ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE DI UNA INFRASTRUTTURA BASATA SU KAFKA E STORM PER IL MOBILE CLOUD COMPUTING

Relazione finale in: Programmazione a Oggetti

Presentata da:
LORENZO PELLEGRINI

Relatore:
Prof. MIRKO VIROLI

Co-relatore:
Ing. PIETRO BRUNETTI

ANNO ACCADEMICO 2014–2015
SESSIONE II

PAROLE CHIAVE

Mobile computing

Cloud mediated

Closest source

Storm

Kafka

Alla mia famiglia

Indice

Introduzione

Lo scopo dell'elaborato di tesi è l'analisi, progettazione e sviluppo di un prototipo di una infrastruttura cloud in grado di gestire un grande flusso di eventi generati da dispositivi mobili. Questi utilizzano informazioni come la posizione assunta e il valore dei sensori locali di cui possono essere equipaggiati al fine di realizzare il proprio funzionamento. Le informazioni così ottenute vengono trasmesse in modo da ottenere una rete di device in grado di acquisire autonomamente informazioni sull'ambiente ed auto–organizzarsi.

La costruzione di tale struttura si colloca in un più ampio ambito di ricerca che punta a integrare metodi per la comunicazione ravvicinata con il cloud al fine di permettere la comunicazione tra dispositivi vicini in qualsiasi situazione che si potrebbe presentare in una situazione reale.

A definire le specifiche della infrastruttura e quindi a impersonare il ruolo di committente è stato il relatore, Prof. Mirko Viroli, mentre lo sviluppo è stato portato avanti da me e dal correlatore, Ing. Pietro Brunetti. Visti gli studi precedenti riguardanti il cloud computing nell'area dei sistemi complessi distribuiti, Brunetti ha dato il maggiore contributo nella fase di analisi del problema e di progettazione mentre la parte riguardante la effettiva gestione degli eventi, le computazioni in cloud e lo storage dei dati è stata maggiormente affrontata da me.

In particolare mi sono occupato dello studio e della implementazione del backend computazionale, basato sulla tecnologia Apache Storm, della componente di storage dei dati, basata su Neo4j, e della costruzione di un pannello di visualizzazione basato su AJAX e Linkurious.

A questo va aggiunto lo studio su Apache Kafka, utilizzato come tecnologia per realizzare la comunicazione asincrona ad alte performance tra le componenti.

Si è reso necessario costruire un simulatore al fine di condurre i test per verificare il funzionamento della infrastruttura prototipale e per saggiarne l'effettiva scalabilità, considerato il potenziale numero di dispositivi da sostenere che può andare dalle decine alle migliaia.

La sfida più importante riguarda la gestione della vicinanza tra dispositivi e la possibilità di scalare la computazione su più macchine. Per questo motivo è stato necessario far uso di tecnologie per l'esecuzione delle operazioni di memorizzazione, calcolo e trasmissione dei dati in grado di essere eseguite su un cluster e garantire una accettabile fault–tolerancy. Da questo punto di vista i lavori che hanno portato alla costruzione della infrastruttura sono risultati essere un'ottima occasione per prendere familiarità con tecnologie prima sconosciute.

Quasi tutte le tecnologie utilizzate fanno parte dell'ecosistema Apache e, come esposto all'interno della tesi, stanno ricevendo una grande attenzione da importanti realtà proprio in questo periodo, specialmente Apache Storm e Kafka.

Il software prodotto per la costruzione della infrastruttura è completamente sviluppato in Java a cui si aggiunge la componente web di visualizzazione sviluppata in Javascript.

La tesi è così suddivisa.

Nella prima parte viene descritto l'ambito in cui si colloca l'elaborato, eseguite brevi panoramiche delle tecnologie prese in considerazione per la comunicazione ravvicinata, esposte le problematiche relative al loro uso, introdotte le motivazioni e le tecnologie utilizzate per permettere la comunicazioni cloud–mediated tra device.

Nella seconda parte si espongono i requisiti concordati con il relatore mentre nella terza parte è analizzata la specifica concordata e sono definite le componenti che sono state sviluppate suddividendo il lavoro tra me e Pietro Brunetti. Inoltre è descritto il ciclo che compie il contenuto informativo al fine di essere correttamente trattato.

Nella quarta parte si entra nel dettaglio della progettazione e delle tecnologie scelte in modo da permettere una comprensione dei meccanismi utilizzati per ottenere il comportamento, le prestazioni richieste e un buon disaccoppiamento delle componenti.

Negli ultimi due capitoli viene descritto come è avvenuto lo sviluppo dei moduli, entrando nel particolare delle parti da me trattate, e i test, conclusi

positivamente, condotti per appurare il buon funzionamento delle singole componenti e dell'intera infrastruttura.

Capitolo 1

Panoramica

In questo capitolo viene descritto l'ambito in cui si colloca il progetto, gli antefatti e le motivazioni che hanno spinto a costruire una infrastruttura atta a permettere comunicazioni tra dispositivi e cloud e a caricarsi di calcoli solitamente eseguiti dai device mobili.

Verrà eseguita una panoramica sulle tecnologie esistenti in campo di comunicazione P2P e cloud, mostrandone le principali differenze e problematiche.

1.1 L'ambito del mobile computing

Negli ultimi anni abbiamo potuto assistere a una esplosione nella diffusione di dispositivi mobili in termini di numeri, varietà e qualità. Una massiccia parte di questi è data da device utilizzati quotidianamente da persone di tutto il mondo come smartphone, tablet, laptop, ecc [6]. A questi va ad aggiungersi la realtà dei device usati per IoT e domotica che vantano un numero sempre maggiore di applicazioni e stanno conoscendo una iniziale diffusione proprio in questi ultimi tempi [7].

Ciò che accomuna questi dispositivi è il poter disporre di un notevole numero di sensori. Nel corso degli anni i produttori di dispositivi sono andati via via aggiungendo i più disparati sensori, che variano dal comune GPS o accelerometro ai più sofisticati di pressione o battito cardiaco, come conseguenza di una forte domanda da parte dell'utenza consumer [1].

Altra importante caratteristica che accomuna tali dispositivi è la possibilità di trasmettere dati seguendo svariati standard di comunicazione. Tutti

i moderni smartphone permettono un collegamento alla rete voce e dati di uno o addirittura più operatori telefonici, dispongono di una antenna WiFi e spesso anche di un modulo per la comunicazione ravvicinata, in primis Bluetooth. Altri dispositivi, specialmente quelli dedicati a IoT e domotica, montano moduli meno comuni, più specifici per l'ambito in cui devono operare.

Quello che si va definendo è un grande, inespresso, potenziale in termini di capacità di computazione, trasferimento dati e percezione dell'ambiente che sarebbe possibile utilizzare per applicazioni complesse. La diversità dei dispositivi in termini di sensori, autonomia e tecnologie di comunicazione rappresenta lo scoglio da superare per conseguire tale obiettivo.

1.1.1 Una rete di sensori

I dispositivi mobili sono nati come portale di accesso ai contenuti e piattaforma di lavoro.

Nel corso degli anni, grazie all'esplosione della sensoristica, questi si sono trasformati da consumatori a produttori di contenuti.

È sufficiente pensare a tutte quelle applicazioni che fanno uso della posizione del device per fornire il proprio servizio. I contenuti non si muovono più esclusivamente nella direzione che va dal fornitore al device ma anche in quella opposta a definire una collaborazione.

Tra tutti i dati che un dispositivo può fornire è sempre stato al centro dell'attenzione quello relativo alla posizione. La sua interpretazione più banale è rappresentata dalla possibilità di definire la collocazione nello spazio di un oggetto.

Tuttavia questa non rappresenta l'unica informazione utilizzabile: la posizione permette di calcolare la distanza tra due o più dispositivi, verificare se un dispositivo si trova in una certa area, contare il numero di device collocati in una certo raggio, tracciare la storia dei movimenti, ecc. e in base a queste informazioni etichettare il dispositivo o l'ambiente circostante.

Al giorno d'oggi esistono numerose applicazioni che fanno uso di queste tecniche. È sufficiente esplorare lo store dei dispositivi consumer per trovare:

- Mappe e navigatori completi di informazioni sul traffico, ottenute analizzando la velocità e la densità dei dispositivi ed effettuando previsioni basate sulla storia passata.

- Applicazioni dedicate al personal training con memoria dei percorsi da cui viene estrapolata la lunghezza e l'elevazione.
- Sistemi di suggerimento automatico riguardante gli spostamenti, basate sulla analisi dei percorsi seguiti quotidianamente.
- Applicazioni in campo ludico, basate sulla vicinanza a certe location ben definite.

Tali applicazioni appaiono spesso banali ma nascondono un alto grado di complessità dovuto alla gestione della posizione, un dato non sempre facile da trattare.

Un obiettivo che si cercherà di conseguire nei prossimi anni sarà la creazione di una rete di sensori nella quale questi dispositivi avranno un ruolo di primaria importanza. La costruzione di tale rete aprirebbe a una infinità di applicazioni che vanno dalla gestione delle emergenze, alla coordinazione di veicoli; dalla fruibilità di dati multimediali, alla domotica.

Se l'interazione tra un numero elevato, teoricamente infinito, di dispositivi vicini tra di loro fosse semplice e facilmente fattibile, l'unico ostacolo consisterebbe nella progettazione e implementazione di applicativi e algoritmi in grado di sfruttare le funzionalità di tale rete formata da device–sensori.

1.1.2 Interazioni tra device

L'interazione tra più dispositivi permetterebbe di acquisire informazioni sull'ambiente non più limitate a un singolo punto di vista. Una collaborazione di questo tipo rendebbe possibile ai device di diventare coscienti di ciò che li circonda. Questo aprirebbe la strada ad applicazioni in grado di fornire una elaborazione di questi dati all'utente per poi rendere disponibili a cascata i risultati ottenuti a tutti i dispositivi interconnessi.

Un insieme di device in continua comunicazione può dar vita a una infrastruttura in grado di fornire un servizio fruibile da chiunque ne abbia bisogno (un utente o altri device). È sufficiente pensare alle applicazioni di domotica e IoT, basate su interazioni di dispositivi vicini, per accorgersi delle potenzialità di un servizio così costruito.

Quello a cui si è giunto alla fine della fase di progettazione è tuttavia indipendente dalla specifica implementazione e riguarda ugualmente dispositivi vicini, ma anche sparsi su una più vasta area, in movimento e con una densità e numero non prevedibili.

Un primo approccio alla costruzione di una rete così flessibile sembrerebbe consistere nell'utilizzare le tecnologie più comuni già presenti sui moderni dispositivi per instaurare una comunicazione tra vicini. Una volta aperto un canale sarebbe possibile scambiare dati per poterne poi eseguire computazioni, eventualmente trasmettendone i risultati e prendendo decisioni.

Da questo punto di vista la tecnica di comunicazione ideale sarebbe basata su scambi di messaggi opportunistici. Tale concezione è basata sul fatto che un device possa identificare i suoi vicini e trasmettere/ricevere dei dati in broadcast. In questo modo sarebbe possibile ottenere una comunicazione funzionante con un qualsiasi numero di device e costruire una rete in grado di collaborare per ottenere consapevolezza dell'ambiente in cui essi si trovano immersi.

Una altro metodo consiste nell'appoggiarsi a una infrastruttura posta nel cloud che si comporti da ponte per scambiare dati tra dispositivi, per memorizzare valori o addirittura per alleggerire il carico computazionale dei device mobili.

Nella infrastruttura da noi creata abbiamo seguito questo ultimo approccio.

1.1.3 Comunicazione ravvicinata

Negli ultimi tempi vanno emergendo standard sempre più evoluti nell'ambito della comunicazione ravvicinata. Spesso si tratta di una evoluzione di predecessori già ampiamente utilizzati e possono essere presenti meccanismi di retrocompatibilità atti a favorire la transizione verso le nuove versioni.

Ciò su cui ci si va a focalizzare è permettere scambi di dati ad alte prestazioni e con consumi ridotti. Basare una rete per lo scambio di dati su queste tecnologie permetterebbe di preservare l'autonomia dei dispositivi mobili e si potrebbe ottenere un soddisfacente throughput. Esempi di tecnologie già diffuse o che stanno recentemente ricevendo attenzioni sono Bluetooth Low Energy, ZigBee, Z–Wave, Ant.

Problematiche da affrontare

Le tecnologie per la comunicazione ravvicinata attualmente più diffuse funzionano egregiamente con un numero ridotto di device interconnessi, risul-

Figura 1.1: Logo di Bluetooth

tando però inadeguate nel momento in cui tale numero fosse troppo elevato: il delay nelle comunicazioni può variare largamente in base alla disposizione fisica dei dispositivi o al numero di hop frapposti nel trasferimento di un pacchetto.

Va inoltre aggiunto che queste si basano su una infrastruttura che prevede la presenza di un master.

Una infrastruttura che si basa su tale concezione potrebbe essere inadeguata per la rete che si intende costruire. Come già spiegato la tecnica di trasmissione ideale consisterebbe nella possibilità di trasmettere e ricevere broadcast in qualsiasi momento, senza pesanti procedure di acknowledgment o accesso a una rete.

A questo va aggiunto che una tale rete necessiterebbe di una tecnologia universalmente accettata mentre attualmente è presente una forte frammentazione.

Una tecnologia ideale dovrebbe:

- Permettere comunicazioni broadcast e senza collisioni
- Avere bassi consumi
- Consentire un elevato throughput
- Essere universalmente diffusa

Di seguito una panoramica dei metodi di trasmissione ravvicinata più diffusi e del perché non rappresentano la tecnologia appena definita.

Figura 1.2: Logo del WiFi

Bluetooth

Bluetooth è lo standard industriale per la trasmissione di dati a distanza ravvicinata più diffuso in ambito mobile e, considerate le sue caratteristiche, largamente usato nell'ambito della domotica.

Qualità principali sono l'economicità e la bassa potenza richiesta. A seconda della versione le distanze di trasmissione vanno dagli uno ai cento metri con un un throughput che va da 723.1 kbit/s ai 24 mbit/s [5].

I dispositivi che prendono parte a una comunicazione bluetooth si uniscono per formare una "piconet". All'interno di essa è sempre presente un master che si occupa di inviare il proprio clock in modo tale da permettere agli slave di sincronizzarsi con lui. Più piconet possono far parte di una "scatternet".

Nonostante tale tecnologia soddisfi i requisiti di basso consumo, elevato throughput e diffusione permane comunque la problematica legata all'architettura master–slave e al numero limitato di interazioni possibili tra dispositivi, attualmente limitata a sette contemporaneamente. Inoltre le distanze coperte non sono così elevate e vengono fortemente limitate dalla presenza di ostacoli fisici.

WiFI Direct

WiFi–Direct è uno standard per la trasmissione di dati che condivide le caratteristiche di velocità e distanza coperta del classico WiFi. Il supporto da parte dell'industria è in forte aumento.

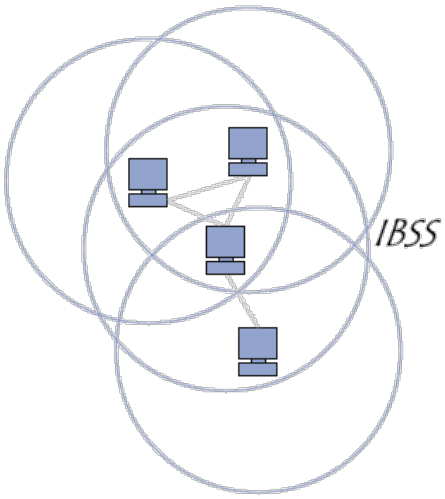


Figura 1.3: Esempio di rete IBSS

Lo standard prevede di eseguire un Access Point su ogni device che intenda far parte di una rete. Questi stabiliscono una comunicazione utilizzando il WiFi Protected Setup.

WiFi–Direct soddisfa i requisiti di throughput, diffusione e presenta una buona distanza coperta anche in presenza di ostacoli. Tuttavia non rappresenta una vera soluzione P2P e le comunicazioni atte a stabilire un collegamento tra dispositivi sono onerose e prevedono lo scambio di un grande numero di frame. In particolare la caratteristica P2P grandemente publicizzata è relativa solamente alle prime fasi di setup della rete: una volta che i device sono connessi viene stabilito un "group owner" [14].

WiFi Ad-Hoc

Un standard interessante è quello del WiFi Ad-Hoc, cioè WiFi in cui è assente una infrastruttura [17].

L'assenza di un master permette una vera comunicazione P2P broadcast tra tutti i dispositivi in possesso di una antenna WiFi anche di vecchia generazione, dal momento che questo standard è parte del IEEE 802.11 sin dai suoi albori.

Il numero di frame iniziali richiesti per instaurare una comunicazione tra dispositivi è molto basso ma allo stesso tempo questo porta a una sicurezza inesistente dovuta alla possibilità di eseguire liberamente sniffing e attacchi man–in–the–middle.

Le comunicazioni si basano sull'invio di frame a un indirizzo MAC prefissato (BSSID), che identifica la rete. I dispositivi che intendano far parte di una rete publicizzano la loro appartenenza a tale rete inviando periodicamente alcuni frame che contengono anche la definizione di un nome umanamente leggibile (ESSID). Se esistono più reti con lo stesso nome ma con indirizzi MAC differenti allora viene eseguita una procedura di merge, che rappresenta il punto debole di questa tecnologia.

Lo studio del WiFi Ad–Hoc è stata affrontata da me e dal collega Marco Nobile nel corso del tirocinio curricolare tenutosi presso il PSLab della Facoltà di Ingegneria e Scienze Informatiche di Cesena. In breve, dallo studio è risultato che allo stato attuale delle cose tale metodo di comunicazione è praticamente inutilizzabile massivamente in ambito mobile per via dello scarso interesse da parte dei produttori dei moduli hardware nel supportare tale tecnologia.

In particolare molti produttori non forniscono hardware o driver che permettano di esplicitare l'indirizzo MAC della rete, che viene calcolato casualmente. Dispositivi con nomi di rete identici ma indirizzi diversi portano alla esecuzione della procedura di merge, generalmente mal implementata.

1.2 La computazione Cloud Mediated

Il progetto di creazione di un sistema per la comunicazione cloud–mediated tra dispositivi si inserisce in un più ampio contesto di ricerca riguardante la possibilità di costruire una rete auto–organizzante. Tale rete dovrebbe essere in grado di mostrare comportamenti emergenti complessi svolgendo una serie di computazioni elementari e/o la loro composizione. Lo studio di tale contesto esula largamente dagli obiettivi di questa tesi ma ha permesso di inquadrare con maggior consapevolezza il suo fine ultimo.

La comunicazione cloud–mediated rappresenta un'alternativa all'uso delle tecnologie per lo scambio di dati a distanza ravvicinata. Una infrastruttura basata su cloud non solo permetterebbe lo scambio di informazioni con buone performance e con la sicurezza di avere la piattaforma sempre

disponibile, ma aprirebbe anche le porte alla possibilità di spostarvi le computazioni in modo da aumentare le prestazioni in relazione alla capacità computazionale dei dispositivi.

Come si vedrà in seguito il sistema costruito mima le comunicazioni broadcast tra dispositivi vicini.

1.2.1 Applicazioni location-based

Come già esposto in precedenza la posizione rappresenta un dato prezioso e sfruttato da applicazioni riguardanti una grande varietà di campi. Tale dato può riferirsi a un modello di riferimento che può essere un semplice piano cartesiano o il pianeta; un piano bidimensionale o uno spazio tridimensionale.

Nelle applicazioni location–based che interagiscono con un servizio posto nel cloud il dato relativo alla posizione viene inviato periodicamente dai dispositivi ed è rappresentato da coordinate GPS. La ricezione del dato da parte del servizio porta alla restituzione di una risposta coerente con la nuova posizione del dispositivo.

Il caso trattato in questa tesi riguarda la interazione tra dispositivi posti a distanza ravvicinata utilizzando il cloud, per cui si rende necessario sviluppare un sistema che riceva e memorizzi le posizioni più aggiornate di tutti i dispositivi al fine di mantenere una lista aggiornata dei vicini rispetto a ognuno di essi. Tale lista, se necessario, verrà restituita ai dispositivi.

Essendo l'ambito quello del mobile computing, il vicinato varia più volte nella vita di un dispositivo e le modifiche relative all'ambiente circostante possono avvenire in qualsiasi momento.

1.3 Database basati sui grafi

Tutti i dati devono essere memorizzati in una qualche forma che ben rappresenti il contesto e che ne favorisca l'accesso e la manipolazione.

Una rete di device in grado di comunicare tra di loro ben si presta a essere espressa sotto forma di un grafo nel quale i nodi sono rappresentati dai device e gli archi, pesati e non diretti, rappresentano la relazione di vicinato.

Figura 1.4: Logo di Neo4j

La presa di coscienza di tale relazione equivale all'aggiunta di un arco, l'aggiornamento della distanza al cambio del peso di questo, l'allontanamento alla sua eliminazione.

Al fine di memorizzare tale struttura dati si è fatto affidamento al database NoSQL Neo4j.

1.3.1 Neo4j

Neo4j è un database basato su grafi sviluppato dalla Neo Technology, Inc. Si tratta di un software open source per cui sono previste sia una versione Enterprise a pagamento sia una versione gratuita sotto forma di Community Edition con licenza GPL v3. L'intero sistema è implementato con il linguaggio Java.

Neo4j è utilizzato in ambito di produzione in grandi realtà come Ebay, Walmart, Pitney Bowes, Crunchbase, Megree, Tomtom, ecc [12].

Possibilità offerte

Il database permette di organizzare i dati sotto forma di grafo, cosa che lo rende particolarmente adatto allo scopo, con prestazioni assolutamente eccellenti. Le aziende sopra citate utilizzano questo software per gestire un numero di operazioni che può arrivare alle decine di migliaia per secondo, in ambienti in cui errori potrebbero portare a problemi gravi. Il database è completamente transazionale.

Modalità di funzionamento

Neo4j prevede due modi di funzionamento:

- Embedded: una istanza del database può essere creata all'interno di un altro applicativo in esecuzione su di una Java Virtual Machine.

  Questo metodo di funzionamento permette all'applicativo ospitante di eseguire operazioni utilizzando le apposite API di basso livello messe a disposizione dal database, inclusa la gestione delle transazioni e la possibilità di inserimento di dati in batch in maniera non concorrente.

- Standalone: il database viene lanciato in un apposito processo. Questa è la modalità più utilizzata in ambito di produzione in quanto permette una buona configurabilità in termini di memoria allocabile, quantità di pagine in memoria, ecc.

Per entrambe le modalità di funzionamento è previsto il salvataggio di tutti dati in una directory sul file system per cui è possibile passare da una tipologia all'altra utilizzando gli stessi dati.

Come già accennato Neo4j è un database NoSQL e prevede che tutte le query siano scritte in un linguaggio dichiarativo chiamato Cypher. Tale linguaggio è stato appositamente pensato per operare che una struttura dati basata su grafi.

Interfaccia grafica

Il database mette a disposizione un'interfaccia grafica web particolarmente user friendly con cui eseguire query Cypher e visualizzarne i risultati sotto forma di un grafo. Tale interfaccia è disponibile solamente se Neo4j è eseguito in modalità standalone.

Questa caratteristica è stata fondamentale in fase di debugging e testing in quanto permette di visualizzare tutti i nodi e gli archi di interesse contenuti nel database.

Plugin spaziale

Il fatto di essere scritto in Java e di essere open source ha permesso alla comunità di creare plugin facilmente installabili sul server.

Uno di questi, Neo4j Spatial, permette di memorizzare dati spaziali (punti e poligoni) e interrogare il database in modo da ottenere, in maniera efficiente sfruttando un RTree, le geometrie che si trovano a una distanza data rispetto a una certa posizione [15].

Come verrà esposto nel capitolo riguardante la progettazione, questo ha semplificato di molto la gestione delle posizioni al costo di una più macchinosa fase di setup iniziale delle strutture dato memorizzate nel database.

1.4 Tecnologie per la computazione in Cloud

La componente nel cloud deve poter gestire una serie di eventi con buone performance e resistenza ai guasti. La scelta delle tecnologie da utilizzare è ricaduta su software provenienti dall'ecosistema Apache.

Si tratta di progetti open–source continuamente in fase di sviluppo e scritti, come la quasi totalità del software prodotto dalla Apache Software Foundation, interamente in Java, Scala e Clojure. Questo rende tali software in grado di essere eseguiti su un qualsiasi sistema dotato di una Java Virtual Machine, pertanto lo sviluppo dell'intera infrastruttura è avvenuto utilizzando il linguaggio Java.

Java e tutti i linguaggi basati su JVM sono da sempre al centro di un dibattito inerente le performance degli applicativi sviluppati in tali linguaggi. Storm e Kafka, scritti completamente in tali linguaggi, rappresentano i migliori tra i sistemi del loro campo; tale fatto è confermato dalla loro diffusione in grandi aziende.

In particolare Kafka è ormai considerato in uno stadio di sviluppo maturo ed è largamente utilizzato, anche per via del ruolo che può coprire in una qualsiasi infrastruttura, in una grande varietà di ambienti.

Tra i "grandi utenti" di Storm e Kafka spiccano LinkedIn, Yahoo, Paypal, Goldman Sachs, Spotify, WebMD, Groupon, Alibaba, The Weather Channel, ecc.

Tale successo è da ricercare, oltre che nelle ottime performance, nella loro facilità di setup e configurazione e al supporto per l'interoperabilità offerto in veste di librerie aggiuntive sempre aggiornate all'ultima versione dei sistemi.

Figura 1.5: Logo di Storm

1.4.1 Apache Storm

La componente di backend è basata su Apache Storm, un sistema open source per la computazione distribuita di eventi.

Inizialmente pensato e prototipato da Nathan Marz, è entrato a far parte dell'ecosistema Apache da Dicembre 2013, inserito all'interno dell'Apache Incubator e vi è uscito nel Settembre 2014 diventando un progetto principale [3] [9].

Possibilità offerte

Storm permette di distribuire la gestione di una serie di eventi. La computazione avviene in realtime, con ottime performance, ed è garantito il completamento delle operazioni (fault tolerant). È possibile definire la granularità della computazione per ogni tipo di operazione sotto forma di Worker (processi) e Task (threads) che a loro volta sono potenzialmente distribuiti su macchine fisiche differenti.

Come si vedrà nei prossimi capitoli, queste features rappresentano dei requisiti per l'infrastruttura.

Storm si basa su un sistema di topologie definite in maniera programmatica. Questa procedura consiste nella descrizione dei punti di ingresso dei dati, delle operazioni da effettuare e le direzioni riguardanti i flussi al suo interno. Come già accennato Storm è basato su JVM per cui questo si traduce nella specifica di classi contenenti la logica della computazione, il numero di istanze da costruire, la quantità di processi e thread da lanciare e infine la struttura dei dati trattati.

Un inseme di slave in esecuzione su un cluster è in grado di eseguire computazioni riguardanti più topologie contemporaneamente in modo da permettere il più completo utilizzo possibile delle macchine.

Una descrizione più pratica e decisamente più esplicativa è demandata al capitolo inerente la progettazione.

1.4.2 Gestione dello stream di eventi

L'infrastruttura creata deve gestire un grande quantità di eventi. I messaggi in arrivo dai device devono essere trattati nel loro ordine di arrivo, cioè come una coda.

Questo rappresenta una situazione comune nel campo della computazione di eventi. Al fine di poter gestire tale flusso è comune appoggiarsi a un broker di messaggi.

Un broker si occupa di memorizzare e rendere disponibili dei dati al fine di potervi accedere non appena possibile o necessario. Una tale componente ben si presta ad essere utilizzata in ambienti in cui il flusso è variabile e comunque particolarmente sostenuto. Solitamente la manipolazione del flusso dati avviene in termini di letture e scritture FIFO, anche se, a seconda del broker, potrebbero essere possibili metodologie differenti.

Ovviamente ogni broker presente sul mercato presenta caratteristiche differenti ed è necessario scegliere quello che si ritiene più adatto al particolare obiettivo da conseguire.

I principali benefici nell'utilizzare un broker sono:

- Interporre un buffer tra un insieme di produttori di numero sconosciuto e un consumatore.

  Tale pratica permette una certa flessibilità sulla quantità di messaggi consumabili in un certo lasso di tempo. Se i messaggi risultassero troppi la coda si accollerebbe la funzione di tampone, ovviamente limitato dalla memoria (RAM o disco) a lei assegnata, in modo da permettere al consumatore di non perdere gli eventi ricevuti.

- Disaccoppiare le implementazioni di produttore e consumatore.

  Ovviamente utilizzare una coda di messaggi permette di evitare l'interfacciamento diretto tra due componenti differenti appartenenti allo stesso sistema che solitamente è bene mantenere separati.

Figura 1.6: Logo di Kafka

Anche se la tecnologia alla base della coda venisse modificata sarebbe sufficiente sostituire la libreria precedentemente utilizzata con una nuova, fornita dal produttore del broker. In tali termini il ruolo svolto da una coda rappresenta una interfaccia il cui cambio di implementazione è relativamente indolore.

- Permettere l'accesso ai dati in qualsiasi momento.

  Anche se il consumatore si trova offline al momento dell'arrivo del messaggio, la coda mantiene in memoria il dato fin quando non verrà consumato. Questo particolare permette, anche a persone con bassa preparazione sul tema, la costruzione di sistemi fault–tolerant basando tale proprietà sulle possibilità offerte dal broker.

1.4.3 Apache Kafka

Apache Kafka è un message broker open source distribuito e ad alte prestazioni, basato sulla scrittura di "log". Kafka è completamente scritto in Scala e pertanto eseguibile su un qualsiasi sistema dotato di JVM [2]. Kafka presenta una architettura publish-subscrive.

Il concetto di "log" è l'intuizione alla base delle eccellenti prestazioni offerte da Kafka. Al fine di implementare una coda di messaggi persistente il broker esegue scritture demandandone la gestione al sistema operativo. La componente di gestione dei file è quindi minimale e si basa sull'efficienza dei moderni sistemi operativi nell'eseguire l'operazione di append.

Ogni operazione di aggiunta di un messaggio alla coda è quindi vista come l'aggiunta di una linea di log.

I log possono essere compattati o eliminati solitamente basandosi su un approccio lazy che predilige il ritardare tali operazioni a momenti in cui il broker non è impegnato nella gestione di grandi flussi, oppure a quando tale operazione diventa non rimandabile.

Possibilità offerte

Kafka permette la suddivisione di messaggi in topic, la parallelizzazione di letture e scritture anche su più macchine, la gestione e la cancellazione dei messaggi in base a una chiave o a un timeout.

Oltre a questo offre ridondanza attraverso un sistema di repliche in modo da poter mantenere i contenuti sempre disponibili anche a seguito di guasti.

Un cluster può quindi essere utilizzato da più applicazioni contemporaneamente e per scopi diversi. Kafka infatti permette una grande configurabilità nella gestione di ogni tipologia di dato trattata.

Si tratterà dei metodi utilizzati per la suddivisione, la replica e la parallelizzazione di letture e scritture dei dati nella parte relativa alla progettazione della infrastruttura.

1.5 Visualizzazione dei risultati

È importante che tale sistema distribuito, basato su sensori, debba poter essere osservabile al fine di compiere analisi sia quantitative che qualitative.

Nasce quindi la necessità di creare una interfaccia in grado di rendere visibili i dati desiderati. Un approccio basato su web, in particolare sfruttando le possibilità offerte da Javascript riguardanti le comunicazioni asincrone, rappresenta una buona soluzione.

Al fine di semplificare la gestione della componente grafica dell'interfaccia è stato scelto di utilizzare la libreria Linkurious.

1.5.1 Linkurious

Linkurious è una libreria javascript open source per il rendering di grafi all'interno di un browser web [8]. Si tratta di un fork di Sigma, un'altra libreria per il rendering di grafi, da cui si differenzia per un motore di

rendering leggermente modificato e una grande varietà di plugin out–of–the–box.

Linkurious permette di eseguire azioni di aggiunta, modifica ed eliminazione di nodi e archi, impostare colorazioni e dimensioni personalizzate per ogni elemento, rilevare le interazioni da parte degli utenti, utilizzare più viste dello stesso grafo e la possibilità di cambiare il livello di dettaglio al variare dello zoom.

Tutto ciò è possibile utilizzando uno dei motori di rendering predefiniti, Canvas e WebGL, oppure costruendo un motore personalizzato.

La libreria deve essere guidata da un applicativo in grado di ottenere, organizzare e comunicare i dati da visualizzare. A tal fine è stato necessario costruire un applicativo AJAX che si interfacciasse con un servizio web per ottenere i dati richiesti.

1.5.2 AJAX

AJAX, acronimo di Asynchronous JavaScript and XML, è una tecnica di comunicazione utilizzata per ottenere una pagina web dinamica. Grande parte delle pagine web moderne fa uso di queste tecnologie per offrire una maggiore possibilità di interazione e per presentare all'utente una versione sempre aggiornata dei dati visualizzati [16].

Esempi di applicazioni di questo tipo sono le interfacce utilizzate per accedere alla casella mail, motori di ricerca con suggerimento dei risultati durante la digitazione, mappe, piattaforme per l'e–commerce, ecc.

Se ben costruita, una applicazione web che utilizza queste metodologie è del tutto paragonabile, in termini di reattività e facilità di utilizzo, a un applicativo eseguito in locale con le stesse finalità.

Il termine Asynchronous rispecchia la natura non bloccante degli applicativi Javascript. Vi è infatti la necessità di non bloccare il flusso di controllo per attendere l'arrivo del risultato di una interrogazione al servizio web in quanto Javascript è concepito per l'esecuzione su un solo thread. Al fine di poter ottenere una interfaccia reattiva è necessario lanciare la richesta desiderata per poi ottenerne, tramite la notifica ricevuta da un apposito callback, il risultato.

Tutte le comunicazioni avvengono tra il browser e il servizio web sotto forma di richieste HTTP. Negli ultmi anni sono andati a definirsi due principali formati per strutturare i dati scambiati tra browser e servizi web: XML

e JSON. In particolare quest'ultimo sta ricevendo particolari attenzioni e i servizi che preferiscono JSON rispetto a XML è in aumento. Al contrario XML sta ricevendo maggiori attenzioni nel campo della definizione di interfacce grafiche e strutture dati. In ogni caso tramite AJAX è possibile scambiare dati di qualsiasi tipo.

Capitolo 2

Il caso di studio

In questo capitolo vengono esposti gli obiettivi e i requisiti del progetto.

Gli obiettivi erano da subito chiari mentre i requisiti, anche se espressi in linea di massima, lasciavano più margine di scelta riguardo alla parte di progettazione e scelta delle tecnologie.

Nell'ultima sezione viene esposto il caso particolare di computazione usato come test.

2.1 Descrizione della infrastruttura

Si intende sviluppare una infrastruttura basata su cloud che permetta a device mobili che dispongono di una connessione di rete di scambiare tra loro messaggi.

L'obiettivo finale è permettere la computazione di un valore, a cui d'ora in avanti mi riferirò con il nome di "stato", come risultato di una funzione. Tale funzione incapsulerà un algoritmo più o meno complesso e deve poter essere intercambiabile.

Le possibili funzioni hanno in comune i dati richiesti come input, che sono composti dal valore assunto dai sensori presenti sul device e dall'insieme degli stati ricevuti dai dispositivi nelle vicinanze e le relative distanze.

Ogni device dovrà poter scambiare dati riguardanti sia il proprio stato sia il valore dei propri sensori in ogni momento tenendo in considerazione la natura mobile dei dispositivi ed considerando le risorse a disposizione in termini di capacità di computazione ed autonomia come limitate.

Il vicinato è dato dall'insieme di device fisicamente vicini a lui in un raggio fissato uguale per tutti. Tutte le misure riguardanti la posizione e le distanze dovranno rapportarsi al sistema di riferimento del globo terrestre, per cui le coordinate devono essere espresse come latitudine e longitudine.

2.1.1 Tipologie di comunicazione

Devono essere possibili due tipologie di interazione con il cloud:

- Nella prima tipologia i device mobile eseguono la computazione della funzione.

  I device necessiteranno dei valori dello stato dei vicinati e invieranno periodicamente all'infrastruttura cloud il risultato della computazione e la loro posizione.

- Nella seconda i device mobile non eseguono la computazione.

  I device dovranno solamente inviare il valore dei propri sensori (oltre alla posizione) e la computazione della funzione dovrà avvenire in cloud.

2.2 Descrizione dell'interfaccia esterna

L'interfaccia attraverso la quale i device mobili si collegano all'infrastruttura cloud deve essere semplice, permettere uno scambio di informazioni veloce e senza spreco di risorse da parte dei dispositivi.

A tal fine non è necessario che il device ottenga il risultato dell'operazione nella stessa interazione in cui invia i propri dati.

Deve essere possibile ottenere i dati più aggiornati riguardanti ogni device utilizzando una apposita interfaccia.

2.3 Descrizione del backend

La componente di backend deve permettere la computazione realtime dello stream di eventi generato dai device, cioè deve fornire un risultato in tempi brevi e deve essere possibile distribuire le operazioni su più macchine fisiche in modo da poter scalare in caso di necessità.

Tale infrastruttura dovrà essere disponibile online in ogni momento al fine di non perdere nessuno dei messaggi consegnati dai device.

2.4 Implementazione della funzione

Come caso di studio e test della piattaforma da noi costruita è stato implementato l'algoritmo "closest source".

I device sono suddivisi in "sorgenti" e non sorgenti. Una sorgente è tale in quanto designata arbitrariamente o a seguito della presa di coscienza di certi avvenimenti che riguardano quel dispositivo. Nella implementazione di test supponiamo che sia presente su ogni nodo un sensore dal valore booleano che va a definire se un device è una sorgente o meno.

Solitamente i nodi sorgente sono in numero molto inferiore rispetto alla totalità dei device.

Lo stato di un device è rappresentato da un valore numerico che indica la sua distanza (in metri, kilometri, ecc.) dalla sorgente più vicina. In ogni momento un device percepisce la distanza dai suoi vicini e il loro stato.

La funzione somma, alla distanza da ogni vicino, il relativo stato. Ciò che si ottiene è una lista di distanze da una qualsiasi sorgente attraverso ogni vicino. Tali distanze non rappresenteranno la distanza in linea d'aria ma saranno indicative della lunghezza di un percorso utilizzabile per raggiungere una sorgente. Il risultato della funzione, che andrà a definire il nuovo stato, consisterà nel minimo tra questi valori.

Tale valore viene poi ritrasmesso ai vicini che aggiorneranno il proprio stato. Quello che si ottiene è una rete auto organizzante in cui i device riescono a percepire la distanza da una qualsiasi sorgente attraverso continui raffinamenti.

Tale algoritmo può essere eseguito sia su cloud che sui dispositivi e tiene conto della natura mobile dei dispositivi per cui ben si presta a un test dell'intera infrastruttura.

2.5 Testing

Al fine di testare l'infrastruttura, considerate le problematiche legate alla disponibilità di device reali, è da considerare sufficiente l'utilizzo di un si-

mulatore in grado di riprodurre il comportamento di alcuni dispositivi in movimento.

Deve essere misurato il tempo, calcolato per ogni device, che intercorre tra l'invio di un messaggio e la ricezione della prima risposta da parte di uno qualsiasi tra i dispositivi interessati. Questi ultimi sono rappresentati, nel caso in cui sia il cloud a computare, dalla sola origine dell'evento oppure, nel caso in cui la computazione avvenga in locale, di vicini.

Questa misurazione deve avvenire fissando la frequenza di computazione con tempi via via più dilatati e con un numero crescente di dispositivi.

Durante i test si può mantenere un vicinato di circa 10/20 nodi.

2.5.1 Visualizzazione dei risultati

Al fine di poter interpretare i risultati è stata ritenuta necessaria la costruzione di un viewer che permettesse di visualizzare graficamente i device in movimento.

Riguardo alla tecnologia da utilizzare non sono state espresse specifiche richieste così come non è stato ritenuto necessario che tramite tale interfaccia fosse possibile manipolare il contenuto presente sul cloud.

Il viewer deve visualizzare in maniera differente le sorgenti rispetto agli altri device, ad esempio tramite una apposita colorazione o forma, e deve visualizzare le relazioni di vicinanza che intercorrono tra i nodi.

Oltre a questo deve essere possibile visualizzare dati relativi ai nodi quali il valore dei sensori e lo stato.

2.6 Casi d'uso

Sono facilmente individuabili due tipologie di attori: un "computational device" e un "non computational device" (o "sensor device"). Queste due tipologie di dispositivo si interfacciano con il cloud inviando dati differenti e scatenando perciò comportamenti differenti da parte del backend.

Dalla figura 2.1 appare evidente come il punto in comune consista nell'aggiornamento del vicinato, procedura necessaria per fornire il servizio a entrambe le tipologie di device.

In questo senso ciò per cui differiscono le due tipologie di attori è la necessità di disporre delle informazioni riguardanti il vicinato: ai fini della

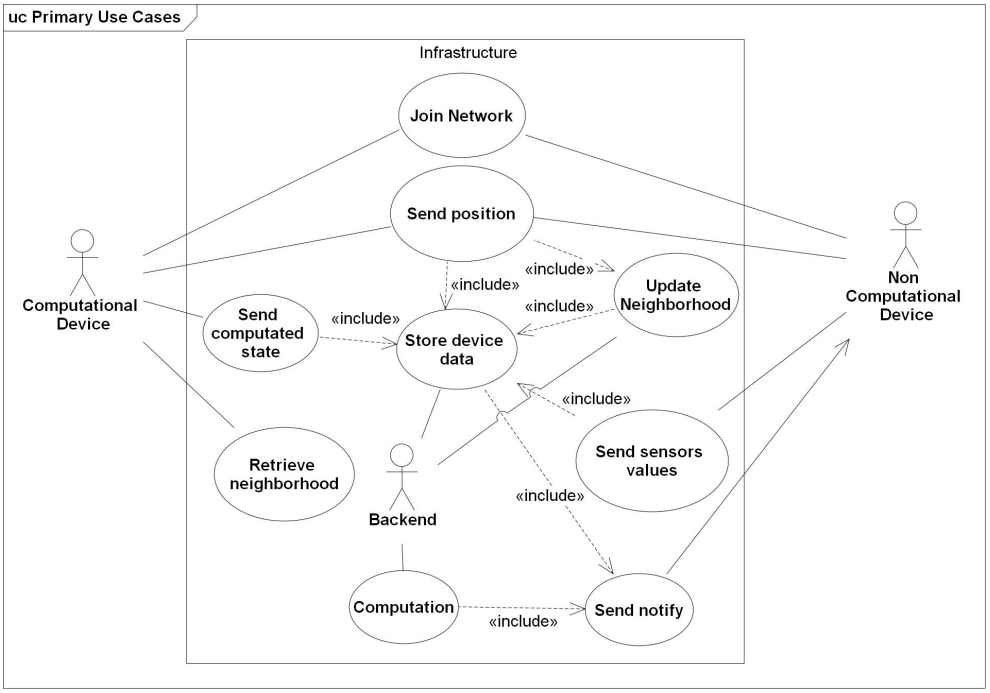


Figura 2.1: Casi d'uso principali

computazione su device è necessario che il dispositivo abbia percezione di ciò che lo circonda per cui è stato definito il caso d'uso "Retrieve neighborhood".

Al contrario il device non computazionale non necessita di un tale dato e si comporta come un sensore posto nell'ambiente. L'unica interazione di ritorno consiste nella ricezione della notifica di avvenuta computazione che, a seconda dell'implementazione del client in esecuzione sul device, può tornare utile in vari modi, ad esempio per dilazionare l'invio del valore dei sensori in modo che avvenga a intervalli regolari.

Capitolo 3

Analisi del problema

In questo capitolo vengono analizzati i requisiti espressi nel capitolo precedente in modo da delineare una struttura di massima, indipendente dalle tecnologie utilizzate, che rappresenti l'intera infrastruttura.

3.1 Backend

In questa sezione vengono snocciolati i requisiti del backend. I requisiti riguardano fondamentalmente la tipologia dei dati trattati, le performance da ottenere, la distribuibilità e la possibilità di mantenere sempre online la componente nel cloud.

3.1.1 Dati trattati

Appare da subito evidente come sia stato necessario trattare tre tipologie di dato:

- La posizione.

  Un vincolo espresso in fase di stesura dei requisiti riguarda il fatto che la "visibilità" dei device sul vicinato debba essere legata a una distanza massima, eventualmente espressa sotto forma di costante, configurabile da file o da linea di comando, uguale per tutti.

  In mancanza del dato riguardante la posizione dei device l'infrastruttura non avrebbe senso di esistere: lo scambio di dati si ridurrebbe a una coda di messaggi broadcast e sarebbe impossibile filtrare i dati in

base alla posizione, cioè si perderebbe la possibilità di far convivere un numero elevato e variamente distanziato di dispositivi.

Le coordinate sono espressamente riferite al sistema GPS per cui sono rappresentate da numeri a virgola mobile.

- Il valore dei sensori.

  Ogni device possiede dei sensori da cui può ottenere dei valori.

  In questo progetto si considera implicito che i device posseggano dei sensori conosciuti e utili ai fini della computazione della funzione descritta in fase di stesura dei requisiti.

  In qualunque caso, considerato che la funzione iniettabile è sconosciuta e variabile, così come anche i valori dei sensori presi in considerazione, è stato ritenuto necessario trattare i dati riguardante i sensori come opachi, cioè non interpretabili.

- Lo stato.

  Tale dato rappresenta il modo con cui i device percepiscono il vicinato ed è dato dal risultato della computazione di una funzione che prende in ingresso lo stato dei vicini, le relative distanze e il valore dei sensori locali.

  Come nel caso dei sensori la tipologia di dato da trattare è sconosciuta e quindi è stato ritenuto necessario trattare tale dato come opaco.

Ovviamente sia i dati riguardanti i sensori sia i dati riguardanti lo stato sono interpretabili dalle specifiche implementazioni delle funzioni.

3.1.2 Scalabilità

È stato espresso un esplicito vincolo sulla scalabilità della infrastruttura. Nello specifico è stato chiesto di poter distribuire le operazioni su più macchine fisiche in modo da poter far fronte a un flusso di eventi variabile. Questo ha posto la necessità di organizzare una rete di macchine in grado di cooperare.

A fine di distribuire il calcolo è stato necessario suddividere ogni operazione, che si vedrà poi necessaria al fine di conseguire gli obiettivi, in sotto–operazioni più semplici e distribuibili.

È stato specificato il requisito che il backend sia sempre online per cui l'infrastruttura deve far fronte a eventuali guasti per recuperare in autonomia la propria capacità di computazione e, se necessario, redistribuire il carico.

3.1.3 Performance

La computazione deve avvenire in realtime, cioè dall'arrivo di un messaggio da parte di un device alla conclusione della sua gestione deve intercorrere un lasso di tempo il più corto possibile.

Tale reattività è ottenibile solo a seguito di un'attenta organizzazione del flusso di dati, l'eliminazione di tutti i colli di bottiglia, le possibilità offerte da una rete di macchine cooperanti e da tecnologie particolarmente performanti.

Questo requisito, anche se chiaramente esplicitato, sarebbe potuto risultare implicito in quanto la rete è basata su device reali e in movimento. Tempi di gestione troppo lunghi andrebbero a scontrarsi con il continuo cambio di stato dell'ambiente reale e della posizione del device, per cui risultati arrivati con troppo ritardo potrebbero risultare inutili o errati.

3.2 Interfaccia esterna

I device devono potersi interfacciare con la componente cloud attraverso un'interfaccia semplice. Un ulteriore vincolo consiste nel fatto che i device sono connessi a una rete internet standard per cui la scelta tra i metodi di comunicazione utilizzabili è dovuta ricadere tra i protocolli di rete maggiormente supportati, in particolar modo soluzioni basate su comunicazioni TCP o UDP.

Se si desiderasse utilizzare comunicazioni orientate alla connessione le interazioni non dovrebbero prolungarsi più del dovuto in quanto potrebbero sussistere problemi di connettività.

Se invece si propendesse all'utilizzo di una comunicazione connectionless si renderebbe necessario ingegnare un metodo per confermare al device la corretta ricezione del messaggio.

Considerando le problematiche dovute allo scambio di messaggi UDP, in particolar modo legate alla presenza di NAT (o altri filtri) e all'assenza

di un sistema di conferma della ricezione è stata ritenuta più adatta una soluzione basata su TCP.

3.2.1 Invio dei dati al cloud

L'invio dei dati al cloud dovrebbe avvenire senza dispendio di risorse da parte del device, in una modalità facilmente implementabile e che permetta di accorgersi di eventuali errori di rete per poter ritentare successivamente l'invio.

È stata ritenuta conveniente una comunicazione in cui il device non attende il risultato delle operazioni ma solamente la conferma di avvenuta ricezione. Dovranno essere le tecnologie lato cloud a garantire che il dato ricevuto venga effettivamente trattato anche in caso di errori interni o guasti.

Questo permette di prevenire errori di comunicazione che invece potrebbero sussistere nel caso in cui la connessione venga tenuta aperta al fine di attendere il risultato: un errore di connessione dovuto a un'improvvisa interruzione della rete è assolutamente possibile in ambito mobile; in alcuni casi potrebbe essere impossibile distinguere il caso in cui il dato è stato ricevuto dal caso in cui il dato non è stato ricevuto per cui si renderebbe necessario il ripetere l'invio anche quando non necessario.

3.2.2 Restituzione di risultati

Considerata un'architettura di questo tipo si è reso necessario definire un meccanismo a parte per la restituzione dei risultati.

Esistono due principali metodologie che rappresentano due correnti di pensiero contrapposte:

- Polling: il device interroga periodicamente l'interfaccia esterna al fine di verificare se la computazione è completata e quindi ottenerne il risultato. Questo metodo ha come lato positivo la semplicità architetturale del dispositivo ma ha come grande svantaggio la continua apertura di connessioni e la non reattività all'arrivo del risultato: un periodo di polling troppo lungo può portare a un calo di reattività; viceversa un periodo troppo corto può portare alla formulazione di richieste inutili.

- Subscrive: il device riceve il risultato sotto forma di notifica dal cloud. Questo metodo ha come lato positivo la reattività nel ricevere il risultato ma ha come lato negativo il dover mantenere aperto un canale di comunicazione con esso.

È stato scelto questo ultimo approccio in quanto si è considerata la reattività ai risultati particolarmente importante. I risultati indirizzati al device mobile possono essere accodati e, anche in caso di errori di rete, prelevati non appena la connessione di rete risulti disponibile.

A questo va aggiunto che questa scelta ha permesso di slegare l'interfaccia esterna dal backend computazionale: la componente che si occupa della ricezione dei dati dai device sarà differente dalla componente che si occupa di ricevere e comunicare i risultati.

Questo ha permesso di utilizzare concetti e tecnologie differenti nella progettazione di tali componenti permettendo una maggiore flessibilità.

3.2.3 Suddivisione in reti

È stato deciso di suddividere i dati trattati in reti.

Ogni rete è rappresentata da device che condividono la stessa funzione da computare. In questo modo è possibile mandare in esecuzione più istanze della infrastruttura, suddividere i device e adottare differenti politiche di gestione in base alla rete di appartenenza mantenendo un'unica interfaccia esterna.

In un ipotetico caso, non oggetto di questa tesi, in cui si necessiti eseguire computazioni riguardanti differenti funzioni, una suddivisione in reti permetterebbe all'interfaccia esterna di dirigere i messaggi verso il backend corretto in base all'identificativo della rete a cui sono destinati i dati.

Un caso meno generale, considerato nella infrastruttura costruita, è la possibilità di avere reti differenti ma che utilizzano la stessa funzione. Un solo backend è in grado di gestirle mantenendo i dati separati.

In qualsiasi caso il device deve eseguire una tantum una procedura di accesso a una rete, di cui conosce già l'identificativo. In seguito si rende necessario che il device comunichi la rete di appartenenza e un proprio identificativo, univoco in quella rete, a ogni interazione con il cloud.

3.3 Architettura Logica

In questa sezione viene definita una architettura conforme ai requisiti ma comunque generica e indipendente dalle tecnologie utilizzate.

3.3.1 Componenti richieste

Nell'architettura si possono delineare quattro componenti:

- Un applicativo o libreria in esecuzione su device mobile.

  Questa componente deve poter gestire il ciclo di ricezione–computazione–trasmissione interfacciandosi con il cloud e supportare sia il caso in cui la computazione debba avvenire su device sia il caso in cui debba avvenire su cloud.

- Interfaccia di ricezione dei dati.

  Questa componente deve ricevere i dati inviati dai device e consegnarli alla componente di computazione. Suo compito è restituire tempestivamente una conferma di avvenuta ricezione.

- Backend computazionale.

  Il backend deve organizzare ed effettuare la computazione e restituire un risultato in tempi brevi. Deve prevedere scalabilità, distribuibilità e tolleranza ai guasti. Dovrà inoltre occuparsi della memorizzazione delle informazioni più recenti relative ai device in modo da poterle utilizzarle lui stesso e renderle sempre disponibili per la consultazione.

- Interfaccia per la restituzione dei risultati.

  Questa componente deve accodare e inviare una notifica ai device interessati nel momento in cui le operazioni del backend vengono concluse.

  I diretti interessati sono rappresentati dal device che ha originato l'evento, a cui d'ora in avanti mi riferirò per brevità come "origine", e, solamente nel caso in cui la computazione dello stato debba avvenire su device, il relativo vicinato.

  Nel caso in cui la computazione debba avvenire su cloud, ciò che viene inviato all'origine è un ack di conferma dell'avvenuta computazione.

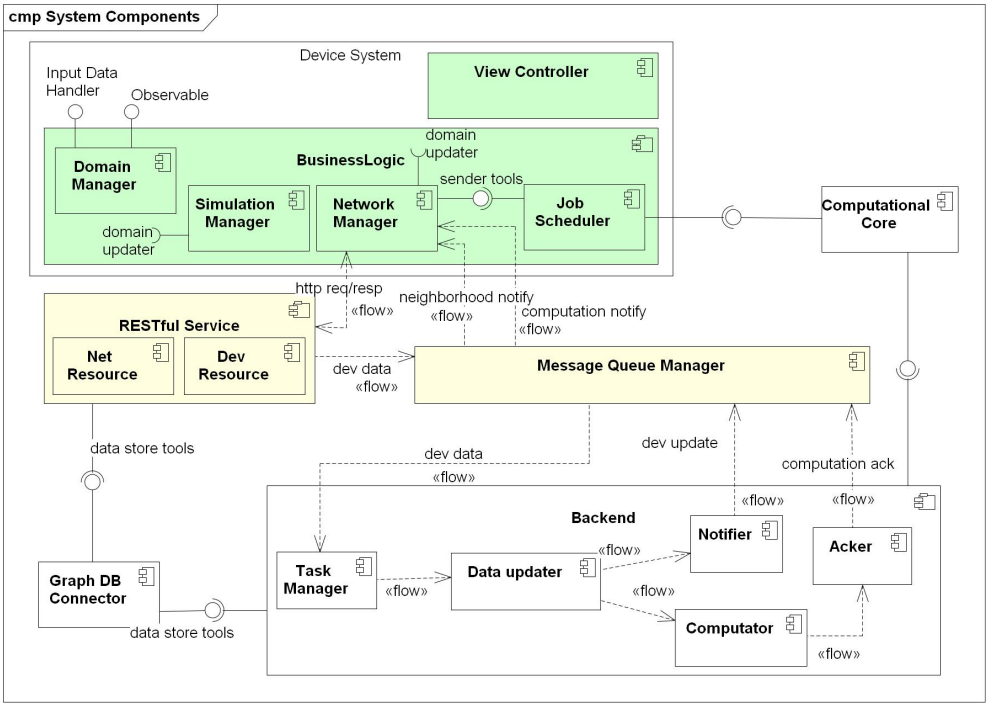


Figura 3.1: Componenti della infrastruttura

Nel caso in cui la computazione debba avvenire su device, all'origine viene segnalata la distanza aggiornata dai vicini mentre, simmetricamente, al vicinato vengono notificati i dati aggiornati riguardanti il suo stato e la distanza.

3.3.2 Iterazione tra le componenti

Le interazioni tra le componenti sono basate sull'invio di dati e su una risposta rappresentata da un semplice ack senza il contenuto informativo inerente al risultato da restituire, che potrebbe non sempre essere ottenibile in tempi rapidi.

La figura 3.2 mostra l'ordine delle interazioni ad alto livello delle componenti.

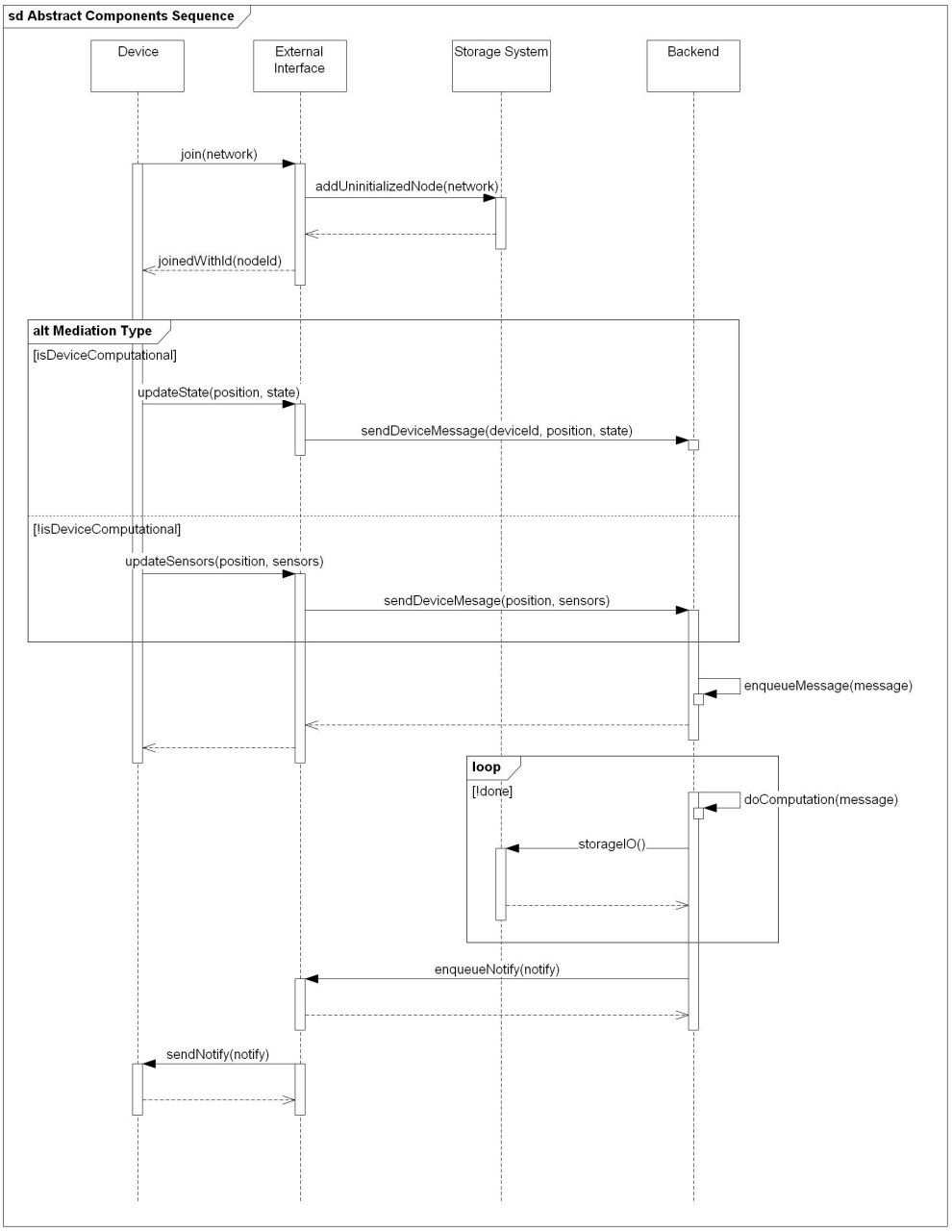


Figura 3.2: Sequenza delle interazioni

3.3.3 Tecniche di comunicazione

È stato deciso che gli scambi di dati tra tutte le componenti debba avvenire sotto forma di oggetti JSON. Vi sono diversi punti a favore di tale scelta.

Le librerie per la manipolazione di questa tipologia di dato sono facilmente reperibili per ogni linguaggio, l'overhead aggiunto dalla sua struttura è quasi trascurabile e il parsing e la costruzione dei messaggi ha un impatto computazionale minimo anche per device poco prestanti.

Inoltre tale formato ben si presta a una eventuale espansione dell'infrastruttura: sarebbe possibile aggiungere campi senza compromettere la struttura esistente dei pacchetti per ampliare i servizi offerti mantenendo allo stesso tempo una retrocompatibilità.

L'uso di un formato già esistente, ben collaudato e con una rappresentazione dei dati umanamente leggibile permetterebbe a persone estranee di potersi interfacciare con la struttura esistente senza doversi scontrare con un formato costruito ad hoc.

3.3.4 Distribuzione della computazione

Al fine di distribuire la computazione si rende necessario suddividere ogni operazione in un numero più ampio possibile di sotto operazioni.

Ciò di cui si deve occupare il backend è memorizzare i nuovi valori, eseguire eventualmente una computazione e restituire il risultato. Queste operazioni sono state scomposte in:

- Parsing del messaggio e suddivisione dei dati.
- Memorizzazione dei dati relativi allo stato.
- Memorizzazione dei dati relativi ai sensori.
- Memorizzazione dei dati relativi alla posizione e aggiornamento della lista dei nodi facenti parte del vicinato.
- Computazione basata sullo stato del vicinato e sul valore dei sensori.
- Nel caso in cui la computazione avvenga in cloud, l'accodamento della notifica, destinata all'origine, di avvenuta computazione.

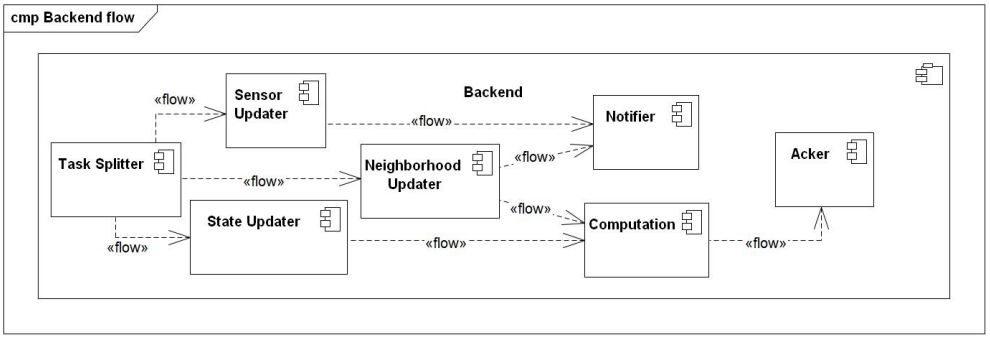


Figura 3.3: Schema del flow dei messaggi

- Nel caso in cui la computazione avvenga su dispositivo, l'accodamento delle notifiche, destinate al vicinato, contenenti il nuovo stato dell'origine e la distanza aggiornata; simmetricamente, destinate all'origine, le notifiche riguardanti i dati inerenti al vicinato.

Considerate queste operazioni si è ipotizzato un flow dei messaggi come illustrato nella figura 3.3

Capitolo 4

Progettazione

In questo capitolo viene esposto come è avvenuta la progettazione dell'infrastruttura, la scelta delle tecnologie da utilizzare e i metodi di interazione tra le componenti.

La presentazione segue il percorso dei messaggi inviati dal device. La prima parte riguarda l'interfaccia esterna, che costituisce il primo step del flusso informativo, la seconda riguarda il backend mentre l'ultima riguarda il broker dei messaggi.

4.1 Interfaccia esterna

L'interfaccia esterna esposta ai device è stata progettata come servizio web HTTP. Considerata l'analisi precedentemente effettuata tale scelta ottempera al requisito di semplicità e ha come intrinseca natura l'interazione basata su ricezione e conferma dell'avvenuta memorizzazione.

In particolare il servizio presenta una struttura RESTful.

4.1.1 Servizio REST

L'architettura del servizio REST si basa sul concetto di risorse a cui cui è possibile, tramite opportune richieste GET e POST, accedervi e modificarne il contenuto.

REST è l'acronimo di REpresentational State Transfer, termine coniato da Roy Fielding. Si definisce RESTful un sistema che rende disponibile, come servizio web HTTP, un insieme di risorse singolarmente raggiungibili

attraverso uno specifico url. Tratti fondamentali di un servizio RESTful sono l'architettura client–server, l'assenza di uno stato inerente alle interazioni tra client e server, la possibilità di eseguire, se esplicitamente dichiarato dal server, caching lato client e una struttura a livelli.

Le operazioni eseguibili su un determinato servizio devono essere ben definite, così come devono esserlo i contenuti e le tipologie di risposta ottenibili.

Le architetture REST si contrappongono a quelle RPC, acronimo di Remote Procedure Call, dove le richieste sono rappresentate dalla esplicita definizione di una procedura a cui vengono passati parametri. Al contrario REST vede ogni operazione come accesso, creazione o modifica di una una risorsa [13].

Di seguito la struttura del servizio realizzato.

- /nets risponde alle richieste GET e POST.

  Quando viene effettuata una richiesta POST il servizio crea una nuova rete. Nel corpo della richiesta possono essere incluse delle proprietà che verranno memorizzate sul database.

  Quando viene effettuata una richiesta GET il servizio restituisce la lista delle reti sotto forma di array json. Tale lista è formata da oggetti contenenti il "netId" e le proprietà della rete come definite in fase di creazione.

- /nets/netId risponde unicamente a richieste GET.

  Il servizio restituisce le proprietà della rete come oggetto JSON.

- /nets/netId/devs risponde alle richieste GET e POST.

  Quando viene effettuata una richiesta POST il servizio inserisce nel database un record relativo a un device e restituisce un identificatore univoco per quella rete. Si parlerà a breve di questa procedura nella descrizione della fase di join.

  Quando viene effettuata una richiesta GET il servizio restituisce una array di interi che rappresentano gli identificatori dei device contenuti nella rete. Insieme agli ID non viene restituito nessun altro dato inerente ai device.

- /nets/netId/devs/devId/position risponde alle richieste GET.

  Il servizio restituisce un oggetto contenente latitudine e longitudine del dispositivo. Questo valore è sempre disponibile a patto che il device abbia completato la procedura di join.

- /nets/netId/devs/devId/sensors risponde alle richieste GET.

  Il servizio restituisce un oggetto contenente il valore dei sensori del dispositivo. Questo dato è disponibile nel caso in cui il device non computi il suo stato in locale. Se è il cloud a dover eseguire la computazione allora il valore dei sensori deve essere inviato e quindi sarà reso disponibile a questo url.

- /nets/netId/devs/devId/state risponde alle richieste GET.

  Il servizio restituisce un oggetto contenente il valore dello stato del dispositivo. Questo dato è disponibile sia nel caso in cui il device computi il suo stato in locale sia in quello in cui sia il cloud a eseguire la computazione.

- /nets/netId/devs/devId/neighbors risponde alle richieste GET.

  Il servizio restituisce un array di oggetti che includono l'identificatore del vicino e la sua distanza.

- /nets/netId/devs/devId/msg risponde alle richieste POST.

  Il servizio memorizza il messaggio in modo che venga processato dal backend. Si parlerà di questa procedura a breve.

Join

Un device esegue l'accesso seguendo una semplice procedura di join.

Inizialmente viene inviata una richiesta POST verso la risorsa /nodes della rete prescelta. Il servizio web, tramite un apposito connettore, si interfaccia con un database in cui va ad aggiungere un record. Questo permette al servizio di ottenere un identificatore unico e inutilizzato da fornire in risposta al device.

Inoltre, sempre nell'ambito di questa operazione, viene inizializzata la struttura destinata a contenere i dati di quel dispositivo, il cui stato viene memorizzato come "uninitialized".

Da quel momento in poi il device si identificherà al servizio web attraverso l'invio del codice ricevuto.

Invio dati

A seconda della tipologia i device inviano una delle seguenti combinazioni di dato:

- Posizione e stato: rappresenta il caso in cui è il device mobile a computare il risultato della funzione iniettata.

  È stato già discusso riguardo alla necessità di trattare lo stato come un valore opaco in quanto il suo contenuto può variare a seconda dell'implementazione della funzione iniettata. A tal fine esso viene gestito come una stringa. Se si presentasse la necessità di memorizzare dati complessi sarebbe comunque possibile trattarli come stringa in base64.

- Posizione e sensori: rappresenta il caso in cui è il cloud a computare il risultato della funzione iniettata.

  I sensori sono gestiti come un oggetto JSON interpretato come mappa in cui la chiave è il nome del sensore. I valori, esattamente come nel caso dello stato, sono trattati come dato opaco. Se si presentasse la necessità di memorizzare il valore di un sensore come dato binario sarebbe sufficiente memorizzare il dato sotto forma di stringa in base64.

L'invio avviene eseguendo una richiesta POST indirizzata alla risorsa /msg del nodo. Se il device non aveva mai inviato dati relativi alla sua posizione allora il suo stato verrà modificato, dalla componente di backend, da "uninitialized" a "initialized". Questa modifica rende il dispositivo visibile al vicinato.

Il pacchetto ricevuto dal device viene memorizzato in una coda di messaggi che opera da ponte tra il servizio web e il backend computazionale.

Tale coda permette un disaccoppiamento tra le specifiche implementazioni delle due componenti, oltre a permettere la restituzione di una conferma di avvenuta ricezione non appena il messaggio viene considerato accettato dal broker: come esporrò a breve sarà questo elemento a conseguire l'obiettivo di garantire la computazione anche a seguito di guasti.

4.2 Computazione degli eventi: Apache Storm

La componente di backend è basata su Apache Storm. Come già esposto questa tecnologia permette ottime prestazioni, distribuibilità della computazione e fault–tolerancy [4].

La scelta tra le tecnologie presenti sul mercato è ricaduta su questa per via del fatto di essere open source e gratuita, ben documentata, continuamente aggiornata dalla comunità, performante, facile da installare, configurare e mandare in esecuzione. A questo va aggiunto il supporto per l'interazione con la tecnologia utilizzata per implementare la coda di messaggi che, come accennato nel capitolo di presentazione, appartiene anch'essa all'ecosistema Apache.

In questa sezione viene brevemente esposta la struttura interna di Storm e, parallelamente, viene descritto come sono stati utilizzati gli strumenti messi a disposizione dalla piattaforma al fine di ottenere il comportamento richiesto.

4.2.1 Potenzialità offerte

Come già esposto nel capitolo di panoramica, Storm permette la la computazione di una serie di eventi in maniera completamente distribuita.

Al fine di poter eseguire una topologia su un cluster si rende necessario che tutte le macchine che lo compongono possano comunicare tra di loro ed essere in grado di eseguire l'applicativo principale, cioè possedere una Java Virtual Machine. Non vi è alcuna necessità che le macchine eseguano lo stesso sistema operativo o che posseggano caratteristiche hardware simili, anche se questo è ovviamente consigliabile.

Una volta mandato in esecuzione il cluster provvede a tutte le operazioni di gestione interna, tra cui il recupero nel caso di errori: teoricamente non è necessaria la supervisione continua da parte di un amministratore.

4.2.2 Deploy di una topologia

All'interno della infrastruttura deve essere selezionata una macchina in cui eseguire il controller del cluster, chiamato "nimbus", che si occupa del setup delle topologie, di gestire la suddivisione del carico e il ripristino degli slave. Non vi è alcun requisito particolare riguardo alle caratteristiche della

macchina su cui mandare in esecuzione nimbus, nemmeno in termini di potenza, in quanto esso consuma una quantità di risorse computazionali molto limitate.

Insieme al nimbus è possibile mandare in esecuzione una componente "ui" che si occupa di offrire un pannello di controllo web a cui è possibile accedere per monitorare lo stato della rete, chiudere le reti, eseguire una operazione di ribilanciamento, visualizzare statistiche sul numero dei dati computati o visualizzare lo stack trace delle eccezioni.

Su ogni macchina del cluster deve essere eseguito uno "slave" che si occupa di attendere le configurazioni delle topologie da lanciare e mandarle in esecuzione secondo le disposizioni del nimbus.

Una volta conclusa questa parte di configurazione del cluster il deploy di una topologia risulta estremamente semplice: è sufficiente indicare un jar contenente il codice e le risorse da utilizzare.

Concretamente il deploy si traduce nei seguenti passi:

- In primis viene caricato un pacchetto jar. Al suo interno deve essere presente un classe con un entry point "main". Il codice eseguito richiamando il main deve, programmaticamente, definire la topologia e le configurazioni da trasferire su ogni nodo.
- Una volta che tale topologia è stata definita il nimbus trasferisce il pacchetto jar su tutti gli slave, insieme alle configurazioni.
- Al termine dell'upload vengono creati i nodi della topologia sotto forma di istanze di diverse classi, come definito dal main. Ognuna di esse rappresenta un task e non condivide la propria memoria, incluso il valore dei campi, con gli altri task. A tal fine risulta impossibile l'uso di campi statici se non per la definizione di costanti.
- I thread istanziati provvedono a richiamare il metodo prepare di ogni task per poi mettersi in attesa di un dato da computare.

4.2.3 Struttura interna

L'interazione tra le componenti di Storm si basa sullo scambio di messaggi sotto forma di tuple. L'ordine in cui i campi appaiono in ogni tupla scambiata è definito nella fase di preparazione e ogni campo è identificato da un nome. I dati contenuti nelle tuple possono essere di una qualsiasi classe

serializzabile. Storm offre nativamente la possibilità di serializzare i dati primitivi, che rappresentano la tipologia di dato più utilizzata.

Il tipo dei campi non viene definito in alcuna maniera, per cui a runtime è possibile usare tipi di dato differenti a seconda del contesto purché sia stato definito un opportuno serializzatore.

A ogni tupla è legato un valore identificativo (intero a 64 bit) utilizzato per tracciare i suoi movimenti internamente al sistema e per ripetere la computazione in caso di fallimento.

Considerato che Storm mette a disposizione la tupla come concetto alla base dello scambio di dati, i campi dei pacchetti JSON ricevuti sono suddivisi in modo tale da poterli trasmettere selettivamente ai task corretti attraverso tuple diversamente strutturate.

Spout

Gli Spout rappresentano il punto di ingresso delle tuple all'interno di Storm.

Il loro compito consiste nel raccogliere/ricevere eventi, solitamente omogenei tra di loro, eventualmente trasformarli e filtrarli, ed emetterli sotto forma di tuple.

In un sistema possono esistere più spout e ognuno di essi può a sua volta essere composto da più flussi di controllo, eventualmente distribuiti su più macchine, rendendo quindi possibile emettere più tuple contemporaneamente.

In fase di progettazione della componente di backend è stata individuata la necessità di definire un solo spout che si occupa della lettura dei messaggi dalla coda.

Bolt

Le tuple emesse vengono consegnate a uno o più bolt. Queste componenti eseguono una computazione ed emettono ulteriori tuple che verranno ricevute da altri bolt. È buona pratica trovare un tradeoff tra la quantà di computazioni eseguite da un bolt e il numero di essi utilizzati per ottenere un certo obiettivo. A seconda della infrastruttura di rete e delle caratteristiche hardware delle macchine, un numero di bolt elevato, ognuno dei quali programmato per eseguire una minima computazione, può essere preferibile o meno a un numero di bolt più ristretto ma che aggrega più operazioni.

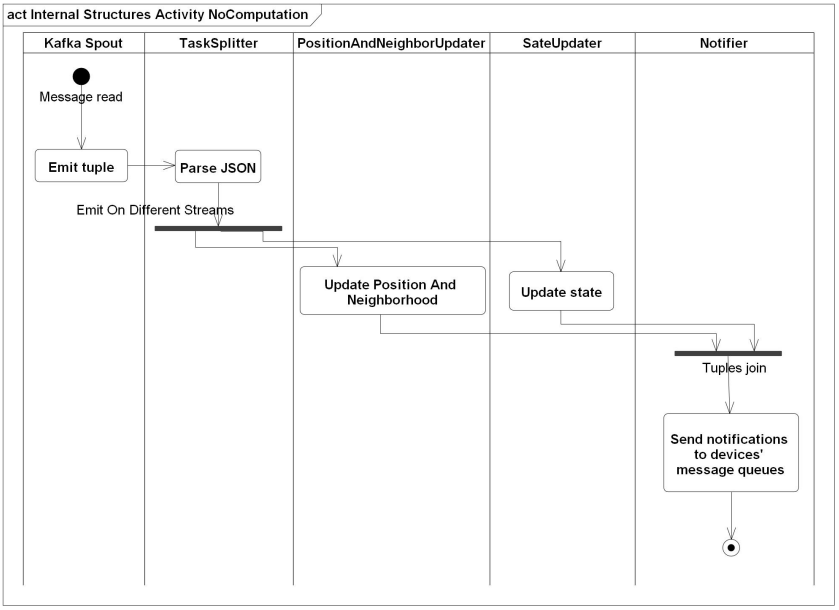


Figura 4.1: Gestione del flusso nel caso di computazione su device

Le operazioni da eseguire, definite in fase di analisi, trovano una diretta rappresentazione in un bolt. La suddivisione del lavoro in piccole operazioni permette di sfruttare al meglio le potenzialità offerte da un cluster.

In figura 4.1 è esposto il flusso di gestione nel caso in cui è il device a fornire lo stato aggiornato mentre in figura 4.2 è esposto il caso in cui è il cloud a doverlo calcolare utilizzando la funzione iniettata.

Streaming e grouping

In fase di progettazione di un sistema basato su Storm è necessario definire come le tuple debbano essere trasmesse per ottenere il risultato desiderato. Ogni qual volta che uno spout o un bolt emette una tupla questa viene posta in uno stream. Vari bolt possono sottoscrivere a un dato stream, per cui ogni bolt riceve una esatta copia di quella tupla.

Per ogni bolt possono esistere più stream di output e per ognuno di essi la definizione dei dati contenuti nella tupla è differente. Storm mette sempre implicitamente a disposizione uno stream, “default“, che spesso rappresenta l’unico utilizzato.

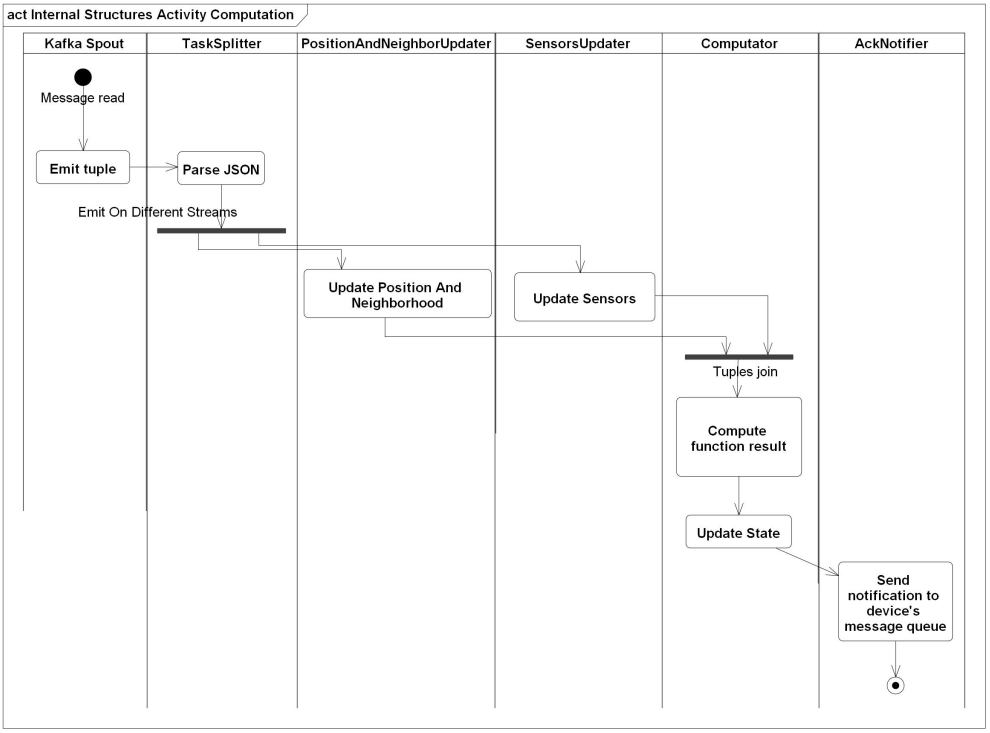


Figura 4.2: Gestione del flusso nel caso di computazione sul cloud

Grazie a questo meccanismo è possibile definire un DAG dove i nodi sono rappresentati dai bolt e gli archi dagli stream.

Ogni tupla ricevuta raggiunge uno solo dei task dei bolt a cui è destinata. Il comportamento predefinito è di smistarle in modo da suddividere equamente il carico tra tutti i task. Tuttavia spesso è desiderabile inviare le tuple che condividono caratteristiche comuni a uno solo di essi. Questo può essere necessario ai fini di migliorare le prestazioni con del caching, per eseguire join degli output o per evitare delle corse critiche.

A tal fine Storm prevede più meccanismi di grouping. Di seguito i più utilizzati:

- Shuffle Grouping: il grouping predefinito, distribuisce equamente il carico tra tutti i task.
- Fields Grouping: le tuple sono suddivise in base al valore di uno o più campi. Tutte le tuple con una certa accoppiata di campi vengono sempre inviate a un certo task.
- Global Grouping: l'intero stream è inviato al task con id più basso.
- Local Grouping: se un task del bolt destinatario è contenuto nello stesso worker allora le tuple vengono inviate ad esso. Se tale task non è presente allora questo metodo si comporta come il Shuffle Grouping.

Nell'ambito della progettazione è stato ritenuto necessario far uso alternativamente di Shuffle e Fields grouping a seconda del contesto. Questa parte verrà discussa più approfonditamente nel capitolo riguardante lo Sviluppo.

4.3 Storage dei dati: Neo4j

Considerata la struttura a tuple di Storm è stata ritenuta una soluzione onerosa e poco praticabile quella di scambiare i dati tra bolt sotto forma di tuple multiple, specialmente quelli relativi alla lista e le proprietà dei device vicini.

Appare evidente la necessità di disporre di un sistema di memorizzazione accessibile, oltre che dai bolt del cluster Storm, anche da parte del servizio web. In particolare quest'ultimo che deve anche occuparsi di gestire

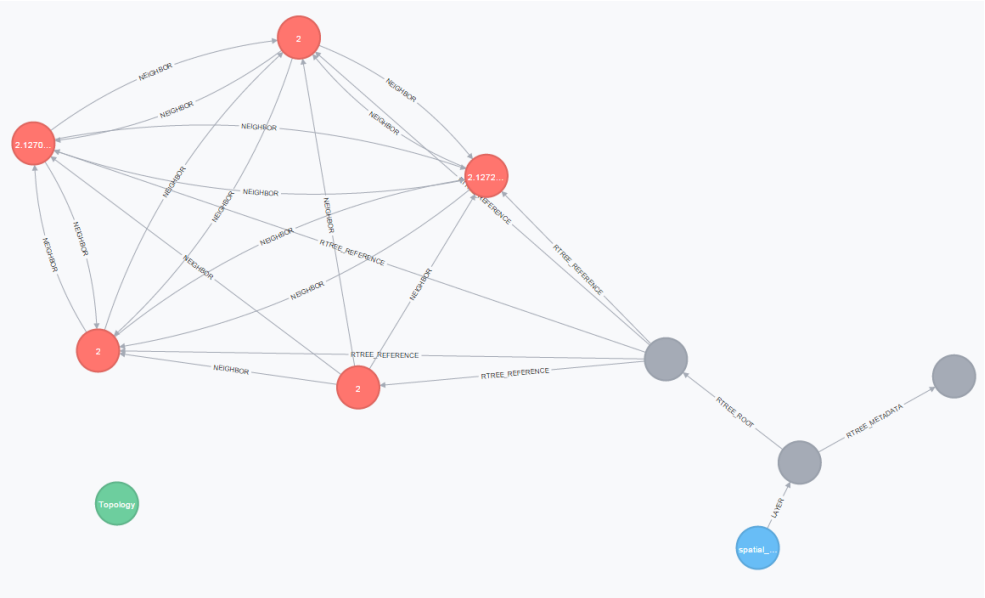


Figura 4.3: Vista di una semplice rete in Neo4j

la procedura di join e questo, come precedentemente descritto, prevede la creazione immediata di un record relativo al dispositivo.

Tra le soluzioni presenti sul mercato è stato scelto Neo4j, un database NoSql basato su grafi. In questa sezione vado ad esporre le sue caratteristiche e il modo in cui è utilizzato all'interno della infrastruttura.

4.3.1 Un database basato sui grafi

Neo4j è stato scelto per via della struttura a grafi che ben rispecchia l'ambito in cui deve operare l'infrastruttura. È possibile memorizzare dati sia all'interno dei nodi sia all'interno degli archi. I dati così memorizzati vengono letti e scritti come oggetti JSON per cui è impossibile memorizzare dati binari se non codificandoli, ad esempio, in base64.

È possibile indicare un metodo per indicizzare ed etichettare i nodi e dare un nome alle relazioni, cioè agli archi. Grazie a questa rappresentazione dei dati è possibile con poco sforzo tradurre uno schema Entity Relationship in un equivalente sistema basato su grafi [11] [10].

Il vicinato viene gestito come nodi connessi da archi non diretti. Nel momento in cui la posizione di un device viene aggiornata, la proprietà "distance" degli archi collegati al relativo nodo è aggiornata per rispecchiare la nuova distanza tra i due device connessi. Quando un dispositivo viene valutato fuori dal raggio di visibilità allora l'arco viene eliminato. Tutte le distanze sono memorizzate come metri.

4.3.2 Accesso al database via REST

Un'ulteriore feature che ha portato alla scelta di Neo4j come sistema di memorizzazione dei dati è la possibilità di utilizzare una interfaccia REST, messa a disposizione nativamente dal database, per l'accesso, sia in lettura che in scrittura, ai dati. Se desiderato tale interfaccia web mette a disposizione il sistema di autenticazione "Basic", basato sull'invio di username e password all'interno degli header HTTP.

Tale API REST ha permesso di implementare in breve tempo una libreria di interfacciamento utilizzata sia dalla componente di backend che dal servizio web. A tal fine la libreria è stata trattata come progetto indipendente.

Neo4j mette a disposizione due metodi per l'accesso ai dati:

- Il primo si basa sulla query diretta di valori utilizzando un sistema a risorse.

  Questo metodo rappresenta il vero metodo di accesso RESTful messo a disposizione: le risorse come nodi e archi sono accessibili fornendo, all'interno dell'url della richiesta, l'id univoco che individua l'entità all'interno dell'intero database.

  Tale metodo ha come grande vantaggio la semplicità di utilizzo ma presenta diversi svantaggi. Non è infatti possibile definire quali dati restituire come risultato della query: come risposta vengono inviati valori relativi a metadati, la lista dei vicini, tutti i campi delle entità considerate, url interrogabili per navigare nel grafo, ecc.

  Questo va a formare un messaggio di risposta di dimensioni considerevoli. Tale pacchetto dovrà essere inviato su una connessione TCP e poi interpretato da una libreria per la manipolazione di dati JSON per cui si tratta di un metodo piuttosto inefficiente. La prima versione della libreria utilizzava questo metodo per l'accesso ai dati.

- Il secondo metodo si basa sull'invio di una query.

  Come precedentemente accennato Neo4j è un database NoSQL e prevede l'uso di un linguaggio studiato ad–hoc per operare con i grafi chiamato Cypher. Al fine di poter utilizzare questo potente linguaggio il servizio web offerto dal database espone un "Transactional Cypher HTTP endpoint".

  Esso consiste in un url fissato a cui è possibile inviare, tramite a una richiesta POST, uno o più statement che verranno eseguiti in una transazione.

  Il metodo di utilizzo più semplice prevede l'uso di una unica richiesta per definire una lista di statement da eseguire in serie. Neo4j permette inoltre di eseguire più statement in una stessa transazione ma su richieste differenti utilizzando un sistema di commit e rollback basato su chiamate HTTP.

La versione finale della libreria ha fatto uso di questo ultimo metodo che permette, a livello di query, di filtrare i dati ritornati. Questo riduce enormemente lo scambio di dati nella rete mantenendo invariato il contenuto informativo utilizzato.

Neo4j non possiede un metodo per utilizzare delle stored procedures. Al suo posto è prevista una parametrizzazione delle query inviate: il pacchetto JSON contentente gli statement prevede un campo contenente i parametri utilizzati mentre all'interno delle query tali parametri sono indicati racchiudendo tra parentesi graffe il loro nome. Query identiche e parametrizzate portano Neo4j ad eseguire ottimizzazioni nella loro gestione.

I dati ritornati dal server sono divisi per statement. Ogni statement, compresi quelli di modifica dei dati in cui è presente una istruzione di return, presenta infatti un risultato e una eventuale lista di errori incontrati.

Appare evidente come il "Transactional Cypher HTTP endpoint", seppur considerato parte della "REST API" di Neo4j, non consiste in un metodo di accesso RESTful, ma bensì RPC. Ciò che viene considerato RESTful è la gestione delle trasazioni su più richieste: ogni transazione è rappresentata da un id univoco e ogni interazione è basata su richieste verso un url che include tale identificatore.

4.3.3 Neo4j Spatial

Neo4j presenta grandi possibilità in termini di personalizzazione del comportamento tramite plugin. La comunità intorno a questo database ne ha costruiti diversi per varie applicazioni tra cui inserimento in batch a partire da un certo formato di dato, eliminazione automatica di dati, sincronizzazione con un altro database, esposizione di una interfaccia grafica alternativa, ecc.

Al fine di trattare i dati relativi alle posizioni dei device è stato scelto di utilizzare il plugin Neo4j Spatial. Esso permette l'inserimento di geometrie (punti e poligoni) per poterne poi eseguire delle interrogazioni quali WithinDistance, Touch, Overlap, Contain, ecc.

Il plugin permette di disporre contemporaneamente, all'interno dello stesso database, di informazioni differenti organizzando i dati sotto forma di layer.

Un layer è definito dal nome delle proprietà da trattare come coordinate e le possibili geometrie contenute. Al fine di mantenere separati i dati tra ogni layer, è previsto un sistema di RTree distinti, utilizzati per l'indicizzazione. I dati relativi ai device di una rete sono memorizzati in un unico layer.

Nella costruzione della libreria è stato utilizzata unicamente la geometria SimplePoint. Tale geometria prevede la presenza dei campi lon e lat (nomi configurabili) che devono essere valori a virgola mobile di coordinate presenti sul globo. Il plugin si occupa di mantenere l'indice aggiornato al variare di queste proprietà.

Al fine di ottenere la nuova lista di vicini è stata utilizzata la interrogazione WithinDistance che prende come parametro il centro e il raggio di ricerca dei device.

Una volta ottenuti tali device vengono creati o aggiornati degli archi contenenti la proprietà "distance" correttamente impostata. Tale distanza deve essere calcolata lato client in quanto non viene restituita dal plugin.

Dalla figura 4.3 è possibile vedere una minimale rete così come memorizzata all'interno del database. Nell'immagine sono visibili i nodi che rappresentano i device, in questo caso tutti considerati come vicini tra di loro, gli archi che rappresentano le relazioni di vicinato e i nodi supplementari creati dal plugin per l'indicizzazione su RTree. A questo si aggiunge un nodo che rappresenta il record contenente le proprietà della rete.

4.4 Broker Kafka

Il broker di messaggi Kafka è stato utilizzato sia nella fase di gestione della ricezione dei messaggi sia nella fase di restituzione del risultato.

Come già accennato nel capitolo di panoramica, Kafka permette di trattare dati di diversa tipologia all'interno dello stesso cluster. In questa sezione viene esposta la struttura del cluster da noi progettata.

4.4.1 Analisi della struttura

Kafka suddivide i messaggi in topic, identificati da un nome, e ogni topic in partizioni, identificate da un id progressivo.

Un topic raggruppa messaggi dello stesso tipo e prevede la possibilità di fornire una configurazione ad–hoc potenzialmente differente da quella impostata per altri topic. Le configurazioni di default sono memorizzate in un file di configurazione del broker e sono utilizzate se non espressamente definite in fase di creazione di un nuovo topic. È comunque possibile modificare questi valori nel corso della vita del topic, pagandone il prezzo sotto forma di un tempo di riconfigurazione e ribilanciamento che varia in base alla quantità di log memorizzati e alla precedente configurazione.

Le configurazioni riguardano il tempo per cui devono rimanere memorizzati i messaggi, il numero di partizioni, il numero di repliche, la politica di flush dei dati su disco, i timeout per verificare la coerenza dei log nel cluster, ecc.

Topic e partizioni

Un producer può inviare messaggi a un certo topic e a una certa partizione. Tale messaggio viene memorizzato sotto forma di log appendendone il contenuto a un file sul file system. Tale file rappresenta un partizione. Un consumer può leggere messaggi da un topic e partizione arbitrari.

Può esistere un solo consumer in lettura di una certa partizione e più consumer possono leggere da partizioni differenti dello stesso topic, per cui il livello di parallelizzazione ottenibile nella lettura dei messaggi è limitato dal numero di partizioni definite per quel topic. Per ogni consumer il broker tiene traccia dell'ultimo offset letto e dell'ultimo confermato in modo che sia possibile accorgersi, anche a seguito di disconnessioni forzate o guasti da parte di uno dei due attori, a quale messaggio si era arrivati.

In linea con la metodologia di memorizzazione basata su append, un messaggio all'interno della coda è identificato dal suo offset, cioè dalla sua posizione interna alla partizione.

Replica e master

Kafka offre un meccanismo di replica per evitare che, in seguito a guasti, i contenuti vengano persi o non risultino disponibili. Il numero di repliche effettuate per un certo topic deve essere definito nella fase di creazione del topic come parametro di configurazione.

Quando viene creato un topic il broker decide internamente a quali macchine distribuire le copie principali e le repliche: per ogni partizione è sempre presente un "master" che si occupa di gestirne le richieste di lettura e scrittura. Gli slave, in numero pari al valore indicato in fase di definizione delle repliche, copieranno i messaggi presenti nella copia del master in una copia locale. Ogni macchina è master per certe partizioni e slave per altre partizioni per cui una istanza del broker può trovarsi a dover gestire partizioni di topic diversi, alcune in modalità slave e altre in modalità master. Il broker tenta di suddividere le copie master uniformemente tra le macchine che compongono il cluster.

Nel momento in cui, in seguito a un guasto, il master di una certa partizione non appare più disponibile si attiva l'elezione del master tra le macchine slave. Una volta ripristinato dal guasto un broker ritenta la sincronizzazione con il cluster a partire dai dati precedentemente memorizzati evitando di dover riclonare l'intero contenuto delle partizioni.

Appare da subito evidente come sia possibile costruire una infrastruttura con alcune caratteristiche di fault–tolerantcy a partire da una coda di messaggi Kafka ben configurata.

4.4.2 Restituzione dei risultati

Si sono già descritte le motivazioni che hanno portato a scegliere di restituire la notifica di avvenuta computazione al device attraverso una coda di messaggi. Tale coda potrebbe essere implementata differentemente rispetto alla coda utilizzata internamente per scambiare dati tra il servizio REST e il backend. Ai fini di accodare notifiche in uscita sarebbe infatti sufficiente una semplice coda che permetta letture e scritture in serie senza pretese in

termini di performance in quanto il dispositivo leggerà tali dati attraverso una connessione di rete in maniera seriale.

Per via della facile configurabilità e semplicità di interfacciamento è stato scelto di utilizzare la stessa tecnologia della coda interna, cioè Kafka, in quanto rappresenta una soluzione ottimale anche per questo caso.

4.5 Suddivisione e contenuto dei topic

I dati trattati dalla coda dei messaggi sono di due tipi: eventi in entrata e notifiche in uscita. Per queste due categorie sono state adottate due politiche differenti di gestione che hanno portato a suddividere e nominare i topic nei seguenti due modi:

- Al fine di accodare i messaggi ricevuti dai device si è ritenuto opportuno utilizzare un solo topic suddiviso su un buon numero di partizioni in modo da ottenere buone performance in termini di throughput. Tale topic "msg" accoderà, come già precedentemente specificato, stringhe rappresentanti pacchetti JSON. Tale pacchetto dovrà contenere l'id della rete, l'id del device nella rete e il contenuto informativo inviato dal dispositivo mobile.

  La scelta di utilizzare un solo topic il cui nome è definito a priori ricade nella impossibilità pratica di conoscere il numero e l'id dei device che prenderanno parte alle reti, che a loro volta sono in numero e valore degli identificatori sconosciuto.

- Al fine di restituire le notifiche e i risultati ai dispositivi le notifiche vengono accodate su un topic specifico per ogni device, il cui nome è costruito secondo il pattern "dev.netId.devId".

  Tale dinamicità nell'uso di più topic e nomi è data dalla possibilità di creazione e configurazione programmatica dei topic e alla conoscenza, da parte del backend, degli identificativi richiesti per ricostruirne i nomi.

Capitolo 5

Sviluppo

In questo capitolo viene descritto come sono state sviluppate le componenti della infrastruttura. L'esposizione inizia con la parte riguardante il servizio REST, che rappresenta il punto di ingresso delle informazioni all'interno del sistema, e si conclude con la restituzione dei risultati al device.

Viene inoltre brevemente illustrato il funzionamento del pannello di visualizzazione delle reti.

5.1 Strumenti utilizzati

L'infrastruttura è stata sviluppata interamente in linguaggio Java. Come IDE di sviluppo è stato utilizzato "Eclipse Mars for Java EE Developers" in cui è stato installato "M2Eclipse" che estende Ecplise in modo da integrare Maven. Tale sistema è stato utilizzato per gestire le numerose dipendenze e permettere un semplice packaging in jar degli elementi da deployare.

L'implementazione è stata suddivisa in progetti appartenenti allo stesso workspace. La directory contenente il workspace è stata posta sotto il sistema di controllo delle versioni Git con un repository ospitato da BitBucket. Tale repository è stato utilizzato sia da me che Brunetti utilizzando un sistema di branching suddivisi per progetto, periodicamente uniti al master per verificare l'integrazione delle componenti del sistema.

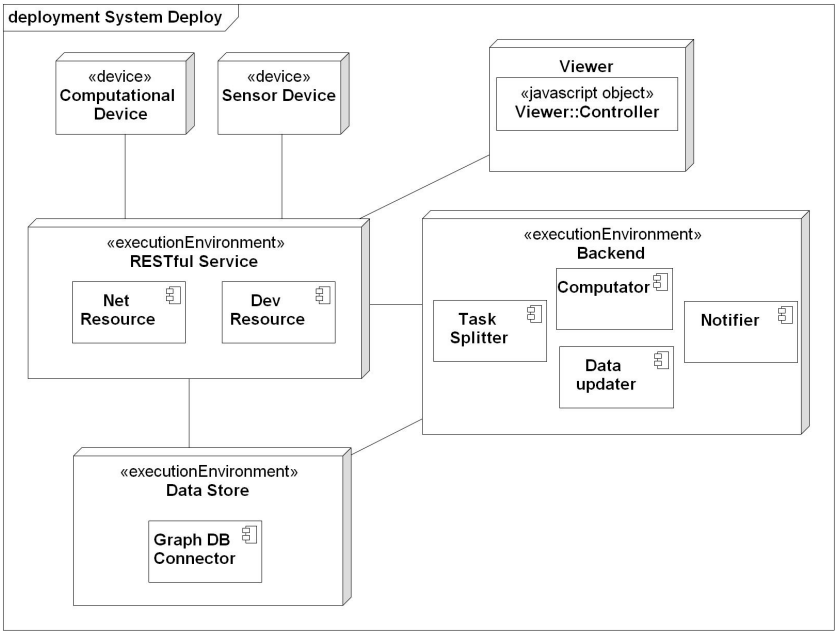
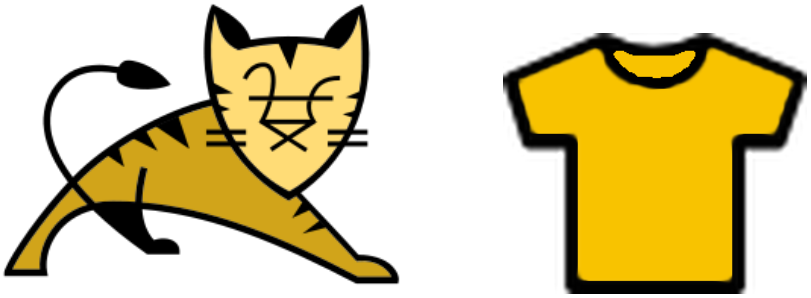


Figura 5.1: Schema di deploy



(a) Tomcat (b) Jersey

Figura 5.2: Le tecnologie utilizzate per il servizio REST

5.2 Servizio REST

L'ambiente di esecuzione scelto per il servizio REST è Apache Tomcat.

Tomcat è un server web open source che fornisce l'ambiente di esecuzione per applicazioni Java EE. Esso funge da loro gestore e contenitore per cui all'interno della stessa istanza di Tomcat possono essere mandati in esecuzione più servizi web.

Un servizio che necessita di essere mandato in esecuzione sul server deve essere compresso e inviato come file WAR, acronimo di Web ARchives, che contiene sia il codice del servizio sia le informazioni di esecuzione in un file di configurazione XML.

Al fine di facilitare la costruzione di una struttura REST è stato fatto uso del framework Jersey.

Jersey rappresenta la implementazione di riferimento delle API JAX–RS, contenute all'interno della specifica di Java EE, che fa uso di un sistema di annotazioni dei metodi e delle classi per definire la struttura del servizio REST in termini di percorsi delle risorse, metodi accettati, input atteso e tipo di dato restituito.

5.2.1 Comunicazione con il cluster Storm

Al fine di poter inviare dati al cluster Storm, dove avviene la gestione dei dati e la computazione dello stato, si è già descritta la presenza di una coda di messaggi posta tra il servizio web il backend.

Il servizio web utilizza le librerie fornite con Kafka in modo da impersonare un producer. Le richieste indirizzate a Tomcat possono essere gestite contemporaneamente senza provocare problemi di concorrenza e Kafka riesce ad gestire le richieste di scrittura anche se arrivate contemporaneamente. Da questo punto di vista l'interfaccia esterna fornita dal servizio REST rappresenta un sottile strato di astrazione che nasconde la gestione interna dei messaggi: ciò di cui deve preoccuparsi il servizio è rimbalzare i pacchetti al broker dei messaggi.

Grazie alla presenza di un servizio web posto tra i device e la coda dei messaggi sarebbe possibile, in futuro, implementare sistemi di controllo sui dati ricevuti e i device connessi quali filtri e autenticazione.

Come già descritto i messaggi ricevuti vengono scritti su un unico topic diviso in più partizioni. Tale numero di partizioni deve essere tale per

cui la lettura dei messaggi possa avvenire con la più alta parallelizzazione possibile. Il limite è posto dalle risorse dedicate a Kafka e dal numero di consumer allocati a Storm.

5.3 Il cluster Storm

In questa sezione si espone in dettaglio come sono state implementate le procedure interne al backend al fine di memorizzare le informazioni in arrivo dal device, eseguire la computazione e notificare i dispositivi in attesa di informazioni.

5.3.1 Kafka Spout

Il cluster Storm preleva i messaggi dal broker ed emette le stringhe lette, rappresentanti messaggi JSON, sotto forma di una unica tupla con un solo campo. Ad occuparsi di questa procedura è uno spout fornito come libreria sviluppata congiuntamente dagli sviluppatori di Kafka e Storm.

In fase di definizione della topologia, la libreria prevede che venga fornito il nome del topic da cui leggere. Il numero di partizioni da cui leggere contemporaneamente è dato dal numero più basso tra i task allocati allo spout e le partizioni trovate sul topic a runtime. Fare in modo che questi due numeri combacino è ovviamente un'ottima pratica, anche se non necessaria. Se il numero di partizioni trovate risultasse maggiore al numero di task allora uno o più task dovrebbero accollarsi la lettura di più partizioni; viceversa se tale numero risultasse minore allora alcuni task rimarrebbero fermi. Questo ultimo caso, se Storm è eseguito in modalità debug, viene fatto notare dai task bloccati con l'emissione di una linea di log.

5.3.2 Split dei campi

Una volta emessa, la stringa viene recapitata a un bolt che si occupa di interpretarla come oggetto JSON ed analizzarne i campi contenuti. Al fine di eseguire il parsing della stringa è stata utilizzata la libreria org.json.

Il pacchetto deve sempre contenere la posizione del dispositivo. Insieme a questo può essere alternativamente presente il dato riguardante lo stato o i sensori, a seconda se il dispositivo è computazionale o meno.

Questi dati devono essere inviati ai bolt che si occupano di gestire la memorizzazione delle singole parti. Per evitare di inviare l'intero contenuto del pacchetto a tutti i bolt, considerato che questi non utilizzerebbero tutti i campi contenuti, è stato fatto uso di più stream. Come già accennato uno stream è definito dalla sorgente da cui le tuple vengono emesse e dalla loro struttura.

Tutte le tuple emesse devono contenere i dati per identificare il dispositivo, cioè l'id della rete e del device. A questo si aggiungono i dati specifici: lo stream relativo all'aggiornamento dello stato conterrà lo stato, quello dei sensori il valore dei sensori, quello della posizione la latitudine e la longitudine.

Al fine di evitare aggiornamenti contemporanei sugli stessi dati è stato utilizzato un grouping basato sull'id della rete e del device. In questo modo tutte le tuple inerenti a un device vengono consegnate a un solo task dei bolt che si occupano di aggiornarne i vari dati.

5.3.3 Aggiornamento dei dati

I bolt che si occupano dell'aggiornamento dei dati sottoscrivono agli stream precedentemente definiti e ricevono le tuple contenenti i dati da memorizzare. A tal fine, in fase di configurazione della topologia, devono essere definiti i dati relativi al supporto di memorizzazione da utilizzare. Nel caso specifico, avendo fatto uso di Neo4j, questi consistono nell'url della interfaccia web del database e le eventuali credenziali da utilizzare per l'autenticazione Basic.

Mentre l'aggiornamento di stato e sensori è rappresentato da una unica semplice query, l'aggiornamento della posizione e successivamente del vicinato è invece composta da più operazioni. Ai fini della implementazione del relativo bolt questo non rappresenta un problema in quando tale complessità è stata incapsulata all'interno della libreria di interfacciamento con il database.

Una volta completate le operazioni, se queste hanno successo, viene emessa una tupla contenente l'id della rete e del device oltre a una flag utilizzata per indicare quale tipo di operazione è stata effettuata dal quel bolt.

Anche le tuple emesse da questi bolt vengono raccolte utilizzando il grouping per id di rete e device.

5.3.4 Bolt di computazione

La componente che si occupa di eseguire la computazione, implementata all'interno di un apposito bolt, deve attendere il completamento delle operazioni di aggiornamento della posizione, del vicinato e dei sensori. Una volta che tali operazioni sono state completate è necessario eseguire la lettura di tali dati dal database in quanto non vengono trasmessi all'interno delle tuple per le ragioni spiegate nella fase di progettazione.

L'attesa della tupla che segnala il completamento delle operazioni da parte di due bolt non è una operazione banale. Storm non prevede un meccanismo nativo per l'esecuzione di tale operazione di "join". Al fine di poter attendere l'arrivo di entrambe le tuple è ovviamente necessario che il task del bolt in loro attesa sia il medesimo, motivo per il quale le tuple sono raggruppate per id di rete e device.

Si è reso necessario fare affidamento a una struttura dati mantenuta in memoria in grado di mantenere l'informazione circa l'arrivo di un evento per un lasso di tempo ben definito. È necessario che l'evento di ricezione rimanga memorizzato per un tempo pari al limite di timeout di gestione di una tupla definito in fase di creazione della topologia. Infatti, se ci fosse un errore nella gestione della memorizzazione di un dato, il bolt in attesa potrebbe mantenere erroneamente memorizzato l'evento inerente al completamento dell'operazione di memorizzazione dell'altro dato. Il tempo di timeout è fornito da una valore, espresso in secondi, sempre presente nel bundle delle configurazioni di una topologia.

Al fine di gestire correttamente la fase di "join" è stato fatto uso della classe Cache contenuta all'interno della libreria Guava. Essa consiste in una mappa che si occupa di eliminare i dati dopo un certo timeout configurabile. Il timeout relativo a un dato è esprimibile rispetto al momento dell'ultimo accesso, dell'ultima modifica o entrambi. La pulizia dei dati obsoleti non viene effettuata da un thread lanciato dalla libreria ma bensì non appena viene eseguita una qualsiasi operazione sulla mappa e in modo da spalmare la operazioni necessarie lungo le chiamate ai suoi metodi.

Il bolt di computazione memorizza all'interno della cache una coppia chiave–valore in cui la chiave è rappresentata dall'id della rete e del dispositivo mentre il valore è la tupla ricevuta. La computazione viene lanciata non appena vengono ricevute entrambe.

Se avvenisse un errore durante la computazione oppure durante la me-

morizzazione dello stato restituito dalla funzione iniettata, allora le tuple precedentemente memorizzate vengono dichiarate come fallite in modo da innescare il sistema di replay degli eventi integrato in Storm.

5.3.5 Notifiers

Una volta completato l'aggiornamento dei dati ed effettuata l'eventuale computazione si rende necessario restituire il risultato ai device. Se la computazione della funzione avviene in cloud allora un bolt si occupa di inserire, nella coda dei messaggi assegnata al device che ha originato l'evento, la notifica di avvenuta computazione. Se invece la computazione avviene su device un differente bolt si occuperà di inserire, sia nella coda dei messaggi dell'origine sia in quella dei suoi vicini, le informazioni relative alle nuove distanze e agli stati dei dispositivi.

Resituzione dei risultati

Da questo deriva che i bolt addetti alla notifica utilizzino la libreria fornita con Kafka per impersonare un consumer. I topic utilizzati per la restituzione dei risultati sono creati dal servizio web all'atto dell'accesso del device a una rete per cui vi è la certezza di trovare il topic necessario già esistente.

Come già accennato il risultato è restituito utilizzando un topic per ogni device. È stato ritenuto opportuno utilizzare una sola partizione per ogni topic in quanto non vi è la necessità di alte performance in lettura e scrittura. Il device esegue continuamente un consumer Kafka al fine di ricevere i risultati in tempo reale seguendo la metodologia publish–subscrive; sarebbe comunque possibile costruire una implementazione del client in esecuzione sul device che disattivi il consumer e lo riattivi in modo da mimare una procedura di polling.

5.4 Device

I device si occupano di inviare i dati alla rete, ottenere i risultati, effettuare il rilevamento del valore dei sensori, compresa la lettura della posizione da un modulo GPS, ed eventualmente computare il proprio stato.

Mentre i dati dalla rete arrivano in maniera asincrona, tramite un task che si occupa di modificare i valori in memoria in base al messaggio ricevuto

dal cloud, il rilevamento dei sensori e l'eventuale computazione avviene in un ciclo.

5.4.1 Ciclo di computazione

All'inizio del ciclo viene ottenuta una copia dei dati ottenuti dalla rete. Questo lascia libero il task che si occupa di leggere le notifiche dalla rete di operare su una propria copia dei dati. Dopo aver effettuato tale copia vengono letti i valori dei sensori.

Dopo queste operazioni il dispositivo può presentare due comportamenti differenti:

- Se è un "computational device" lancia l'algoritmo contenuto nella funzione iniettata. Il risultato della computazione rappresenta il nuovo stato del dispositivo, che viene inviato al servizio web in un pacchetto contenente anche la posizione aggiornata.

- Se invece il device non è computazionale, cioè se la computazione deve avvenire in cloud, non viene eseguita alcuna ulteriore operazione e viene inviato un pacchetto contenente la posizione e il valore dei sensori.

5.4.2 Delay

Completate queste procedure il device attende un periodo di tempo prefissato prima di ricominciare il ciclo. Nel caso di un device non computazionale l'attesa inizia da quando viene ricevuta la notifica di completamento delle operazioni.

Per un device computazionale l'attesa inizia immediatamente e, anche nel caso in cui venissero ricevuti aggiornamenti inerenti al cambiamento di stato di un vicino, il ciclo non viene ricominciato prima che sia trascorso il lasso di tempo fissato. Questo accorgimento previene la possibilità che avvenga un invio di messaggi a cascata a ogni modifica nell'assetto del vicinato.

5.5 Viewer

Al fine di visualizzare la rete è stato costruito un pannello web che permette la sola lettura dei dati.

Il viewer, costruito secondo il principio MVC, è suddiviso in componenti:

- Una componente per il listing delle reti.

  Il suo compito è ottenere una lista delle reti disponibili e le loro proprietà. Una volta ottenute, esse vengono visualizzate come una lista di schede clickabili. All'interno di ogni scheda è visualizzato l'identificatore della rete, un eventuale nome (se presente) e una la lista delle proprietà.

- Una componente per il fetch dei dati dei dispositivi.

  Il suo compito è ottenere la lista dei nodi, verificare quali sono stati aggiunti e quali rimossi modificando il model di conseguenza, far partire le richieste per ottenere dati relativi ai nuovi nodi ed aggiornarli periodicamente.

- Una componente di visualizzazione del grafo.

  Tale componente, come esposto precedentemente, è costruita intorno alla libreria grafica Linkurious. L'uso di questa specifica libreria è incapsulato al suo interno e quindi si presta ad un eventuale un cambio di implementazione del motore grafico. Grazie a questa separazione è possibile eseguire del rendering personalizzato.

Una volta che l'utente ha selezionato una rete viene caricata una schermata in cui sarà possibile visualizzare i nodi con le relative relazioni di vicinanza. Contestualmente al cambio di vista viene mandato in esecuzione il relativo controller, che esegue il fetch dei dati dei dispositivi e aggiorna la schermata grafica richiamando i metodi della componente di view.

La componente di view modifica i dati memorizzati all'interno di Linkurious utilizzando le API esposte. È stato costruito un metodo per eseguire il rendering dei nodi e degli archi in base ad alcune loro proprietà. Questo deve essere implementato in base alla tipologia dei sensori trattati e dello stato. Ad esempio è stata fornita una modalità per visualizzare i nodi sorgente con un colore differente da quello degli altri nodi.

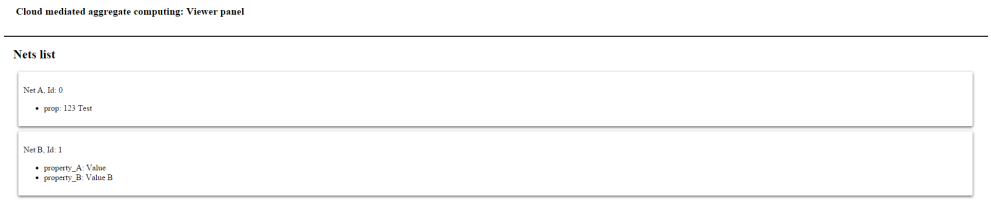


Figura 5.3: Viewer: lista delle reti

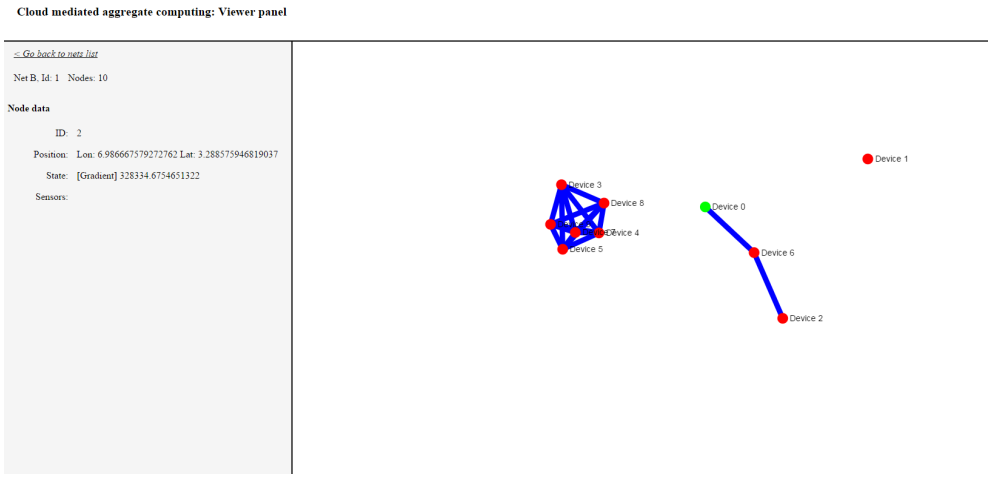


Figura 5.4: Viewer: vista di una rete e valori di un nodo

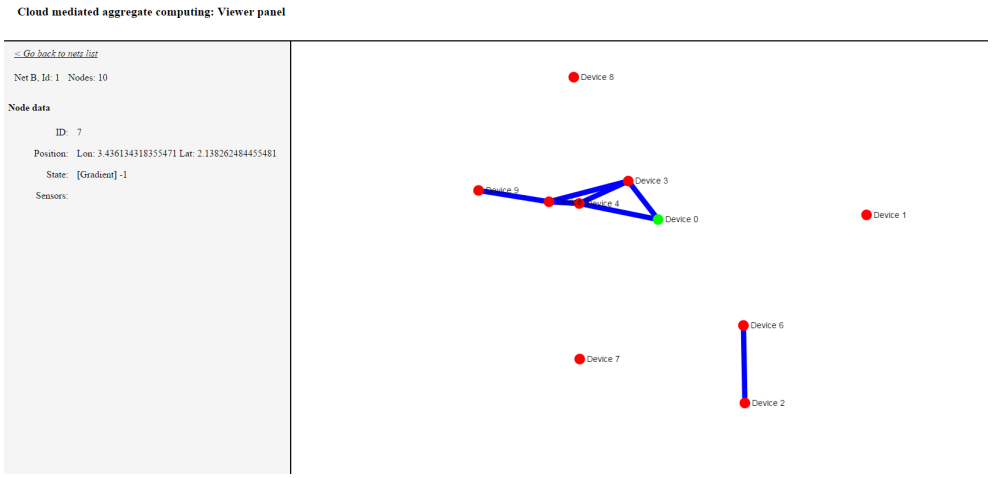


Figura 5.5: Viewer: nodo senza connessioni a una sorgente

L'aggiornamento dei dati riguardanti la lista dei nodi, valore dei sensori, dello stato e della posizione può avvenire a frequenze differenti per ognuno di questi dati.

La app esegue su un'unica pagina web e utilizza un sistema basato su hash per gestire la navigazione indietro e il refresh del browser in modo tale da non dover utilizzare più pagine mantenendo invariata l'esperienza d'uso.

Capitolo 6

Test

In questo capitolo viene esposto il testing dell'infrastruttura da noi costruita.

I test effettuati si dividono in: delle singole componenti, in cui sono state testate componenti indipendenti utilizzando appositi mock objects e test fixtures; in locale, in cui l'intera infrastruttura è stata testata, completa di tutte le componenti, su una sola macchina; in cluster, dove la infrastruttura è stata installata e mandata in esecuzione su più macchine.

6.1 Test delle componenti

Durante lo sviluppo sono stati effettuati test sulle singole componenti che costruiscono il sistema. Questi sono stati pensati e implementati basandosi sulle situazioni attese durante la vita delle singole componenti software.

La motivazione che ha spinto alla formulazione di questi test è da ricercare nel valore aggiunto che questi apportano al software finale. L'attenzione posta nella definizione di test porta alla costruzione di codice di più alta qualità e meno affetto da bug rispetto a uno equivalente in termini di funzionalità ma non supportato da test rigorosi.

Al fine di organizzare al meglio il testing e permettere di verificare che il comportamento delle componenti rimanesse consistente con i risultati attesi durante lo sviluppo, sono stati implementati dei test utilizzando il framework JUnit.

Figura 6.1: Logo di JUnit

6.1.1 JUnit

JUnit è un framework per il testing di unità inizialmente sviluppato da Kent Beck e Erich Gamma. È in continuo sviluppo e vede una grande partecipazione comunitaria. JUnit è utilizzato in progetti di qualsiasi dimensione e in ogni ambito, è ufficialmente supportato come ambiente di testing di default dai maggiori sistemi di build automatizzata e rappresenta la libreria esterna più utilizzata all'interno di progetti Java, tanto che i maggiori IDE la inseriscono nelle librerie predefinite.

I test sono solitamente racchiusi in una o più classi poste eventualmente in un companion package solitamente recante il nome di "test" o addirittura, come accade per la configurazione di default di Maven, in una directory differente del progetto.

In ogni classe sono definiti dei metodi pubblici e senza parametri che contengono i passi dei test e i relativi controlli di corretto funzionamento. I metodi da utilizzare per i test sono segnalati dall'apposita annotazione @Test mentre è possibile definire dei metodi da richiamare prima e dopo l'inizio di ogni test, utilizzando @Before e @After, oppure prima e dopo tutti i test, utilizzando @BeforeClass e @After Class.

6.1.2 Libreria di interfacciamento a Neo4j

È stato testato il comportamento della libreria di interfacciamento con il database e il funzionamento del plugin spaziale.

A tal fine il test procede nella creazione di nodi appartenenti alla stessa rete verificando che le proprietà inserite e restituite combacino, che la lista di nodi per una rete sia completo, che le distanze misurate per il vicinato siano

corrette, che le procedure di cancellazione funzionino e che il movimento dei nodi nello spazio porti alla corretta ridefinizione del vicinato.

Viene anche verificato che nodi appartenenti a reti diverse non vengano considerati erroneamente vicini, che le proprietà delle reti restituite combacino con quelle inizialmente definite e che la procedura di cancellazione di una rete funzioni.

6.1.3 Funzionamento del backend

È stata testata la parte di infrastruttura che comprende la coda di messaggi, il backend Storm e la implementazione della funzione Closest Source.

Per lanciare questi test si è reso necessario mandare in esecuzione manualmente il broker di messaggi Kafka e il database Neo4j mentre la pulizia del database e della coda dei messaggi è effettuata dagli appositi metodi @After e @Before, che vanno a costruire una test fixture.

Durante il test vengono creati diversi nodi "uninitialized", viene mandata in esecuzione localmente una topologia Storm mentre su un numero variabile di thread vengono eseguiti gli invii di posizioni fittizie sotto forma di pacchetti indirizzati alla coda di messaggi, mimando il comportamento del servizio web. I test, con esito positivo, verificano che vengano inizializzati i nodi e che il vicinato computato sia coerente con il risultato atteso.

Nel test relativo alla situazione in cui la computazione deve avvenire in cloud viene anche verificato che lo stato dei device, rappresentato dalla lunghezza dal percorso più corto verso una sorgente, sia correttamente calcolato.

6.2 Test della infrastruttura

I test riguardanti l'intera infrastruttura sono stati svolti sia mandando in esecuzione tutte le componenti su una singola macchina, sia utilizzando un piccolo cluster.

In questa sezione vengono esposti i metodi utilizzati, le loro motivazioni e i risultati ottenuti.

6.2.1 Test locali

Al fine di testare l'infrastruttura completa si è dapprima proceduto a un test localizzato su un'unica macchina. I test sono stati realizzati con un numero limitato di istanze di Kafka e utilizzando un basso numero di task per i bolt di Storm, che è stato eseguito in modalità debug.

I test, eseguiti con esito positivo, si sono appoggiati a un simulatore di dispositivi, costruito da Brunetti, in grado di simulare sia device computazionali che non con una grande configurabilità in termini di numero di dispositivi, frequenze di invio dei dati, frequenza e ampiezza degli spostamenti e scelta casuale di sorgenti tra i nodi.

L'esigenza di mandare in esecuzione l'infrastruttura con un basso numero di istanze e task era legata alla scarsa disponibilità in termini di risorse, specialmente di memoria RAM, nelle macchine disponibili per il test, che dovevano eseguire contemporaneamente Tomcat, Kafka, Storm, Neo4j, il simulatore e il viewer, software tutti in esecuzione su una o più Java Virtual Machine e su un browser web. Grazie a questa configurazione è stato possibile mantenere in esecuzione una topologia con dieci nodi in movimento sia per il caso di computazioni su device che per il caso di computazione su Storm.

6.2.2 Test su cluster

L'infrastruttura completa è stata anche testata, con successo, su tre macchine fisiche, due delle quali in possesso di 4GB e la terza di 16GB di memoria RAM.

All'interno della macchina da 16GB, più performante delle altre due, è stato mandato in esecuzione il nimbus, Neo4j e Tomcat mentre nelle altre si sono spartiti il cluster Storm e Kafka.

Tutte e tre le macchine hanno installato un sistema operativo basato su Linux ma sarebbe possibile far dialogare macchine con sistemi operativi completamente differenti.

Il simulatore invece è stato sempre mandato in esecuzione su un PC a parte.

6.2.3 Test di performance

I test sulle performance sono proceduti variando il numero di device, il delay del ciclo interno al dispositivo e la complessità delle computazioni effettuate. In particolare la frequenza dei cicli è stata mantenuta inversamente proporzionale al numero di dispositivi. Questa scelta è stata dettata dalla necessità di assecondare le risorse computazionali a disposizione sia del simulatore sia delle macchine su cui è stata installata l'infrastruttura, oltre che per simulare situazioni reali in cui, all'aumentare dei dispositivi, si è spesso interessati a rilassare il tempo di monitoraggio.

La configurazione utilizzata per la simulazione genera sempre un vicinato compreso tra i dieci e i venti dispositivi.

Nei grafici qui presentati l'asse delle ascisse indica il numero di device simulati mentre l'asse delle ordinate il tempo intercorso in media tra l'invio di un evento e la ricezione di un risultato.

I pacchetti, normalmente di circa 200 byte, sono stati artificialmente aumentati di dimensione fino a portarli a 1KB. Questa operazione è stata effettuata al fine di permettere di testare le performance con un flusso dati di grande dimensione e allo stesso tempo simulare un aumento di complessità nelle computazioni effettuate: il device o il cloud procedono a eseguire operazioni che prevedono un carico linearmente dipendente alla dimensione del pacchetto.

Tali test sono stati effettuati in ambiente distribuito.

Per via della mancanza di disponibilità di una piattaforma in grado di simulare un grande numero di device, i test sono stati condotti con un numero di dispositivi limitato.

Le misurazioni effettuate riguardanti la computazione su dispositivo hanno prodotto un risultato possibile e atteso, anche se non si tratta della situazione sperata. Il simulatore aveva a disposizione risorse limitate per condurre i test per cui i dispositivi simulati, caricando di operazioni cpu–expensive la macchina fisica, hanno influito negativamente sulle performance di tutto il pool di device.

D'altro canto i test riguardanti la computazione su cloud hanno restituito risultati incoraggianti. I tempi misurati ben riflettono l'aumento di complessità dovuto all'incremento di dimensione del pacchetto e allo stesso tempo dimostrano che la infrastruttura riesce a sostenere il carico dato da decine di dispositivi mantenendo inalterato il tempo di risposta.

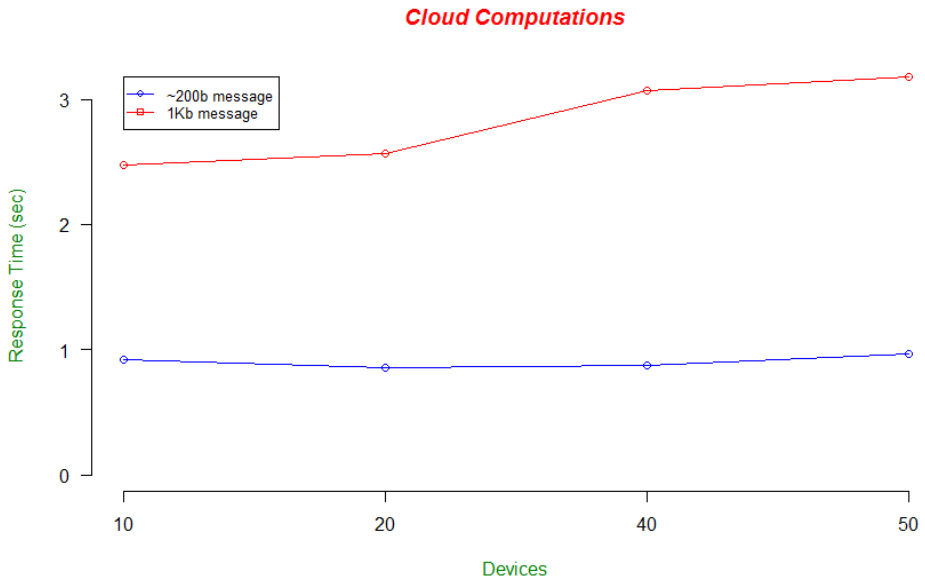


Figura 6.2: Test con computazione su cloud

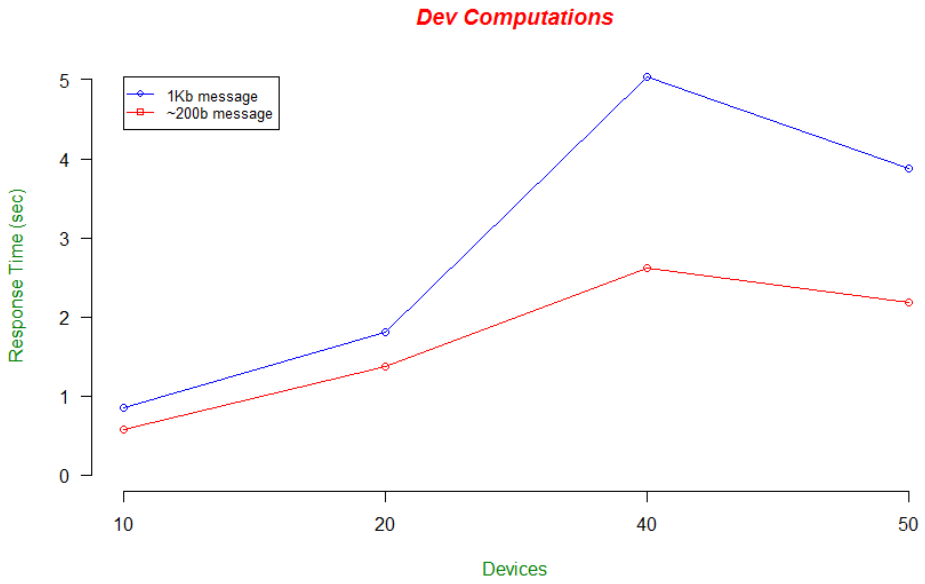


Figura 6.3: Test con computazione su dispositivo

Conclusioni

Il progetto ha rappresentato una sfida e allo stesso tempo un'ottima esperienza in quanto sono state utilizzate tecnologie prima sconosciute e ha riguardato un sistema di notevoli dimensioni.

Apache Storm si è rivelata una piattaforma in grado di scalare facilmente e rimanere online senza necessità di supervisione, può gestire grandi quantità di dati per restituire un risultato in tempi brevi e la sua configurazione risulta semplice. Gli sforzi dedicati al suo sviluppo da parte della Apache Software Foundation, il crescente utilizzo in ambito di produzione di importanti aziende, il supporto da parte delle compagnie di cloud hosting sono segnali che questa tecnologia prenderà sempre più piede come soluzione per la gestione di computazioni distribuite orientate agli eventi.

Allo stesso modo Apache Kafka, un progetto più maturo di Storm, permette di gestire senza grandi preoccupazioni una grande quantità di eventi. A questo va aggiunto che la configurazione di un cluster risulta estremamente semplice e con la stessa facilità il broker permette l'inserimento e la rimozione di macchine fisiche in qualsiasi momento. Queste caratteristiche hanno portato alla sua larga diffusione in ambito di produzione di grandi aziende.

L'accoppiata di queste due tecnologie potrebbe costituire la base di svariate applicazioni di tipologia differente rispetto a quella dell'elaborato.

L'infrastruttura costruita è conforme ai requisiti specificati in termini di correttezza dei risultati forniti e scalabilità della computazione.

Durante tutto il ciclo di sviluppo è stata data particolare attenzione alla modularità di tutte le componenti, che sono ben disaccoppiate. La struttura creata rappresenta la base su cui costruire, in futuro, applicazioni complesse.

Il collo di bottiglia riscontrato per le simulazioni con computazione su dispositivo evidenzia la necessità di utilizzare più macchine fisiche anche per

l'esecuzione del simulatore.

Questi test non sono stati possibili per mancanza di strumentazione e motivi di tempo.

Nel futuro immediato sarà necessario un testing che coinvolga un cluster con più macchine in modo da poter effettuare i tuning richiesti per ottenere le massime prestazioni. A questo andrebbe aggiunto, in un futuro più lontano, un test in ambiente reale, con dei device e una rete dati non simulati, in modo da verificare il comportamento effettivo ed effettuare eventuali aggiustamenti.

In termini di prestazioni l'infrastruttura reagisce bene, particolarmente se i calcoli vengono effettuati su cloud. Va inoltre considerato che le simulazioni, da noi condotte su piccola scala, sarebbero riproducibili molto facilmente su più larga scala grazie alla lungimiranza nella scelta delle tecnologie utilizzate per la realizzazione del backend.

Ringraziamenti

Ringrazio il relatore, Prof. Mirko Viroli, per l'opportunità offerta di affrontare lo sviluppo di un sistema complesso, la fiducia dimostrata sul successo dell'impresa e la disponibilità mostrata nel fornire consigli e materiale.

Ringrazio il correlatore, Ing. Pietro Brunetti, per l'inestimabile valore in termini di collaborazione, pazienza, disponibilità e flessibilità mostrata per permettermi di adempiere a tutti i doveri universitari che ho dovuto portare avanti in parallelo insieme a tutti i lavori riguardanti la tesi.

Un ringraziamento particolare va a tutte le persone, in particolar modo alla mia famiglia, sempre presente nei momenti di difficoltà, che mi sono state vicine durante il lungo e intenso periodo su cui ho lavorato alla tesi e nel corso di tutto il percorso universitario, senza le quali la mia esperienza triennale non avrebbe mai visto i successi e le relative soddisfazioni che ho ottenuto.

Bibliografia

[1] S. Ali, S. Khusro, A. Rauf, and S. Mahfooz. Sensors and mobile phones: Evolution and stateof-the-art. *Pakistan Journal Of Science*, 2014. http://www.researchgate.net/publication/272482886_Sensors_and_Mobile_Phones_Evolution_and_State-of-the-Art.

[2] Apache Software Foundation. Sito web ufficiale di Apache Kafka. https://kafka.apache.org.

[3] Apache Software Foundation. Sito web ufficiale di Apache Storm. https://storm.apache.org/.

[4] Apache Software Foundation. Storm Fault Tolerance. https://storm.apache.org/documentation/Fault-tolerance.html.

[5] Bluetooth SIG, Inc. Bluetooth basics. http://www.bluetooth.com/Pages/Basics.aspx.

[6] Danyl Bosomworth. Mobile Marketing Statistics 2015. http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/.

[7] John Greenough. The 'Internet of Things' Will Be The World's Most Massive Device Market And Save Companies Billions Of Dollars. http://uk.businessinsider.com/how-the-internet-of-things-market-will-grow-2014-10.

[8] Linkurious SAS. Sito web ufficiale di Linkurious. http://linkurio.us/product/.

[9] N. Marz. Storm: distributed and fault-tolerant realtime computation. https://vimeo.com/40972420.

[10] Neo Technology, Inc. Graph Data Modeling Guidelines. `http://neo4j.com/developer/guide-data-modeling/`.

[11] Neo Technology, Inc. Import Data Into Neo4j. `http://neo4j.com/developer/guide-importing-data-and-etl/`.

[12] Neo Technology, Inc. Neo4j case studies. `http://neo4j.com/case-studies/`.

[13] L. Richardson and S. Ruby. *Restful web services*. O'Reilly Media, 2007.

[14] Thinktube Inc. Why Wi-Fi Direct can not replace Ad-hoc mode. `http://www.thinktube.com/tech/android/wifi-direct`.

[15] Vari collaboratori. Neo4j Spatial Repository and Documentation. `https://github.com/neo4j-contrib/spatial`.

[16] Wikipedia. AJAX. `https://it.wikipedia.org/wiki/AJAX`.

[17] Wikipedia. Wireless ad hoc network. `https://en.wikipedia.org/wiki/Wireless_ad_hoc_network`.

Elenco delle figure