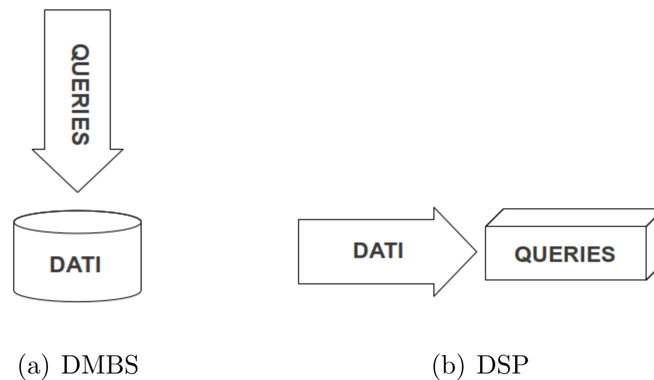
L'ultimo componente del sistema è quello che effettua il riconoscimento degli eventi compositi a partire dagli eventi primitivi, interpretando una serie di *regole di definizione di evento*, che descrivono come gli eventi compositi siano formati a partire da quelli primitivi, che li compongono.
Sia per la natura spesso intrinsicamente distribuita dei componenti della rete che si occupano di rilevare l'accadimento degli eventi primitivi (per esempio i sensori disposti omogeneamente nelle aree di interesse), sia per la dislocazione dei sinks interessati al riconoscimento delle sequenze che generano eventi complessi e alla notifica delle stesse, si è cercato di introdurre degli *algoritmi distibuiti di processing* così che anche l'engine operi in maniera distribuita all'interno della rete, ricalcando l'ambiente in esame e scalando meglio al crescere di quest'ultimo. Per ottenere questo risultato, però, si devono tenere in considerazione vari aspetti del *processing distribuito* che complicano lo sviluppo di un algoritmo.
Dal momento che consideriamo uno scenario in cui i vari nodi che compongono il sitema risiedono su macchine distinte, la comunicazione tra i vari componenti dell'engine è affidata ai *messaggi*. Un problema che si deve affrontare risulta essere quello di limitare l'overhead introdotto, rispetto al caso centralizzato, proprio dall'invio dei messaggi impiegati, per lo scambio di informazioni e per coordinare il lavoro dei vari processi che compongono l'engine. I vari processi che attuano l'algoritmo di riconoscimento delle sequenze di eventi semplici necessitano di comunicare tra di loro, non solo per trasmettere i risultati parziali, ma anche per sincronizzarsi riguardo alle operazioni già intraprese e quelle da intraprendere. Nel caso distribuito, infatti,

è possibile che un processo, prima di procedere con le sue operazioni debba attendere dei dati remoti, ottenuti da un altro elemento dell'engine.
Oltre a questo primo problema, la necessità di coordinamento tra i processi impone di considerare il problema del *clock synchronization* e perlomeno definire una relazione *happened before*, che permetta di ordinare i messaggi in transito nel sistema. Questo è necessario, sia per quanto riguarda i messaggi contenenti informazioni circa gli eventi o le sequenze degli stessi, sia per quelli di coordinamento tra i componenti dell'engine. Nel primo caso si corre il rischio di non intercettare sequenze di eventi primitivi valide o al contrario di individuarne alcune in maniera impropria; nel secondo caso la mancanza di sincronia tra i vari nodi può portare a comportamenti anomali da parte dell'engine.
Infine nel caso in cui diversi elementi cerchino di accedere ed elaborare contemporaneamente gli stessi dati si avranno degli accessi remoti concorrenti da parte di processi che devono condividere le stesse risorse e che di conseguenza necessitano di sincronizzarsi tra loro introducendo così dei meccanismi di *distributed concurrency control*. Ne può essere esempio il caso di un engine partizionato su nodi differenti, sulla base dell'evento composito da riconoscere, che debba elaborare eventi che interessano differenti regole.
Recentemente sono stati sviluppati una serie di sistemi con l'obiettivo di implementare il ruolo di middleware per questa classe di problemi: il riconoscimento degli eventi complessi. Questi sistemi, e i loro linguaggi di definizione delle regole, sono stati proposti solo recentemente e non risultano soddisfare esaustivamente tutti i requisiti dell'ambito applicativo.
Sebbene alcuni di questi linguaggi siano esplicitamente designati per catturare ed elaborare notifiche di eventi, la maggior parte sono stati sviluppati con il generico scopo di elaborare, in tempi brevi, una grande quantità di dati, che confluiscono dalla periferia al centro del sistema. Di conseguenza i sistemi sviluppati possono essere riferiti a due differenti categorie sulla base dell'approccio utilizzato: i primi sono noti col nome di *Complex Event Processing (CEP) systems* [25] e si basano su un'architettura *publish-subscribe* di tipo content-based, mentre i secondi prendono il nome di *Data Stream Processing (DSP) systems* [8] e derivano dall' evoluzione delle basi di dati

attive.

1.1 DSP systems

A questa tipologia appartengono i *DSMSs (Data Stream Management Systems)*, termine coniato per contrapporli ai *DBMSs (DataBase Management Systems)*, dai quali derivano. A differenza dei DBMSs, i DSMSs non necessitano che i dati vengano indicizzati prima di essere elaborati, bensì permettono la loro analisi mentre fluiscono dalle *sources* ai *sinks*. Loro principale caratteristica è quella di non operare su un insieme relativamente stabile di dati eseguendo query diversificate sugli stessi, viceversa utilizzare un insieme stabile di query da eseguire in tempi brevi su grandi moli di tuple in continua evoluzione come mostrato in figura 1.1.



(a) DMBS (b) DSP

Figura 1.1: Differenza tra DBMS e DSP

In particolare, concentrandosi sul loro linguaggio di definizione delle regole, si può osservare come questi ultimi non siano adeguati per il riconoscimento di patterns composti da elementi correlati da complesse relazioni temporali. Per descrivere questa mancanza consideriamo CQL, un linguaggio di riferimento tra i DSMSs. La sua sintassi è molto simile a quella di SQL ed ogni regola di definizione degli eventi è composta a partire da tre operatori:

Stream-to-Relation (S2R): anche note come finestre. Si occupano di partizionare un flusso informativo, creando una tradizionale tabella di una base di dati.

Relation-to-Relation (R2R): sono tipicamente operatori SQL standard;

Relation-to-Stream (R2S): generano nuovi flussi di dati dalle tabelle, dopo la loro manipolazione.

Ogni regola CQL è composta da un operatore S2R, uno o più operatori R2R, e un operatore R2S. Come conseguenza, l'intera operazione di elaborazione avviene in tabelle relazionali che non tengono traccia dell'ordinamento tra i dati. L'unico meccanismo che tiene conto dei *timestamps*, le finestre, operano separatamente su porzioni differenti di dati e precedentemente all'elaborazione degli stessi. Questo, unito alla mancanza di operatori temporali specifici, ed al fatto che i timestamps, se non introdotti artificiosamente nello schema delle tuple, non possono essere utilizzati dagli operatori R2R, rende complesso e poco naturale individuare e rappresentare ordinamenti temporali tra i dati.

Distribuzione dell'elaborazione: *Operator Placement*

Per quanto riguarda i DSP gli sforzi nel distribuire l'elaborazione sono mirati, visto il paradigma impiegato, a distribuire in maniera ottimale gli operatori SQL ll'interno della rete. In ambienti in cui i dati vengono collezionati agli estremi della rete, infatti, è desiderabile effettuare l'elaborazione delle query *in-network*: ovvero eseguire parte dell'esecuzione in nodi intermedi della rete così da minimizzare la trasmissione dei dati verso il nodo radice.
La sfida maggiore in questo caso è quella dell'*operator placement* ovvero la definizione di un algoritmo di scelta del posizionamento, lungo i nodi della rete, degli operatori (dai meno ai più costosi in termini computazionali) che effettuano i calcoli della query. Diventa dunque fondamentale definire un modello ed una relativa funzione di costo che tengano in considerazione sia il procesing necessario sia l'overhead introdotto, così da poter discriminare il miglior livello della gerarchia di rete ove posizionare il singolo operatore.

Un'ulteriore difficoltà in questo ambito e quella di individuare una soluzione ottimale al problema dal momento che alcuni operatori possono essere impiegati da query differenti, indirizzate verso radici diverse.
Sono, infine, stati studiati casi [27] in cui l'operator placement possa avvenire in maniera automatica, e non manualmente come avviene nella maggior parte dei sistemi DSP distribuiti. A questo scopo si può strutturare un *ovelay network* basata sui flussi di dati che associ una funzione di costo ad una rappresentazione astratta della rete e dei flussi di dati, consentendo così di prendere decisioni sul posizionamento degli operatori in maniera automatica e dinamica.

1.2 CEP systems

I CEP system, a differenza dei DSP systems, per la loro architettura di tipo publish-subscribe content-based, si basano su linguaggi che sono stati esplicitamente sviluppati con lo scopo di individuare eventi compositi a partire da quelli primitivi. Questi linguaggi, infatti, descrivono nativamente il flusso informativo come notifiche di eventi accaduti in un determinato istante di tempo.
Negli ultimi anni sono state proposte diverse soluzioni di questo tipo. Inizialmente basate su un *dispatcher* centralizzato, nel quale vengono accentrate tutte le funzionalità di riconoscrimento di patterns e di notifica di eventi compositi, si sono presto spostate verso soluzioni di tipo distribuito per migliorare la scalabilità del sistema.
La maggior parte dei linguaggi su cui si basano questi sistemi, però, sono estremamente semplici e presentano serie limitazioni per quanto riguarda la loro espressività. Ad esempio, alcuni di questi linguaggi forzano le sequenze di eventi primitivi, componenti di quelli compositi, ad essere temporalmente adiacenti tra loro, rendendo impossibili esprimere vincoli temporalmente più complessi.
Come soluzione implementativa per i CEP systems e, più precisamente, per gli algoritmi di riconoscimento dei patterns, il paradigma degli automi è il più impiegato nei sistemi esistenti. Normalmente questi algoritmi sono limi-

tati al riconoscimento di una singola *sequenza* di eventi, per ogni regola di definizine di evento e hanno politiche di selezione e di consumo degli eventi impiegati nel riconoscimento, definite a priori, quindi non personalizzabili.

Progettazione di un CEP system

Nella progettazione di un sistema CEP, che vada oltre i limiti di quelli esistenti, bisogna tenere in considerazione vari aspetti e vari livelli sui quali operare delle scelte. A livello più alto è necessario ideare un linguaggio di definizione degli eventi compositi che permetta di effettuare quest'operazione in maniera non ambigua e che abbia un'espressività tale da permettere la produzione di regole complesse sulle base delle notifiche di eventi primitivi che possono verificarsi. In questa fase le difficoltà principali si hanno nell'ottenere un linguaggio formale che non presenti forme di ambiguità e, contemporaneamente, possegga tutti gli operatori necessari. Se da un lato, infatti, una riduzione degli operatori disponibili rende meno probabile la produzione di grammatiche ambigue, dall'altro fa sì che sia più complesso e artificioso e, a volte, perfino impossibile descrivere complesse relazioni temporali, limitando l'espressività del linguaggio che si vuole specificare. Nel trovare un compromesso bisogna, dunque, tenere in considerazione anche la semplicità di utilizzo dello stesso. É necessario, inoltre, produrre un'*Application Programming Interface (API)* che presenti ed esporti tutte le funzionalità del middleware e dichiari le procedure delle quali possono avvalersi gli utilizzatori dello stesso.
Un secondo livello su cui bisogna ragionare nella produzione di un sistema CEP è quello dell'elaborazione stessa degli eventi primitivi. In questa fase bisogna operare delle scelte per definire un algoritmo che si occupi di filtrare in maniera efficiente tutti gli eventi verificatisi nella rete allo scopo di individuare, per ogni evento composito installato, tutte le sequenze valide. In questa fase le alternative devono essere valutate principalmente sulla base di due fattori che determinano i requisiti a cui l'agoritmo dovrà attenersi: la velocità nell'effettuare il filtraggio ed il riconoscimento di una singola sequenza e la quantità di memoria impiegata, a regime, per tener traccia degli eventi

primitivi che possono potenzialmente generare un evento composito. Il primo è dovuto alla produzione, negli scenari considerati, di una grande quantità di notifiche da parte della rete e all'importanza della loro tempestiva ricezione, da parte dei sottoscrittori, per sfruttare appieno i vantaggi dovuti al sistema CEP ed ottenere un tempo di risposta accettabile per l'ambito di impiego. Il secondo, invece, è imputabile alle limitazioni che i nodi della rete potrebbero presentare sulla base delle loro risorse. Quest'ultimo requisito risulta essere ancora più stringente nel caso si voglia simulare un' intera rete su un'unica macchina, per avere indicazione circa le performances del sistema.
Un altro fattore da tener presente nello sviluppo di un sistema CEP è rappresentato dalla distribuzione dell'elaborazione sui *cores* di un sistema multiprocessore, avvantaggiandosi, in questa maniera di un *processing* parallelo. Bisogna a questo scopo suddividere l'engine in moduli il più indipendenti possibili, così da mitigare i ritardi introdotti dalla collisione degli accessi concorrenti ad una risorsa, tenendo comunque in considerazione i requisiti di memoria precedentemente discussi. Ad esempio suddividendo l'engine sulla base delle regole degli eventi compositi, bisognerà individuare un compromesso tra la duplicazione degli eventi primitivi (dal momento potrebbero essere validi per diverse sequenze) e il tempo di attesa per accedere agli stessi da parte di processi, in attesa che venga rilasciato il *lock*.
Nel caso si scelga di sviluppare un algoritmo distribuito non solo localmente ma, come esposto precedentemente, all'interno della rete, bisognerà trarre vantaggio della distribuzione facendo sì che tutte le operazioni di filtraggio e di inviduazione dei risultati vengano affettuate nei nodi che per primi posseggono tutte le informazioni necessarie, nell'albero che va dai generatori degli eventi primitivi ai sottoscrittori degli eventi compositi. In questa fase bisogna prestare attenzione, di conseguenza, a come l'engine suddivide la computazione sulla base della sua posizione gerarchica all'interno dell'albero. Così facendo si compie un'ottimizzazione dell'engine, analoga a quella che si ottiene con l'operator placement.

1.3 Obiettivi di questo lavoro

Per rispondere alla necessità di espressività e di efficienza, è stato pensato *T-Rex*, un middleware studiato per supportare semplicemente ed in maniera efficiente l'elaborazione di eventi complessi in sistemi su larga scala.
É stato scelto come paradigma quello dei sistemi CEP e si è deciso di strutturarlo in maniera distribuita. Questa scelta è stata effettuata con l'obiettivo di ottimizzare contemporaneamente la quantità di messaggi in transito sulla rete e la politica di operator placement. Impiegando un modello pubblish-subscribe, infatti, l'engine non necessita di inviare a tutti i nodi l'intero *stream* informativo, come avviene dei DSP, ma i singoli eventi attraversano la rete e possono, grazie alla distribuzione dell'algoritmo, non essere più ritrasmessi verso la radice della stessa una volta che violino la definizione della sequenza valida. Otteniamo inoltre il risultato di effettuare tutte le operazioni di selezione e filtraggio degli eventi non appena sia possibile mediante una scomposizione delle operazioni da compiere sugli eventi primitivi, che tenga conto degli advertisements dei generatori, per individuare le sequenze valide. Questo processo di scomposizione fa sì che partizioni della regola, contenenti determinati eventi, vengano istallate esclusivamente in nodi che ne hanno ricevuto notifica dal proprio sottoalbero (ovvero ne è stata notificata la generazione da parte di un figlio, appartenente alla porzione dell'albero che ha come radice il nodo stesso, mediante un advertisement).
Per ottenere questo risultato, T-Rex combina un linguaggio per le regole di definizione di eventi, concepito esplicitamente con l'obiettivo di permettere operazioni quali il filtraggio, la combinazione e l'aggregazione di eventi primitivi, con un algoritmo efficiente, per interpretare le regole e processare una grande mole di eventi in ingresso.
In particolare, il linguaggio utilizzato è *TESLA*, che permette la descrizione in maniera semplice e naturale di relazioni complesse che legano eventi primitivi e la loro aggregazione in eventi compositi.
Scopo di questo lavoro è stato ideare il T-Rex engine, che si occupa di tradurre le regole di TESLA in un algoritmo di riconoscimento dei patterns. Per raggiungere questo obiettivo, si è inizialmente utilizzato un algoritmo basato

sugli *automi* e successivamente uno basato sugli *stack*.
Un ulteriore scopo è stato quello di strutturare un ambiente di simulazione, per poter verificare le performances del middleware ideato. L'ambiente di simulazione impiegato è stato Omnet++, tramite il quale sono stati definiti una serie di moduli che replicano le operazioni del middleware e dell'hardware di rete. Sono stati studiati inoltre diversi scenari, per confrontare e valutare differenti soluzioni.

1.4 Organizzazione dell'esposizione

Questo documento procede, nel Capitolo 2, con la presentazione del linguaggio TESLA, impiegato nella definizione delle regole. Verranno prese in esame le sue principali caratteristiche e gli operatori messi a disposizione.
Verrà mostrata così la sua espressività nel definire vincoli temporali e la naturalezza con cui gli stessi possono essere personalizzati, permettendo la rappresentazione di un vasto insieme di patterns.
Successivamente, nel Capitolo 3, viene presentato l'algoritmo di T-Rex durante tutte le sue fasi di funzionamento e viene, inoltre, descritto il sistema, sia nella sua iniziale versione con automi, sia nella successiva che impiega gli stack. Di questa seconda implementazione dell'algoritmo di riconoscimento, ne viene analizzato il funzionamento per i vari operatori del linguaggio TESLA che implementa.
Il successivo Capitolo 4 analizza il funzionamento del sistema dal punto di vista della distribuzione analizzando le fasi ed i componenti che la permettono.
Il Capitolo 5 si occupa, invece, di descrivere Omnet++, lo strumento di simulazione impiegato e mostrare l'ambiente di sviluppo e tre protocolli presi in esame per il confronto.
Le implementazioni, sia dell'ambito simulativo, sia di quello algoritmico, sono descritte nel Capitolo 6.
La descrizione delle simulazioni effettuate ed i risultati ottenuti vengono successivamente documentati nel Capitolo 7.

Nel successivo Capitolo 8 vengono presentati dei lavori connessi a T-Rex. Infine, nel Capitolo 9 si esprimono le conclusioni finali sul lavoro effettuato.

Capitolo 2

TESLA

In questo capitolo presentiamo TESLA *(Trio-based Event Specification LAnguage)* [14], il linguaggio di definizione delle regole scelto per T-Rex. Consideriamo un'applicazone di controllo ambientale, che elabora informazioni provenienti da una rete di sensori. I sensori notificano la posizione in cui si trovano, la temperatura che misurano, e la presenza di fumo, nel caso in cui un utente dell'applicazione voglia essere avvisato della presenza di un incendio.
Vi è la necessità di insegnare al sistema questa situazione, sulla base dei dati grezzi provenienti dai sensori. A seconda dell'ambiente, dei requisiti dell'applicazione, delle preferenze dell'utente, la nozione di incendio può essere definita con differenti modalità.
Prendendo, ad esempio, quattro possibili definizioni dell'occorrenza di questo evento, analizziamo le caratteristiche che un sistema CEP dovrebbe rendere disponibili.

i. Un incendio avviene quando una temperatura superiore a 45 gradi e la presenza di fumo vengono segnalati dalla stessa area entro 3 minuti. La segnalazione dell'incendio deve registrare la temperatura attualmente misurata.

ii. Un incendio avviene quando viene rilevata una temperatura superiore a 45 gradi e non viene segnalata pioggia nell'ultima ora.

iii. Un incendio avviene quando è rilevata la presenza di fumo e la temperatura media negli ultimi 3 minuti è superiore a 45 gradi.

iv. Un incendio avviene quando viene registrata una sequenza di almeno 10 temperature con valori crescenti e la presenza di fumo viene segnalata entro 3 minuti.

Le regole qui descritte mostano le caratteristiche tipiche che sono richieste ad un sistema CEP.
In primo luogo, le regole devono dar modo di *selezionare* degli eventi sulla base di vincoli.
Vengono presentate due tipologie di vincoli: la prima tipologia consente di selezionare degli elementi sulla base del valore che posseggono (ad esempio seleziono nella regola *(i)* gli elementi che riguardano temperature superiori a 45 gradi).
La seconda consente la selezione sulla base di relazioni temporali tra differenti elementi (ad esempio, sempre nella regola *(i)*, sono interessato alle segnalazioni di fumo generate entro 3 minuti da quelle della temperatura superiore a 45 gradi).
La regola *(i)* introduce, inoltre, la *parametrizzazione*, ovvero la possibilità di definire vincoli sui valori di elementi differenti (ad esempio, la selezione delle notifiche di temperatura e fumo provenienti dalla stessa area).
La regola *(ii)* dimostra la necessità di un operatore di *negazione*, che richiede l'assenza di un determinato evento (ad esempio l'assenza di una segnalazione di pioggia entro un'ora dalla rilevazione di una temperatura superiore a 45 gradi).
La regola *(iii)* presenta un caso in cui vi è la necessità di rappresentare dei valori *aggregati* (ad esempio, definendo una funzione da applicare ad uno specifico sottoinsieme di valori).
Infine la regola *(iv)* dimostra come la selezione ed i vincoli temporali possano essere combinati in maniera complessa (ad esempio, richiedendo una sequenza di valori crescenti); questo tipo di vincolo è utile per descrivere le regole che hanno lo scopo di individuare un andamento specifico dei dati.
I linguaggi esistenti, appartenenti sia alla categoria dei DSP systems che a

quella dei CEP systems, presentano delle limitazioni nel definire in maniera semplice e non ambigua i vincoli mostrati nell'esempio precedente.

Limitazioni dei DSP systems

Considerando gli elementi di questi categoria (in particolar modo CQL [6], che ben rappresenta le loro capacità espressive) si può notare che, traducendo lo stream di dati in tabelle relazionali, viene persa ogni traccia dell'ordinamento tra i singoli eventi, a meno di inserire artificiosamente, come parte dello schema relazionale, informazioni sull'ordinamento temporale (ad esempio un timestamp). Come risultato, una regola come la *(iv)* dell'esempio precedente, che considera una sequenza di elementi, risulta difficile da descrivere, o addirittura impossibile, se non sono state incluse nello schema della tupla espliciti riferimenti temporali. Ovvero, tutte le interrogazioni che selezionano un sottoinsieme di elementi tra quelli precedentemente ricevuti, utilizzando dei vincoli temporali, non trovano una traduzione naturale in CQL (e più in generale nei linguaggi proposti dai DSP systems), data la mancanza di operatori che descrivano esplicitamente il concetto di sequenza. Sono sviluppati con l'obiettivo di definire trasformazioni continue di stream in ingresso in stream in uscita, astrazione utile nel caso di elaborazione di dati in tempo reale, ma che mostra numerose limitazioni quando si tratta di descrivere patterns complessi, che selezionano solo una partizione dell'insieme dei dati ricevuti.
Esistono DSP system che estendono l'espressività di CQL: un esempio è ESL. Con questi sistemi diventa possibile produrre patterns con la complessità necessaria a descrivere le regole presentate negli esempi precedenti, ma rimane complesso e lontano da una sintassi naturale. Questi sitemi incorrono, inoltre, nella mancanza di rigorosità semantica, che affligge anche i CEP systems esistenti.

Limitazioni dei CEP systems

Questo modello di sistemi, specificatamente sviluppato per catturare complesse relazioni temporali tra gli eventi, si addice meglio ad esprimere in maniera naturale le regole del nostro esempio. Ciò nonostante, i linguaggi introdotti dai sistemi esistenti soffrono di due problemi.
La maggior parte di questi linguaggi sono estremamente semplici e hanno grandi limitazioni in termini di espressività (ad esempio, raramente consentono operazioni di parametrizzazione e di aggregazione).
In secondo luogo, raramente viene fornita, insieme al linguaggio, una descrizione formale dello stesso e ciò porta ad ambiguità di tipo semantico.
Un esempio di quest'ultimo può essere mostrato analizzando la precedente regola *(i)*. Questa definisce una semplice congiunzione di due eventi primitivi, operazione supportata da praticamente tutti i CEP systems (ad esempio [16] e [9]).

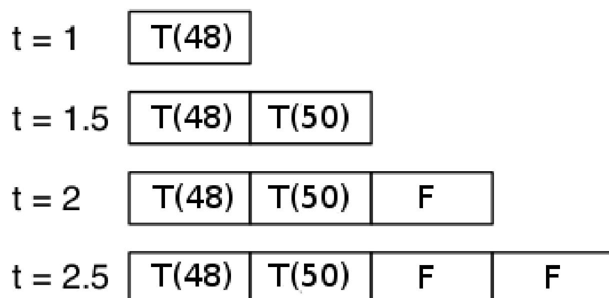


Figura 2.1: Esempio di ricezione di eventi primitivi

Considerando la Figura 1 emergono, comunque alcuni problemi. Supponiamo che tutti gli eventi derivino dalla stessa area e che $T(x)$ rappresenti la notifica dell'evento $temperatura = x$, mentre F la notifica della presenza di fumo.
Inizialmente viene registrato dal sistema un evento $T(48)$ seguito da $T(50)$. Quando viene ricevuto l'evento F, quante segnalazioni dell'evento composito incendio devono essere segnalate? una per ogni coppia $< T(.), F >$ o semplicemente una? ed in quest'ultimo caso la segnalazione dell'incendio, quale delle due temperature deve riportare?

La regola *(i)*, espressa nel linguaggio naturale, presenta un'ambiguità, ma purtroppo parte del CEP system (ad esempio [21], [28]) presenta linguaggi che non sono molto più precisi. Questo problema, legato alla selezione degli eventi primitivi interessati, prende il nome di *policy* di selezione degli eventi. Ora, supponendo che successivamente un secondo evento F venga segnalato, i due eventi $T(48)$ e $T(50)$ sono già stati impiegati nel riconoscimento del pattern descritto dalla regola *(i)*: devono essere considerati nuovamente o meno?
Questo secondo problema prende il nome di *riuso degli eventi*. Questi esempi mostrano, oltre alla mancanza di una semantica precisa, una limitata espressività, che potrebbe non essere sufficiente per esprimere differenti comportamenti, che differenti applicazioni richiedono.

2.1 TRIO: breve descrizione

TESLA è stato proposto con l'obiettivo di superare le limitazioni precedentemente mostrate dagli altri CEP systems. Ogni regola descritta tramite questo linguaggio, considera i dati in ingresso come notifiche di eventi e definisce come gli eventi compositi sono generati a partire da questi ultimi.
Nonostante una sintassi semplice, con un insieme limitato di operatori, TESLA è particolarmente espressivo e flessibile e consente di specificare vincoli temporali e di contenuto, parametrizzazione, negazioni, sequenze, aggregati, timers, e politiche completamente personalizzabili, per la selezione ed il riuso degli eventi.
Allo stesso tempo TESLA non incorre nelle ambiguità semantiche da cui sono affetti la maggior parte degli altri linguaggi CEP, dal momento che la sua semantica è formalmente specificata utilizzando *TRIO* [20, 24]: una logica temporale del primo ordine.
Il singificato di una formula TRIO non è assoluto: viene definito sulla base del corrente istante di tempo, che viene lasciato implicito nella formula. Queste caratteristiche lo rendono ideale per definire gli eventi e la loro occorrenza.
L'alfabeto di TRIO è composto da un insieme di nomi per variabili, funzioni e predicati, ed un insieme di operatori, compresi quelli proposizionali ($\wedge$, $\neg$),

i quantificaori (∀) e gli operatori temporali (*Futr, Past*).
TRIO è un linguaggio tipizzato, ma oltre ai tipi consentiti come valori di variabili, valori di ritorno e parametri delle funzioni, vi è un ulterioriore tipo di dato obbligatoriamente di tipo numerico: il *dominio temporale*. Viene fatta una distinzione tra le variabili, le funzioni e i predicati *dipendenti dal tempo* e quelle *indipendenti dal tempo*.
La sintassi di TRIO è ricorsivamente definita come segue:

- ogni variabile è un termine
- ogni funzione n-aria applicata a n termini è un termine

Se il termine è una variabile, il suo tipo è il tipo della variabile; se il termine è il risultato dell'applicazione di una funzione, allora il suo tipo è quello del codominio della funzione.

- Ogni predicato n-ario applicato a n termini di tipo appropiato è una formula
- Se A e B sono formule, $\neg A$ e $A \wedge B$ sono formule
- Se A è una formula e x è una variabile indipendente dal tempo, $\forall x\, A$ è una formula
- Se A è una formula e t è un termine di tipo temporale, allora *Futr(A, t)* e *Past(A, t)* sono formule

Sono poi definiti come al solito abbreviazioni per gli operatori proposizionali $\vee$, $\rightarrow$, $true$, $false$, $\leftrightarrow$ e per il quantificatore $\exists$.
Focalizzandoci sui due operatori temporali *Futr* e *Past*, il loro significato è il seguente: la formula *Futr(A, t)* (*Past(A, t)*) è vera se A è vera per t unità di tempo nel futuro (nel passato) rispetto all'istante corrente che viene lasciato implicito nella formula. Molti operatori temporali possono essere derivati da *Futr* e *Past*, TESLA ne utilizza due:*Alw(A)* e $WithinP(A, t_1, t_2)$, dove A è una formula e t_1, t_2 termini di tipo temporale. Sono così definiti:

$$Alw(A) = A \wedge\ \forall t(t > 0 \rightarrow Futr(A, t)) \wedge\ \forall t(t > 0 \rightarrow Past(A, t))$$

$$WithinP(A, t_1, t_2) = \exists x(t_1 \leq x \leq t_1 + t_2 \wedge Past(A, x))$$

2.2 Struttura delle regole di TESLA

Ogni regola TESLA ha la seguente struttura:

$$\begin{array}{ll} define & CE(Att_1 : Type_1, ..., Att_n : Type_n) \\ from & Pattern \\ where & Att_1 = f_1, .., Att_n = f_n \\ consuming & e_1, .., e_n \end{array}$$

Intuitivamente le prime due righe definiscono un evento composito, a partire dagli eventi primitivi che la costituiscono, specificando la struttura dell'evento definito e il pattern di eventi semplici che portano al suo riconoscimento. La clausola *where* definisce i valori attuali per gli attributi Att_1, .., Att_n dell'evento composito utilizzando un insieme f_1, .., f_n di funzioni dipendenti dagli argomenti definiti nella clausola *Pattern*.
Infine, la clausola opzionale *consuming* definisce l'insieme di eventi primitivi che devono essere invalidati dal riconoscimento della regola.

2.3 Semantica delle regole

In TRIO possiamo esprimere l'occorrenza di un evento tramite un predicato dipendente dal tempo, che ha valore *true* nel momento in cui l'evento si verifica. D'altro canto in TESLA più eventi dello stesso tipo, anche con lo stesso valore degli attributi, possono avvenire allo stesso tempo.
Per differenziarli bisogna allora introdurre il concetto di *label*: un identificativo unico globale per la notifica degli eventi[1]. Mentre possiamo assumere che gli eventi primitivi, provenienti da fonti esterne, abbiano già la loro *label*, dobbiamo definire quelle degli eventi compositi definiti tramite le regole TESLA.

[1]Le *labels* sono necesserie solo per tradurre regole TESLA in formule TRIO, dal momento che non appaiono in TESLA che opera ad un più altro livello d'astrazione.

È necessario che, dato un insieme di eventi s, questo soddisfi una regola r al più una volta. Definiamo, allora, una funzione *lab* iniettiva che ritorni una nuova *label*, a partire da un identificatore della regola e un insieme di *labels* (quelle dell'insieme delle notifiche s che soddisfano la regola):

$$\forall\ r_1, s_1, r_2, s_2$$
$$((lab(r_1, s_1) = lab(r_2, s_2)) \ \leftrightarrow\ (r_1 = r_2 \wedge s_1 = s_2))$$

Una volta introdotte le *labels* possiamo utilizzarle per definire formalmente l'occorrenza di un evento tramite il predicato $Occurs(Type, Label)$, che ha valore *true* nel momento in cui l'evento di tipo *Type* con *label Label* avviene nel sistema. Il fatto che le *labels* siano uniche e che, data una notifica, essa avvenga una volta sola, viene formalmente catturato dalle seguenti formule:

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L((Occurs(e_1, l) \wedge Occurs(e_2, l)) \rightarrow e_1 = e_2)$$

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L, \forall t > 0\ (Occurs(e_1, l) \rightarrow$$
$$(\neg Past(Occurs(e_2, l), t) \wedge \neg Futr(Occurs(e_2, l), t)))$$

dove L è l'insieme di tutte le *labels* valide e E è l'insieme di tutti gli eventi. La prima formula descrive il fatto che nello stesso istante di tempo non possono esistere due notifiche con la stessa *label* e tipo diverso.
La seconda garantisce che, se un evento con *label* l si verifica all'istante t, non si possono verificare altri eventi con la stessa *label* in istanti differenti.
Introduciamo adesso il dominio N di tutti i nomi validi per gli attributi degli eventi. Dal momento che questi ultimi possono avere differenti tipi, definiamo una funzione indipendente dal tempo per ogni tipo X: $attVal_X : L \times N \rightarrow X$, che associa ad una combinazione di *label* l e nome dell'attributo n il valore dello stesso, nella notifica che posside *label* l. Assumiamo per semplicità che tutti gli attributi abbiano lo stesso dominio V, in modo da poter utilizzare una singola funzione $attVal$, per associare i nomi degli attributi ai loro valori. La generica regola descritta nel Capitolo 2.2 è tradotta nella seguente formula

TRIO (omettendo la clausula consuming):

$$Alw\ \forall l_1, .., l_m \in L, \forall n_1, .., n_n \in N$$
$$((Occurs(CE, lab(r, \{l_1, .., l_m\})) \ \leftrightarrow \ Pattern) \wedge$$
$$(Pattern \ \rightarrow \ attVal(lab(r, \{l_1, .., l_m\}), n_1) = f_1) \wedge$$
$$(Pattern \ \rightarrow \ attVal(lab(r, \{l_1, .., l_m\}), n_n) = f_n))$$

dove $\mathbb{N}$ è l'insieme di tutti i numeri naturali e $r \in \mathbb{N}$ rappresenta un identificatore univoco della regola TESLA, essendo $l_1, .., l_m \in L$ le *labels* di tutte le notifiche degli eventi individuate da *Pattern* e $n_1, .., n_n$ i nomi degli attributi per l'evento di tipo CE.

2.4 Operatori di TESLA

Vengono di seguito presentati tutti gli operatori disponibili in TESLA, per definire paterns validi.

2.4.1 Occorrenza di un evento

La tipologia più semplice di pattern per un evento rappresenta l'occorrenza di un singolo evento, il quale soddisfa una serie di restrizioni sul valore di uno o più attributi (*constraints*). Come esempio consideriamo la seguente situazione: *generare una notifica dell'evento incendio se la temperatura in una stanza supera i 45 gradi; la notifica deve contenere il nome della stanza.* Questi requisiti possono essere tradotti nella seguente regola TESLA:

$$\begin{array}{ll} define & Fire(Room) \\ from & Temp(Val > 45)\ as\ T \\ where & Room = T.Room \end{array}$$

Come mostra l'esempio TESLA pone le *constraints* di un evento tra parentesi, dopo la dichiarazione del tipo dell'evento. In questo esempio è necessaria una singola *constraint*, ma è possibile definirne anche un insieme separato da operatori logici (*and* e *or*). Per accedere ai campi di un evento TESLA

utilizza una notazione *nome-evento.nome-attributo*. Un nome è associato ad un evento tramite la parola chiave *as*, che può essere omessa e sostituita dal tipo dell'evento, nel caso in cui la regola utilizzi un unico evento di quel tipo.

2.4.2 Policy di sezione

Per descrivere l'occorrenza di diversi eventi correlati, TESLA mette a disposizione tre differenti policies di selezione degli stessi tramite gli operatori: *each-within*, *first-within*, and *last-within*[2]. Ognuno di questi operatori vincola l'occorrenza di un evento a quella di un altro, introducendo una finestra temporale di rilevamento *window*.
Prendiamo ad esempio le seguenti regole:

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke()\ and \\ & each\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\ where & Val = Temp.Val \end{array}$$

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke()\ and \\ & last\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\ where & Val = Temp.Val \end{array}$$

Entrambe le regole definiscono l'evento $Fire$ a partire dagli eventi $Smoke$ e $Temp$. La prima regola porta però alla notifica di un evento di tipo $Fire$ per ogni evento *Temp* maggiore di 45 gradi, occorso entro 5 minuti dalla rilevazione della presenza di fumo, mentre la seconda genera una singola notifica dell'evento *Fire*, selezionando solamente l'ultimo evento *Temp* maggiore di 45 gradi entro 5 minuti dalla riconosciuta presenza di fumo.
Nel primo caso viene definita dall'operatore *each-within* una policy di selezione multipla, dal momento che vengono selezionati tutti gli eventi disponi-

[2]Questi operatori sono anche detti operatori di composizione di eventi.

bili nella finestra temporale specificata.
Nel secondo caso, invece, *last-within* esprime una policy di selezione singola (l'operatore *first-within* si comporta analogamente a *last-within*, selezionando il primo elemento disponibile nella finestra temporale). TESLA offre anche una versione generale dell'operatore *last-within* e *first-within*, chiamati *k-last-within* e *k-first-within*.
Questi operatori possono essere impiegati per individuare il k-esimo elemento della finestra temporale, a partire rispettivamente dall'inizio o dalla fine della stessa, introducendo anch'essi una policy di selezione singola.
TESLA consente inoltre la definizione di regole che individuano l'occorrenza di più eventi che possono essere connessi in maniera seriale o parallela tra di loro, e nel caso non venga specificata una finestra temporale, ne viene utilizzata una di defaut. La regola seguente mostra un esempio che contiene sia relazioni seriali che parallele tra gli eventi:

$$\begin{array}{ll} define & D() \\ from & A()\ and\ each\ B()\ within\ 5min\ from\ A\ and \\ & last\ C()\ within\ 3min\ from\ A\ and \\ & last\ D()\ within\ 6min\ from\ B\ and \\ & first\ E()\ within\ 2min\ from\ D\ and \\ & E\ within\ 8min\ from\ A \end{array}$$

Si può notare, in particolare, l'uso dell'operatore *within* nell'ultima riga che introduce un'ulteriore finestra temporale per l'evento *E()* precedentemente definito.

2.4.3 Parametrizzazione

Consideriamo ancora l'esempio di regola della precedente sezione, che definisce l'evento *Fire* utilizzando l'operatore *each-within*. Venire a conoscenza dell'occorrenza di un evento *Smoke* e di un evento *Temp* entro 5 minuti può non essere un'informazione utile per definire l'evento composito, a meno che le due rilevazioni non provengano dalla stessa area.

Per esprimere questa tipologia di relazioni, TESLA mette a disposizione l'-operatore *$*. Supponendo che entrambi i due eventi primitivi posseggano un attributo chiamato *Area*, la seguente regola mostra l'uso dei parametri, per obbligare i due eventi *Temp* e *Smoke* a provenire dalla stessa area:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke(Area = \$x) \ and \\
 & each \ Temp(Val > 45 \ and \ Area = \$x) \\
 & within \ 5min \ from \ Smoke \\
where & Val = Temp.Val
\end{array}
$$

2.4.4 Negazione

Le applicazioni, spesso, necessitano di prendere in considerazione non solo la rilevazione di un evento, ma anche il suo non verificarsi. Ad esempio potremmo voler riconosce la presenza di un incendio dalla presenza di una segnalazione di fumo e da una temperatura maggiore di 45 gradi, provenienti dalla stessa area, in assenza di pioggia. Per venire incontro a queste esigenze TESLA introduce l'operatore *not*, che definisce un intervallo temporale nel quale non deve verificarsi un evento di un certo tipo. Questo intervallo può essere determinato in due modi: usando due eventi come limiti dell'intervallo[3] o usando un evento di riferimento insieme con una durata della finestra temporale. Le seguenti regole mostrano entrambi i casi:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke(Area = \$x) \ and \\
 & each \ Temp(Val > 45 \ and \ Area = \$x) \\
 & within \ 5min \ from \ Smoke \ and \\
 & not \ Rain(Area = \$x) \ between \ Temp \ and \ Smoke \\
where & Val = Temp.Val
\end{array}
$$

[3] questa sintassi è consentita solo quando l'ordine relativo tra i due eventi che definiscono l'intervallo è conosciuto. Questo accade quando gli eventi appartengono alla stessa catena di eventi

$define$	$Fire(Val)$
$from$	$Smoke(Area = \$x)$ and
	$each\ Temp(Val > 45\ and\ Area = \$x)$
	$within\ 5min\ from\ Smoke\ and$
	$not\ Rain(Area = \$x)\ within\ 5min\ from\ Smoke$
$where$	$Val = Temp.Val$

2.4.5 Aggregazione

L'aggregazione ha come scopo quello di applicare una funzione da un insieme di valori S, per generare un nuovo valore v. TESLA consente di utilizzare v, ovunque sia possibile utilizzare un valore; in particolare v può essere assegnato a un attributo dell'evento composito che si vuole generare, o può essere utilizzato come vincolo di selezione degli eventi rilevanti.
TESLA individua i valori da aggregare all'interno di un intervallo temporale e, come per le negazioni, questo può essere definito a partire da due eventi che lo limitano, o da una combinazione di un evento e una durata della finestra temporale. Un esempio dell'uso di questo operatore è il seguente:

$define$	$HighVal(Name, Val)$
$from$	$Stock(Name = \$y, Val = \$x)\ and$
	$last\ Opening()\ within\ 1day\ from\ Stock\ and$
	$\$x > Avg(Stock(Name = \$y).Val)$
	$between\ Opening\ and\ Stock$
$where$	$Val = S.Val,\ Name = S.Name$

Questa regola genera un'evento composito *HighVal*, quando il valore di uno *Stock* supera il valore medio calcolato dall'ultima *Opening*.

2.4.6 Riuso degli eventi

Mentre gli operatori *xxx-within* consentono agli utenti di definire delle policies di selezione, TESLA utilizza la clausola *consuming*, per venire incontro alla

necessità di definire una policy di consumo degli elementi già impiegati nel riconoscimento di una regola. Gli utenti hanno così la possibilità di specificare quali eventi debbano venire invalidati nelle rilevazioni future. Come esempio si consideri la seguente regola:

$$
\begin{array}{ll}
define & Fire(Val) \\
from & Smoke() \textit{ and each } Temp(Val > 45) \\
 & \textit{within } 5min \textit{ from Smoke} \\
where & Val = Temp.Value \\
consuming & Temp
\end{array}
$$

Questa regola consuma tutti gli eventi *Temp* selezionati, in modo tale che una successiva rilevazione dell'evento *Smoke* non generi una notifica *Fire*, fino a che un nuovo evento *Temp* non venga segnalato.

2.4.7 Gerarchia di eventi

TESLA consente di impiegare eventi compositi, definiti in una regola, all'interno della definizione di un'altra regola; in questa maniera è possibile costruire facilmente una gerarchia di eventi.
Questo approccio si adatta al meglio alla natura di molte applicazioni nelle quali le *sources* rendono disponibile un grande volume di informazioni di basso livello, che devono essere filtrati e combinati a diversi livelli d'astrazione. Ad esempio, in un'applicazione di previsione metereologica i sensori generano informazioni riguardo le temperature misurate ed il luogo di misurazione. Come pirmo passo, l'applicazione potrebbe definire delle informazini aggregate sui dati grezzi generati (quale la tempertura media in un certo arco temporale, proveniente da una determinata area). Successivamente, questi eventi potrebbero essere utilizzati per definire patterns che individuino determinate tendenze. Infine, questi andamenti individuati possono essere impiegati, combinati con altre informazioni di alto livello (come la direzione e l'intensità dei venti), per fornire previsioni metereologiche.

2.4.8 Ripetizioni

Tramite la definizione di policies di selezione e consumo degli eventi e di gerarchie tra gli stessi, descritte rispettivamente nelle sezioni 2.4.2, 2.4.6 e 2.4.7, TESLA consente di introdurre l'individuazione di ripetizioni, senza la necessità di nuovi operatori. Ad esempio, se vogliamo individuare ogni ripetizione di un evento A, nel quale l'attributo Val non decresce, per notificare un evento B che contenga il numero di A impiegate nel suo riconoscimento, possiamo farlo, definendo un insieme di regole come il seguente:

$define$	$RepA(Times, Val)$
$from$	$A()$
$where$	$Times = 1\ and\ Val = A.Val$

$define$	$RepA(Times, Val)$
$from$	$A(\$x)\ and\ last\ RepA(Val \leq \$x)\ within\ 3min$
	$from\ A$
$where$	$Times = RepA.Times + 1\ and\ Val = \$x$
$consuming$	$RepA$

$define$	$B(Times)$
$from$	$RepA()$
$where$	$Times = RepA.Times$

2.4.9 Timers

Molte apllicazioni richedono che una determinata regola venga valutata ad intervalli regolari di tempo. TESLA consente di definire regole periodiche, utilizzando un evento di tipo speciale chiamato *Timer*. Ad esempio, se si ha la necessità di valutare una regola esclusivamente alle 9.00 di Venerdì, lo si può fare utilizzando, nella clausola *from*, la seguente selezione:

$$Timer(H = 9, M = 00, D = Friday)$$

.

Capitolo 3

T-REX: Algoritmo

Le regole generate mediante gli operatori presentati nel Capitolo 2.4 definiscono una sequenza di eventi primitivi, imponendo dei vincoli sul loro contenuto e sui tempi della loro acquisizione. In particolare, ogni regola lega l'occorrenza dell'evento composito, che definisce con quella dell'ultimo evento primitivo della sequenza: chiamiamo questo evento *terminatore*.
Lo scopo di un algoritmo di elaborazione è, inizialmente, quello di tradurre la regola in una serie adeguata di strutture dati, atte a descrivere la sequenza di eventi primitivi, legata alla generazione dell'evento composito. Successivamente, nella fase di funzionamento a regime, il suo compito è di analizzare lo storico degli eventi primitivi, verificatisi nel sistema, alla ricerca di una o più occorrenze del pattern definito.
Infine, per ogni serie di eventi individuati nella precedente fase, ha l'obiettivo di generare un nuovo evento, del tipo e con gli attributi appropriati, sulla base di quanto determinato dalla regola stessa.
Nell'implementare questo algoritmo per un *CEP engine*, due sono gli approcci, opposti, che si possono seguire.
Riferendosi al primo, si possono elaborare gli eventi primitivi in maniera incrementale, man mano che si presentano nel sistema, memorizzando di volta in volta i risultati parziali della computazione.
Utilizzando il secondo, si possono memorizzare tutti gli eventi primitivi, postponendo la fase di elaborazione, necessaria al riconoscimento delle sequenze,

finchè un potenziale terminatore non si verifichi nel sistema.
Nello sviluppo di T-Rex sono stati impiegati entrambi gli approcci: inizialmente è stato seguito il primo, tramite un algoritmo che ha preso il nome di *Automa*, dal momento che memorizza i risultati parziali sotto forma di automa a stati finiti, ed implementa un sottoinsieme delle funzionalità di T-Rex [12].
Successivamente è stato impiegato il secondo approccio, tramite un algoritmo che ha preso il nome di *Stacks*; che organizza lo storico degli eventi semplici verificatisi nel sistema sotto forma di stacks. Questo secondo algoritmo presenta un sottoinsieme maggiore di funzionalità e riconosce la maggior parte degli operatori di TESLA presentati nel Capitolo 2.4[1].
Una regola di TESLA definisce un ordinamento parziale tra gli eventi che la compongono. Consideriamo ad esempio la seguente regola:

$$
\begin{array}{ll}
define & CE() \\
from & A(Va > 1) \\
 & and\ each\ B(Vb > 2)\ within\ 2\ min\ from\ A \\
 & and\ each\ C(Vc < 3)\ within\ 4\ min\ from\ A \\
 & and\ each\ D(Vd = 5)\ within\ 4\ min\ from\ B \\
 & and\ D\ within\ 5\ min\ from\ C \\
 & and\ each\ E()\ within\ 3\ min\ from\ B
\end{array}
$$

Il grafo che individua l'ordinamento tra gli eventi (o *grafo d'ordine*) viene mostrato in Figura 3.1, dove una freccia dall'evento e_1 all'evento e_2 rappresenta l'evento e_1, non deve essere notificato dopo l'evento e_2. Questo grafo rappresenta come alcuni eventi siano correlati tra di loro (direttamente o indirettamente), mentre altri non lo siano. Il grafo d'ordine deve essere aciclico, dal momento che non è possibile soddisfare regole che presentano dipenden-

[1]Non viene data la possibilità di definire gerachie di eventi compositi (Capitolo 2.4.7), i timer (Capitolo 2.4.9) e le ripetizioni (Capitolo 2.4.8). Quest'ultime, come precedentemente mostrato, sono ottenibile senza introdurre nuovi costrutti ma utilizzando quelli già presenti.

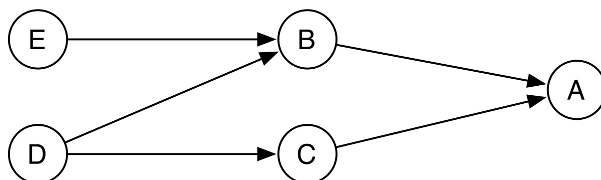


Figura 3.1: Grafo d'ordine della regola precedente

ze temporali di tipo circolare e devono avere una radice, cosicchè una regola dipenda da un unico evento di riferimento: il terminatore.

3.1 Prima implementazione: *Automa*

L'Automa si basa sul concetto di sequenze elementari, sfruttando il paradigna degli automi a stati finiti. Una singola *sequence* risulta essere, in questa implementazione, un sottoalbero costituito da un unico percorso, dalla radice ad una delle foglie.
Gli stati in comune tra più sequenze vengono chiamati *join points*. In primo luogo T-Rex, in questa implementazione, traduce ogni regola definita in ciò che chiamiamo *automaton model*, che è composto da una o più *sequence model*. Inizialmente, ogni sequence model viene instanziata in una *sequence*. Differenti sequenze vengo generate a tempo d'esecuzione, man mano che si presentano nel sistema nuove notifiche di eventi primitivi. Più nello specifico, quando una nuova notifica di evento primitivo viene registrata da T-Rex, possono avvenire differenti cose:

- vengono create nuove sequenze dalla duplicazione di quelle esistenti;
- le sequenze esistenti si muovono da uno stato al successivo;
- le sequenze esistenti vengono eliminate, o perchè giungono ad uno stato finale, il che rappresenta il riconoscimento della regola, o perchè vengono invalidate e non hanno più possibilià di proseguire.

Considerando la seguente regola:

$\underline{RegolaR1}$

$define \quad Fire(area : string,\ measuredTemp : double)$

$from \quad Smoke(area = \$a)$

$and\ each\ Temp(area = \$aandvalue > 45)\ within\ 5\ min$
$from\ Smoke$

$and\ each\ Wind(area = \$aandspeed > 20)\ within\ 5\ min$
$from\ Smoke$

$and\ not\ Rain(area = \$a)\ between\ Smoke\ and\ Wind$

$where \quad area = Smoke.area\ and\ measuredTemp = Temp.value$

La sua elaborazione da parte di T-Rex genera due sequenze di eventi che concorrono nel riconoscimento dell'evento *Fire*. Queste due sequenze condividono almeno un evento[2]: in questo caso l'evento *Smoke*. Inoltre la regola precedente definisce delle relazioni addizionali tra gli eventi:

1. gli eventi *Smoke*, *Wind* e *Temp* devono riferirsi alla stessa area. Questo è descritto utilizzando il parametro $a;

2. l'intervallo temporale che intercorre tra *Wind* e *Smoke* non deve includere la presenza di un evento *Rain*. È dichiarato utilizzando l'operatore not-between;

In generale l'algoritmo, data una regola R, opera come segue: dapprima vengono identificate le sequenze di eventi catturati da R e viene costruito un sequence model per ognuna di esse. Ogni evento della sequenza viene mappato su uno stato nel sequence model e le transizioni tra due stati s_1 e s_2 vengono etichettate con il contenuto ed i vincoli temporali che un evento sopraggiunto deve rispettare per provocare la transizone. I vari sequence models e le relazioni tra di loro, ovvero i join points, vengono poi memorizzati in un unico automation model. Nello specifico, oltre a collegare gli stati

[2]Questo è vero per ogni regola dal momento che tutte le sequenze sono costruite a partire dal *terminatore*.

condivisi da due o più sequence models, viene tenuta traccia dei parametri e delle negazioni presenti nella regola (che sono ortogonali alle sequenze). La Figura 3.2 mostra i due sequence models $M1$ e $M2$, che derivano dalla regola *R1* e nell'area tratteggiata i join points. Per descrivere come l'algoritmo

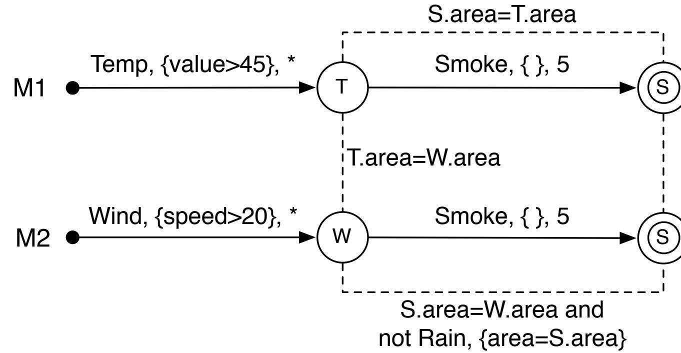


Figura 3.2: Automa per la regola *R1*

elabora gli eventi che si presentano nel sistema, prendiamo in esame la singola sequenza $M1$, descritta dalla regola:

$\underline{RegolaR2}$

$define$	$Fire(area : string,\ measuredTemp : double)$
$from$	$Smoke(area = \$a)$
	$and\ each\ Temp(area = \$a\ and\ value > 45)\ within\ 5\ min$
	$from\ Smoke$
$where$	$area = Smoke.area\ and\ measuredTemp = Temp.value$

La Figura 3.3 mostra una possibile evoluzione dell'automa, corrispondente alla sequenza $M1$. Quando si presenta nel sistema un nuovo evento e l'algoritmo agisce nel seguente modo: (i) controlla che il tipo, il valore degli attributi e il timestamp di e soddisfi una transizione di stato per una delle sequenze esistenti; in caso negativo l'evento viene immediatamente scartato. Altrimenti, se una sequenza Seq nello stato s_1 può impiegare e, per raggiungere lo stato s_2, (ii) crea una copia S' di Seq e, (iii) usa l'evento e, per

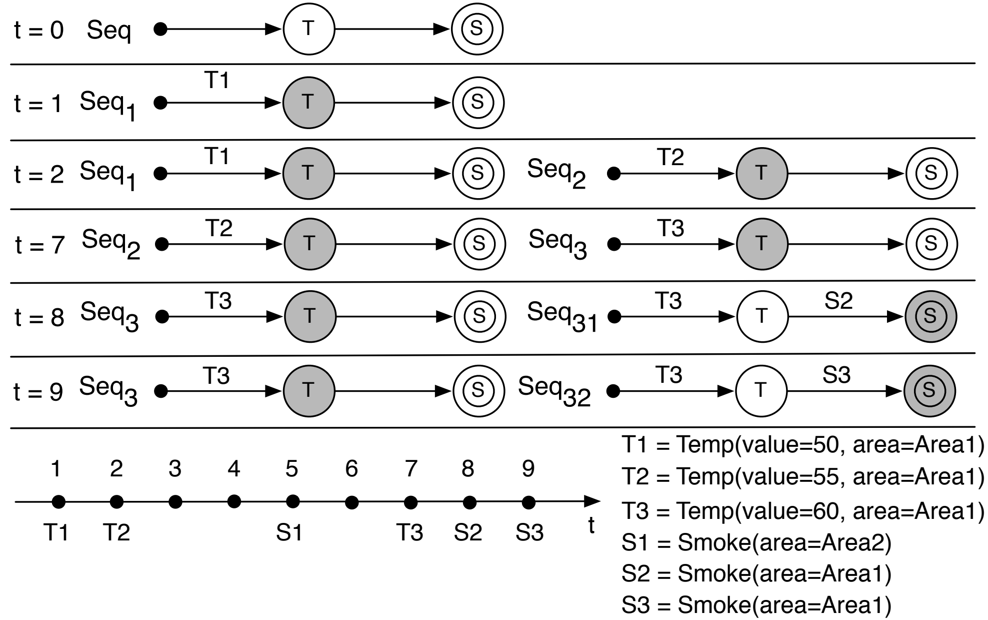


Figura 3.3: Un esempio di elaborazione per la regola *R2*

muovere S' nello stato s_2. La sequenza oiginale *Seq* rimane nello stato s_1, in attesa di eventi successivi. Le sequenze vengono rimosse quando risulta impossibile procedere nello stato successivo, a causa di un vincolo temporale scaduto, o nel caso si presenti l'occorrenza di un evento espresso come negazione nella regola. Nell'esempio, all'istante iniziale $t = 0$ una sola sequenza *Seq* del modello $M1$ è presente nel sistema, in attesa nel suo stato iniziale. Al tempo $t = 1$ un evento di tipo *Temp* ($T1$) i cui parametri, così come il suo timestamp, rispettano i vincoli della transizione che porta allo stato T. Viene dupilcato l'automa, creando la sequenza Seq_1 che avanza nello stato T. Allo stesso modo, al tempo $t = 2$, l'arrivo dell'evento $T2$ di tipo *Temp*, genera una nuova sequenza Seq_2, a partire da *Seq*, ed effettua una mossa che la porta nello stato T. Al tempo $t = 5$ si presenta nel sistema un evento *Smoke*, ma non rispetta i vincoli sul parametro *Area* e viene scartato. Successivamente, al tempo $t = 7$, la Seq_1 viene eliminata, dal momento che non vi è la possibilità che un successivo evento *Smoke* S_1 rispetti i vincoli temporali. Inoltre il nuovo evento verificatosi, $T3$, genera come in precedenza una nuova sequenza Seq_3, come duplicato di *Seq*. Al tempo $t = 8$, la

Seq_2 viene rimossa a causa dei vincoli temporali, mentre l'arrivo dell'evento S_2 di tipo *Smoke*, proveniente dall'*Area* corretta, duplica la sequenza Seq_3, generando la sequenza $Seq_3 1$ e la porta in un suo stato finale S. Questo implica il riconoscimento di una sequenza valida, composta dalla successione degli eventi $T3$ e $S2$. Dopo il riconoscimento, la sequenza $Seq_3 1$ può essere rimossa dal sistema. Alla stessa maniera, l'arrivo al tempo $t = 9$ dell'evento $S3$, causa la creazione della sequenza $Seq_3 2$ ed il riconoscimento della sequenza valida composta da $T3$ e $S3$.
La presenza di una negazione che vincoli l'evento e a non occorrere nell'intervallo i viene gestita in maniera differente a seconda che l'intervallo temporale sia espresso utilizzado due eventi (e_1 e e_2) o tramite un evento ed una finestra temporale (e_3 e t). Nel primo caso, quando si presenta un evento e, tutte le sequenze che hanno già raggiunto lo stato associato con l'evento e_1, ma non ancora quello associato all'evento e_2, vengono eliminate; nel secondo caso, invece, tutti gli eventi e vengono memorizzati. Quando la sequenza raggiunge lo stato associato all'evento e_3, viene controllato che non sia stato memorizzato alcun evento e entro t secondi. Se è così la sequenza può procedere, altrimenti viene eliminata.
Normalmente una regola TESLA cattura eventi generati da più sequenze. Quando una sequenza S, istanza del sequence model M, può compiere una mossa che la porti nello stato X, condiviso con i modelli $M_1, \ldots, M_n$, tramite l'impiego dell'evento e, l'algoritmo avanza S allo stato X solo se e causa la transizione allo stato X per almeno una sequenza di ogni modello $M_1, \ldots, M_n$; in caso contrario S viene eliminata. Controlli analoghi vengono effettuati per verificare relazioni tra differenti sequenze, come quelli risultanti dall'uso dei parametri.
Altre caratteristiche che vengono prese in considerazione da questo algoritmo sono gli operatori che definiscono policy di consumo e di selezione. La prima di queste due caratteristiche non presenta difficoltà nell'essere implementata tramite l'Automa: quando una sequenza S di un regola R giunge ad uno stato finale usando e consumando un evento e, possiamo rimuovere tutte le altre sequenze della regola R che fanno uso dell'evento e e non sono ancora giunte ad uno stato finale. La seconda caratteristica invece, presenta più difficoltà:

duplicando le sequenze ad ogni transizione di stato vengono catturate tutte le possibili sequenze di eventi, che soddisfano i vincoli di una regola; viene perciò implementata una politica di selezione di tipo *eache-within*. Per quanto riguarda gli operatori *first-within* e *last-within* l'unica implementazione possibile è quella di ritardare la decisione circa la validità della sequenza, al raggiungimento del loro stato finale[3]. Si consideri ad esempio la regola :

$\underline{RegolaR3}$

$define$	$Fire(area : string,\ measuredTemp : double)$
$from$	$Smoke(area = \$a)$
	$and\ last\ Temp(area = \$aandvalue > 45)\ within\ 5\ min$
	$from\ Smoke$
$where$	$area = Smoke.area\ and\ measuredTemp = Temp.value$
$consuming$	$Temp$

Se supponiamo che vengano ricevuti due eventi di tipo Temp: $T1$ e $T2$ in quest'ordine, quando arriva $T2$ non possiamo scartare $T1$, poichè l'arrivo di un evento *Smoke* potrebbe generare un evento composito e portare all'eliminazione dell'evento $T2$ impiegato per il riconoscimento, rendendo di fatto $T1$ l'ultimo evento di tipo *Temp* notificato e, perciò, candidato valido a generare un'altro evento composito *Fire*.
La Figura 3.4 mostra l'architettura del sistema implementato tramite questo algoritmo. Le regole vengono tradotte dal *Rule Manager* in una serie di automation models e viene costruito un indice, per indirizzare gli eventi entranti nel sistema verso i modelli che ne aspettano la notifica. Da questi automation models vengono poi generate le varie sequenze, sulla base dell'ordine d'arrivo degli eventi. Gli eventi compositi vengono infine generati ed indirizzati ai sottoscrittori, sulla base dell'elaborazione dei join points e delle sottoscrizini ricevute.

[3]Sono possibili ottimizzazioni specifiche per la singola regola (ad esempio se questa non contiene alcuna clausola *consuming*).

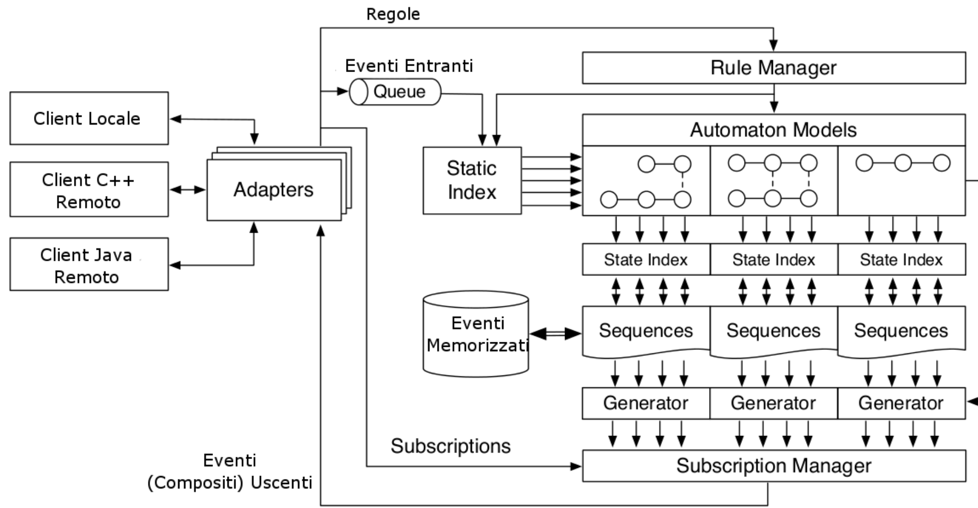


Figura 3.4: Architettura del sistema implementato tramite Automa

Difetti riscontrati

Questa implementazione, sebbene diminuisca potenzialmente la quantità di elaborazioni da compiere al riconoscimento del terminatore (rendendo il sistema più reattivo), porta ad un maggiore consumo di memoria, necessaria a memorizzare i risultati parziali. Questo problema, che potrebbe non essere determinante in un sistema reale, a seconda delle caratteristiche dei nodi che compongono la rete, risulta, invece, essere considerevole in un ambiente di simulazione ed, in particolare, in presenza di molte policy di selezione di tipo *each-within*. In quest'ultima situazione, infatti, tutti i nodi della rete vengono instanziati su una sola macchina e per ogni nodo, per ogni regola, all'arrivo di un evento consono, l'automa vine duplicato, come visto. Nel caso di operatori *each-within*, ogni successivo evento riconosciuto nello stesso stato, genera una dupliclazione dell'automa, portando così ad un più rapido esaurimento delle risorse disponibili. Per questo motivo si è deciso di sviluppare un algoritmo che seguisse un differente metodo di elaborazione, con l'obiettivo di poter simulare reti di nodi di dimensioni maggiori, rispetto a quelle che l'*Automa* permette, indipendentemente dalle policy di selezione adottate nella definizione delle regole.

3.2 Seconda implementazione: ***Stacks***

Mentre l'algoritmo ad automi elabora le regole in maniera incrementale, con l'arrivo degli eventi nell'*engine*, quello facente uso degli stacks utilizza un metodo diametralmente opposto: memorizza tutti gli eventi ricevuti, fino alla ricezione di un potenziale terminatore, ritardando fino a questo momento la fase di generazione dei risultati. Questa seconda implementazione utilizza il modello degli stacks per rappresentare i nodi del grafo d'ordine. Ogni evento facente parte della definizione della regola viene, quindi, tradotto da una pila e da una serie di informazioni che permettano di rappresentare le relazioni che intercorrono tra i nodi del grafo: i successivi stack, che sono a questo direttamente connessi, seguono il percorso che va dalla radice verso le foglie, la finestra temporale riferita allo stack che lo precedono, i vincoli sui valori degli attributi, la tipologia della pila (occorrenza, negazione o aggregazione di eventi) e le policy di selezione e consumo. Vengono inoltre memorizzate informazioni riguardo i parametri definiti dalla regola.
Per mostrare come avviene la traduzione della regola ed il riconoscimento di un evento composito, prendiamo in considerazione la seguente regola *R4*:

$\underline{Regola R4}$

$define \quad ComplexEvent()$

$from \quad D(p = \$x)$

$and\ each\ B(p = \$x\ and\ v > 45)\ within\ 5\ min\ from\ D$

$and\ each\ A(p = \$x)\ within\ 3\ min\ from\ B$

$and\ last\ C()\ within\ 8\ min\ from\ D$

La regola *R4* è composta da una sequenza di eventi primitivi **A**,**B**,**C** e **D**, di cui l'ultimo è terminatore della stessa (ovvero radice del grafo che la descrive). L'algoritmo crea quattro pile per memorizzare la regola, come mostrato in Figura 3.5. Ogni pila è etichettata con il tipo degli eventi primitivi che accetta, con l'insieme dei vincoli sugli attributi tra graffe e il livelo logico dello stack tra quadre. Viene inoltre rappresentato graficamente, mediante una

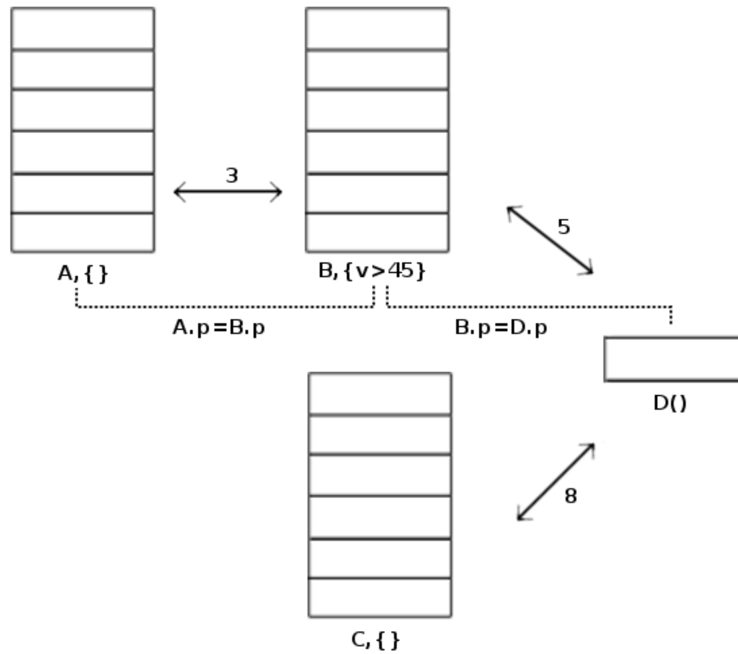


Figura 3.5: Stacks per la regola R4

freccia a due estremità, la finestra temporale che intercorre tra due eventi primitivi[4] e, tramite una linea tratteggiata, i vincoli addizionali di tipo parametrico. Quando un nuovo evento e giunge nel sistema, viene elaborato; viene inizialmente controllato se il suo tipo ed i valori delle variabili corrispondano a quelli di una o più pile della regola. Se questo non si verifica, viene immediatamente scartato, altrimenti viene inserito nello stack appropriato e l'ordinamento viene mantenuto sulla base del timestamp posseduto dall'evento. Nel caso gli stack interessati siano differenti dall'ultimo (nell'esempio quello etichettato con l'evento **D**), ovvero quello rappresentante un terminatore, non vengono effettuate successive elaborazioni. Se, invece, e risulta essere un evento valido per l'ultimo stack di livello l (S_l)[5], viene avviata un'analisi degli eventi presenti negli stacks, per estrarre tutte le sequenze di eventi, valide per la regola. Più nello specifico, l'algorimo funziona in due fasi. La prima

[4]Questa informazione viene associata, nell'implementazione dell'algoritmo, alla pila più vicina alle foglie dell'albero della regola.

[5]Per questo stack è necessaria una sola posizione dal momento che il terminatore genera eventi compositi e viene successivamente consumato.

è dedicata alla rimozione degli eventi che hanno superato i vincoli temporali imposti dalla regola ed opera come segue:

- il timestamp dell'evento e viene utilizzato per individuare l'indice i del primo elemento valido per ogni stack direttamente connesso a S_l $(S_{l-1,1}, \ldots, S_{l-1,n})$, utilizzando l'informazione del vincolo temporale;

- tutti gli eventi negli stacks $S_{l-1,1}, \ldots, S_{l-1,n}$ aventi indice $i' < i$ vengono rimossi dallo stack, dal momento che non hanno possibilità di appartenere ad una finestra temporale valida per un successivo evento e;

- il procedimento viene ripetuto lungo l'albero, per ogni pila, considerando il timestamp del primo evento rimasto nello stack $S_{x,j}$ per determinare quali eventi eliminare dalle pile $S_{x-1,1}, \ldots, S_{x-1,k}$.

La seconda si occupa dell'effettiva individuazione delle sequenze corrette e della generazione dell'evento composito corrispondente:

- e viene combinato con tutte le permutazioni degli eventi memorizzati negli stack di livello S_{l-1} a questo direttamente connessi, ovvero le pile $S_{l-1,1}, \ldots, S_{l-1,n}$ che soddisfano i vincoli, creando una sequenza parziale di $n + 1$ eventi;

- ogni sequenza parziale viene successivamente impiegata per selezionare gli elementi validi dagli stack di livello S_{l-2}, direttamente connessi a $S_{l-1,1}, \ldots, S_{l-1,n}$, in maniera ricorsiva, finchè non vengono raggiunte le foglie dell'albero, generando zero, uno o più sequenze valide, composte da un evento per ogni stack presente nella regola;

- viene generato un evento composito per ogni sequenza valida individuata.

Per meglio comprendere come questo processo funzioni, si prenda in considerazione la Figura 3.6: gli eventi memorizzati in ogni pila sono rappresentati con il loro tipo, il loro valore per l'attributo p, ed il loro timestamp. Quando l'evento $D(p = 3)@20$ entra nel sistema, essendo un evento che interessa

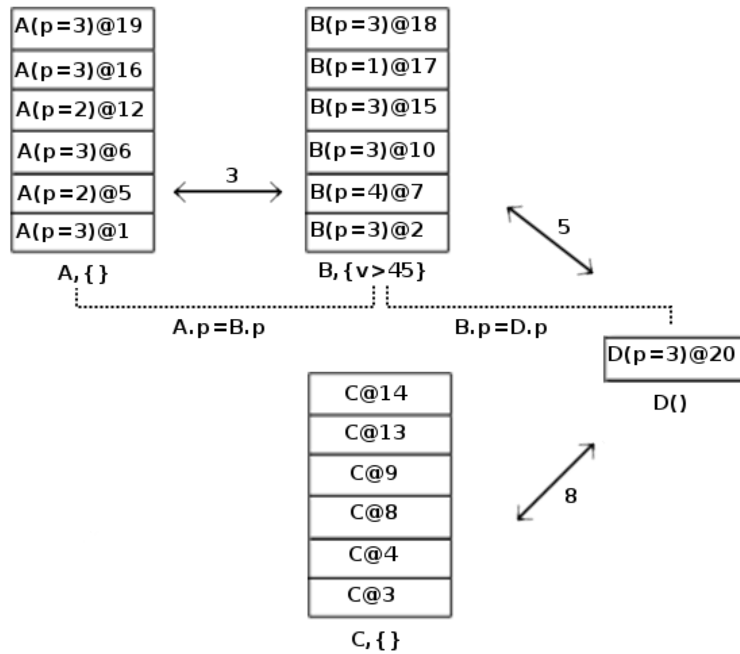


Figura 3.6: Un esempio di elaborazione utilizzando gli Stacks

l'ultimo stack e ne rispetta i vincoli, avviene l'elaborazione. L'evento viene inserito nella pila corrispondente, dopo di che, il suo timestamp (*20*) viene impiegato per rimuovere dalle pile, a lui direttamente connesse (ovvero la pila B e C), gli eventi troppo datati. Lo stack B ha come finestra temporale 5, mentre lo stack C, 8. Come conseguenza, dalla prima vengono rimossi gli elementi che posseggono un timestamp inferiore a 15 (nell'esempio i primi tre eventi a partire dal fondo), mentre dalla seconda quelli con timestamp inferiore a 12 (ovvero i primi quattro eventi a partire dal fondo). Successivamente, per la pila B che possiede altri stack di livello inferiore a questa direttamente connessi, viene considerato il timestamp del primo elemento (il più vecchio), ovvero 15. Sulla base di quest'ultimo, vengono rimossi dallo stack A gli elementi con timestamp minore di 12, essendo la finestra temporale indicata tra gli stack B ed A, di 3 unità. Dalla prima pila vengono dunque rimossi i primi tre elementi, a partire dal fondo della stessa.
Tutti gli eventi che si trovano nella pila vengono allora considerati per individuare sequenze valide. A partire dall'elemento terminatore ($D(p = 3)@20$),

unico risultato parziale individuato al momento, viene selezionata una porzione dello stack B su base temporale e dei parametri[6]. Nel caso mostrato, ad esempio, vengono selezionati: $B(p = 3)@18$ e $B(p = 3)@15$; $B(p = 1)@17$ viene scartato dal momento che rispetta i vincoli temporali, ma non quelli dei parametri. A partire dagli eventi individuati vengono costruiti i risultati parziali:

$$\langle D(p = 3)@20, B(p = 3)@18\rangle$$

$$\langle D(p = 3)@20, B(p = 3)@15\rangle$$

Successivamente, viene effettuata la stessa operazione sullo stack C, per ogni risultato parziale individuato finora. Per definire la finestra temporale corretta, viene utilizzato l'evento dello stack padre nella regola (in questo caso D) presente nei risultati parziali. Dell'evento parziale $\langle D(p = 3)@20, B(p = 3)@18\rangle$ viene impiegato l'evento $D(p = 3)@20$ e, tramite quest'ultimo, individuata la finestra di eventi valida $\langle C@14\rangle, \langle C@13\rangle$. La loro combinazione porta alla generazione dei seguenti risultati parziali:

$$\langle D(p = 3)@20, B(p = 3)@18, C@14\rangle$$

$$\langle D(p = 3)@20, B(p = 3)@18, C@13\rangle$$

Alla stessa maniera, dall'evento parziale $\langle D(p = 3)@20, B(p = 3)@15\rangle$, avente anch'esso come elemento di riferimento per lo stack C l'evento $D(p = 3)@20$, vengono individuati i risultati parziali

$$\langle D(p = 3)@20, B(p = 3)@15, C@14\rangle$$

$$\langle D(p = 3)@20, B(p = 3)@15, C@13\rangle$$

A partire dai quattro risultati elencati precedentemente, la stessa operazione viene applicata allo stack A; in questo caso sono i timestamps degli eventi B che sono presi in considerazione per filtrare quelli memorizzati in questo

[6]Eventi che posseggono valori di attributi che non rispettano i vincoli (ad esempio $v <= 45$) non vengono nemmeno inseriti nello stack dal momento che non potranno mai far parte di una sequenza valida.

stack. In particolare per entrambi i risultati $\langle D(p = 3)@20, B(p = 3)@18, C@14\rangle$, $\langle D(p = 3)@20, B(p = 3)@18, C@13\rangle$, contenenti $B(p = 3)@18$, risulterà valido solo l'evento $A(p = 3)@16$ generando

$$\langle D(p = 3)@20, B(p = 3)@18, C@14, A(p = 3)@16\rangle$$

$$\langle D(p = 3)@20, B(p = 3)@18, C@13, A(p = 3)@16\rangle$$

mentre $\langle D(p = 3)@20, B(p = 3)@15, C@14\rangle$, $\langle D(p = 3)@20, B(p = 3)@15, C@13\rangle$ sulla base di $B(p = 3)@15$ non indidueranno alcun evento di tipo A ammissibile[7] e verranno perciò scartati. Avendo ispezionato tutti gli stack, dal momento che quest'ultima operazione ha restituito un insieme vuoto di risultati, le sequenze valide per la regola sono:

$$\langle D(p = 3)@20, B(p = 3)@18, C@14, A(p = 3)@16\rangle$$

$$\langle D(p = 3)@20, B(p = 3)@18, C@13, A(p = 3)@16\rangle$$

Essendo complete, viene generato un evento complesso del tipo appropriato per ognuna di queste.
In questa maniera viene implementato l'operatore che definisce l'occorrenza di un evento. Nel seguito vengono descritti gli altri operatori implementati, mediante l'algoritmo che fa uso degli stacks.
La Figura 3.7 mostra l'architettura del sistema implementato tramite questo algoritmo. Le regole vengono tradotte dal *Rule Manager* in un'insieme di stacks e viene costruito un indice per indirizzare gli eventi entranti nel sistema verso gli stack, che ne aspettano la notifica. Da questi vengono individuate le varie sequenze valide, sulla base degli elementi presenti nelle pile. Gli eventi compositi generati sono inviati ai sottoscrittori, sulla base delle sottoscrizini ricevute, come descritto dal successivo Capitolo.

[7] $A(p = 2)@12$ non può essere selezionato dal momento che, pur trovandosi in una finestra temporale valida, possiede un valore dell'attributo p non valido.

Figura 3.7: Architettura del sistema implementato tramite Stacks

3.2.1 Window e policy di selezione e consumo

Come abbiamo visto, il processo di selezione delle sequenze valide si basa sull'individuazione, dalla radice alle foglie, di sottoinsiemi di eventi validi. Questi sottoinsiemi (chiamati *windows*) permettono di implementare le differenti policy di selezione e di consumo, che TESLA mette a disposizione. Per quanto riguarda le prime, si possono distinguere tre casi:

***each-within*:** vengono selezionati ed elaborati, a partire da un evento di livello x, tutti gli elementi facenti parte della window, individuata sullo stack di livello $x-1$ che si sta analizzando (così come è stato mostrato nell'esempio precedente);

***last-within*:** viene preso in considerazione dall'elemento di livello x, nel processo iterativo di elaborazione delle sequenze valide, solamente l'ultimo elemento valido appartenente alla windows dello stack di livello $x-1$ che si sta analizzando;

***first-within*:** viene preso in considerazione dall'elemento di livello x, nel processo iterativo di elaborazione delle sequenze valide, solamente il primo elemento valido appartenente alla windows dello stack di livello $x-1$ che si sta analizzando;

Per quanto riguarda le policy di consumo degli eventi, una volta individuate le sequenze valide che generano degli eventi compositi, gli eventi primitivi che le compongongono vengono rimossi dalle rispettive pile, che sono mantenute ordinate sulla base dei timestamps.

3.2.2 Parametrizzazione

Si è già visto che i parametri, che appartengono a stacks che fanno parte dello stesso percorso, che va dalla radice ad una foglia del grafo d'ordine, vengono valutati direttamente nella fase di selezione delle windows. Parametri che intercorrono fra stack che si trovano su percorsi differenti, invece, vengono valutati una volta ottenute le sequenze complete. Avviene dunque una fase di filtraggio dei risultati, nella quale le sequenze che non rispettano i vincoli su questi parametri (detti parametri esterni) vengono scartate. I parametri che fanno parte della regola vengono quindi partizionati in due insiemi all'atto della crezione degli stacks:

parametri interni: si riferiscono a stacks appartenenti allo stesso percorso radice-foglia. Questi vengono memorizzati come caratteristica dello stack più vicino alla foglia tra i due e contengono l'informazione circa quello a cui si riferiscono. Così facendo, è possibile valutarlo in fase di selezione degli elementi della window.

parametri esterni: si riferiscono a stacks appartenenti a differenti percorsi radice-foglia. Risultano essere indipendenti dagli stack a cui si riferiscono e vengono valutati, sequenza per sequenza, al termine dell'algoritmo di elaborazione.

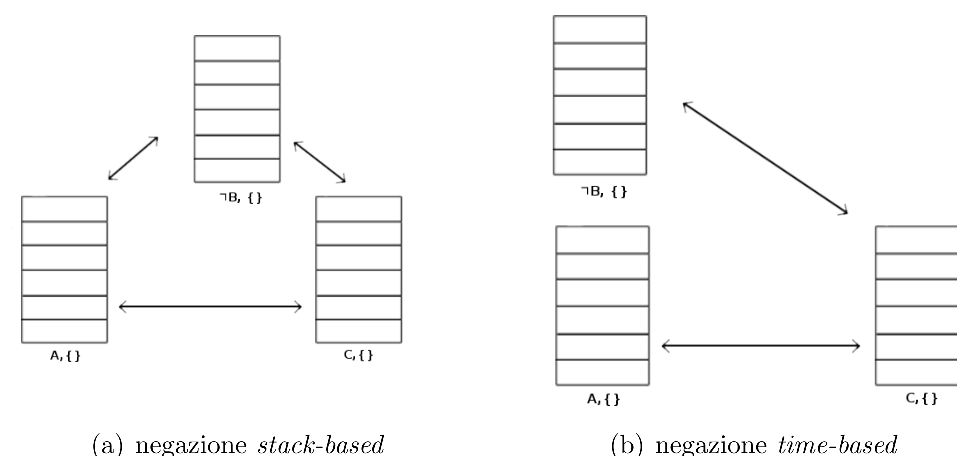
Questa operazione di suddivisione dei parametri in due suttoinsiemi non risulta essere necessaria ai fini della correttezza dei risultati. I parametri che vincolano gli attributi di due eventi potrebbero essere, infatti, valutati nella loro totalità al termine dell'individuazione delle sequenze valide, come avviene per i parametri esterni, senza alterare il numero e la composizione degli eventi compositi rilevati. Questa ottimizzazione viene introdotta allo scopo di rendere più efficiente il processo di filtraggio, soprattutto nel caso di regole contenenti politiche di selezione each-within. Questa policy, infatti, aumenta in maniera esponenziale il numero di eventi primitivi, memorizzati nella pila, che devono essere considerati dall'algoritmo condizionandone direttamente le performances. Filtrarli sulla base dei parametri, durante la fase di individuazione delle finestre temporali, attenua il problema riducendone

il numero, ma risulta possibile esclusivamente per eventi appartenenti allo stesso percorso radice-foglia.

3.2.3 Negazione

Come descritto nel Capitolo 2.4.4 è possibile specificare in una regola due differenti tipologie di negazioni: negazioni *stack-based* e *time-based*. Le prime dichiarano una window sulla base di due altri eventi; in questo caso i due eventi presi in considerazione devono appartenere allo stesso percorso radice-foglia. Le seconde sono descritte a partire da un evento e un valore indicante la dimensione temporale della window.

La Figura 3.8 mostra come vengono implementate, dall'agloritmo degli



(a) negazione *stack-based* (b) negazione *time-based*

Figura 3.8: Tipologie di negazioni

stacks, entrambe le tipologie. Nel primo caso (Figura 3.8(a)) la pila contenente gli eventi **B**, che non si vuole si verifichino nel tempo che intercorre tra un evento **C** ed uno **A** individuati come validi, viene valutata per ogni elemento individuato nello stack **A**, a partire da ogni evento di **C**. Solo a questo livello, infatti, si hanno le informazioni sull' *upperbound* ed il *lowerbound* della finestra temporale che si vuole individuare su **B**. Utilizzando, infatti, il timestamp dell'elemento **A** come *lowerbound* e quello dell'elemento **C**, che lo ha selezionato, come *upperbound*, si ottiene un range temporale. Se in questo periodo la pila **B** ha ricevuto degli eventi, l'intera sequenza parziale

viene scartata. Nel secondo caso (Figura 3.8(b)), invece, la pila di livello x contenente la negazione si comporta come una qualsiasi pila. A partire da ogni elemento della pila **C** di livello $x + 1$ si valuta se lo stack **B** contiene degli eventi appartenenti alla window definita; in caso affermativo l'intero risultato parziale viene eliminato.
La verifica delle negazioni viene effettuata a livello x, ad ogni passo dell'algoritmo ricorsivo, prima di valutare i rami di livello $x-1$, in quanto la presenza di eventi nella finestra temporale descritta dalla regola, porta alla rimozione dell'intera sequenza parziale. Anticiparla evita, quindi, di generare sequenze parziali che non potranno mai essere parte di una sequenza valida.

3.2.4 Aggregazione

Così come l'operatore di negazione, anche quello di aggregazione può presentare una finestra temporale *stack-based* o *time-based*. Anche in questo caso, le pile che rappresentano queste tipologie di eventi, vengono gestite come in quello delle negazioni. A differenza di quest'ultime, però, a partire dal sottoinsieme di eventi individuato dalla finestra temporale, viene generato un unico valore, che non è impiegato per filtrare le sequenze valide da quelle non valide. Per questo motivo questa elaborazione viene effettuata in una fase conclusiva dell'algoritmo, quando le sequenze valide sono già state correttamente individuate.

Capitolo 4

T-REX: Distribuzione

Con l'obbiettivo di testare quali vantaggi porti l'impiego di un sistema di riconoscimento completamente distribuito, è stato deciso di adottare tale approccio per strutturare il middleware. In particolare bisogna considerare il problema di individuare i principali requisiti degli scenari applicativi e, di conseguenza, i parametri che permettono di misurarne le prestazione. A seconda dello specifico scenario appicativo, infatti, può essere necessario incentrare l'ottimizzazione del sistema sulla riduzione della latenza, sulla diminuzione del traffico di rete o sulla distribuzione del carico tra i vari nodi. Tra questi parametri, però, la latenza ed il traffico di rete non sono completamente sovrapposti o, al contrario, in contrapposizione, restando strettamente connessi, mentre non è stata individuata una chiara relazione con i tempi di elaborazione, che ci proponiamo di misurare e analizzare. É stato, dunque, deciso di focalizzare la nostra analisi sui due aspetti, fondamentali nelle prestazioni di un sistema reale, che presentano un comportamento congiunto: minimizzare il traffico di rete prodotto da T-Rex durante il suo funzionamento a regime e ridurre il ritardo introdotto per riconoscere una sequenza valida.
Il primo aspetto è stato affrontato inviando verso nodi, esclusivamente gli eventi primitivi che interessano la sequenza da riconoscere. A questo scopo è stato ideato un protocollo di tipo publish-subscribe, che definisce l'interazione tra i vari nodi che cooperano nella rete, in maniera tale da poterli

discriminare, durante la fase di installazione di una regola, sulla base della loro posizione e degli eventi che vedranno transitare.
É stato inoltre distribuito l'algoritmo di rinoscimento tramite le informazioni ottenute dalla distribuzione del protocollo di comunicazione. Questo ha consentito di ottenere un duplice vantaggio: se da un lato ha reso possibile un effettivo filtraggio degli eventi lungo la gerarchia della rete, ottenendo una riduzione del numero di messaggi in transito, dall'altro, per come è stato studiato, consente di effettuare le operazioni necessarie al riconoscimento nel primo nodo, che possiede tutte le informazioni indispensabili, partizionando lungo la rete anche l'elaborazione. Viene perciò preso in esame ed affrontato anche il secondo aspetto che si è discusso precedentemente: elaborare le sequenze in maniera distribuita senza gravare pochi nodi di tutto il lavoro, ma permettere che siano più nodi con capacità computazionali minori a cooperare, al fine di portare vantaggi dal punto di vista del tempo di elaborazione, necessario all'attuazione dell'algoritmo.
Mediante la combinazione di questi accorgimenti otteniamo così un sistema di distribuzione del middleware che va oltre il semplice *operator placement* discusso per i sistemi DSP nel Capitolo 1.1. Se da un lato infatti, la distribuzione dell'algoritmo consente di effettuare le operazioni di filtraggio degli eventi appartenenti a sotto-regole (partizioni della regola definita tramite TESLA ed installata nel nodo radice della rete), ottenendo l'equivalente di un *operator placement* ottimo, dall'altro la natura stessa dei sistemi CEP e l'algoritmo di comunicazione pubblish-subscribe sviluppato, su cui si basa T-Rex, consente di anullare il transito di informazioni non necessarie al riconoscimento e di non duplicare le informazioni condivise da differenti regole.
Con l'intento di eliminare l'invio di ogni informazione non necessaria è stato pensato, inoltre, un differente sistema di indirizzamento degli eventi validi. Oltre al sistema ad albero singolo, è stata prevista un'altra tipologia di scomposizione della rete ad albero multiplo. Nel primo caso, a partire dai nodi della rete, viene definita un'*overlay network* con l'individuazione di una radice ed un albero di nodi, lungo il quale gli eventi risalgono; una volta raggiunta la radice, si ha la certezza che tutti gli eventi compositi siano stati riconosciuti, e da qui vengono rispediti verso tutti i sottoscrittori che

ne hanno fatto richiesta. Nel secondo caso invece, viene stabilita un'overlay network per ogni nodo della rete, che abbia come radice il nodo stesso, e viene impiegata quella del stottoscrittore dell'evento, per individuare le sequenze valide. Non è più necessario, una volta riconosciuto l'evento composito, ritrasmetterlo lungo rete, dal momento che una volta raggiunta la radice, ha raggiunto il sottoscrittore che ne richiede la notifica.

4.1 Distribuzione del procollo

Il protocollo di comunicazione che è stato sviluppato per T-Rex si pone l'obiettivo, come affermato in precedenza, di indirizzare gli eventi individuati in un determinato nodo della rete, come determinanti nel riconoscimento di una sottoregola $r \in R$, verso tutti e soli i nodi che hanno interesse nei confronti dalle regola R. Al fine di ottenere il suo scopo, mediante un paradigma di tipo pubblish-subscribe, gli eventi vengono rappresentati mediante i messaggi inviati tra nodi e tengono traccia delle informazioni necessarie al riconoscimento di una sequenza valida.
Il protocollo si occupa inizialmente di collezionare informazioni riguardo la topologia della rete sottostante e calcolare lo *shortest path tree (SPT)*. Questa operazione viene effettuata per il solo nodo con *ID* minore, nel caso della soluzione ad albero singolo mentre l'*SPT* viene calcolato a partire da ogni nodo della rete nel caso di una soluzione ad albero multiplo. Sulla base delle informazioni collezionate, vengono memorizzati gli *ID* del proprio nodo padre e dei propri nodi figli per ogni albero. Successivamente, l'operazione compiuta dal protocollo è quella di collezionare gli *advertisements* da parte dei nodi, direttamente connessi ai generatori degli eventi primitivi. Gli advertisements rappresentano la dichiarazione, da parte di ogni nodo della rete, di essere direttamente connesso ad un generatore di eventi e riporta, inoltre, anche il riferimento al tipo di evento che verrà generato. In seguito, sulla base degli advertisements, avviene la scomposizione delle regole in sottoregole da parte del *rule deployer* ed il loro invio verso i nodi che ricevono effettivamente gli eventi che le compongono. Questa fase è il cardine della distribuzione dell'algoritmo di riconoscimento e verrà dunque trattata nel

successivo capitolo. Vengono inoltre inoltrati lungo la rete dei messaggi di sottoscrizione che rappresentano l'interesse da parte di un determinato nodo ad un evento composito, il cui identificativo è presente nel pacchetto di sottoscrizione. Nel caso di protocollo ad albero singolo, le sottoscrizioni locali vengono inviate, così come avviene per gli advertisements, a tutti i nodi della rete, in maniera tale da costruire per ogni nodo una tabella di routing, che consenta di indirizzare verso il nodo corretto gli eventi compositi riconosciuti. Per quanto riguarda il caso ad albero multiplo invece, le sottoscrizioni vengono tradotte nella definizione della regola relativa e sostituiscono la fase di installazione delle stesse; ogni sottoscrizione rappresenta e viene gestita come l'installazione di una regola che ha come radice il sottoscrittore stesso. Gli eventi primitivi, rappresentati dalle pubblicazioni, attraversano la rete dalle foglie verso la radice, subendo un filtraggio che propaga esclusivamente quelli validi[1]. Vengono perciò inseriti nel *payload* del pacchetto di rete gli attributi dell'elemento applicativo che rappresenta un evento primitivo. Il payload contiene quindi informazioni circa la tipologia dell'evento che rappresenta, il nome ed il valore degli attributi a lui associati, un contatore di riferimenti al pacchetto[2] ed il timestamp, rappresentante l'istante di arrivo dell'evento. Quest'ultimo inoltre, insieme con le informazioni riguardo il mittente e l'insieme degli alberi interessati, vengono impiegati come campi di attributi del pacchetto di rete.

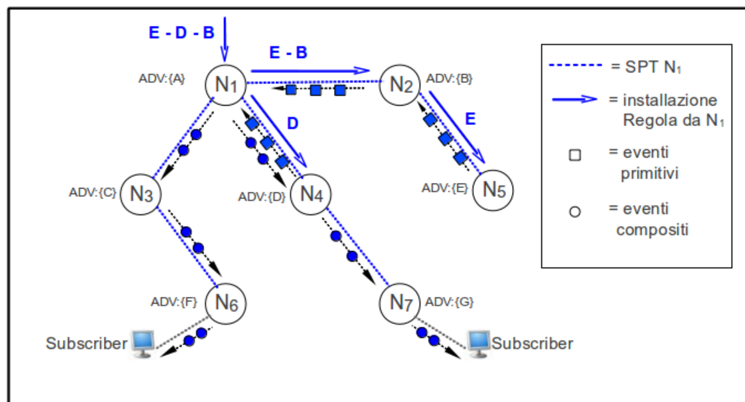
Per meglio descrivere le differenze tra le due tipologie di protocollo viene confrontata, in Figura 4.1, l'installazione di una regola, l'instradamento dei pacchetti generati che la compongono e la successiva notifica ai sottoscrittori, degli eventi compositi individuati. In entrambi i casi i nodi sono connessi in una rete fisica totalmente connessa, non mostrata in figura per comodità, ed i loro advertisements sono esplicitati[3]. I sottoscrittori sono connessi ai nodi N_6 e N_7 e la regola per la quale notificano il loro interesse è composta dalla sequenza di eventi semplici E, D e B, la cui produzione avviene mediante le

[1] Lo stesso evento primitivo (ad esempio, con buona probabilità, il terminatore) può essere valido per più sequenze. Non viene inviato duplicato, besì inviato una volta sola.

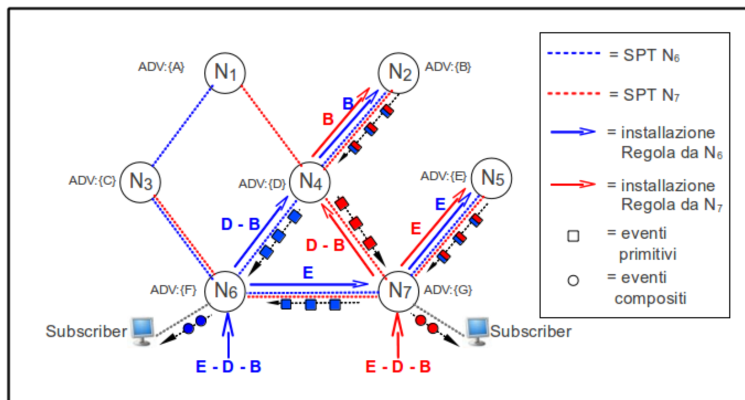
[2] Questo contatore serve per quanto riguarda l'ambiente di simulazione per tener traccia dei messaggi invalidati, allo scopo di non non occupare memoria inutilmente

[3] Indicati mediante ADV:{...}.

sorgenti connesse rispettivamente al nodo N_5, N_4 e N_2.
Nel caso di protocollo ad albero singolo (mostranto in Figura 4.1(a)) viene impiegato un solo SPT che rappresenta l'*overlay network* e lungo quest'albero tutte le regole vengono scomposte ed installate, dalla radice N_1 fino ai nodi foglia. All'atto della pubblicazione degli eventi primitivi E da parte del nodo N_5, D da parte del nodo N_4 e B da parte di N_2, questi vengono elaborati ed inoltrati verso N_1. Una volta collezionati ed individuati quelli compositi che rispettano i vincoli della regola, vengono inoltrate dal nodo N_1 ai nodi N_6 e N_7, nuovamente lungo lo SPT, notifiche dei risultati ottenuti, che raggiungeranno i sottoscrittori interessati.
Nel caso di protocollo ad albero multiplo (mostranto in Figura 4.1(b)), invece, viene generato uno SPT per ogni nodo della rete (vengono mostrati in figura esclusivamente quelli per il nodo N_6 e N_7) e defînto un insieme di *overlay networks* al quale ogni nodo appartiene. L'installazione delle regole avviene simultaneamente alla sottoscrizione dei client ad una specifica regola. É la sottoscrizione, infatti, che avvia la scomposizione e l'installazione delle regole, lungo gli SPT individuati a partire dai nodi N_6 e N_7. Anche in questo caso gli eventi primitivi generati dalle sorgenti, precendentemente elencate, risalgono lo SPT ma, come visibile in figura, verranno instradati lungo tutti i rami degli alberi interessati, giungendo ai nodi N_6 e N_7, che dovranno notificare la loro presenza esclusivamente ai sottoscrittori locali ad essi direttamente connessi.



(a) Albero singolo



(b) Albero multiplo

Figura 4.1: Confronto tra protcollo ad albero singolo e multiplo

4.2 Distribuzione dell'algoritmo

In questo capitolo vengono affrontate le soluzioni impiegate per distribuire non solo la comunicazione tra i nodi, come descritto precedentemente, ma anche l'algoritmo di riconoscimento delle sequenze valide per le regole che definiscono gli eventi compositi.
I componenti che si prendono carico di distribuire l'elaborazione delle sequenze valide sono:

- Rule Deployer;
- Rule Handler.

Il primo modulo viene impiegato durante l'installazione delle sotto-regole e si occupa di determinare le partizioni, che compongono la regola completa e la posizione all'interno della rete, nella quale devono essere installate. Il secondo è incaricato, invece, di gestire in maniera distribuita l'algoritmo, durante la fase di ricezione degli eventi primitivi, il loro filtraggio sulla base delle informazioni installate nei nodi e la raccolta dei risultati da inoltrare verso la radice; il tutto comunicando con il modulo che implementa l'algoritmo descritto nel Capitolo 3.2. Questi due moduli, usati in maniera congiunta, permettono di distribuire il carico computazionale in maniera del tutto trasparente per l'algoritmo di riconoscimento, che si comporta come se, di fatto, una sotto-regola sia in realtà una regola completa.
Nel seguito vengono presentate le scelte fatte riguardo il *Rule Handler* ed il *Rule Deployer* e descritto il loro funzionamento.

4.2.1 Rule Deployer

Il Rule Deployer viene impiegato nella fase iniziale, durante l'installazione delle regole, ed impiega le informazioni ottenute dall'indiviuazione degli SPT e dalla raccolta degli advertisements da parte di tutti i nodi della rete. Considerando un nodo N_1 di livello x, riceve in ingresso la definizione di una regola e ne ricava in uscita un elenco di sotto-regole, suddivise sulla base dei nodi figli del nodo N_1 di livello $x - 1$. Per spiegare come questa suddivisione avvenga prendiamo come esempio la situazione mostrata in Figura 4.2.

`Esempio`

$\underline{RegolaR5}$

$define$	$CE()$
$from$	$D()$
	$and\ each\ B()\ within\ 5\ min\ from\ D$
	$and\ each\ A()\ within\ 3\ min\ from\ B$
	$and\ last\ C()\ within\ 8\ min\ from\ D$

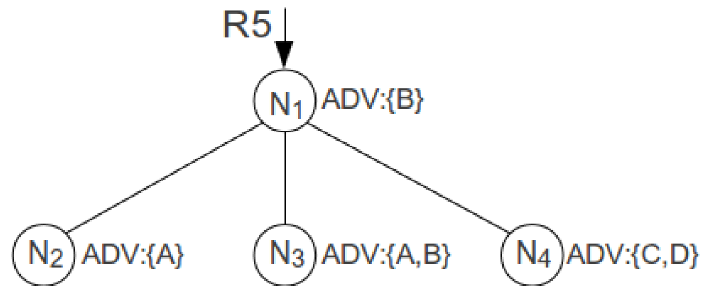


Figura 4.2: Situazione di esempio per la regola R5

La regola R5 viene installata nel nodo N_1 il quale richiede la sua scomposizione al proprio Rule Deployer associato. Questo impiega le informazioni circa gli advertisements[4] degli eventi, sia quelli ricevuti dai propri figli N_2, N_3 e N_4 che quelli locali, per individuare quali partizionamenti della regola siano possibili.
Le regole che guidano questa scomposizione sono le seguenti:

- Scompongo una regola inviando una sotto-sequenza a un figlio N_j solo se tutti gli eventi relativi sono stati notificati esclusivamente da N_j;
- Se un evento è notificato da più figli viene loro inviato in una sotto-regola contenente esclusivamente quell'evento;

[4] ADV:{...} in figura

- Se una negazione è interamente contenuta in una sotto-regola (incluse le dipendenze dei parametri) allora puo' essere unita alla regola; diversamente la negazione viene inviata come una regola a parte;
- Se un aggregato è interamente contenuto in una sotto-regola (incluse le dipendenze dei parametri) allora puo' essere unito alla sotto-regola, altrimenti l'aggregato viene inviato come una regola a parte;
- I parametri che sono completamente all'interno di una sotto-regola vengono trasmessi senza modifiche;
- I parametri tra diverse sotto-regole vengono salvati come parametri *esterni*;
- Le sotto-regole indirizzate al nodo stesso non vengono prese in considerazione, dal momento che verrà impiegata la regola stessa.

Di conseguenza nella situazione di Figura 4.2 il Rule Deployer scomporrà la regola R5 ottenendo le seguenti regole e sotto-regole da installare per ogni nodo:

Nodo N_1: Questo nodo installerà esclusivamente la regola R5.

Nodo N_2: Il nodo riceverà dal nodo N_1 la sotto-regola $R5_1$ contenente solo l'evento A.

Nodo N_3: Il nodo riceverà dal nodo N_1 due sotto-regole: $R5_1$ inviata anche al nodo N_2 e $R5_2$ contenente esclusivamente l'evento B.

Nodo N_4: Il nodo Riceverà dal nodo padre N_1 una sola sotto-regola $R5_3$ definita dall'intera sotto-sequenza di $R5$ contenente gli eventi C e D, essendo questo l'unico figlio che ne ha notificato la generazione.

Quest'operazione viene effettuata a cascata da tutti i nodi per ogni regola che ricevono dal proprio padre e indipendentemente per ogni albero, nel caso di protocollo ad albero multiplo. L'operazione inizia con una regola completa e viene avviata dall'installazione delle regole, nel caso di albero singolo, o dalle sottoscrizioni, nel caso ad albero multiplo.

4.2.2 Rule Handler

In ogni nodo della rete è il *Rule Handler* ad avere l'incarico di gestire gli stack che compongono la regola ed indirizzare gli eventi entranti verso le pile corrette, sulla base delle informazioni memorizzate dallo *static index*, nella fase di installazione della regola nel nodo. Questo modulo ha come obbiettivo di fare da tramite tra l'algoritmo di riconoscimento, che possiede solo le informazioni riguardo la sequenza da individuare, ed il *Processor*, modulo del nodo di rete che si occupa di gestire l'interazione tra nodi e l'invio dei messaggi del protocollo. Agisce inoltre, da tramite per i messaggi tra il *Processor* ed il *Rule Deployer* e mantiene le statistiche del traffico di rete, impiegando il *Traffic Monitor*. Sono stati sviluppati tre differenti *Rule Handler* sulla base della tipologia di protocolo impiegato:

Centralizzato Il Rule Handler in questo caso offre esclusivamente funzionalità legate all'installazione di regole complete, inizializzando uno *Stacks Rule* per ogni regola.

Distribuito É il modulo impiegato dalla versione distribuita di T-Rex; interviene nella scomposizione delle varie sotto-regole inizializzando uno *Stacks Rule* per ogni regola ed effettuando le operazioni di filtraggio come descritto.

Push-Pull Estende il caso precedente intervenendo nella gestione dei messaggi necessari al protocollo, nella gestione delle funzionalità descritte successivamente nel Capitolo 4.4.

Di seguito verranno mostrate le principali operazioni effettuate da questo modulo per garantire la corretta distribuzione del protocollo e dell'algoritmo di riconoscimento, considerando il solo caso distribuito.

Caso Distribuito

Comunicando col *Processor*, il *Rule Handler* effettua la sua prima operazione durante la fase di installazione delle regole. É questo, che per ogni messaggio di definizione di una regola (o sotto-regola), produce un'istanza differente di

Stacks Rule e tiene traccia mediante alcune strutture dati, tra le quali lo *Static Index*, della tipologia di regola, degli eventi primitivi che la descrivono e della loro posizione all'interno dello *Stacks Rule* appena creato. All'arrivo di una pubblicazione ricevuta dal *Processor* è il *Rule Handler*, mediante queste informazioni memorizzate, che la indirizza a tutte e solo le pile che ne attendono l'occorrenza. L'evento viene, dunque, duplicato e memorizzato e, nel caso di pubblicazione di un terminatore, la computazione dei risultati viene avviata e questi ultimi sono collezionati. I risultati ottenuti dall'algoritmo vengono, allora, filtrati dei duplicati mediante il *Dup Remover*, prima di essere ritrasmessi al *Processor*[5], che li impiegherà sulla base delle regole del protocollo distribuito com descritto nel Capitolo 4.1.

4.3 Fasi del protocollo

Vengono di seguito riassunte e descritte le varie fasi del protocollo distribuito sottostante T-Rex.

4.3.1 Start-up

Nella prima fase del protocollo, lo SPT individuato viene trasmesso a tutti i nodi della rete, così che ognuno di essi possa memorizzare la propria posizione nell'albero (ovvero nodo padre e insieme di nodi figli). Questo processo, nel caso di tipologia ad albero multiplo, viene iniziato indipendentemente da ogni nodo della rete, portando così all'individuazione di alberi aventi radice nel nodo iniziale ed identificati dal suo stesso *ID*. In questo modo, al termine di questa fase di *start-up*, ogni nodo possiede le tabelle di routing per indirizzare i messaggi attraverso lo SPT, che ha come radice qualsiasi nodo della rete.

[5] suddivisi per albero nel caso di protocollo ad albero multiplo.

4.3.2 Advertisements

Durante la fase in cui gli advertisements vengono collezionati, ogni nodo memorizza localmente quelli provenienti dai generatori di eventi primitivi a lui direttamente connessi. Successivamente, invia l'insieme degli advertisements locali, aggregati in un messaggio, verso il proprio nodo padre. Questo, una volta ricevuti gli advertisements che ogni suo figlio invia, li ritrasmette insieme con i propri advertisements locali, in un messaggio che instrada verso il proprio padre. Questo procedimento viene effettuato per ogni albero, nel caso di albero multiplo, fino a raggiungere la radice dello stesso. Così facendo, il singolo nodo ha la possibilità di scomporre una regola in più sotto-regole, sapendo da quali figli riceverà quali eventi, e di trasmettere loro solo la porzione di regola che li riguarda.
Il pacchetto a livello applicativo contiene i seguenti campi:

Advertisements Insieme delle tipologie degli eventi primitivi che verranno ricevuti dai generatori di eventi.

Viene successivamente incapsulato in un messaggio a livello di rete contenente informazioni riguardo il mittente e gli alberi lungo i quali dovrà essere trasmesso.

4.3.3 Gestione delle regole

La gestione delle regole installate nel sistema viene effettuata da due componenti:

- il *rule deployer*;
- il *rule handler*.

Il primo si occupa di scomporre una regola inviando una sotto-regola ad un figlio f, se e solo se sono stati ricevuti esclusivamente da f gli *advertisements* di tutti gli eventi primitivi relativi. Se una negazione o un aggregato è interamente contenuto in una sotto-regola (inclusi i vincoli parametrici), ovvero sono contenuti l'evento o entrambi gli eventi che ne definiscono la

finestra temporale, allora può essere inviata come parte della stessa; altrimenti è inviata come una sotto-regola a parte. I parametri che sono completamente all'interno di una sotto-regola vengono trasmessi senza modifiche, mentre quelli definiti da eventi di differenti sotto-regole, vengono valutati solo a questo livello dell'albero (o richiesti ai figli pull nel caso di tipologia push-pull). Il secondo, invece, si occupa di inizializzare, per ogni regola o sotto-regola ricevuta, gli stack e lo static index.
Il pacchetto a livello applicativo contiene i seguenti campi:

ID Identificativo della regola descritta dal pacchetto;

Predicates Insieme delle tipologie di eventi primitivi dei quali si vuole individuare l'occorrenza;

Negations Insieme delle tipologie di eventi primitivi dei quali si vuole determinare l'assenza nella sequenza;

Aggregates Insieme delle tipologie degli eventi dei quali si vuole calcolare un valore aggregato;

Parameters Insieme dei parametri da valutare durante l'individuazione di un risultato valido;

Consuming Elenco degli eventi primitivi che verranno consumati dopo un loro impiego per l'individuazione di una sequenza valida;

CE Template Informazioni circa i campi che dovrà contenere l'evento composito che verrà generato dalla regola.

Viene successivamente incapsulato in un messaggio a livello di rete, contenente informazioni riguardo il mittente, gli alberi lungo i quali dovrà essere trasmesso, l'identificatore della regola, il fatto che si tratti di una regola piuttosto che di una sottoregola e altri campi necessari per l'algoritmo, che verrà descritto nel Capitolo 4.4.

4.3.4 Subscriptions

A seconda che il protocollo impiegato sia ad albero singolo o multiplo, le *subscription* vengono gestite in maniera differente. Nel primo caso vengono collezionate localmente quelle provenienti dai generatori di eventi primitivi direttamente collegati al nodo, da ogni singolo nodo. Successivamente, così come gli *advertisements*, vengono trasmesse a partire dalle foglie del SPT, fino al raggiungimento della radice dello stesso. Nel secondo caso, invece, le sottoscrizioni ad una regola non fanno altro che installare la regola stessa nella rete e propagarla lungo il proprio SPT verso le foglie. In questo caso verranno impiegate le strutture i messaggi ed i moduli della precedente fase, che verrà così sostituita da questa. Così facendo, ogni nodo installerà una quantità maggiore di regole e sotto-regole, ma una volta individuata una sequenza valida e inviatale alla radice dell'albero corrispondente, essa ha già raggiunto l'insieme completo dei sottoscrittori che ne hanno fatto richiesta. Il pacchetto a livello applicativo contiene i seguenti campi:

Subscriptions Insieme degli eventi compositi ai quali ci si sottoscrive;

Constraints Insieme di vincoli sull'evento, oggetto della sottoscrizione.

Viene successivamente incapsulato in un messaggio a livello di rete, contenente informazioni riguardo il mittente e gli alberi lungo i quali dovrà essere trasmesso.

4.3.5 Publications

All'arrivo di un evento, la sua gestione è affidata al *rule handler* che si occupa di estrapolare le informazioni riguardo l'evento dal *payload* del messaggio di rete ricevuto. Esso lo memorizza negli stack di ogni regola che ne attende l'occorrenza, mediante strutture create nella fase di gestione delle regole[6]. Se l'evento è terminatore per una o più regole o sotto-regole, avvia inoltre l'elaborazione, colleziona i risultati e invia gli eventi primitivi (o gli eventi

[6] Nella fase riguardante le sottoscrizioni nel caso ad albero multiplo.

compositi se si tratta di una regola) individuati incapsulandoli in un messaggio di rete e trasmettendolo verso il proprio padre per l'albero appropriato. Gli eventi che risultano validi per l'individuazione di differenti sequenze di diverse regole (o sotto-regole) non vengono duplicati, bensì gestite tramite un modulo: il *Dup Remover*. Questo ha come obbiettivo quello di aggregare le informazioni riguardo molteplici pacchetti a livello applicativo, rappresentanti lo stesso evento primitivo, in un unico pacchetto a livello di rete, che permetta le ricostruzione delle informazioni iniziali, quali i nodi ai quali essere inviati e l'evento contenuto.
Il pacchetto a livello applicativo contiene i seguenti campi:

Event Type tipologia dell'evento primitivo che è stato ricevuto e si sta trasmettendo lungo la rete;

Attributes Insieme dei valori degli attributi dell'evento oggetto della sottoscrizione;

Timestamp Indicazione temporale della generazione dell'evento.

Viene successivamente incapsulato in un messaggio a livello di rete, contenente informazioni riguardo il mittente e gli alberi lungo i quali dovrà essere trasmesso.

4.3.6 Notifications

Anche per quanto riguarda le *notification* bisogna distinguere tra il caso ad albero singolo e quello ad albero multiplo. Se si impiega un singolo albero, una volta che il nodo radice del SPT avrà collezionato delle sequenze di eventi complete, si occuperà di instradarlo verso i nodi che si sono sottoscritti all'evento composito corrispondente, inviandolo ai figli, dai quali ha ricevuto una *subscription* appropriata. Se, invece, si impiegano alberi multipli, le notifiche non risultano necessarie, dal momento che, come mostrato precedentemente, l'instradamento degli eventi compositi avviene contestualmente alla fase di individuazione e installazione degli stessi.
Il pacchetto a livello applicativo viene rappresentato come una *publication*;

viene successivamente incapsulato in un messaggio a livello di rete di tipologia specifica, contenente anch'esso informazioni riguardo il mittente e gli alberi lungo i quali dovrà essere trasmesso, con lo scopo di poter essere discriminato da un evento primitivo e di conseguenza ritrasmesso immediatamente verso i sottoscrittori.

4.4 Modalità alternativa: Push-Pull

Il funzionamento del protocollo per ora esposto si riferisce al caso del protocollo push; nel caso di impiego del protocollo push-pull sono necessarie alcune differenti operazioni per quanto riguarda la suddivisione delle regole e l'inoltro delle sequenze valide di eventi, individuate. In questo caso, le sotto-regole, generate da un nodo a livello x dell'albero, vengono partizionate in: una regola master e zero, una o più regole slave (a seconda di quante sotto-regole sono state prodotte), seguendo le stesse modalità descritte precedentemente. Informazioni circa la tipologia della sotto-regola sono inviate, insieme con la regola stessa, ai nodi figli di livello $x-1$, che si occuperanno di installarle. I nodi che ricevono la regola master operano come nel protocollo push descritto, memorizzando gli eventi pervenuti ed elaborando e trasmettendo i risultati di quest'ultima, non appena si presenta nel nodo un evento terminatore. Nel caso di ricezione di sotto-regole slave, invece, gli eventi derivanti da un terminatore vengono memorizzati dal nodo e non trasmessi immediatamente al prorio padre. Il rule handler del nodo padre di livello x, riceverà dunque solamente i risultati dell'elaborazione proveniente dal nodo associato alla sotto-regola master. Quando questo si verifica, richiede esplicitamente a tutti gli altri figli di inviargli i risultati, non ancora inviati, delle loro elaborazioni memorizzate; questi sono filtrati sulla base di vincoli temporali e parametrici, ottenuti da quelli provenienti dalla sotto-regola master (mediante un messaggio di *awk* contenente i vincoli appena citati). Questa attende la ricezione dei risultati da parte di tutti i suoi figli slave di livello $x-1$, prima di procedere alla sua elaborazione (insieme con un *ack* che rappresenta l'invio di tutti i risultati collezionati). Il padre tiene inoltre traccia di alcune statistiche sul traffico generato dai suoi figli e, ad intervalli

regolari, aggiorna la tipologia delle sotto-regole, scegliendo come master la sotto-regola gestita dal figlio, che ha generato meno traffico nell'ultimo intervallo di tempo tra due aggiornamenti. Così facendo, viene gestito in maniera più efficiente il traffico generato all'interno della rete, effettuando un bilanciamento del carico tra i nodi della stessa.
Per determinare il comportamento di un nodo che ha installato una sotto-regola, il protocollo Push-Pull tiene in considerazione, oltre alla scelta del master basata sul traffico di rete, anche delle finestre temporali relative tra il terminatore della sotto-regola master e i terminatori delle altre sotto-regole. Queste informazioni, individuate dal Rule Deployer del nodo di livello superiore e trasmesse in fase di installazione, permettono di discriminare il comportamento in fase di individuazione dei risultati. Una sotto-regola può infatti possedere una finestra aperta nel passato, nel futuro o entrambi i casi a seconda che il suo terminatore, rispetto a quello della regola master, si trovi piu vicino, più lontano o in una differente sequenza, della radice del grafo che descrive la regola completa. Nel caso di una finestra aperta nel futuro, il nodo, sebbene non sia stato scelto come master, si comporterà come tale, inviando i risultati al livello superiore, non appena questi saranno individuati, senza attendere l'arrivo di una richiesta di *awk*. I tre casi di finestra sono esemplificati in Figura 4.3. Nella figura, la sotto-regola selezionata come master è individuata dalla rettera M, mentre, le sotto-sequenze che definiscono le varie sotto-regole sono delimitate da un riquadro tratteggiato e nominate sopra lo stesso; sono presentate sotto ogni ramo del grafo, inoltre, i vincoli temporali tra i vari eventi. Si può notare come, scelta la sotto-regola master R_2, le sotto-regole R_1, R_3 e R_4 presentino rispettivamente: una finestra aperta nel passato di 15 unità di tempo, una finestra aperta nel futuro di ampiezza 5 ed entrambe le finestre (nel passato ampia e nel futuro) aperte.
Allo scopo di permettere un ritardo della computazione, i terminatori ricevuti vengono memorizzati in due distinte code a seconda del livello gerarchico:

- Awk Pending Terminators.
- Ack Pending Terminators;

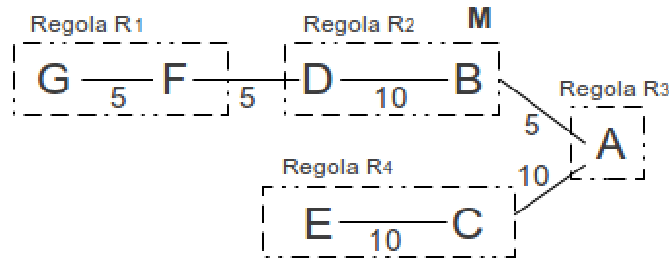


Figura 4.3: Esempio di finestre aperte tra sotto-regole

La prima coda contiene i risultati di un nodo al livello gerarchico $x-2$ che effettua il riconoscimento di una sotto-regola slave ed è in attesa di un messaggio di *awk* da parte del proprio padre (di livello $x-1$). La seconda coda rappresenta,invece, i risultati che sono stati ricevuti da un nodo di livello $x-1$, in attesa di ricevere un *ack* da parte di tutti i figli slave. Una volta ricevuti questi messaggi, può allora avviare la computazione. Se la sotto-regola in questione è master rispetto al nodo di livello x padre, o se possiede una finestra aperta nel futuro, allora i pacchetti risultanti possono essere inviati direttamente verso questo nodo, altrimenti se è slave viene impiegata la coda *Awk Pending Terminators* per memorizzare i risultati, in attesa di una richiesta da parte del nodo padre, mediante un messaggio di *awk*. Spostando la computazione in avanti nel tempo, si ottiene il vantaggio di poter filtrare i risultati, sulla base dei parametri contenuti nella sotto-regola master.
In Figura 4.4 viene rappresentato un esempio di uso delle code *AwkPendingTerminator* e *AckPendingTerminator* durante il normale funzionamento del sistema. Viene rappresentata una gerarchia di tre livelli di nodi, etichettati mediante la sotto-regola ricevuta. Vengono, inoltre, identificate le sotto-regole che sono state selezionate come master e la sequenza temporale degli eventi generati dai nodi di più basso livello. Si consideri che ogni evento rispetti i vincoli temporali e parametrici imposti dalla regola.
Come si può vedere in figura, gli eventi primitivi che rappresentano il terminatore di una sotto-regola slave, non vengono inviati immediatamente verso il nodo padre, bensì memorizzati nella coda *AwkPendingTerminator*; nell'e-

sempio gli eventi $D1$ e $D5$ per il nodo N_3 e gli eventi $B3$ e $B9$ per il nodo N_5.
Al tempo 7 viene generato, dal nodo N_4, un evento C. Questo rappresenta il terminatore di una regola selezionata come master e, di consenguenza, è inoltrato verso il nodo N_1 senza essere memorizzato in alcuna coda. Il nodo N_1, ricevuto $C7$, terminatore anche per la sua sotto-regola, lo memorizza nella coda *AckPendingTerminator* per identificarlo come in attesa di ricevere gli eventi individuati dalle sotto-regole slave. Contemporaneamente inoltra un messaggio di *awk* verso il nodo N_3 che si occuperà di inoltrare i risultati della sua computazione, effettuata sulla base dei terminatori memorizzati nella coda *AwkPendingTerminator* (in questo caso $D1$ e $D5$). Ricevuti questi eventi il nodo N_1 inserirà il terminatore $C7$ nella pila *AwkPendingTerminator*, in quanto la regola installata è slave rispetto al livello x.
All'arrivo dell'evento $A11$ al nodo N_6, terminatore di una regola master di livello $x - 2$, questo è inoltrato direttamente al nodo N_2 e memorizzato nella coda *AckPendingTerminator*. Anche in questo caso, come nel precedente, il nodo contenente la regola slave (N_5) viene risvegliato mediante un messaggio *awk* e si occupa di trasmettere i risultati dell'elaborazione della regola, a partire dai terminatori memorizzati nella sua coda *AwkPendingTerminator*, ovvero $B3$ e $B9$. A differenza del nodo N_1, però, il nodo N_2 di livello $x-1$ ha installato una sotto-regola master perciò individua ed invia immediatamente i risultati ottenuti a partire da $A11$. Giunti al livello x, viene effettuata la stessa operazione, notificando il nodo N_1 mediante *awk* e collezionando gli eventi necessari al riconoscimento dell'intera regola.
In Figura 4.5 vengono mostrati gli stati in cui un evento può trovarsi durante il funzionamento del sistema e le condizioni che scatenano un cambiamento di stato.

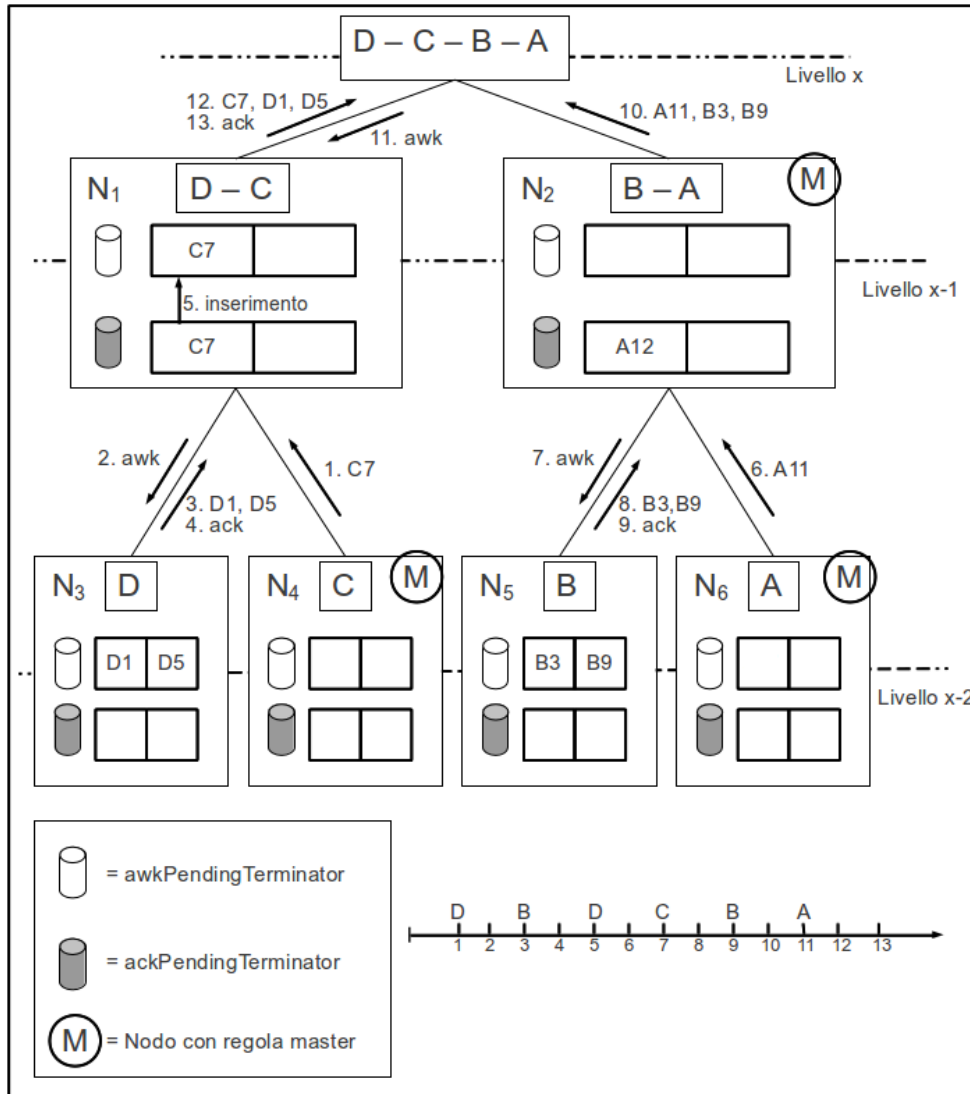


Figura 4.4: Esempio di impiego del protocollo Push-Pull

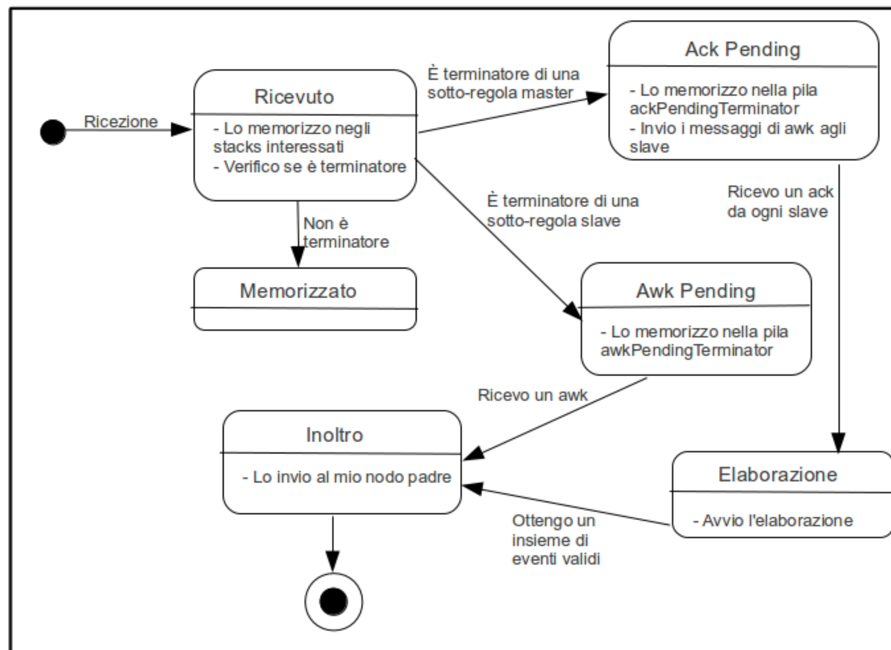


Figura 4.5: Stati possibili per un evento

4.5 Protocolli implementati

Sono stati dunque implementati, sulla base di quanto detto precedentemente, cinque differenti protocolli, con lo scopo di confrontarli ed ottenere informazioni circa i loro vantaggi a seconda degli scenari simulati:

- Centralizzato;
- Distribuito ad albero singolo;
- Distribuito ad albero multiplo;
- Push-Pull ad albero singolo;
- Push-Pull ad albero multiplo.

Il caso centralizzato è stato simulato con l'obbiettivo di avere un riferimento da confrontare con gli altri e poter determinare il vantaggio, in termini di tempo di elaborazione e traffico di rete, introdotto dalle varie tipologie di distribuzione. I risultati delle simulazioni sono presentati nel Capitolo 7.

Capitolo 5

Ambiente di simulazione: OMNet++

Viene di seguito fornita una panoramica sui simulatori ad eventi discreti, ed in particolare sul simulatore scelto: OMNet++

5.1 OMNet++

Scopo di questa presentazione di OMNet++ è quello di fornirne una panoramica, in modo da poterne descrivere i punti di forza, il modello di programmazione e i componenti principali, senza soffermarsi sui dettagli che possono essere facilmente reperiti nel manuale utente [5]. OMNet++ è un simulatore ad eventi discreti, pertanto viene inizialmente descritto che cosa si intende per simulatore ad eventi discreti.

5.2 Simulatori ad eventi discreti

Un sistema ad eventi discreti è un sistema in cui i cambiamenti di stato (*eventi*) avvengono in istanti di tempo discreti.
Tali eventi sono istantanei, per cui è nullo il tempo impiegato per il cambiamento di stato. Si assume quindi che niente (nulla di interessante) accada tra due eventi consecutivi.

Nel sistema, di conseguenza, non si verificano cambiamenti di stato tra l'occorrenza di due eventi; contrariamente a quanto avviene nei sistemi continui, in cui i cambiamenti di stato sono continui. Tutti i sistemi che possono essere modellizzati come sistemi ad eventi discreti possono essere simulati tramite un sistema di questo tipo (*Discrete Event Simulation (DES)*). Una rete di computer, ad esempio, può essere vista come un sistema ad eventi discreti, di cui alcuni esempi di eventi possono essere: l'inizio della trasmissione di un pacchetto, la fine della trasmissione di un pacchetto, l'estra- zione dalla coda di un pacchetto. Ciò implica che tra due eventi come l'inizio della trasmissione di un pacchetto e la fine della stessa, non si verifichi nessun evento rilevante per il funzionamento del sistema. L'istante temporale nel quale accadono gli eventi è comunemente detto *timestamp* e si possono distinguere differenti modelli temporali:

tempo viruale: il tempo all'interno del modello di simulazione, scandito dall'occorrenza degli eventi;

tempo reale: il tempo trascorso all'esterno del modello simulativo, ovvero il tempo di vita del programma che esegue la simulazione;

tempo di CPU: il tempo effettivo durante il quale il programma ha effettivamente utilizzato la CPU.

Nelle simulazioni ad eventi discreti, l'insieme degli eventi futuri viene memorizzato in una struttura dati comunemente chiamata *Future Event Set (FES)*. Il simulatore si comporta eseguendo il seguente pseudo-codice:

```
inizializzazione
while(l'FSE non è vuoto e la simulazione non è terminata)do{
estrai il primo evento dall'FES e
tempo virtuale = timestamp di e
processa e[1]
}
end while
terminazione
```

[1]eventuale rimozione/inserzione di nuovi eventi dal/nel FES sulla base dell'evento *e*

La fase di inizializzazione include la creazione del modello da simulare e l'inserimento degli eventi iniziali nel FES, mentre la fase di finalizzazione si occupa della scrittura dei files contenenti le informazioni statistiche sulla simulazione appena conclusa. Gli eventi sono processati strettamente in ordine di timestamp, di modo che venga mantenuta la causalità tra gli eventi e nessun evento con timestamp t_1 possa influire su un altro con timestamp t_2 (con $t_1 > t_2$). È da notare che la funzione che si occupa di processare un evento fa parte del codice fornito dall'utente. La simulazione si ferma quando non sono rimasti più eventi nel FES, quando è stato raggiunto il tempo massimo di esecuzione stabilito, o perché le statistiche raccolte hanno raggiunto il desiderato livello di accuratezza. Prima che il programma di simulazione termini completamente, vengono registrate, su file, tutti i dati e le statistiche indicati dall'utente.

5.3 Punti di forza di OMNet++

Prima di iniziare a descrivere più dettagliatamente gli aspetti principali di OMNet++, vengono qui di seguito elencati i punti di forza di tale simulatore, per cui esso è stato preferito ad altri:

- è gratuito, per scopi di ricerca accademica[2] , ed è opensource;
- il kernel del simulatore è chiaramente separato dal modello della simulazione tramite una semplice interfaccia, per cui l'unica preoccupazione, da parte dell'utente, è di implementare il modello secondo le API del kernel;
- è altamente modulare: i componenti del modello della simulazione possono essere dunque strutturati in maniera gerarchica; risulta facile aggiungere nuove funzionalità ad un modulo estendendone un altro. Tali moduli sono inoltre riutilizzabili e combinabili tra loro;

[2] per scopi commerciali, bisogna acquistare una licenza di OMNest, la sua controparte a pagamento

- ha una architettura tale per cui il modello, la tipologia della rete e le simulazioni (i parametri del modello) sono configurabili in files separati. Risulta quindi semplice passare da uno scenario all'altro, o cambiare alcuni parametri della simulazione;
- è in grado simulare efficientemente topologie di rete molto grandi.
- fornisce numerose classi ausiliari e funzionalità accessorie: generatori di numeri casuali (secondo numerose distribuzioni di probabilità), classi per raccogliere i dati, moduli statistici per elaborare i dati raccolti, gerarchie di pacchetti, incapsulamento di pacchetti;
- ha una comoda interfaccia grafica, in grado di visualizzare ogni singolo dettaglio del modello implementato, e dei pacchetti in transito nella rete;
- ha un'ottima documentazione, che rende estremamente semplice imparare ad usare il simulatore sfruttandone appieno tutte le funzionalità;
- dispone, infine, di una grossa comunità di utilizzatori, per cui non è difficile trovare risposte a domande relativa all'utilizzo del simulatore.

5.4 Caratteristiche Principali

Moduli e loro gerarchia

Un modello di OMNet++ consiste in una gerarchia di moduli annidati, che comunicano tramite messaggi tra loro.
Come esemplificato nella Figura 5.1, il modulo padre della gerarchia è il *system module*, il quale contiene dei sotto-moduli, che a loro volta possono contenere altri sotto-moduli, in modo da poter riflettere nel modello la struttura logica del sistema da simulare.
I moduli che contengono sotto-moduli sono detti *compund module*, mentre quelli che sono alla fine della gerarchia sono detti *simple module*. I simple module sono i moduli che contengono gli algoritmi del modello, implementati

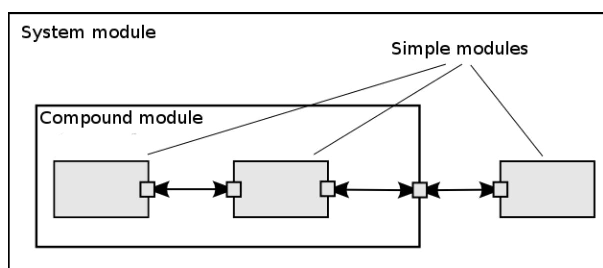


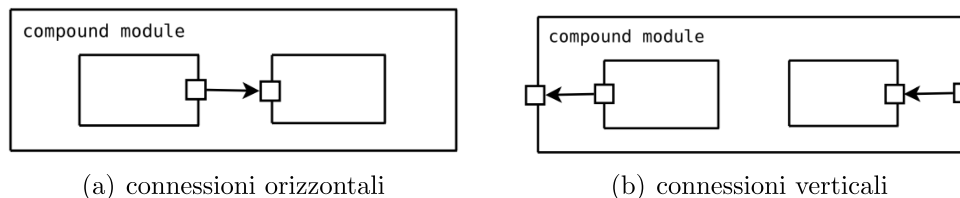
Figura 5.1: Gerarchia dei moduli di OMNet++

dall'utente in `C++`, estendendo la classe `cSimpleModule`.
Visti esternamente, sia i *simple module* che i *compound module* sono identici, pertanto l'utente può facilmente decidere di reimplementare la funzionalità di un *simple module* all'interno di un *compound module* e viceversa, senza modificare la struttura del modello. I tipi dei moduli possono essere memorizzati esternamente al modello in cui vengono utilizzati, pertanto si possono creare vere e proprie librerie di componenti.

Messaggi, interfacce e link

I moduli comunicano tra loro scambiandosi messaggi. Tali messaggi possono contenere strutture dati di qualunque complessità.
I *simple module* possono spedire messaggi direttamente alla loro destinazione, oppure attraverso un cammino predefinito, utilizzando le *interfacce (gates)* e le connessioni.
Il tempo virtuale di un modulo avanza quando esso riceve un messaggio, il quale può provenire da un altro o anche dallo stesso, dando così la possibiltà di implementare facilmente un timer.
Per i messaggi tra differenti moduli per i quali vengono impiegate le interfacce si fa differenza tra *output gate* e *input gate*, con lo scopo di inviarli e riceverli rispettivamente.
Ogni *connessione (link)* tra i moduli viene creata all'interno di un singolo livello della gerarchia. Di conseguenza, all'interno di un compound module, si possono creare connessioni orizzontali, tra le interfacce di due suoi sotto-moduli (Figura 5.2(a)), oppure verticali tra l'interfaccia di un sotto-modulo

e quella del compound module che lo contiene (Figura 5.2(b)).



(a) connessioni orizzontali (b) connessioni verticali

Figura 5.2: Tipologie di connessioni tra moduli

I messaggi generati da un simple-module ed indirizzati ad un altro simple module, per essere processati, tipicamente viaggiano attraverso una serie di connessioni, sia verticali sia orizzontali. Durante questo percorso i compound module agiscono semplicemente come dei relay, facendo passare i messaggi dal loro interno al loro esterno.

Modello di trasmissione dei pacchetti

Alle connessioni tra i moduli possono essere assegnati tre parametri che facilitano la modellazione di reti di comunicazione: il ritardo di propagazione, il tasso di errore e la velocità di trasmissione. Tutti e tre questi parametri sono opzionali e possono essere differenziati per ogni singola connessione, oppure possono essere definite tipologie di link riutilizzabili in tutto il modello. Il ritardo di propagazione, ovvero la quantità di tempo che passa da quando viene spedito il primo bit a quando tale bit giunge a destinazione, viene implementato ritardando la schedulazione del messaggio in maniera appropriata. Il tasso di errore, ovvero la probabilità che un bit non venga trasmesso correttamente, viene impiegato per simulare link non affidabili. Infine la velocità di trasmissione concorre a determinare il tempo di trasmissione di un pacchetto, ovvero per quanto tempo il simulatore rende il canale non disponibile per altre trasmissioni.

Parametri

Ogni modulo può avere dei parametri e i valori di tali parametri possono essere assegnati sia nella definizione del modulo, sia in un file di configurazione separato: il file `omnetpp.ini`. Tale file ha una sintassi propria e viene interpretato dal simulatore all'avvio della simulazione. I parametri possono essere stringhe, numeri, valori booleani o persino alberi XML. I parametri numerici possono assumere valori statici, generati casualmente secondo una certa distribuzione di probabilità, oppure generati mediante funzioni definite dall'utente. Un esempio di file `omnetpp.ini` si può vedere in Figura 5.3.

```
[General]
network = Network
sim-time-limit = 1810s
output-vector-file = results.vec
output-scalar-file = results.sca
output-scalar-file-append = true
num-rngs = 2
[Parameters]
Network.debug = false
Network.numBrokers = 5
Network.brokers[*].publisher.numAdvertisements = 100
Network.brokers[*].simpleEventsMaxId = 300
Network.brokers[*].complexEventsMaxId = 500
Network.brokers[*].singleTree = true
Network.brokers[*].pushPull = false
Network.brokers[*].distributed = true
Network.brokers[*].centralized = false
Network.brokers[*].subscriber.numSubscriptions = 50
```

Figura 5.3: Esempio di file `omnet.ini`

Topologia del modello

La struttura del modello, ovvero le interconnessi tra i suoi moduli (simple o compound), sia verticalmente (lungo la gerarchia), sia orizzontalmente (allo stesso livello gerarchico), è definita in un file detto *NED file*. Tramite il NED file si possono quindi definire i simple module, i loro parametri, i moduli compound (come insieme interconnesso di quelli precedentemente definiti, siano essi simple o compound). La rete è costituita da un compound module, il quale a sua volta ha al suo interno i nodi connessi tra loro a descriverne la topologia. I nodi stessi della rete possono essere a loro volta dei compound module. Un esempio di NED file si può vedere in Figura 5.4.

```
module Node
{
    parameters:
        bool debug;
        int numLinks;
        int simpleEventsMaxId;
        int complexEventsMaxId;
        bool singleTree;
        bool pushPull;
        bool distributed;
        bool centralized;
        @display("bgb=321,374;bgl=2");
    gates:
        inout links[numLinks];
    submodules:
        processor: Processor {
            parameters:
                singleTree = singleTree;
                pushPull = pushPull;
                centralized = centralized;
                distributed = distributed;
                simpleEventsMaxId = simpleEventsMaxId;
                complexEventMaxId = complexEventsMaxId;
                debug = debug;
                @display("p=153,131;i=block/cogwheel");
        }
        topologyManager: TopologyManager {
            parameters:
                debug = debug;
                numLinks = numLinks;
                @display("p=153,290;i=block/layer");
        }
        reorderer: Reorderer {
            parameters:
                debug = debug;
                numLinks = numLinks;
                @display("p=153,209;i=block/queue");
        }
    connections:
        processor.lowerOut --> reorderer.upperIn;
        processor.lowerIn <-- reorderer.upperOut;
        topologyManager.upperOut --> reorderer.lowerIn;
        topologyManager.upperIn <-- reorderer.lowerOut;
        for i=0..numLinks-1 {
            topologyManager.lowerGates[i] <--> links[i];
        }
}
```

Figura 5.4: Esempio di *NED file*

Modello di programmazione

I simple module contengono gli algoritmi, implementati come funzioni `C++`. Chi scrive la simulazione può usare tutta la potenza e la flessibilità offerti da tale linguaggio, come ad esempio i concetti della programmazione object oriented (ereditarietà, polimorfismo, etc...).
Tutti gli oggetti usati da una simulazione, come i messaggi, i moduli, le code, sono rappresentati come delle classi `C++`. Essi sono stati progettati per cooperare insieme efficientemente, creando un potente framework di programmazione delle simulazioni. Pertanto la libreria delle classi utilizzabili dalla simulazione, fornita da OMNet++, è costituita da classi come i moduli, le interfacce, le connessioni, i parametri, i messaggi, le code, classi per raccogliere i dati prodotti dalla simulazione, classi statistiche e classi che stimano le distribuzioni. Tutte queste classi sono implementate con lo scopo di semplificare l'analisi degli oggetti durante l'esecuzione della simulazione, per visionarne le caratteristiche come il nome, le variabili di stato oppure il contenuto di code e array. Grazie a tale caratteristica OMNet++ è anche in grado di fornire una interfaccia grafica con cui è possibile esaminare la simulazione in maniera approfondita. Il modello della simulazione è implementabile tramite un paradigma *event-driven*, oppure in uno stile simile ai processi dei sistemi operativi, dove il flusso della simulazione viene esplicitamente passato da un processo all'altro. Quest'ultimo è espressamente sconsigliato dall'autore di OMNet++ a meno di casi in cui quello event-driven risulti incompatibile col contesto applicativo. Pertanto in questo lavoro è stato impiegato il paradigma event-driven, approfondito qui di seguito. Come detto precedentemente, in una simulazione gli eventi avvengono nei simple module. Pertanto sono questi che racchiudono tutto il codice `C++` che reagisce e genera gli eventi, ovvero il comportamento del modello. Nello stile event-driven, un simple module viene implementato estendendo la classe `cSimpleModule`, che non offre particolari funzionalità, ma espone tre funzioni, che è necessario ridefinire per implementare la logica applicativa:

- `void initialize();`
- `void handleMessage(cMessage * msg);`
- `void finish();`

`initialize()` si incarica di costruire la rete, istanziare i simple e compound module necessari e connetterli come definito nel NED file.
`handleMessage()` viene chiamata quando il modulo in questione deve processare un evento, pertanto è tramite questa funzione che viene implementato il comportamento del sistema. Questa, che riceve come parametro l'evento stesso, viene invocata dal kernel del simulatore nel momento in cui un messaggio raggiunge il modulo.
Infine `finish()` viene invocata per ogni modulo quando la simulazione termina correttamente e tipicamente viene impiegata per scrivere le statistiche finali.

Esecuzione della simulazione

Quando viene avviata la simulazione, OMNet++ innanzitutto legge il file di configurazione `omnetpp.ini` che contiene tutte le impostazioni che ne determinano l'esecuzione; ovvero con quali interfacce utente, se in modalità batch, oppure interattiva e con quali valori dei parametri. Se eseguito in modalità grafica, il simulatore permette di ispezionare tutti i dettagli relativi ai moduli della simulazione ed ai messaggi a run-time. L'output della simulazione viene scritto in file di dati chiamati *scalar* e *vector*, i quali contengono rispettivamente i singoli campioni delle misure di interesse raccolti e valori numerici delle quantità aggregate. OMNet++ fornisce, infine, anche dei programmi grafici per poter visualizzare e filtrare i dati contenuti nei file scalar e vector.

Capitolo 6

Dettagli implementativi

In questo capitolo verranno descritti i dettagli riguardanti l'implementazione dei vari componenti che compongono il sistema. Dapprima verrà presentato il lavoro riguardante il simulatore, mostrando i moduli che sono stati implementati, le relazioni e lo scambio di messaggi che intercorre tra di loro. Successivamente verrà esaminata la logica applicativa del sistema ponendo l'attenzione, anche in questo caso, ai moduli che la compongono ed al loro funzionamento coordinato.
Per quanto riguarda l'ambito simulativo il mio contributo è stato principalmente focalizzato sull'ideazione e lo sviluppo del modulo *Processor*, tramite tra il sistema simulato mediante Omnet++ e la sottostante implementazione del riconoscimento di eventi compositi e di gestione delle regole.
In quest'ultimo ambito, il mio lavoro è incentrato, invece, sulla definizione e l'implementazione del modulo atto a gestire la memorizzazione degli eventi primitivi ed il successivo riconoscimento delle sequenze che descrivono una determinata regola: lo *StacksRule*.
Nell'affrontare quest'argomento, prendiamo come riferimento lo scenario distribuito, che rappresenta il principale risultato di questo lavoro, nelle sue accezioni ad albero singolo ed ad albero multiplo.
Verranno inoltre impiegati *structure diagrams* e *behavior diagrams* UML, per mostrare rispettivamente la struttura dei componenti e le interazioni che intercorrono tra di loro.

6.1 Architettura della simulazione

Il modello di simulazione che abbiamo sviluppato definisce la rete a partire da un insieme di nodi che presentano le stesse funzionalità. Nell'algoritmo T-Rex distribuito che vogliamo testare, infatti, tutti i nodi hanno la possibilità di pubblicare eventi primitivi, effettuare sottoscrizioni ad eventi compositi, definire nuovi eventi compositi, descritti da una nuova regola installata nel sistema. Il singolo nodo, rappresentato tramite un compound module è composto da diversi simple modules, mostrati in Figure 6.1. Come evidenziato dalla figura, ogni nodo è composto di cinque simple modules che cooperano per eseguire il protocollo:

- il *Publisher*;
- il *Subscriber*;
- il *RuleManager*;
- il *TopolgyManager*;
- il *Processor*.

I primi tre moduli si occupano di simulare i client connessi al nodo e, quindi, di generare le tipiche richieste che sopraggiungono al sistema. Il primo si occupa infatti di generare gli *advertisements* e le successive *publications* di eventi primitivi, provenienti dai generatori di eventi primitivi connessi direttamente al nodo e, successivamente, inviarle attraverso la rete (rispettivamentente nella fase di start-up e durante il funzionamento a regime del sistema). Il secondo invece, gestisce ed instrada le *subscriptions* dei client che sono interessati ad eventi compositi, definiti da regole già installate nel sistema. É il *RuleManager*, infine, che si occupa di gestire l'installazione delle regole da parte dei clients connessi alla rete. A partire dal basso, il primo simple module che compone il nodo è il *TopologyManager*. Questo comunica

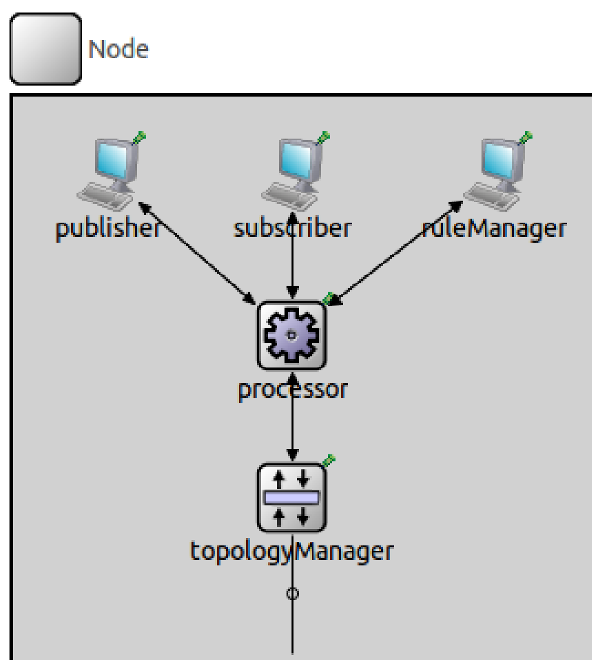


Figura 6.1: Architettura di un nodo del modello di simulazione

direttamente con gli altri nodi che compongono la rete e ha lo scopo di individuare e memorizzare gli SPT durante la fase di start-up, mentre, durante la fase di funzionamento a regime, di instradare i messaggi del protocollo e quelli contenenti gli eventi generati da T-Rex. Questa operazione viene effettuata sulla base delle tabelle di instradamento generate durante la prima fase. Infine l'ultimo componente del nodo, quello che comunica effettivamente col le classi C++ che implementano T-Rex, è il *Processor*. Questo modulo riceve gli eventi dai nodi figli o dai client lui connessi e, comunicando col rule handler, inoltra i risultati elaborati verso il suo padre lungo lo SPT. Si occupa, inoltre, di gestire le varie fasi del protocollo, sulla base dello scenario che si vuole simulare.
Oltre a questi cinque moduli, vi è anche il *WorkloadHandler*; questa classe si occupa di individuare l'insieme delle regole installate nella rete, gli advertisements e le sottoscrizioni che i singoli nodi gestiranno. Inoltre coordina la creazione delle pubblicazioni sulla base dei parametri fissati in fase di definizione dell'ambiente da simulare. Un ultima classe che concorre alla

simulazione del funzionamento dei nodi della rete è il *TrafficMonitor*, anch'esso instanziato uno per nodo componente la rete. Questa non comunica con gli altri simple modules facenti parte del nodo, ma direttamente col rule handler. Quest'ultimo, ricevendo un evento, ne comunica la provenienza, così da poter collezionare statistiche circa il traffico all'interno della rete. Queste statistiche vengono anche utilizzate, nel caso di algoritmo push-pull, per decidere quale sotto-regola (di conseguenza quale figlio) individuare come master, e di coseguenza quali computazioni avviare istantaneamente e quali posticipare.

6.1.1 Diagrammi di struttura

Presentiamo nei paragrafi successivi dei diagrammi di struttura UML allo scopo di mostrare e descrivere le classi C++ che implementano i moduli del simulatore.
Verrà descritto, per ogni classe, l'impiego delle differenti funzionalità messe a disposizione e le strutture dati usate.

Publisher

La classe che implementa le funzionalità del Publisher è mostrata in Figura 6.2. Le strutture dati *localAdvertisements* e *localAvdertisementsList* sono necessarie per memorizzare gli advertisements che saranno distribuiti lungo la rete, ovvero gli eventi primitivi che verranno generati. Possiede, inoltre, attributi che gli permettono di individuare la sua posizione all'interno della rete e di impiegare il canale di comunicazione col Processor associato. Questi sono *myAddr*, *lowerIn* e *lowerOut* che rappresentano rispettivamente l'indirizzo identificativo del nodo di appartenenza, il canale in ingresso e in uscita verso il Processor, associato allo stesso nodo.
Ulteriore struttura dati posseduta è il *pubTimer*; un insieme di timers (messaggi diretti al simple module stesso) che vengono aggiornati, sulla base dei parametri di simulazione, per determinare la frequenza di generazione degli eventi primitivi. Infine possiede riferimenti ad un *Encoder* ed a un *Work-*

loadHandler, che verranno impiegati per la codifica dei pacchetti applicativi della simulazione, mostrati in Figura 6.10(a) e per la generazione del traffico. Le funzionalità presentate da questa classe, oltre alle funzioni ridefinite a partire dal cSimpleModule, sono scatenate dall'elaborazione dei messaggi ricevuti, come mostrato nel Capitolo 5.4. Queste sono fondamentalmente due; *createAndSendAdvertisements()* e *createAndSendPublications()* che generano, rispettivamente, advertisements e publications di eventi primitivi.

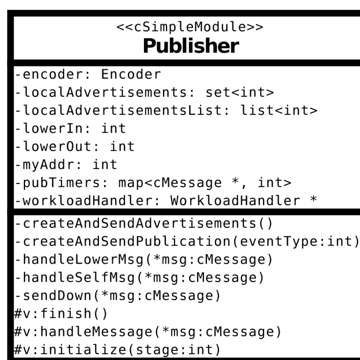


Figura 6.2: Classe Publisher

Subscriber

La classe che implementa questo modulo necessita di una struttura dati per memorizzare l'insieme degli eventi compositi a cui i client, da questa rappresentati, si sono sottoscritti: *localSubscription*. Possiede inoltre tutti i riferimenti ad altre classi, gli attributi necessari alla propria identificazione ed all'identificazione dei propri canali di comunicazione elencati per il Publisher. A differenza di quest'ultimo, però, necessita di effettuare anche una fase di decodifica dei messaggi ricevuti, di conseguenza possiede un riferimento al Decoder. Sono presenti infine altri due attributi necessari per fini statistici; *totalRegisteredDelay* e *numReceivedMessage*, ovvero la somma di tutti i ritardi di computazione di un evento composito e il numero di messaggi ricevuti.
La principale funzionalità esposta è rappresentata dalla funzione *createAndSendSubscription()*, anch'essa facente uso del WorkloadHandler.

Figura 6.3: Classe Subscriber

Rule Manager

Il RuleManager, così come il Subscriber ed il Publisher, presenta *myAddr*, *lowerIn* e *lowerOut*, allo scopo di identificare il nodo di appartenenza ed i propri canali. Presenta inoltre il riferimento al WorkloadHandler, mostrato più dettagliatamente in Figura 6.8.
Analogamente alle precedenti classi, si occupa della generazione dei pacchetti dell'algoritmo, in particolare, di quelli rappresentanti le regole di definizione degli eventi compositi. Effettua questo compito mediante la funzione *createAndSendRules()* con l'impiego del WorkloadHandler.

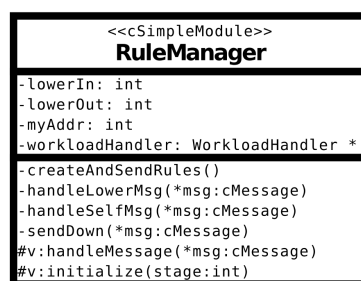


Figura 6.4: Classe RuleManager

Topology Manager

Il TopologyManager, incaricato di simulare il livello di rete dei vari nodi, possiede le strutture dati *neighborsAddresses* e *neighborsGates* impiegate come tabelle di routing e collezionate durante l'iniziale fase di start-up, descritta nel Capitolo 4.3.1, mediante la funzione *sendNeighborDiscovery()*. Possiede inoltre, a scopi statistici, variabili che tengono traccia del numero di bit e di messaggi transitati per la specifica interfaccia di rete; rispettivamente *totBitSentDown* e *totMsgSentDown*.

```
<<cSimpleModule>>
TopologyManager
-myAddr: int
-neighborsAddresses: map<int, int>
-neighborsGates: map<int, int>
-totBitSentDown: long
-totMsgSentDown: long
-upperIn: int
-upperOut: int
-handleLowerMsg(*msg:cMessage)
-handleUpperMsg(*msg:cMessage)
-sendDown(*msg:cMessage)
-sendNeighborDiscovery()
-sendUp(*msg:cMessage)
#v:finish()
#v:handleMessage(*msg:cMessage)
#v:initialize(stage:int)
+getAddr(): int
```

Figura 6.5: Classe TopologyManager

Processor

La classe Processor ha come obbiettivo la gestione del protocollo. Necessita dunque dei parametri della simulazione, dell'identificativo del nodo a cui si riferisce(*myAddr*), dei nodi che compongono la rete(*allNodes*), della sua posizione all'interno degli SPT individuati nella fase di start-up (*children* e *parents*) e delle regole installate nella rete, a partire dal proprio nodo e dagli altri (*myRules* e *rulesList*). Oltre a queste informazioni tiene traccia, mediante le strutture dati *advertisements*, *eventSubscriptions*, *localAdvertisements*, *localEventSubscriptions* e *localSubscription* degli advertisements e delle sottoscrizioni, sia locali, che di tutta la rete. Possiede inoltre i riferimenti a tutti i propri canali di comunicazione, con i client connessi al nodo

e con l'interfaccia di rete (il TopologyManager). Per indirizzare le richieste alla logica applicativa, inoltre, tiene traccia dei RuleHandlerDist (*ruleHandlerDist*) suddivisi per SPT. Memorizza infine il tempo di elaborazione totale nella variabile *overallProcessingTime* e impiega un *encoder* e un *decoder* per codificare e decodificare le informazioni che la simulazione riceve e trasmette da e verso la logica applicativa, sotto forma di pacchetti.
Le principali funzionalità che vengono attivate dai messaggi ricevuti sono:

handleComplexEvent() Gestisce la ricezione di un evento composito;

handlePublicationDist() Gestisce gli eventi ricevuti dai propri nodi figli;

handlePublisherMsg() Gestisce gli eventi pubblicati dai clienti a lui connessi;

handleRulePkt() Gestisce la ricezione di una regola o sotto-regola proveniente dal proprio nodo padre;

handleRuleManagerMsg() Gestisce l'installazione di una regola da parte di un rule manager a lui direttamente connesso;

sendXXX() Una serie di funzioni per l'inoltro di messaggi verso specifiche destinazioni.

Figura 6.6: Classe Processor

TrafficMonitor

Il TrafficMonitor presenta delle strutture dati impiegate per memorizzare le relazioni tra regole e rispettive sotto-regole (*subRules*), quelle tra le sotto-regole e gli eventi primitivi che le compongono, e viceversa (*subRulesEvents* e *eventsSubRules*), il computo di quante regole impiegano un determinato evento (*eventsCount*) ed il peso di ogni sotto-regola(*subRulesWeight*).
Le funzionalità offerte sono l'installazione di una nuova regola e l'aggiornamento delle statistiche, sulla base delle pubblicazioni transitate. La seconda avviene mediante la funzione *processPubPkt()*, mentre la prima tramite *installNewRule()* e impiega funzioni private *computeWeight()*, *computeEventWeight()* e *computeSubRuleWeight()*

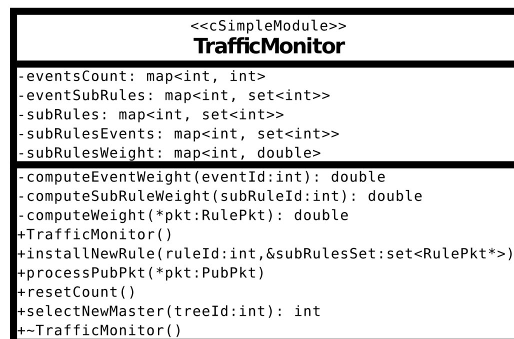


Figura 6.7: Classe TrafficMonitor

WorkloadHandler

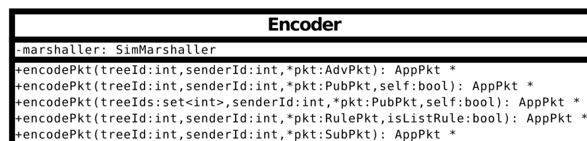
Il WorkloadHandler si occupa di fornire alle classi Publisher, Subscriber e RuleManager i dati riguardo gli advertisements, le sottoscrizioni e le regole da installare. Questa operazione viene compiuta sulla base degli attributi della classe, che rappresentano i parametri dell'ambito che si vuole simulare. Vengono impiegate a tal scopo le funzioni *setAdvertisements()*, *setSubscriptions()* e *setRule()*.
Un'ulteriore funzionalità viene esportata mediante *createPublication()* che si occupa della produzione degli eventi primitivi che verranno trasmessi dal Publisher attraverso la rete.

Figura 6.8: Classe WorkloadHandler

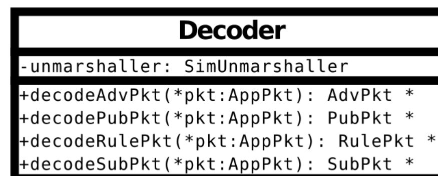
Encoder e Decoder

La prima classe (in Figura 6.9(a)) si occupa di generare un pacchetto di livello applicativo per la simulazione, a partire da un pacchetto della logica applicativa, ovvero dell'engine di riconoscimento. Effettuano questa operazione tramite la funzione *encodePkt()* disponibile per ogni tipologia di pacchetto. La seconda (in Figura 6.9(b)) effettua l'operazione contraria mediante le funzioni:

- *decodeAdvPkt()*;
- *decodePubPkt()*;
- *decodeRulePkt()*;
- *decodeSubPkt()*.



(a) Classe Encoder



(b) Classe Decoder

Figura 6.9: Classi Encoder e Decoder

Pacchetti del simulatore

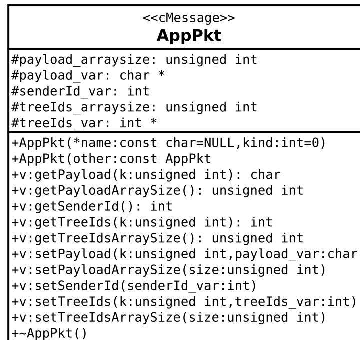
Vengono qui mostrati i campi che descrivono i pacchetti impiegati a livello della simulazione. Questi sono:

AppPkt rappresentano i pacchetti a livello applicativo in transito tra i nodi (Figura 6.10(a));

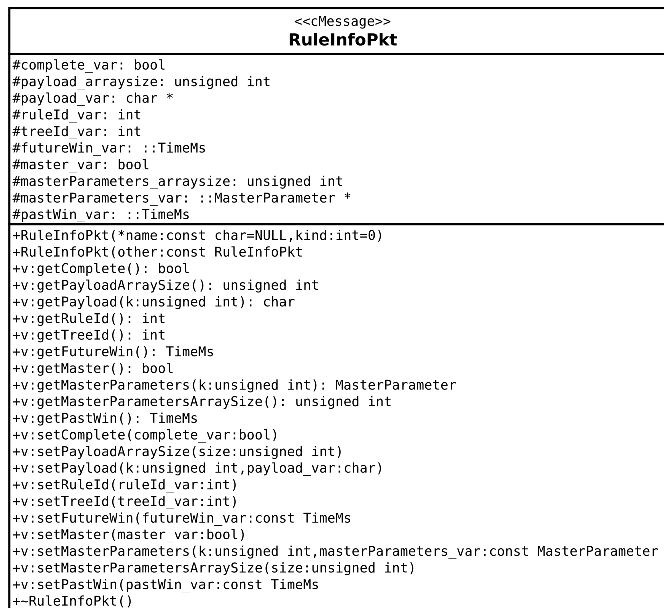
RuleInfoPkt rappresentano i pacchetti contenenti le informazini necessarie all'installazione dei una regola di riconoscimento(Figura 6.10(b));

TopologyManagerPkt rappresentano i pacchetti a livello di rete che viaggiano sui canali simulati d a Omnet++(Figura 6.10(c)).

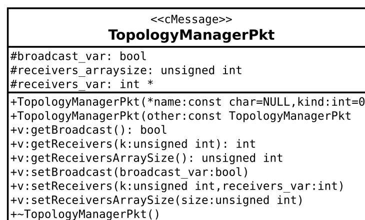
Altre due classi di pacchetti sono rappresentati da *NeighborDiscoveryPkt* e *TreeDiscoveryPkt* che descrivo i pacchetti di rete inviati in broadcast per determinare la topologia della stessa. Vengono impiegati durante la fase di start-up e sono mostrati in Figura 6.11.



(a) Classe AppPkt

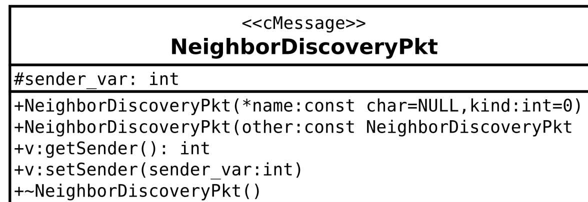


(b) Classe RuleInfoPkt

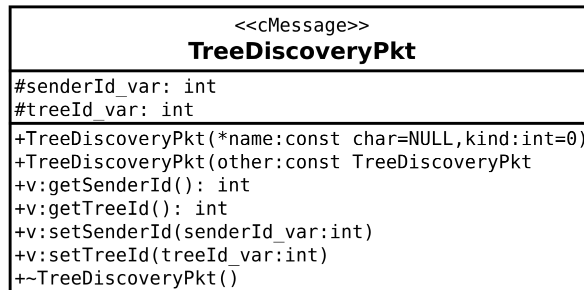


(c) Classe TopologyManagerPkt

Figura 6.10: Pacchetti principali impiegati dal simulatore



(a) Classe NeighborDiscoveryPkt



(b) Classe TreeDiscoveryPkt

Figura 6.11: Altri pacchetti impiegati dal simulatore

6.1.2 Diagrammi d'interazione

Vengono qui mostrati alcuni diagrammi UML atti a definire l'interazione dei moduli presentati nel Capitolo 6.1.1, durante le varie fasi di funzionamento del sistema.
Dapprima viene mostrata in Figura 6.12 l'interazione nel caso di generazione di una regola da parte di un RuleManager. Questi impiega la funzione *createAndSendRule()* per comunicare col WorkloadHandler e ricevere[1] la regola da installare. Successivamente, mediante un Encoder, genera il corrispondente pacchetto applicativo, che viene inoltrato al Processor[2]. Quest'ultimo, dopo aver decodificato il pacchetto, comunica con la logica applicativa. In questo caso, dopo aver elaborato la regola, verrà restituita una serie di pacchetti contenenti le sotto-regole da inviare ai propri figli. É mediante il Topology manager che questo avviene, una volta ricodificati i pacchetti come pacchetti applicativi della simulazione.

[1]Il fatto che il modulo interpellato ritorni un risultato al chiamante viene mostrato in figura tramite una freccia tratteggiata

[2]l'invio di messaggi viene identificato da una freccia più chiara

Successivamente (Figura 6.13) viene mostrato come avviene l'interazione tra i moduli del simulatore, per effettuare una pubblicazione di un evento primitivo da parte di un Publisher. Anche quest'ultimo, dopo aver comunicato col WorkloadHandler e ricevuto l'evento da pubblicare, invia al Processor il pacchetto del simulatore appositamente codificato tramite l'Encoder. Questi si pone da tramite tra la simulazione e la logica applicativa e ne riceve l'elenco di pacchetti da inoltrare verso la radice. Questo avviene esclusivamente se l'evento generato è terminatore per una regola installata nel nodo ed il suo arrivo porta all'individuazione di almeno una sequenza valida, impiegando gli eventi precedentemente notificati al nodo. Viene effettuato, come nel caso precedente, con l'ausilio del TopologyManager.
Viene invece mostrato in Figura 6.14 il diagramma d'interazione per la ricezione di un terminatore, per una regola installata nel nodo, proveniente da un altro nodo della rete. In questo caso il primo modulo che riceve il massaggio è il TopologyManager, che, dopo averne estrapolato il contenuto, lo inoltra al Processor. Come nel caso precedente, comunicando con le classi che implementano l'engine, riceve in risposta eventuali eventi primitivi componenti le sequenze valide e li inoltra, tramite il TopologyManager, verso i nodi di livello superiore.
L'ultimo diagramma d'interazione mostrato (in Figura 6.15) rappresenta la ricezione di un evento composito. In questa situazione il Processor, ricevuto l'evento, verifica se qualche Subscriber locale si sia ad esso sottoscritto, prima di ritrasmetterlo agli altri nodi. Se questa circostanza è verificata, il messaggio viene trasmesso anche al sottoscrittore interessato.

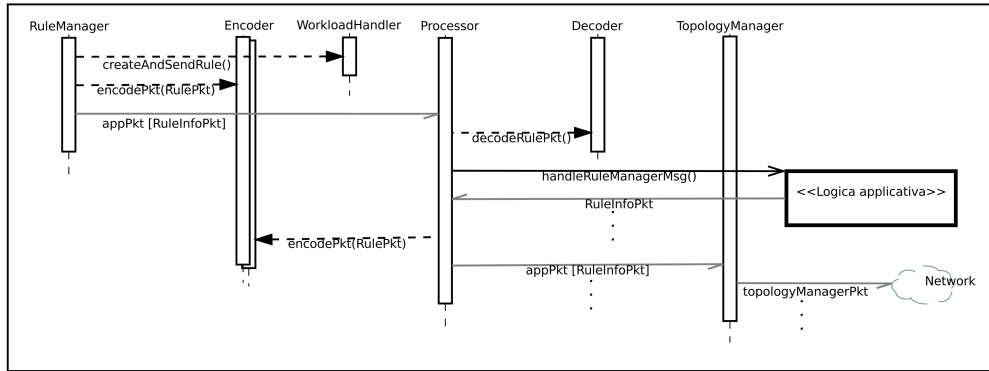


Figura 6.12: Generazione di una regola

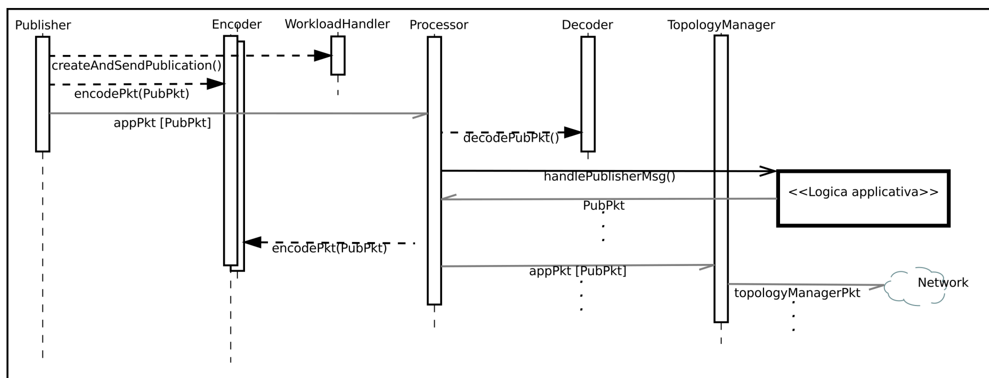


Figura 6.13: Pubblicazione di un evento

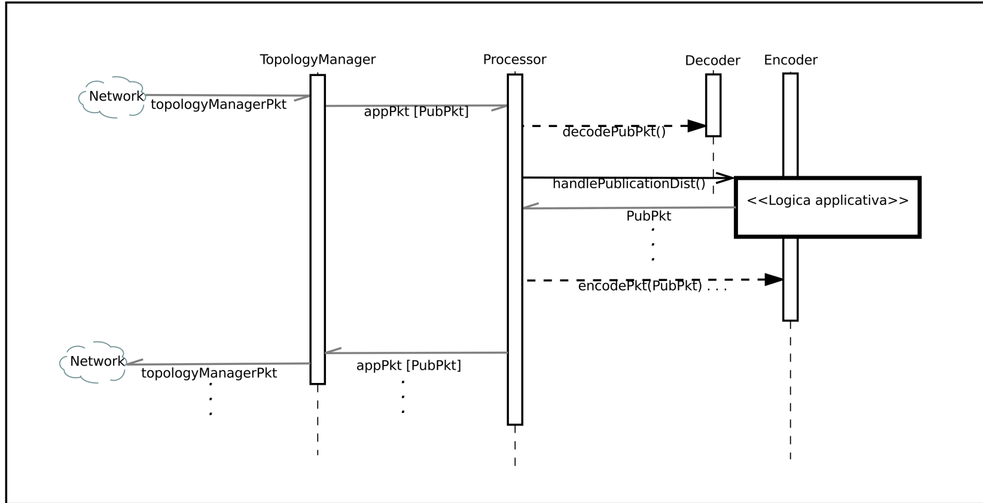


Figura 6.14: Ricezione di un terminatore

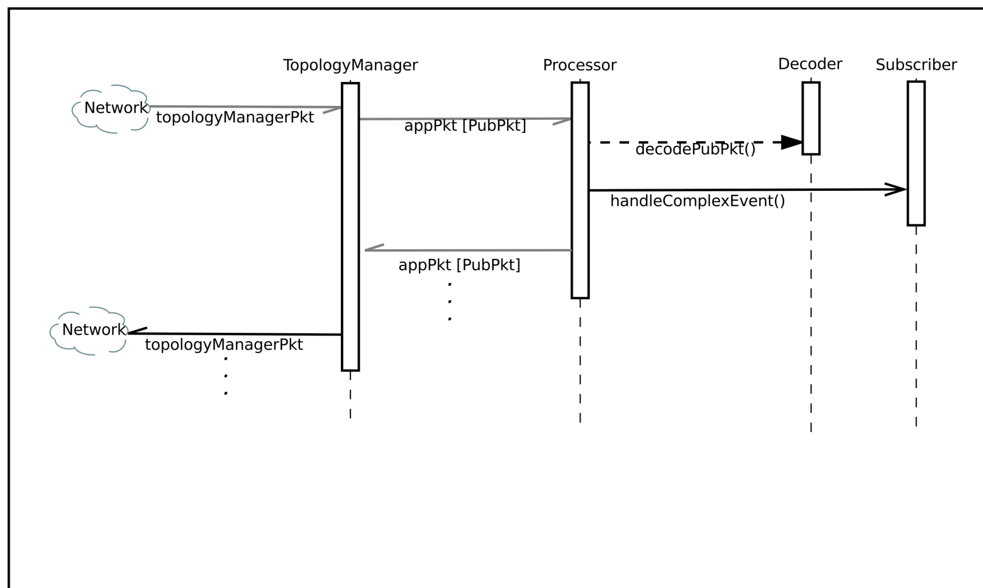


Figura 6.15: Ricezione di un evento composito

6.2 Architettura della logica applicativa

La logica applicativa di T-Rex e le classi che la compongono sono mostrate in Figura 6.16; ovvero l'engine di riconoscimento degli eventi compositi, a partire dalla ricezione di eventi semplici.
La classe che comunica con la simulazione è il RuleHandler specifico per la tipogia di algoritmo che si vuole simulare. In questo capitolo, come nel precedente, prendiamo in cosiderazione il caso distribuito. Questi comunica direttamente con l'IndexingTable, il RuleDeployer e il DupRemover.
Il primo tiene traccia delle regole e degli eventi che le compongono, considerando inoltre i vincoli imposti, così da poter effettuare anticipatamente una prima fase di filtraggio.
Il secondo è impiegato per individuare, sulla base degli advertisements, la partizione delle regole che interssano i nodi figli e, nel caso di protocollo push-pull, individuare le sotto-regole master e slave durante gli aggiornamenti.
Infine il DupRemover è incaricato di filtrare gli eventi appartenenti a sequenze valide, eliminandone i duplicati. Questo processo di elisione prende in considerazione i risultati provenienti da tutti gli StacksRule gestiti dal RuleHandler.
Quest'ultima classe viene istanziata dallo stesso RuleHandler ed implementa l'algoritmo di riconoscimento descritto nel Capitolo 3, memorizzando le informazioni riguardanti le pile tramite la classe Stack.

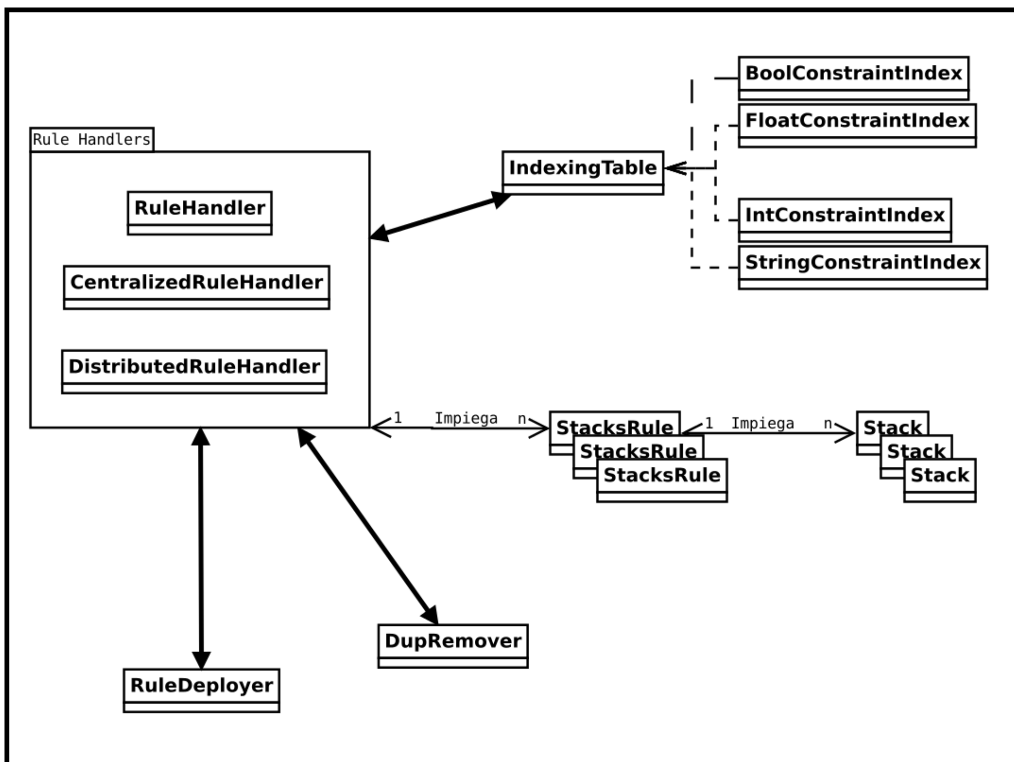


Figura 6.16: Architettura della logica applicativa

6.2.1 Diagrammi di struttura

Nei successivi paragrafi i diagrammi di struttura UML mostrati si riferiscono alle classi C++ che implementano la logica applicativa di T-Rex, ovvero l'engine. Verranno presentate funzionalità messe a disposizione e strutture dati usate.

Stack

La classe Stack tiene traccia degli attributi di una pila, ovvero della tipologia di evento associato (*kind*), delle negazioni direttamente correlate con la pila (*linkedNegations*), le pile di livello inferiore (*lookBackTo*) e quella di livello superiore (*refersTo*), direttamente connesse nel grafo d'ordine e la finestra temporale associata a quest'ultima relazione (*win*).

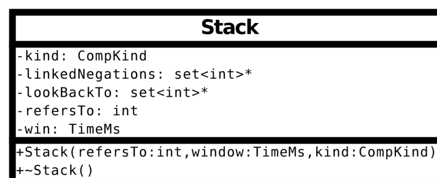


Figura 6.17: Classe Stack

IndexingTable

Questa classe presenta, suddivisi per tipologia, l'insieme dei vincoli presenti nelle regole installate:

- boolIndex;
- floatIndex;
- intIndex;
- stringIndex.

Memorizza inoltre la totalità dei predicati impiegati (*usedPreds*).
Le funzioni esibite sono *installRulePkt()* e *processMessage*; la prima è impiegata per completare le strutture dati descritte in precedenza, la seconda per

individuare quali pile degli StacksRule sono interessate alla pubblicazione passata come parametro.

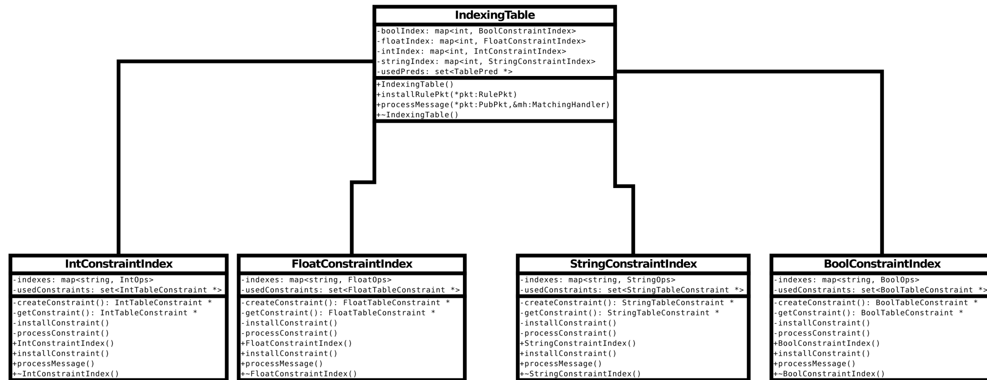


Figura 6.18: Classe IndexingTable

DistributedRuleHandler

La classe DistributedRuleHandler, per operare da interfaccia tra simulazione ed engine, possiede l'identificativo del nodo di rete, così come il riferimento alle altre classi: il *DupRemover*, l'*IndexingTable* e il *RuleDeployer*. Contiene inoltre delle strutture dati atte a memorizzare l'elenco delle regole complete (*completeRules*) e l'associazione tra le sotto-regole e gli StacksRule che le implementano (*rules*).
Espone le funzionalità per processare una pubblicazione, tramite *processPubPkt()*, e le regole ricevute, mediante *processRuleInfoPkt()*.

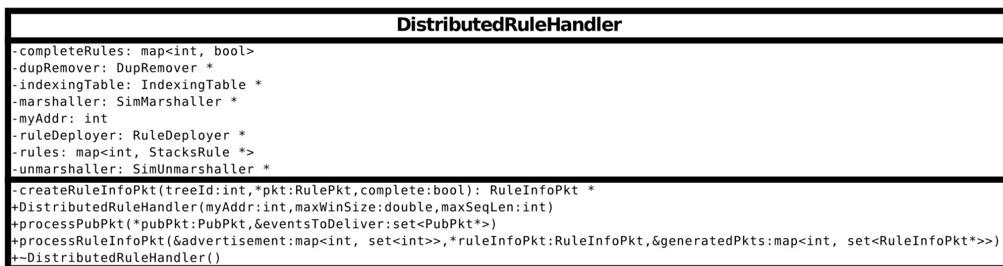


Figura 6.19: Classe DistributedRuleHandler

StacksRule

La classe StacksRule presenta strutture dati atte a memorizzare le informazioni necessarie alla gestione delle pile, che rappresentano le regole installate.

- Aggregati;
  - *aggregates*;
  - *aggrsNum*;
  - *aggsSize*;
  - *aggregateParameters*.
- Negazioni;
  - *negations*;
  - *negsNum*;
  - *negsSize*;
  - *negationParameters*.
- Predicati;
  - *stacks*;
  - *stacksNum*;
  - *stacksSize*.
- Parametri.
  - *branchStackParameters*;
  - *endStacksParameters*.

Mantiene le pile delle pubblicazioni ricevute suddivise per aggregati (*receivedAggs*), negazioni (*receivedNegs*) e predicati (*receivedPkts*). Inoltre memorizza i riferimenti tra pile (*referenceState*), l'identificativo della regola (*ruleId*), l'insieme degli eventi che devono essere consumati dal processo di

selezione delle sequenze valide (*consumingIndex*) e l'informazione riguardo la completezza della regola (*completeRule*).
Impiega le funzioni *addToAggregatesStack()*, *addToNegationStack()* e *addToStack()*, per memorizzare le pubblicazioni. *startComputation()*, chiamata con parametri differenti sulla base del protocollo impiegato dalla simulazione, è responsabile dell'individuazione di eventi compositi a partire dalla ricezione di un terminatore. Per svolgere questa mansione impiega le funzioni private tra cui le pricipali sono:

- *removeOldPacketsFromStack()*;
- *getPartialResults()*;
- *getWinEvents()*;
- *check Parameters()*;
- *checkNegation()*;
- *createComplexEvent()*.

```
StacksRule
-aggregateParameters: map<int, set<Parameter *>>
-aggregates: map<int, Aggregate *>
-aggrsNum: int
-aggsSize: int
-branchStackParameters: map<int, set<Parameter *>>
-completeRule: bool
-compositeEventId: int
-consumingIndexes: set<int>
-endStackParameters: set<Parameter *>
-eventGenerator: CompositeEventGenerator *
-negationParameters: map<int, set<Parameter *>>
-negations: map<int, Negation *>
-negsNum: int
-negsSize: int
-receivedAggs: map<int, vector<PubPkt *>>
-receivedNegs: map<int, vector<PubPkt *>>
-receivedPkts: map<int, vector<PubPkt *>>
-referenceState: map<int, int>
-ruleId: int
-stacks: map<int, Stack *>
-stacksNum: int
-stacksSize: int
-addAggregate(eventType:int,*constraints:Constraint,constrLen:int,lowIndex:int,&lowTime:TimeMs,highIndex:int,*name:char,&fun:AggregateFun)
-addNegation(eventType:int,*constraints:Constraint,constrLen:int,lowIndex:int,&lowTime:TimeMs,highIndex:int)
-addParameter(index1:int,*name1:char,index2:int,*name2:char,type:StateType,*pkt:RulePkt)
-checkNegation(negIndex:int,*partialResult:PartialEvent): bool
-checkParameter(*pkt:PubPkt,*partialEvent:PartialEvent,*parameter:Parameter): bool
-checkParameters(*pkt:PubPkt,*partialEvent:PartialEvent,&parameters:set<Parameter*>): bool
-clearStacks()
-createComplexEvents(*partialEvents:list<PartialEvent*>,&results:set<PubPkt*>)
-deletePartialEvents(*partialEvents:list<PartialEvent*>)
-fillResults(*partialEvents:list<PartialEvent*>,&results:set<PubPkt*>)
-fillResultsWithAggregates(*partialEvents:list<PartialEvent*>,&results:set<PubPkt*>)
-getPartialResults(*pkt:PubPkt,&stackAwkParameters:set<AwkParameter*>): list<PartialEvent\*>*
-getWinEvents(*partialEvents:list<PartialEvent*>,index:int,tsUp:TimeMs,mode:CompKind,*partialEvent:PartialEvent,&awkParameter:set<AwkParameter*>)
-parametricAddToStack(*pkt:PubPkt,&parStacksSize:int,&parReceived:vector<PubPkt*>)
-parametricStartComputation(*pkt:PubPkt,&results:set<PubPkt*>,&masterParameters:set<MasterParameter*>,&parameterValue:map<int, set<int>>,&awkParameters:set<AwkParameter*>)
-removeConsumedEvent(*partialEvents:list<PartialEvent*>)
-removeOldPacketsFromStack(&minTS:TimeMs,&parStacksSize:int,&parReceived:vector<PubPkt*>)
-removePartialEventsNotMatchingParameters(*partialEvents:list<PartialEvent*>,&parameters:set<Parameter*>)
+StacksRule(*pkt:RulePkt,complete:bool)
+addToAggregateStack(*pkt:PubPkt,index:int)
+addToNegationStack(*pkt:PubPkt,index:int)
+addToStack(*pkt:PubPkt,index:int)
+startComputation(*pkt:PubPkt,&results:set<PubPkt*>,&awkParameters:set<AwkParameter*>)
+startComputation(*pkt:PubPkt,&results:set<PubPkt*>,&masterParameters:set<MasterParameter*>,&parameterValue:map<int, set<int>>)
+~StacksRule()
```

Figura 6.20: Classe StacksRule

DupRemover

Il DupRemover possiede informazioni riguardo la quantità di pacchetti attesi della sequenze, memorizzate nelle variabili *maxSeqLen*, *maxWinSize* e *queueSize*. Durante il suo funzionamento conserva l'elenco dei pacchetti ricevuti e li memorizza in due strutture dati: *receivedPktsSet* e *receivedPktsList*.
La funzionalità di rimozione dei duplicati viene esibita tramite la funzione *removeDuplicates()*.

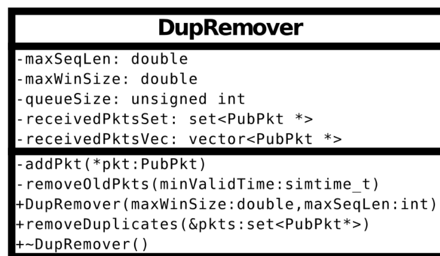


Figura 6.21: Classe DupRemover

RuleDeployer

La classe RuleDeployer presenta la funzione *getSubRules()* che, impiegando le altre funzioni private, permette il partizionamento della regola, seguendo quanto descritto nel Capitolo 4.2.1.

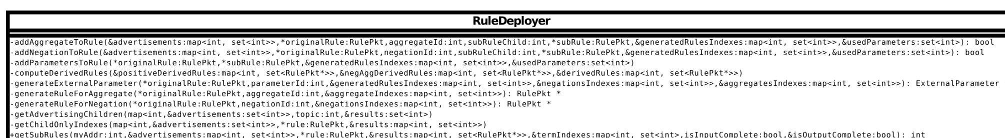


Figura 6.22: Classe RuleDeployer

6.2.2 Diagrammi d'interazione

Il primo diagramma presenta l'elaborazione di una regola. Il pacchetto contenente le informazioni riguardanti la regola viene ricevuto dal DistributedRuleHandler. Esso dapprima installa la regola nell'IndexingTable, successivamente crea un'istanza di StacksRule, che si incaricherà del riconoscimento

delle sequenze valide per la specifica regola. Quest'ultimo per svolgere la propria funzione istanzierà uno Stack per ogni evento componente la regola stessa. Infine il DistributedRuleHandler, impiegando il RuleDeployer, scompone la regola in sotto-regole e le inoltra alla simulazione. Questo processo avviene anche nel caso di sotto-regole, che verranno ulteriormente scomposte. La Figura 6.24 mostra invece l'interazione tra le classi nel caso di ricezione di un evento. In questo caso il DistributedRuleHandler, comunicando con l'IndexingTable, ottiene l'insieme degli StacksRule ed i rispettivi Stack interessati dall'evento e lo comunica loro. Lo StacksRule ne memorizza un riferimento nelle pile appropriate. Successivamente, nel caso in cui l'evento sia anche terminatore per la regola gestita dallo StacksRule, avvia la computazione dei risultati. Ques'ultima classe, impiegando le informazioni memorizzate negli Stack, li comunica al Distribuited RuleHandler, che a sua volta li rende disponibili al simulatore. Avviene precedentemente una fase di rimozione dei duplicati degli eventi in comune, anche trasversalmente tra regole differenti associate allo stesso DistributedRuleHandler, grazie all'impiego del DupRemover.

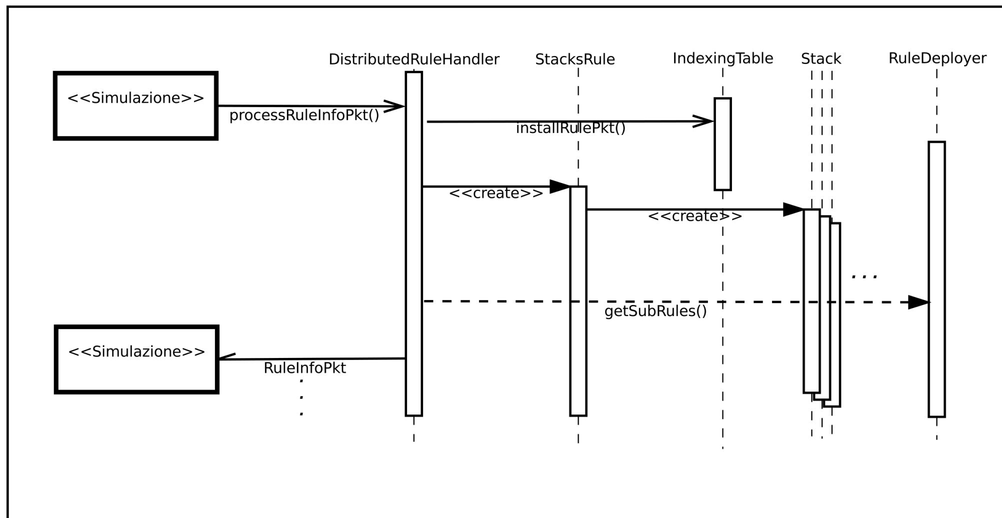


Figura 6.23: Elaborazione di una regola

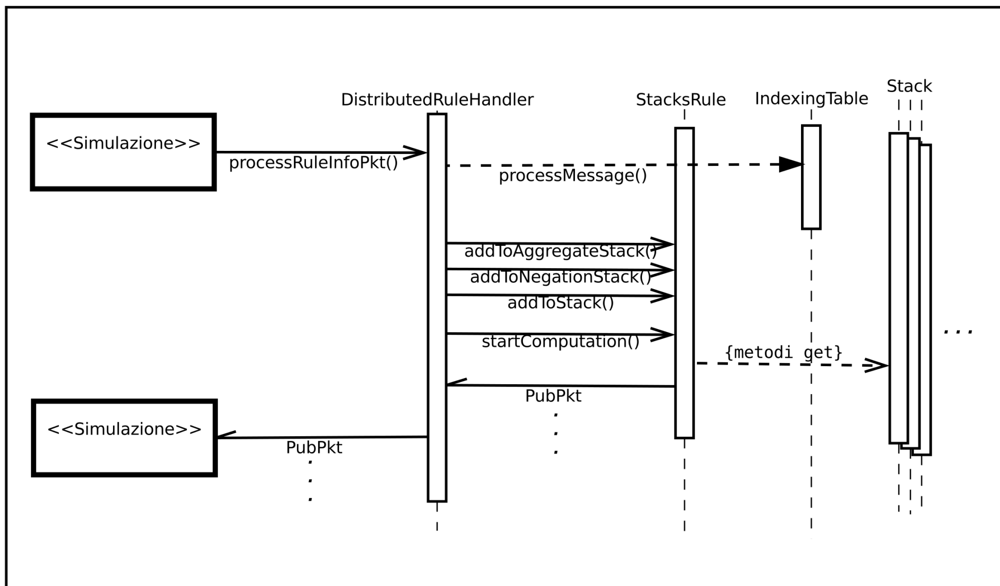


Figura 6.24: Elaborazione di un evento

6.3 Casi d'uso

Vengono presentati in questo capitolo alcuni casi d'uso, allo scopo di descrivere la sequenza di operazioni effettuate tra i vari nodi durante le principali operazioni compiute da questi ultimi, per l'implementazione dell'algoritmo T-Rex.
Il primo diagramma delle attività, mostrato in Figura 6.25, si riferisce all'installazione di una regola all'interno di una rete. Viene mostrato il processo, dalla pubblicazione della regola, da parte del RuleManager, all'invio della sottoregole generate agli altri nodi di livello inferiore. Vengono suddivise le attività tra il simulatore e l'engine di riconoscimento.
Il secondo diagramma delle attività, presente in Figura 6.26, presenta la gestione delle pubblicazioni degli eventi. Il processo è seguito dalla pubblicazione dell'evento, da parte del generatore dello stesso, all'invio dell'evento composito individuato. Vengono suddivise le attività tra il simulatore e l'engine di riconoscimento.

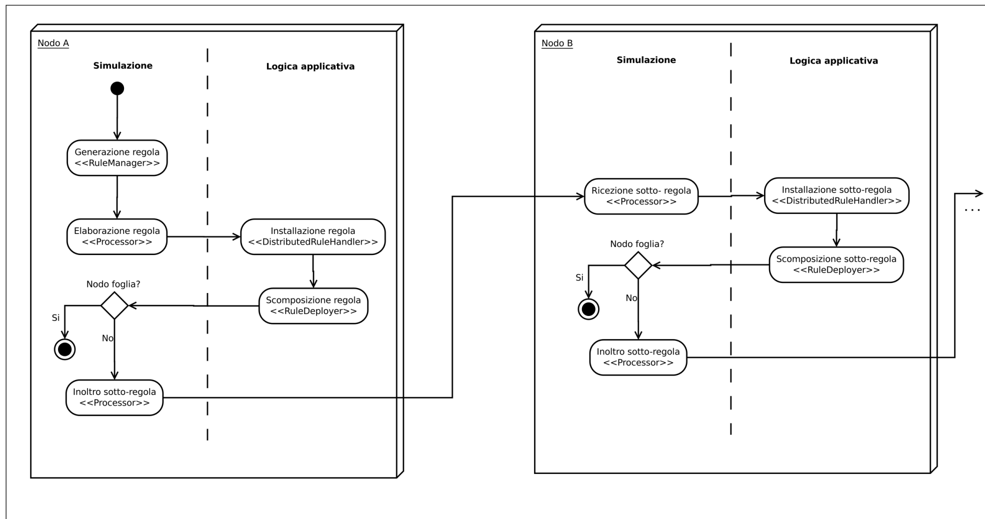


Figura 6.25: Installazione di una regola

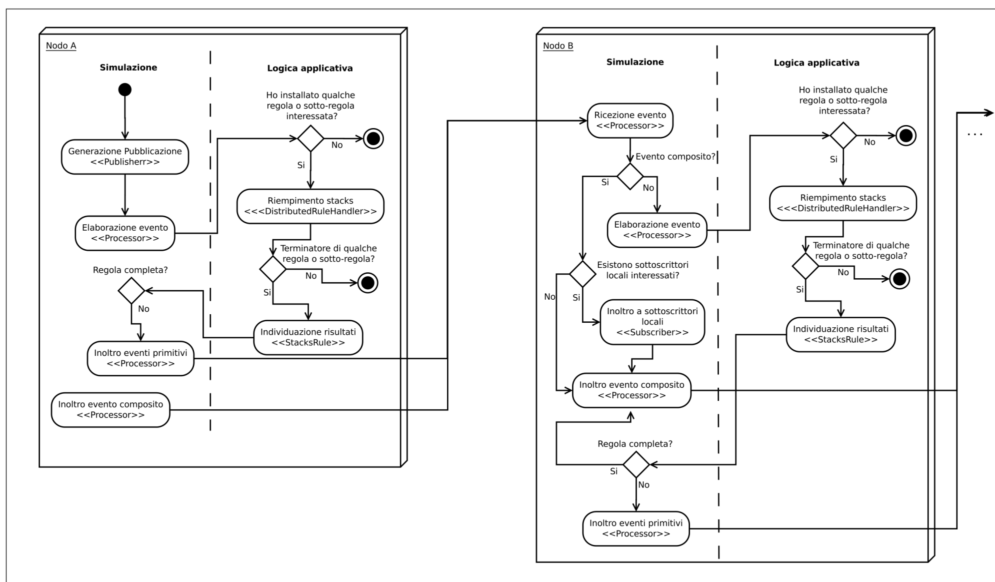


Figura 6.26: Gestione di una pubblicazione

Capitolo 7

Simulazioni effettuate

In questo capitolo viene mostrata la fase di esecuzione delle simulazioni. Viene inizialmente descritto, nel Capitolo 7.1, lo scenario di riferimento, a partire dal quale le simulazioni sono state eseguite. Successivamente nel Capitolo 7.2 viene fatto un paragone con i risultati ottenuti con l'algoritmo degli automi. Viene in seguito (Capitolo 7.3) descritto il metodo di simulazione dei parametri. Questi, accompagnati dai risultati ottenuti mediante simulazione, sono mostrati nel Capitolo 7.4

7.1 Scenario di riferimento

Tutti i risultati presentati successivamente sono stati collezionati utilizzando una piattaforma hardware dotata di processore AMD Phenom a 2.8GHz, con 6 cores e 8GB di RAM DDR3, impiegando Linux come sistema operativo. I singoli test, effettuati per ogni parametro che abbiamo voluto simulare, sono stati ripetuti 10 volte con *seed* differenti per il generatore di numeri casuali e sono state riportate le medie dei valori riscontrati.
La piattaforma hardware impiegata, nonostante il sistema venga simulato mediante un simulatore ad eventi discreti, risulta essere fondamentale per analizzare i risultati delle simulazioni. Il processo di filtraggio e *matching*, infatti, non viene semplicemente simulato dal nostro impianto simulativo, bensì effettuato effettivamente a run-time. La prima conseguenza di questa

scelta è che la piattaforma hardware influenza direttamente le performances del sistema, dal punto di vista dei tempi di elaborazione. In secondo luogo la scelta di compiere effettivamente il *processing* consente di misurare ed impiegare nella simulazione dei tempi effettivi, e non semplicemente stimati, per ritardare l'inoltro dei risultati all'atto della ricezione di un terminatore, col vantaggio di ottenere una confidenza maggiore sui dati rappresentati nei grafici. Un'ulteriore conseguenza di questa scelta è quella di ottenere un sistema che risulta essere facilmente impiegabile in uno scenario reale, dal momento che la logica applicativa è stata effettivamente realizzata.
É stato individuato, dunque, un insieme di valori di riferimento per i parametri della simulazione, che permettono il confronto dei risultati ottenuti. Lo scenario risulta costituito da una rete contenente 20 nodi, 200 tipologie differenti di eventi primitivi e 150 eventi compositi disponibili per i RuleManager. Ogni nodo, tramite i client ad esso direttamente connessi, pubblica 20 advertisements (e di conseguenza genera 20 tipologie di eventi primitivi), e 15 sottoscrizioni ad eventi compositi. La località delle pubblicazioni, ovvero probabilità che eventi primitivi dello stesso tipo provengano dallo stesso nodo della rete, è massima. Il rate medio delle pubblicazioni da parte del singolo generatore di eventi primitivi (Il Publisher) è di 5 eventi per secondo. Le regole sono generate casualmente sulla base dei seguenti parametri: sono costituite da sequenze di 3 eventi primitivi, le finestre temporali delle singole pile sono in media di 60 secondi, non presentano aggregati e negazioni e tutti i vincoli posseggono un singolo valore possibile. Infine la politica di selezione degli eventi è equamente distribuita tra *first-within* e *last-within*[1].
Il risultato di generare advertisements, regole, sottoscrizioni e pubblicazioni che rispettino i valori imposti dalla configurazione dei parametri è ottenuto mediante l'impiego del WorkloadHandler. Questo modulo infatti, sulla base dei parametri del simulatore, ha il compito di generare regole e sottoscrizioni che impieghino gli eventi primitivi e le stesse regole prodotte in maniera uniforme rispetto alle risorse a disposizione. Altro ruolo fondamentale è quello di generare gli advertisements dei vari nodi sulla base del parametro di località, introdotto per valutare quanto la prossimità topologica influenzi le

[1] nessun evento viene selezionato con un politica *each-within*.

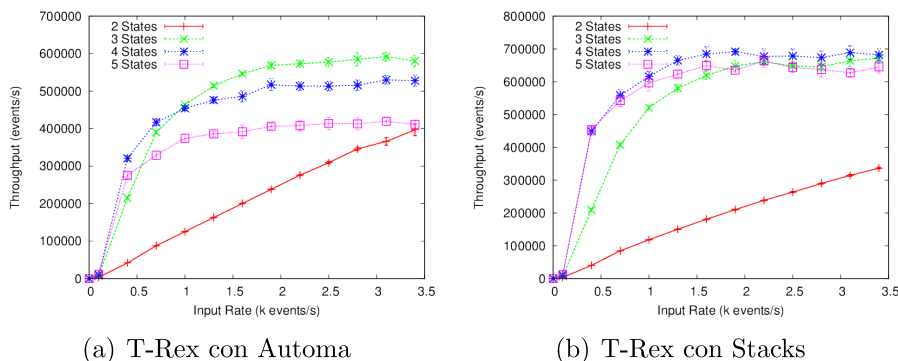
prestazioni del sistema. Infine si occupa di generare il traffico dei Publisher alternando le tipologie di evento in maniera opportuna, così che la frequenza di inoltro della sorgente rispetti i vincoli della simulazione ed i singoli eventi siano equiprobabili. Si ottiene, perciò, un traffico di rete che presenta delle caratteristiche costanti e permette, dunque, il confronto dei valori delle singole simulazioni.

7.2 Confronto con gli automi

Oltre a confrontare tra di loro i vari protocolli mediante Omnet++, l'algoritmo di riconoscimento che fa uso degli stacks (TRex2 in figura) è stato valutato in relazione alla versione che impiega gli automi [12] (TRex in figura). In questa sezione viene brevemente descritto lo scenario di rifermento impiegato per le simulazioni e vengono mostrati i risultati di questo confronto.
Tutti i risultati sono stati ottenuti impiegando lo stesso hardware descritto nel Capitolo 7.1. Si è fatto uso di client locali per generare gli eventi primitivi ad una frequenza costante, così da trascurare l'impatto dello strato di comunicazione e confrontare i due algoritmi di riconoscimento. Sono state sottodimensionate le code di ingresso così da amplificare gli effetti di una differenza di tempo di elaborazione, tramite una perdita di pacchetti. Ogni test è stato effettuato 10 volte e sono stati utilizzati i valori medi dei risultati ottenuti per produrre i grafici.
Vengono confrontate le prestazioni del filtraggio degli eventi, in presenza delle differenti politiche di selezione, al variare del numero degli stati, ovvero della lunghezza delle sequenze delle regole.

each-within

Nel caso di T-Rex con automi (Figura 7.1(a)), il troughput massimo del sistema si assesta intorno ai 600000 eventi/s e si riscontra una perdita di pacchetti anche per il caso con solo 2 stati componenti la regola. Nel caso di T-Rex con stacks (Figura 7.1(b)), invece, il sistema riesce a gestire fino a 700000 eventi/s e, nel caso più semplice di 2 stati per regola, non si sperimentano sensibili perdite di pacchetti anche a frequenza di invio di 3500 eventi/s. In Figura 7.2 viene confrontato il tempo di elaborazione medio. In entrambi i casi questo cresce al crescere del numero di stati o stacks, ma in quest'ultimo caso risulta essere dimezzato.



(a) T-Rex con Automa (b) T-Rex con Stacks

Figura 7.1: Througput a confronto nel caso *each-within*

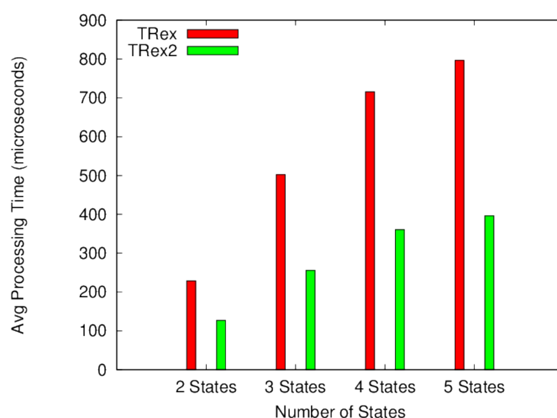
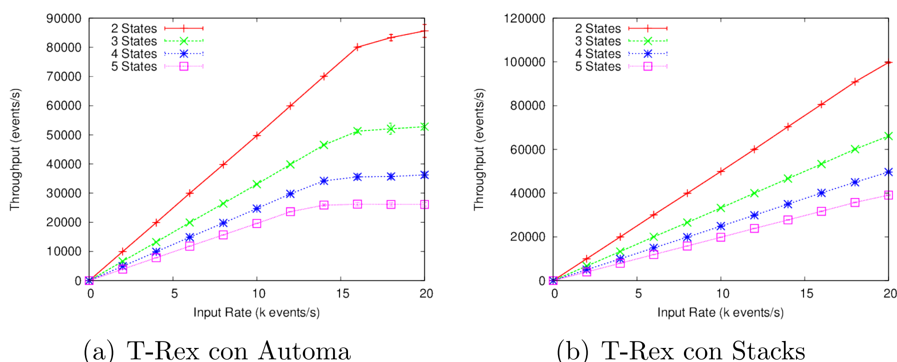


Figura 7.2: Tempo di elaborazione a confronto nel caso *each-within*

last-within

Nel caso di T-Rex con automi (Figura 7.3(a)) il troughput massimo del sistema si assesta intorno ai 85000 eventi/s e si riscontra una perdita di pacchetti anche per il caso con solo 2 stati componenti la regola, ma cresce linearmente fino ad una frequenza di inoltro di 15000 eventi/s. Nel caso di T-Rex con stacks (Figura 7.3(b)), invece, il sistema riesce a gestire fino a 100000 eventi/s ed in tutti i casi testati (sequenze di lunghezza da 2 a 5) non si sperimentano sensibili perdite di pacchetti per frequenza di invio fino a 20000 eventi/s. In Figura 7.4, infine, viene confrontato il tempo di elaborazione medio. Nel caso con gli stacks questo è sensibilmente minore e resta costante al crescere della lunghezza della sequenza.



(a) T-Rex con Automa (b) T-Rex con Stacks

Figura 7.3: Througput a confronto nel caso *last-within*

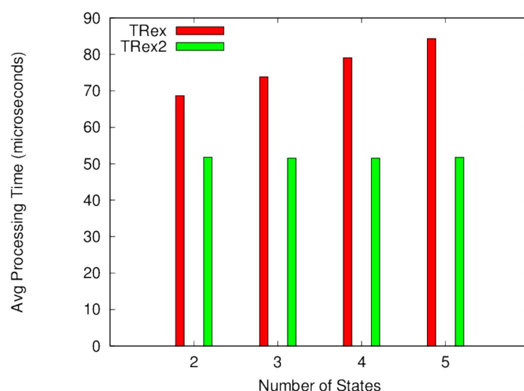
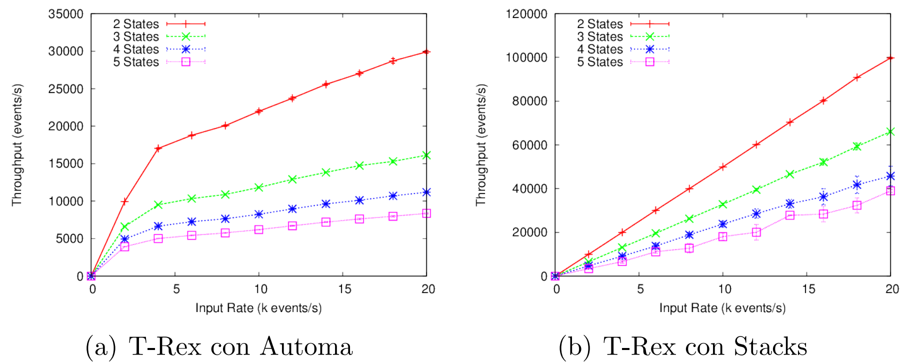


Figura 7.4: Tempo di elaborazione a confronto nel caso *last-within*

first-within

Nel caso di T-Rex con automi (Figura 7.3(a)) il troughput massimo del sistema si assesta intorno ai 30000 eventi/s e si riscontra una perdita di pacchetti anche per il caso con solo 2 stati componenti la regola, a partire da una frequenza di inoltro di circa 4000 eventi/s. Nel caso di T-Rex con stacks (Figura 7.3(b)), invece, il sistema riesce a gestire fino a 100000 eventi/s e, nei casi con sequenze di lunghezza da 2 a 4, non si sperimentano sensibili perdite di pacchetti per frequenza di invio fino a 20000 eventi/s. Un comportamento non perfettamente lineare si riscontra esclusivamente nel caso di sequenze lunghe 5 eventi. In Figura 7.4 viene confrontato il tempo di elaborazione medio. Nel caso con gli stacks questo è da 4 a 6 volte minore e resta costante al crescere della lunghezza della sequenza.



(a) T-Rex con Automa (b) T-Rex con Stacks

Figura 7.5: Througput a confronto nel caso *first-within*

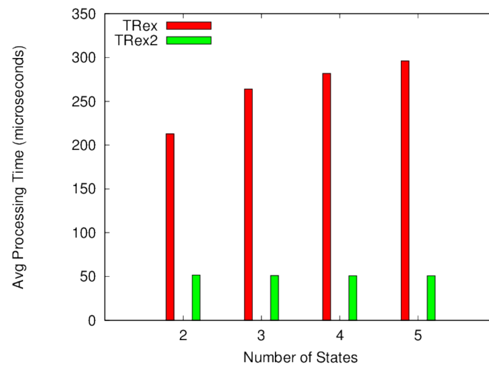


Figura 7.6: Tempo di elaborazione a confronto nel caso *first-within*

7.3 Esecuzione delle simulazioni

Per effettuare le simulazioni abbiamo preso in considerazione le cinque differenti implementazioni del protocollo presentate nel Capitolo 4.5:

- centralizzato;
- distribuito ad albero singolo;
- distribuito ad albero multiplo;
- push-pull ad albero singolo;
- push-pull ad albero multiplo.

Il protocollo centralizzato possiede un solo SPT ed un solo nodo che installa tutte le regole (la radice del SPT individuato nella fase di start-up), le quali non vengono scomposte in sotto-regole. Gli eventi primitivi vengono trasmessi fino alla radice e solamente a questo livello avviene l'elaborazione, che individua quelli adatti a generare l'evento composito definito dalla regola. Questo procollo serve come riferimento per ottenre un confronto con un possibile CEP sistem non distribuito. Le altre due topologie (distribuito e push-pull) differiscono esclusivamente per la presenza nell'ultima delle due del meccanismo che si occupa di suddividere le sotto regole tra regole master e regole slave. Si vuole valutare, mediante il loro confronto, quanto il maggior carico computazionale introdotto dal sistema di *load-balancing* pesi sul sistema, rispetto al vantaggio stesso di bilanciare il traffico degli eventi semplici che fluiscono nella rete. Entrambi questi protocolli, inoltre, saranno simulati nella loro versione ad albero singolo ed ad albero multiplo, esplorando vantaggi e svantaggi del rendere distribuita non solo l'elaborazione degli eventi compositi, ma anche la scomposizione delle regole che li definiscono. Nei grafici prodotti, i valori dell'algoritmo centralizzato è etichettato *Centralized*, quelli ad albero singolo rispettivamente *DistributedTree* e *Push-PullTree* ed infine quelli ad alberi multipli *Distributed* e *Push-Pull*.

7.4 Parametri simulati e risultati

A partire dalle cinque tipologie di protocollo e da uno scenario di riferimento verranno fatti variare diversi parametri che permetteranno di analizzare le performances di T-Rex in differenti ambiti applicativi, individuandone punti di forza e debolezze.
I pincipali parametri che abbiamo valutato sono:

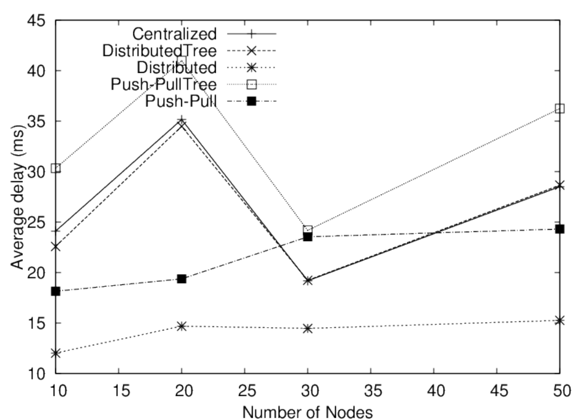
- numero dei nodi della rete;
- numero delle regole;
- numero degli eventi primitivi;
- frequenza di pubblicazione (*rate*);
- numero di valori validi per ogni attributo;
- dimensioni delle regole;
- numero di sottoscrizioni;
- località degli eventi.

Oltre a questi è stato, inoltre, considerato l'impatto di altri parametri sulle performances del sistema. Qesti ultimi sono mostrati nel Capitolo 7.4.9.

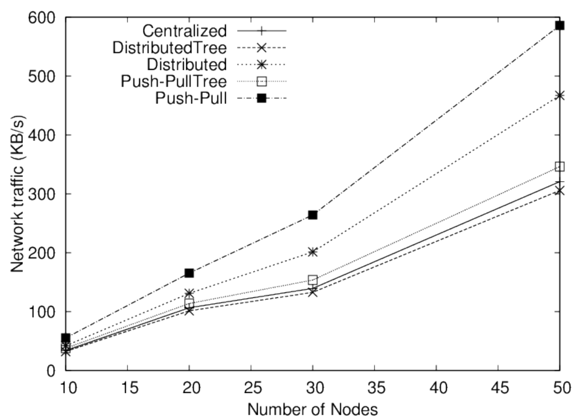
7.4.1 Numero di nodi

Il numero di nodi (*Number of Nodes* in figura) influenza direttamente, sia le pile impiegate all'interno della rete, sia i messaggi in transito. Al crescere del numero di nodi, come visibile in Figura 7.7(b), il traffico generato aumenta in maniera direttamente proporzionale. Per quanto riguarda la latenza è visibile in Figura 7.7(a) un sensibile peggioramento delle prestazioni per un numero di 20 nodi nei casi ad albero singolo e nel caso centralizzato. Questo è dovuto al fatto che aumenta il numero di nodi e di conseguenza degli stacks istanziati, ma viene mantenuto fisso il numero di regole installate nella rete. Dopo un iniziale aumento di occupazione delle risorse dei singoli nodi, che

ha il suo massimo per 20 nodi, si ha esperienza di un riuso efficiente delle pile lungo la rete, dal momento che il singolo nodo genererà risultati validi per più sottoscrittori. Questo effetto si può notare nel caso di rete con 30 nodi. Successivamente le prestazioni subiscono nuovamente un peggioramento dovuto alla graduale saturazione delle risorse. Non è visibile, nel caso ad albero multiplo, che differenzia l'installazione delle regole in un nodo, sulla base della provenienza delle sottoscrizioni.



(a) Latenza (*Average delay*)

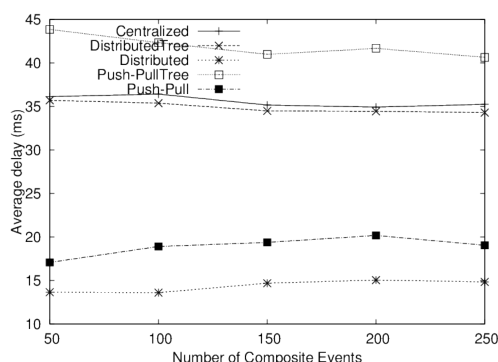


(b) Traffico di rete (*Network Traffic*)

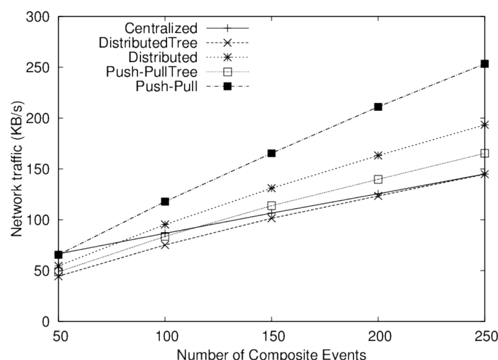
Figura 7.7: Prestazioni al variare del numero di nodi

7.4.2 Numero delle regole

Il numero di regole installate nella rete influenza direttamente la quantità di stack istanziati in ogni nodo della rete e, all'arrivo di un nuovo evento, la probabilità che questo sia un terminatore per una regola e avvii l'elaborazione degli eventi compositi (*Number of Composite Events* in figura). Questo parametro è rappresentato dal numero di eventi compositi disponibili. Influenza maggiormente i procolli ad albero multiplo, dal momento che in questo caso la singola regola viene installata in un nodo una volta per ogni sottoscrizione. All'aumentare del numero di regole, come mostrato in Figura 7.8(b), aumenta il traffico in transito all'interno della rete, dal momento che una quantità maggiore di permutazioni degli eventi primitivi rappresenterà sequenze valide e, di conseguenza, le invierà verso la radice.



(a) Latenza (*Average delay*)

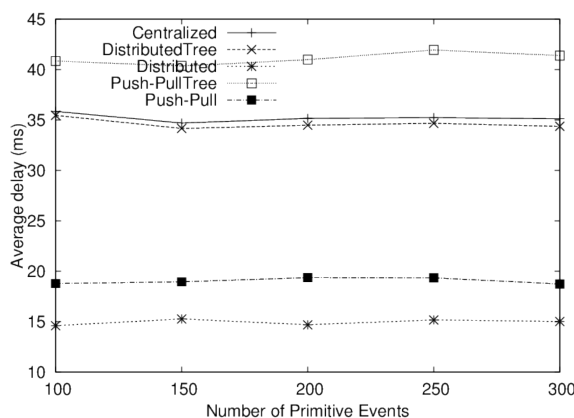


(b) Traffico di rete (*Network Traffic*)

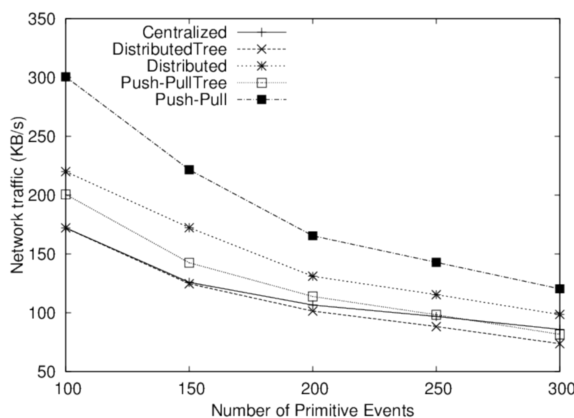
Figura 7.8: Prestazioni al variare del numero delle regole

7.4.3 Numero di eventi primitivi

Il numero di eventi primitivi (*Number of Primitive Events* in figura) rappresenta le differenti tipologie degli stessi, tra i quali i Publisher si possono scegliere in fase di pubblicazione. Al loro aumentare, come visibile in Figura 7.9(b), diminuisce il traffico di rete, dal momento che diminuisce la probabilità che l'evento generato sia terminatore per una regola. Questo riduce il numero di computazioni avviate dall'algoritmo di riconoscimento nei vari nodi della rete ed il successivo inoltro di messaggi.



(a) Latenza (*Average delay*)

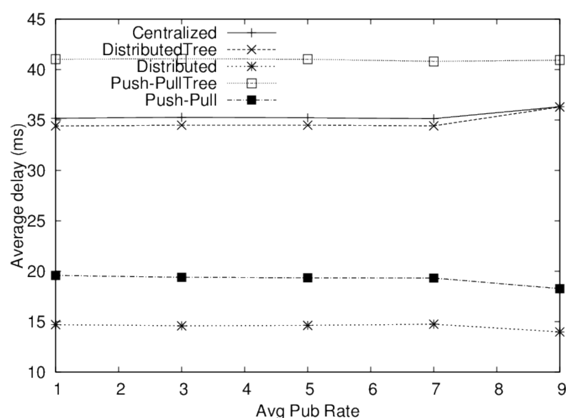


(b) Traffico di rete (*Network Traffic*)

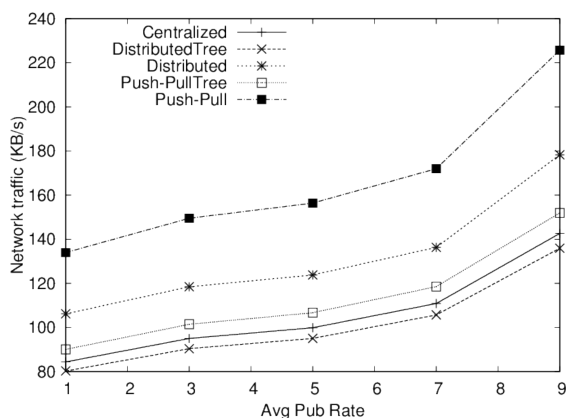
Figura 7.9: Prestazioni al variare del numero di eventi primitivi

7.4.4 Frequenza delle pubblicazioni

La frequenza delle pubblicazioni (*Avg Pub Rate* in figura) è espressa come numero medio di eventi primitivi al secondo, pubblicati dai client (i Publisher) della rete. L'aumentare di questo parametro comporta un maggior traffico di pacchetti in transito, come mostrato dalla Figura 7.10(b), dal momento che aumenta la probabilità di individuare, durante la fase di elaborazione, un numero maggiore di eventi all'interno delle finestre temporali. Le singole sotto-regole generano, dunque, un numero maggiore di sequenze valide e ciò influenza direttamente il numero di eventi da inoltrare verso la radice.



(a) Latenza (*Average delay*)

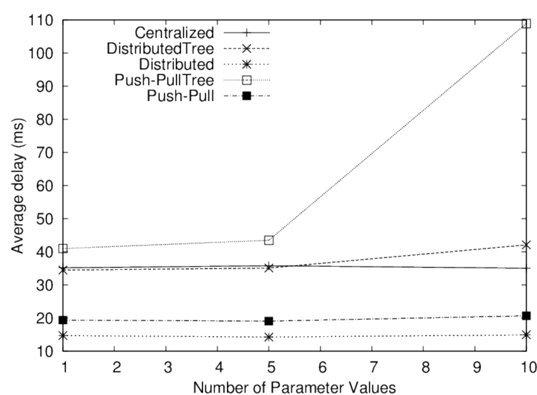


(b) Traffico di rete (*Network Traffic*)

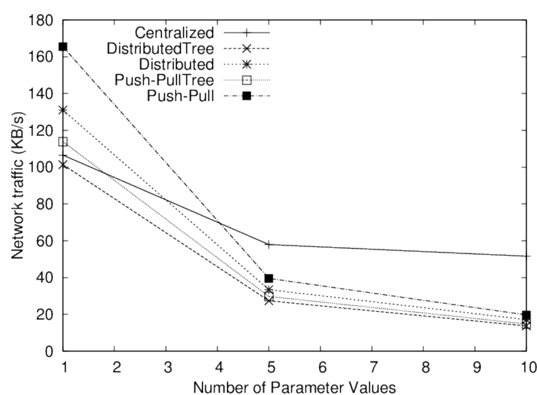
Figura 7.10: Prestazioni al variare del *rate* di pubblicazione

7.4.5 Numero di valori validi per ogni attributo

Il numero di valori validi per attributo (*Number of Parameters Values* in figura) rappresenta il dominio al quale appartengono gli attributi stessi. Al crescere di questo parametro diminuisce la probabilità per il singolo evento di essere valido per una regola. Dal momento che un evento non valido, a causa del valore dei suoi attributi, non viene memorizzato nello stack corrispondente, questo parametro influenza direttamente la quantità di risultati individuati ad ogni passo dell'elaborazione e la dimensione delle pile. Il suo crescere determina, di conseguenza, una diminuzione del traffico in transito nella rete visibile in Figura 7.11(b).



(a) Latenza (*Average delay*)

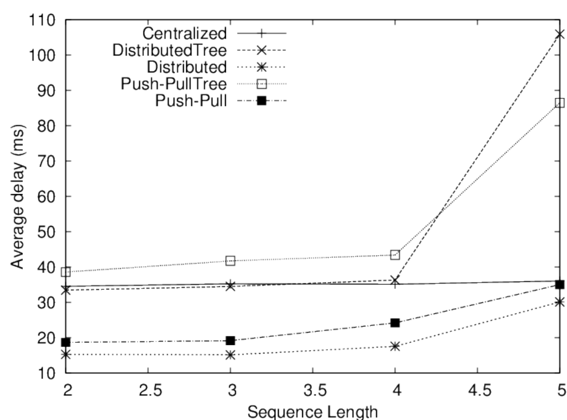


(b) Traffico di rete (*Network Traffic*)

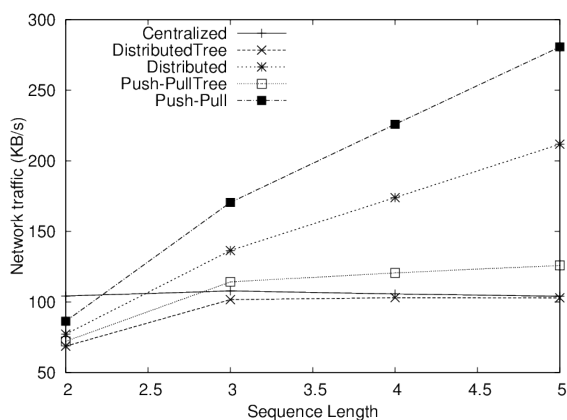
Figura 7.11: Prestazioni al variare del numero di valori per attributo

7.4.6 Dimensione delle regole

La dimensione delle regole, determinata dalla lughezza delle sequenze da riconoscere, è mostrata in Figura 7.12 ed identificata come *Sequence Length*. Rappresenta la lunghezza del percorso dalla radice alla foglia in una regola e determina direttamente il numero di passi che vengono effettuati da T-Rex. In base alle dimensioni delle regole, inoltre, gli algoritmi ad albero multiplo che implementano T-Rex, scompongono in maniera efficiente la regola in sotto-regole di dimensioni minori rispetto al caso centralizzato, mostrando una latenza minore (visibile in Figura 7.12(a)).



(a) Latenza (*Average delay*)

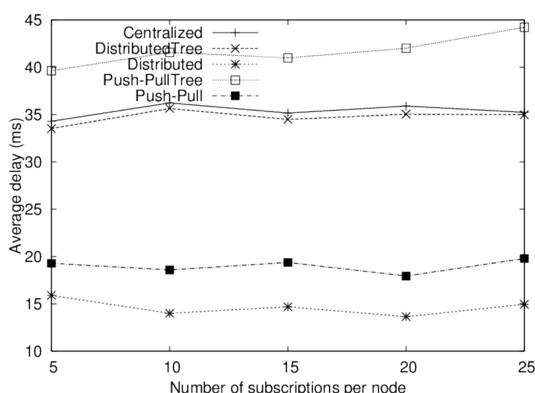


(b) Traffico di rete (*Network Traffic*)

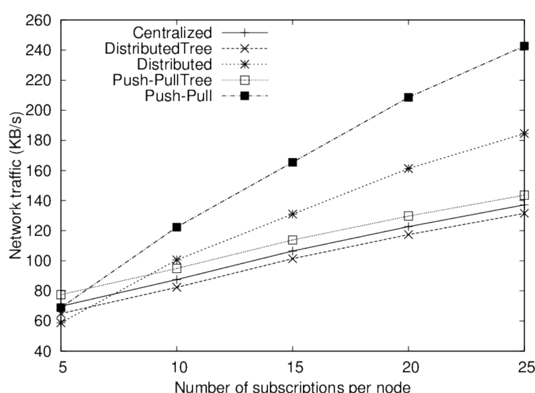
Figura 7.12: Prestazioni al variare della lunghezza delle sequenze

7.4.7 Numero di sottoscrizioni

Il numero delle sottoscrizioni, mostrato come *Numeber of subscriptions per node* in Figura 7.13, è definito dal numero di eventi compositi, tra i quali i client (i Subscriber) possono scegliere. All'aumentare di questo parametro aumenta il traffico di rete generato, come mostrato in Figura 7.13(b), dal momento che aumenta il numero di eventi riconosciuti che dovranno essere inoltrati ai sottoscrittori. Il singolo evento composito, infatti, dovrà essere notificato a più sottoscrittori. Nel caso ad albero multiplo, inoltre, i nodi topologicamente vicini ai sottoscrittori duplicheranno le sottoregole, con conseguente incremento del traffico, ottenendo altresì una latenza nettamente inferiore (come mostrato dalla Figura 7.13(a)).



(a) Latenza (*Average delay*)

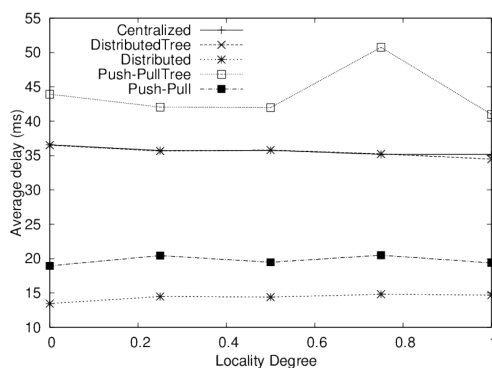


(b) Traffico di rete (*Network Traffic*)

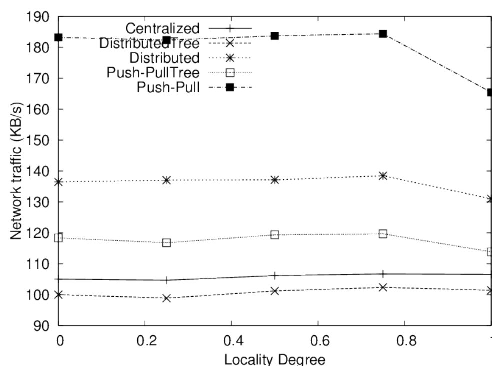
Figura 7.13: Prestazioni al variare del numero di sottoscrizioni

7.4.8 Località degli eventi

La località dagli eventi (*Locality degree* in figura) rappresenta la vicinanza topologica dei nodi che presentano *advertisements* per lo stesso tipo di evento primitivo e, di conseguenza, anche quella delle pubblicazioni di una stessa tipologia di eventi. Una località maggiore, che indica una maggiore vicinanza topologica, porta a performances migliori, soprattutto negli algoritmi ad albero multiplo. Un maggiore località porta, infatti, ad una maggiore scomposizione delle regole e di conseguenza ad un filtraggio degli eventi più efficiente. Nel caso di alberi multipli, inoltre, la maggiore scomposizione e localizzazione delle sotto-regole porta ad una minore duplicazione delle stesse, nei nodi centrali rispetto al SPT di più sottoscrizioni. Questo è visibile, per valori di località alti nella Figura 7.14.



(a) Latenza (*Average delay*)



(b) Traffico di rete (*Network Traffic*)

Figura 7.14: Prestazioni al variare della località degli eventi

7.4.9 Altri parametri

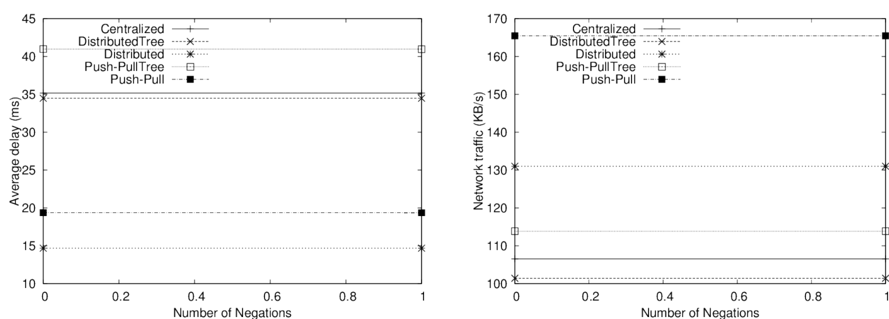
Gli altri parametri che sono stati valutati non influenzano le prestazioni del sistema in termini di traffico di rete e di Latenza e sono mostrati nel seguito per completezza.
Questi sono:

Numero di negazioni fatto variare da 0 a 1 (Figura 7.15)

Numero di aggregati fatto variare da 0 a 1 (Figura 7.16)

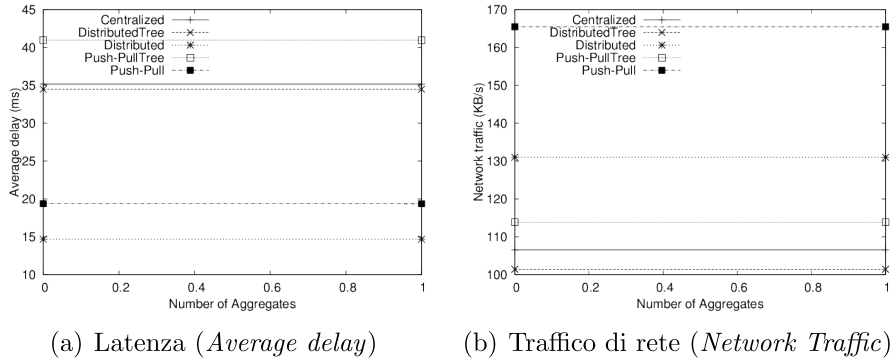
Dimensione delle finestre temporali fatte variare da 30 secondi a 120 secondi (Figura 7.17)

Percentuale di *each-eithin* fatta variare da 0 a 30% (Figura 7.18)



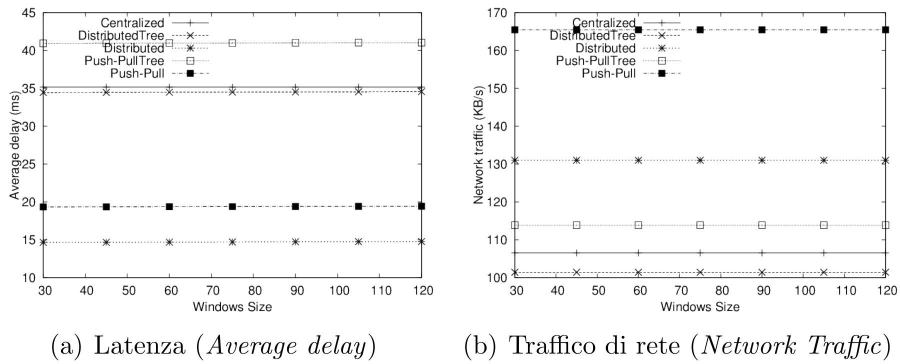
(a) Latenza (*Average delay*) (b) Traffico di rete (*Network Traffic*)

Figura 7.15: Prestazioni al variare del numero di negazioni



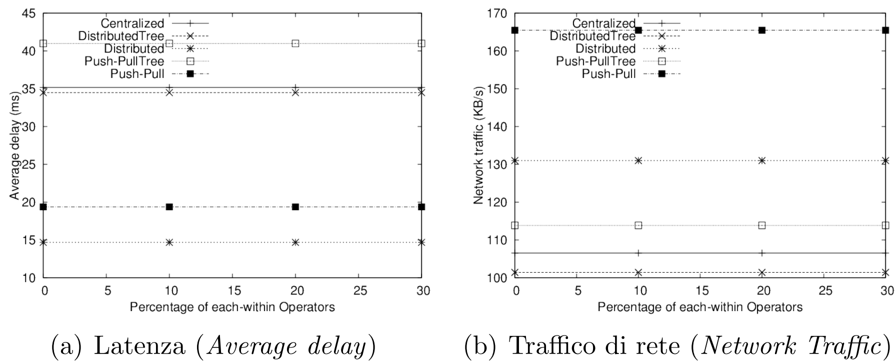
(a) Latenza (*Average delay*) (b) Traffico di rete (*Network Traffic*)

Figura 7.16: Prestazioni al variare del numero di aggregati



(a) Latenza (*Average delay*) (b) Traffico di rete (*Network Traffic*)

Figura 7.17: Prestazioni al variare della dimensione delle finestre



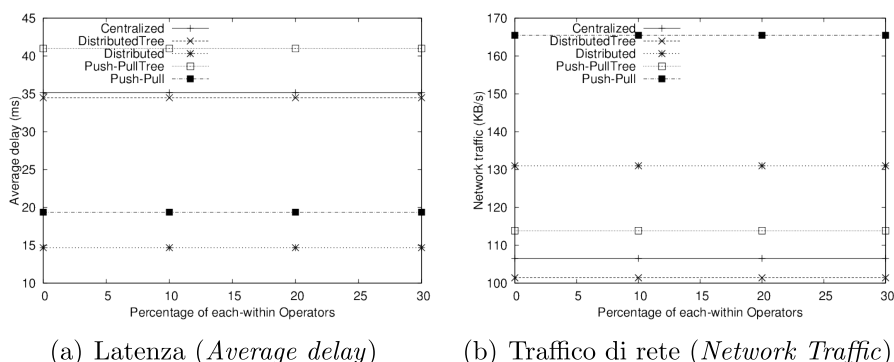
(a) Latenza (*Average delay*) (b) Traffico di rete (*Network Traffic*)

Figura 7.18: Prestazioni al variare della percentuale di each

7.5 Analisi generale dei risultati

Nell'effettuare questa analisi prendiamo inizialmente in considerazione i risultati ottenuti nel precedente Capitolo 7.2. Considerando il primo dei due,che confronta l'implementazione dell'algoritmo mediante automi con quella che impiega le pile, non solo la quantità di eventi che il singolo nodo della rete riesce ad analizzare senza decadimento di prestazioni è nettamente superiore, ma anche il tempo impiegato per effettuare il filtraggio delle informazioni è drasticamente diminuito. Per quanto riguarda quest'ultimo parametro il risultato migliore è senza dubbio quello di riuscire a mantenere costante il tempo di elaborazione nei casi di selezione singola (*firt-within* e *last-within*), indipendentemente dalla complessità della regola. Questo parametro, nel caso facente uso degli automi, degrada in maniera sensibile le prestazioni del singolo nodo, in termini di tempo di elaborazione necessario.
Questo porta, dunque, ad avere un sistema composto da nodi altamente reattivi e capaci di gestire un grande numero di informazioni in transito (quasi 700000 eventi/s, con un tempo di elaborazione medio di 400 microsecondi nel caso più complesso simulato). É possibile valutare anche la capacità di T-Rex di rispettare i vincoli di uno scenario altamente distribuito che presenta differenti requisiti, mediante i risultati ottentuti nel Capitolo 7.4.
Per effettuare quest'analisi prendiamo in considerazione le simulazioni effettuate, mediante Omnet++, di un'intera rete di nodi che implementa non solo la logica applicativa, ma anche i protocolli di comunicazione introdotti da T-Rex. Possiamo, senza perdita di generalità, impiegare il grafico riguardante l'andamento delle prestazioni del sistema al variare della percentuale di politiche di selezione *each-within*, riportata in Figura 7.19 per comodità. Dal momento che al variare di quest'ultima i valori restano costanti, il grafico mostra chiaramente i risultati dei vari protocolli in relazione al caso centralizzato, in termini di latenza del riconoscimento e del traffico di rete generato.

Un primo risultato, trasversale rispetto al protocollo, risiede proprio nella natura di questi grafici: la politica di selezione multipla *each-within*, che maggiormente negli altri sistemi CEP e DSP tende a far esplodere il numero



(a) Latenza (*Average delay*) (b) Traffico di rete (*Network Traffic*)

Figura 7.19: Prestazioni al variare della percentuale di each

di risultati da individuare e trasmettere, non presenta un impatto sensibile sulla latenza e sul traffico del sistema, anche per percentuali del 30%[2] degli eventi selezionati con questa modalità.
Confrontando invece i vari protocolli si può notare come, rispetto al caso centralizzato, il caso distribuito nella sua accezione ad albero multiplo (Distributed in Figura 7.19(a)) risulti maggiormente performante, per qualsiasi parametro considerato, dal punto di vista della latenza introdotta, mentre la sua accezione ad albero singolo (DistributedTree in Figura 7.19(b)) prevale rispetto agli altri protocolli, per quanto concerne il traffico di rete.
Questo risultato dimostra come T-Rex sia in grado di soddisfare i tipici requisiti delle applicazioni che impiegano i sistemi CEP, nonostante la loro varietà, fornendo differenti implementazioni adatte ai due vincoli considerati nel Capitolo 4: la latenza e il traffico di rete.

[2]Superiore alle normali necesità degli ambiti applicativi che, altrimenti, non filtrerebbero abbastanza informazioni da permettere un'effettiva fruibilità degli eventi compositi da parte degli utilizzatori.

Capitolo 8

Lavori correlati

Nell'ambito dell'*Information Flow Processing(IFP)*, ovvero nell'ambito di applicazioni distribuite geograficamente che richiedono la continua elaborazione di dati entro determinati vincoli temporali, si collocano, insieme ai DSP systems, i CEP system (come descritto nel Capitolo 1), di cui T-Rex fa parte. Per confrontare e dare una collocazione a quest'ultimo è necessario fare un'analisi degli altri sistemi esistenti, delle loro tipologie, dei loro modelli di computazione, delle funzionalità che presentano ed infine delle maggiori problematiche di cui risentono.
L'ambito dei sistemi CEP identifica il flusso di informazioni come notifiche di eventi avvenuti nel mondo esterno, che necessitano di essere filtrati e combinati, così da individuare eventi di più alto livello. Nonostante i contributi a questa tipologia di sistemi provengano da svariati ambiti (sistemi distribuiti, automazione dei processi di business, sistemi di controllo monitoraggio di rete, network di sensori e middleware) si può individuare nel dominio del *publish-subscribe* la radice comune di tutti i sistemi CEP realizzati. A differenza del normale impiego dei sistemi publish-subscribe, tuttavia, i sistemi CEP necessitano di estendere il linguaggio di sottoscrizione, consentendo la definizione di sequenze complesse.
Questo confronto tra T-Rex e gli altri progetti connessi all'ambito degli IFP systems viene effettuata nel successivo capitolo.

IFP systems e loro caratteristiche

Verrà descritto nel seguente capitolo un insieme significativo di sistemi IFP e mostrate le loro caratteristiche, così da presentare a grandi linee i principali sistemi in uso. Successivamente si analizzeranno gli aspetti generali dei sistemi IFP e si definirà, tra le varie scelte possibili, la collocazione di T-Rex.

ODE

Ode [19, 22] è un active-database per scopi generici object-oriented. Il database è gestito e manipolato, impiegando il linguaggio O++, un'estensione del C++ che supporta oggetti persistenti.
Le regole sono implementate mediante l'impiego di *constraints* e *triggers.* I primi vengono impiegati per adattare il sistema a differenti modelli di consistenza, sulla base dello scenario applicativo, mentre i secondi sono associati a sequenze di operazioni da attuare automaticamente, quando determinati metodi sono chiamati sugli oggetti che compongono la base di dati. Entrambi possono essere classificati come regole di riconoscimento, impiegando un linguaggio basato su patterns.
Ode non offre un supporto completo per la computazione periodica di regole, ma è possibile specificare azioni da eseguire dopo un determinato periodo di inattività di uno specifico trigger.
É possibile specificare delle sequenze, data la natura prettamente centralizzata del sistema, che permettono un ordinamento completo tra le regole basato sul tempo assoluto della singola macchina.
Le sequenze vengono tradotte in automi a stati finiti, visto che l'espressività del linguaggio è simile a quella delle espressioni regolari.
É infine possibile definire delle politiche di selezione multipla, mentre non viene considerato il consumo degli eventi già impiegati.

Cayuga

Cayuga [9] è un sistema di monitoraggio di eventi. É basato su *CEL* (*Cayuga Event Language*), un linguaggio che riprende fortemente i concetti definiti dai linguaggi dichiarativi tipici dei database. Possiede una struttura composta di una clausola *SELECT*, che filtra i flussi di dati in ingresso, una clausola *FROM*, che permette di specificare patterns da riconoscere, ed una clausola *PUBLISH*, che produce i dati in uscita.
Cayuga non introduce alcun sistema di finestre temporali, ma permette di specificare regole annidiate. La semantica di tutte le operazioni compiute da una regola è definita mediante l'algebra delle query [15] e viene inoltre descritto un metodo per tradurre le stesse in una serie di automi non deterministici. Questi, operando in parallelo per la stessa regola, consentono di ottenere policy di selezione multiple.
Anch'esso non permette il consumo degli eventi e non permette la distribuzione dell'elaborazione.

NextCEP

NextCEP [30] è un sistema CEP distribuito. Similmente a Cayuga, utilizza un linguaggio che include i tradizionali operatori SQL, insieme con operazioni connesse al riconoscimento di sequenze. Questo viene effettuato traducendo le regole in automi non deterministici, anch'essi simili a quelli impiegati da Cayuga.
Una sostanziale differenza con il precedente sistema descritto è rappresentata dalla possibilità di effettuare il riconoscimento della singola regola in maniera distribuita attraverso una serie di nodi connessi in maniera forte (ambiente di elaborazione suddiviso in *cluster*).
Introduce, inoltre, un modello di costo per le operazioni, effettuando il loro riordino, in maniera tale da ottenere il miglior risultato per quanto riguarda determinate dimensioni di interesse (ad esempio l'uso della CPU e il tempo di elaborazione), senza alterare i risultati individuati.

Amit

Amit [2] è un sistema di controllo a run-time per l'implementazione di applicazioni reattive e proattive. Amit presenta un componente, il *situation manager*, appositamente designato all'elaborazione delle notifiche ricevute, con l'obbiettivo di individuare delle sequenze, chiamate *situations*, ed inoltrarle verso i sottoscrittori.
Possiede un linguaggio di riconoscimento fortemente espressivo e flessibile, che permette la definizione di congiunzioni, negazioni, parametri, sequenze e ripetizioni. Viene introdotto il concetto di finestra temporale, vincolata a due eventi chiamati *initiator* e *terminator*. Consente, inoltre, la definizione di quantificatori di selezione (il primo, l'ultimo, tutti gli eventi validi) e una politica di consumo degli eventi definibile dall'utilizzatore. La strategia di riconoscimento è centralizzata e tutti gli eventi sono memorizzati in un solo nodo, sulla base della loro validità temporale definita mediante la finestra.

PB-CED

PB-CED [3] viene presentato come un sistema per l'individuazione di eventi complessi, impiegando dati provenienti da sorgenti distribuite. L'approccio impiegato viene definito *Plen-Based Complex Event Detection* e si pone come obbiettivo quello di implementare un sistema che consenta di definire un *plan* efficiente nella valutazione delle regole.
Il linguaggio di riconoscimento è semplificato alle sole operazioni di congiunzione, disgiunzione, negazione e sequenze. Non vengono, invece, implementati il riuso degli eventi e le iterazioni nella definizione dei patterns. Non è presente, inoltre, un sistema di finestre temporali che, al contrario, considera esclusivamente la durata dei singoli eventi.
PB-CED impiega una traduzione delle regole in automi non deterministici ed adotta una soluzione centralizzata per il riconoscimento delle sequenze. Alcuni componenti architetturali sono distribuiti nelle vicinanze delle sorgenti, ma operano esclusivamente una funzione di memorizzazione, senza effettuare alcuna elaborazione dei dati.
Viene introdotto un modello di costo, che tiene in considerazione diversi

parametri, sulla base del quale viene determinato il *plan*, combinando differenti automi, con l'obbiettivo di ritardare l'elaborazione degli eventi più ad alta frequenza.

Aurora/Borealis

Aurora [1] è un DSMS che definisce regole mediante un linguaggio imperativo chiamato SQuAL. Questo descrive le regole in maniera grafica, adottando un paradigna di frecce e boxes, che rende le connessioni tra gli operatori esplicite. É possibile suddividere gli operatori in due macro-categorie: a finestra ed a singolo operatore. La prima applica una funzione, con un solo parametro, ad una finestra temporale che scorrendo include nuovi elementi, prima di ripetere il processo. La seconda, invece, opera su una singola informazione, oggetto per oggetto della regola.
Aurora presenta una politica di selezione multipla degli eventi, ma non considera il consumo degli eventi. É possibile, inoltre, associare una QoS specifica ad ogni flusso di uscita consentendo, così, la personalizzazione della politica di esecuzione delle operazioni. Gli eventi non possiedono timestamp, ma vengono elaborati secondo l'ordine di arrivo nel sistema.
Il progetto è stato esteso con lo scopo di distribuire la computazione, sia su un singolo dominio amministrativo, che su differenti domini. Queste implementazioni, chiamate Aurora* e Medusa, operano entrambe mediante la definizione di *overlay networks* e sono state recentemente riunite in *Borealis stream processor*

Esper

Esper [16] è considerato il maggior progetto open-source CEP. Possiede un ricco linguaggio dichiarativo per la definizione delle regole, denominato *EPL* (*Event Processing Language*). Include tutti gli operatori SQL, ampliandoli con costrutti ad-hoc per la definizione di finestre e la generazione dei risultati. EPL possiede due metodologie per l'espressione delle sequenze: la prima, mediante vincoli annidiati, consente di definire congiunzioni, disgiunzioni, negazioni e iterazioni, la seconda, impiega espressioni regolari. En-

trambi i metodi posseggono la stessa espressività. Esper consente sia scenari centralizzati che basati su clusters.

Modelli computazionali e funzionalità

Tutti i sistemi descritti precedentemente ed, in generale, la maggior parte dei CEP systems, introducono come modello computazionale quello degli automi, derivato dalla traduzione di espressioni regolari, che rappresenta la massima espressività dei loro linguaggi di definizione. Dal punto di vista delle funzionalità presentate all'utilizzatore, invece, la soluzione risulta essere più varia.

Ode, Cayuga, PB-CED ed Amit consentono esclusivamente uno scenario centralizzato, mentre NextCEP, Aurora ed Esper permettono la distribuzione della computazione tra più nodi, ma esclusivamente tra clusters di lavoro, impedendo una capillare distribuzione su tutta la rete. Tutti i sistemi proposti consentono la scelta di differenti policy di selezione degli eventi, ma solo Esper e Amit di personalizzarne il loro impiego. Questi ultimi due sono, inoltre, gli unici sistemi che danno la possibilità di definire policy di consumo degli eventi, che individuano sequenze valide.

Dal punto di vista delle funzionalità introdotte, le differenze sono più marcate. Tutti i sistemi, a parte Aurora, il quale consente solo la selezione, permettono la selezione, congiunzione e disgiunzione di eventi, ma solo Esper consente di avere un sistema personalizzabile di finestre temporali (mentre NextCEP Aurora ed Amit ne presentano uno generico di finestre che scorrono all'arrivo degli eventi). Caygua, Aurora, PB-CED e NextCEP non offrono la possibilità di riconoscere ripetizioni ed il quarto neanche la parametrizzazione. Le negazioni non vengono considerate da Aurora, Cayuga e NextCEP. La produzione di aggregati è consentita da tutti i sistemi esaminati, ma il loro riconoscimento non è presente in Ode. Infine nessuno dei sistemi contempla un sistema di rimozione dei duplicati.

Per quanto riguarda T-Rex la scelta è ricaduta sul modello a pile che, nel nostro caso, come mostrato nel Capitolo 7, mostra delle performances nettamente superiori rispetto all'impiego degli automi. Viene supportato sia uno

scenario centralizzato, che uno completamente distribuito lungo tutta la rete (Capitolo 4). Vengono inoltre resi disponibili tutti gli operatori discussi in precedenza (Capitoli 2 e 3) e gestita la rimozione dei duplicati in maniera architetturale (Capitolo 6.2.1).

Operator placement

Negli esempi precedentemente proposti la distribuzione è permessa esclusivamante tra nodi appartenenti allo stesso cluster. Le proposte più avanzate, nell'ambito dei CEP systems, presentano l'adozione di un'architettura completamente distribuita per il CEP engine, ma hanno diverse limitazioni e problematiche. A titolo di esempio citiamo Microsoft StreamInsight [23], come piattaforma che si propone di prendere in considerazione il problema della distribuzione del sistema CEP che implementa. Questo sistema propone una soluzione di ottimizzazione della distribuzione di intere query di riconoscimento, mediante un'analisi effettuata preventivamente, sulla base di determinate metriche di costo e statistiche dei flussi informativi in ingresso.
Nell'ambito della ricerca di una soluzione a questo problema (denominato *operator placement*) tutti i meccanismi sviluppati nei CEP system moderni offrono soluzioni simili. Questo approccio al problema, però, presenta diversi svantaggi. In primo luogo l'analisi dello scenario ottimale, effettuato mediante tecniche legate all'ambito di ricerca operativa, viene effettuata offline e, di conseguenza, non risulta essere tollerante a cambiamenti sensibili della frequenza di generazione degli eventi primitivi, non prevedendo un sistema di monitoraggio online del traffico in transito nel sistema. In secondo luogo non vi è una chiara definizione delle metriche più significative. Questo comporta che, sulla base di differenti metriche di costo impiegate, l'algoritmo di operator placement produca risultati differenti e, a volte, addirittura in contraddizione. Come esempio di questa variabilità dell'operator placement è sufficiente considerare che differenti metriche di costo portano a ritenere più efficiente, all'interno dello stesso nodo di rete, condividere gli operatori in comune tra differenti eventi compositi, mentre altre ottengono il risul-

tato opposto, ovvero che convenga duplicarli. Infine, non sempre i sistemi CEP distribuiti, quali ad esempio, StreaInsight consentono di distribuire i singoli operatori che compongono l'algoritmo di riconoscimento, ma solo la distribuzione di intere unità computazionali.
Differentemente T-Rex non solo automatizza il processo di operator placement delle singole operazioni in maniera strutturale, ma effettua quest'operazione online, durante l'installazione della regola, sulla base dell'attuale topologia di rete e dei generatori connessi (come descritto nel Capitolo 4). Inoltre, nella sua versione Push-Pull, descritta nel Capitolo 4.4, introduce il concetto di operator placement dimanico. Grazie all'introduzione di un monitor del traffico di rete costantemente attivo, infatti, il sistema si adatta al traffico attuale ed, ad intervalli regolari, viene riordinata la gerarchia delle operazioni da effettuare. Nell'operare questo ulteriore adattamento alle condizioni della rete, T-Rex ottiene prestazioni confrontabili con la tipologia distribuita del protocollo per quanto riguarda la latenza ed il traffico generato.

Tipologie di CEP systems

Questa categoria di sistemi si è sviluppata per superare le limitazioni dei DSP systems, descritte nel capitolo 1.1, i quali lasciano ai propri client la responsabilità di associare una semantica ai dati che vengono elaborati. Per ovviare a questo problema, i CEP systems lo affrontano da un ottica diametricalmente opposta: associare una semantica precisa ai dati in transito, considerandoli notifiche di eventi accaduti nel mondo esterno ed osservati dalle sorgenti.
Fondamentalmente si possono differenziare i sistemi CEP in due macrocategorie: *Topic based* e *Content based*.
I primi permettono ai sottoscrittori di esprimere l'interesse esclusivamente nei confronti di categorie predefinite; in questo caso i publisher devono definire, in fase di notifica di un evento, l'insieme di appartenenza.
I secondi, invece, permettono la specifica di complessi filtri, per selezionare l'interesse a determinati eventi. T-Rex appartiene a questa categoria.

In quest'ultimo ambito sono stati sviluppati differenti linguaggi per rappresentare il contenuto degli eventi e definire i filtri di sottoscrizione: da semplici coppie attributo/valore [11] a complessi schemi XML [4, 7]. T-Rex si pone nella categoria dei CEP systems che impiegano un linguaggio strutturato, in particolare, impiega un linguaggio formale, TESLA, definito a partire da una logica temporale e descritto nel Capitolo 2. Questo consente di specificare una semantica disambigua per entrambe le definizioni (eventi e filtri di sottoscrizione).

Il parallelismo ed i sistemi CEP

Nel lavoro presentato l'elaborazione delle regole avviene esclusivamente in maniera sequenziale e si è sfruttato il sistema multiprocessore con l'obiettivo di effetture un numero di simulazioni maggiori, eseguendole in maniera parallela. Altri lavori si pongono come scopo quello di esplorare i vantaggi introdotti dalla computazione parallela. É infatti possibile sfruttare un'architettura parallela per ottimizzare il processo di individuazione delle sequenze valide. Due sono le possibili scelte che si possono effettuare: parallelizzare tra di loro diverse regole oppure parallelizzare le operazioni necessarie ad effettuare l'elaborazione di ogni singola regola. In questo secondo ambito le scelte possono ulteriormente suddividersi tra impiego di hardware *general-purpose* e quello di harwdare dedicato.
Un esempio del primo scenario è l'utilizzo di CUDA [26], un'architettura per la programmazione su GPU, che presenta una spiccata natura parallela. In particolare lo scopo è considerare gli operatori di maggior interesse offerti dai sistemi CEP esistenti, valutando differenti algoritmi di elaborazione e confrontando la loro implementazione per CPU multi-core e per CUDA [13]. Per quanto riguarda il secondo scenario, invece, un esempio è l'impiego di FPGA (Field-programmable Gate Array) [18], un circuito integrato digitale la cui funzionalità è programmabile via software. In questo ambito diversi studi sono stati compiuti per sviluppare piattaforme per effettuare il *matching* di pattern di eventi, supportando alte frequenze di traffico a bassa latenza, esplorando differenti soluzioni con differenti gradi di parallelismo [33, 29].

Capitolo 9

Conclusioni

L'ambito dei sistemi CEP, al quale T-Rex appartiene, presenta una vasta gamma di scenari di impiego. Questi sono spesso caratterizzati da una grande quantità di dati confluenti dalla periferia della rete al centro del sistema. Due sono principalmente i vincoli imposti dall'ambito applicativo, nel quale questi flussi di dati vengono generati e processati.
Il primo riguarda la capacità di gestire la grande mole di informazioni generate in maniera distribuita da molteplici nodi della rete ad una frequenza non nota a priori, senza saturare le risorse di rete e, quindi, congestionarla. Un ambito di impiego potrebbe essere un sistema di riconoscimento di frodi che impieghi le transizioni dei circuti bancari come eventi primitivi.
Il secondo riguarda la tempistica di elaborazione delle informazioni che devono essere rese disponibili a livello applicativo, non appena vengono generate. Un esempio di realizzazione potrebbe essere un sistema di rilevamento di incendi che impieghi sensori di fumo e di temperatura.
Scopo principale del lavoro è stato quello di individuare, nella distibuzione dell'algoritmo e del protocollo, un vantaggio in termini di reattività del sistema e di occupazione delle risorse di rete. La qualità dei risultati ottenuti in questi due ambiti, mediante le differenti soluzioni ad albero singolo e multiplo che sono state sviluppate, hanno esplicitato come la distribuzione del protocollo e dell'elaborazione influenzi direttamente, in maniera positiva, le performances dell'intero sistema. Confrontando le performances dell'algo-

ritmo di riconoscimento ideato e descritto in questo lavoro, che fa uso degli stacks, con quelle della versione ad automi, riusciamo inoltre a dimostrare che T-Rex, mediante un approccio stilistico innovativo basato sulle pile, ottiene delle prestazioni nettamente superiori congiuntamente all'impiego di TESLA come linguaggio di definizione delle regole. Risultati analoghi, di maggiore efficienza degli stacks rispetto agli automi, sono stati individuati nell'ambito delle risorse di sistema impiegate. La realizzazione descritta in questo lavoro impiega, infatti, un quantitativo di memoria nettamente inferiore, tale da permetterci di sfruttare completamente il parallelismo offerto dal multiprocessore.
Prendendo in considerazione gli altri sistemi ideati per il rilevamento distribuito di eventi complessi, di cui abbiamo discusso nel Capitolo 8, possiamo osservare che nessuno di essi impiega un algoritmo di riconoscimento con un paradigma simile. Ci si aspetta, dunque, che i vantaggi degli stacks riscontrati in questo lavoro siano estendibili anche ad altri casi, come indicano alcune prove condotte su Esper.

Sviluppi futuri

Un primo ambito di sviluppo introdotto da questo lavoro è rappresentato, per quanto specificato precedentemente, dall'analisi dell'effettiva potenzialità del paradigma a stacks, rispetto a quello ad automi nell'ambito dei sistemi CEP. Per ottenere questo risultatato sarà necessario introdurre questa modalità di elaborazione in altri sistemi e confrontarne le performances. Per quanto riguarda T-Rex i lavori futuri saranno incentrati a sviluppare e migliorare il protocollo Push-Pull, sia nella sua versione ad albero singolo, che in quella ad albero multiplo. Considerando alcuni parametri, i due protocolli si comportano in maniera più efficiente di quello centralizzato, mentre presentano performances peggiori in altre situazioni. Sulla base di questo iniziale risultato una possibile motodologia di sviluppo consiste nell'indagare i motivi che determinano questo comportamento, cercando nuovi scenari di confronto, per verificare se sia duvuto ad una mancata comprensione degli ambiti più favorevoli a questa modalità. Un primo approccio potrebbe essere quello

di simulare ambienti di grosse dimensioni con un carico fortemente variabile, sia su base topologica, che su base temporale, così da valutare appieno la capacità dei due algoritmi (ad albero singolo ed ad albero multiplo) di effettuare *load balancing* del traffico in maniera efficiente.
Contemporaneamente un'altra possibile strada da percorrere potrebbe essere quella di snellire il protocollo, limitando l'*overhead* introdotto nella gestione dell'individuazione e dell'aggiornamento della regole master, così da diminuire il divario con il caso distribuito.

Bibliografia

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[3] U. Akdere, M. Çetintemel and N. Tatbul. Plan-based complex event detection across distributed sources. In *Proc. VLDB Endow. 1*, pages 66–77, 2008.

[4] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of 26th IEEE International Conference on Very Large Data Base*, pages 53–64, SanFrancisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[5] András Varga. *OMNeT++ User Manual Version 4.1*, 2010.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[7] H. K. Y. Ashayer, G. Leung and H Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *Proceedings of the Workshop on Distributed Event-based Systems, co-located with the 22th International Conference on Distributed Computing Systems*, Vienna, Austria, 2002. IEEE Computer Society Press.

[8] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, New York, NY, USA, 2002. ACM.

[9] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *SIGMOD*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[10] Krysia Broda, Keith Clark, Rob Miller 0002, and Alessandra Russo. Sage: A logical agent-based environment monitoring and control system. In *AmI*, pages 112–117, 2009.

[11] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, August 2003.

[12] G. Cugola and A. Margara. Complex event processing with t-rex. *Technical report*, 2010.

[13] G. Cugola and A. Margara. Low latency complex event processing on parallel hardware. Technical report, Politecnico di Milano, 2011. Submitted for Publication.

[14] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *DEBS*, pages 50–61, 2010.

[15] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[16] Esper, http://espertech.com/, 2010.

[17] Event zero, http://www.eventzero.com/solutions/environment.aspx.

[18] Fpga, http://en.wikipedia.org/wiki/field-programmable_gate_array.

[19] Narain H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB*, pages 327–336, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[20] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[21] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, pages 249–269. Springer-Verlag New York, Inc., 2005.

[22] Narain H. Lieuwen, D. F. Gehani and R. M. Arlein. The ode active database: Trigger semantics and implementation. In *Proceedings of the Twelfth International Conference on Data Engeneering*, pages 412–420, Washington, DC, USA, 1996. IEEE Computer Society.

[23] Streaminsight, msdn.microsoft.com/en-us/library/ee362541.aspx.

[24] Angelo Morzenti, Dino Mandrioli, and Carlo Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992.

[25] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[26] NVIDIA. *CUDA C Programming Guide 3.2*, 2010.

[27] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Wels, and Margo Seltzer. Network-aware operator placement for stream-processing systems. Technical report, Harvard University, 2005.

[28] Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18:44–55, 2004.

[29] Mohammad Sadoghi, Harsh Singh, and Hans-Arno Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 27–32, New York, NY, USA, 2011. ACM.

[30] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. Distributed complex event processing with query optimization. In *International Conference on Distributed Event-Based Systems (DEBS '09)*, New York, NY, USA, 2009. ACM.

[31] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

[32] Fusheng Wang and Peiya Liu. Temporal management of rfid data. In *VLDB*, pages 1128–1139. VLDB Endowment, 2005.

[33] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex event detection at wire speed with fpgas. *Proc. VLDB Endow.*, 3:660–669, September 2010.