Università degli Studi di Milano
Facoltà di Scienze e Tecnologie
Corso di Laurea in Sicurezza Informatica

SISTEMA DI MONITORAGGIO E VALUTAZIONE PER CYBER RANGES

Relatore: Prof.ssa Chiara Braghin

Tesi di:
Alessandro Della Torre
Matricola: 893181

Anno Accademico 2019-2020

Ringraziamenti

Un ringraziamento speciale a mia Mamma e mio Papà, per me grandissima fonte di ispirazione e orgoglio.
A mio fratello Michele per essermi stato d'esempio e per avermi dato la resilienza e la perseveranza nel raggiungere i miei obbiettivi.
A mia sorella Eleonora che mi rallegra ogni giorno grazie al suo sorriso.
Ad Anna Lucia Gorio per essermi stata vicina nei momenti più bui e per avermi sopportato ogni giorno senza mai smettere di credere nelle mie potenzialità.
Alle mie amiche per tutti i momenti indimenticabili che mi hanno reso la persona che sono.
A Michele Stafforini, Matteo Maiorana, Matteo Fedeli, Alessandro Stamera, Alessandro Grassi e Fabrizio La Rocca per aver reso questi anni un momento di serenità, di amicizia e di collaborazione.
Alla professoressa Chiara Braghin, per la sua disponibilità e professionalità.

Indice

Introduzione

L'obiettivo di questa tesi è stato quello di progettare e sviluppare un sistema di monitoraggio e valutazione per *Cyber Range*.

Con questo termine si intende una piattaforma di training innovativa, nel quale il trainee ha a disposizione diversi scenari virtuali che in genere cercano di ricreare componenti tipici di alcune ambientazioni reali, quali società industriali, di servizi, impianti di produzione di energia o installazioni militari, dove fare pratica sia nella messa in sicurezza del sistema, sia nella ricerca di particolari vulnerabilità. Diventa quindi importante essere in grado di analizzare in tempo reale le azioni svolte dal trainee, verificarle, valutarle ed eventualmente dare la possibilità all'insegnante di fornirgli aiuto in caso di necessità.
Le cyber range puntano a risolvere un problema che esiste da tempo all'interno della sicurezza informatica, ovvero mettere in pratica le proprie competenze in ambienti controllati di training, senza dovere fare pratica con i sistemi reali andando incontro a possibili problematiche sia tecniche che legali.
Negli ultimi anni questo strumento si sta rendendo necessario in quasi ogni campo della cybersecurity. Alcuni esempi possono essere:

- **Militare**: Vengono forniti strumenti di emulazione di attacchi informatici come tentativi di DOS alle infrastrutture sensibili.
  Il dipartimento che si occupa dello sviluppo di strumenti di difesa militari statunitensi DARPA(Defense Advanced Research Projects Agency) sta sviluppando la cyber-range nazionale NCR(National Cyber Range) proprio a questi scopi.

- **Aziendale**: Tantissime aziende non sanno come formare il proprio personale per rispondere a possibili attacchi; infatti viene organizzato spesso del training di aggiornamento, ma senza strumenti che permettano uno sviluppo attivo di competenze che potrebbero fornire un aiuto essenziale.

- **Università**: Gli insegnanti potrebbero tenere sotto controllo in tempo reale i propri alunni, vedendo gli argomenti che creano criticità e migliorando la metodologia dell'insegnamento, fornendo un metodo di aggiornamento costante

e al passo con i tempi. Un esempio è la Cyber-Range sviluppata dall'azienda Leonardo, messa anche a disposizione dell'Università di Genova.

Il punto di partenza: Threat Arrest

Inizialmente sono stato inserito in un progetto, chiamato Threat Arrest, finanziato dalla Comunità Europea e che comprende la creazione di una cyber range sviluppata dall'Università degli studi di Milano insieme ad altre importanti aziende del settore. Il progetto punta a creare una infrastruttura innovativa, con modelli e standard realizzativi elevati che permetterebbero di abbattere problematiche tecnico organizzative da sempre facente parti dei processi di sviluppo tradizionali, uno dei punti di forza di questa piattaforma riguarda l'utilizzo dei un approccio Model Driven. Con questo termine si definisce la volontà di creare una cyber ranges con facilità di deploy e riconfigurabilità insieme alla specifica di una pre-definita grammatica, eseguita e validata dai tool già modellati e con interfacce pronte all'uso, che permetterebbe di evitare errori derivanti da configurazioni manuali non sempre predicibili nelle infrastrutture tradizionali. Questo approccio permette inoltre un rapido sviluppo collaborativo tra i diversi parners del progetto garantendo maggiore qualità dei risultati finali. Partendo dalle specifiche del progetto, mi è stato chiesto di sviluppare una infrastruttura che permettesse di ricevere le informazioni prese dalle macchine virtuali, utilizzate dai trainee durante il training, elaborarle e verificare la loro validità utilizzando dei database a grafi, ritornando infine una valutazione complessiva dei singoli utenti. E' stato un compito molto arduo che ho deciso di risolvere utilizzando un approccio top-down andando a dividere le diverse componenti necessarie alla realizzazione dell'infrastruttura finale tenendo sempre comunque presente la necessità di interazione delle stesse.

Organizzazione della tesi

Il tempo impiegato nello sviluppo dell'applicazione può ritenersi lo stesso che è stato speso nella ricerca di una possibile architettura e degli strumenti che potevano essere utilizzati: la cronologia di eventi che mi ha portato dalla fase iniziale alla stesura della tesi è riassunta nel capitolo 2.
Una parte fondamentale è stata l'implementazione del database a grafi, un modello di raccolta di informazioni totalmente nuovo da quelli che avevo precedentemente studiato, che mi ha portato ad approfondire l'utilizzo dello strumento Neo4j e del suo linguaggio specifico, Cypher, di cui parlerò nel capitolo 3.
Nell'ultimo capitolo, il quarto, vengono spiegate tutte le funzionalità che l'applicazione possiede andando sempre più in dettaglio fino ad arrivare ad una analisi del codice implementato al suo interno. Viene anche posta una implementazione degli

agent all'interno delle macchine che mi ha permesso di testare le reali capacità offerte da questa soluzione.

Capitolo 1

Analisi dello stato dell'arte

In breve, il sistema di valutazione da implementare doveva essere in grado di operare in su due fronti: da un lato, doveva essere in grado di raccogliere tutte le azioni eseguite dal trainee durante l'esecuzione della sessione di trainig sulla cyber range; dall'altro lato, doveva essere in grado di costruire quello che nella tesi chiameremo *actual path*, ovvero la sequenza ordinata delle azioni eseguite, e confrontarlo con l'*expected path*, ovvero la sequenza di passi considerata corretta, già noto al sistema fin dall'inizio.

Risulta evidente come il sistema dovesse risolvere diverse problematiche: essere in grado di raccogliere tutte le informazioni di interesse, poterle spedire all'esterno della macchina virtuale su cui avveniva il training, memorizzarle sotto forma di *expected* e *actual trace* e alla fine confrontarle. Questo ha richiesto che l'architettura fosse in grado di gestire le varie fasi e ha comportato un'attenta analisi dello stato dell'arte per capire se si potessero adattare sistemi già esistenti.

In particolare, nodi problematici che hanno guidato il lavoro sono i seguenti:

- Ci sono standard di riferimento?
- E' possibile un monitoraggio delle syscall?
- Differenze tra l'utilizzo di nosql e i db a grafi?
- Interoperabilità con la suite Elastic?
- Scenari di training con l'utilizzo di tale suite?
- Possibilità di analisi e gestione del rischio?
- Possibilità sull'utilizzo di tecniche di Machine Learning?
- Possibile utilizzo/creazione di un ambiente realtime?

Nella prima fase ho svolto un lavoro di ricerca e verifica sugli strumenti idonei a performare le seguenti operazioni:

- Raccolta di informazioni sulle VM
- Trasferimento dei log(delle singole VM) ad un Message Collector
- Analisi dei log
- Impiego di DBMS a grafi che permettessero il salvataggio permanente delle informazioni.
- Distribuzione delle VM ai trainee

Cercando in internet, ho trovato una bozza del NIST, creata dal National Initiative for Cybersecurity Education (NICE) e dal Cyber Range Project Team, che offriva delle linee guida basilari per la creazione di una Cyber Range: queste mi hanno agevolato nella scelta degli strumenti adatti a questo scopo.

1.1 Raccolta Trasferimento e Analisi log

Una delle domande guida sopraelencate mi ha portato ad approfondire la suite ELK (Elasticsearch, Logstash e Kibana), che fornisce un ambiente completo di raccolta, analisi e visualizzazione dei log: questa soluzione è molto apprezzata per le sue doti di efficienza, malleabilità e multifunzionalità.
La suite adotta diversi strumenti, quali: Filebeat per la raccolta di informazioni sui singoli host/vm, Logstash come log-collector, Elasticsearch come query-engine e Kibana, che fornisce una interfaccia con Elasticsearch.
Dopo un confronto con altri programmi di raccolta, ho ritenuto più opportuna una eventuale implementazione di EFK(Elasticsearch, Fluentd e Kibana), delegando a FluentBit la raccolta di informazioni e utilizzando Fluend unicamente come log-collector. Ho confrontato le caratteristiche degli strumenti interni di ogni singolo layer, andando ad identificare quali fossero le peculiarità di ognuno.
Inizialmente ho valutato gli strumenti di raccolta delle informazioni sugli host tramite l'utilizzo di Filebeat, Fluentbit, Osquery. Osquery è uno strumento che permette di utilizzare il singolo host come se fosse un database, dando la possibilità di performare query al suo interno per raccogliere informazioni. Funziona su tutte le maggiori distribuzioni, ma presenta alcuni problemi di interoperabilità con gli altri strumenti come, ad esempio, Elasticsearch. Un' altra grande criticità di questa soluzione è il suo non poter essere utilizzata come sistema asincrono di eventi definiti da pattern specifici. Filebeat fa parte della suite Elastic ed è perfettamente integrato con Logstash, ma

non fornisce la stessa interoperabilità con altri sistemi di log-collector.

Fluentbit ha come punti di forza le seguenti caratteristiche:

- Utilizza meno risorse in confronto a quelle necessarie per l'esecuzione di Filebeat
- Offre una maggiore interoperabilità, permettendo diversi formati di output e la possibilità di integrazione con strumenti quali Apache kafka, Fluentd etc..
- Può essere utilizzato per raccogliere metriche sull'utilizzo
- Possibiltà di integrazione con sistemi più complessi come, ad esempio, Kubernotes

In seguito ho valutato le differenze tra Fluentd e Logstash come log-collector:

- **Piattaforma**
  Entrambi operano sia su Windows sia su Linux
- **Configurazione**
  Logstash utilizza una tecnica di configurazione di tipo 'if-then-else'
  Fluentd: Utilizza un linguaggio a TAG; possiede molti plugin per le risorse di input e offre funzionalità aggiuntive(in confronto a Filebeat) insieme ad una maggiore interoperabilità e facilità di utilizzo.
- **Performance**
  Logstash consuma più memoria di Fluentd, la quale tuttavia riesce a scalare in modo migliore le performance.
  Entrambi hanno la loro versione leggera, rispettivamente: Fluent-bit, Elastic beats.
- **Supporto**
  Nonostante Logstash abbia una vasta applicazione, Fluentd risulta leader in quanto costantemente supportato e aggiornato con la collaborazione di aziende del calibro di Amazon, Microsoft, Nintendo etc..

Sulla base di queste analisi, ho preferito scegliere una implementazione che utilizzasse Fluentbit come agente sulle vm per la raccolta di informazioni, insieme, eventualmente, a Osquery e Fluentd come log-collector. L'utilizzo di Osquery è comunque consentito, poichè Fluentd offre un plugin che ne permette l'interoperabilità.

Ho verificato la possibilità di impiegare database a grafi insieme ad altri strumenti di elaborazione di log quali Splunk e Apache Sorl. L'esito è risultato negativo per i seguenti motivi:

- Dato il costraint di utilizzare strumenti open, ho dovuto scartare la possibilità di impiegare Splunk
- Sorl possiede una ampia documentazione, tuttavia dimostra limitate capacità di elaborazione e un esiguo numero di strumenti a disposizione.

Una volta scelto Elasticsearch come log-engine, avevo trovato una soluzione che mi offrisse realtime, affidabilità ed efficienza e rispondesse ai requisiti posti inizialmente.

1.2 Utilizzo dei Grafi

Si rendeva necessario ricorrere ad una implementazione che utilizzasse i grafi per i seguenti motivi:

- Eterogeneità dei log presi dai diversi tool
- Eterogeneità delle macchine e del sistema
- Migliore sistema di interrogazione del database in confronto ai modelli relazionali
- Utilizzo ottimizzato di modellli di Machine Learning.

Lo strumento di indicizzazione offerto da Elasticsearch (per l'utilizzo dei grafi) non ha potuto essere applicato, poichè la licenza non ne permetteva il riutilizzo. Essendo in cerca di altre soluzioni, ho deciso di ricorerre a Neo4j come DBMS a grafi in virtù della sua efficienza, della grande quantità di documentazione disponibile e del suo largo supporto. Tra i possibili metodi per connettere Elasticsearch con Neo4j, ho considerato le soluzioni offerte da GraphAware e le API rest di Elasticsearch. Ho ritenuto opportuno riassumere le funzionalità che questa soluzione avrebbe potuto offrire, insieme ad un esempio di come sono state effettuate le scelte implementative durante il percorso di ricerca.

1.2.1 GraphAware

Descrizione

Insieme di tool/plugin/framework che dovrebbero fornire un'integrazione delle funzionalità tra Neo4j e Elasticsearch.
Nello specifico, la società mette a disposizione 2 moduli:

1. GraphAware Neo4j Elasticsearch Integration
2. GraphAware Graph-Aided Search

Considerazioni

Questi moduli sono forniti con licenza GPL e sono rilasciati da GrahpAware, un'azienda che fornisce soluzioni di analisi dei dati su db a grafi come Neo4j. Ha rilasciato questi strumenti utilizzabili solo con le versioni free sia di Neo4j che di Elasticsearch; per l'utilizzo della versione Premium, è necessario l'acquisto dei moduli/strumenti aggiuntivi. Tuttavia, se si utilizzano le versioni Premium, bisogna valutare l'opportunità di impiegare questi strumenti anziché quelli nativi offerti dalle piattaforme stesse.

GraphAware Neo4j Elasticsearch Integration

Essa fornisce un'integrazione bidirezionale tra Noe4j e Elasticsearch. Comprende diverse componenti tra cui:

- GraphAware Neo4j Framework
- GraphAware Neo4j UUID
- GraphAware Neo4j Elasticsearch Integration

Il primo dei due moduli fornisce comunicazione tra Neo4j -> Elasticsearch.
Offre i seguenti moduli:

- **Embedded Mode / Java Development**:
  Se si sviluppano 'applicazioni' su Noe4j, questo modulo può essere inserito come dipendenza nel progetto Java. Pensavo potesse rivelarsi funzionale alla realizzazione di un middleware con chain utili a controllare il percorso di apprendimento delle singole macchine virtuali.
- Possibilità di fornire autenticazione nel caso si utilizzasse Shild plugin di Elasticsearch, ovvero un sistema di protezione dinamico fornito all'interno della Suite di Elastic.
- **Cypher Procedures — Searching for nodes or relationships**
  Queste procedure permettono di performare ricerche tra i nodi indicizzati permettendo un'ulteriore ricerca utilizzando il risultato appena ottenuto.
  Ogni query di ricerca viene inviata attraverso una procedura che che viene performata con gli indici preconfigurati per replicarli in Elasticsearch.

GraphAware Neo4j Framework

Framework che permette lo sviluppo di piattaforme che utilizzano i dati di neo4j.

- **GraphAware Server**: Neo4j server extension che permette agli sviluppatori di creare rapidamente API on top Neo4j usando spring o jax-rs.
- **GraphAware Runtime**: Ambiente runtime utilizzabile sia per embedded sia server deployment che permette di utilizzare moduli precompilati chiamati 'GraphAware Runtime Modules' che estendono le funzionalità del database fornendo:
  - Transparently enriching/modifying/preventing ongoing transactions in real-time
  - Performing continuous computations on the graph in the background
- Viene fornito anche un sistema di API in modo da creare sistemi embedded in Java.

GraphAware Graph-Aided Search

Questo modulo è un plugin di Elasticsearch che permette agli utenti di boostare/-filtrare i propri risultati utilizzando i dati salvati nel database Neo4j. Una volta performata la ricerca nel database di Elasticsearch, prima di ritornare il risultato all'utente vengono chieste informazioni aggiuntive a Neo4j tramite REST API.

Motivazioni sull'utilizzo di GraphAware

Le soluzioni offerte da GraphAware non sono più supportate dall'azienda che le metteva a disposizione; L'implementazione tramite API necessitava dello sviluppo di un applicativo apposito e si avevano difficoltà nell'elaborare il traffico in tempo reale.

1.3 Distribuzione ambienti virtuali

Come strumenti di distribuzione di ambienti di lavoro mi sono focalizzato su Kubernotes in quanto permetteva nativamente l'utilizzo della suite EFK. Kubernotes viene utilizzato come sistema di gestione di container in modo distribuito e permette, inoltre, l'utilizzo di Docker per il deploy di applicazioni all'interno dei singoli container.
Questa soluzione avrebbe inoltre permesso di centralizzare i log delle singole applicazioni tramite un apposito POD in Kubernotes dove ,in seguito, sarebbero stati presi da Fluentd e inviati ad Elasticsearch.
Uno dei problemi risiedeva nell'utilizzo di vere e proprie VM che potevano offrire ambienti di training utili ai fini della piattaforma ma questi due strumenti forniscono

solo il dispiegamento di container e applicazioni; ho quindi iniziato ad informarmi sulla possibilità di utilizzare Kubevirt: offre l'unione delle due soluzioni in un ambiente efficiente e scalabile.
In seguito mi sono state date indicazioni che la funzionalità di dispiegamento delle VM agli utenti era già stata implementata e che la suite scelta per assolvere questo compito era Openstack. Ho spostato la ricerca sugli strumenti utilizzabili su questa piattaforma, con particolare attenzione alle possibili implementazioni della suite EFK.
Openstack in passato utilizzava EFK ma, negli ultimi tempi, ha adottato una soluzione che implementa l'utilizzo di Rsync. Ho deciso di continuare ad adottare Fluentd come log collector poichè è comunque possibile ricevere i log provenienti da Rsyslog per poi inviarli ad Elasticsearch.

1.4 Cambio di direzione

Per gli obbiettivi che dovevo raggiungere mi serviva uno strumento all'interno di Elasticsearch che utilizzasse il paradigma Evento-Azione in modo che, una volta caricato il ReferenceGraph (template di azioni che l'utente dovrebbe performare), potesse eseguire delle azioni in base ai log in ingresso.
Questo strumento viene chiamato Watcher: permette di specificare un insieme di condizioni per cui l'evento deve verificarsi, come l'arrivo di un log o il risultato di una query arbitraria all'interno dei log, permette inoltre di specificare condizioni nidificate o una chain di condizioni complesse.
Una volta performate le condizioni, rende possibile l'esecuzione di una azione definita precedentemente come: l'invio di una email all'amministratore, performare ulteriori query salvando i risoltati etc..
Sembrava lo strumento che stavo cercando, quando sono entrato in contrasto con la licenza che non ne permetteva l'utilizzo: infatti serviva la licenza Premium e andava contro le precondizioni che mi erano state poste.

A questo punto, confrontandomi con la professoressa, siamo arrivati alla conclusione che Elasticsearch non fosse più la soluzione che faceva al caso nostro e ho iniziato ad informarmi sui possibili metodi per creare un applicativo che gestisse in maniera indipendente i log e li inserisse all'interno del database.
Dopo una approfondita ricerca decisi che sarebbe stata un'ottima soluzione adottare Apache Kafka, uno strumento di streaming delle informazioni che permetteva di risolvere parte dei problemi che avevamo in quel momento quali:

- Utilizzo di plugin per la creazione di streaming di dati all'interno di Neo4j

- Utilizzo di una struttura molto efficiente e che permetteva la suddivisione delle informazioni tramite delle code gestibili in modo indipendente
- Largo supporto di apache Kafka per il largo utilizzo a livello aziendale
- Largo supporto per il plugin gestito, aggiornato e sviluppato direttamente dal team di Neo4j

Dopo avere svolto una riunione con un docente responsabile della infrastruttura si decise di adottare RabbitMQ, e non Kafka, come strumento di Log-collector: questa scelta è stata giustificata dalla volontà di mantenere l'infrastruttura semplice e con un carico di lavoro minimo.

1.5 Architettura Finale

Alla fine del percorso di ricerca degli strumenti, mi è stato quindi chiesto di elaborare una architettura con le seguenti componenti:

- **Log Forwarder** in Java presente nelle VM per l'invio dei messaggi a RabbitMQ
- **RabbitMQ-Server** gestito tramite Docker utilizzato come Log-Collector e gestore delle code.
- **Java-Application** che funge da middleware tra RabbitMQ e Neo4j .

Capitolo 2

Database per gli eventi

2.1 Breve introduzione a Neo4j

In informatica, le *'basi di dati a Grafi'* o *'database a grafi'* si riferiscono ad un modello che utilizza dei nodi e degli archi per archiviare l'informazione. Negli ultimi anni si sono evolute in modo esponenziale grazie alle caratteristiche che le rendono valide alternative agli altri database. Alcune di queste sono:

- Maggiore velocità nell'associazione di informazioni
- Maggiore scalabilità per basi di dati con grandezze elevate
- Carenza di regole stringenti a cui sono sottoposti i database relazionali,che li rende preferibili quando bisogna lavorare con dati eterogenei
- disponibilità di potenti algoritmi,utili a sintetizzare situazioni reali difficilmente gestibili tramite i database tradizionali

Per sviluppare il database della nostra applicazione abbiamo preferito utilizzare Neo4j, uno tra i più famosi di questa tipologia. Alcune peculiarità lo rendono migliore dei suoi concorrenti:

- Numerose applicazioni d'esempio,sviluppate con i più comuni linguaggi di programmazione, interagiscono con esso
- Offre tantissime funzionalità liberamente accessibili che lo rendono idoneo per quasi tutti i progetti; le opzioni a pagamento riguardano funzionalità di sicurezza e scalabilità, insieme a possibilità di supporto nello sviluppo e mantenimento della base di dati

- Offre diverse versioni utilizzabili per quasi tutti i sistemi operativi, sia lato Client che lato Server. Queste offrono due tipi di interfacce: una web che ne permette una facile gestione all'interno della rete, una nativa del sistema operativo, resa disponibile dalla versione desktop
- La creazione e l'utilizzo di Cypher: un linguaggio di Data Manipulation Language SQL-Based che utilizza l'Ascii-art per performare operazioni all'interno della base di dati.

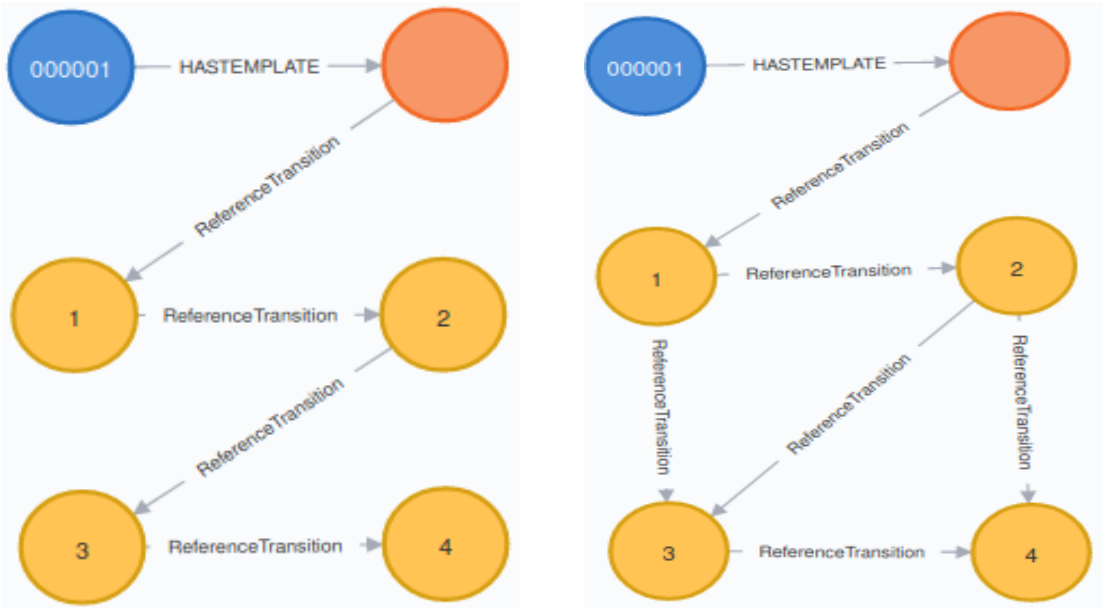
Cypher

Farò una breve introduzione alla Ascii-Art utilizzata da Cypher: essa tornerà utile in seguito per capire e descrivere la struttura dei grafi all'interno del paper. Un nodo viene definito da due parentesi tonde con all'interno i parametri *(label:NodeName)* dove: il *label* può essere un nome a piacere utilizzato per identificare il nodo all'interno delle query, *NodeName* identifica la tipologia di nodo(possiamo immaginarcelo come il nome della tabella a cui si riferisce).
Per ogni nodo possiamo definire delle proprietà aggiuntive al suo interno, identificate da parentesi graffe. Quindi, un esempio di nodo è il seguente:
(label:NodeNameproprietà: 'EsempioDiProprietà')
Le relazioni sono strutturate con parentesi quadre e offrono la stessa struttura dei nodi, con la conseguente possibilità di aggiunta di Label e proprietà. Per cristallizzare le spiegazioni sopracitate, riportiamo un esempio reale con 2 nodi e una relazione che li collega:
(m:Macchinaid:1, marca:'Fiat')-[p:Possiede]->(m:Motoreid:1, marca: 'Volkswagen')
Cypher fornisce anche un modo per modificare la struttura del database e performare query al suo interno.

2.2 Struttura Database

Prima di procedere, preferisco fornire alcune definizioni utili a capire il background implementativo utilizzato di seguito nella tesi. Due di queste riguardano la differenza tra TraineeGraph e ReferenceGraph.
Con ReferenceGraph mi riferisco al template o cammino minimo di azioni necessarie al raggiungimento degli obiettivi di una singola macchina virtuale o di uno specifico scenario. Con TraineeGraph si intende il cammino che rappresenta il training del singolo trainee, collegato da una parte con l'utente, dall'altra con la macchina virtuale/scenario(a livello implementativo tra questi due elementi non vi sono differenze).
La struttura dei due grafi all'interno del database è la seguente:

ReferenceGraph



(a) 1.ReferenceGraph (b) 2.ReferenceGraph

Figura 1: An example graph

Il ReferenceGraph è formato da 3 componenti: ReferenceGraph, ReferenceNode, ReferenceTransition. Il ReferenceGraph è il nodo iniziale del del grafo collegato alla VM tramite la relazione 'HASTEMPLATE'; questa decisione viene estesa a tutte le relazioni tra una qualsiasi macchina virtuale(o scenario) e i rispettivi nodi ReferenceGraph collegati.
All'interno del ReferenceGraph possono essere definite informazioni specifiche di quel cammino come: tempo medio di svolgimento, data creazione e data di ultima modifica, identificativo dell'utente che lo ha inserito, numero di nodi del cammino, difficoltà associata a questa sessione di training. I ReferenceNode rappresentano gli stati del training, collegati tra loro da relazioni di tipo ReferenceTransition che definiscono il cambiamento da uno stato all'altro. Sono identificati da un nome, che rappresenta il numero dello stato, e da una proprietà 'status' che indica quando il training è arrivato alla conclusione.
L'assunzione che le ReferenceTransition comportino un cambiamento di stato è un passo chiave: se così non fosse, bisognerebbe cambiare radicalmente l'architettura dell'applicazione, per fare in modo di avere una verifica delle azioni eseguite dall'utente, in quanto non è più detto che esse comportino, per l'appunto, un cambiamento di stato. All'interno della figura 1.ReferenceGraph troviamo degli esempi di azioni che

ci si aspetta che il trainee compia; in questi casi specifici si è ipotizzato l'utilizzo della VirtualMachine(o Scenario) *SeedUbuntu*.

Il ReferenceGraph raffigurato in figura n1 può essere realizzato tramite questa query in Cypher:

```
match(SeedUbuntu:ScenarioVm{name:"000001"})
create (e:ReferenceGraph),
(en_1:ReferenceNode{name: '1', status: 'Iniziale'}),
(en_2:ReferenceNode {name: '2', status: 'Intermedio'}),
(en_3:ReferenceNode {name: '3', status: 'Intermedio'}),
(en_4:ReferenceNode {name: '4', status: 'Stop'})
create (e)<-[:HASTEMPLATE]-(SeedUbuntu)
create (e)-[:ReferenceTransition]->(en_1)
create (en_1)-[:ReferenceTransition{Action: 'ls'}]->(en_2)
create (en_2)-[:ReferenceTransition{Action: 'pwd'}]->(en_3)
create (en_3)-[:ReferenceTransition{Action: 'cmd'}]->(en_4)
```

Procedendo nell'analisi dei ReferenceGraph, mi sono accorto che sarebbe stato utile potere implementare scenari più complessi, che comprendessero dei salti di nodi: infatti non è detto che ci sia sempre una linearità nei percorsi e neppure che ci sia un solo modo per arrivare alla soluzione finale. Questa casistica è riassunta in figura n2 e può essere creata attraverso la seguente query:

```
match(SeedUbuntu:ScenarioVm{name:"000001"})
create (e:ReferenceGraph),
(en_1:ReferenceNode{name: '1', status: 'Iniziale'}),
(en_2:ReferenceNode {name: '2', status: 'Intermedio'}),
(en_3:ReferenceNode {name: '3', status: 'Intermedio'}),
(en_4:ReferenceNode {name: '4', status: 'Stop'})
create (e)<-[:HASTEMPLATE]-(SeedUbuntu)
create (e)-[:ReferenceTransition]->(en_1)
create (en_1)-[:ReferenceTransition{Action: 'sudo'}]->(en_3)
create (en_1)-[:ReferenceTransition{Action: 'ls'}]->(en_2)
create (en_2)-[:ReferenceTransition{Action: 'fdisk'}]->(en_4)
create (en_2)-[:ReferenceTransition{Action: 'pwd'}]->(en_3)
create (en_3)-[:ReferenceTransition{Action: 'cmd'}]->(en_4)
```

Utilizzando Neo4j si ha la possibilità di poter usufruire di archi orientati: grazie all'utilizzo di questa implementazione e al flusso non ricorsivo(o retroattivo) dei cammini, possiamo dedurre che l'utente arriverà per forza alla conclusione del training e non si troverà mai in una situazione di loop all'interno del ReferenceGraph. Non è stato dato deciso a priori alcun numero minimo o massimo di relazioni(o azioni) che il ReferenceGraph deve avere, infatti questa decisione dipende molto dallo scenario

che si vuole modellare.
Prendiamo come esempio un attacco di SQLi all'interno di un WebService che fornisce una schermata di autenticazione: l'azione che viene ritornata all'applicazione tramite i log della VM è la stringa di caratteri inserita dall'utente nel campo *username*. In questo caso specifico, avremo presumibilmente tanti archi direzionati dal primo nodo(o stato) del cammino verso il secondo e ultimo nodo, che rappresenta lo stato di riuscita di autenticazione tramite questa tipologia di attacco: infatti, se si analizzano esclusivamente le tipologie di SQLi mediante tautologie, ci potrebbero essere n possibilità da tenere in considerazione. Questo esempio potrebbe portare alla semplice conclusione che sarebbe necessario modellare il problema sotto un'altra prospettiva, magari prendendo in considerazione il solo risultato dell'azione, ma spero renda chiaro il concetto che porre un numero minimo o massimo di transizioni a disposizione all'interno del ReferenceGraph potrebbe risultare un vincolo inutile e non necessario. Potrebbero verificarsi anche casi in cui non ci sia un solo stato conclusivo, ma più stati conclusivi che modellano la fine del training che un utente potrebbe raggiungere tramite cammini diversi. Una possibilità, non implementata all'interno dell'applicativo, comprende la possibilità di goal intermedi o cammini secondari(diversi dal flusso di training principale) fini a se stessi. Questa tipologia di implementazione non va confusa con quella descritta precedentemente: nel primo caso, infatti, ci possono essere più stati finali del training principale, mentre nel secondo caso il raggiungimento di questi obbiettivi secondari sarebbe un punteggio aggiuntivo, ma non rappresenterebbe la fine del training principale, bensì solo del sottografo relativo a quel sotto-goal.

TraineeGraph

Il TraineeGraph è simile al ReferenceGraph, ma con delle informazioni aggiuntive al suo interno riguardanti il training che il trainee sta svolgendo. Un TraineeGraph è relativo ad un singolo ReferenceGraph, infatti il singolo training che l'utente sta svolgendo sarà relativo ad un solo modello risolutivo e non vengono presi in considerazione possibili join tra i diversi modelli. I nomi degli attributi e degli archi dati a questa tipologia di sottografo sono molto simili a quelli utilizzati all'interno del ReferenceGraph, infatti si chiamano rispettivamente TraineeGraph, ActualNode e ActualTransition. Parlando di TraineeGraph, non è collegato soltanto alla VM a cui fa riferimento, ma anche al nodo *Trainee* rappresentante l'utente che sta svolgendo(o ha già svolto) il training. Questa relazione tra il nodo Trainee e quello TraineeGraph può essere di due tipologie in base allo stato del training: nel caso in cui l'utente abbia concluso il training, la relazione sarà di tipo *COMPLETATO*, nell'altro caso il training viene considerato attivo, quindi la tipologia di relazione sarà *SVOLGIMENTO*. All'interno del nodo TraineeGraph possono esserci altre informazioni quali: il tempo

impiegato dal trainee fino a quel momento, il punteggio parziale o complessivo, il numero di relazioni o il numero di nodi, informazioni relative ad una possibile 'ripresa a freddo' del training. Per quanto riguarda le ActualTransition, queste contengono le seguenti informazioni: azione che determina il cambiamento di stato con relativo timestamp(ovvero quando avviene il cambiamento di stato).

Capitolo 3

Progettazione e sviluppo dell'applicativo

3.1 Descrizione Applicazione

L'applicazione è fondamentalmente un applicativo che, in base all'input, prende in considerazione delle casistiche d'esecuzione differenti, per poi seguire dei passi predefiniti(basati sulle casistiche) e produrre infine un risultato.

Ci sono due tipologie di input all'interno della applicazione, quali:

1. **Utilizzo delle API Rest**:
   L'applicativo fornisce 2 tipologie di API differenti, che permettono la gestione del ReferenceGraph e del TraineeGraph. Per gestione intendo la possibilità di Richiedere, Inserire, Eliminare questi sottografi. Questa funzionalità è molto importante, perchè se non ci fosse si renderebbe necessaria la presenza di un addetto esperto sulla struttura del Database, del suo linguaggio dedicato(CYPHER), che gestisca manualmente ogni possibile situazione delle azioni sopracitate.

2. **Utilizzo della coda di RabbitMQ**:
   In questo caso, l'applicazione avvia un listener sulla coda di RabbitMQ contenente i diversi log delle VM(o scenari) utilizzate dai trainee durante il training. I messaggi utilizzano come formato JSON. Questa scelta è stata compiuta per garantire velocità di trasferimento delle informazioni e una maggiore interoperabilità tra i diversi componenti che devono dialogare tra di loro. Una volta che i messaggi arrivano alle code, sono presi in ordine di arrivo tramite una politica FIFO; la coda di input, come le altre code specificate in seguito, viene definita o creata appena l'applicativo viene lanciato. Quando un messaggio arriva dentro l'applicazione viene fatto un parsing del suo contenuto estrapolando i singoli

parametri risiedenti al suo interno (definito in JSON) e inviati all'algoritmo di matching che penserà alla loro gestione.

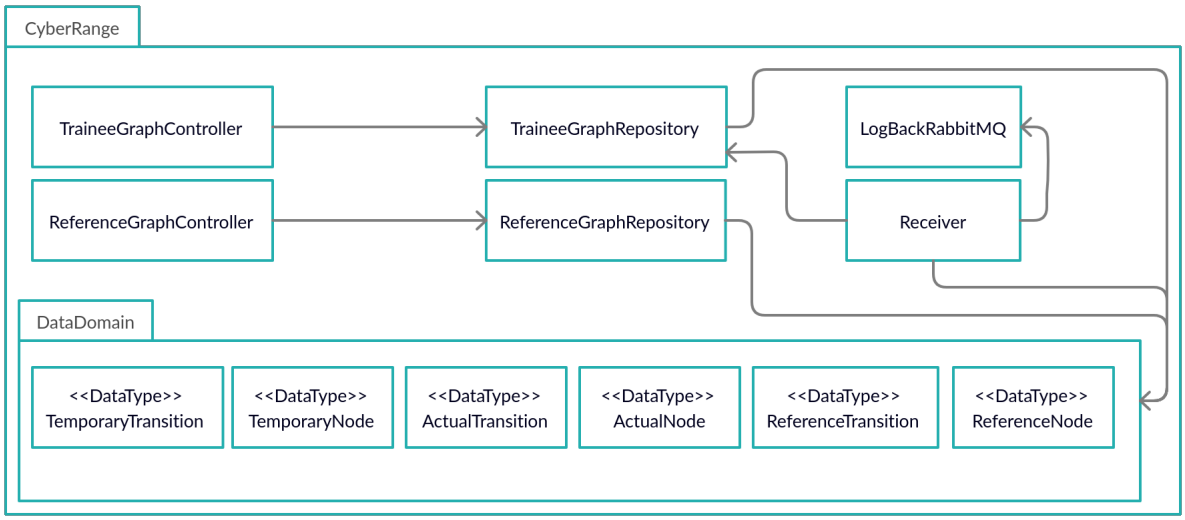
Le diverse casistiche che devono essere prese in considerazione sono:

- Analisi dello scenario di training, in base a quello può essere specificata l'appartenenza dell'utente ad un team (BlueTeam, RedTeam etc..) e l'utilizzo di scenari più complessi con più team che lavorano simultaneamente(cooperando o in concorrenza tra di loro)
- Analisi sull'odine dei log: Sono stati implementati due metodi che si riferiscono a due casistiche differenti.
    1. Nella prima viene coperto il caso in cui l'ordine dei log in arrivo non deve combaciare in modo diretto con l'ordine dei nodi nel ReferenceGraph.
    2. Nella seconda, se l'ordine delle operazioni non combacia perfettamente viene avviata una procedura di ripristino del cammino.

I relativi output possono essere:

- Invio dei singoli percorsi alle relative code, che possono essere: TemporaryTrace o CompleteTrace (nel caso di utilizzo delle code di RabbitMQ)
- Per quanto riguardale Api REST: vengono ritornati valori booleani per quanto riguarda le operazioni di inserimento o eliminazione, nel caso si richieda un ReferenceGraph o un TraineeGraph viene semplicemente ritornato il grafo specificato nella richiesta.

3.2 Dettagli Implementativi



3.2.1 Neo4j DAO-Repository Implementation

Inizialmente vi era una sola classe Repository che implementava tutti i metodi per interagire con il database, in seguito si è reso necessario inserire i diversi controller relativi ai diversi sottografi e si è preferito differenziare anche i relativi repository in modo che, in caso di bisogno, si sarebbe potuto delegare a thread separati la gestione delle singole richieste ottimizzando la complessità computazionale e i tempi di risposta. Le classi *ReferenceGraphRepository* e *TraineeGraphRepository* contengono al loro interno i metodi che interagiscono con il database seguendo la seguente struttura:

```
@Query("queryDB")
valoreRitorno NomeMetodo(parametr1, parametroN);
```

Inizialmente non si aveva necessità di far tornare record dal database, infatti interessava solo inserire i log al suo interno; in seguito si è reso necessario dovendo inserire la gestione degli scenari di training e i controlli per incrementare la stabilità dell'applicazione. A tal scopo sono state create le classi ReferenceNode, ReferenceGraph, ActualNode, TraineeGraph, ReferenceTransition, ActualTransition che permettevano una gestione più ottimizzata e coerente sui domini di dati che venivano trattati.

3.2.2 Data Domain

Il dominio dei dati trattati all'interno della applicazione dovevano avere una struttura coerente con quelli utilizzati nel DB a grafi, per questo sono state create delle classi

apposite per quasi tutte le singole tipologie di nodi e delle relazioni che definiscono le loro interazioni. Oltre al motivo citato nella sezione precedente, questa soluzione si è resa necessaria ci si è resi conto che non era più possibile effettuare ogni singola operazione con una sola query in Cypher ma bisognava scomporre un problema in più pezzi facendo richieste diverse. Dal momento in cui ho spezzettato le query, queste dovevano ritornarmi dei valori interni al database che dovevano essere poi utilizzati nelle query successive, da qui la creazione dei domini di dati. Come ho appena spiegato, non si è reso necessario implementare tutti i possibili domini dei dati ma soltanto quelli necessari per performare le operazioni, infatti alcuni dati raccolti dal database potevano essere modellati con tipologie già presenti all'interno del linguaggio o delle librerie Java.

3.3 Algoritmo di Scoring

All'interno di una CyberRange, una parte fondamentale è il punteggio che viene assegnato a ciascuno dei Trainii; questo servirà a dargli una idea di quanto hanno ancora da migliorare sui singoli argomenti approfonditi nei diversi scenari e avere un quadro generale delle loro competenze.
Anche ai professori risulteranno utili questi punteggi poichè permetteranno di dare una chiara idea di quali argomenti sarebbe meglio rivedere o revisionare, nel caso la media dei punteggi per argomento non dovesse raggiungere una specifica soglia, per gli anni successivi. Di seguito andrò a spiegare come è stato modellato il calcolo dei punteggi, per farlo mi sono attenuto al paper dei docenti della mia facoltà.

Sia L_r la lunghezza del cammino più corto nel ReferenceGraph G_r e sia L_t la lunghezza performata dal training del Trainee all'interno del TraineeGraph G_t, la funzione che calcola il punteggio è la seguente:

$$s(G_r, G_t) = \begin{cases} l_r/l_t \text{ se esiste un percorso dal nodo iniziale al nodo finale in } G_t \\ 0 \text{ altrimenti} \end{cases}$$

Questa modellazione del problema è stata implementata in 2 metodi diversi:

1. **Realtime**: Il calcolo del punteggio viene fatto ogni volta che viene aggiunto un arco al relativo grafo del trainee e viene inserito nel primo nodo(ovvero il referenceGraph)

2. **Post-mortem**: Con post-mortem si intende che l'operazione viene effettuata alla fine del training avendo la visione complessiva e completa dei percorsi fatti

Ci tengo a specificare che il calcolo dello scoring funziona solo con un cammino di azioni senza più percorsi che portano allo stesso nodo.
Il motivo di questa carenza è che altrimenti dovevo controllare quali cammini sono stati svolti all'interno del ReferenceGraph seguendo il flusso di esecuzione. Avevo trovato una soluzione che ritornava tutti i cammini possibili, dal primo nodo fino a quello finale, e li confrontavo con quello fatto dall'utente calcolando il risultato, l'algoritmo era altamente inefficiente e ho preferito rimuoverla per una implementazione futura completamente in Cypher.

3.4 Il labirinto degli Scenari

Andando più a fondo nell'analisi delle casistiche abbiamo identificato diversi comportamenti che il sistema doveva modellare tramite l'algoritmo di matching e andrò a darvi una spiegazione per ognuno di essi in modo che, quando guarderete l'algoritmo, abbiate chiaro il suo comportamento. Inizialmente abbiamo definito il TraineeGraph come il grafo delle azioni che l'utente compie e che comportano un cambiamento tra i diversi stati del training: questo termine è, però, una generalizzazione di più scenari distinti.

3.4.1 L'ordine è importante?

Un primo esempio di scelta implementativa che è stata fatta riguarda l'ordine delle operazioni in ingresso e se queste dovevano seguire in modo rigoroso le azioni presenti nel ReferenceGraph. Porto un semplice esempio in modo da rendere chiaro l'aspetto del problema. L'obbiettivo del training è di fare imparare al trainee i comandi della bash di linux e gli viene messa a disposizione una macchina virtuale con solo il terminale, senza permessi di amministratore, dove deve assolvere i seguenti compiti:

1. Diventare amministratore
2. Scoprire quali file e cartelle sono presenti all'interno di una cartella target
3. Leggere il contenuto del file 'tesi.txt'

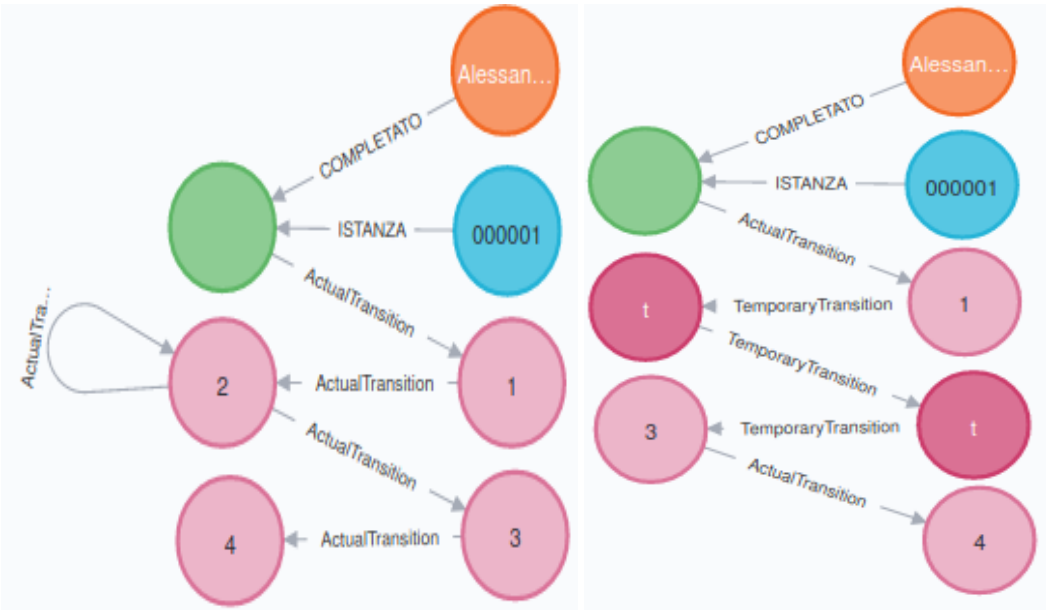
Ammettendo che per assolvere il secondo e il terzo compito non sia necessario avere i permessi di amministratore, nessuno dei seguenti comandi è propedeutico agli altri mettendo quindi il trainee nella condizione di eseguirli in ordine arbitrario.
Cambiando esempio, ho sempre uno scenario di training in cui l'utente vuole sempre imparare i comandi di linux ma per assolvere il secondo e il terzo compito deve avere i permessi di amministratore: in questo caso l'ordine di esecuzione delle operazioni è importante. Analizziamo il primo caso messo a disposizione avendo un algoritmo

di matching che effettua il confronto dando importanza all'ordine delle operazioni. Sotto il punto di vista matematico ho n azioni che possono essere eseguite in $n!$ modi diversi. Questa soluzione porterebbe a diversi problemi quali:

- Aumento esponenziale della dimensione del ReferenceGraph
- Aumento del tempo e del carico computazionale per effettuare il Matching tra il comando in ingresso e il ReferenceGraph.

Per ovviare a questo problema si è quindi deciso che l'ordine delle operazioni riguardasse il singolo scenario di training e che non ci fosse una decisione uniforme per tutti gli scenari. L'algoritmo sceglie il metodo da utilizzare in base ad un flag deciso dallo sviluppatore, in seguito potrà essere passato come parametro nei messaggi o inserito all'interno del primo nodo del ReferenceGraph.

Adesso che dovrebbe essere chiaro il perchè l'ordinamento è importante, iniziamo a definire il comportamento del sistema sistema quando si imbatte in una di queste situazioni, tenendo come riferimento il ReferenceGraph in figura 2.ReferenceGraph.



(a) TraineeGraph (b) TraineeGraph

Figura 2: Esempi di TraineeGraph

Ordinamento semplice

Iniziamo analizzando il programma che prende in ingresso i seguenti messaggi:

```
{'timestamp': 1, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'ls', 'SessionID':'000011'}

{'timestamp': 2, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'ll', 'SessionID':'000011'}

{'timestamp': 3, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'pwd', 'SessionID':'000011'}

{'timestamp': 4, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'cmd', 'SessionID':'000011'}
```

In questo caso verrebbe a crearsi un grafico come quello in figura 1.TraineeGraph dove:

- Il nodo in **arancione** identifica il *Trainee*
- Il nodo in **azzurro** identifica lo *ScenarioVm*
- Il nodo in **verde** identifica il *TraineeGraph*
- I nodi in **rosa** identificano gli *ActualNode* e i collegamenti tra essi le diverse ActualTransition

Se guardiamo l'*ActualNode 1* notiamo che ha un arco che punta a se stesso, questo perchè il comando *ll* non rientra all'interno delle azioni definite nel ReferenceGraph, l'utente quindi non è stato in grado di passare allo step successivo del training. Da adesso in poi definiremo gli archi che partono e tornano agli stessi nodi come degli errori da parte del Trainee durante la sessione di training.

Ordinamento con salto nodo

Analizziamo ora il caso in cui vengono passati i seguenti log, ritenendo sempre importante l'ordinamento, ma saltando un nodo all'interno del cammino.

```
{'timestamp': 1, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'ll', 'SessionID':'000011', 'TeamID':'null',
    'ScenarioID':'null'}

{'timestamp': 2, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'ls --full', 'SessionID':'000011', 'TeamID':'null',
    'ScenarioID':'null'}
```

```
{'timestamp': 3, 'hostname':'893181', 'ScenarioVM':'000001',
    \textbf{'Action': 'pwd'}, 'SessionID':'000011', 'TeamID':'null',
    'ScenarioID':'null'}

{'timestamp': 4, 'hostname':'893181', 'ScenarioVM':'000001',
    'Action':'cmd', 'SessionID':'000011', 'TeamID':'null',
    'ScenarioID':'null'}
```

Come possiamo notare, nelle prime due righe del log vengono passate due azioni: (*ll* e *ls –full*) che non risiedono nel ReferenceGraph, nella terza si passa direttamente al nodo 3 saltando il nodo 2 e non performando l'azione *ls*.
Per il sistema questo comportamento è inaspettato poichè l'utente ha performato una azione, che porta ad uno stato successivo, che non risiede all'interno del ReferenceGraph; seguendo questa logica, ogni azione performata dal penultimo cambiamento di stato fino ad adesso(conteggiate come sbagliate) può essere stata utile al trainee per trovare questo nuovo cammino.
Come il sistema si accorge di questo salto di nodi e come fa ad intervenire verrà analizzato in seguito nella sezione *Algoritmo di Matching*, di seguito andremo ad elencare i passi che esegue la funzione che ripristina il cammino in caso sia importante l'ordine delle operazioni:

1. Identificazione dell'ActualNode $\boldsymbol{a}$, precedente al nodo appena saltato, che chiameremo ***a'***

2. Viene memorizzata ogni transazione $\boldsymbol{r} \in \boldsymbol{R}$ tale che (a)-[r:ActualTransition]-¿(a)

3. Nel caso in cui *size(R)=0*, collego *a* con *a'* tramite una TemporaryTransition, ossia una transazione di stato contenente gli stessi parametri di una ActualTransition ma cambiandogli la tipologia in modo da poterla classificare come *transazione temporanea* che necessita di revisione.

4. Per ogni transazione *r* in R performo le seguenti azioni:

   - Se *size(R) > 1* e sono alla prima iterazione, creo il primo TemporaryNode *tn*(ovvero un nodo con parametri analoghi agli ActualNode ma con tipologia diversa in modo da poterli distinguere dai precedenti) e una TemporaryTransition dall'ActualNode *a* a *tn*
   - Se *size(R) > 1* e non sono alla prima iterazione del ciclo, creo un nuovo TemporaryNode e collego al precedente
   - Nel caso *size(R) = 1* collego il TemporaryNode creato al passo precedente con *a'*

Nel caso non si tenesse conto dell'ordine di esecuzione, non verrebbero creati TemporaryNode ne TemporaryTransition ma, bensì, un grafo con solo ActualNode e ActualTransition.
S è deciso di creare e partire con il grafico dal primo ActualNode e non dal nodo TraineeGraph in modo che, nel caso si saltasse un nodo alla prima azione, non bisognasse aggiungere ulteriori casistiche all'algoritmo di maching complicandone l'esecuzione.

3.4.2 Scenari improbabili

In questa sezione tratterò alcune decisioni implementative relative a scenari che hanno statisticamente poca possibilità di accadere ma che per coerenza ho dovuto gestire in modo da mantenere la basi di dati stabile. Premetto che alcuni di questi non riguardano prettamente questo capitolo, ma è giusto specificarli qui in modo da poterli raggruppare e avere una visione complessiva dei problemi risolti e poter entrare più nel dettaglio nelle decisioni che riguardano anche questo capitolo.

Duplicazione messaggio iniziale

Abbiamo detto precedentemente che ci sono due metodi per la creazione dell'TraineeGraph: tramite API e tramite messaggi in ingresso. Nel caso in cui non venga creato l'TraineeGraph tramite API ma vengano ricevuti per errore due messaggi iniziali con gli stessi parametri, l'applicazione crea un'TraineeGraph quando il primo dei due messaggi viene ricevuto. Questa operazione avviene all'interno dell'TraineeGraphRepository dentro la funzione 'createTraineeGraph' a cui passo il nome della VM sui cui l'utente sta facendo training e l'hostname della macchina(ovvero la matricola del trainee). La funzione completa, scritta tramite Cypher, è la seguente:

```
MATCH (t:trainee {matricola: $hostname}),
(vm:Vm {name: $matricola})
WHERE NOT EXISTS{
MATCH(t:trainee {matricola: '893180'}) -[:SVOLGIMENTO]->
      (TraineeGraph) <- [:ISTANZA]- (vm)
}
CREATE (a:ActualTrace),
(ann:ActualNode{name: '1', status:'Iniziale'})
CREATE (a)<-[:ISTANZA]-(vm)
CREATE (a)<-[:SVOLGIMENTO]-(t)
CREATE (a)-[:ActualTransition{timestamp: '0'}]->(ann)
```

All'arrivo del secondo messaggio, l'applicazione riconosce che c'è già un'ActualTrace relativo e questo viene scartato. Questo particolare controllo lo effettuo tramite questo comando all'interno della funzione:

```
MATCH(t:trainee {matricola: '893180'}) -[:SVOLGIMENTO]->
      (ActualTrace) <- [:ISTANZA]- (vm)
}
```

3.5 Gestione della code RabbitMQ

La classe Receiver ha un ruolo di rilievo all'interno del progetto poiché è dove è stato svolto la maggior parte del lavoro e dove risiede la vera capacità dell'applicativo di rispondere ai messaggi presi dalle code di RabbitMQ e ritornare i relativi output in base alle diverse casistiche. Di seguito andrò ad inserire porzioni di codice (sia del TraineeGraphRepository sia della classe Receiver) in modo da poterlo analizzare e mettere in risalto quali sono state le decisioni logiche che si sono dovute prendere nello sviluppo delle diverse implementazioni. Innanzitutto viene creata una istanza della classe *TraineeGraphRepository* che permette l'interazione con il database Neo4j e una di *LogbackRabbitMQ*.

```
private final TraineeGraphRepository TraineeGraphRepository;
private final LogbackRabbitMQ logbackRabbitMQ;
```

Di seguito vengono inizializzati i flag che regolano rispettivamente: se deve essere conteggiato l'ordine delle operazioni, quale tipologia di scoring system utilizzare.

```
int FLAG_ORDINE = 1;   // 0 CON ORDINE | 1 SENZAORDINE
int FLAG_SCORING = 1; // 0 POSTMORTEM | 1 REALTIME
```

Attraverso il metodo *receiveMessage(String message)* prendo in ingresso come parametro di tipo stringa(String) il messaggio in json dalla coda di RabbitMQ e lo passo alla funzione *parseJSON* in cui viene inizializzato un *JSONObject obj* con il messaggio appena preso e effettuo il parsing di tutti i parametri in esso contenuti prendendo:

- **hostname** dell'uente che sta facendo training
- **action**: l'azione che potrebbe performare il cambiamento di stato
- **vm**: L'identificativo dello scenario a cui si fa riferimento
- **sessionID**: Identificativo della sessione di lavoro
- **teamID**: Identificativo del team a cui l'utente potrebbe appartenere
- **scenarioID**: Il relativo scenario a cui il team del trainee potrebbe partecipare

- **timespamp**: Il tempo, calcolato in millisecondi, in cui viene performata l'azione all'interno della macchina di training

Una volta finito questo procedimento, viene chiamata la funzione *Controller* in cui vengono passati tutti i parametri appena presi; in questa risiede l'algoritmo di matching. Per rendere chiara l'idea del flusso d'esecuzione dell'algoritmo ne riporto una versione riscritta in pseudocodice divisa in sezioni e in seguito andrò ad approfondire ogni sezione.

3.5.1 Algoritmo Matching tramite Controller

```
1_RamoPrincipale
if(controlloFineSessione) {
  1.1_RamoInSessione
  if(controlloAppartenenzaTeam) {
      1.1.0_CreazioneTeamScenario
  }
  if(ControlloComandoInReferenceGraph) {
    1.1.1_ComandoInReferenceGraph
    if(ControlloOrdineLog == 0) {
          1.1.2_ValeOrdine
          if(ControlloSaltoNodo) {
            1.1.3_NessunSaltoNodo
                CreazioneArcoCorretto
          }
          else {
            1.1.4_SaltoNodo
                if(size(R)>0) {
                  1.1.5_ProceduraRipristino
                }
                else {
                  1.1.6_CasisticaParticolare
                }
          }
        }
        else {
          1.1.7_CreazioneArcoCorretto
        }
  }
  else {
```

```
    1.2_AzioneErrata
  }
}
else {
        2_ComandoFuoriTraining
}
```

1_RamoPrincipale

Le funzioni con carattere bold definite all'interno di questa sezione non risiedono all'interno del TraineeGraphRepository bensì all'interno della stessa classe Receiver e verranno analizzate più in dettaglio nella prossima sezione. Per prima cosa vengono inizializzate le variabili che racchiudono le seguenti informazioni: ID del primo nodo, ID dell'ultimo nodo e il nome dello stesso.

```
Long id_firstN = null;
Long id_lastN = null;
Long nome_lastN = null;
```

La prima funzione del *TraineeGraphRepository* che viene chiamata ritorna, se esiste, l'identificativo del TraineeGraph relativo al messaggio controllando le seguenti informazioni:

- Se esiste un utente con la specifica matricola del messaggio
- Che questo utente abbia una relazione di tipo *COMPLETATO* con il TraineeGraph
- Prende il nodo *ScenarioVm* identificato dal nome contenuto nella variabile *vm* e controlla se esiste una relazione di tipo *ISTANZA*, con al suo interno un parametro *SessionID* che combacia con quello del messaggio, con il nodo TraineeGraph precedente
- Ritorna l'id di tale nodo se esiste, altrimenti un valore null

Tutte queste operazioni sono performate dalla seguente query in Cypher:

```
MATCH (t:Trainee {matricola: $hostname}) -[:COMPLETATO]->
    (tt:TraineeGraph) <- [:ISTANZA{SessionID: $sessionID}]-
    (vm:ScenarioVm {name: $vm})
RETURN ID(tt)
```

Faccio un controllo che l'ID ritornato non sia nullo e ho due possibilità:

1. La sessione è ancora in corso: entro nel ramo 1.1_RamoInSessione

2. La sessione a cui il messaggio si riferisce è conclusa:
   si entra nel ramo *2._ComandoFuoriTraining* e viene ritornato il relativo messaggio d'errore.

1.1_RamoInSessione

Per prima cosa lancio una query che mi crea sia il TraineeGraph e i collegamenti ad esso associati, sia il primo ActualNode; nello specifico, la funzione in cypher chiamata è la seguente:
Long idPrimoNodo = (Long) TraineeGraphRepository.createTraineeGraph(vm, hostname, sessionID);

```
MATCH (t:Trainee {matricola: $hostname}),(vm:ScenarioVm {name: $vm})
WHERE NOT EXISTS{
MATCH(t:Trainee {matricola: $hostname}) -[:SVOLGIMENTO]->
    (TraineeGraph) <- [:ISTANZA{SessionID: $sessionID}]- (vm)
}
CREATE (a:TraineeGraph {score: 0, numRelazioni: 0}),
    (ann:ActualNode{name: '1', status:'Iniziale'})
CREATE (a)<-[:ISTANZA {SessionID: $sessionID}]-(vm)
CREATE (a)<-[:SVOLGIMENTO]-(t)
CREATE (a)-[:ActualTransition{timestamp: '0'}]->(ann)
RETURN ID(ann)
```

Sempre di seguito eseguo un controllo che mi permette di risparmiare capacità computazionale, infatti:

- Se esiste già il cammino, l'ID ritornato dalla query avrà valore *null* e dovrò performare un'altra query per andare a riprendere i valori relativi all'ID del primo e dell'ultimo ActualNode del TraineeGraph
- Se non esiste il cammino posso evitare di performare ulteriori query infatti il valore ritornato dalla precedente funzione sarà l'ID sia del primo che dell'ultimo nodo

Prendo il nome(ovvero l'attributo name) all'interno dell'ultimo nodo del TraineeGraph attraverso la funzione *getNomeLastNode* in cui gli passo come parametro l'ID del primo nodo del sottografo.

Il controllo successivo che viene fatto riguarda l'appartenenza del trainee ad un team e quindi al relativo scenario. Ho ipotizzato che il parametro team e scenario ci fossero sempre all'interno dei messaggi che vengono inviati all'applicativo, nel caso il trainee stesse facendo una sessione di training da solo viene inserito il valore null ad entrambi i parametri. Quindi, nel caso questi valori non fossero null e sempre nel

caso non esistessero i relativi nodi e relazioni nel grafo, li creo all'interno del ramo *1.1.0_CreazioneTeamScenario*.

Una parte fondamentale di tutto il programma è la funzione che adesso andremo ad analizzare chiamata **controlloLogInReferenceGraph**, questa non fa altro che controllare se l'azione performata dall'utente esiste o meno all'interno del ReferenceGraph relativamente allo specifico *ScenarioVm*. La funzione in cypher è scritta di seguito.

```
match p=(vm:ScenarioVm{name: $vm})-[:HASTEMPLATE]->
      (et:ReferenceGraph)-[*1..]->(en:ReferenceNode)-
      [r*1..{Action: $action}]->(enn:ReferenceNode)
return distinct [en.name, enn.name, enn.status]
```

Una volta preso il nodo *ScenarioVm* con lo stesso identificativo passato nel messaggio, controllo se esiste una relazione di tipo *HASTEMPLATE* che lo collega al ReferenceGraph utilizzato per confrontare le azioni del trainee. Da questo nodo chiedo di controllare le relazioni di tutti i successivi ReferenceNode e verificare se al loro interno esiste un parametro *Action* con valore quello effettuato dall'utente durante il training che ha comportato un cambiamento di stato. In questa specifica query ritorno una terna di valri, ovvero:

1. **en.name**: Il nome del nodo antecedente a quello raggiunto dall'azione
2. **enn.name**: Il nome del nodo successivo all'azione performata dall'utente
3. **en.status**: Lo stato del nodo che verrà utilizzato per controllare se il training è arrivato alla sua conclusione

Nel caso il valore di ritorno della terna avesse valore *null* si entrerebbe nella sezione *1.2_AzioneErrata*, altrimenti l'azione viene ritenuta corretta e si entrerebbe nella sezione *1.1.1_ComandoInReferenceGraph*. Da notare che i parametri ritornati saranno di fondamentale importanza all'interno del programma e, come successo per gli ID e i nomi dei primi nodi presi precedentemente, richiederli all'interno di questa query permette di risparmiare carico computazionale evitando di lanciare altre 2 query distinte.

1.1.1_ComandoInReferenceGraph

La prima cosa che viene fatta una volta entrati in questo ramo è inserire i 3 parametri, presi nella funzione precedente, e inserirli in 3 variabili distinte per una migliore gestione. In questo istante viene inserito il controllo sull'ordine delle operazioni derivante dal flag settato precedentemente prima dell'esecuzione dell'applicativo; ovviamente non avrebbe senso inserire il controllo ad una granularità maggiore poichè porterebbe ad una aggiunta inutile di operazioni di controllo e ad una peggiore gestione e

comprensione del codice. Come già accennato precedentemente, sono stati presi in considerazione due scenari di ordinamento:

- Conta l'ordine delle operazioni: Possibili salti nodo portano a casistiche particolari o all'utilizzo della operazione di ripristino
- Senza contare l'ordine non vengono fatti controlli aggiuntivi semplificando la gestione delle azioni del trainee

1.1.2_ValeOrdine

In questa sezione analizzeremo in dettaglio quali operazioni aggiuntive sono necessarie per soddisfare i requisiti espressi precedentemente. Il primo algoritmo pensato per controllare se ci fosse stato un salto nodo all'interno del ReferenceGraph performava queste operazioni:

1. Identificazione del nodo successivo al cambiamento di stato(che chiameremo n) portato dall'azione compiuta dal trainee
2. Identificazione del nodo antecedente a quello nominato al punto $1(t)$.
3. Dato che i nodi all'interno del ReferenceGraph seguivano un ordine specifico inserito all'interno del campo name, prendevo questo valore da entrambi i nodi sopracitati e verificavo che *size(t[name]) +1 == size(n[name])*

Questa soluzione non prendeva in considerazione lo sviluppo di ReferenceGraph più complessi come quello in figura 2.ReferenceGraph in cui ci potesse essere nativamente un salto di nodo all'interno del cammino. Per ovviare a questo problema, si è deciso di valutare l'esistenza di una relazione tra i nodi al punto 1. e 2.
Viene lanciata la funzione findArcoInReferenceNode con i seguenti parametri:

- **vm**: L'identificativo dello ScenarioVm a cui il training si riferisce
- **id_f**: Il numero contenuto all'iterno del campo name del nodo a cui mi sono riferito al punto 1 del precedente elenco
- **id_l**: Il numero contenuto nel campo name dell'ultimo nodo che deve essere aggiunto al TraineeGraph

Vengono utilizzati questi parametri poichè erano già stati ricavati da funzioni precedenti, evitando ulteriori query al sistema. La funzione esegue la seguente query nel database:

```
match p=(vm:ScenarioVm{name:$vm})-[:HASTEMPLATE]->
    (e:ReferenceGraph)-[r*0..]->(en:ReferenceNode{name: $id_f})-
    [et:ReferenceTransition{Action: $action}]->
    (enn:ReferenceNode{name: $id_l})
return distinct ID(enn)
```

Il valore ritornato non è altro che l'ID relativo al nodo del ReferenceGraph associato al cambiamento di stato performato dall'ultima azione; una volta ottenuto controllo che questo valore non sia null: in caso affermativo chiamo la funzione **creazioneArcoCorretto()** che mi crea l'ActualTransition e l'ActualNode relativi, in caso contrario si è verificato un salto di un nodo ed entro nel ramo [1.1.4_SaltoNodo].

1.1.4_SaltoNodo

Prima di passare subito all'utilizzo della funzione di ripristino, utilizzo la funzione contSelfNodeRelationship per ritornarmi quante self-relationship ci siano sullo stesso nodo(precedentemente definito ***a***); una volta fatto controllo se sono maggiori di zero. In caso affermativo chiamo la funzione **proceduraDiRipristino** che andrà a crearmi il cammino con i TemporaryNode e le TemporaryTransition associate; In caso non esistano relazioni di questo tipo sul nodo si entra all'interno di una casistica particolare(definita dal ramo 1.1.6_CasisticaParticolare) dove non vengono creati TemporaryNode ma solo una TemporaryTransition tra il nodo *a* e quello *a'*. Una considerazione importante da fare è la seguente: ogni volta che si arriva ad un possibile ramo conclusionale dell'algoritmo in cui viene performata una azione, viene anche fatto un controllo sulle funzioni di scoring.

```
if(FLAG_SCORING == 1) {
        score_REALTIME(id_firstN , nome_lastN);
}
```

Il codice sopra descritto(inserito solo nel ramo 1.1.6 e non in 1.1.5) controlla il flag FLAG_SCORING, in base al suo valore si decide se chiamare la funzione di scoring realtime o meno; questa operazione viene ripetuta spesso all'interno del programma, per evitare di creare ripetizioni utilizzerò la dicitura **controlloScoringRealtime()** per richiamare a questa porzione di codice.

1.1.7_CreazioneArcoCorretto

In questo specifico ramo viene presa in considerazione la casistica in cui non viene conteggiato l'ordine dei nodi e l'azione eseguita dal trainee risiede all'interno del ReferenceGraph. L'unica cosa che rimane da fare è chiamare la funzione **creazioneArcoCorretto** che permetterà di inserire la relativa ActualTransition con il successivo ActualNode.

1.2_AzioneErrata

A seguito dell'esito positivo sul controllo del valore null ritornato dalla funzione che verifica l'esistenza dell'azione computa dal trainee con quelle presenti nel ReferenceGraph, viene utilizzata l'istanza del TraineeGraphRepository per chiamare la funzione createSelfArco. Il suo scopo è chiaro, creare un arco che parte dall'ultimo nodo del TraineeGraph e punta a se stesso. **controlloScoringRealtime()**

3.5.2 Funzioni specifiche

Precedentemente abbiamo analizzato tutti i possibili rami che il sistema creava per gestire ogni possibile casistica e, una volta arrivati alla fine di queste, dicevamo che chiamava delle funzioni specifiche ma senza mai definire il loro funzionamento, cosa che invece andremo a fare in questa sezione.

creazioneArcoCorretto()

Questa funzione viene chiamata all'interno dei rami: 1.1.3, 1.1.7 e prevede di inserire all'interno del database un'ActualTransition e un ActualNode collegati rispettivamente all'ultimo ActualNode relativo alla penultima azione che ha comportato un cambiamento di stato da parte del trainee. La funzione all'interno del TraineeGraphRepository che viene chiamata è *createRightArco* e utilizza la seguente query in cypher:

```
MATCH(an) where ID(an)=$id_l
CREATE (a:ActualNode {name: $name, status: $status})
CREATE (a) <- [at:ActualTransition {timestamp: $timestamp,
    Action: $action}]-(an)
```

I parametri che gli vengono passati sono:

- **id_l**: ID ultimo nodo del cammino fino a quel momento
- **name, status**: Nome e stato del nuovo ActualNode che viene creato
- **timestamp, action**: Quando l'azione è stata performata e che tipo di azione ha portato al cambiamento di stato, queste informazioni vengono utilizzate per la creazione della TemporaryTransition

controlloScoringRealtime()

Come è stato fatto con la funzione sopracitata, definirò un'altra macro chiamata **controlloEndTraining()** che performa la sequente porzione di codice:

```
if(lastNode_status.equals("Stop")) {
        endTraining(id\_firstN, vm, hostname, sessionID);
}
```

Al suo interno controllo che lo stato dell'ultimo nodo inserito all'interno del grafo sia 'Stop', se si chiamo la funzione che si occupa della chiusura del training del trainee.

proceduraDiRipristino()

Questa funzione, chiamata nel ramo *1.1.5_ProceduraRipristino*, inizia facendosi ritornare tutte le self-relationship, in ordine crescente di timestamp, dell'ultimo nodo presente nel TraineeGraph tramite la funzione *getSelfNodeRelationship(lastNode_id)* con il seguente codice all'interno del TraineeGraphRepository:

```
MATCH (an:ActualNode)<-[at:ActualTransition]-(an) WHERE ID(an)=$id
    WITH an,at ORDER BY at.created_at DESC RETURN an, collect(at)
```

Una volta salvate vengono eliminate dal nodo a cui appartenevano poichè in seguito dovranno essere trasformate in TemporaryTransition. L'utilità di questa funzione non risiede soltanto nel differenziare i cammini ritenuti 'inaspettati' dal sistema all'interno del database per performare query statistiche ma, grazie ad essa, è possibile inserire questa serie di operazioni in un vettore che verrà inviato ad una coda specifica di RabbitMQ che potrà essere analizzata da operatori esterni per verificare la veridicità dei cammini e, nel caso, modificare il ReferenceGraph di quello *ScenarioVm* relativo al training che quel trainee stava svolgendo.
L'operazione di inserimento di questo cammino all'interno di un vettore inizia con l'istanziazione dello stesso all'interno del programma, ovvero risiede nella creazione di un *ArrayList¡TemporaryNode¿*; ogni volta che verrà creato un nuovo TemporaryNode darò per scontato che questo venga inserito all'interno di questa lista che chiameremo *TemporaryArrayList*.
Inizialmente l'algoritmo per la creazione di questo particolare cammino era stato pensato come un'unico ciclo che, in base al numero di relazioni ritornate precedentemente, creava i relativi TemporaryNode e le relative TemporaryTransition; questa soluzione comportava i seguenti problemi:

- Il primo TemporaryNode va collegato all'ultimo ActualNode presente nel cammino(dove risiedevano le self-relationship) e questo comportava l'aggiunta di controlli all'interno dell'algoritmo che dovevano essere fatti per ogni iterazione del ciclo

- Stessa problematica per quanto riguarda l'ultima relazione che parte dall'ultimo TemporaryNode e arriva al primo ActualNode(relativo al cambiamento di stato dell'ultima azione eseguita)

Per ovviare a questa aggiunta di controlli ho deciso di differenziare la creazione dei TemporaryNode e delle relative TemporaryTransition in 3 funzioni che andrò a spiegare di seguito.

1. Creazione del primo TemporaryNode tramite la funzione createFirstTemporaryNode
2. Sia n il numero di self-relationship estratte precedentemente, il ciclo effettua da 1 fino ad (n-2) iterazioni in cui crea delle TemporaryTransition collegate ad un successivo TemporaryNode
3. Una volta usciti dal ciclo, viene creata una TemporaryTransition tra l'ultimo TemporaryNode e il quello sopracitato ActualNode

Per far si che le informazioni salvate all'interno dell'ArrayList *TemporaryArrayList* vengano inviate alla lista di RabbiMQ, ho preferito utilizzare la libreria Gson di google in modo da trasformarlo i un vettore in JSON. Una volta finita la trasformazione invio i dati alla lista *TemporaryTrace* tramite il seguente comando:

```
logbackRabbitMQ.send(json,"TemporaryTrace");
```

Una volta fatto eseguo le seguenti macro: **controlloScoringRealtime()** , **controlloEndTraining()**.

scoring_REALTIME()

Lo scoring_REALTIME conta il numero di relazioni contenute nel ReferenceGraph fino a quel punto del training, queste vegnono divise con il numero di relazioni attualmente presenti all'interno del TraineeGraph ottenendo così un numero compreso tra [0,1]. Per evitare di lanciare una query che conta il numero di relazioni all'iterno del TraineeGraph, si è deciso di utilizzare un contatore inserito nel nodo TraineeGraph che viene incrementato di uno ogni volta che la funzione viene chiamata; per quanto riguarda il numero di relazioni all'interno del ReferenceGraph devo per forza richiamare la funzione *getNumRelazioniReferenceGraph*. Una volta ottenuto il risultato aggiorniamo il contatore e concludiamo la funzione.

scoring_POSTMORTEM()

Lo scoring_POSTMORTEM utilizza la stessa funzione di assegnazione del punteggio del precedente metodo, con la differenza che questa viene eseguita soltanto quando il percorso di training giunge alla sua conclusione. Il numero di relazioni all'interno del TraineeGraph viene calcolato attraverso la funzione *contRelTraineeGraph_POSTMORTEM* ma il risultato viene sempre inserito all'interno del nodo *TraineeGraph*.

endTraining()

Questa funzione è l'ultima che andremo ad analizzare e decreta anche la fine di tutto il percorso di training performato dall'utente. Al suo interno vengono eseguite le seguenti operazioni:

1. La prima operazione che viene eseguita è, nel caso ci sia, eliminare il contatore di relazioni all'interno del TraineeGraph
2. Viene ritornato il grafico ***G*** delle operazioni eseguite fino a quel momento
3. ***G*** viene convertito in JSON(tramite la libreria Gson) e inviato alla coda *End-Training* di RabbitMQ
4. viene chiamata la funzione *scoring_POSTMORTEM()*
5. Come ultima operazione cambio la tipologia di relazione che collega il Trainee-Graph al nodo Trainee da *SVOLGIMENTO –¿ COMPLETATO*

3.6 RabbitMQ implementation

Ho ritenuto più opportuno non istallare RabbitMQ all'interno del mio pc (avente S.O. Ubuntu Linux 20.10), bensì pormi già in un'ottica implementativa futura scegliendo quindi di utilizzare Docker per fare il deploy dell'applicazione. Per lanciare l'applicazione utilizzavo un terminale linux lanciando il seguente comando:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672
    rabbitmq:3-management
```

3.7 Sviluppo Agent

Arrivato ad un certo punto dello sviluppo della applicazione ho avuto la necessità di verificare se l'algoritmo di maching potesse funzionare in una casistica reale infatti, fino ad adesso, le venivano passati log preconfigurati in base agli scenari da testare. L'operazione di raccolta delle informazioni all'interno dei singoli agent doveva essere fatta da un'altro tesista ma, avendone necessità immediata, decisi di sviluppare da solo un sistema di raccolta log con le conoscenze acquisite nelle fasi iniziali di ricerca dei componenti.
Utilizzai Fluentbit per la raccolta dei comandi digitati tramite terminale, in seguito questi dovevano essere modificati per poi inviarli al MessageBroker di RabbitMQ;

come agent decisi di utilizzare il mio RaspberryPi2 collegato alla lan domestica in cui avevo abilitato la possibilità di connessione tramite SSH. Trovai le prime complicazioni quando mi accorsi che non esistevano plugin che permettessero un output diretto tra Fluentbit e RabbitMQ, allora iniziai ad informarmi su una possibile implementazione che comprendesse Fluentd per il passaggio dei log a RabbitMQ, sapendo possibile una comunicazione tramite Fluentbit e Fluentd. Trovai due plugin, resi disponibili per Fluentd, che ne permettessero l'interfacciamento con RabbitMQ:

- Il primo richiedeva l'istallazione di Docker, non potevo permettermelo data la carenza computazionale a disposizione dell'agent
- Il secondo dava problemi con la creazione delle code dei messaggi in arrivo quando volevo una architettura il più dinamica possibile quindi decisi di scartare anche questa possibilità

Trovandomi senza opzioni a disposizione, decisi di risolvere il problema creando un client di RabbitMQ in Python, reso disponibile dalla documentazione, e collegarlo a fluentd. Il client utilizzato è il seguente:

```
#!/usr/bin/env python
import pika
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
    routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

Questa soluzione permette la connessione con RabbitMQ ma consente l'invio di un solo messaggio di log preconfigurato e andava quindi modificato in modo che, per ogni messaggio ricevuto da Fluentd, questi venissero messi in un buffer in cui il nostro client leggesse e inviasse di conseguenza i log a RabbitMQ. Una volta chiaro come doveva essere l'infrastruttura iniziai con l'installazione di Fluentbit, Fluentd e alla loro configurazione, lasciando il client RabbitMQ per ultimo.

Fluentbit

Di seguito posto il contenuto del file di configurazione di Fluentbit, chiamato *td-conf.conf*.

```
[SERVICE]
    Flush 1
    Log_Level    info

#GetBashLog
[INPUT]
    Name    tail
    Path    ~/.bash_history

[FILTER]
    Name    record_modifier
    Match   *
    Record  hostname ${HOSTNAME}
    Record vm SeedUbuntu
    Record timestamp ${TIMESTAMP}

#Output
#[OUTPUT]
#    Name stdout
#    Match *

#Output Fluentd
[OUTPUT]
    Name http
    Match *
    Host 192.168.1.3
    Port 8888
    Format json

    Uri /VM
    header_tag Fluentbit
```

Per prima cosa dovevo prendere l'elenco dei comandi digitati su terminale, in linux questi vengono salvati sul file:
' ~ /.bash_history' e sono visibili tramite il comando *history*. Serviva quindi utilizzare un programma di lettura del contenuto del file simile al comando *tail -f* della

bash linux. Nei plugin messi a disposizione da Fluentbit ne esisteva uno che assolveva esattamente questo compito e la sua configurazione è inserita all'interno del tag *[INPUT]*, dove viene definito inoltre:

- Il nome del plugin, in questo caso *tail*
- Il percorso, o path, in cui leggere il contenuto del file. In questo caso ho definito 2 percorsi: uno è relativo al vero file in cui vengono salvati i comandi e mi serviva quando dovevo fare dei test reali della applicazione, l'altro è un percorso ad un file creato da me con dei comandi ad-hoc per testare l'applicazione con scenari specifici.

Un'altro motivo per cui questo plugin era perfetto era che la bash scriveva in automatico i nuovi comandi su nuove righe e il plugin tail leggeva solo il contenuto di nuove linee creando una simmetria delle operazioni. Tramite il tag filter utilizzavo il plugin *record_modifier* per inserire nuovi parametri nell'evento di log, creato dal plugin precedente, in modo da aggiungergi:

- L'hostname della macchina, ipotizzo la matricola dell'utente
- Il nome della VM, in questo caso definita manualmente
- Non meno importante il timestamp che indica quando è stato chiamato questo evento di log, servirà per creare la cronologia di azioni all'interno del DB

Tramite il tag *[OUTPUT]* passavo a schermo le specifiche dell'evento in modo da verificare eventuali anomalie nei 2 passaggi precedenti. Il plugin successivo invia l'evento, e i suoi relativi parametri, a fluentd tramite il protocollo HTTP all'indirizzo: 192.168.1.3(ovvero la stessa macchina in cui risiedeva Fluentbit) alla porta 8080. Inoltre, viene definito il formato con cui questi dati vengono passati, in questo caso json; L'uri(ovvero Uniform Resource Identifier) che identifica univocamente la risorsa inviata e il tag che dovevano avere i dati all'interno dell'header HTTP.

Fluentd

Come detto precedentemente, l'utilizzo di fluentd si è reso obbligatorio per due ragioni di fondo:

1. Carenza di plugin per Fluentbit verso RabbitMQ
2. Carenza di un plugin di output che permettesse di eseguire programmi esterni con passaggio di parametri: infatti Fluentbit è utilizzato come agent specifico nella raccolta di informazioni e non per il log-forwarding, per quello esiste Fluentd.

In questo caso non ho fatto altro che creare un file di configurazione di Fluentd che permettesse le azioni sopracitate:

```
<source>
  @type http
  @id http_input
  bind 0.0.0.0
  port 8888
</source>

# OUTPUT STDOUT
#<match **>
#  @type stdout
#  @id stdout_output
#</match>

<match **>
  @type exec
  command python /home/pi/configs/sender.py
  <buffer>
    @type file
    path /home/pi/configs/buff.*
    flush_interval 1s # for debugging/checking
  </buffer>
  <format>
    @type json
  </format>
</match>
```

Inizialmente bisogna specificare che la sintassi di specifica dei file di configurazione cambia tra Fluentd e Fluentbit. Tramite il tag *¡source¿* si definisce dove Fluentd prende i dati in ingresso, in questo caso un server in ascolto su localhost alla porta 8888 e che riceve dati tramite il protocollo HTTP. In seguito, faccio il match con qualsiasi input e definisco quale plugin andrò ad utilizzare: ovvero uno di tipo *exec*. La struttura di questo plugin definisce che bisogna specificare quale comando deve essere eseguito, questa operazione vene fatta nella linea successiva con la seguente sintassi:
command ¡linguaggioDelProgramma¿ ¡binPath¿
In questo caso viene eseguito il programma, scritto in Python, che risiede al path *'/home/pi/configs/sender.py'*.
Una volta definito quale comando va eseguito, bisogna specificare il modo in cui gli

vengono passati gli eventi di input: questa operazione avviene tramite l'utilizzo di un buffer di memoria. Vanno specificati 3 parametri nella sua configurazione, quali: la sua tipologia, dove debba essere salvato e l'intervallo di aggiornamento dello stesso. Questo file viene passato come parametro al programma che dovrà essere quindi modificato di conseguenza per riuscire a leggere le informazioni al suo interno.

```
#!/usr/bin/env python
import pika
import sys

connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='192.168.1.11')
    )
channel = connection.channel()

channel.queue_declare(queue='SeedUbuntu')

input = file(sys.argv[-1])
for line in input:
    message = line.split("\n")[0]
    channel.basic_publish(exchange='', routing_key='SeedUbuntu',
        body=message)

connection.close()
```

Per poterlo eseguire, bisogna aver installato il modulo di Python che si occupa della gestione delle operazioni del client RabbitMQ, ovvero *pika*. Una volta importati i moduli necessari, si crea la connessione con il broker di RabbitMQ che risiede all'indirizzo *192.168.1.11*.
Creo un canale di comunicazione e definisco a quale coda questo debba inviare le informazioni, ad esempio alla coda *SeedUbuntu*. Eseguo un ciclo che effettua tante iterazioni quante sono le linee presenti nel file, per ognuna di queste:

1. Prendo il token relativo alla prima linea, separata dalla precedente tramite il carattere ' n'

2. Tramite il canale di comunicazione pubblico il contenuto all'interno della coda specificata.

Una volta finite le iterazioni del ciclo *for* chiudo il canale. Utilizzando questa implementazione ho riscontrato dei problemi con il formato dei messaggi che venivano

scritti all'interno delle code, inviati come stringhe di byte. La soluzione risiedeva nel convertire la tipologia di dati ricevuti all'interno del client da *String* a *byte[]*.

Capitolo 4

Conclusioni

Durante il lavoro di tesi si è riusciti a creare una infrastruttura funzionante di ricezione, controllo e salvataggio delle azioni eseguite da un training real-time, all'interno di una innovativa piattaforma di Cyber Range. Si riesce a ricevere efficacemente i messaggi dal broker di RabbitMQ; tramite l'algoritmo di matching è stato possibile confrontare le azioni ricevute con i template già presenti all'interno del database, verificare la fine del percorso di apprendimento e restituire il grafo completo di azioni a delle specifiche code di RabbitMQ.
Riguardo alle richieste tecniche, l'applicativo è stato scritto totalmente in Java implementando al suo interno funzioni di interoperabilità con il message broker RabbitMQ e con il database a grafi Neo4j. L'algoritmo di matching possiede sia funzioni che hanno permesso di implementare uno scoring system basato sui paper a disposizione, sia due diverse modalità di confronto delle azioni.
Sono state implementate funzionalità aggiuntive come Api REST per inserimento, eliminazione e richiesta di scenari di treaning ed inoltre un sistema di rilevazione di azioni anomale performate dall'utente, che fa scaturire delle procedure di riconfigurazione automatica dei cammini pubblicandoli in code specifiche, in attesa che vengano supervisionati da attori esterni. Insieme alle funzionalità sopracitate è stato possibile modificare l'algoritmo di matching in modo che comprendesse template di azioni non lineari, scenari complessi che prevedono uno scontro tra diversi team e gruppi di utenti che performano azioni di training collettive in scenari specifici.
In futuro potranno essere implementate molteplici funzioni di scoring, funzionanti anche su cammini complessi o non lineari; una volta definito il protocollo di comunicazione dei grafi di azioni, sarà possibile completare lo sviluppo delle funzioni all'interno delle Api REST; si potrebbero inoltre implementare sistemi basati sull'intelligenza artificiale per il pattern matching di azioni anomale performate dagli utenti.

Questo progetto mi ha consentito di mettere in pratica le conoscenze acquisite durante tutto il percorso di studi, mettendomi nelle condizioni di fare esperienza sulla ricerca e sullo sviluppo di applicativi complessi non affrontati durante lo studio, come ad esempio i database a grafi o i vari message broker.
Per concludere, ciò che personalmente ritengo più importante è stata la capacità di mettermi in gioco e di non arrendermi di fronte alle molteplici difficoltà incontrate durante tutto il tirocinio; essere riuscito a concludere questo progetto è motivo di grande soddisfazione e mi ha fornito ancora più motivazione nell'intraprendere un percorso ancora più difficile e complesso, come la laurea magistrale. Questa tesi, quindi, non è che l'inizio di un percorso che non si appresta a concludersi adesso.

Bibliografia

[1] Neo4j Developer, documentazione Neo4j.
https://neo4j.com/developer/

[2] Neo4j - Cypher comunity, Cypher Online Manual.
https://neo4j.com/developer/cypher/

[3] Fluentd Manual.
https://docs.fluentd.org/

[4] Fluentbit Manual.
https://docs.fluentbit.io

[5] Elasticsearch Product Manual.
https://www.elastic.co/guide/index.html

[6] THREAT-ARREST Platform's initial reference architecture.
https://www.threat-arrest.eu/html/PublicDeliverables/D1.3-THREAT-ARREST_platform_s_initial_reference_architecture.pdf

[7] THREAT-ARREST Official Website.
https://www.threat-arrest.eu

[8] A Model Driven Approach for Cyber Security Scenarios Deployment.
https://link.springer.com/chapter/10.1007/978-3-030-42051-2_8