POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Selezione di Percorsi basata su Consenso Asincrono e Distribuito in Edge Computing tramite Database a Grafo

Relatore:
Prof. Guido Marchetto

Candidato:
Sebastiano Gazzè

Firma:

Correlatore:
Prof. Flavio Esposito

Firma:

A.A 2016/2017

Indice

1 – Introduzione

Negli ultimi anni la virtualizzazione delle reti di telecomunicazioni ha portato grossi vantaggi sia sull'ambito della ricerca, permettendo riproducibilità di esperimenti o di nuove architetture a larga scala, sia alla comunità business, permettendo ai provider di infrastruttura, di contenuti o di servizi, di condividere, noleggiare o dare in affitto parte delle proprie risorse, risparmiando sui costi di gestione e sui costi fissi delle infrastrutture. La tecnologia di virtualizzazione permette infatti di condividere non solo risorse computazionali come nel classico paradigma di c*loud computing*, ma anche di garantire banda e performance di rete, di fatto convertendo comunicazioni senza connessione o *connectionless* (come la rete Internet) in comunicazioni orientate alla connessione o *connection-oriented*.

Questo cambiamento di paradigma da reti fisiche a reti virtuali ha portato non solo vantaggi ma anche complicazioni dal punto di vista della gestione della rete, dato che si deve adesso gestire un ecosistema più complesso. Per risolvere molti di questi problemi, la ricerca si è avvalsa di tecniche che permettono la programmabilità delle reti, note come *Software-Defined Networking (SDN)*. La capacità di programmare (o riprogrammare) componenti di reti che tipicamente erano eseguiti in hardware ha permesso l'avanzamento di applicazioni fino a quel momento impensabili. Tra le tante applicazioni abilitate dalle SDN, che prevede una entità (*controller*) centralizzata capace di sovrascrivere o sostituire gli algoritmi classici di indirizzamento (ad esempio OSPF), di recente interesse troviamo le applicazioni note come *edge computing*.

Il paradigma di *edge computing* potenzia tramite delega computazionale i nodi *wireless* o *wired* che si trovano ai bordi della rete, normalmente troppo deboli per poter eseguire determinati task. Spostare le capacità decisionali all'edge della rete tramite un SDN controller ha il vantaggio di una maggiore controllabilità ma lo svantaggio di non avere più un’architettura distribuita, ossia la scarsa resilienza alle

perdite di connettività o potenziali indisponibilità dei nodi della rete fisica (all'edge). Inoltre, un approccio distribuito è notoriamente più scalabile. Per questo motivo, in questa tesi si è cercato di combinare i vantaggi della tecnologia *edge computing* abilitata dalle SDN, senza rinunciare alla scalabilità delle architetture di gestione delle risorse distribuite.

In particolare, verrà presentato il progetto di un'architettura di rete in ambito *edge computing*, basata su *controller SDN multipli*, ognuno dei quali gestisce una partizione dell'edge della rete e interagisce con gli altri tramite un *algoritmo di consenso asincrono*, grazie al quale è possibile prendere decisioni di concerto in base a informazioni di rete salvate in un *database a grafo*.

Il *database a grafo* (in questo caso Neo4j), rispetto ad uno classico relazionale, ha il vantaggio di modellare perfettamente una rete di calcolatori, perché di fatto essa stessa è un grafo, e permette di fare richieste di stato più complesse in maniera più rapida sulla partizione gestita dal controllore SDN, la cui implementazione è stata effettuata con *Floodlight*. Sebbene il design del sistema permetta di eseguire il consenso asincrono (per accordarsi sul valore minimo o sul massimo) su qualsivoglia parametro di rete, l'implementazione si è concentrata sull'ottenimento di un consenso col fine di scegliere il cammino migliore in base a una funzione di costo che considera con diverso peso alcuni parametri di rete. Il controller *Floodlight* aiuta a prelevare con frequenza configurabile lo stato della rete, e degli appositi processi eseguono un consenso sul re-instradamento dei flussi di traffico in base a tali metriche.

Di seguito verranno descritti nel dettaglio i singoli componenti e verranno illustrati dei grafici ottenuti facendo girare diverse istanze dell'algoritmo di consenso in condizioni differenti, sul *testbed* a larga scala *GENI*, al fine di mostrare le sue caratteristiche principali.

2 – Background

Di seguito verrà analizzato più nel dettaglio il contesto in cui questo lavoro è stato sviluppato. Il primo sotto capitolo riguarderà quindi l'*edge computing*, il secondo il *software-define networking (SDN)* e il terzo gli *algoritmi di consenso*.

2.1 – Edge Computing

I dati prodotti all'edge della rete stanno aumentando sempre di più, per cui risulterebbe molto efficiente che essi vengano processati nell'edge stesso. Questo problema è stato già preso in considerazione da altri lavori quali quello sui *micro datacenter* [1, 2], quello sulle *cloudlet* [3] o quello sul *fog computing* [4], in quanto il *cloud computing* spesso risulta inefficiente nel processare dei dati prodotti all'edge della rete. Di seguito verrà spiegato più nel dettaglio perché l'*edge computing* è più efficiente del *cloud computing* per alcuni tipi di servizi.

2.1.1 – Perché l'edge computing

Collocare tutti task sul cloud è stato provato essere un modo efficiente di processare i dati, considerato che la sua potenza di calcolo surclassa le capacità di ciò che si trova all'edge della rete. Tuttavia, comparata alla velocità di elaborazione dei dati, che è in rapido sviluppo, la larghezza di banda è in pratica in una situazione di stallo. Con la crescente quantità di dati generati all'edge, la velocità nel trasporto dei dati sta diventando il collo di bottiglia per il paradigma di calcolo *cloud-based*. Per esempio, alcuni aerei di linea possono generare circa 5 GB/sec di dati [5], ma la larghezza di banda tra di essi e il satellite, oppure una stazione base che si trova sul suolo terrestre, non è sufficientemente elevata per la trasmissione dei dati.

Prendiamo in considerazione un veicolo autonomo, come un'auto. Essa genererà 1 GB/sec di dati e richiede un’elaborazione di essi in tempo reale affinché il veicolo prenda delle decisioni corrette [6]. Se tutti i dati necessiterebbero di essere inviati sul cloud per essere processati, i tempi di risposta sarebbero troppo elevati. Senza contare

che l'attuale larghezza di banda della rete, nonché la sua affidabilità verrebbero messe alla prova, in quanto sarebbe necessario supportare potenzialmente molti veicoli all'interno della stessa area. E' chiaro che i dati, in questo caso, hanno bisogno di essere processati all'edge della rete, in modo tale da avere tempi di risposta più brevi, una più efficiente elaborazione di essi, e una minore pressione sulla rete.

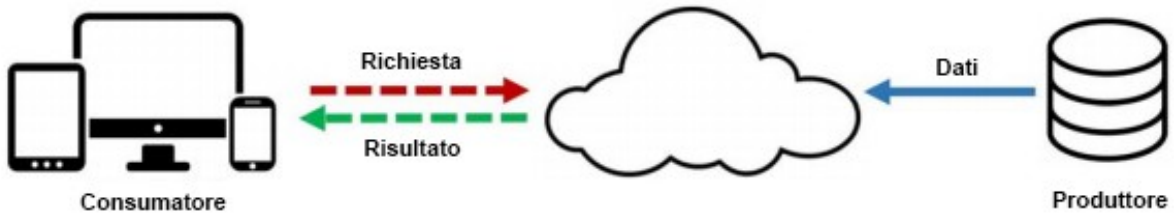


Figura 2.1.1.1

Un'altra ragione per andare verso l'*edge computing* è che quasi tutti i tipi di dispositivi elettrici, si prevede che entreranno a far parte della cosiddetta *IoT (Internet of Things)*, e giocheranno sia il ruolo del *produttore* che quello del *consumatore*. Parliamo di cose come i sensori della qualità dell'aria, le barre a LED e ci saranno persino dei forni a microonde connessi a Internet. Ormai non ci sono dubbi sul fatto che il numero di *things* all'edge della rete è destinato a superare il miliardo entro pochi anni. Risulta quindi chiaro che la quantità di dati prodotta da essi sarà mastodontica, e il classico modello di *cloud computing* non sarà più in grado di gestirli tutti in modo efficiente. Ciò significa che gran parte dei data prodotti nel contesto *IoT* non sarà più trasmessi verso il cloud, ma verrà invece consumata nell'edge della rete stesso. La *Figura 2.1.1.1* mostra la struttura del classico *cloud computing*. I produttori generano dati e li trasmettono verso il cloud, mentre i consumatori inviano al cloud richieste per consumarli. Le due linee tratteggiate rossa e verde, indicano rispettivamente una richiesta di consumo di dati e il risultato che il cloud invia al consumatore che ne aveva fatto richiesta in precedenza. Ad ogni modo, questa struttura non è sufficiente in ambito *IoT*. Innanzitutto, la quantità di dati

all'edge è troppo elevata, il che porterebbe a un enorme e non necessaria larghezza di banda nonché ad un uso spropositato delle risorse di calcolo. Poi c'è da dire che i requisiti in termini di protezione della privacy, sarebbero un grosso ostacolo all'uso del *cloud computing*. Infine, molti dei nodi in ambito *IoT*, presentano dei vincoli dal punto di vista energetico, e il modulo di comunicazione wireless in genere consuma davvero tanta energia, quindi l'*offloading* di parte dei task sull'edge potrebbe essere molto più efficiente dal punto di vista del consumo energetico.

Anche il cambiamento del ruolo da solo *consumatore* a *consumatore/produttore* sta creando delle necessità che spingono verso l'*edge computing*. Nel paradigma di *cloud computing* infatti, i dispositivi finali all'edge tipicamente giocano il ruolo dei consumatori di dati, basti pensare a quando uno smartphone viene usato per guardare un video su YouTube, ma resta il fatto che al giorno d'oggi, la gente produce anche dei dati con i loro dispositivi mobili. Per esempio, è davvero normale che la gente scatti delle foto o registri dei video con lo smartphone e che poi li condivida attraverso servizi basati sul cloud come Instagram, YouTube etc [7]. Inoltre, ogni minuto gli utenti di questi servizi condividono una quantità di foto e video inimmaginabile fino a qualche anno fa ed ogni foto/video non è raro che occupi molto in termini di memoria, e quindi necessiti anche di una grossa quantità di banda per farne l'upload. In questi casi, il file dovrebbe essere adattato in termini di risoluzione prima di essere caricato sul cloud. Un altro esempio potrebbe essere quello relativo ai *dispositivi medici indossabili*. Dato che i dati raccolti da tali dispositivi all'edge della rete sono tipicamente privati, processarli nell'edge stesso potrebbe aiutare a proteggere la privacy dell'utente meglio rispetto a una soluzione in cui essi vengono caricati sul cloud.

2.1.2 – Cosa è l'edge computing

Con *edge computing* ci si riferisce all'abilitazione di tecnologie che permettono che l'elaborazione dei dati sia eseguita all'edge della rete, sui dati in arrivo per conto di

servizi cloud-based, e su quelli in uscita per conto di *servizi IoT*. Si definisce *"edge"* qualsiasi risorsa di rete o di elaborazione che si trova nel percorso che va dalla sorgente dei dati ai data center che stanno sul cloud. Per esempio, uno smartphone può essere visto come l'edge tra le *body things* e il cloud, così come un gateway in una smart home è l'edge tra le *home things* e il cloud. Insomma l'*edge computing* vuole che l'elaborazione dei dati dovrebbe avvenire in prossimità della loro sorgente. Per alcuni *edge computing* è interscambiabile con *fog computing* [8], ma nel primo caso il focus è più spostato sulle *things* mentre nel secondo sulle infrastrutture. Si prevede che l'*edge computing* possa avere nei prossimi anni un grosso impatto sulla nostra società, al pari di quello che interessa il classico *cloud computing*.

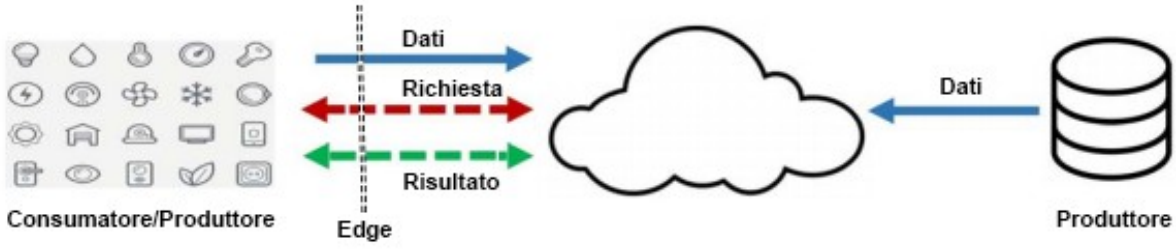


Figura 2.1.2.1

La *Figura 2.1.2.1* illustra gli stream di elaborazione in entrambi i sensi tipici del paradigma dell'*edge computing*. Qui le *things* non sono solo consumatori, ma anche produttori. All'edge, le *things* non possono solamente richiedere servizi e contenuti dal cloud, ma hanno anche il compito di eseguire dei task per conto di esso. L'edge può eseguire *computing offloading*, *data storage*, *caching and processing*, ma anche distribuire richieste a consegnare servizi dal cloud all'utente. E' chiaro che a questo punto l'edge stesso necessita di essere ben progettato per rispettare in modo efficiente i requisiti in servizi quali affidabilità, sicurezza, e protezione della privacy.

2.1.3 – Benefici dell'edge computing

Con l'*edge computing* si vuole, come detto, spostare l'elaborazione dei dati vicino

alla loro sorgente. Questo ha diversi benefici se comparato al tradizionale paradigma di *cloud-based computing*. Qui di seguito sono esposti diversi risultati che dimostrano i potenziali benefici del nuovo paradigma. Dei ricercatori hanno creato una piattaforma *proof-of-concept* su cui far girare applicazioni di riconoscimento facciale [9], e i tempi di risposta sono scesi da 900 a 169 ms, spostando l'elaborazione dal cloud verso l'edge. Altri [10] hanno usato le *cloudlets* per fare l'*offloading* dei calcoli nel campo dell'assistenza cognitiva indossabile, e i risultati mostrano che il miglioramento dei tempi di risposta è compreso tra 80 e 200 ms. Inoltre, il consumo di energia potrebbe essere ridotto dal 30% al 40% in questo modo. Infine c'è il prototipo di *Clonecloud* in [11] che sembra poter ridurre di 20 volte il tempo di esecuzione e l'energia per le applicazioni testate.

2.2 – SDN e controller multipli

Ricerche recenti in ambito *Software-Defined Networking* (SDN) si avvalgono di *controller multipli e distribuiti* per ragioni di di *scalabilità* e *affidabilità*. L'uso di controller multipli permette alla rete di scalare senza introdurre *colli di bottiglia* o *single point of failure*. Inoltre fornisce ad essa le proprietà di *ridondanza* e *tolleranza ai guasti*. Questi *controller SDN multipli* hanno bisogno di comunicare tra di loro per poter sincronizzare le informazioni contenute nei loro stati. Per questi motivi, essi sono soggetti a problemi simili a quelli che affliggono i database distribuiti. Uno dei più grossi è quello relativo al trade-off tra la *consistenza* e la *disponibilità* in presenza di partizionamenti della rete, problema che è stato definito da Eric Brewer, come la *congettura CAP* dove C sta per Consistenza, A per Availability (Disponibilità), e P per Partizionamento. Questa congettura afferma che, in presenza di partizionamenti della rete, un sistema distribuito è costretto a scegliere tra la consistenza dei dati e la disponibilità del sistema. I sistemi che preferiscono la consistenza alla disponibilità sono detti fortemente consistenti, e tra di loro ce ne sono alcuni che sono in grado di cambiare il loro comportamento andando di fatto a variare il loro grado di consistenza. In ambito SDN, il livello di consistenza delle informazioni di stato

scambiate tra i controller distribuiti può avere effetti negativi sulle performance delle applicazioni di rete, in base a quali indicatori si sta considerando. C'è una moltitudine di applicazioni SDN, ognuna avente indicatori di performance differenti, e come tale alcune di esse possono tollerare, per esempio, l'inconsistenza delle informazioni di stato a vantaggio di una maggiore disponibilità.

2.3 – Algoritmi di consenso

Trovare un accordo tra i processi in un sistema distribuito è un requisito fondamentale per un'ampia gamma di applicazioni. Molte forme di coordinazione richiedono ai processi di scambiare informazioni per negoziare tra di loro e per, infine, raggiungere un'interpretazione o un accordo comune, prima di intraprendere azioni specifiche dell'applicazione. Un esempio classico è quello di una decisione di *commit* nei sistemi database, dove i processi decidono collettivamente di eseguire la funzione *commit* o la funzione *abort* nei confronti di una transazione alla quale partecipano. Nel seguito verrà approfondito il problema della fattibilità di progettare algoritmi per raggiungere un accordo secondo vari modelli di sistema e di fallimento. Ma prima è meglio soffermarsi sul alcune definizioni.

Modelli di fallimento

Tra gli n processi nel sistema, al massimo f processi possono essere difettosi. Un processo difettoso può agire in qualsiasi modo permesso dal modello di malfunzionamento supposto. Nel modello *fail-stop*, un processo potrebbe bloccarsi nel bel mezzo di uno step, che potrebbe essere l'esecuzione di un'operazione locale o l'elaborazione di un messaggio in un evento di invio o ricezione. In particolare, potrebbe inviare il messaggio solo a un sottoinsieme della destinazione selezionata prima di andare in tilt. Nel modello di fallimento bizantino, un processo potrebbe agire arbitrariamente. La scelta del modello di fallimento determina la fattibilità e la complessità della risoluzione del consenso.

Comunicazione sincrona / asincrona

Se un processo soggetto a fallimento decide di inviare un messaggio al processo P_i ma fallisce, poi P_i non potrà rilevare il mancato arrivo del messaggio in un sistema asincrono, perché questo scenario è indistinguibile da uno scenario in cui il messaggio ci mette molto tempo ad arrivare. A causa di questo si ha l'impossibilità di raggiungere un accordo nei sistemi asincroni in qualsiasi modello di fallimento. In un sistema sincrono invece lo scenario nel quale il messaggio non è stato inviato può essere riconosciuto dal destinatario previsto, alla fine della fase. Il destinatario previsto può gestire il mancato arrivo di un messaggio atteso assumendo quello di un messaggio contenente alcuni dati di default, e poi procedendo con la fase successiva dell'algoritmo.

Connettività di rete

Il sistema possiede piena connettività logica, ovvero, ogni processo può comunicare con qualsiasi altro con uno scambio di messaggi diretto.

Identificazione del mittente

Un processo che riceve un messaggio conosce sempre l'identità del processo che lo ha inviato. Questa supposizione è importante, perché persino in caso di comportamento bizantino, anche se il *payload* utile del messaggio può contenere dati fittizi mandati da mittenti maligni, i sottostanti protocolli di livello di rete possono rivelare la vera identità del processo di invio. Quando più messaggi sono attesi dallo stesso mittente in un singolo round, implicitamente supponiamo un algoritmo di *scheduling* che invia questi messaggi in sotto-round, in modo che ogni messaggio inviato in quel round possa essere univocamente identificato.

Affidabilità del canale

I canali sono affidabili e solo i processi possono fallire (a seconda di uno dei vari modelli di fallimento). Questa è un'ipotesi semplificativa ma nonostante ciò il problema dell'accordo è o irrisolvibile o risolvibile in modo complesso.

Messaggi autenticati vs. non autenticati

Nelle pagine successive verranno considerati solo dei messaggi non autenticati. Con messaggi non autenticati, quando un processo malizioso trasmette un messaggio ad altri processi, esso può falsificare il messaggio e sostenere che è stato ricevuto da un altro processo, e può anche manomettere i contenuti di un messaggio ricevuto prima di trasmetterlo. Quando un processo riceve un messaggio, non ha alcun modo di verificare la sua autenticità. Un messaggio non autenticato viene anche detto messaggio *orale o* messaggio *non firmato*. Usando l'autenticazione tramite tecniche come *firme digitali*, è più facile risolvere il problema dell'accordo perché, se qualche processo falsifica il messaggio o manomette i contenuti di un messaggio ricevuto prima di trasmetterlo, il destinatario può rilevare la falsificazione o manomissione. Pertanto dei processi maliziosi possono procurare meno danni.

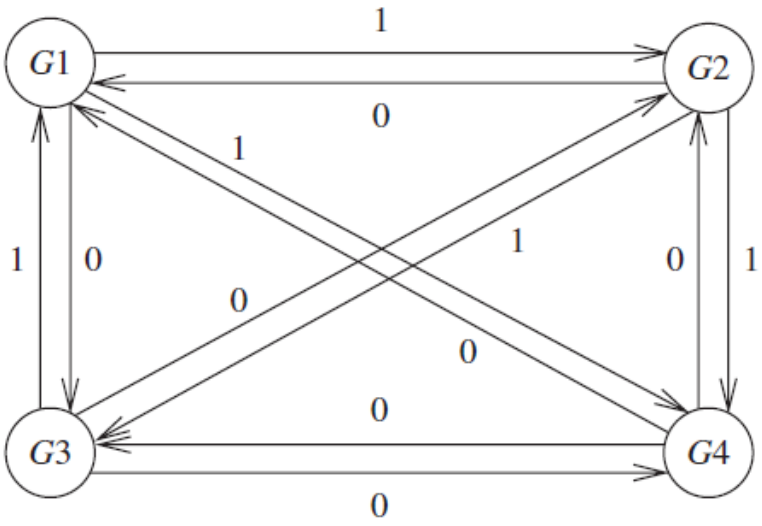


Figura 2.3.1

Variabile dell’accordo

La variabile dell’accordo può essere booleana o multi-valore, e non ha bisogno di essere un numero intero. Considerando una variabile booleana, è più semplice analizzare un algoritmo e allo stesso tempo i risultati non vengono inficiati.

Consideriamo adesso, la difficoltà nell’ottenere un accordo utilizzando il seguente esempio, che è ispirato alle lunghe guerre combattute dall’Impero bizantino nel medioevo.

Quattro accampamenti dell’esercito attaccante, ognuno comandato da un generale, sono accampati intorno alla fortezza di Bisanzio (l’attuale Istanbul). Potevano riuscire nell’attacco solo se attaccavano simultaneamente. Perciò, dovevano raggiungere un accordo sull’ora dell’attacco. L’unico modo che avevano per poter comunicare era inviando messaggi tra di loro. I messaggeri modellano i messaggi. Un sistema asincrono è modellato da messaggeri che impiegano un tempo illimitato per viaggiare tra due accampamenti. Un messaggio perso è modellato da un messaggero catturato dal nemico. Un processo bizantino è modellato da un generale traditore. Il traditore proverà a sabotare il meccanismo di raggiunta dell’accordo, dando informazioni fuorvianti agli altri generali. Per esempio, un traditore potrebbe informare un generale di attaccare alle 10 del mattino e informare un altro generale di attaccare a mezzogiorno. O potrebbe non mandare affatto un messaggio a qualche generale. Analogamente, potrebbe manomettere i messaggi che riceve da altri generali, prima di trasmettere questi messaggi. Un semplice esempio di comportamento bizantino è mostrato nella *Figura 3.2.1*. Sono mostrati quattro generali; una decisione consensuale su un valore booleano deve essere raggiunta. I vari generali stanno trasmettendo agli altri generali valori della variabile di decisione potenzialmente fuorvianti, il che genera confusione e chiaramente complica non di poco il raggiungimento di un accordo.

3 – Dettagli implementativi

Di seguito verranno descritti l’architettura e i singolo componenti.

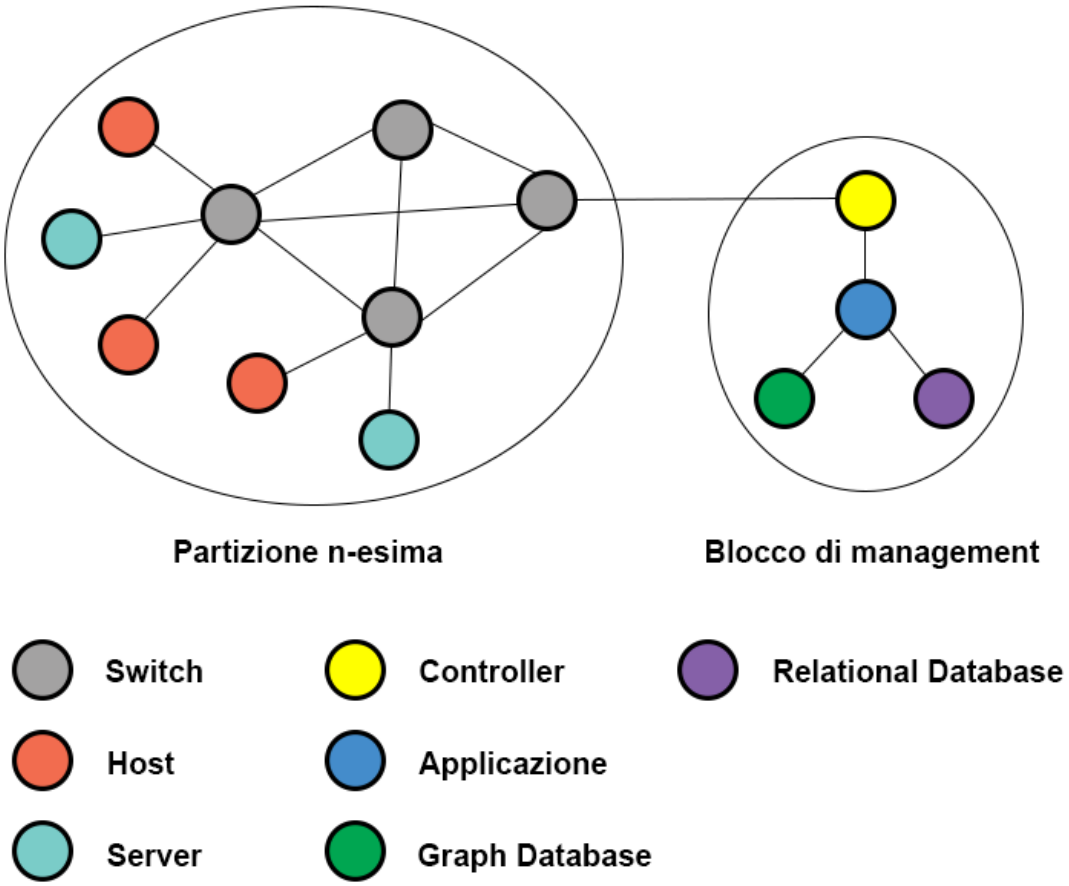


Figura 3.1.1

Come si può vedere nella *Figura 3.1.1*, l’edge della rete è diviso in partizioni ognuna delle quali contiene diversi switch a cui sono collegati degli host e dei server, e questi switch fanno capo a un controller (nello specifico Floodlight). Esso comunica con l’applicazione, la quale è collegata ad un database relazionale (nello specifico MySQL) e ad uno a grafo (nello specifico Neo4j). Nell *Figura 3.1.2* invece è mostrato il modo in cui è pensato l’edge della rete.

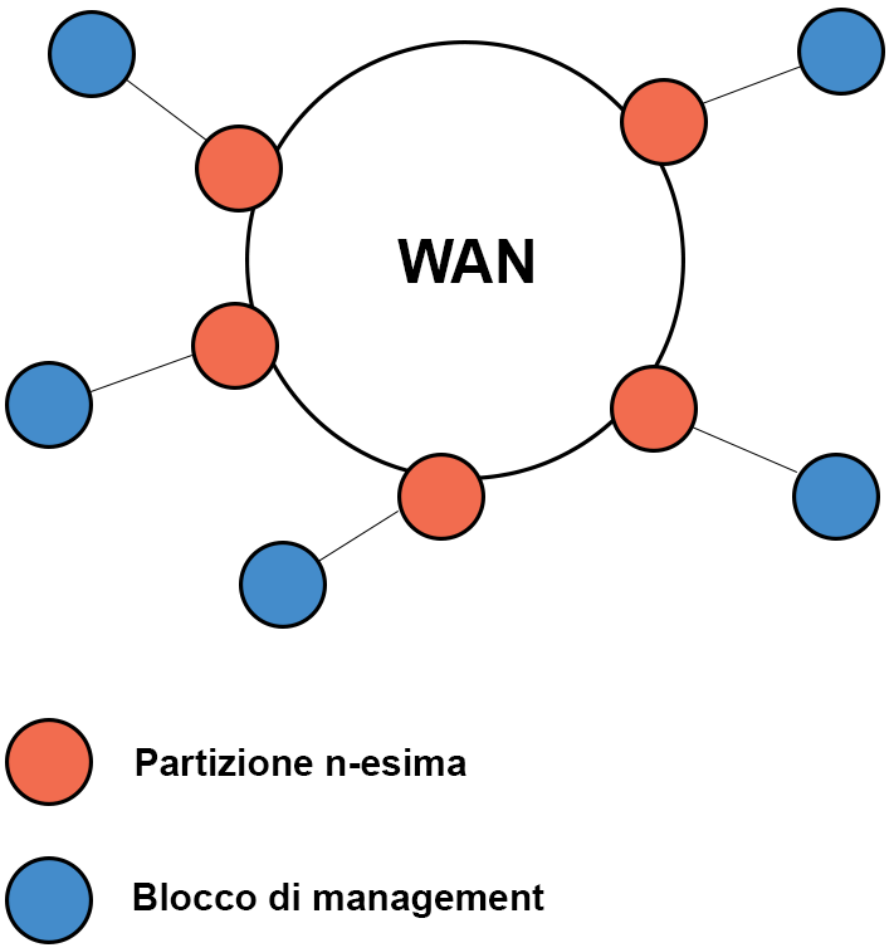


Figura 3.1.2

3.1 – Funzionamento generale

Negli esperimenti fatti ognuna delle partizioni indicate è stata simulata con Mininet, ed è gestita, come detto, da un Floodlight controller. Si è scelto esso e non ONOS perchè quest'ultimo è un vero e proprio sistema operativo che presenta tante cose che non ci servivano. L'alternativa sarebbe stata Ryu, ma ho optato per Floodlight per via della mia maggiore esperienza nella programmazione in Java rispetto a quella in Python.

Floodlight, tra le altre cose, espone un'interfaccia REST dalla quale è possibile

recuperare dei file JSON che descrivono lo stato della rete gestita dal controllore, e tramite i quali è quindi possibile ricostruirla all'interno del database a grafo Neo4j. Questo viene quindi fatto periodicamente, e ogni volta viene calcolato un hash della rete (con SHA256) per confrontare quella appena ricostruita dagli ultimi JSON ricevuti con quelle salvate in Neo4j. Nel momento in cui si trova un riscontro, non viene creato un nuovo grafo che rappresenta la rete, ma si vanno solo a salvare sui link i nuovi valori relativi alle latenze e alla banda (o qualsiasi altro parametro di rete che può essere recuperato dall'applicazione chiedendo a Floodlight o in altri modi).

Dato che basta un link temporaneamente down o un nuovo host per far cambiare l'hash della rete, ci si aspetta che entro poco tempo il database contenga tanti grafi, e le prestazioni nel reperire le informazioni potrebbero quindi diminuire drasticamente. Per questo, l'applicazione dispone di un thread che con cadenza regolare (non troppo spinta) va a controllare le date di modifica relative a tutti i grafi, e sposta quelli che non vengono aggiornati da tanto (in base ad una certa soglia che si può impostare tramite un file di configurazione) all'interno del database relazionale, serializzandoli in formato JSON (nello specifico MySQL). Tornando a prima, sarebbe meglio dire quindi, che quando viene costruito un nuovo grafo a partire dai JSON e se ne calcola l'hash, se esso è relativo ad un grafo salvato in Neo4j, si vanno a salvare i nuovi valori in esso, altrimenti, prima di crearne uno nuovo, si va prima a controllare se in realtà esso è relativo ad un grafo che era stato in precedenza spostato sul database relazionale. In questo caso esso viene ripristinato all'interno di Neo4j e poi vengono salvati i nuovi dati, se invece tale hash non è relativo nemmeno ad un un grafo presente in MySQL, allora viene creato un nuovo grafo in Neo4j.

Ogni partizione gestita da un controllore è del tipo della *Figura 3.1.3.* I nodi in verde sono degli host/server, mentre quelli in azzurro degli switch. Si è pensato di dividere gli switch in edge switch e backbone switch. Ai primi è possibile collegare degli host/server, mentre ai secondi no, essi sono collegabili solo con altri switch, che siano edge switch o backbone switch. All'atto pratico, per fare gli esperimenti, si è

identificato in Mininet gli edge switch grazie al loro ID che inizia con “ee:ee”. Quello dei backbone switch inizia invece con “bb:bb”. Poi ci sono quelli di accesso alla cloud che iniziano con “cc:cc”.

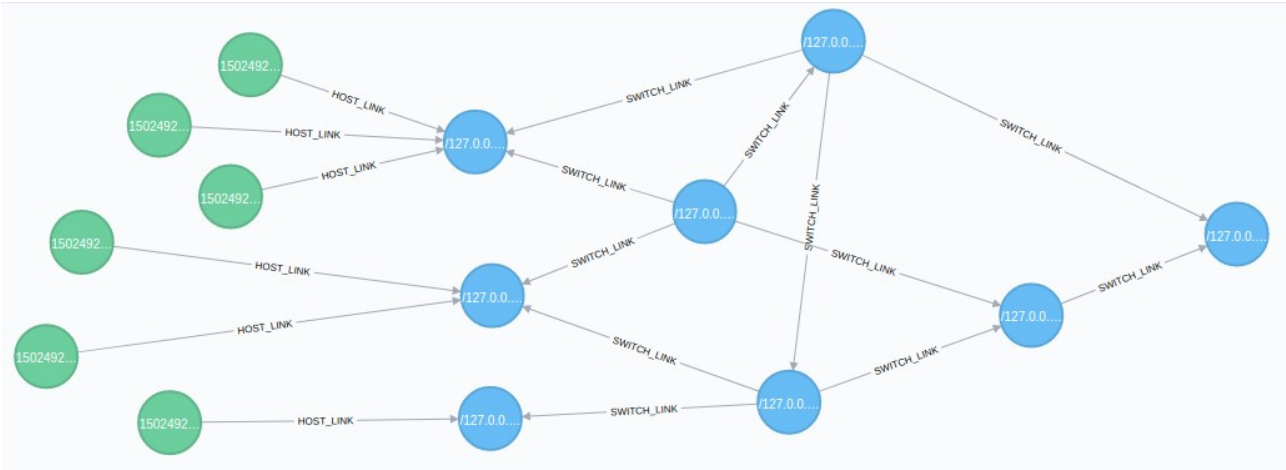


Figura 3.1.3

Il *Frammento 3.1.1* mostra come viene calcolato il percorso migliore che ha un nodo per arrivare allo switch di accesso alla cloud. I primi due parametri sono il peso che viene dato ai valori di latenza salvati sui link (latencyWeight) e ai valori relativi alla banda (bandwidthWeight), mentre il terzo parametro si riferisce a quanti valori deve prendere in considerazione la funzione che calcola il costo. Tale costo è la prima cosa che viene calcolata, come si può vedere. Poi in sostanza si va a prendere l’hash del grafo più recente che è presente nel database, perché ce ne possono essere diversi, e si va a cercare lo switch di accesso alla cloud, che abbiamo assunto essercene uno soltanto. A quel punto si vanno a trovare, tramite l’algoritmo di Dijkstra, che Neo4j implementa nativamente, tutti i percorsi che partono da degli switch edge e finiscono sullo switch di accesso alla cloud. Di questi viene selezionato solo quello che, in base alla funzione di costo scelta (vedi *Frammento 3.1.2*), risulta il migliore.

In pratica il peso di questo percorso viene usato come valore iniziale dal thread che implementa il consenso per la partizione in questione. Possiamo immaginare quindi

un nuovo nodo che si connette ad una certa partizione ed inizia a funzionare normalmente, ma subito dopo, questo evento fa scattare la richiesta di consenso da parte dell'applicazione che gestisce la partizione, la quale vuole capire se in realtà ci sarebbe un altro percorso migliore per lui se fosse collegato ad un'altra partizione. Ci sarà quindi un thread che chiederà all'NMS (vedi capitolo relativo all'algoritmo di consenso) di iniziare un'istanza dell'algoritmo, e alla fine si possono verificare tre scenari fondamentalmente, che illustrerò qui di seguito.

```
public synchronized WeightedPath getDijkstraPath(double latencyWeight, double
bandwidthWeight, int numValues)
{
    WeightedPath path = null;

    try (Transaction tx = graphDB.beginTx())
    {
        DijkstraCost cost = new DijkstraCost(latencyWeight, bandwidthWeight,
        numValues);
        PathFinder<WeightedPath> finder =
            GraphAlgoFactory.dijkstra(PathExpanders.forTypeAndDirection(
            RelationshipType.withName("SWITCH_LINK"), Direction.BOTH), cost, 1);

        //Getting the netHash of the most recent graph
        String netHash = graphDB.getAllRelationships()
            .stream()
            .filter(rel -> rel.isType(RelationshipType.withName("SWITCH_LINK")))
            .map(rel -> rel.getProperty("lastTimeHashCalculated") + "#" +
                rel.getProperty("netHash"))
            .distinct()
            .max((pair1, pair2) -> Long.compare(Long.parseLong(pair1.split("#")
                [0]), Long.parseLong(pair2.split("#")[0])))
            .map(maxPair -> maxPair.split("#")[1])
            .orElse("");

        //It's assumed to exist only one cloud point access per partition
        Node destNode = graphDB.findNodes(Label.label("Switch"), "dpid",
            "cc:cc:00:00:00:00:00:01")
            .stream()
            .filter(node -> node.getRelationships(
                RelationshipType.withName("SWITCH_LINK"),
                    Direction.BOTH).iterator().next().getProperty("netHash")
                        .toString().equals(netHash))
            .findFirst()
            .orElse(null);

        if (destNode != null)
        {
            path = graphDB.findNodes(Label.label("Switch"))
                .stream()
                .filter(node ->
                    node.getProperty("dpid").toString().startsWith("ee:ee"))
                .filter(node -> node.getRelationships(
                    RelationshipType.withName("SWITCH_LINK"),
```

```
                        Direction.BOTH).iterator().next().getProperty("netHash")
                            .toString().equals(netHash))
                .map(nodeEdgeSwtich -> finder.findAllPaths(nodeEdgeSwtich,
                    destNode))
                .flatMap(paths -> Iterables.asList(paths).stream())
                .filter(p -> Iterables.count(p.nodes(), n ->
                    n.getProperty("dpid").toString().startsWith("ee:ee")) == 1)
                .min((path1, path2) -> Double.compare(path1.weight(),
                    path2.weight()))
                .orElse(null);

            tx.success();
        }
        else {
            tx.failure();
        }
    }

    return path;
}
```

Frammento 3.1.1

Nel primo, si ha che il percorso migliore risulta essere quello che il nodo stava seguendo di default, quindi per caso si era attaccato all'edge switch che garantisce, dati alla mano, il percorso migliore verso il cloud.
Nel secondo, si ha che il percorso migliore risulta trovarsi nella stessa partizione di partenza, ma è diverso da quello di default, quindi in questo caso serve dire al Floodlight controller, interagendo con esso tramite la sua northbound interface, di spostare l'host da un edge switch verso un altro.
Nel terzo caso invece, viene fuori che il percorso migliore si trova in un'altra partizione, quindi serve dire al Floodlight controller che gestisce la partizione di partenza, di staccare l'host dall'edge switch a cui si era attaccato, e poi dire a quello che gestisce la partizione finale di attaccare l'host all'edge switch prescelto.

```
public class DijkstraCost implements CostEvaluator<Double>,
CostAccumulator<Double>
{
    private final double latencyWeight;
    private final double bandwidthWeight;
    private final int numValues;

    public DijkstraCost(double latencyWeight, double bandwidthWeight, int
        numValues)
    {
```

```
        this.latencyWeight = latencyWeight;
        this.bandwidthWeight = bandwidthWeight;
        this.numValues = numValues;
    }

    public double getLatencyWeight() {
        return latencyWeight;
    }

    public double getBandwidthWeight() {
        return bandwidthWeight;
    }

    public int getNumValues() {
        return numValues;
    }

    @Override
    public Double addCosts(Double val1, Double val2) {
        return val1 + val2;
    }


    @Override
    public Double getCost(Relationship rel, Direction dir)
    {
        int[] latencies = (int[]) rel.getProperty("latencies");
        int fromLat = latencies.length > numValues ? latencies.length -
            numValues : 0;
        double avgLat = Arrays.stream(Arrays.copyOfRange(latencies, fromLat,
            latencies.length)).average().orElse(0);

        double[] bwTxStart = (double[])
            rel.getProperty("bwTxPercentagesFullStartSwi");
        int fromTxStart = bwTxStart.length > numValues ? bwTxStart.length -
            numValues : 0;
        double avgBwTxStart = Arrays.stream(Arrays.copyOfRange(bwTxStart,
            fromTxStart, bwTxStart.length)).average().orElse(0);

        double[] bwRxStart = (double[])
            rel.getProperty("bwRxPercentagesFullStartSwi");
        int fromRxStart = bwRxStart.length > numValues ? bwRxStart.length -
            numValues : 0;
        double avgBwRxStart = Arrays.stream(Arrays.copyOfRange(bwRxStart,
            fromRxStart, bwRxStart.length)).average().orElse(0);

        double[] bwTxEnd = (double[])
            rel.getProperty("bwTxPercentagesFullEndSwi");
        int fromTxEnd = bwTxEnd.length > numValues ? bwTxEnd.length -
            numValues : 0;
        double avgBwTxEnd = Arrays.stream(Arrays.copyOfRange(bwTxEnd, fromTxEnd,
            bwTxEnd.length)).average().orElse(0);

        double[] bwRxEnd = (double[])
            rel.getProperty("bwRxPercentagesFullEndSwi");
        int fromRxEnd = bwRxEnd.length > numValues ? bwRxEnd.length -
            numValues : 0;
        double avgBwRxEnd = Arrays.stream(Arrays.copyOfRange(bwRxEnd, fromRxEnd,
            bwRxEnd.length)).average().orElse(0);

        //Since they are values fullness, the bottleneck is the bigger value
        double avgBwDirOneFull = avgBwTxStart > avgBwRxEnd ? avgBwTxStart :
```

```
            avgBwRxEnd;
        double avgBwDirTwoFull = avgBwRxStart > avgBwTxEnd ? avgBwRxStart :
            avgBwTxEnd;
        double avgBandwidth = (avgBwDirOneFull + avgBwDirTwoFull) / 2;

        //A multiplier to strongly penalize links with very high values of full
        bandwidth
        int multiplier = 1;

        if (avgBandwidth > 95) {
            multiplier = 200;
        }
        else if (avgBandwidth > 90) {
            multiplier = 150;
        }
        else if (avgBandwidth > 80) {
            multiplier = 100;
        }
        else if (avgBandwidth > 70) {
            multiplier = 75;
        }
        else if (avgBandwidth > 50) {
            multiplier = 50;
        }
        else if (avgBandwidth > 25) {
            multiplier = 10;
        }

        return latencyWeight * avgLat + bandwidthWeight * multiplier *
            avgBandwidth;
    }
}
```

Frammento 3.1.2

3.2 – L'algoritmo di consenso

Il pezzo cruciale di questo lavoro è sicuramente l'algoritmo di consenso, grazie al quale è possibile far sì che le varie partizioni prendano una decisione di concerto, a prescindere dallo scenario d'uso. Gli attori presenti sono fondamentalmente due: i peer e il *Network Management System* (che da qui in avanti chiamerò *NMS*). Quando uno dei peer necessita che venga eseguita un'istanza dell'algoritmo, comunica la sua volontà all'*NMS*, il quale risponde inviando a lui e agli altri peer un oggetto che contiene i parametri di configurazione necessari a eseguire l'algoritmo. Questo è suddiviso in diverse fasi, che di seguito andrò ad illustrare nello specifico, ognuna delle quali va ad aggiornare il suo stato finchè non si verifica una condizione di terminazione, in coincidenza della quale si spera sia stato raggiunto il consenso. Esso

inoltre è suddiviso in round, in ognuno dei quali la componente server resta in attesa di messaggi dagli altri peer, mentre la componente client si occupa di fare il broadcast dello stato verso di essi. E' infine possibile impostare l'algoritmo affinchè il consenso venga fatto sul valore minimo o su quello massimo.

3.2.1 – Fase di Preparazione

Quando l'algoritmo inizia a girare, il suo stato si trova nella fase di *Preparazione*, e va da sè che al primo round, il valore che viene inviato in broadcast sia quello iniziale (anche se tecnicamente c'è, la seppur remota possibilità che, dato che il server viene attivato prima del client, il peer riceva un messaggio da parte di un altro peer che dichiara di conoscere formalmente quale è il minimo/massimo, e in tal caso ciò che fa è aggiornare il proprio stato, fidandosi di esso, e inoltrare quindi fin da subito tale valore per far convergere prima l'algoritmo). Ad ogni round, il server rimane in attesa di messaggi finchè scade il timeout o finchè non viene ricevuto un numero di messaggi pari ad una percentuale del numero di peer che partecipano al consenso (entrambi i valori sono impostabili attraverso un file di configurazione che l'*NMS* invia a ciascun peer, come spiegato nell'apposita sezione), mentre il client termina non appena ha completato il broadcast. Nello stato è presente una struttura dati in cui viene memorizzato il miglior messaggio ricevuto da ciascun peer (a prescindere che sia stato ricevuto direttamente da quest'ultimo o da qualcun'altro che l'ha semplicemente inoltrato), e alla fine di ogni round, prima di riattivare server e client, essa viene scandita per capire se ci sono le possibilità di passare alla fase successiva.

```
public void preparationPhase()
{
    Message decMsg = getDecidedProposalMsg();

    if (decMsg != null)
    {
        hasDecided = true;
        phase = EnumPhase.TERMINATION;
        proposal = decMsg.getProposal();
    }
    else
    {
        bestMsgPerProcess
```

```
            .values()
            .stream()
            .filter(msg -> msg.getProposal().compareTo(proposal) < 0)
            .map(msg -> msg.getProposal())
            .min((prop1, prop2) -> prop1.compareTo(prop2))
            //If not present means proposal already contains the current minimum
            .ifPresent(minProp -> {
                proposal = minProp;
            });

        //5 total peers -> needed 2 peers (with "this" are 3 > numProcs / 2)
        //6 total peers -> needed 3 peers (with "this" are 4 > numProcs / 2)
        if (getNumPeersAtLeastOneMessage() >= (int) (cnfg.getNumProcs() / 2)) {
            phase = EnumPhase.DECISION;
        }
    }
}
```

Frammento 3.2.1.1

Facendo riferimento allo *Frammento 3.2.1.1*, per prima cosa viene controllato se c'è un peer che ha già preso una decisione su un certo valore, e se così, eventuali altri peer che hanno deciso, hanno scelto quello stesso valore. Questo perchè, come spiegherò più avanti, per decidere c'è bisogno della maggioranza semplice. In caso di riscontro positivo, il peer si allinea a quanto deciso da tale maggioranza, e si va direttamente alla fase di *terminazione*, senza passare per ovvi motivi da quella di *decisione*. Se invece il riscontro è negativo, si va a scandire la struttura dati e si aggiorna lo stato con il valore minore ricevuto fino a quel momento. In altre parole, se un peer si accorge che un altro peer sta inviando in broadcast un valore migliore (minore o maggiore in base al caso) del suo, dal round successivo inizierà ad inoltrarlo lui stesso per permettere all'algoritmo di convergere. Fatto questo, si va a controllare se ci sono le condizioni per poter passare alla fase di *decisione*. In sostanza, dato che, affinchè si prenda una decisione su un valore è necessario che ci sia una maggioranza semplice a supportarlo, è inutile passare alla fase di *decisione* se ancora non si conosce il valore proposto da almeno la metà più uno dei peer partecipanti al consenso. Quindi in questa fase non c'è ancora bisogno che una tale quantità di peer sia d'accordo su un certo valore, ma basta aver ricevuto da essi almeno un messaggio.

3.2.2 – Fase di Decisione

La seconda fase è quella di *decisione*, che viene eseguita per la prima volta nello stesso round in cui è stata eseguita per l'ultima volta quella di *preparazione*. Questo ovviamente per avere dei vantaggi in termini di prestazioni. Essendo arrivato l'algoritmo a questo punto, è certo che almeno la metà più uno dei peer ha espresso la propria preferenza, quindi ciò che è necessario fare è controllare se in questo insieme di peer c'è una maggioranza semplice che concorda sul medesimo valore.

```
public void decisionPhase()
{
    //This method also considers the proposal of this peer
    getNumMsgsByProposal()
        .entrySet()
        .stream()
        //5 total peers -> needed value = 3
        //6 total peers -> needed value = 4
        .filter(entry -> entry.getValue() >= (int) (cnfg.getNumProcs() / 2 + 1))
        .map(entry -> entry.getKey())
        //Formally only one proposal can pass the filter
        .findFirst()
        .ifPresent(majorityProp -> {
            hasDecided = true;
            //The most broadcasted proposal could be worse than
            //the current one but I choose it though
            proposal = majorityProp;
            phase = EnumPhase.TERMINATION;
        });
}
```

Frammento 3.2.2.1

Facendo riferimento all *Frammento 3.2.2.1*, il metodo *getNumMsgsByProposal()* crea una mappa (key, value) in cui la key è uno dei valori proposti dai vari peer, e il value il numero di peer che l'hanno proposto. Ne segue che, ammesso che ci sia un valore supportato da una maggioranza semplice, non può essercene un altro, per cui basta prendere il primo che soddisfa tale requisito e aggiornare lo stato, decretando alla fine anche l'inizio della fase di *terminazione*, la quale può anche non esistere in realtà, perchè nel file di configurazione è possibile settare il numero massimo di round che può durare, e chiaramente si può fissare quel numero a 0. Da notare che si potrebbe anche finire per accordarsi su un valore che non è il migliore in assoluto fra quelli

proposti. Si prenda in considerazione la *Figura 3.2.2.1* per capire meglio questo concetto.

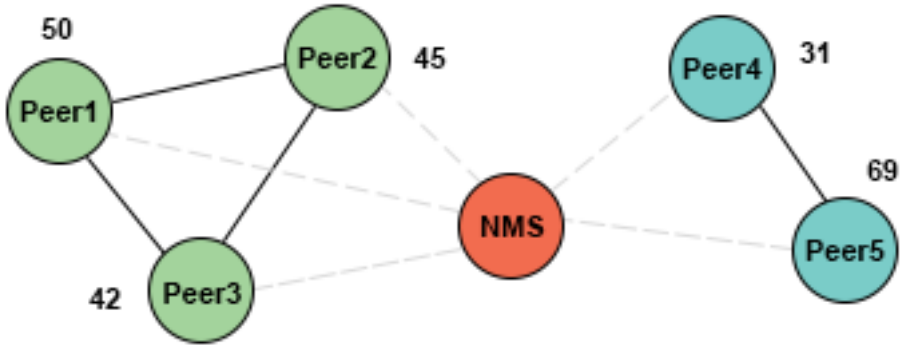


Figura 3.2.2.1

Le linee tratteggiate tra l'*NMS* e i vari peer, indicano che esso all'inizio è stato in grado di comunicare con ciascuno di loro, per cui ognuno di essi ha ricevuto il file di configurazione in cui, tra le altre cose, c'è anche la lista degli indirizzi IP dei 5 peer che partecipano al consenso. Ne segue che ogni peer cerca di scambiare messaggi con tutti gli altri 4 come già detto. In questo caso però, per via di qualche problema di rete, si finisce per avere due partizioni, una contenente i peer 1, 2, 3 e l'altra i restanti peer 4, 5. Supponendo che si fa il consenso sul valore minimo, si ha che i primi tre si accordano sul valore 42 e gli altri due sul 31 (sebbene questi rimangano bloccati nella fase di *preparazione*), e poichè tre peer rappresentano una maggioranza semplice, l'algoritmo termina convergendo sul 42, nonostante esso non sia il minimo assoluto.

3.3 – Fase di Terminazione

Non appena si è presa una decisione, inizia come detto la fase di *terminazione*, la cui durata in round è fissata a priori nel file di configurazione. In tale fase, per la prima metà dei round i peer continuano a inviare in broadcast il loro stato, mentre per la seconda metà viene attivato solo il server, questo per far sì che i buffer si svuotino e

concludere in modo più pulito. C'è da dire che in questa fase, i peer inviano dei messaggi che contengono, oltre alle normali informazioni, anche una lista dei valori iniziali relativa ai peer facenti parte della maggioranza semplice che ha permesso di decidere sul valore contenuto nel messaggio. Questo ovviamente serve ad aiutare l'algoritmo a convergere spingendolo verso la fase di *conoscenza*.

3.4 – Fase di Conoscenza

La fase di *conoscenza* risulta svincolata dalla catena che lega le precedenti tre e l'algoritmo può passare ad essa in qualsiasi momento, giacchè essa rappresenta solo la situazione in cui un certo *peerX* ha ricevuto almeno un messaggio da tutti gli altri, e poichè ogni messaggio contiene, oltre al valore scelto, anche quello con cui il peer che lo invia aveva iniziato, *peerX* può stabilire con certezza quale sia il valore migliore in assoluto (minimo o massimo che sia). Va da sè che questa situazione può verificarsi durante una qualsiasi delle fasi precedenti, e a prescindere dal round, anzi non è per niente raro che l'algoritmo converga già al primo round. Non appena *peerX* entra in questa fase, esegue un ultimo broadcast per informare gli altri peer della sua "scoperta", mentre i peer che ricevono tale messaggio lo inoltrano a tutti (anch'essi per un'ultima volta), e aggiornano ovviamente il loro stato. Alla fine, sia *peerX* che gli altri, sono pronti per l'ultima fase, quella di *conferma*. Ma prima di entrare più nel dettaglio è bene valutare meglio come l'algoritmo può evolversi a partire dalla situazione riassunta nella *Figura 3.2.2.1* e illustrata in precedenza. Di seguito andrò a descrivere più nel dettaglio i vari possibili scenari.

3.4.1 – Scenario 1

Supponendo che nel file di configurazione non sia specificato che la fase di *terminazione* duri 0 round, e supponendo che i problemi di rete che avevano portato ad avere le due partizioni descritte non vengano risolti, nè entro lo scadere del timeout generale nè entro la fine della suddetta fase, come già anticipato, quello che

si ha è che la partizione formata da tre peer riesce comunque a prendere una decisione, quindi essi passano alla fase successiva o dopo aver eseguito l'ultimo round della fase di *terminazione* o allo scadere del timeout generale, quale che sia dei due l'evento che si verifica per primo. Ecco lo stato finale di ciascuno dei tre peer.

Fase : Terminazione	**Peer1**	**Peer2**	**Peer3**	**Peer4**	**Peer5**
Proposta iniziale:	50	45	42	-	-
Proposta finale:	42	42	42	-	-
Sorgente Prop. Fin.:	*Peer3*	*Peer3*	*Peer3*	-	-

Tabella 3.4.1.1

Per quanto riguarda invece la partizione con i restanti due peer, si ha che essi rimangono bloccati nella fase di *preparazione*, in quanto sanno bene di non essere i soli partecipanti al consenso e a giudicare dalla dimensione della lista dei peer contenuta nel file di configurazione, sanno anche che una maggioranza semplice è composta da almeno 3 peer in questo caso. Ne segue che riescono a uscire da questa stasi, solo quando scade il timeout generale. Ecco lo stato finale di questi due peer.

Fase: Preparazione	**Peer1**	**Peer2**	**Peer3**	**Peer4**	**Peer5**
Proposta iniziale:	-	-	-	31	69
Proposta finale:	-	-	-	31	31
Sorgente Prop. Fin.:	-	-	-	*Peer4*	*Peer4*

Tabella 3.4.1.2

3.4.2 – Scenario 2

Stavolta continuiamo a considerare l'ipotesi che la fase di *terminazione* non duri 0 round, ma a differenza di prima, il problema di rete che aveva creato le due partizioni viene risolto. In particolare, come si può vedere nella *Figura 3.4.2.1*, ritorna ad

esserci connettività tra *Peer2* e *Peer4*.

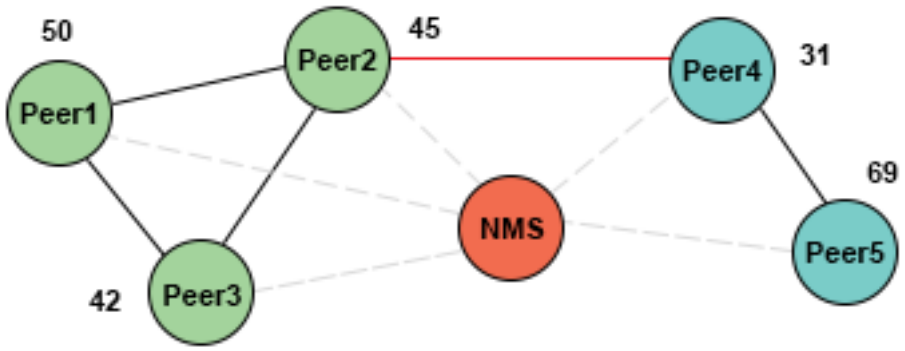


Figura 3.4.2.1

Non appena possibile quindi, *Peer2* riceverà messaggi da *Peer4* e viceversa. Consideriamo il caso in cui l'evento che si verifica prima è il primo (se si verifica per secondo cambia davvero poco), cioè che *Peer2* riceve un messaggio da *Peer4*. Tale messaggio contiene il valore 31, che è migliore di quello su cui si erano accordati i tre peer in precedenza, ma nonostante ciò tale scelta rimane (in quanto c'è la fase di *decisione* è alle spalle), e lo stato di *Peer2* diventa il seguente, mentre quello di *Peer1* e *Peer3* rimane uguale a quello della *Tabella 3.4.1.1*.

Fase: Terminazione	**Peer1**	**Peer2**	**Peer3**	**Peer4**	**Peer5**
Proposta iniziale:	50	45	42	31	-
Proposta finale:	42	42	42	31	-
Sorgente Prop. Fin.:	*Peer3*	*Peer3*	*Peer3*	*Peer4*	-

Tabella 3.4.2.1

Subito dopo *Peer4* riceve un messaggio da *Peer2*, e poichè quest'ultimo si trova nella fase di *terminazione*, tale messaggio contiene, oltre alla coppia *42 – Peer3*, anche

l'insieme delle proposte iniziali di tutti i peer che fanno parte della maggioranza semplice che ha decretato la proposta finale, quindi Φ = *(50 – Peer1, 45 – Peer2)*. Alla fine lo stato di *Peer4* sarà quindi il seguente.

Fase: Conoscenza	**Peer1**	**Peer2**	**Peer3**	**Peer4**	**Peer5**
Proposta iniziale:	50	45	42	31	69
Proposta finale:	42	42	42	31	31
Sorgente Prop. Fin.:	*Peer3*	*Peer3*	*Peer3*	*Peer4*	*Peer4*

Tabella 3.4.2.2

A questo punto *Peer4* conosce tutte le proposte iniziali di tutti i peer che hanno partecipato al consenso, ed è quindi in grado di stabilire con certezza quale sia la migliore (in questo caso quella col valore minimo), per cui dalla fase di *preparazione* passa direttamente a quella di *conoscenza*, e come detto, esegue un ultimo broadcast per permettere anche agli altri di convergere sulla coppia *31 – Peer4*. Al round successivo quindi anche *Peer2* e *Peer5* entreranno nella fase di *conoscenza*, e in quello ancora dopo pure *Peer1* e *Peer3*. Il problema è che potrebbe non esserci abbastanza tempo, o perchè il numero di round della fase di *terminazione* non è sufficiente, o perchè scatta il timeout generale nel mentre. La situazione più grave è quella in cui uno dei due eventi si verifica subito dopo che *Peer2* ha ricevuto il suddetto messaggio da *Peer4* con la fase fissata a *conoscenza*, mentre *Peer5* non lo ha ancora ricevuto. Ciò che avviene è che si va a rimettere in discussione ciò che era stato deciso dalla maggioranza semplice senza riuscire ad allineare in tempo tutti i peer facenti parte di essa alla nuova e definitiva proposta finale e senza riuscire a crearne un'altra con altri peer che si trovano tutti d'accordo sul minimo assoluto. Lo stato di *Peer2* e *Peer4* sarà quindi quello della *Tabella 3.4.2.2*, mentre quello di *Peer1* e *Peer3* resterà quello della *Tabella 3.4.1.1* e quello di *Peer5* quello della *Tabella 3.4.1.2*. A prima vista potrebbe sembrare che in realtà adesso ci sono due peer ancora fermi al valore precedente (*Peer1* e *Peer3* che sostengono ancora il 42), ma tre

(quindi la metà più uno) che sono d'accordo sul nuovo (*Peer2*, *Peer4* e *Peer5* che sostengono il 31), ma il problema sta nel fatto che *Peer5* non ha fatto in tempo ad uscire dalla fase di *preparazione*, e questo rappresenta un problema della fase di *conferma*.

3.5 – Fase di Conferma

Considerato quanto appena detto, diventa chiaro il motivo per cui c'è bisogno di avere una fase di *conferma*. Pensando ad una normale situazione in cui magari già al primo round ogni peer riceve un messaggio da tutti gli altri e si passa subito in massa alla fase di *conoscenza*, ciò potrebbe sembrare poco utile, se non addirittura una perdita di tempo. Infatti ci si aspetta che quando un peer chiede conferma, tutti gli altri la diano, e questo anche nel caso lo scambio di messaggi termini con la situazione descritta nella *Figura 3.2.2.1* (in cui saranno due i peer a dare conferma al massimo). Ovviamente in tutto ciò c'è sempre da considerare che un peer possa crashare o avere problemi di connettività anche durante la fase di *conferma*, e in quel caso potrebbe non riuscire a confermare o farlo troppo tardi, quando il timeout relativo a quest'ultima fase sarebbe ormai scaduto. Il discorso diventa più delicato quando si verifica ciò che ho descritto alla fine del paragrafo precedente. Andando a guardare più da vicino il *Frammento 3.5.1*, si può notare innanzitutto che la prima cosa che si fa è andare a controllare se si è arrivati alla fase di *conferma* prima dello scadere del timeout generale. In quel caso il peer si addormenta per il tempo restante, in modo tale che, dato che i clock di tutti i peer sono sincronizzati con quello dell'*NMS* (il quale ha calcolato e inviato i vari timeout all'interno del file di configurazione), al risveglio è certo che nessuno di loro stia ancora inviando/ricevendo messaggi, per cui non c'è più possibilità che il loro stato muti. A quel punto inizia la fase di *conferma* vera e propria, in cui i peer che si trovano nelle fasi di *decisione*, *terminazione* o *conoscenza*, e che si rendono conto che la proposta su cui è andato a convergere il consenso era quella che avevano proposto loro inizialmente, chiedono conferma agli altri per capire se a quel punto c'è ancora una maggioranza semplice a supportarla,

maggioranza che sarebbe ormai definitiva. Inutile sottolineare che solo un peer al massimo può ottenere conferma da una maggioranza semplice, e in quel caso esso invierà il risultato del consenso all'*NMS*.

```
Instant preConfirmation = Instant.now();
Instant limit = cnfg.getConsensusOverTimestamp();

//If the consensus has terminated before the scheduled moment, it's needed
//to attend in such way to be sure no one peer (clocks are synchronized)
//will change its status. At that point it's possible ask for confirmations
//(algorithm is partition tolerant)
if (preConfirmation.isBefore(limit))
{
    long toSleep = Duration.between(preConfirmation, limit).toMillis();
    Thread.sleep(toSleep);
}

PeerConfServer server = new PeerConfServer(sockConf, status, execServer, cnfg);
Future<?> futureServer = execConfirm.submit(server);
Proposal currentProposal = status.getProposal();
Proposal initialProposal = status.getInitialProposal();

//The node that had proposed the agreed value ask for confirm to other peers
if (status.getHasDecided() && currentProposal.equals(initialProposal))
{
    status.setPhase(EnumPhase.CONFIRMATION);
    Message confMsg = new Message(-1, status);
    PeerConfClient client = new PeerConfClient(confMsg, execClient, cnfg);
    Future<Integer> futureClient = execConfirm.submit(client);
    int numConfirms = futureClient.get();

    //5 total peers -> needed confirmations = 2
    //6 total peers -> needed confirmations = 3
    if (numConfirms >= (int) (cnfg.getNumProcs() / 2))
    {
        logger.info("Proposal has been confirmed by a majority of peers");
        logger.info("Communicating the result to the NMS...");
        String[] addressNMS = cnfg.getNmsAddress().split(":");
        int port = Integer.parseUnsignedInt(addressNMS[1]);
        InetSocketAddress address = new InetSocketAddress(addressNMS[0], port);
        socketNMS = new Socket();
        socketNMS.connect(address, cnfg.getSocketTimeoutConnect());
        socketNMS.shutdownInput();
        oos = new ObjectOutputStream(socketNMS.getOutputStream());
        //Not used confMsg because it's needed an up-to-date timestamp
        Message resultMsg = new Message(-1, status);
        oos.writeObject(resultMsg);
    }
    else {
        logger.info("Proposal NOT confirmed by a majority of the peers");
    }
}

futureServer.get();
```

Frammento 3.5.1

Dunque, tornando a quanto detto alla fine del precedente paragrafo, *Peer3* chiederà conferma per il valore 42 e *Peer4* per il 31, ma nessuno dei due riceverà conferme da un numero sufficiente di peer. *Peer3* solo da *Peer1* e *Peer4* solo da *Peer2*, perchè *Peer5*, sebbene alla fine inviasse messaggi contenenti il 31, non ha mai potuto prendere una decisione su di esso. Insomma il consenso non viene raggiunto, sebbene ad un certo punto c'era una maggioranza semplice di peer che concordavano sul 42. Per evitare una situazione del genere si potrebbe impedire che, dopo aver preso una decisione, un peer possa scegliere un'altra proposta (diversa dalla precedente) solo perchè ci sarebbero le condizioni per passare alla fase di *conoscenza*. In quel caso quindi, la fase di *terminazione* servirebbe solo a rendere più probabile che alla fine tutti i peer facenti parte della maggioranza abbiano effettivamente deciso.

3.6 – Terminazione dell’algoritmo

La terminazione dell’algoritmo non è un argomento banale come possa sembrare, perché esso garantisce solo che ad un certo punto il consenso verrà raggiunto, ma essendoci di mezzo la rete, può capitare di tutto. Se per esempio un peer si rende conto, ad un certo punto, che tutti gli altri hanno preso una decisione, non può per questo fermare la propria esecuzione, perché non può essere certo che tutti gli altri siano a conoscenza di ciò che egli ha deciso. Infatti, sebbene sia raro, può capitare che il peer in questione riesca a ricevere benissimo ma abbia problemi a inviare, quindi nel caso estremo non ci sarebbe nessuno informato sulle sue decisioni. Ma allora che si fa? Lo si fa andare avanti all’infinito sperando che ad un certo punto riesca a fare il *broadcast* del suo stato? Chiaramente una cosa del genere non è pensabile. Inoltre, anche nel caso tutti gli altri abbiano ricevuto i suoi messaggi, nessun peer può essere certo che qualsiasi altro peer sappia che è stato raggiunto il consenso. E’ un problema che non può essere risolto in modo formale con un numero finito di messaggi scambiati, per cui nessun peer può terminare in modo safe. Si potrebbe informare il nodo centrale (l’NMS), delle singole decisioni dei peer, e fare

in modo che questi ultimi terminino solo dopo che esso ha comunicato loro che possono farlo (perché si è reso conto che un numero sufficiente di loro ha deciso), ma sorgerebbero altri problemi. Per esempio, potrebbe essere lui quello irraggiungibile o che non riesce ad inviare messaggi. Si è quindi trovata una soluzione di compromesso in cui un peer dopo essere entrato nella fase di *terminazione* (qui un peer invia un messaggio che contiene oltre al valore deciso, anche la lista delle proposte iniziali dei peer che formano la maggioranza a cui appartiene il peer in questione, in modo tale da incrementare le probabilità che gli altri peer entrino nella fase di *conoscenza* e terminino nel migliore dei modi), può restarci per al massimo un certo numero di round e inoltre invia il suo stato in *broadcast* solo durante la prima metà di essi. Inoltre c'è anche un timeout generale, quindi se i round della fase di terminazione sono tali da sforarlo, l'algoritmo passa avanti comunque. Ultimo ma non meno importante, quando un peer entra nella fase di conoscenza, fa un ultimo *broadcast* per aiutare gli altri a convergere, e procede. A quel punto i peer che sono usciti prima del timeout fissato per il consenso, attendono fino a che scada, in modo tale che da lì in avanti nessuno di loro può veder variare il proprio stato e si fa la fase di *conferma* con questa certezza. Poi si è già visto cosa accade. Insomma si è quindi trovato un compromesso non potendo attendere all'infinito, e giocando con i parametri di configurazione si ottengono buoni risultati.

3.7 – Il file di configurazione

Come già accennato, l'NMS invia ai peer un oggetto che contiene diversi parametri che lui ha recuperato dal file di configurazione. Adesso darò una spiegazione su di essi, almeno dei più importanti.

NMS settings
nmsAddress=192.86.139.74:6666
nmsSocketTimeout=50
peerResponseOffsetTimeout=10000

Testing settings

failureProbability=15

failureProbabilityConfirmPhase=20

areFailuresDefinitive=true

resolvedFailureProbability=0

General settings

socketTimeoutConnect=500

broadcastOffsetTimeout=2500

consensusOffsetTimeout=3500

confirmationOffsetTimeout=8000

maxRoundsTerminationPhase=2

Server settings

socketTimeout=200

serverTimeoutMultiplier=75

serverTimeoutMultiplierTermination=40

handlersServerTimeout=250

maxRoundMsgsPercentage=100

Client settings

maxAttemptsSendMsg=3

delayBetweenSendAttempts=30

handlersClientTimeout=300

Il *broadcastOffsetTimeout* rappresenta l'offset a partire dal quale l'NMS calcola il timestamp oltre il quale i peer devono smettere di inoltrare il loro stato. Prima è stato spiegato che in realtà i peer smettono di fare *broadcast* dopo metà dei round impostati per la fase di *terminazione*, ed è vero, ma c'è anche questo parametro. Se si vuole ignorarlo basta settarlo con un valore superiore al *consensusOffsetTimeout.*

Il *consensusOffsetTimeout* rappresenta l'offset a partire dal quale l'*NMS* calcola il timestamp oltre il quale il consenso viene fermato e si passa alla fase di *conferma*.

Il *confirmationOffsetTimeout* rappresenta l'offset a partire dal quale l'*NMS* calcola il timestamp oltre il quale la fase di *conferma* è considerata finita, e quindi, anche qualora ci fossero dei thread che stanno ancora ricevendo messaggi di conferma, essi verrebbero ignorati perché alla ricezione si fa un confronto sull'ora corrente del peer in questione che riceve e sul timestamp contenuto nel messaggio stesso.

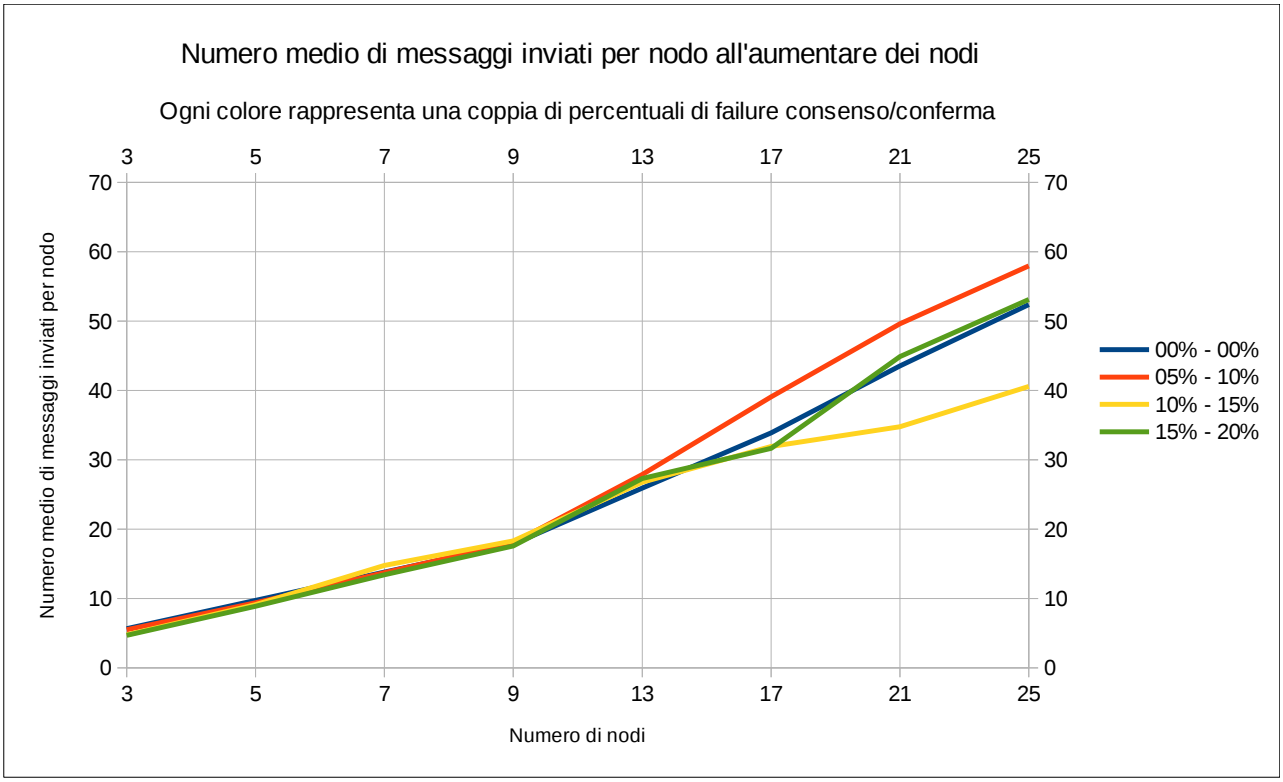
Infine a partire dal *peerResponseOffsetTimeout*, l'*NMS* si calcola il timestamp oltre il quale smette di attendere il risultato del consenso. Da notare che se per allora non ha ricevuto alcuna risposta, non può discriminare il mancato consenso dal fatto che il messaggio contenente il risultato non è riuscito ad arrivare in tempo per problemi di rete.

Tutti questi timestamp vengono calcolati dall'*NMS* a partire da un istante di tempo assoluto che sceglie lui stesso. Questo perché gli orologi di sistema dei vari peer (e dell'*NMS*) sono da considerare sincronizzati tra di loro, nel senso che recuperano l'ora dallo stesso server. Da notare che, se per qualche problema di rete, l'oggetto di configurazione dovesse arrivare con molto ritardo ad un certo peer, questo potrebbe potenzialmente nemmeno iniziare a inviare messaggi o persino a riceverne, perché al momento dell'arrivo dell'oggetto si sono già superati quei timeout rispettivamente.

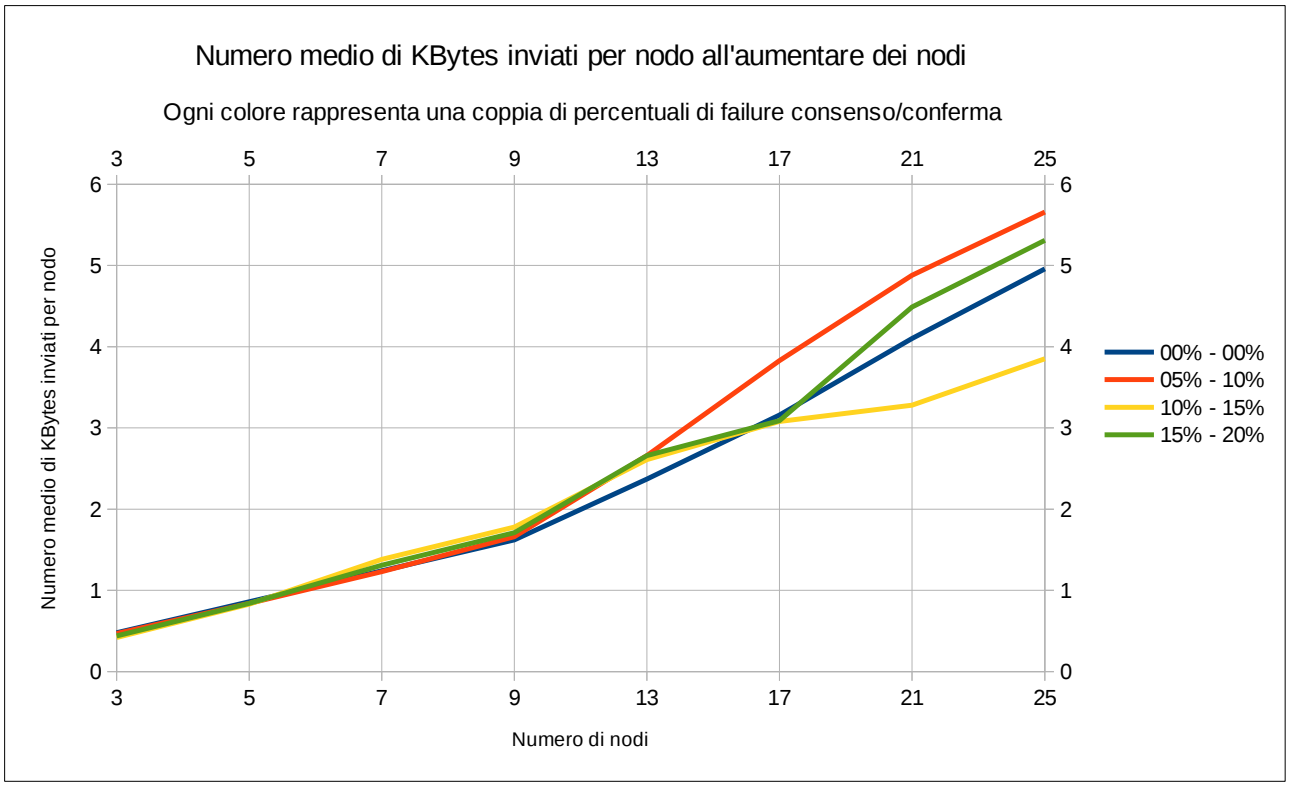
4 – Risultati sperimentali

Di seguito verranno mostrati alcuni grafici ottenuti facendo girare diverse istanze dell'algoritmo sul testbed GENI sotto condizioni diverse. La scritta 05%-10% (e simili) che comparirà sui grafici, sta a significare che si è impostata una probabilità di fallimento dei nodi pari al 5% nelle normali fasi dell'algoritmo e al 10% nella sola fase di conferma.

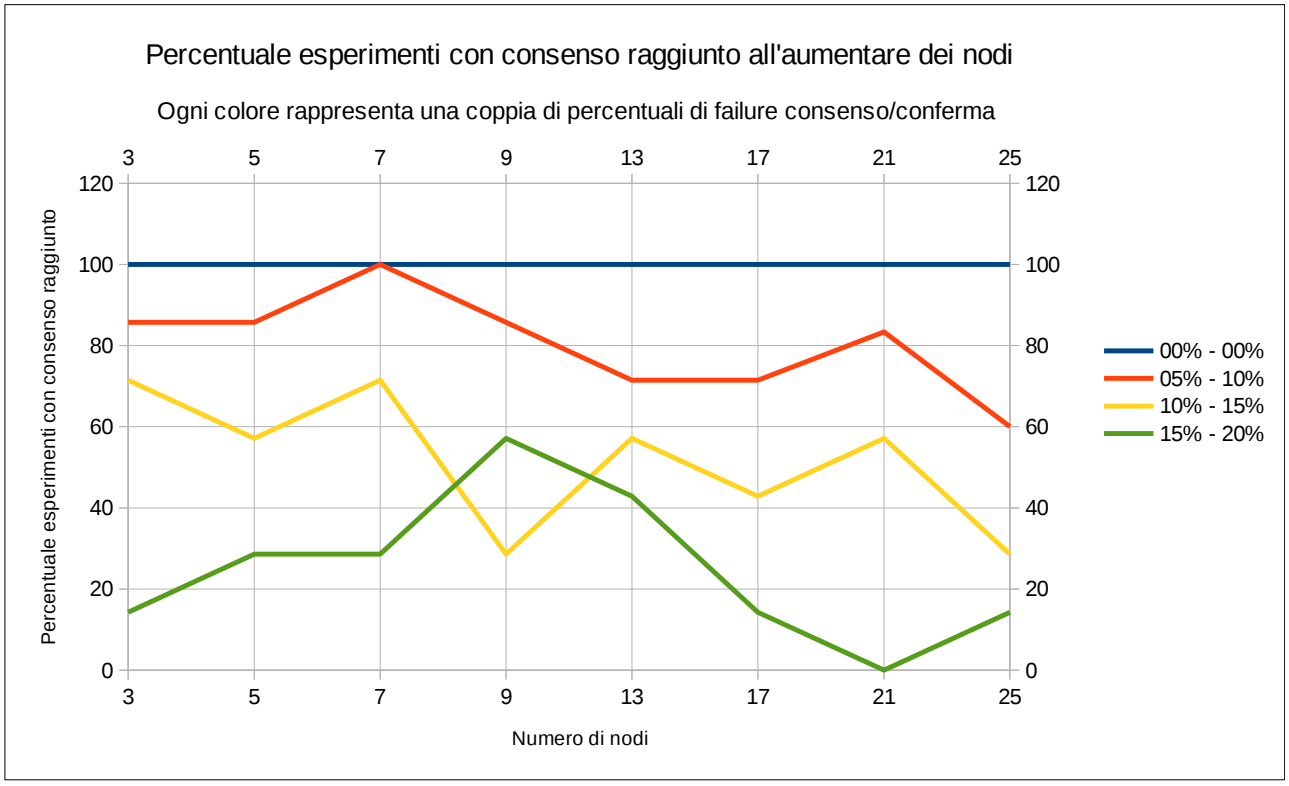
4.1 – Esperimento 1



4.2 – Esperimento 2



4.3 – Esperimento 3



5 – Bibliografia

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, “The cost of a cloud: Research problems in data center networks,” ACM SIGCOMM Comput. Commun. Rev., vol. 39, no. 1, pp. 68–73, 2008.

[2] E. Cuervo et al., “MAUI: Making smartphones last longer with code offload,” in Proc. 8th Int. Conf. Mobile Syst. Appl. Services, San Francisco, CA, USA, 2010, pp. 49–62.

[3] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” IEEE Pervasive Comput., vol. 8, no. 4, pp. 14–23, Oct./Dec. 2009.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the Internet of things,” in Proc. 1st Edition MCC Workshop Mobile Cloud Comput., Helsinki, Finland, 2012, pp. 13–16

[5] Boeing 787s to Create Half a Terabyte of Data Per Flight, Says Virgin Atlantic. Accessed on Dec. 7, 2016. [Online]. Available: https://datafloq.com/read/self-driving-carscreate-2-petabytes-data-annually/172

[6] Self-Driving Cars Will Create 2 Petabytes of Data, What are the Big Data Opportunities for the Car Industry? Accessed on Dec. 7, 2016. [Online]. Available: http://www.computerworlduk.com/news/data/boeing-787screate-half-terabyte-of-data-per-flight-says-virgin-atlantic-3433595/

[7] Data Never Sleeps 2.0. Accessed on Dec. 7, 2016. [Online]. Available: https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/

[8] (2016). OpenFog Architecture Overview. OpenFog Consortium Architecture Working Group. Accessed on Dec. 7, 2016. [Online]. Available: http://www.openfogconsortium.org/wp-content/uploads/OpenFog-Architecture-Overview-WP-2-2016.pdf

[9] S. Yi, Z. Hao, Z. Qin, and Q. Li, “Fog computing: Platform and applications,” in Proc. 3rd IEEE Workshop Hot Topics Web Syst. Technol. (HotWeb), Washington, DC, USA, 2015, pp. 73–78.

[10] K. Ha et al., “Towards wearable cognitive assistance,” in Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services, Bretton Woods, NH, USA, 2014, pp. 68–81.

[11] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, “CloneCloud: Elastic execution between mobile device and cloud,” in Proc. 6th Conf. Comput. Syst., Salzburg, Austria, 2011, pp. 301–314.

[12] L. Lamport, “Paxos Made Simple” in ACM SIGACT News, vol. 32, no. 4, pp. 18–25, 2001.

[13] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Léon, J. M., “Megastore: Providing Scalable, Highly Available Storage for Interactive Services” in Proceedings of the Conference on Innovative Data Systems Research, pp. 223-234, 2011.

[14] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: an engineering perspective”, in Proceedings of the twenty-sixth annual ACM Symposium on Principles of Distributed Computing, 2007, pp. 398–407

[15] M. Burrows, “The Chubby Lock Service for Loosely-Coupled Distributed Systems”, in Proceedings of OSDI 2006.