Alma Mater Studiorum · Università di Bologna

**facoltà di scienze matematiche, fisiche e naturali**
**Corso di Laurea Triennale in Informatica**

# Algoritmo "Session correlation management with Radix Tree": Sviluppo di un'implementazione apposita per il linguaggio JOLIE

**Tesi di Laurea in Paradigmi di Programmazione**

**Relatore:**
**Chiar.mo Prof.**
**Maurizio Gabbrielli**

**Presentata da:**
**Davide Lerose**

**Sessione II**
**Anno Accademico 2009/2010**

# Introduzione

Il Service-Oriented Computing (SOC) è un paradigma di programmazione utile alla creazione di applicazioni, basate su un'architettura Service-Oriented (SOA), nella quale le applicazioni sono costituite da servizi. Uno degli aspetti principali del SOC è la composizione. Ciascun servizio, infatti, ne consente la composizione di altri, complessi, combinando tra loro varie funzionalità. La modellazione di queste composizioni di servizi può avvenire seguendo un approccio basato sull'orchestrazione o un approccio di tipo coreografico: il primo offre un punto di vista locale, mentre il secondo offre una visione globale della situazione del SOA. La tecnologia basata sui SOC più utilizzata ai giorni nostri è il Web Service, un sistema software basato sull'interoperabilità tra diversi elaboratori in rete. Ciò è ottenuto tramite lo scambio di appositi "messaggi" (tipicamente SOAP) formattati utilizzando XML e trasportati tramite HTTP.
A differenza di un approccio di tipo Object Oriented, in un SOA non esiste un meccanismo che garantisca l'identificazione di una nuova sessione creata. Una soluzione a questo problema, pubblicata da WS-BPEL, consiste nell'utilizzo dei Correlation Set. I Correlation Set sono un sottoinsieme delle variabili di una sessione, definito a priori, i cui valori vengono utilizzati per identificare la sessione stessa. Un'implementazione di questo meccanismo di correlazione è fornita dal linguaggio Service-Oriented JOLIE. In questo linguaggio, sviluppato in Java, sia il messaggio che la sessione contengono un insieme di variabili utilizzate come Correlation Set. All'arrivo di un messaggio ne vengono letti i valori delle variabili; in seguito lo si indirizza alla

sessione avente le medesime variabili ed il medesimo valore.
La complessità attuale dell'algoritmo utilizzato da JOLIE per gestire la correlazione è lineare rispetto al numero di sessioni presenti. Con l'algoritmo descritto in questa tesi si vuole fornire un'alternativa avente complessità molto inferiore e dunque avente prestazioni migliori nella ricerca delle sessioni. "Tale algoritmo è stato proposto in [15]".
L'implementazione dell'algoritmo presentata, ribattezzata "Session Correlation management with Radix Tree", ha complessità esponenziale sul numero delle variabili del Correlation set, ma è indipendente dal numero di sessioni presenti. Questo comporta un considerevole miglioramento, poiché in un sistema Service-Oriented l'aspetto prestazionale è influenzato soprattutto dal numero di sessioni presenti ed il numero di variabili nel Correlation set si può considerare costante. Questo è un risultato ottenibile, poiché la ricerca in una struttura a Radix Tree è indipendente dal numero di foglie, ma rimane dipendente dalla lunghezza della stringa utilizzata come chiave di ricerca. Come vedremo in dettaglio in seguito, la chiave utilizzata per memorizzare le sessioni e per ricercarle è il risultato della composizione dei valori delle variabili che formano il Correlation set.
Lo sviluppo di questo algoritmo è stato improntato per poter consentire una futura integrazione con il linguaggio JOLIE, allo scopo di migliorarne le prestazioni nell'ambito della correlazione di sessioni. Come vedremo, a questo si è voluta aggiungere la possibilità di integrare il suddetto algoritmo con un qualunque linguaggio Service-Oriented implementato in Java. A tal scopo sono state fornite delle funzioni apposite che consentono di integrare l'algoritmo senza modificare nient'altro all'interno del codice sorgente.

# Indice

# Capitolo 1

# Il Service-Oriented Computing

## 1.1 Service-Oriented Computing.

Il Service-Oriented Computing (SOC) fa riferimento all'insieme di concetti, principi e metodi che rappresentano un paradigma di programmazione utile alla creazione di applicazioni basate su un'architettura Service-Oriented (Service-Oriented Architecture o SOA) [1]. In una SOA le applicazioni sono costituite da servizi semplici e indipendenti ma componibili a formare servizi più complessi. L'idea su cui si basano SOC e SOA è separare nettamente l'ingegnerizzazione del software dalla programmazione vera e propria, così da enfatizzare la prima rispetto alla seconda. A questo scopo, il SOC separa lo sviluppo delle applicazioni in tre componenti distinte e ben definite:

- **Creatori di servizi:** usano linguaggi di programmazione tradizionali quali Java, C e C# per implementare componenti di programmi. Questi componenti vengono quindi incapsulati, fornendo interfacce grazie che ne consentano l'utilizzo. Questi servizi possono essere paragonanti ad una fabbrica di mattoncini Lego.

- **Fornitori di servizi:** Consentono il pubblico accesso ai servizi creati, oltre ad agevolare i creatori di applicazioni nella ricerca dei servizi di cui necessitano. Possiamo paragonarli ad un negozio specializzato di mattoncini Lego.

- **Creatori di applicazioni:** Sono programmatori che, anziché creare applicazioni basandosi sui componenti di base di un linguaggio di programmazione, utilizzano i servizi come componenti di un linguaggio ad alto livello. Riprendendo l'esempio dei mattoncini, possiamo paragonare i creatori di applicazioni ai bambini, che usano i mattoncini (servizi) per creare oggetti complessi (applicazioni).

Da quanto detto finora si deduce che uno degli aspetti principali del SOC è la composizione. L'interfaccia pubblica fornita da ciascun servizio, infatti, consente la composizione di complessi servizi, riutilizzando funzionalità già fornite dagli stessi. La modellazione di queste composizioni di servizi può avvenire seguendo due approcci:

- **Orchestrazione (Orchestration):** descrive come i servizi interagiscano tra loro a livello di messaggi scambiati. Il risultato consiste nella specifica di un processo "long-running" di alto livello, la quale descrive un'applicazione eseguibile da un proprietario. Secondo questo approccio, il processo implementa il punto di vista di uno dei partner, e il flusso è sempre controllato da esso. Il processo risultante può essere a sua volta esposto come servizio.

- **Coreografia (Coreography):** tramite questo approccio viene tenuta traccia della sequenza di messaggi che può coinvolgere più applicazioni attraverso una visione globale del processo. Si punta a fornire una rappresentazione più collaborativa delle interazioni (ogni attore descrive la sua parte e tutti gli attori si trovano allo stesso livello). Nella coreografia non è presente un punto di controllo centralizzato.

Riassumendo, possiamo dire che l'orchestrazione offre un punto di vista locale, mentre la coreografia uno globale.

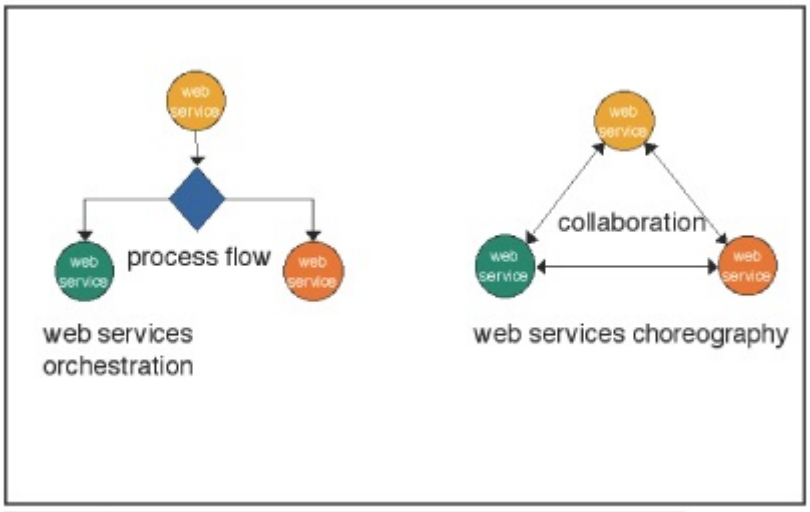


Figura 1.1: Orchestrazione e Coreografia

## 1.2 Web Service

La tecnologia basata sui SOC più utilizzata ai giorni nostri è il Web Service [2]. Un Web Service (servizio web) è un sistema software progettato per supportare l'interoperabilità tra diversi elaboratori su di una medesima rete. La caratteristica fondamentale di un Web Service è di offrire un'interfaccia software (ad esempio WSDL [3]) grazie alla quale altri sistemi possano interagire con lo stesso. A tal fine, gli altri sistemi attivano le operazioni descritte nell'interfaccia tramite appositi "messaggi" (tipicamente SOAP [4]) formattati secondo lo standard XML [5] e trasportati tramite il protocollo HTTP [6].
Come accennato in precedenza, l'interfaccia dei Web Services viene generalmente descritta utilizzando il linguaggio WSDL:

### 1.2.1 Web Services Description Language (WSDL)

è un linguaggio formale, in formato XML, utilizzato per la creazione di descrizioni di Web Service. Mediante WSDL può essere infatti descritta l'interfaccia pubblica di un Web Service, cioè un elenco di informazioni (XML)

riguardanti le modalità di interazione con un determinato servizio. Un "documento" WSDL contiene, infatti, relativamente al Web Service descritto, informazioni riguardanti:

- cosa può essere utilizzato (le "operazioni" messe a disposizione dal servizio);
- come utilizzarlo (il protocollo di comunicazione da utilizzare per accedere al servizio, il formato dei messaggi accettati in input e restituiti in output dal servizio ed i dati correlati), ovvero i "vincoli" (bindings, in inglese) del servizio;
- dove utilizzare il servizio (cosiddetto end-point del servizio che solitamente corrisponde all'indirizzo - in formato URI - che rende disponibile il Web Service).

Le operazioni supportate dal Web Service ed i messaggi scambiabili con lo stesso sono descritti in maniera astratta, quindi collegati ad uno specifico protocollo di rete e ad uno specifico formato. Grazie all'utilizzo di standard basati su XML ed alle interfacce pubbliche, applicazioni software implementate in diversi linguaggi di programmazione e su diverse piattaforme hardware possono quindi essere utilizzate, sia per lo scambio di informazioni sia per effettuare operazioni complesse, su reti aziendali come su Internet. L'interoperabilità fra diversi software e diversi sistemi operativi è resa possibile dall'uso di standard aperti e accessibili a tutti.
Facendo riferimento alla Coreografia ed all'Orchestrazione nell'ambito dei Web Services, ad oggi i linguaggi di rifermento per la gestione di questi ultimi sono, rispettivamente, WS-CDL [7] e WS-BPEL [8].

### 1.2.2 Web Services - Choreography Description Language (WS-CDL)

è un linguaggio basato su XML che descrive una collaborazione tra pari definendo, da un punto di vista globale, i loro comportamenti osservabili

comuni e complementari. Con lo scambio di messaggi tra pari si cerca di ottenere il raggiungimento di un comune scopo (come da definizione di coreografia).

### 1.2.3 Web Services - Business Process Execution Language (WS-BPEL)

Nasce nel 2002, definendo un standard di settore per l'Orchestrazione e l'esecuzione dei processi di business. è un linguaggio che permette la modellazione del comportamento dei partecipanti nelle interazioni basate sui Web Services. Un processo BPEL può rappresentare un Web Service, che può essere utilizzato a sua volta da altri Web Service esterni chiamati partner, oppure essere esso stesso il richiedente di Web Services offerti da altri partner. Riguardo la sua implementazione, BPEL esprime la logica di funzionamento di un processo attraverso una grammatica basata su XML, la quale è interpretabile mediante un motore di orchestrazione opportunamente progettato. Il compito di questo motore di orchestrazione è di coordinare i vari servizi che compongono il processo. Una delle particolarità di BPEL consiste nel fatto che i Web Service possono avere una composizione ricorsiva. Infatti, un processo definito da un insieme di Web Service collaboranti può essere esso stesso un Web Service. Questo nuovo servizio sarà quindi visibile all'esterno come un servizio semplice, di cui è definita un'interfaccia Web Services Description Language (WSDL) e nascondendo di conseguenza la sua composizione interna.
Comunque sia, entrambi i linguaggi non offrono definizioni formali del loro comportamento. Una conseguenza di ciò è che, per un medesimo programma, ambienti di esecuzione diversi possono portare a risultati diversi.

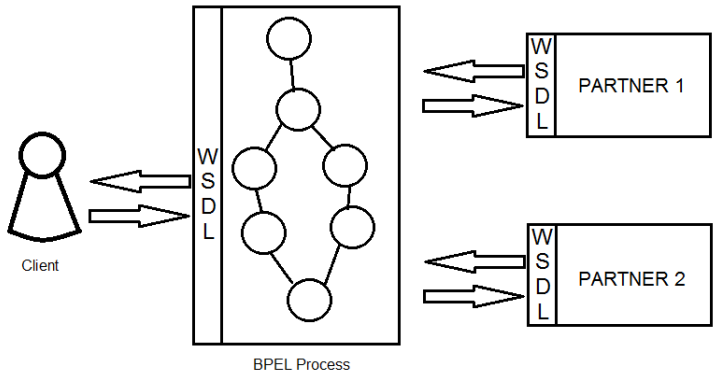


Figura 1.2: BPEL Interface

## 1.3 Correlation Set

Come si è potuto notare, i Web Services sono basati su un intenso utilizzo di comunicazione tramite messaggi, garantito dallo sfruttamento di primitive implementate da ogni linguaggio operante in quest'ambito. Queste primitive sono chiamate *operation* (operazioni), e vengono suddivise in due categorie: "operazioni in entrata" (*input operation*) e "operazioni in uscita" (*output operations*). Per fornire una classificazione più dettagliata, si hanno quattro tipi di operations (due per ciascuna categoria):

- **One-Way operations (Input):** sono operazioni che hanno il compito di rimanere in attesa di un messaggio in arrivo.

- **Request-Response operations (Input):** simili alle One-Way operations, all'arrivo di un messaggio spediscono al mittente una risposta.

- **Notification operations (Output):** complementari rispetto alle One-Way operations, sono utilizzate per inviare messaggi.

- **Solicit-Response operations (Output):** complementari rispetto alle Request-Response operations, hanno lo scopo di inviare un messaggio e ricevere una risposta.

Nella loro applicazione pratica, i Web Sevices possono avere più istanze di esecuzione di un medesimo modello comportamentale. Queste istanze sono chiamate **Sessioni**, e vengono definite formalmente come istanze di esecuzione del comportamento di un servizio, associato al suo stato locale (i.e. Sono un associazione comportamento-stato). Di conseguenza, uno dei problemi principali legati al paradigma orientato ai servizi (SOC) consiste nell'identificazione delle sessioni a cui un determinato servizio vuol fare riferimento. Dato che due sessioni aventi lo stesso comportamento potrebbero essere distinte solo tramite il loro stato, una soluzione a questo problema è stata data da WS-BPEL tramite i **Correlation Set** [9].
Possiamo così descrivere il meccanismo dei Correlation Set: una sessione X è composta dalla coppia $(C, S)$, dove C rappresenta il comportamento della sessione e S il suo stato locale; lo stato S della sessione è dato dall'associazione tra tutte le variabili locali V e tutti i loro rispettivi valori v $(V- > v)$. Di conseguenza, due sessioni $X = (C, S_1) e Y = (C, S_2)$ risulterebbero non distinguibili se e solo se $S_1 = S_2$, cioè se e solo se lo stato locale risultasse identico. Per distinguere due sessioni, quindi, è possibile definire un sotto insieme delle variabili che compongono lo stato della sessione stessa. Stabilendo, poi, che questo sotto insieme rappresenta una sorta di identificativo della sessione, si impone che questo sia sempre diverso da quello delle altre sessioni. I Correlation Set operano esattamente in base a questo principio: infatti, essi sono un sotto insieme predefinito di V, che possiamo definire come VC. Formalmente, si ha che $V = VC + VR$, dove VR rappresenta l'insieme delle variabili di V non comprese nel Correlation Set. Quindi, indicando con vc l'insieme dei valori delle variabili di VC, si ha che due sessioni X e Y aventi, $VC_x = VC_y$, risultano non distinguibili per correlazione se e solo se $vc_x = vc_y$.

# Capitolo 2

# JOLIE

## 2.1 Il linguaggio

JOLIE (acronimo di Java Orchestration Language Interpreter Engine) [10] è un linguaggio di programmazione basato sul Service-Oriented Computing. Il linguaggio è, di conseguenza, fortemente basato sui servizi. In JOLIE, non si fa distinzione tra servizi remoti e locali. Questo comporta la possibilità di spostare fisicamente questi ultimi senza che vi sia la necessità di modificare la struttura interna del sistema: è sufficiente modificare la nuova locazione a cui fa riferimento il servizio spostato. Come spiegato nel capitolo 1, nell'ambito dei SOC la possibilità di combinare servizi è molto ampia, al punto che questi possono essere a loro volta composti da ulteriori servizi, reiterando il pattern all'infinito. Questa possibilità è disponibile anche in JOLIE, dove ne è fornito un meccanismo che la implementa nativamente, semplificando la composizione di servizi.
JOLIE è sviluppato tramite Java, ed ha quindi un forte legame con esso. Difatti, è possibile creare servizi Java per poi utilizzarli all'interno di un applicazione JOLIE sfruttandone le API. Data la capacità di JOLIE di processare le richieste HTTP, si ha inoltre la possibilità di interagire direttamente con i Browser o con i Web Server.
Le Service-Oriented Architecture sono composte tipicamente da un eleva-

to numero di servizi, molti dei quali forniscono funzionalità ausiliare ad un singolo servizio. Questa struttura è implementata in JOLIE, garantendo la possibilità di incapsulare vari servizi in un servizio unico, ed è eseguita dalla medesima JVM (Java Virtual Machine).
Data questa struttura composta dei SOA, è utile avere un unico punto di convergenza a cui fare riferimento come destinazione. Questo endpoint avrà come scopo quello di inoltrare i messaggi ai servizi richiesti. è possibile implementare in JOLIE questo meccanismo in modo semplice. Nel caso in cui un servizio interno venga rimpiazzato o spostato, gli utenti accederanno sempre e comunque al servizio principale, lasciando a quest'ultimo la responsabilità di inoltrare le richieste.

## 2.2 Gestione attuale delle sessioni in JOLIE

Essendo JOLIE un linguaggio basato sul Service-Oriented Computing, è molto importante, ai fini prestazionali, il modo in cui è implementata la gestione delle sessioni. Infatti, in una SOA è fondamentale il meccanismo che consente una corretta identificazione delle sessioni a cui fare riferimento e, dato che tipicamente il numero di sessioni attive è molto elevato, è di fondamentale importanza anche la velocità con cui queste possano essere reperite. In JOLIE, l'identificazione delle sessioni è effettuata fornendo nativamente il meccanismo dei Correlation Set. Siccome vengono utilizzati dei protocolli *stateless* (i.e. privi di stato, quali SOAP e HTTP), ci si affida al contenuto dei messaggi per poter identificare la sessione destinataria.
Questo meccanismo può essere spiegato approssimativamente nel seguente modo: sia il messaggio che la sessione contengono entrambi un insieme di variabili utilizzate come Correlation Set. Possiamo indicare le variabili di questo insieme come *variabili di correlazion*e, le quali avranno associato un determinato valore a tempo di esecuzione. All'arrivo di un messaggio si leggono le sue variabili di correlazione e lo si indirizza alla sessione avente le medesime variabili col medesimo valore.

In realtà, in JOLIE, la questione è più complesse, dato che esiste la possibilità (presente in pochissimi altri linguaggi) di avere più di una variabile di correlazione, oltre al fatto che alcune variabili sono non definite. Ipotizziamo, ad esempio, che si voglia creare un servizio S che consenta la comunicazione tra due client, specificando che la comunicazione venga gestita da una sessione. Ipotizziamo inoltre che ogni client possa essere identificato univocamente tramite il suo nome (X e Y), e che ciascun client possa comunicare con un solo altro nello stesso momento. Prendendo in esame una sola variabile di correlazione che identifichi il primo richiedente di inizio conversazione, il mittente sarebbe costretto a fornire il proprio ID X. Questo potrebbe causare un problema di sicurezza (il destinatario Y potrebbe usare l'ID della controparte per fingere di essere X in una futura conversazione), oppure potrebbe semplicemente non essere realizzabile (un firewall potrebbe bloccare determinati messaggi in arrivo). È quindi auspicabile l'utilizzo di due variabili di correlazione, una per X ed una per Y. Per attuare questa variante si potrebbe utilizzare un servizio centrale (SC) che gestisca le richieste di conversazione (come ad esempio un server di Instant Messaging). Alla ricezione di un messaggio di tipo *attendo_conversazione(*$id_1$*)* (con $id_1 = X$), indicante che X è disponibile per iniziare una conversazione, SC inizializzerà una sessione S avente come prima variabile di correlazione $id_1 = X$, e come seconda variabile $id_2$ non definita. Quando Y volesse iniziare una conversazione con un altro utente, potrebbe inviare un messaggio m di tipo *conversazione_casuale(*$id_2$*)* (con $id_2 = Y$). Dato che $id_2$ non è definita, allora m correla con S (o, nell'eventualità, con una qualunque sessione avente $id_2$ non definito). A questo punto, SC memorizza $id_2$ come Y: di conseguenza, entrambi i client potranno, d'ora in avanti, fare riferimento alla sessione S tramite il loro ID.
L'esempio sopra citato è utile a dimostrare l'utilità e le potenzialità di questo meccanismo basato sui Correlation Set. Il costo di questo metodo in termini prestazionali risulta dalla forte dipendenza con il numero di sessioni presenti, poiché si tratta di effettuare una ricerca in un insieme di dimensioni tipicamente elevate. Una stima precisa delle prestazioni dei differenti algoritmi che

gestiscono la correlazione non è semplice da effettuare, dato che la documentazione presente a riguardo è molto scarna ed in alcuni casi l'implementazione degli stessi non è coerente con con le specifiche.

## 2.3 Limiti prestazionali

In realtà, sulla base dell'osservazione del codice sorgente, risulta come la maggioranza delle implementazioni della gestione delle correlazioni seguano due approcci differenti e ben definiti: il primo si basa sull'utilizzo di un array associativo (ad esempio una Hash Table), dove le variabili del Correlation set vengono utilizzate come chiavi di ricerca (PHP e JavaEE, ad esempio, utilizzano questo approccio). Il secondo metodo consiste, invece, nel memorizzare le associazioni Correlation set - sessioni in una struttura a lista, dove la ricerca di una correlazione si riduce alla scansione della lista.
Nell'implementazione corrente di JOLIE si è scelto il secondo approccio, poiché questo è l'unico utilizzabile in presenza di una sola variabile di correlazione. Tecnicamente, la prima soluzione ha prestazioni migliori dato che la complessità non dipende dal numero di sessioni presenti. Il secondo approccio, al contrario, è fortemente dipendente dal numero di sessioni presenti e all'aumentare di quest'ultimo perde efficienza con complessità lineare.
Come si è visto, in un sistema Service-Oriented è di fondamentale importanza la complessità con cui viene gestita la correlazione, dato che è una delle operazioni più eseguite e che una carenza in questo senso comporta una riduzione considerevole nelle prestazioni. Per questo motivo si è voluto creare un nuovo algoritmo che consentisse l'utilizzo di più di una variabile di correlazione, oltre a non avere complessità lineare nel numero di sessioni, e anzi, di complessità tendenzialmente indipendente dal loro numero.
Nel capitolo seguente verrà fornita una visione di insieme dell'algoritmo che si è voluto implementare. Questo algoritmo si basa sui Radix Trees, è indipendente dal numero di sessioni ed ha complessità esponenziale sul numero di variabili di correlazione presenti. Dato che tipicamente il numero di tali

variabili non supera le tre unità, si può dire che l'algoritmo ha complessità costante pari a tale numero.

# Capitolo 3

# Algoritmo "Session correlation management with Radix Tree"

## 3.1 Introduzione all'algoritmo

Come accennato alla fine del capitolo precedente, la complessità attuale dell'implementazione dell'algoritmo che gestisce la correlazione è lineare nel numero di sessioni presenti. Con l'algoritmo presentato dal Dr. Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro e Fabrizio Montesi [15] (che in queste pagine è stato modificato in minima parte) si vuole fornire un'alternativa di complessità molto inferiore, dunque avente prestazioni migliori.
L'idea su cui si basa l'algoritmo per migliorare le prestazioni è quella di utilizzare i Radix Tree per memorizzare la correlazione delle sessioni, in modo da trovare più efficientemente le sessioni che correlano con il messaggio.
Per poter approfondire il meccanismo in questione è necessario prima formalizzare alcuni concetti descritti in precedenza:

- **Correlation Set:** il correlation set (c-set) di un servizio è rappresentato da un insieme finito di variabili $VC = v_1, v_2, ...., v_n$ (dove $v_1...v_n$ possono essere variabili nominate ad esempio come *nome, cognome ecc.*)

- **Istanza (di c-set):** identifica l'associazione tra il valore posseduto

da una data variabile di correlazione e la variabile stessa. L'istanza può essere rappresentata da una funzione di questo tipo: *F(nome) = “Giovanni”*. Questo si può interpretare come “l'istanza della variabile nome possiede il valore *“Giovanni”*”.

- **Definire una variabile:** un messaggio in arrivo potrebbe contenere valori associati a variabili facenti parte del correlation set, quindi, in questo caso, si dice che un messaggio definisce una variabile di correlazione $v$ quando contiene un valore associato ad essa, cioè quando possiede un istanza di $v$. Dato un messaggio $m$ definiamo con $F_m(v)$ l'istanza della variabile $v$ nel messaggio $m$; quando, viceversa, non è presente l'istanza di una variabile $v'$, si dice che essa non è definita. Analogamente, una sessione $s$ definisce una variabile $v$ quando ne possiede un'istanza che denoteremo con $F_s(v)$.

- **Correlazione (Correlation):** ipotizziamo di avere un servizio $S$ comprendente un Correlation set *VC*. In questo caso, si avrebbe una sessione $s$ correlata con un messaggio $m$ se e solo se $\forall v \in VC$ se $F_m(v)$ e $F_s(v)$ sono definiti allora si ha che $F_m(v) = F_s(v)$.
  Di conseguenza, esiste la possibilità che un messaggio correli con più di una sessione, dato che è possibile che esistano variabili non definite. Infatti, la definizione di correlazione non esclude che un messaggio possa correlare tramite le sole variabili che definisce. Si consideri, ad esempio, una situazione in cui due sessioni aventi le variabili del correlation set siano definite nel seguente modo:
  *s : nome = “Mario”* e *cognome = “Rossi”*
  *s' : nome = “Mario”* ma cognome non definito
  Alla ricezione di un messaggio *m( “Mario” , “Rossi” )*, questo potrebbe correlare con entrambe le sessioni, dato che per *s'* la variabile definita è nome e si ha che $F_m(nome) = F_{s'}(nome)$, come da definizione. In realtà, per poter correlare con la sessione più ragionevole ($s$ nell'esempio) si sfrutta un concetto più restrittivo che definiamo qui di seguito.

- **Massima copertura (Maximal coverability):** riprendiamo in esame il servizio $S$ avente il solito set di variabili di correlazione $VC$ e un insieme di sessioni $s_1, ..., s_n$ che correlano con il messaggio $m$. Una sessione $s_i$ ha massima copertura con il messaggio $m$ se e solo se non esiste nessun altra sessione $s_j$ tale che:
  $\forall v \in VC$ se $F_{s_i}(v)$ è definita allora $F_{s_j}(v)$ è definita
  $\exists v \in VC$ tale che $F_{s_i}(v)$ non è definita mentre $F_{s_j}(v)$ e $F_m(v)$ sono definite.
  Data questa definizione, potremmo dire che si creano degli insiemi di sessioni basati sulle caratteristiche delle variabili. Di questi "insiemi" di sessioni, abbiamo la certezza che solo una possa correlare con il messaggio. Questo è garantito dalla definizione stessa delle sessioni, le quali per loro natura possiedono istanze diverse delle variabili del correlation set. In caso contrario si avrebbero sessioni non distinguibili la cui gestione è definita nell'implementazione del linguaggio. Come vedremo, nell'implementazione del nuovo algoritmo di cui andremo a discutere, sono ammesse sessioni col medesimo correlation set a patto che esse siano interscambiabili (cioè non faccia differenza quale delle due si scelga).

## 3.2 Struttura dell'algoritmo

### 3.2.1 Radix Tree

I Radix Tree (o Patricia Tree) [11] sono delle strutture dati ad albero ordinate, utilizzate per memorizzare stringhe di caratteri. In questa struttura ad albero vi è un nodo per ogni prefisso comune, e gli archi dell'albero vengono etichettati per una più veloce identificazione; i dati (in questo caso le sessioni) vengono salvati nelle foglie. I Radix Tree, a differenza dei normali alberi, consentono la memorizzazione nei nodi di serie di caratteri anziché di caratteri singoli. Questa modifica strutturale dà la possibilità di ridurre ulteriormente la ricerca a prefissi comuni, e non solo a caratteri comuni.

Intuitivamente, appare chiaro che la ricerca in una data struttura è indipendente dal numero di foglie, ma dipendente al più dalla lunghezza della stringa cercata, dato che si hanno miglioramenti prestazionali nel caso in cui la stringa sia leggermente scomposta all'interno dell'albero.
A riprova di ciò, potremmo fare il seguente esempio: data una lista di invitati ad una festa, utilizzando l'algoritmo lineare saremmo costretti a scorrere la lista ogni volta alla presenza un nuovo invitato e, nel caso più sfortunato, quest'ultimo potrebbe essere in coda alla suddetta lista. Se la lista fosse organizzata tramite Radix Tree, all'arrivo dell'invitato “Mario Rossi” scorremmo nel caso più sfortunato ogni singola lettera. Questo è esemplificativo di come l'algoritmo lineare sia dipendente dal numero di invitati mentre quello basato sui Radix Tree non ne sia influenzato, e di come quest'ultimo sia molto più efficiente nel memorizzare dati con stringhe (chiavi) che abbiano lunghi prefissi in comune.
Le operazioni di aggiunta di una stringa al Radix Tree, quella di rimozione di una stringa e quella di ricerca hanno un costo computazionale di $O(n)$, nel peggiore dei casi, dove $n$ rappresenta la stringa di lunghezza massima presente nel radix tree.

L'idea su cui si basa l'algoritmo presentato in questa tesi è consiste nello sfruttare questa potenzialità dei radix tree per poter correlare le sessioni con i messaggi in arrivo: si utilizza il valore delle variabili del correlation set come chiave di ricerca. Riassumendo, data una sessione avente variabili del correlation set *nome* e *cognome*, si compongono i valori di tali variabili in modo da ottenere una stringa che verrà utilizzata per memorizzare la sessione all'interno del radix tree.
Nella fattispecie, il tutto è organizzato in *cluster* (gruppi) di radix tree, uno per ogni sottoinsieme di variabili presenti nel correlation set specificato dal comportamento del servizio.

### 3.2.2 Cluster

Come accennato, all'inizializzazione di un servizio viene creato un cluster per ciascun sottoinsieme di variabili che compongono il correlation set (la descrizione dettagliata del funzionamento dell'algoritmo in questo ambito sarà fornita nella sezione 3.3). Ipotizziamo, ad esempio, di avere tre variabili di correlazione *a, b, c*: per poter coprire tutti i sottoinsiemi di variabili (*[a]; [b]; [c]; [a,b]; [a,c]; [b,c]; [vuoto]*), come risultato otterremmo $2^n$ cluster, ed in questo caso $2^3 = 8$.
Di seguito indicheremo con *C(v,...,v')* il cluster che "copre" le variabili *v,....,v'*. La "copertura" serve ad indicare che nel suddetto cluster verranno memorizzate le sole sessioni che definiscono tali variabili. Riprendendo l'esempio precedente, i cluster risultanti dalla considerazione di tre variabili (*a, b, c*) sono: *C(a); C(b); C(c); C(a,b); C(a,c); C(b,c); C(a,b,c)*, e per ultimo il caso vuoto *C(empty)*. Di conseguenza, nel cluster *C(c)* avremo le sole sessioni che definiscono la sola variabile di correlazione *c*, in *C(a,b)* le sessioni che definiscono solo le variabili *a* e *b* e via dicendo.
Alla ricezione di un messaggio, si verificheranno quali variabili del correlation set questo definisce. Grazie a queste informazioni, verrà costruita una "lista di ricerca" nei cluster, a cominciare da quello con massima copertura (vedi 3.1), e seguendo con tutti gli altri, fino a trovare una sessione che correli e che dovrà ricevere il messaggio. Nel caso in cui non venga trovata alcuna sessione, l'algoritmo non genererà un errore: si limiterà invece a notificare il fatto che nessuna sessione è stata trovata.

### 3.2.3 Generazione delle chiavi

Per creare una chiave che sia utile alla ricerca all'interno dei radix tree, sono state concatenate le stringhe dei valori delle variabili coperte dal cluster, intervallate dal carattere # (il cui utilizzo è vietato altrove). Poniamo, ad esempio, un radix tree che memorizza le sessioni specificanti *nome*, *cognome*.

Si vogliono inserire ora quattro nuove sessioni all’interno di tale radix tree:

- $S_1$ avente *nome* = *“Dante”* e *cognome* = *“Alighieri”*
- $S_2$ avente *nome* = *“Dante”* e *cognome* = *“Alfieri”*
- $S_3$ avente *nome* = *“Danilo”* e *cognome* = *“Alighieri”*
- $S_4$ avente *nome* = *“Dante”* e *cognome* = *“Alighiero”*

Le stringhe utilizzate come chiavi per le quattro sessioni risulteranno quindi essere:

- per $S_1$ “#Dante#Alighieri#”
- per $S_2$ “#Dante#Alighieri#”
- per $S_3$ “#Danilo#Alighieri#”
- per $S_4$ “#Dante#Alighiero#”

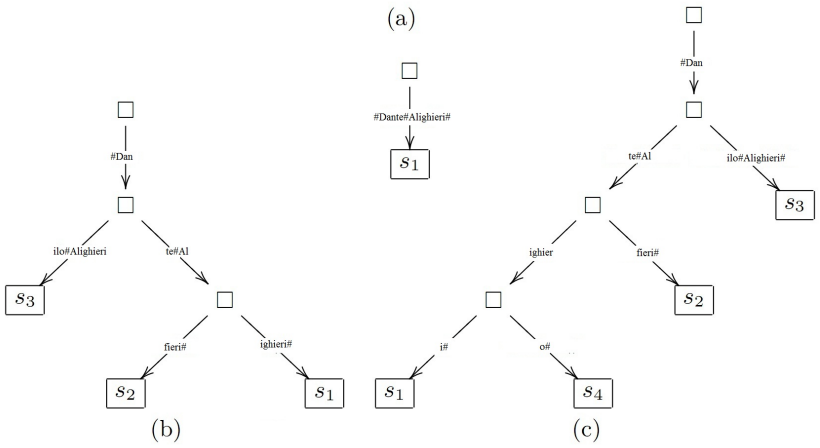


Figura 3.1: Esempio composizione di un Radix Tree

In seguito ad una attenta analisi, risulta evidente come questo meccanismo di concatenazione di stringhe abbia un punto debole nel momento in cui le variabili della sessione sono più di una. Infatti, come vedremo più avanti,

i cluster necessitano di più radix tree per poter correlare, nel caso in cui essi "coprano" più variabili.
Ipotizzando la presenza di una sola variabile all'interno del correlation set (ad esempio nome), si ha $2^1 = 2$ cluster, cioè *C(nome)* e *C(empty)*. Inizialmente, all'interno del cluster *C(nome)* è presente un radix tree vuoto, composto dal solo nodo radice. Sono create in seguito due sessioni *S* e *S'*, aventi la variabile nome memorizzata rispettivamente come *"#Dante#"* e *"#Danilo#"*. Si può notare come, negli angoli dell'albero, siano presenti i prefissi comuni di maggiore lunghezza: all'arrivo di un messaggio *m* che definisca il valore "#Danilo#" per la variabile *nome*, risulta evidente dalla figura che, per effettuare la ricerca della sessione che correli con *m*, è sufficiente scorrere l'albero del cluster *C(nome)* nei suoi nodi. Nell'ipotetico caso in cui non fosse possibile trovare una sessione correlante all'interno del cluster *C(nome)*, avremmo in risposta una qualunque delle sessioni presenti in *C(empty)*. Le sessioni presenti in quest'ultimo cluster, infatti, hanno la medesima copertura: l'una o l'altra, dunque, è indifferente.
Come accennato, la presenza di più di una variabile di correlazione genera problemi nella ricerca delle sessioni correlanti. Riprendiamo l'esempio in cui sono presenti due variabili nel correlation set (*nome* e *cognome*) e le quattro sessioni definite in precedenza ($S_1, ..., S_4$): i cluster risultanti dalla presenza di due variabili sono $2^2 = 4$, nella fattispecie *C(empty), C(nome), C(cognome) e C(nome, cognome)*. Le quattro sessioni ($S_1, ..., S_4$), definendo entrambe le variabili, sono memorizzate tutte all'interno di *C(nome, cognome)*, mentre gli altri cluster rimangono vuoti. Alla ricezione di un messaggio specificante, ad esempio, solo il valore *"alfieri"* per la variabile del correlation set *cognome*, viene effettuata una prima ricerca all'interno del cluster *C(cognome)*, la quale non avrà, però, esito positivo. Si prosegue, dunque, con il cluster successivo avente la massima copertura in questo caso *C(nome, cognome)*. Se per tale cluster (che comprende più di una variabile) si considera la presenza di un solo radix tree (il quale possiederebbe un struttura rappresentabile nel seguente modo: *#nome#cognome#*), risulterà evidente come esso non possegga un

prefisso composto dal cognome specificato nel messaggio. In realtà, nessun cognome potrà essere utilizzato come prefisso in una siffatta struttura. Dalla definizione di correlazione (vedi 3.1), però, la sessione $S_2$ correlerebbe con il messaggio, dimostrando così la necessità di avere più di un radix tree per tale cluster. Per risolvere questo problema, dunque, sarebbe necessario avere un radix tree per ogni sottoinsieme delle variabili definite in tale cluster: in realtà, come vedremo in seguito, è sufficiente un sottoinsieme di tale numero di radix tree per coprire tutte le variabili specificate nel cluster.

### 3.2.4 Generazione dei radix tree

Tendenzialmente, il numero dei radix tree di un cluster sarebbe esponenziale secondo il numero di variabili definite: se un cluster memorizza sessioni che definiscono $n$ variabili, si dovrebbero avere, teoricamente, $2^n$ radix tree. L'algoritmo di generazione dei radix tree RTG (*Radix Trees Generator*) che andremo a presentare ha lo scopo di ridurre al minimo il numero di radix tree che è necessario creare per ciascun cluster. Prendendo come esempio il cluster *C(nome, cognome)*, si può notare come il radix tree con struttura *#nome#cognome#* copra contemporaneamente il caso di un messaggio specificante *nome* e *cognome*, un messaggio specificante il solo *nome*, dato che quest'ultimo può essere comunque utilizzato come prefisso, e per finire anche un messaggio che non definisce alcuna variabile.
Di seguito si fornisce una formalizzazione del concetto: dato un cluster avente un numero di variabili $n$, si definisce $SEQ_i$ una sequenza $v_1, ..., v_n$ di variabili, dove ogni $SEQ_i$ è composto da un prefisso della sequenza ($SEQ_{i+1}$) successiva.
Si consideri, ad esempio, un cluster *C(nome, cognome, età)*. Alcune delle sequenze risultanti potrebbero essere:

- $SEQ_1$ : *nome*
- $SEQ_2$ : *nome*, *cognome*
- $SEQ_3$ : *nome*, *cognome*, *età*

Per rappresentare la struttura di un radix tree, si utilizzerà dunque la notazione $RT(SEQ_1, ..., SEQ_m)$ con $SEQ_m = v_1, ..., v_h$ (con $h \leq n$), stante ad indicare un radix tree della forma $\#v_1\#...\#v_h\#$. Con la dicitura $RT(SEQ_1, ..., SEQ_m)$, invece, si vogliono invece indicare, oltre alla struttura del radix tree, le variabili di correlazione che esso dà la possibilità di controllare. Un radix così descritto dà la possibilità di controllare l'esistenza di una sessione che definisca ciascuna delle combinazioni di variabili descritte dalle sequenze $SEQ_i$. Si prenda, ad esempio, il cluster descritto in precedenza. Uno dei radix tree contenuto al suo interno potrebbe essere rappresentato come:

- *RT(empty, [nome], [nome, cognome], [nome, cognome, età])*

Usando questa notazione, si può affermare il bisogno di un numero minimo di schemi di radix tre *RT*, numero tale per cui per ogni sottoinsieme $I$ di variabili definite nel cluster esista una sequenza $SEQ_j$ che contenga tutte e le sole variabili in *I*.

### 3.2.5 Algoritmo RTG (*Radix Trees Generator*)

per risolvere questo problema, è utile una rappresentazione tramite grafo. Dato un insieme di variabili *V*, si crea un grafo *G(V)*; si denota poi con *S(V)* l'insieme composto da tutti i sottoinsiemi di variabili di *V*. Si ha inoltre che:

- i nodi sono etichettati con gli elementi di *S(V)*
- vi è un arco da $u$ a $v$ se le variabili contenute in $u$ sono presenti anche in $v$ ($u, v \in S(V)$)
- l'arco ($u$,$v$) è etichettato con le variabili con cui differiscono $u$ e $v$ (usando notazione insiemistica possiamo indicare con $u$ / $v$)

Nella figura seguente, è rappresentato un esempio di come risulterebbe un grafo costruito partendo dalle variabili a, b e c. Per problemi legati alla disponibilità di spazio, non sono state inserite tutte le etichette degli archi.

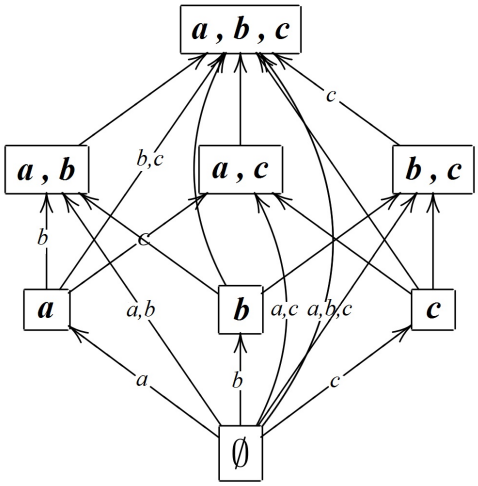


Figura 3.2: Esempio di grafo G(V)

Ogni cammino del grafo rappresenta un possibile schema di radix tree *RT*. Data questa rappresentazione, possiamo ridurre il problema alla ricerca del numero minimo di cammini necessari alla copertura di tutti i nodi.
L'algoritmo RTG (del quale è stata data un'implementazione descritta in 3.3) opera nel seguente modo:
Il grafo *G(V)* viene suddiviso in livelli. Ciascun livello è costituito dal numero di variabili che ogni nodo contiene, cosicché il livello $i$ è formato dai nodi aventi $i$ variabili (Il livello con il nodo vuoto è il livello 0). Si considerino i livelli a coppia partendo da quelli più in basso (i.e. Livelli 0 e 1). Le coppie di livelli $i$ e $i+1$ sono considerati come un grafo bipartito diviso in base ai nodi dei due livelli. *Secondo la teoria dei grafi, un grafo bipartito è un grafo non orientato tale per cui l'insieme dei suoi vertici si può partizionare in due sottoinsiemi; questi insiemi, a loro volta, sono tali per cui ogni vertice di ognuna di queste due parti è collegato solo ai vertici dell'altra.* Da questo grafo così creato sono eliminati gli archi seguendo l'algoritmo di massima avvicinabilità (*maximum bipartite matching*). Si ha un *maximum matching* quando i nodi di un grafo sono collegati tramite un arco o, al più, con un altro nodo, e contemporaneamente il numero di nodi collegati tra loro è il più

alto possibile. Il *maximum bipartite matching* è, invece, il *maximum matching* applicato ad un grafo bipartito. Questa scelta degli archi è ripetuta nei livelli successivi $i+1$ con $i+2$, $i+2$ con $i+3$ ed a seguire, fino alla fine del grafo, cioè al livello $n$. Il grafo $G'(V)$ così ottenuto conterrà il numero minimo di cammini che coprono tutti gli insiemi di variabili in $S(V)$.
Prima di fornire l'algoritmo in pseudocodice, verrà esposta la notazione utilizzata. Si indicherà con $S$ l'insieme delle variabili del cluster (ad esempio *a, b, c*): la funzione $level(x)$ restituirà l'elenco di nodi al livello $x$ nel grafo, cioè l'insieme di nodi con cardinalità $x$; con $bipartite(A, A')$ si indicherà il grafo bipartito creato partendo dai due insiemi di variabili $A$ e $A'$: la funzione $maximum\_matching(B)$ restituirà uno dei maximum matching del grafo bipartito $B$; $M$ rappresenterà il grafo in costruzione.

```
Radix_Trees_Generator(S){
  i = 0;
  M = empty;
  while (i < n){
    L = level(i);
    L' = level(i+1);
    G = bipartite( L, L' );
    M' =  maximum_matching(G);
    M = M U M';
    i++;
  }
  return (M);
}
```

Una volta ottenuto il grafo dall'algoritmo RTG, saràpossibile creare gli schemi dei radix tree seguendo i cammini di tale grafo. Si ricordi, a questo proposito, che uno schema di radix tree è della forma $RT(SEQ_1, ..., SEQ_m)$ con ogni sequenza $i$ di lunghezza minore rispetto alla successiva $i+1$, e che ogni nodo del grafo è rappresentabile come una sequenza di variabili.
Consideriamo, ad esempio, il grafo con tre variabili *a, b, c* mostrato nella figu-

ra precedente. Nella figura successiva si mostrerà una possibile esecuzione dell’algoritmo in questione: si utilizzerà la notazione $\rightarrow$ per indicare gli archi considerati dall’algoritmo di maximum matching (i.e. gli archi in $G$), mentre con $\Rightarrow$ quelli che verranno scelti.(i.e. gli archi in $M'$). Inoltre, solo i nodi racchiusi in linee continue saranno presi in considerazione (i.e. i nodi in $L$ e $L'$). Infine, i nodi processati saranno indicati con una linea tratteggiata.

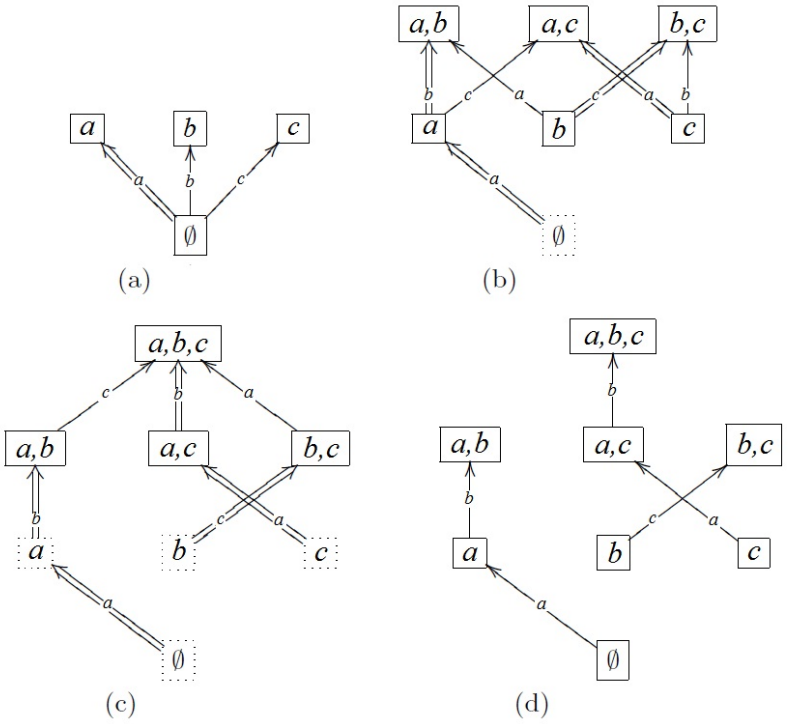


Figura 3.3: Esempio di esecuzione dell’algoritmo RTG

Dopo aver fornito questo formalismo, si può fornire un esempio di esecuzione dell’algoritmo. Ipotizziamo di avere tre variabili $a$, $b$ e $c$: uno dei risultati ottenibili è una situazione che presenta i seguenti cluster con corrispettivi radix tree, dove gli elementi tra parentesi quadre (i.e. “*[a]*”) rappresentano le sequenze ($SEQ_i$) descritte in precedenza:

- $C(empty) : RT([empty])$
- $C(a) : RT([empty, [a])$

- $C(b) : RT([empty, [b])$
- $C(c) : RT([empty, [c])$
- $C(a, b) : RT([empty, [a], [a, b]); RT([b])$
- $C(a, c) : RT([empty, [a], [a, c]); RT([c])$
- $C(b, c) : RT([empty, [b], [b, c]); RT([c])$
- $C(a, b, c) : RT([empty], [a], [a, b], [a, b, c]); RT([b], [b, c]); RT([c], [c, a])$

Si noti come le variabili nella seconda sequenza del terzo radix tree del cluster $C(a, b, c)$ siano in ordine invertito, cosicchè la prima sequenza ($[c]$) sia un prefisso della seconda ($[c, a]$).

## 3.3 Utilizzazione dell'algoritmo

Ottenuta la struttura dell'algoritmo presentato in questa tesi mediante i procedimenti esposti al punto 3.3, si vuole ora fornire un quadro del suo utilizzo.

**Gestione delle sessioni** L'algoritmo RTG presentato in precedenza è utilizzato in ciascun cluster. L'esempio di esecuzione raffigurato nell'immagine precedente, infatti, è uno dei possibili risultati dati dall'utilizzo di suddetto algoritmo all'interno del cluster $C(a, b, c)$.
Per gestire una tale struttura, sono necessarie funzioni che consentano di aggiungere, rimuovere e trovare sessioni all'interno dei vari cluster e dei vari radix tree. Il concetto su cui ci si basa è quello di *propagazione delle informazioni* dei cluster ai radix tree che essi contengono.
Le funzioni di base per la manipolazione dei cluster sono:

- **CLS.add(S):** è l'operazione che si occupa della memorizzazione di una sessione $S$ all'interno del cluster $CLS$. Ciò è effettuato acquisendo le variabili $V$ del correlation set di $S$ e creandone delle stringhe (come

descritto in 3.2.3 *[Generazione delle chiavi]*), seguendo le strutture dei radix tree presenti all'interno del cluster. Per ciascuno di questi radix tree, si utilizza l'operazione $RT.add(S)$, la quale memorizza all'interno di $RT$ la sessione utilizzando come chiave le stringhe ottenute al passaggio precedente. Si supponga, ad esempio, di avere un cluster $C(SEQ_h)$ con $SEQ_h = V_1, ..., V_h$, e che si voglia inserire una sessione $S'$: si inizierà utilizzando la funzione $C(SEQ_h).add(S')$ la quale propagherà la richiesta ad ogni radix tree presente all'interno del cluster utilizzando come chiave il valore delle variabili composte, secondo lo schema del radix tree. Prendiamo in considerazione uno di questi radix tree $RT$ avente come struttura $RT(SEQ_1, ..., SEQ_h)$. Quando è utilizzata la $RT.add(S')$, si crea la stringa seguendo la struttura $\#v_1\#...\#v_h\#$, dove $v_1, ..., v_h$ sono i valori delle variabili $V_1, ..., V_h = SEQ_h$ (la sequenza di lunghezza maggiore di $RT$).

- **CLS.del(S):** quest'operazione sfrutta lo stesso procedimento illustrato per la $CLS.add(S)$ per rimuovere una sessione $S$ anziché aggiungerla.

- **CLS.find(M):** restituisce la sessione che correla con il messaggio $M$. Se non sono state trovate sessioni che correlino con $M$, è restituito “$nulla''$. Come noto, in realtà, un messaggio $m$ definisce un insieme di variabili di correlazione $V_m$, i quali possono non comprendere tutte le variabili definite all'interno del cluster $V_c$. La funzione $CLS.find(M)$ sceglie il radix tree su cui propagare la richiesta scegliendo $RT$ avente $SEQ_i = V_m \cap V_c$ (cioè il radix tree che possiede la sequenza di variabili risultante dall'intersezione tra le variabili di correlazione del messaggio e quelle del cluster). La $RT.find(m)$ risultante, inoltre, può essere applicata se e solo se la sequenza di variabili prese dal messaggio $SEQ_m = x_1, ..., x_m$ è in un ordine tale da rappresentare un prefisso per il radix tree $RT$. In questo caso, la chiave di ricerca risultante risiederà nella forma $\#x_1\#...\#x_m\#$, e la $RT.find(m)$ restituirà una delle sessioni correlanti con tale stringa. Trattandosi della ricerca di un

prefisso, infatti, si potrebbe trovare più di una sessione correlante ma, come si è visto, tali sessioni sono indifferenti l'una dall'altra per quanto concerne la correlazione.

Si possiedono ora elementi sufficienti per fornire una formalizzazione esplicita del funzionamento dell'algoritmo discusso in questa dissertazione. Sono forniti, di seguito, gli algoritmi in pseudocodice delle funzioni $find\_session(m)$ e $add\_session(S)$ (non sarà fornito lo pseudo codice delle $delete\_session(S)$ dato che è intuibile da quest'ultima). Successivamente, verranno chiarite le funzioni utilizzate.

- $allCluster()$ : restituisce l'elenco dei cluster presenti nel servizio
- $getBestMatchingCluster(m, C)$ : cercando all'interno dell'elenco di cluster $C$ questa funzione restituisce il cluster $C'$ avente il numero maggiore possibile di variabili in comune con il messaggio (ad esempio se si ha un messaggio $m(a, b)$ la funzione restituirà il cluster $C'(a, b)$ oppure, nel caso quest'ultimo non sia presente in $C$, verà restituito il cluster $C''(a, b, c)$ e così via).
- $getBestMatchingRT(m, C')$ : ha lo scopo di restituire il radix tree $RT$ che possegga la sequenza di variabili $SEQ_m$ specificate dal messaggio.

```
find_session(m){
  C = allCluster();
  S = null;
  while (S == null or C != null){
    C' = getBestMatchingCluster(m, C);
    RT = getBestMatchingRT(m, C');
    S = RT.find(m)
    if( S == null){
      C = C - C';
    }
  }
```

```
  Return S;
}
```

Una volta controllato un cluster, esso sarà eliminato dalla lista ( $C = C - C'$ ), in modo da non dover essere più controllato.
La funzione *add_session*($s$) risulta essere molto simile alla precedente, in quanto necessita anch'essa del cluster avente il numero massimo possibile di variabili in comune con la sessione passata come parametro. Di seguito è presentata la stessa notazione dell'operazione precedente.

```
add_session(s){
  C = allCluster();
  C' = getBestMatchingCluster(s, C);
  forall RT in C' {
   RT.add(s);
  }
}
```

Intuitivamente, per ottenere lo pseudocodice della funzione *delete_session*($s$) è sufficiente sostituire l'operazione $RT.add(s)$ con una $RT.del(s)$.

# Capitolo 4

# Algoritmo "Session correlation management with Radix Tree": un'implementazione

In questo capitolo è fornita un'implementazione dell'algoritmo discusso nel capitolo precedente. Questa tesi è stata sviluppata utilizzando il linguaggio di programmazione JAVA [12]. L'algoritmo qui presentato è stato creato in JAVA ai fini di migliorare le prestazioni del linguaggio JOLIE, integrando l'algoritmo stesso a questo nuovo linguaggio. Come discusso nel capitolo 2, infatti, tale linguaggio di programmazione Service-Oriented è stato sviluppato in JAVA; inoltre, l'algoritmo di cui dispone attualmente per la gestione della correlazione ha complessità lineare, ed è quindi meno efficiente dell'algoritmo qui presentato.
I concetti e la struttura esposti nel capitolo precedente sono stati rappresentati mediante tre insiemi di classi: l'inseme denominato *Cluster Manager* racchiude il gestore generale delle componenti dell'algoritmo; l'inseme *Cluster* contiene l'implementazione del concetto di Cluster (vedi 3.2); l'insieme *radix tree* comprende l'elenco dei radix tree.
Si indicherà con "[...]" l'interruzione/proseguimento dell'esposizione del codice (per dare modo di commentarlo in sezioni), mentre si indicherà con "[###]"

l’omissione di codice (perché non utile ai fini esplicativi).

## 4.1 Cluster Manager

Ecco implementate le funzioni $add\_session(s)$, $delete\_session(s)$ e $find\_session(m)$ descritte al punto 3.3.

### 4.1.1 ClustersManager.java

Di seguito si fornisce l’interfaccia descritta nella classe *ClustersManager.java*

```
public interface ClustersManager {
  /**
   * Store a session s
   *
   * @param s the session to be stored
   */
   public void add_session(Jolie_Session s);

  /**
   * Remove a session s
   *
   * @param s the session to be removed
   */
  public void del_session(Jolie_Session s);

  /**
   * Returns the session which correlate with a message m. If no sessions
   * correlates with m then null is returned.
   *
```

```
 * @param m the message to correlate with
 * @return the session that correlates with the massage
 *  returns null if nothing found
 */
public Jolie_Session find_session(Jolie_Message m);

/**
 * For printing purpose
 */
public void prettyPrint();

/**
 * Checks if two list have the same variables
 */
public boolean isMaxMatch(List<Jolie_Var> lv1, List<Jolie_Var> lv2);

/**
 * Returns number of common variables between two list
 */
public int numMatchingVar(List<Jolie_Var> lv1, List<Jolie_Var> lv2);


/**
 * JOLIE INTEGRATION SECTION
 */
public List<Jolie_Var> getSessionCsetVar(Jolie_Session s);

public List<Jolie_Var> getMessageCsetVar(Jolie_Message m);

public String getVarName(Jolie_Var v);
```

```
  public String getVarValue(Jolie_Var v);

  public String getSessionID(Jolie_Session s);
}
```

L’interfaccia definisce il comportamento generale su cui si basa l’algoritmo. Come da specifiche Java, infatti, uno degli scopi delle interfacce è definire le linee guida da seguire per implementare un algoritmo. Le funzioni descritte andranno poi implementate in sotto-classi, le quali definiranno effettivamente il loro comportamento. Come si può notare, nalla sezione commentata come *JOLIE INTEGRATION SECTION* è stato fornito un formalismo per implementare in modo immediato una futura integrazione con il linguaggio JOLIE (o con un qualunque altro linguaggio sviluppato in Java e strutturalmente simile a JOLIE). Attualmente, infatti, l’implementazione fornita gestisce classi create ad hoc che imitano la struttura e il comportamento delle rispettive classi JOLIE. Tali strutture e comportamenti sono limitati allo stretto indispensabile, così da rendere possibile un’integrazione.
Le classi in questione sono:

- **Jolie_Message.java**: possiede un elenco di variabili che rappresentano, a loro volta, le variabili del correlation set. è presente, inoltre, un meccanismo utile ad ottenere tali variabili. è utilizzata come parametro nella $find\_session(m)$.

- **Jolie_Var.java**: Implementa il concetto di variabile e, più specificatamente, è utilizzata per indicare le variabili di correlazione. Possiede un nome ed un valore reperibili mediante apposite funzioni.

- **Jolie_Session.java**: rappresenta una sessione e, come le variabili, è caratterizzata da un nome. Lo stato della sessione è indicato in una lista di variabili di cui è stato fornito un *getter* (i.e. Funzione specificatamente creata per restituire una data variabile)

### 4.1.2 ClustersManagerLinear.java

Per motivi di test sono state sviluppate due implementazioni dell'interfaccia *ClustersManager*. Nella classe *ClustersManagerLinear.java*, è stato sviluppato l'algoritmo con complessità lineare, così da poterlo confrontare con l'algoritmo presentato in questa tesi.
Di seguito vediamo una porzione del codice:

```
public class ClustersManagerLinear implements ClustersManager {
  List<Jolie_Session> list;

  public ClustersManagerLinear() {
    this.list = new ArrayList<Jolie_Session>();
  }

  public void add_session(Jolie_Session s) {
    this.list.add(s);
  }

  public void del_session(Jolie_Session s) {
    this.list.remove(s);
  }

  [...]
```

Appare chiaro, dal codice, che le sessioni sono memorizzate in una lista: di conseguenza, un accesso per la ricerca di una sessione specifica necessita, nel peggiore dei casi, della scansione della lista intera.

### 4.1.3 ClustersManagerImpl.java

La classe che implementa l'interfaccia seguendo il comportamento dell'algoritmo presentato in questa tesi è descritta in *ClustersManagerImpl.java*.

Per questioni di lunghezza, il codice sarà discusso in sezioni e non nella sua interezza:

```
public class ClustersManagerImpl implements ClustersManager {
  private List<Cluster> clusters;

  /**
   * @param list
   *  list of all correlation set variables in the Jolie program
   */
  public ClustersManagerImpl(List<Jolie_Var> list) {
    this.clusters = new ArrayList<Cluster>();
    init(list, new ArrayList<Jolie_Var>(), 0, list.size());
  }
  [...]
```

Il costruttore della classe ha il compito di inizializzare la struttura dati in base alle variabili presenti nel correlation set. Tali variabili, infatti, sono alla base della creazione dei Cluster, effettuata dalla funzione *init*(). Questa funzione ha lo scopo di creare la lista dei cluster in base ad ogni singolo sottoinsieme di variabili (vedi 3.2.2 [Cluster]). Qui di seguito si presenta il codice di tale funzione.

```
  [...]
  /**
   * Creates a list of Cluster from all subsets of a list of variables
   * (i.e.: init(<x,y>) => C(empty), C(x), C(y), C(x,y) )
   */
  private void init(List<Jolie_Var> l, List<Jolie_Var> acc, int i, int n) {
    if (i == n) {
      Cluster c;
      if(acc.size() == 0)
        c = new ClusterEmpty(acc, this);
```

```
      else
        c = new ClusterImpl(acc, this);

      this.clusters.add(c);
    }
    else {
      init(l, acc, i + 1, n);

      List<Jolie_Var> tmp = new ArrayList<Jolie_Var>();
      tmp.addAll(acc);
      tmp.add(l.get(i));
      init(l, tmp, i + 1, n);
    }
  }
  [...]
```

Essendo questa la classe su cui è necessario interfacciarsi per sfruttare le potenzialità dell'algoritmo, questa implementa le tre funzioni principali atte alla gestione delle sessioni. Come si nota dal codice seguente, lo scopo di questa classe è di reperire il cluster che copre la maggior parte delle variabili, per poi propagarvi la richiesta.

```
  [...]
  public void add_session(Jolie_Session s) {
    List<Jolie_Var> l = getSessionCsetVar(s);
    findMaxCoveringCluster(l).addSession(s);
  }

  public void del_session(Jolie_Session s) {
    List<Jolie_Var> l = getSessionCsetVar(s);
    findMaxCoveringCluster(l).delSession(s);
  }
```

```
public Jolie_Session find_session(Jolie_Message m) {
  List<Jolie_Var> l = getMessageCsetVar(m);
  Cluster c = findMaxCoveringCluster(l);
  Jolie_Session s = c.findSession(m);

  if(s == null){
    List<Cluster> toCheck = new ArrayList<Cluster>();
    toCheck.addAll(this.clusters);
    toCheck.remove(c);
    s = findMaxCoveringSession(l, toCheck, m, l.size());
  }
  return s;
}
[...]
```

Il compito di reperire il cluster più adatto a soddisfare la richiesta è implementato nel metodo seguente:

```
[...]
/**
 * finds the maximal matching Cluster for given variables
 * (i.e: findCluster(<x,y>) => C(x,y) )
 */
private Cluster findMaxCoveringCluster(List<Jolie_Var> messageVar) {
  Cluster emptyCluster = null;
  for (Cluster c : this.clusters) {
    if (c.getClusterVariables().size() == 0) {
      emptyCluster = c;
    }
    if(isMaxMatch(c.getClusterVariables(), messageVar)){
      return c;
    }
  }
```

```
    return emptyCluster;
  }
  [...]
```

Data la lista di variabili, la funzione scansiona tutti i cluster in cerca di quello che ha lo stesso set di variabili fornite; in caso contrario, restituisce di default il cluster $C(empty)$. L'implementazione della funzione $isMaxMatch()$ utilizzata precedentemente è la seguente:

```
[...]
/**
 * Checks if two list have the same variables
 */
public boolean isMaxMatch(List<Jolie_Var> lv1, List<Jolie_Var> lv2){
  if(lv1.size() == lv2.size()){
    if(lv1.size() == 0)
      return true;
    if(numMatchingVar(lv1, lv2) == lv1.size())
      return true;
  }
  return false;
}

/**
 * Returns number of common variables between two list
 */
public int numMatchingVar(List<Jolie_Var> lv1, List<Jolie_Var> lv2){
  int numMatch = 0;

  for (Jolie_Var v1 : lv1) {
    for (Jolie_Var v2 : lv2) {
      if (getVarName(v1) == getVarName(v2))
        numMatch++;
```

```
      }
    }
    return numMatch;
  }
  [...]
```

Come si nota dal codice finora esposto, si fa un utilizzo massiccio di funzioni per il reperimento di informazioni riguardanti le sessioni e le variabili. Queste funzioni sono implementate nella sezione dedicata all’integrazione con il linguaggio di programmazione (JOLIE).
Di seguito è fornita l’implementazione utilizzata in questo progetto, la quale sarà sostituita con una specifica, a seconda del linguaggio in cui questo algoritmo voglia essere integrato.

```
[...]
  /**
   * JOLIE INTEGRATION SECTION
   */

  public List<Jolie_Var> getSessionCsetVar(Jolie_Session s) {
    return s.getVar();
  }

  public List<Jolie_Var> getMessageCsetVar(Jolie_Message m) {
    return m.getVar();
  }

  public String getVarName(Jolie_Var v) {
    return v.getName();
  }

  public String getVarValue(Jolie_Var v) {
    return v.getValue();
```

```
}

public String getSessionID(Jolie_Session s){
  return s.getName();
}
[###]
```

Un quadro generale della sezione discussa finora è visibile in modo chiaro e diretto grazie all'utilizzo del seguente diagramma UML [13].

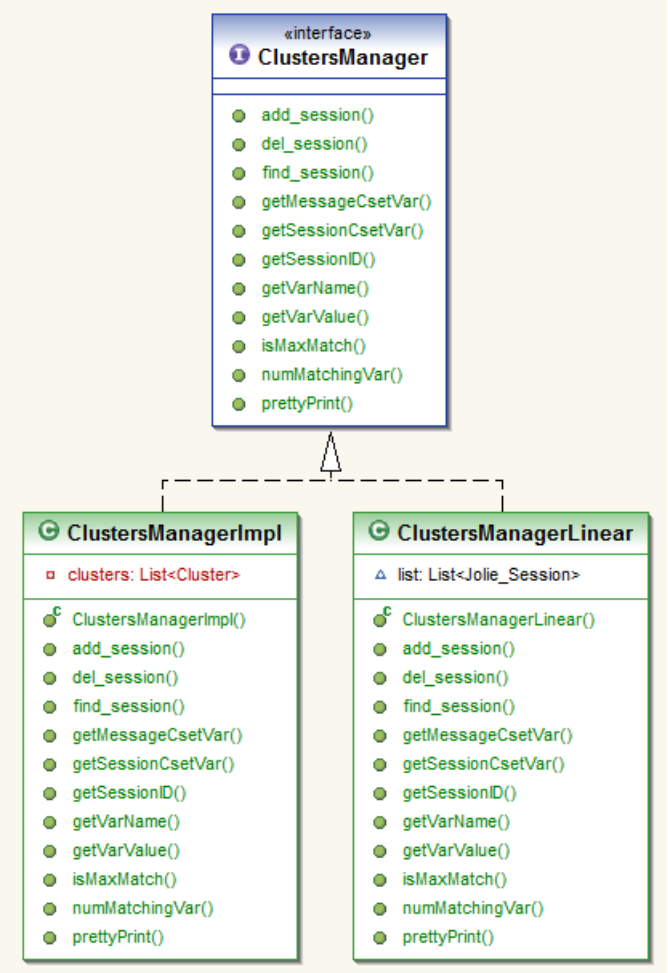


Figura 4.1: Diagramma UML della sezione CLuster Manager

## 4.2 Cluster

Il compito di un cluster è di memorizzare le sessioni che definiscono un set specifico di variabili indicate nella struttura del cluster stesso. Inoltre, è indispensabile, per un cluster, fornire funzioni che consentano di manipolare tale elenco di funzioni.

### 4.2.1 Cluster.java

Nell’interfaccia fornita di seguito (presente nel file *Cluster.java*) si definiscono le funzioni su cui il clustersManager propaga le richieste.

```
public interface Cluster {
  /**
   * Stores a session into the cluster
   *
   * @param s the session to be stored
   */
  public void addSession(Jolie_Session s);

  /**
   * Removes a session from the cluster
   *
   * @param s the session to be removed
   */
  public void delSession(Jolie_Session s);

  /**
   * Finds the session that correlates with provided message
   *
   * @param m the message to correlate with
   * @return the session that correlates with provided massage
   *  returns null if nothing found
```

```
   */
  public Jolie_Session findSession(Jolie_Message m);

  /**
   * For printing purpose
   *
   * @return the string to be printed
   */
  public String prettyPrint();

  /**
   * Gets the cluster's variable structure
   *
   * @return list of cluster's variable
   * (i.e. for cluster C(x,y) returns <x,y>
   */
  public List<Jolie_Var> getClusterVariables();
}
```

### 4.2.2 ClusterEmpty.java

La semplicità concettuale del cluster $C(empty)$ contenente sessioni non specificanti variabili di correlazione ha portato alla creazione di un implementazione separata rispetto a quella degli altri cluster. In questo cluster, data l'assenza di variabili su cui basarsi, l'implementazione mediante radix tree non era fattibile. Le sessioni memorizzate in tale cluster, inoltre, hanno lo stesso livello di correlazione. Si è scelto, dunque, di implementare, in questo caso, mediante la memorizzazione delle sessioni in una struttura a lista gestita tramite hash table.

```
public class ClusterEmpty implements Cluster {
  /**
   * Empty Cluster uses an hash table instead of radix tree
```

```
 */
private Map<String, Jolie_Session> sessions;
/**
 * The cluster's structure (i.e. C(<empty>))
 */
private List<Jolie_Var> clusterVariables;
/**
 * Cluster's manager
 */
private ClustersManager manager;

public ClusterEmpty(List<Jolie_Var> list, ClustersManager cm) {
  this.clusterVariables = list;
  this.manager = cm;
  this.sessions = new HashMap<String, Jolie_Session>();
}

public void addSession(Jolie_Session s) {
  this.sessions.put(manager.getSessionID(s), s);
}

public void delSession(Jolie_Session s) {
  this.sessions.remove(manager.getSessionID(s));
}

public Jolie_Session findSession(Jolie_Message m) {
  if(this.sessions.isEmpty())
    return null;
  else{
    /* returns the first session stored */
    return this.sessions.get(
```

```
        this.sessions.keySet().iterator().next()
      );
    }
  }

  public List<Jolie_Var> getClusterVariables() {
    return this.clusterVariables;
  }

  public String prettyPrint() {
    return "empty";
  }
}
```

Dal codice, appare chiaro che, nel caso di un cluster $C(empty)$, la memorizzazione (tramite $addSession(s)$) e la rimozione (tramite $delSession(s)$) di una sessione sono effettuate rispettivamente tramite una put e una remove su una hash table. La ricerca di una sessione ha un costo computazionale $O(1)$ dato che viene restituita la prima sessione in lista (in quanto ogni sessione è indifferente dall'altra per quanto riguarda la correlazione).

### 4.2.3 ClusterImpl.java

Per quanto risulti immediata e semplice la gestione di un cluster $C(empty)$, risulta allo stesso modo complessa negli altri casi. Tutti gli altri cluster, infatti, implementano il meccanismo di gestione delle sessioni tramite radix tree.
Per gestire la struttura a grafo su cui si basa l'algoritmo di generazione dei radix tree (vedi 3.2.5 [*algoritmo RTG*]), sono state create due classi ad hoc *GraphBuilder* e *GraphNode*.

### GraphNode

Di seguito, una porzione del codice (quella più significativa) della classe GraphNode.

```
class GraphNode {
  List<Jolie_Var> key;
  List<GraphNode> children;
  boolean chosen;

  public GraphNode(List<Jolie_Var> k) {
    this.key = k;
    this.chosen = false;
    this.children = new ArrayList<GraphNode>();
  }

  public void addChild(GraphNode n) {
    this.children.add(n);
  }

  public boolean wasChosen() {
    return this.chosen;
  }

  public boolean choose() {
    if (this.chosen) {
      return false;
    } else {
    this.chosen = true;
      return true;
    }
  }
```

```
public List<Jolie_Var> getKey() {
  return this.key;
}

public void fixList(List<Jolie_Var> l){
  List<Jolie_Var> tmp = new ArrayList<Jolie_Var>();

  for(Jolie_Var v : l){
    for(int i = 0; i < this.key.size(); i++){
      Jolie_Var k = this.key.get(i);

      if(getVarName(v) == getVarName(k)){
        tmp.add(k);
        this.key.remove(i);
        break;
      }
    }
  }
  tmp.addAll(this.key);
  this.key.clear();
  this.key.addAll(tmp);
}

[###]

/**
 * JOLIE INTEGRATION SECTION
 */
public String getVarName(Jolie_Var v) {
  return v.getName();
}
```

```
}
```

La struttura di un nodo del grafo è delineata dalle variabili del correlation set che esso specifica, e queste sono memorizzate all'interno della variabile $List < Jolie\_Var >$ key. Per gestire la scelta di un nodo da parte dell'algoritmo RTG, è stato fornito il metodo *choose*(), in modo da poter specificare se il nodo in questione è stato scelto o meno.
Come noto, risulta indispensabile che una sequenza di variabili nella struttura di un radix tree sia un prefisso della sequenza successiva nella stessa struttura. A questo scopo, è stata implementata la funzione $fixList(List < Jolie\_Var > l)$, per gestire i casi in cui si necessiti un cambiamento nell'ordine delle variabili.
Si vuole porre l'accento sulle ultime righe di codice della classe precedente. Si noti, infatti, come in tale classe *è necessaria un'implementazione del meccanismo di integrazione con il linguaggio JOLIE*, per quanto concerne il reperimento del nome delle variabili.

### GraphBuilder

Di seguito, una porzione del codice (quella più rilevante) della classe *GraphBuilder*.

```
class GraphBuilder {
  List<List<GraphNode>> level; // each level of the graph
  List<GraphNode> roots;

  public GraphBuilder(List<Jolie_Var> list) {
    this.level = new ArrayList<List<GraphNode>>();
    for (int i = 0; i < list.size() + 1; i++) {
      this.level.add(new ArrayList<GraphNode>());
    }
    this.roots = new ArrayList<GraphNode>();
```

```
  GraphNode last = new GraphNode(list);
  build(last, list);
}

[###]

/**
 * Generates a list of root-Node which represents a hierarchy
 * for building a Radix tree.
 *
 * @return a list of root-node
 */
public List<GraphNode> radix_trees() {
  GraphNode found;

  for (List<GraphNode> l : this.level) {
    for (GraphNode n : l) {
      // -- for each node in graph
      found = null;

      for (GraphNode c : n.children) {
        if (c.choose()) {
          found = c;
          break;
        }
      }

      n.children.clear();
      if (found != null){
        found.fixList(n.getKey());
```

```
          n.addChild(found);
        }

        /*
         * if wasn't chosen in lower level than is a root
         */
        if (!n.wasChosen()) {
          roots.add(n);
        }
        // --
      }
    }
    return roots;
  }

  /**
   * Build a graph of all possible subset of given variable
   *
   * @param next The node with biggest number of variables
   * @param a list of variables
   */
   private void build(GraphNode next, List<Jolie_Var> l) {
    // adds a node into a level, based on number of variables
    this.level.get(l.size()).add(next);

    for (int i = 0; i < l.size(); i++) {
      // subset of l.size()-1 elements
      List<Jolie_Var> tmp = new ArrayList<Jolie_Var>();
      for (int j = 0; j < l.size(); j++) {
        if (j != i) {
          tmp.add(l.get(j));
```

```
        }
      }

      GraphNode n = new GraphNode(tmp);
      GraphNode node = findCloneInList(n);

      if (node == null) {
        n.addChild(next);
        build(n, tmp);
      } else {
        // add next node as son of found node
        node.addChild(next);
      }
    }
  }

  private GraphNode findCloneInList(GraphNode toFind) {
    int lvlnum = toFind.getKey().size();

    for (GraphNode n : this.level.get(lvlnum)) {
      if (n.key.equals(toFind.key)) {
        return n;
      }
    }
    return null;
  }
}
```

La classe ha lo scopo di ricreare la struttura a grafo dell'algoritmo RTG. Tramite la funzione *build*(), infatti, è creata una istanza della classe *GraphNode* per ciascun sottoinsieme di variabili del Cluster. Contemporaneamente, grazie alla ricorsione della funzione, ciascun nodo è collegato al successivo,

ottenendo così il grafo completo.
Una volta ottenuto il grafo, grazie alla funzione *radix_trees*(), è possibile ottenere la lista dei nodi-radice grazie ai quali, seguendo il percorso radice-foglia, si compongono le sequenze che rappresentano i vari radix tree.

### ClusterImpl

La classe principale che gestisce un cluster sfruttando le classi appena presentate è *ClusterImpl*. Di seguito, è presentato il codice di tale classe (la parte più significativa), seguito da un commento per ciascuna sezione.

```
public class ClusterImpl implements Cluster {
  /**
   * List<List<Jolie_Var>> represents the radix tree's structure
   * (i.e. RT{<empty>, <name>, <name,surname>})
   */
  private List< Pair< List<List<Jolie_Var>>,
    RadixTree<Jolie_Session> > > radixTrees;
  /**
   * The cluster's structure (i.e. C(name,surname))
   */
  private List<Jolie_Var> clusterVariables;
  /**
   * Cluster's manager
   */
  private ClustersManager manager;

  public ClusterImpl(List<Jolie_Var> list, ClustersManager cm) {
    this.clusterVariables = list;
    this.manager = cm;
    this.radixTrees = new ArrayList<Pair<List<List<Jolie_Var>>,
        RadixTree<Jolie_Session>>>();
    init();
```

```
}
[...]
```

Come si può vedere da questa prima sezione, i vari radix tree che compongono il cluster sono memorizzati all'interno della variabile $radixTrees$ in una struttura a lista. Tale lista è composta di classi di tipo Pair, i quali hanno la caratteristica di associare una chiave (in questo caso la struttura del radix tree) ed un valore (il radix tree). Per rappresentare correttamente la struttura di un radix tree, composta da sequenze di variabili, è stato necessario creare una lista di liste di variabili: $List < List < Jolie\_Var >>$.
La caratterizzazione del Cluster (le variabili che esso copre) è specificata nella variabile $clusterVariables$ sotto forma di lista di variabili.
Per motivi di integrazione con il linguaggio JOLIE, è memorizzata, in questo caso, la classe *ClusterManager*, in quanto questa fornisce funzioni pubbliche adatte allo scopo. Come si vedrà in seguito, infatti, è possibile accedere alle funzioni tale classe ogni volta che si necessita di informazioni riguardanti variabili e sessioni.
Il costruttore della classe necessita di parametri come l'oggetto di tipo ClusterManager, il quale genera il cluster in questione, e la lista delle variabili che formano la caratterizzazione del cluster stesso. Dopo aver inizializzato le variabili, il costruttore sfrutta la funzione *init*(), definita qui di seguito, per inizializzare i vari radix tree.

```
[...]
/**
 * initializes the rt variable
 */
private void init() {
  GraphBuilder gb = new GraphBuilder(this.clusterVariables);
  List<GraphNode> lgn = gb.radix_trees();

  /* build a radix tree for each set of variables */
  for (GraphNode n : lgn) {
```

```
      List<List<Jolie_Var>> mainSet =
        new ArrayList<List<Jolie_Var>>();
      buildList(mainSet, n);
      this.radixTrees.add(
        new Pair<List<List<Jolie_Var>>,
          RadixTree<Jolie_Session>>(
            mainSet,
            new RadixTreeImpl<Jolie_Session>()
          )
      );
    }
  }
  private void buildList(List<List<Jolie_Var>> l, GraphNode root) {
    l.add(root.key);
    if (!root.children.isEmpty()) {
      buildList(l, root.children.get(0));
    }
  }
  [...]
```

Una volta ottenute le liste di nodi-radice, grazie alla classe *GraphBuilder* la funzione utilizza queste informazioni per creare le strutture dei radix tree ($RT(SEQ_1, ..., SEQ_n)$) ed istanziare le classi di tipo $RadixTreeImpl < Jolie\_Session >$. In pratica, lo scopo è di inizializzare la variabile $radixTrees$. Come visto nell’interfaccia, la classe definisce le tre funzioni *add*, *delete* e *find* nel modo seguente:

```
  [...]
  public void addSession(Jolie_Session s) {
    List<Jolie_Var> lsv = manager.getSessionCsetVar(s);
    for (Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>>
        p : this.radixTrees)
    {
```

```
    p.value().insert( buildMaxString(p.key(), lsv) , s);
  }
}

public void delSession(Jolie_Session s) {
  List<Jolie_Var> lsv = manager.getSessionCsetVar(s);
  /* Removes session from all radix trees */
  for (Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>>
    p : this.radixTrees)
  {
    p.value().delete( buildMaxString(p.key(), lsv), s);
  }
}

public Jolie_Session findSession(Jolie_Message m) {
  List<Jolie_Var> listMsgVar = manager.getMessageCsetVar(m);
  if (manager.isMaxMatch(listMsgVar, this.clusterVariables)) {
    Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>> mcrt =
        maxCoveringRt(listMsgVar);
    return mcrt.value().find( buildMaxString(mcrt.key(),
        listMsgVar)
    );
  } else {
    /* find session in maximal matching radix tree */
    Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>> mcrt =
        maxCoveringRt(listMsgVar);
    String str = "";
    if(mcrt == null){
      mcrt = maxCoveringRt(this.getClusterVariables());
      str = "#";
    } else {
```

```
        str = buildMaxString(mcrt.key(), listMsgVar);
      }
      List<Jolie_Session> ls = mcrt.value().searchPrefix(str, 1);
      if(ls.size() > 0){
        return ls.get(0);
      } else {
        return null;
      }
    }
  }
  [###]
[...]
```

Le funzioni *addSession* e *delSession*, come specificato nell’algoritmo, sono molto simili: entrambe si avvalgono della funzione *buildMaxString* per creare la chiave basata sulle variabili ed inserire/rimuovere una sessione in/da ciascun radix tree. La funzione *buildMaxString* (del quale è di seguito fornito il codice) ha il compito di generare la stringa che andrà a comporre la chiave di ricerca della data sessione. La stringa sarà della forma $\#v_1\#...\#v_n\#$, dove $v_1, ..., v_n$ sono i valori delle corrispettive variabili del correlation set della sessione (Si veda 3.2.3 [*Generazione delle chiavi*]).
La funzione *findSession* ha lo scopo di restituire la sessione che meglio correla con il messaggio passato come parametro. A tal proposito, utilizzando la funzione *maxCoveringRt*, si sceglie in quale radix tree effettuare la ricerca, propagando la richiesta.
Di seguito sono presentati i codici delle due funzioni *maxCoveringRt* e *maxCoveringRt*.

```
  [...]
  /**
   * Builds a string like #john#smith# linking together the variables' value
   */
  private String maxCoveringRt(List<List<Jolie_Var>> rtv,
```

```
    List<Jolie_Var> session)
{
  String s = "#";
  List<Jolie_Var> chosen = null;
  /* find string variable structure for storing purpose */
  for (List<Jolie_Var> v : rtv) {
    if (chosen == null) {
      chosen = v;
    } else if (v.size() > chosen.size()) {
      chosen = v;
    }
  }

  /* to avoid different order of variables */
  for (Jolie_Var fv : chosen) {
    for (Jolie_Var sv : session) {
      if (manager.getVarName(fv) == manager.getVarName(sv)) {
        s = s + manager.getVarValue(sv) + "#";
      }
    }
  }
  return s;
}

/**
 * Finds the radix tree to search-in based on message's variable
 *
 * @param message's list of variable to match with a radix tree structure
 * @return The radix tree that has best match with given variable
 */
private Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>>
```

```
    maxCoveringRt(List<Jolie_Var> varList)
{
  Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>> found = null;
  int maxF = -1;

  for(Pair<List<List<Jolie_Var>>, RadixTree<Jolie_Session>>
      p : this.radixTrees)
  {
    for(List<Jolie_Var> l : p.key()){
      int nmv = manager.numMatchingVar(l, varList);
      if(nmv > maxF){
        maxF = nmv;
        found = p;
      } else if(nmv == maxF){
        if(l.size() == varList.size())
          found = p;
      }
    }
  }
  return found;
}
```

Grazie alle potenzialità di UML, è possibile fornire una visione di insieme della gerarchia e della composizione delle classi che implementano il concetto di Cluster.

## 4.3 Radix Tree

L’implementazione della sezione riguardante i radix tree è stata portata a termine modificando parti di un progetto precedente [**?**]. La modifica è apparsa necessaria in quanto occorreva un meccanismo che consentisse di memorizzare più sessioni aventi la medesima chiave di ricerca, meccanis-

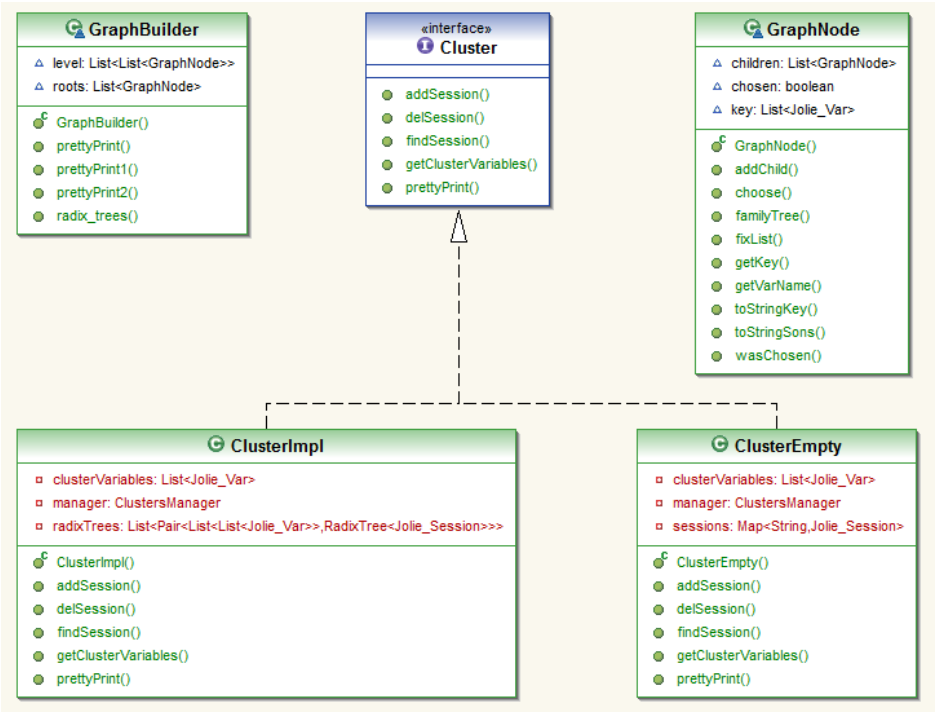


Figura 4.2: Diagramma UML della sezione Cluster

mo non presente in tale implementazione. Questa necessità è dovuta alle scelte progettuali effettuate. Si pensi, ad esempio, al cluster: *C(a,b)* esso contiene i radix tree *RT([empty], [a], [a,b])* e *RT([b])* e, essendo tutte le sessioni memorizzate all'interno di questo cluster, specifica entrambe le variabili a e b. Esiste quindi la possibilità di avere due sessioni con la medesima chiave all'interno del radix tree RT([b]). Ipotizzando di avere due sessioni $S(Dante, Alighieri)$ e $S'(Alfonso, Alighieri)$, infatti, è evidente come alla chiave Alighieri risultino associate entrambe le sessioni, benché distinte.
Per la memorizzazione delle sessioni nella nuova implementazione, è stata scelta una struttura gestita tramite hash table. Come chiave di ricerca in tale struttura è utilizzato l'identificativo della sessione da aggiungere: di conseguenza, all'interno della classe *RadixTreeNode* è implementata una funzione che reperisce tale identificativo. *A scopo di integrazione con JOLIE, quindi, è necessaria una modifica secondo le esigenze della funzione* $getSessionID$.

La rimozione di una sessione ha un comportamento analogo: si utilizza l’identificativo della sessione stessa come chiave di ricerca all’interno della hash table e si rimuove da quest’ultima la sessione trovata.
Essendo sessioni a pari livello di correlazione indifferenti in questo senso, dal momento in cui è effettuata una richiesta *findSession*, utilizzando come chiave di ricerca le variabili del correlation set del messaggio, il nodo foglia trovato restituisce la prima delle sessioni memorizzate.
L’immagine seguente raffigura il diagramma UML della parte riguardante le classi che implementano l’algoritmo che gestisce i radix tree.

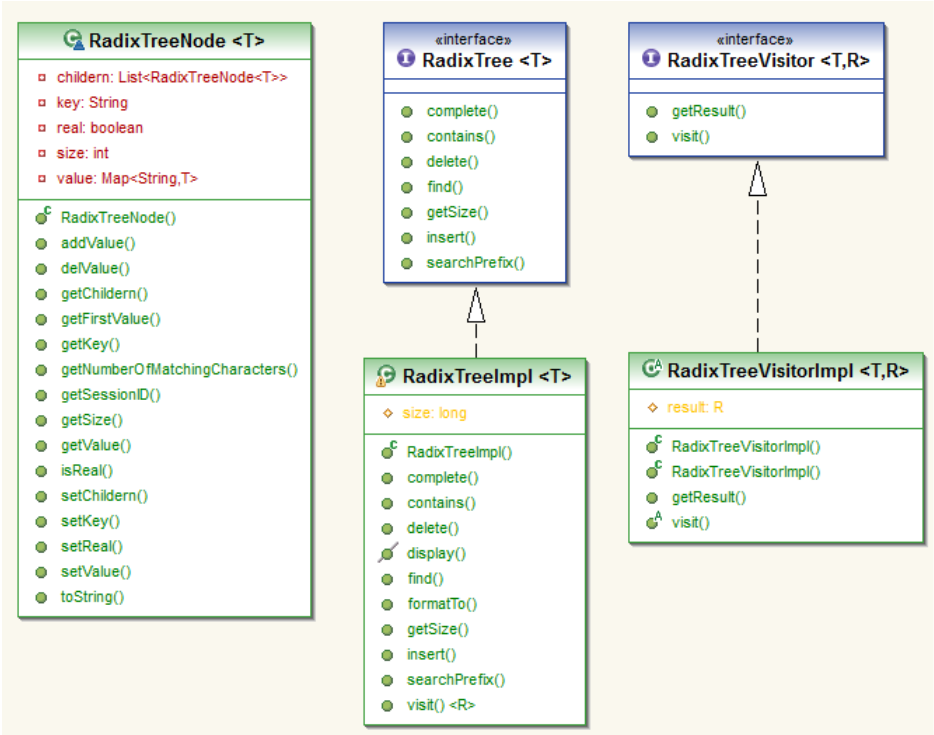


Figura 4.3: Diagramma UML della sezione Radix Tree

# Conclusioni

Ad oggi, sono presenti solo due implementazioni che risolvono il problema della ricerca di sessioni correlanti. La prima di queste si basa su un algoritmo di scansione su una lista, il quale algoritmo effettua la ricerca con complessitá lineare sul numero delle sessioni presenti. Data tale complessitá, questa soluzione non risulta proponibile per i servizi che possiedono un grande quantitativo di sessioni attive. La seconda implementazione, invece, è in grado di trovare una sessione in tempo costante tramite l'utilizzo di hash table, ammettendo peró una sola variabile di correlazione attiva. Come si è visto, la possibilitá di avere piú di una variabile nel correlation set consente di creare servizi che riescono a soddisfare pienamente sia richieste di sicurezza sia particolari richieste architetturali.
In questa tesi è stato definito ed implementato un algoritmo che consente di effettuare una correlazione di sessioni in tempo costante. Si definisce la complessitá in tempo costante poichè essa non dipende dal numero sessioni, ma dal numero di variabili del correlation set. La complessitá della ricerca di una sessione, infatti, è $O(nl)$, in cui $n$ rappresenta il numero di variabili di correlazione e $l$ la lunghezza massima della stringa da cercare (i.e. il valore massimo definibile in una variabile di correlazione). Realisticamente, si puó affermare che il numero di tali variabili non superi le tre unitá e che la lunghezza $l$ non tenda all'infinito. Si possono quindi considerare $n$ e $l$ come due costanti.

Per testare l'efficacia del nuovo algoritmo proposto è stata implementata una classe che simula le richieste di ricerca delle varie sessioni. Tale classe è implementata all'interno del file *Main.java*, e consiste in due test: il primo controlla le funzionalitá dell'algoritmo tramite richieste di aggiunta e rimozione sessioni e di ricerca di specifiche sessioni; il secondo crea di un numero considerevole di sessioni, definito dalla variabile $NUMSESSION$ all'interno del codice, e consente un numero finito di richieste, definito dalla variabile $NUMREQUESTS$. Quest'ultimo test calcola il tempo impiegato dai due algoritmi, cioè quello con costo computazionale lineare e quello presentato in questa tesi, esprimendolo in millisecondi. Nelle tabelle seguenti sono pubblicati tali risultati.

Come evidente dai risultati dei test presentati, l'algoritmo che si basa sull'utilizzo dei radix tree ha un tempo di ricerca costante, che in tale test dipende solo dal numero di richieste effettuate. L'algoritmo ribattezzato Linear, invece, ha un tempo di ricerca crescente al crescere delle sessioni presenti. Si puó dunque affermare di aver ottenuto un incremento prestazionale nella ricerca di sessioni correlanti. Il progetto presentato in queste pagine è stato implementato in modo da poter essere integrato in un secondo momento all'interno del linguaggio di programmazione Service-Oriented JOLIE. Data la sua indipendenza dalla struttura di JOLIE, in realtá, questo progetto potrá essere integrato con un qualunque linguaggio in Java, purché implementi correttamente tutte le funzioni che ne consentono l'integrazione.

| Sessioni | Radix Tree | Linear | Richieste |
|---|---|---|---|
| 1.000 | 1 | 27 | 50 |
| 5.000 | 2 | 130 | 50 |
| 10.000 | 1 | 257 | 50 |
| 25.000 | 2 | 711 | 50 |
| 50.000 | 2 | 1.398 | 50 |
| 100.000 | 1 | 2.690 | 50 |
| 150.000 | 2 | 8.663 | 50 |
| 1.000 | 3 | 53 | 100 |
| 5.000 | 3 | 267 | 100 |
| 10.000 | 3 | 545 | 100 |
| 25.000 | 3 | 1.376 | 100 |
| 50.000 | 3 | 2.765 | 100 |
| 100.000 | 4 | 5.404 | 100 |
| 150.000 | 3 | 9.607 | 100 |
| 1.000 | 4 | 107 | 200 |
| 5.000 | 6 | 525 | 200 |
| 10.000 | 6 | 1.067 | 200 |
| 25.000 | 6 | 2.755 | 200 |
| 50.000 | 6 | 5.635 | 200 |
| 100.000 | 6 | 10.930 | 200 |
| 150.000 | 6 | 16.676 | 200 |
| 1.000 | 10 | 263 | 500 |
| 5.000 | 11 | 1.319 | 500 |
| 10.000 | 12 | 2.646 | 500 |
| 25.000 | 16 | 6.839 | 500 |
| 50.000 | 13 | 13.617 | 500 |
| 100.000 | 14 | 26.687 | 500 |
| 150.000 | 16 | 40.064 | 500 |

Tabella 4.1: Tabella di confronto dei risultati ottenuti
(Tempi espressi in millisecondi)

# Bibliografia


[1] SOA : http://it.wikipedia.org/wiki/Service-oriented_architecture

[2] World Wide Web Consortium(W3C). Web Service Architecture: http://www.w3.org/TR/ws-arch/

[3] World Wide Web Consortium(W3C). WSDL : http://www.w3.org/TR/wsdl

[4] World Wide Web Consortium(W3C). SOAP : http://www.w3.org/TR/soap12-part0/

[5] World Wide Web Consortium(W3C). Extensible Markup Language (XML). http://www.w3.org/XML/.

[6] World Wide Web Consortium(W3C). HTTP - Hypertext Transfer Protocol. http://www.w3.org/Protocols/.

[7] World Wide Web Consortium(W3C). WS-CDL : http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

[8] OASIS. WS-BPEL : http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html

[9] OASIS. Correlation Sets : http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html#_Toc143402867

[10] Fabrizio Montesi. JOLIE : http://www.jolie-lang.org/index.php

[11] Donald R. Morrison. PATRICIA practical algorithm to retrieve information coded in alphanumeric. J. ACM, 15(4):514-534, 1968.

[12] Oracle. Java Programming Language : http://www.oracle.com/technetwork/java/index.html

[13] Grady Booch, James Rumbaugh, Ivar Jacobson. UML : Unified Modeling Language User Guide. Addison-Wesley 1999

[14] Tahseen Ur Rehman. Implementation of Radix Tree/Patricia trie/crit bit tree : http://code.google.com/p/radixtree/

[15] Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro e Fabrizio Montesi. An efficent management of correlation sets. Sottomesso per la pubblicazione, 2010.

# Ringraziamenti

Ringrazio tutti coloro che mi sono stati vicini e mi hanno aiutato nella realizzazione di questa tesi, dal relatore il Dr.Maurizio Gabbrielli a tutti i miei amici e parenti. Vorrei ringraziare in particolar modo la Dr.ssa Sara D'Agostino, la quale ha contribuito nel miglioramento del testo di questo elaborato.