



UNIVERSITÀ DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Triennale in Ingegneria Informatica

**Analisi e valutazione delle prestazioni di
architetture distribuite scalabili per IoT Big Data**

Relatori:

Nesi Paolo

Bellini Pierfrancesco

Candidato:

Giovannoni Alberto

ANNO ACCADEMICO 2021/2022

Ringraziamenti

Vorrei riservare questo spazio della mia tesi ai ringraziamenti verso tutti coloro che hanno contribuito, con il loro supporto, al conseguimento di questo meraviglioso percorso.

Innanzitutto, un grande ringraziamento al Professor Nesi e al Professor Bellini, sempre pronti a guidarmi in ogni fase della realizzazione della tesi con suggerimenti e dritte. Grazie a voi ho accresciuto le mie conoscenze e le mie competenze.

Ringrazio il Dott. Ipsaro Palesi per avermi accompagnato da vicino sin dalle prime fasi del progetto con grande disponibilità e spirito di sacrificio. Grazie anche a tutti i colleghi del DISIT, per avermi accolto e messo a mio agio e, soprattutto, per avermi insegnato tanto.

Desidero ringraziare in modo speciale la mia famiglia per il loro amore e incoraggiamento, e per avermi sempre fatto sentire quanto credessero in me. Senza di loro non avrei mai goduto di così tante opportunità. A mia mamma, che con un semplice sguardo riusciva a capire se qualcosa non andava e mi spronava a parlarne; a mio babbo che ad ogni esame mi ripeteva il solito "Mi raccomando eh... in gamba" che col tempo è diventato così importante da non farmi uscire prima di averglielo sentito dire. Alla mia sorellina che a modo suo mi ha dimostrato quotidianamente il suo affetto e il suo supporto, che ha gioito dei miei successi come fossero suoi. Grazie ai miei nonni, zii e cugini per essermi sempre stati vicini e non avermi mai fatto mancare il loro affetto e supporto.

Un grazie enorme a tutti i miei amici, sia quelli d'infanzia che quelli che ho incontrato durante il mio percorso di vita. Grazie per essermi stati vicino ed aver condiviso con me serate di svago fra una giornata di studio e l'altro. Grazie ai miei vecchi compagni della 5e + Brando

per essere rimasti così uniti anche dopo il diploma. Grazie a Vitto Brando e Ose, con cui ho trascorso interminabili serate sul gioco di turno su cui eravamo in fissa. Grazie a tutte le persone che ho incontrato durante questo percorso di studi, in particolare a Riccardo e Giovanni, i miei compagni di università e molto altro, per aver reso più leggere le giornate di lezione e quelle di studio, per aver condiviso con me successi e fallimenti. Tenete duro che fra poco toccherà a voi spremervi le meningi per scrivere i ringraziamenti della vostra tesi.

Un grazie pieno di amore alla mia fidanzata Elena che c'è sempre stata per me, sia nei momenti felici che in quelli bui. Sei la mia complice a tutto tondo e non riuscirò mai a esprimere a parole quanto tu sia stata fondamentale durante questo percorso di studi e di vita.

Indice

1 INTRODUZIONE	1
1.1 Panoramica	1
1.2 Obiettivo della tesi	1
1.2.1 Data Ingestion	2
1.2.2 Virtual Machine micro 1	2
2 ANALISI DEI REQUISITI	3
2.1 Snap4City	3
2.2 I modelli	4
2.3 I device	7
2.4 L'architettura	8
2.4.1 Orion Filter	8
2.4.2 Orion Broker	8
2.4.3 Knowledge Base	9
2.4.4 WebSocket	9
2.4.5 Apache Kafka	9
2.4.6 MongoDB	10
2.5 NiFi	10
2.5.1 Cos'è?	10
2.5.2 A cosa serve?	10
2.5.3 Interfaccia utente	11
2.5.4 Configurazione	12
2.6 OpenSearch	15
2.6.1 Cos'è e a cosa serve?	15
2.6.2 Come funziona?	15

2.6.3	L'architettura	16
2.7	Il programma	16
2.7.1	Struttura	17
2.7.2	Lo script	18
3	PROGETTAZIONE	24
3.1	Configurazione di partenza	24
3.2	Caricamento della VM	25
3.3	Lettura dei dati dalla VM	27
3.3.1	API utilizzate	27
3.3.2	API spaziali	28
3.3.3	API temporali	28
3.3.4	Creazione dei device	28
3.4	Scrittura su device mobili	29
4	RISULTATI Sperimentali	32
4.1	Caricamento	32
4.1.1	Inserimento su Orion Filter	32
4.1.2	Inserimento su Orion Broker	34
4.1.3	Saturazione degli inserimenti	37
4.1.4	Tempi di attesa	38
4.1.5	Stato della VM micro 1 al termine dell'inserimento	39
4.2	Lettura	40
4.3	KB con 3.000 device	40
4.3.1	Query spaziali	40
4.3.2	Lettura e scrittura	41
4.4	KB con 300.000 device	43
4.4.1	Query spaziali	43
4.4.2	Query temporali	43
4.4.3	Lettura e scrittura	44
4.5	Device mobili	45

5 CONCLUSIONI	47
5.1 Scrittura	47
5.2 Lettura	48
5.3 Scrittura e Lettura contemporanee	48
5.4 Device mobili	48
5.5 Sviluppi futuri	49
6 SITOGRAFIA	50

1

INTRODUZIONE

1.1 Panoramica

Le prime fasi del progetto sono state caratterizzate dalla creazione di un programma python di Data Ingestion, che permetesse di convertire i dati rilevati dai sensori in un formato leggibile e salvabile nei database di Snap4City.

Il salvataggio di tali dati consente la creazione di uno storico su cui effettuare analisi e studi futuri.

Una volta terminata questa fase preliminare, il passo successivo è stato quello di utilizzare questo programma per testare i limiti e le potenzialità a di una macchina virtuale senza carico su cui vengono inseriti massivamente dati provenienti da un programma esterno a quest'ultima.

1.2 Obiettivo della tesi

L'obiettivo di questa tesi è effettuare uno studio delle prestazioni di un'architettura analogia a quella di Snap4City installata su una Macchina Virtuale (VM) di tipologia micro 1. Lo studio viene effettuato attraverso l'uso di due script Python: il primo è il programma di Data Ingestion, che effettua l'inserimento di un grande quantitativo di dati ad una elevata frequenza; il secondo script, utilizza delle chiamate API per stressare la VM con operazioni di lettura.

Le uniche richieste che vengono fatte alla macchina sono quelle relative ai due programmi python. Questo permette alla VM di dedicare ad essi un maggior numero di risorse rispetto ad una qualsiasi piattaforma online in cui le risorse sono condivise fra più utenti.

1.2.1 Data Ingestion

La Data Ingestion è un processo che consiste nell'acquisizione e nell'importazione di dati da diverse fonti, come file, sistemi esterni, sensori e dispositivi IoT, per renderli disponibili per l'elaborazione, l'archiviazione e l'analisi. Questo processo permette di raccogliere, trasformare e caricare i dati in un sistema di gestione dei dati per essere utilizzati per attività di business intelligence e analytics. La Data Ingestion è una fase fondamentale del data management e consente alle aziende di trarre valore dai dati raccolti.

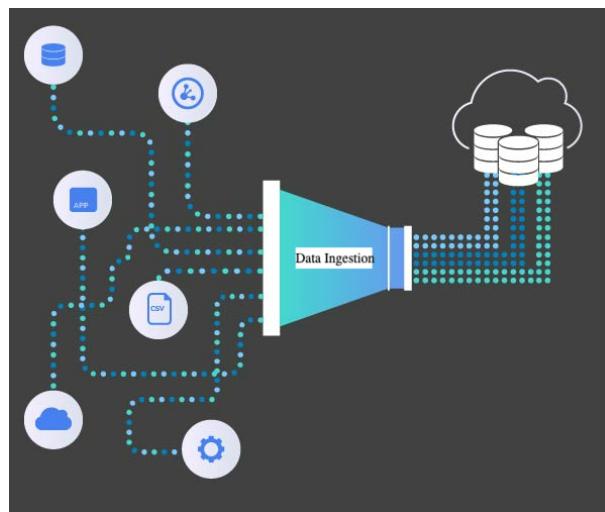


Figura 1.1: Schematizzazione Data Ingestion

1.2.2 Virtual Machine micro 1

La macchina che è stata utilizzata per ricevere e immagazzinare i dati è una Virtual Machine di tipo micro 1, le cui specifiche tecniche sono:

- 32 Gbyte di RAM;
- 500 Gbyte di hard disk
- 16 cores virtuali, ciascuno da 2.1 GHz

All'interno di questa VM è installata un'architettura che emula quella di Snap4City.

2

ANALISI DEI REQUISITI

2.1 Snap4City

Snap4City è una piattaforma di gestione dei dati che aiuta le comunità e le organizzazioni a gestire i dati con soluzioni on-cloud e on-premise. La piattaforma ha attualmente una vasta gamma di attività nei domini smart city e IoT/IoE (Internet of Things/Internet of Everything): definizione di strategie urbane, implementazione di sale di controllo, realizzazione di soluzioni AI etiche e spiegabili e calcolo di KPI utilizzati quotidianamente sia nella gestione della città che dell'industria.



2.2 I modelli

Su Snap4City la gestione dei dati provenienti dai sensori è possibile attraverso la definizione di modelli. Questi sono strutture a partire dalle quali è possibile creare dei duali virtuali dei sensori, i device. È solito creare un modello per ogni tipologia di sensore con cui si intende lavorare. La configurazione consiste nel compilare le seguenti sezioni del modello.

View Model - IBE Air Quality Smart

General Info	IoT Broker	Static Attributes	Values
IBE Air Quality Smart		Open Data coming from CNR IBE sensors	
Name Ok		Description	
air_quality		Sensor	
Device Type Ok		Kind	
CNR IBE		300	
Producer		Frequency	
Refresh Rate			
Healthiness Criteria		Healthiness Value	
Automatically generated			
Key Generation		Edge-Gateway Type	

Cancel

Figura 2.1: *Modello: info generali*

Riempito con informazioni come il nome del modello e la frequenza con cui vengono rilevati i dati dal sensore fisico.

View Model - IBE Air Quality Smart

General Info	IoT Broker	Static Attributes	Values
orionUNIFI ContextBroker	IoT Broker ngsi Protocol	Static Attributes ServicePath <small>only ngsi w/MultiService supports Service/Tenant selection</small>	Values <small>only ngsi w/MultiService supports ServicePath</small>
<input type="button" value="Cancel"/>			

Figura 2.2: *Modello: IoT Broker*

Selezione del Broker.

View Model - IBE Air Quality Smart

General Info	IoT Broker	Static Attributes	Values												
<input type="checkbox"/> Device in Mobility															
Subnature <small>Air Quality Monitoring (Environment)</small>															
<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 20%;">Locality</td> <td style="width: 80%;"><input type="text"/></td> </tr> <tr> <td>Value</td> <td><input type="text"/></td> </tr> <tr> <td>Name</td> <td><input type="text"/></td> </tr> <tr> <td>Value</td> <td><input type="text"/></td> </tr> <tr> <td>Street address</td> <td><input type="text"/></td> </tr> <tr> <td>Value</td> <td><input type="text"/></td> </tr> </table>				Locality	<input type="text"/>	Value	<input type="text"/>	Name	<input type="text"/>	Value	<input type="text"/>	Street address	<input type="text"/>	Value	<input type="text"/>
Locality	<input type="text"/>														
Value	<input type="text"/>														
Name	<input type="text"/>														
Value	<input type="text"/>														
Street address	<input type="text"/>														
Value	<input type="text"/>														
<input type="button" value="Cancel"/>															

Figura 2.3: *Modello: attributi statici*

Aggiunta di eventuali attributi statici e selezione della subnature corrispondente al tipo di sensore.

View Model - IBE Air Quality Smart			
General Info	IoT Broker	Static Attributes	Values
CO2	CO2 Concentration (CO)	parts per million (ppm)	float
Value Name Ok	Value Type Ok	Value Unit Ok	Data Type Ok
	Refresh rate	300	
	Healthiness Criteria	Healthiness Value	
VOC	Volatile Organic Compounds	parts per million (ppm)	float
Value Name Ok	Value Type Ok	Value Unit Ok	Data Type Ok
	Refresh rate	300	
	Healthiness Criteria	Healthiness Value	
airTemperatureIn	Temperature (temperature)	Celsius (°C)	float
Value Name Ok	Value Type Ok	Value Unit Ok	Data Type Ok
	Refresh rate	300	
	Healthiness Criteria	Healthiness Value	
CO	Humidity (humidity)	Percentage (%)	float
Value Name Ok	Value Type Ok	Value Unit Ok	Data Type Ok
	Refresh rate	300	
	Healthiness Criteria	Healthiness Value	
airHumidity	Humidity (humidity)	Percentage (%)	float
Value Name Ok	Value Type Ok	Value Unit Ok	Data Type Ok
	Refresh rate	300	
	Healthiness Criteria	Healthiness Value	

Figura 2.4: *Modello: valori*

Configurazione dei valori di interesse per il device virtuale, includendo il tipo di dato e la frequenza di rilevazione.

2.3 I device

I device sono come un duale virtuale del sensore a cui corrispondono e nei quali vengono inseriti i dati rilevati da tali sensori. Possono essere generati automaticamente tramite Node-RED, a partire dal modello corrispondente, specificando il nome del sensore ed i valori degli attributi statici.

I device sono usati nella piattaforma di Snap4City per ricevere i dati da programmi esterni, come il programma di Data Ingestion citato precedentemente. I dati vengono poi salvati nella piattaforma in modo da formare uno storico per tali informazioni.

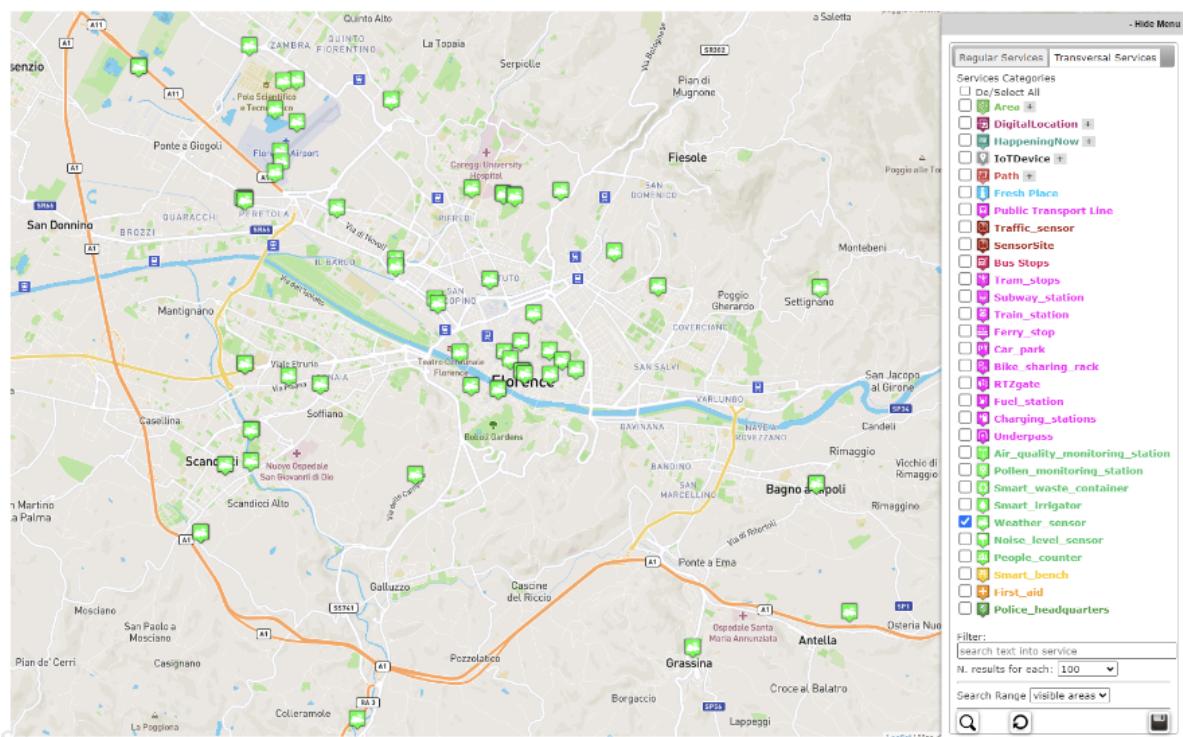


Figura 2.5: Screenshot di ServiceMap dei sensori meteo

In figura 2.5 è mostrato uno screenshot della piattaforma ServiceMap di Snap4City. In essa si vede la posizione di tutti i device meteo della città di Firenze salvati in Snap4City.

Lo stesso vale anche per i sensori della qualità dell'aria, dell'acqua, del traffico e molti altri.

2.4 L'architettura

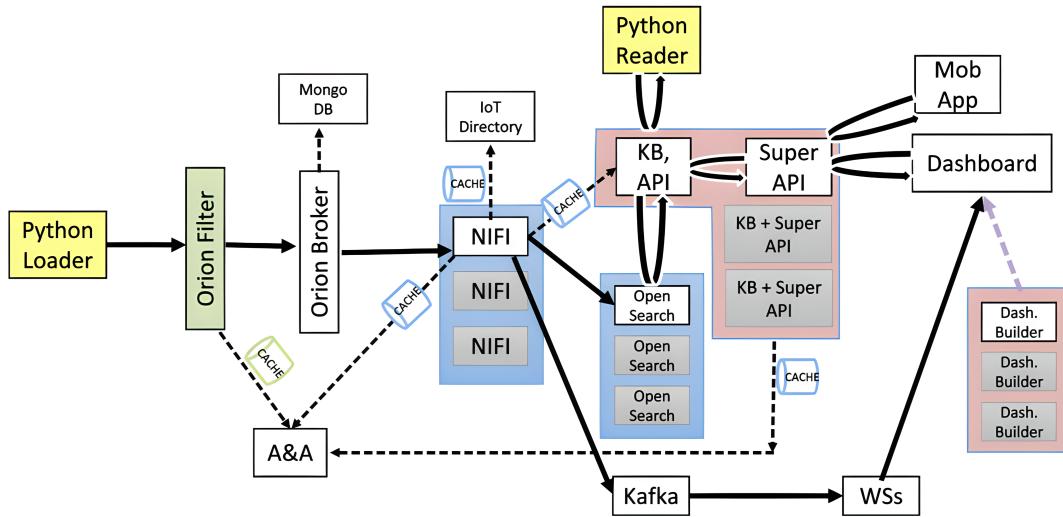


Figura 2.6: Architettura installata nella VM micro 1

In figura 2.6 è raffigurata l'architettura installata nella VM micro 1. Seguendo le frecce è possibile tracciare il percorso compiuto dei dati attraverso l'architettura. Analogamente è possibile seguire il percorso delle richieste di lettura dei dati attraverso le API.

Le cache hanno il compito di memorizzare i risultati delle ricerche recenti, in modo da poterli recuperare rapidamente in caso di richieste analoghe. Ciò consente di ridurre il carico sui sistemi di back-end e di migliorare le prestazioni del sistema.

2.4.1 Orion Filter

Orion Filter è la componente che supporta la gestione delle credenziali degli utenti attraverso l'autenticazione tramite token e l'esecuzione del controllo delle autorizzazioni per garantire che solo gli utenti autorizzati possano accedere ai dati.

2.4.2 Orion Broker

Orion Broker è un software che consente alle applicazioni, ai sistemi e ai servizi di comunicare tra loro e di scambiarsi informazioni. Esso funge da intermediario tra le varie parti, facilitando la convalida, l'archiviazione, e la consegna dei dati alle destinazioni appropriate.

2.4.3 Knowledge Base

La Knowledge Base (KB) è una raccolta organizzata di informazioni e conoscenze, strutturata come una banca dati per facilitare la consultazione, la ricerca e la gestione delle informazioni. Può essere utilizzata per conservare e recuperare informazioni relative a un'ampia varietà di argomenti. Indipendentemente dalla motivazione per cui vengono create, tutte le Knowledge Base hanno le stesse funzioni di base: input e output. La funzione di input consiste nell'immettere informazioni nel database e la funzione di output serve per richiamare tali informazioni quando necessario.

2.4.4 WebSocket

WebSocket (WS) è un protocollo di comunicazione bidirezionale aperto che permette alle applicazioni Web di comunicare in tempo reale con un server. Con WebSocket, il browser e il server possono comunicare in entrambe le direzioni senza dover effettuare una richiesta HTTP o HTTPS. Ciò significa che una volta stabilita la connessione, le parti possono scambiarsi messaggi in tempo reale senza dover riaprire la connessione.

2.4.5 Apache Kafka

Apache Kafka è una piattaforma di messaggistica distribuita progettata per gestire grandi volumi di dati in tempo reale. Consente la creazione di flussi di dati (chiamati "argomenti") ai quali possono essere inviati e letti messaggi in modo asincrono. Kafka è utilizzato principalmente per l'elaborazione di dati in tempo reale, la creazione di pipeline di dati e l'archiviazione di dati a lungo termine.

2.4.6 MongoDB

MongoDB è un database NoSQL open source. In quanto database non relazionale, è in grado di elaborare dati strutturati, semi-strutturati e non strutturati. Utilizza un modello di dati non relazionale, orientato ai documenti e un linguaggio di query non strutturato.

In Snap4City, MongoDB può essere utilizzato per gestire i dati raccolti dai sensori e dalle fonti esterne e può essere utilizzato per alimentare i servizi di analisi e visualizzazione dei dati.

2.5 NiFi

2.5.1 Cos'è?

Apache NiFi è una piattaforma di ingestione di dati in tempo reale, in grado di trasferire e gestire i dati tra diverse fonti e destinazioni.

NiFi (abbreviazione di "Niagara Files") è un potente strumento di gestione del flusso di dati di livello aziendale in grado di raccogliere, instradare, arricchire, trasformare ed elaborare i dati in modo affidabile e scalabile.

2.5.2 A cosa serve?

NiFi è stato creato per automatizzare il flusso di dati tra i sistemi. Un "flusso di dati" viene creato collegando diversi processori per trasferire e modificare i dati, se richiesto da una sorgente dati ad un'altra sorgente dati.

2.5.3 Interfaccia utente

NiFi è una piattaforma basata sul web a cui un utente può accedere tramite l'interfaccia utente web.

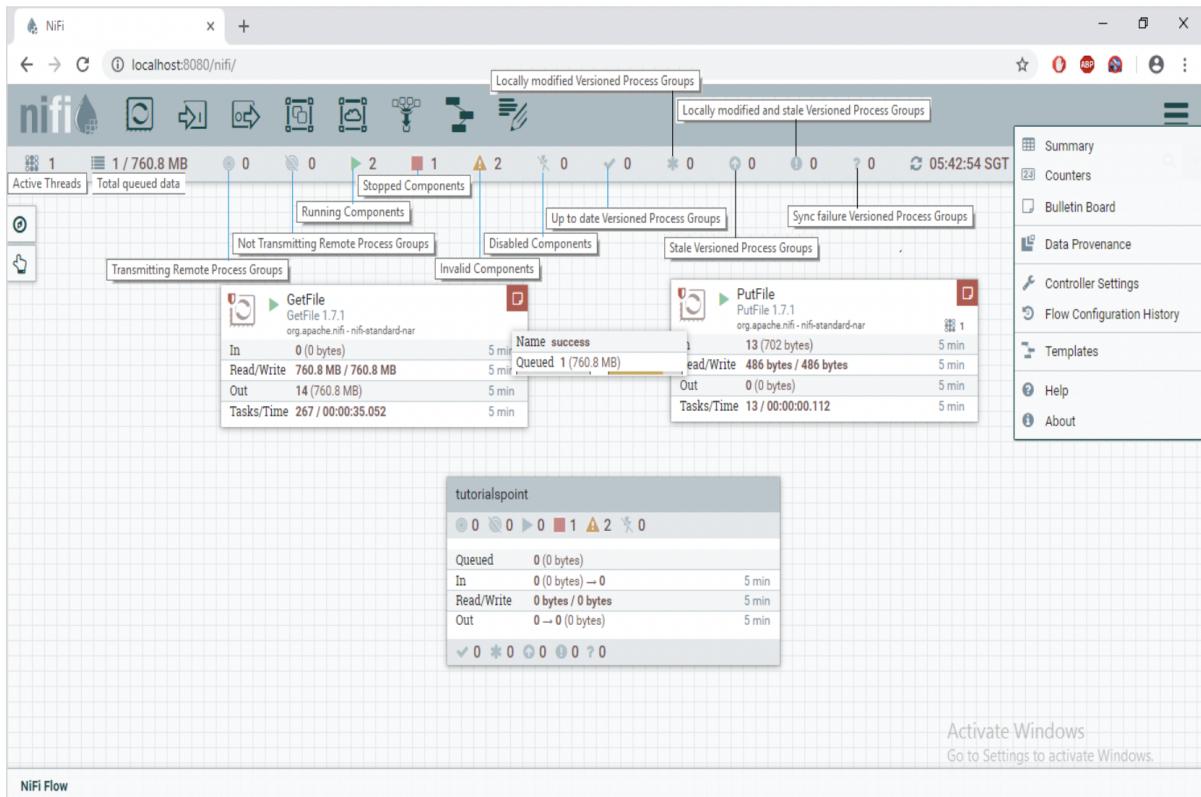


Figura 2.7: Interfaccia NiFi

Come mostrato in figura 2.7, un utente può accedere alle informazioni sui seguenti attributi:

- Thread attivi
- Totale dati in coda
- Trasmissione di gruppi di processi remoti
- Mancata trasmissione di gruppi di processi remoti
- Componenti in esecuzione
- Componenti arrestati
- Componenti non validi
- Componenti disabilitati
- Gruppi di processi con versioni aggiornate
- Gruppi di processi con versione modificati localmente
- Gruppi di processi con versioni obsolete
- Gruppi di processi modificati localmente e con versione obsoleta
- Errore di sincronizzazione dei gruppi di processi con versione

2.5.4 Configurazione

Nella figura 2.8 è raffigurata la configurazione di NiFi presente nella VM micro 1.

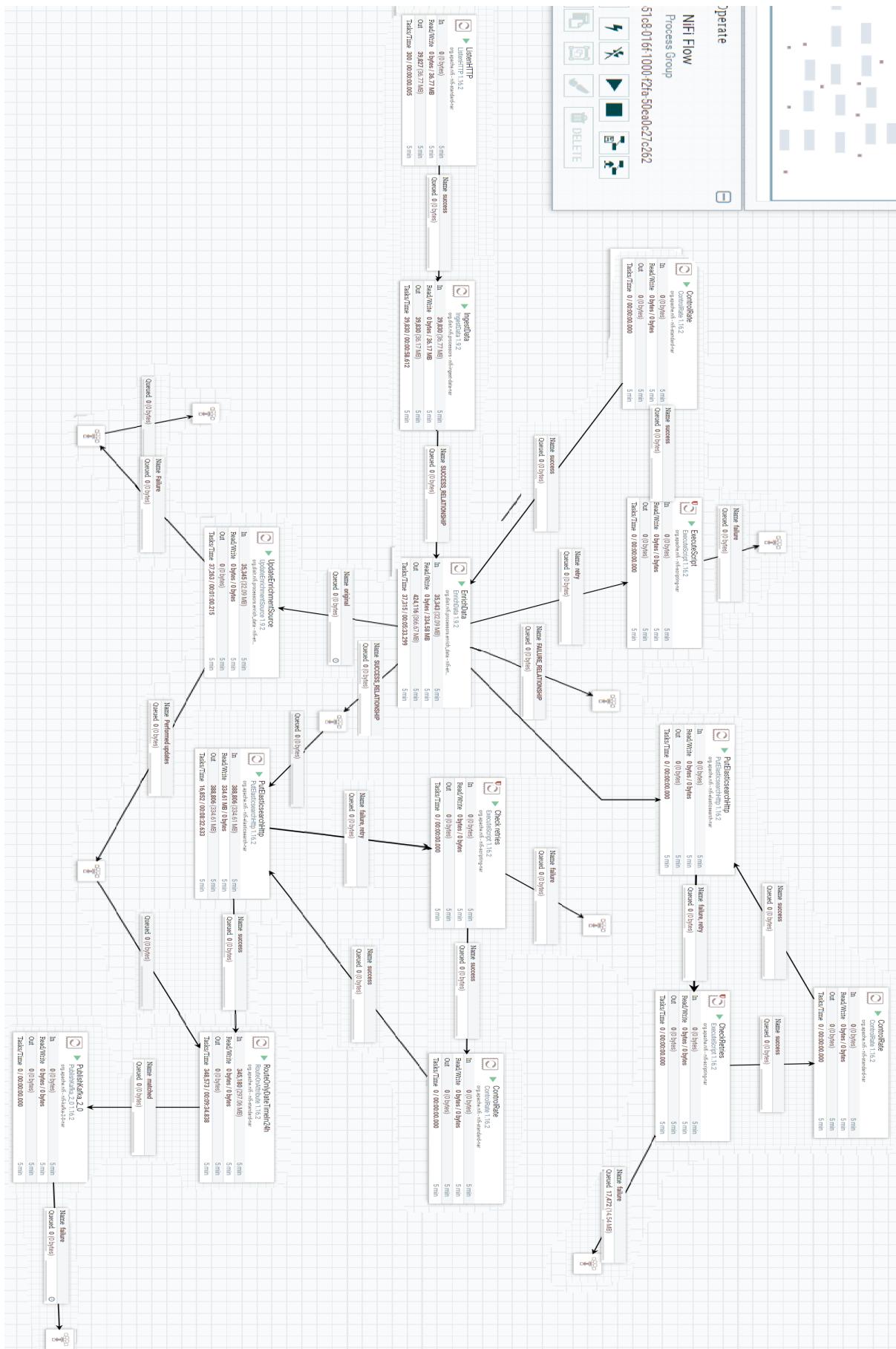


Figura 2.8: Configurazione di NiFi dentro alla VM micro 1

ListenHTTP

Avvia un server HTTP e rimane in ascolto su un determinato percorso di base per trasformare le richieste in ingresso in FlowFiles. L'URI predefinito del servizio sarà `http://hostname:port/contentListener`. Sono supportate solo le richieste HEAD e POST. Se il servizio è disponibile, restituisce "200 OK" con il contenuto "OK".

La proprietà più significativa di ListenHTTP è: `max-thread-pool-size`, il numero massimo di thread che devono essere utilizzati dal server Jetty incorporato. Il valore può essere impostato tra 8 e 1000. Il valore di questa proprietà influisce sulle prestazioni dei flussi e del sistema operativo, pertanto il valore predefinito (200) deve essere modificato solo in casi giustificati. Un valore inferiore al valore predefinito può essere adatto se solo un numero limitato di client HTTP si connette al server. Un valore maggiore può essere adatto se si prevede che un numero elevato di client HTTP effettui richieste al server contemporaneamente.

PutElasticsearchHttp

Scrive il contenuto di un file FlowFile in OpenSearch, utilizzando i parametri specificati, ad esempio l'indice da inserire e il tipo di documento.

RouteOnAttribute

Questo processore instrada i FlowFiles in base ai relativi attributi utilizzando il linguaggio di espressione NiFi. Gli utenti aggiungono proprietà con espressioni linguistiche NiFi valide come valori. Ogni espressione deve restituire un valore di tipo Boolean (true o false).

PublishKafka_2_0

Invia il contenuto di un oggetto FlowFile come messaggio ad Apache Kafka utilizzando l'API Kafka 2.0 Producer. I messaggi da inviare possono essere singoli FlowFiles o possono essere delimitati, utilizzando un delimitatore specificato dall'utente, ad esempio una nuova riga. Il processore NiFi complementare per il recupero dei messaggi è `ConsumeKafka_2_0`.

Configurazione delle componenti

Uno volta eseguita l'installazione di questa configurazione di NiFi, sono state eseguito delle sperimentazioni preliminari di inserimento dati, in modo da verificare dove si creassero delle code. Una volta identificata la componente su cui si generavano le code, si provvedeva ad aumentare i suoi Current Tasks, ovvero il numero massimo di thread in esecuzione in essa.

Le componenti su cui si sono generate code e su cui è stato cambiato il numero di Current Tasks sono:

- EnrichData (8 Current Tasks)
- PutElasticsearchHttp (8 Current Tasks)
- RouteOnlyDateTimeIn24h (16 Current Tasks)
- PublishKafka_2_0 (4 Current Tasks)

Questa è la configurazione finale di NiFi con cui sono state eseguite le sperimentazioni.

2.6 OpenSearch

2.6.1 Cos'è e a cosa serve?

OpenSearch è una suite di analisi dei dati e ricerca distribuita utilizzata per un'ampia gamma di casi d'uso come il monitoraggio delle applicazioni in tempo reale e l'analisi dei dati di registro.

OpenSearch fornisce un sistema altamente scalabile per fornire accesso e risposta rapidi a grandi volumi di dati con uno strumento di visualizzazione integrato, OpenSearch Dashboards, che semplifica l'esplorazione dei dati da parte degli utenti.

2.6.2 Come funziona?

OpenSearch funziona fornendo un file denominato "OpenSearch Description" che descrive come il motore di ricerca deve essere utilizzato. Questo file, solitamente chiamato *opensearch.xml*, contiene informazioni come il nome del motore di ricerca, una descrizione e un elenco di URL che descrivono come effettuare una ricerca e come ottenere i risultati. Il motore di ricerca utilizza le informazioni fornite nel file *opensearch.xml* per sapere come effettuare la ricerca e come visualizzare i risultati.

2.6.3 L'architettura

L'architettura di OpenSearch consiste di tre componenti principali: il client, il server e il motore di ricerca.

- **Il client** è l'applicazione o il browser che consente all'utente di effettuare una ricerca. Il client utilizza le informazioni fornite nel file opensearch.xml per sapere come effettuare la ricerca e come visualizzare i risultati.
- **Il server** è il sistema che ospita il file opensearch.xml, i dati e i servizi che l'utente sta cercando. Il server può anche fornire suggerimenti di ricerca in tempo reale e feed di risultati di ricerca.
- **Il motore di ricerca** è il software che effettua la ricerca sui dati e sui servizi ospitati dal server. Il motore di ricerca può essere personalizzato utilizzando le informazioni fornite nel file opensearch.xml.

In sintesi, il client utilizza le informazioni fornite dal server per sapere come effettuare la ricerca, il server fornisce i dati e i servizi per la ricerca, e il motore di ricerca effettua la ricerca sui dati e sui servizi.

2.7 Il programma

Nella prima fase di lavoro è stato realizzato un programma di Data Ingestion in linguaggio Python. Questo consiste in uno script multi-thread in grado di elaborare ed inserire dati provenienti da sensori fisici, in device virtuali sulla piattaforma Snap4City. Per eseguire la fase di inserimento dei dati velocemente, i dati vengono divisi equamente fra n thread, i quali hanno il compito di inviare messaggi contenenti i dati alla piattaforma di Snap4City, che si occupa dell'inserimento su OpenSearch. Con un'opportuna configurazione, il programma è in grado di lavorare con dati provenienti da qualsiasi tipo di sensore (meteo, traffico, ecc...).

2.7.1 Struttura

Per un corretto funzionamento, il programma ha bisogno di due file di supporto: un file di configurazione, ed un secondo script python che possa eseguire la trasformazione del dato grezzo, proveniente dal sensore, in un dato inseribile su Snap4City. Questo secondo script è detto "Parser".

File di configurazione

Il file di configurazione è un documento JSON che contiene tutte quelle informazioni che possono variare a seconda del caso in cui viene utilizzato il programma. Questo fa sì che il programma non debba essere modificato ogni volta che cambiano le condizioni iniziali del problema o l'uso che ne viene fatto.

```
{
  "kill": 2000,
  "toll": 200,
  "dataFolder": "2022_03",
  "threadNumber": 80,
  "refreshTime": 3,
  "sleepTime": 1,
  "token": {
    "clientID": "*****",
    "clientSecret": "*****",
    "username": "*****",
    "password": "*****",
    "url": "https://www.snap4city.org/..."
  },
  "patch": {
    "url": "https://..."
  },
  "separator": "$",
  "arraySeparator": "%",
  "mapping": {
    "id": "data${%}0$sensor_id",
    "type": "traffic",
    "anomalyLevel": {
      "type": "float",
      "value": "data${%}0$anom_level"
    },
    "averageSpeed": {
      "type": "float",
      "value": "data${%}0$speed"
    },
    "avgTime": {
      "type": "float",
      "value": "data${%}0$t_time"
    }
  }
}
```

Figura 2.9: File di configurazione

Come si può vedere nella figura 2.9, le informazioni contenute nel file sono relative alla configurazione di parametri, come il numero di thread che devono essere creati. Inoltre, sono contenute informazioni utili alla connessione alla piattaforma di Snap4City (nella sezione "token").

Nella sezione "mapping" sono specificate tutte le informazioni che servono al *parser* per convertire i dati grezzi nel formato voluto.

Il "patch" fornisce l'url della piattaforma che contiene i device, al quale vengono poi aggiunti gli indirizzi dei device stessi per eseguire gli inserimenti dei dati.

Parser

Il **Data Parsing** è una procedura che consiste nella conversione di dati da un formato all'altro. Viene generalmente utilizzato per rendere più comprensibili i dati esistenti, spesso non strutturati e illeggibili. In questo caso, il formato in cui devono essere trasformati i dati è dettato dal tipo di device virtuale in cui devono essere inseriti.

Il *parser* prende come input un file JSON contenente tutte le osservazioni da elaborare provenienti dai sensori. Seguendo le istruzioni contenute nella sezione "mapping" del file di configurazione, i dati vengono trasformati per poi essere restituiti al programma di Data Ingestion, in modo che possano essere inviati al device virtuale corrispondente.

2.7.2 Lo script

Scendendo più nel dettaglio nella struttura del programma di Data Ingestion, seguono alcune porzioni di codice python per illustrarne il funzionamento.

I thread

I thread sono definiti in modo da eseguire l'inserimento dei dati che gli vengono assegnati, sui device (figura 2.10). Questi dati sono contenuti nella lista "data" e vengono inviati attraverso la funzione "apiPatch" (figura 2.13).

La funzione "stopThread" serve per poter terminare anticipatamente l'esecuzione dei thread nel caso di fallimenti troppo frequenti. Questo fa parte di un protocollo di prevenzione degli errori realizzato nel programma stesso.

Il numero di thread generati nello script è definito da "threadNumber" nel file di configurazione (figura 2.9).

Una volta che il thread ha eseguito l'inserimento, attende un numero di secondi definito in "sleepTime" nel file di configurazione, per permettere ad altri thread di effettuare l'inserimento. Come tempo di sleep di default è stato utilizzato 1 secondo.

```
class working_thread(Thread):
    def __init__(self, conf, token, data, stop_event, kill, toll):
        Thread.__init__(self)
        self.conf = conf
        self.token = token
        self.data = data
        self.stop_event = stop_event
        self.stop = threading.Event()
        self.kill = kill
        self.toll = toll

    def stopThread(self):
        self.stop.set()

    def run(self):
        for r in self.data:
            if not self.stop.isSet():
                apiPatch(self.conf, r['id'], self.token, r, self.stop_event, self.kill, self.toll)
                time.sleep(sleepTime)
```

Figura 2.10: *Porzione di codice che definisce i thread*

Access Token

Per poter eseguire operazioni di inserimento di dati su piattaforme come quella di Snap4City, è prima necessario identificarsi fornendo le proprie credenziali ed altri identificativi come il "clientID" ed il "clientSecret". In questo modo Snap4City può inoltrare ai sistemi di controllo degli accessi queste credenziali e verificarne la validità. Se le credenziali risultano corrette, Snap4City rilascia un Access Token che non è altro che un'autorizzazione temporanea per effettuare operazioni di inserimento e richiesta dati. Gli Access Token di Snap4City, per questioni di sicurezza, hanno una validità di 25 minuti; è quindi necessario effettuare il "refresh" del token per continuare il lavoro una volta che scade il primo.

In figura 2.11 viene controllato se esiste già un Access Token valido con cui inviare i dati, mentre in figura 2.12 si vede la funzione usata per generare il token, nel caso in cui questo non sia stato ancora creato.

```
if s.cookies.get("access_token") == None or s.cookies.get("access_token") == "":
    access_token, refresh_token = accessToken(config)
    s.cookies.set("access_token", access_token)
    s.cookies.set("refresh_token", refresh_token)
    tokenTime = datetime.now()
```

Figura 2.11: *Porzione di codice usata per il controllo dell'esistenza dell'Access Token*

```

def accessToken(conf):
    access_token = ''
    refresh_token = ''
    payload = {
        'f': 'json',
        'client_id': conf.get('token').get('clientID'),
        'client_secret': conf.get('token').get('clientSecret'),
        'grant_type': 'password',
        'username': conf.get('token').get('username'),
        'password': conf.get('token').get('password')
    }

    header = {
        'Content-Type': 'application/x-www-form-urlencoded'
    }

    urlToken = conf.get('token').get('url')
    currentTime = datetime.now()
    try:
        response = requests.request("POST", urlToken, data=payload, headers=header)
        print(response.text)
        with open('log.txt', 'a') as f:
            f.write(response.text + "\n")
        response.raise_for_status()
    except requests.exceptions.HTTPError as errh:
        print(currentTime, "Http Error:", errh)
    except requests.exceptions.ConnectionError as errc:
        print(currentTime, "Error Connecting:", errc)
    except requests.exceptions.Timeout as errt:
        print(currentTime, "Timeout Error:", errt)
    except requests.exceptions.RequestException as err:
        print(currentTime, "Oops: Something Else", err)
    else:
        token = response.json()
        access_token = token['access_token']
        refresh_token = token['refresh_token']

    return access_token, refresh_token

```

Figura 2.12: Porzione di codice usata per la creazione dell'Access Token

ApiPatch

La funzione ”apiPatch” è il fulcro di tutto lo script, viene eseguita dai thread per effettuare gli inserimenti sui device. La prima parte della funzione si occupa di rigenerare l'access token a intervalli di tempo dettati dalla variabile ”refreshTime”, definita nel file di configurazione (fi-

gura 2.9). La variabile "head" serve per l'autenticazione al sistema di controllo degli accessi di Snap4City, mentre "url" contiene l'url del device su cui effettuare l'inserimento. L'inserimento viene fatto usando "requests.request" a cui vengono passate le risorse appena discusse e il dato "q" da inserire. Nella variabile "response" viene salvata la risposta ottenuta dalla richiesta di inserimento; usando "response.raise_for_status()" viene lanciata un'eccezione nel caso in cui l'inserimento non abbia avuto successo. Se viene lanciata un'eccezione, il blocco "except" la prende e esegue il blocco di istruzioni che effettua il retry dell'inserimento fino a 3 volte; altrimenti viene salvato il dato con il messaggio di errore corrispondente tramite la funzione "saveFail".

```

def apiPatch(conf, device, accessToken, r, stop_event, kill, toll):
    global exec, partial, total, nFailure, dateExp, tokenTime, times

    currentTime = datetime.now()
    dateExp = verifySignature(s.cookies.get("access_token"))
    if datetime.now() - tokenTime > timedelta(days=0, seconds=0, microseconds=0, milliseconds=0,
                                                minutes=refrTime, hours=0, weeks=0):
        tokenTime = datetime.now()
        access_token, refresh_token = refreshToken(config, s.cookies.get("refresh_token"))
        s.cookies.set("refresh_token", refresh_token)
        s.cookies.set("access_token", access_token)
        dateExp = verifySignature(s.cookies.get("access_token"))

    url = conf.get('patch').get('url') + device + '/attrs?elementid=' + device + '&type='
    + conf.get('mapping').get('type')
    head = {
        "Content-Type": "application/json",
        "Accept": "application/json",
        "Authorization": f"bearer {s.cookies.get('access_token')}",
    }
    try:
        exec += 1
        if exec % 100 == 0:
            partial = 0
        q = copy.deepcopy(r)
        for i in ['id', 'type']:
            q.pop(i)
        st = datetime.now()
        response = requests.request("PATCH", url, headers=head, data=json.dumps(q), verify=False)
        e = datetime.now()
        t = e - st
        response.raise_for_status()
        times.append(t.total_seconds())
    except Exception as ex:
        c = 0
        success = False
        while not success and c < 3:
            time.sleep(0.01)
            success = retry(conf, device, accessToken, r)
            c += 1
        if not success:
            saveFail(json.dumps(r), ex)
            check(stop_event, total, partial, kill, toll)

```

Figura 2.13: Porzione di codice che esegue l'inserimento dei dati nei device

Errori

Questa ultima parte illustra il codice utilizzato nel caso la "request" restituisca un messaggio di errore. Come si vede in figura 2.14, la prima azione è quella di ritentare l'inserimento. La funzione può essere richiamata fino a 3 volte sullo stesso dato (come si vede nella parte finale della figura 2.13), attendendo 0,01s fra un tentativo ed il successivo. In figura 2.15 si trova invece il codice che viene richiamato in caso tutti e tre i tentativi ripresentino l'errore. Con questa funzione, il dato e il messaggio di errore vengono uniti in un JSON che viene inserito nella lista "failed" che serve a tenere traccia di tutti i dati non inseriti.

```
def retry(conf, device, accessToken, r):
    try:
        url = conf.get('patch').get('url') + device + '/attrs?elementid=' + device + '&type=' + conf.get(
            'mapping').get(
            'type')

        head = {
            "Content-Type": "application/json",
            "Accept": "application/json",
            "Authorization": f"bearer {accessToken}",
        }
        st = datetime.now()
        response = requests.request("PATCH", url, headers=head, data=json.dumps(r), verify=False)
        e = datetime.now()
        t = e - st
        response.raise_for_status()
        times.append(t.total_seconds())
    return True
except Exception as ex:
    return False
```

Figura 2.14: *Porzione di codice usata per riprovare l'inserimento in caso di fallimento*

```
def saveFail(m, err):
    global total, partial
    total += 1
    partial += 1
    js = '{"JSON": ' + m + ', "error": "' + str(err) + '"}'
    failed.append(json.loads(js))
    with open('failed.txt', 'a') as f:
        f.write(js + "\n")
```

Figura 2.15: *Porzione di codice usata per il salvataggio di dati in caso di errori persistenti*

3

PROGETTAZIONE

3.1 Configurazione di partenza

La configurazione di partenza dell'esperimento è la seguente:

- una VM micro 1 che ha lo scopo di ricevere ed immagazzinare i dati;
- una seconda VM su cui vengono eseguiti gli script di Data Ingestion e di generazione di API che inviano richieste alla prima VM

Sulla VM ricevente sono stati precedentemente creati 10 modelli che differiscono per la subnature. Per ognuno di questi modelli sono stati creati 300 device, ognuno dei quali possiede 10 variabili oltre al dateObserved (che serve ad indicare la data e l'ora di rilevazione del dato). Le coordinate spaziali dei 300 device sono state generate randomicamente, ma in modo tale da rimanere nella zona di Firenze. Su questa VM vengono inseriti dati corrispondenti a 1 anno con un messaggio ogni 30 minuti, per ognuno dei device. In totale quindi, per ogni modello, vengono generati ed inseriti $48 \times 365 \times 300 = 5.256.000$ dati (48 ore * 365 giorni * 300 device).

I modelli creati per l'esperimento e le relative subnature sono:

Modello	Subnature
AirQuality	Air Quality Monitoring
Weather	Weather Sensor
Traffic	Traffic Sensor
Parking	Car Park
Water	Water Resource
Geologist	Geologist
Forestry	Forestry
Noise	Noise Level
Photovoltaic	Photovoltaic System
Irrigator	Smart Irrigator

3.2 Caricamento della VM

La prima fase di sperimentazione viene eseguita durante il caricamento dei dati nei device. I dati inseriti nei device vengono generati randomicamente. Gli inserimenti sono effettuati per tipologie di device, quindi su 300 device per volta. Una volta terminato l'inserimento dei dati corrispondenti all'intero anno, inizia l'inserimento nei device della tipologia successiva. I valori rilevati durante la sperimentazione sono:

- il numero di inserimenti al secondo;
- il tempo di esecuzione totale;
- la media del tempo impiegato da ogni inserimento;
- la percentuale di utilizzo di CPU sulla VM micro 1;
- il transfer rate su disco sulla VM micro 1

I dati rilevati dalla VM micro 1 servono per avere indicazioni sul livello di carico esercitato sulla macchina.

Come fase preliminare, vengono eseguiti degli esperimenti di lunga durata, ovvero inserimento di dati corrispondenti a 3 mesi. Questo serve per trovare una configurazione per NiFi

in modo tale che non si generino code durante gli esperimenti successivi, come già discusso nella sezione 2.5. Durante questi esperimenti, monitorando la schermata di NiFi ed il valore della percentuale di CPU in uso, è possibile identificare le componenti sulle quali si generano le code. Queste sono:

- Enrich Data: il suo numero di Current Tasks viene portato da 4 a 8;
- PutElasticsearchHttp: il suo numero di Current Tasks viene portato da 4 a 8;
- RouteOnlyDateTimeIn24h: il suo numero di Current Tasks viene portato da 10 a 16;
- PublishKafka_2_0: il suo numero di Current Tasks viene portato da 1 a 4;

La configurazione ottenuta è la stessa mostrata nella sezione 2.5, nella quale non si generano code durante le operazioni di inserimento, e quindi è possibile assumere che il tempo di attraversamento dell’architettura da parte del dato sia trascurabile.

Una volta terminata la fase preliminare, inizia la sperimentazione vera e propria. Questa consiste nella generazione ed inserimento di dati corrispondenti ad una settimana, $7*48*300 = 100800$ dati complessivi ad esecuzione (7 giorni * 48 ore * 300 device). A variare fra un esperimento e l’altro è il numero di thread impiegati: 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240.

L’utilizzo dei thread serve per simulare l’inserimento concorrente da parte di più dispositivi sulla piattaforma. Aumentando il numero di thread viene quindi aumentato il carico sull’architettura.

Dopo aver registrato i dati con numero di thread da 20 a 240, l’esperimento viene ripetuto provando ad inserire i dati direttamente su Orion Broker. Questo è fatto perché normalmente i dati devono passare da Orion Filter, il quale si assicura che le credenziali di accesso siano corrette per ogni inserimento. L’operazione di controllo delle credenziali richiede tempo, ma inserendo i dati direttamente su Orion Broker questo controllo viene aggirato.

3.3 Lettura dei dati dalla VM

Terminato il caricamento dei dati sui device, vengono definite delle query da utilizzare per leggere i dati inseriti. Questo permette di analizzare le capacità della VM micro 1 in lettura.

Come per l'inserimento dei dati, viene utilizzato un programma python che genera un numero crescente di thread (2, 10, 25, 50, 75, 100, 150, 200), dove ognuno dei thread effettua una serie di Request tramite la stessa API. In questo caso i thread simulano la presenza di più utenti che vogliono visualizzare i dati contenuti nell'architettura.

Nel primo esperimento, i valori rilevati sono stati:

- Il numero di query eseguite con successo al minuto;
- La percentuale di CPU utilizzata sulla VM micro 1

Come secondo esperimento, vengono eseguiti contemporaneamente gli script di scrittura e di lettura, così da verificare il comportamento dell'architettura nel caso di richieste simultanee di entrambi i tipi.

Per questa sperimentazione sono utilizzati un numero di thread tale che il carico sulla CPU fosse equamente spartito fra le operazioni di scrittura e di lettura.

I carichi sono:

- 40% - 40%
- 50% - 50%
- 60% - 60%

3.3.1 API utilizzate

Le API utilizzate durante questi esperimenti sono di due tipologie: spaziali e temporali. Tali query vanno a stressare parti diverse dell'architettura. Le query spaziali vengono gestite dalla KnowledgeBase (KB) mentre le query temporali da OpenSearch (OS). Con query spaziale si intende una API che richiede informazioni dai device i cui valori di longitudine e latitudine sono compresi in un'area selezionata. Le query temporali selezionano i dati nei device che hanno un dateObserved compreso fra la data di inizio e di fine specificate nella query.

Per studiare la scalabilità della Knowledge Base (KB), l'esperimento viene replicato dopo aver caricato 300.000 device a partire da un nuovo modello. I nuovi device sono suddivisi fra la zona di Firenze e quella di Bologna, per non andare ad aumentare eccessivamente il numero di device a Firenze.

3.3.2 API spaziali

La tipologia di API spaziale utilizzata è la seguente:

```
http://test-dashboard/ServiceMap/api/v1/?maxResults=1&selection=43.771628;  
11.256013&maxDists=1&categories=Noise_level_sensor
```

In questo esempio viene richiesto 1 device di categoria Noise nel raggio di 1 km dal centro di Firenze.

Nell'area delimitata nella query sono compresi 30 device di tipo Noise. Per la sperimentazione con 300.000 device, nella KB nel raggio di 1 km dal centro sono presenti comunque 30 device di tipo Noise. Questo perché i nuovi device sono stati generati a partire da un altro modello.

3.3.3 API temporali

La tipologia di API temporale utilizzata è la seguente:

```
http://test-dashboard/ServiceMap/api/v1/?&serviceUri=http://www.disit.org/km4city/  
resource/iot/orion-1/Organization/noise_1&fromTime=2021-01-01T00:00:00&  
toTime=2021-01-01T23:59:59
```

In questo caso vengono richiesti i dati nel device "noise_1" relativi alle osservazioni comprese fra le 00:00:00 e le 23:59:59 del 2021-01-01.

3.3.4 Creazione dei device

Per creare i 297.000 device mancanti nella KB, viene utilizzato uno script python che invia richieste di creazione di device di tipologia "taxi" con 5 thread e 0 secondi di sleep fra una richiesta e la successiva. Con questa configurazione vengono creati mediamente 3 device al

secondo, il che vuol dire che in un ora di esecuzione, vengono creati più di 10.000 device, che sono quelli contenuti in una piccola città.

3.4 Scrittura su device mobili

L'ultima fase della sperimentazione riguarda l'inserimento di dati su device mobili, ovvero device in cui vengono variati i valori di longitudine e latitudine. Questi esperimenti vengono condotti al fine di valutare le prestazioni delle uniche componenti dell'architettura che non sono usate durante la scrittura statica e la lettura, ovvero WebSocket e Kafka. Queste due componenti, come visto nella sezione 2.4, servono per la comunicazione in tempo reale e quindi sono utilizzate in operazione come la modifica della posizione di un device.

Per quanto riguarda la sperimentazione, viene creato un nuovo modello da cui è possibile creare i device mobili. Questo viene fatto attraverso un'impostazione selezionabile durante la creazione del modello stesso. Per monitorare l'effettivo spostamento dei device, viene creata una dashboard che mostra la posizione dei sensori su una mappa di Firenze. Siccome non è possibile monitorare più di un device per volta, all'interno della dashboard è visualizzato solo un device mobile che effettua spostamenti quando i messaggi sono processati da Kafka e WebSocket.

Nelle figure 3.1 e 3.2 è illustrato il movimento del device mobile (raffigurato dalla macchinina) quando al tempo t_1 viene inserito il messaggio con la nuova posizione del device.

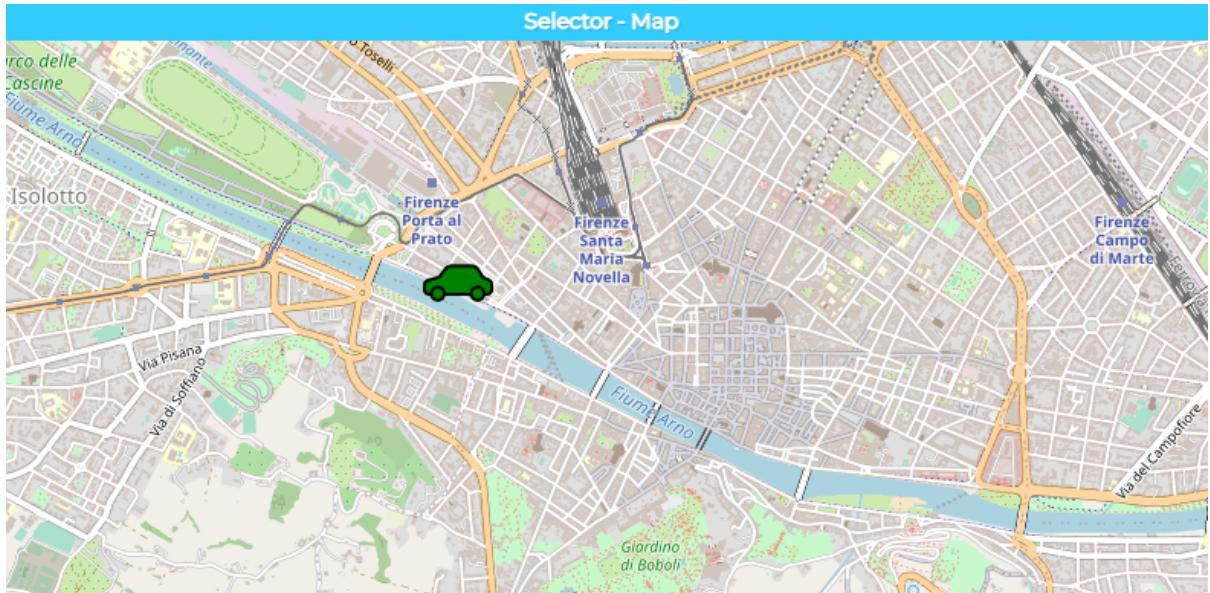


Figura 3.1: *Device mobile al tempo t_0*

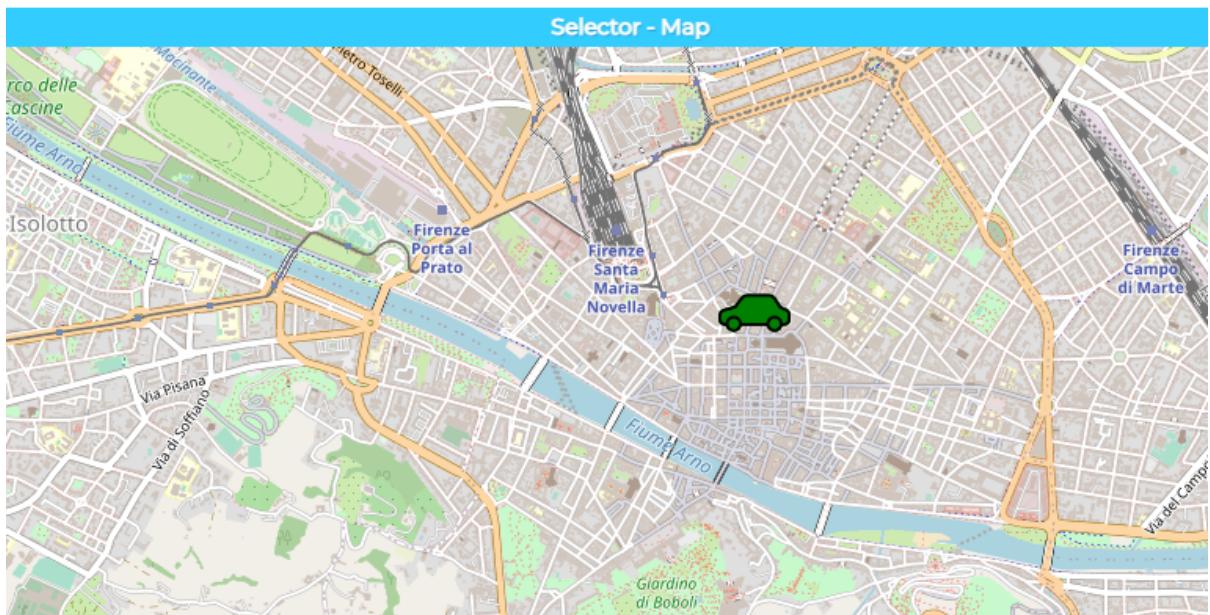


Figura 3.2: *Device mobile al tempo t_1*

Per questa sperimentazione vengono utilizzati:

- un programma Python multi-thread simile a quello usato nella prima sperimentazione che serve per inviare dei messaggi ai device mobili
- un secondo programma che si connette al WebSocket e che conta il numero di messaggi in arrivo

Il primo programma invia per 2 minuti messaggi contenenti la nuova longitudine e latitudine, oltre a 2 valori randomizzati per aggiornare le variabili dei device. Per ogni messaggio inviato dal primo script, in WebSocket arrivano 3 messaggi:

- location (longitudine e latitudine)
- value1
- value2

Durante la sperimentazione, il numero di messaggi inviati con i thread viene aumentato progressivamente in modo da incrementare il carico sulle componenti. I thread effettuano inserimenti su device mobili differenti. L'aumento del numero di thread serve a simulare l'aggiornamento di un numero sempre maggiore di device mobili da parte di più dispositivi.

Entrambi i programmi stampano il tempo che intercorre dall'invio/ricezione del primo e dell'ultimo messaggio. Questo serve a constatare quando il servizio perde la caratteristica di essere *real time*; ovvero, quando la differenza fra il tempo impiegato per inviare e quello per ricevere tutti i messaggi diventa non trascurabile.

4

RISULTATI Sperimentali

4.1 Caricamento

Questa prima raccolta di risultati deriva dagli esperimenti svolti durante la fase di caricamento della VM micro 1.

4.1.1 Inserimento su Orion Filter

Nella figura 4.1 viene rappresentato l’andamento dei tempi di esecuzione (in secondi) per inserimenti tramite Orion Filter effettuati dal programma di Data Ingestion. Durante l’esperimento viene aumentato il numero di thread in esecuzione nel programma stesso. I tempi di esecuzione sono rilevati prendendo il currnet time quando inizia e quando finisce l’invio dei 100800 dati (relativi a una settimana).

I tempi di esecuzione inizialmente scendono velocemente all’aumentare del numero di thread, ma una volta superati i 200 thread, si ha una saturazione e il grafico non scende più come prima.

In figura 4.2 è illustrato l’andamento degli inserimenti al secondo in relazione ai thread impiegati. Questo grafico viene ricavato dividendo i 100800 dati inseriti per il tempo di esecuzione impiegato al variare del numero di thread, ricavando così il numero di inserimenti al secondo. La curva inizialmente sembra assumere un andamento lineare ma, come per la figura 4.1, an-

che questo grafico raggiunge la saturazione in corrispondenza dei 200 thread, non riuscendo a superare i 140 ins/s.

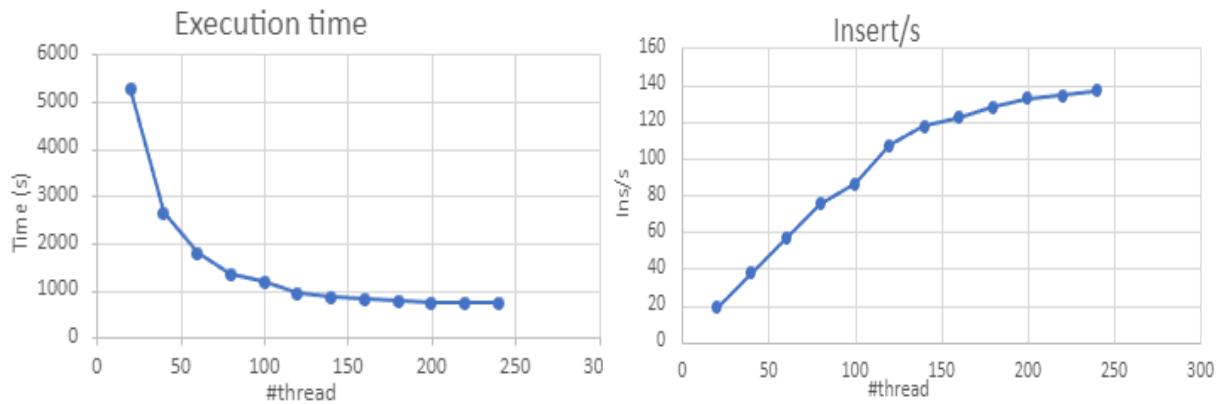


Figura 4.1: *Tempo impiegato dallo script per inviare 100800 dati al variare del numero di thread*

Figura 4.2: *Numero di inserimenti al secondo al variare del numero di thread*

Nelle Figure 4.3 e 4.4 vengono mostrate le informazioni ricavate dalla VM micro 1, più nello specifico l’andamento della percentuale di CPU utilizzata ed il Current Disk Write (CDW). Il Current Disk Write è la quantità di dati in MBytes che vengono scritti sul disco rigido in un secondo.

Questi grafici sono ottenuti facendo la media fra dati rilevati ogni 2 minuti della percentuale di CPU e del Current Disk Write.

La curva in figura 4.4 cresce linearmente e quindi la saturazione degli inserimenti non dipende dalla capacità di scrittura su disco.

Per quanto riguarda la percentuale di CPU in uso, c’è da tenere presente che una CPU è considerata sicura se lavora entro il 75%, quindi con 220 e 240 thread la macchina è in uno stato non sicuro. Al valore della CPU si deve infatti sommare la percentuale utilizzata per operazioni nel kernel che oscilla fra il 5% e l’8%.

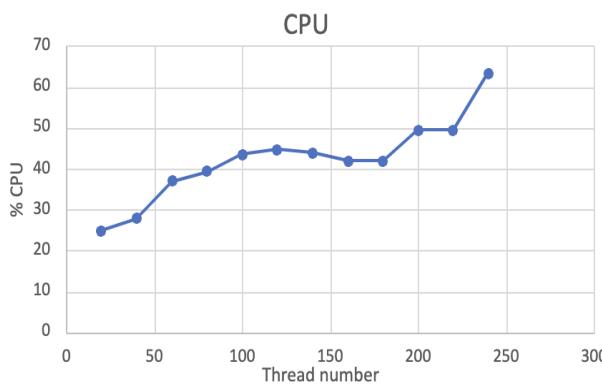


Figura 4.3: *Percentuale di CPU in uso al variare del numero di thread*

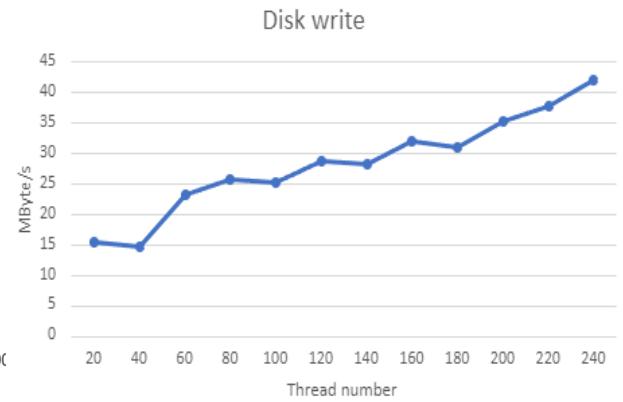


Figura 4.4: *Transfer rate al variare del numero di thread*

Per ricavare il grafico in figura 4.5, viene fatta la media dei tempi di ognuno dei 100800 inserimenti, prendendo il current time prima e dopo l'inserimento. Il valore rilevato è quindi il tempo medio necessario ad effettuare ogni inserimento al variare del numero di thread.

La curva presenta un gomito in relazione ai 120 thread, dopo il quale il tempo medio atteso per ogni inserimento inizia ad aumentare linearmente.

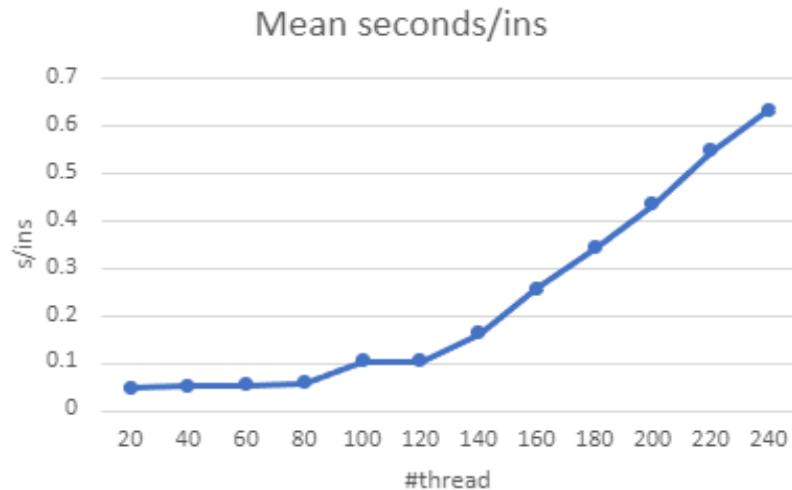


Figura 4.5: *Tempo medio atteso per ogni inserimento al variare del numero di thread*

4.1.2 Inserimento su Orion Broker

In figura 4.6 è riportato il grafico relativo al tempo di inserimento di tutti i 100800 dati tramite Orion Broker, iniziando a misurare in corrispondenza dell'invio del primo dato e terminando con l'invio dell'ultimo. Rispetto alla figura 4.1 relativa alla medesima sperimentazione ma con

inserimenti su Orion Filter, il grafico 4.6 risulta traslato verso il basso. Questo significa che i tempi necessari al completamento dello script sono inferiori.

Il grafico in figura 4.7, come per quello in figura 4.2, è stato ricavato dividendo il numero di dati inseriti (100800) per il tempo complessivo di inserimento. Il grafico riesce a superare i 175 ins/s, mentre la versione con inserimenti su Orion Filter (figura 4.2) arriva al massimo a 140.

Il ginocchio che porta alla saturazione del grafico si trova in corrispondenza dei 200 thread.

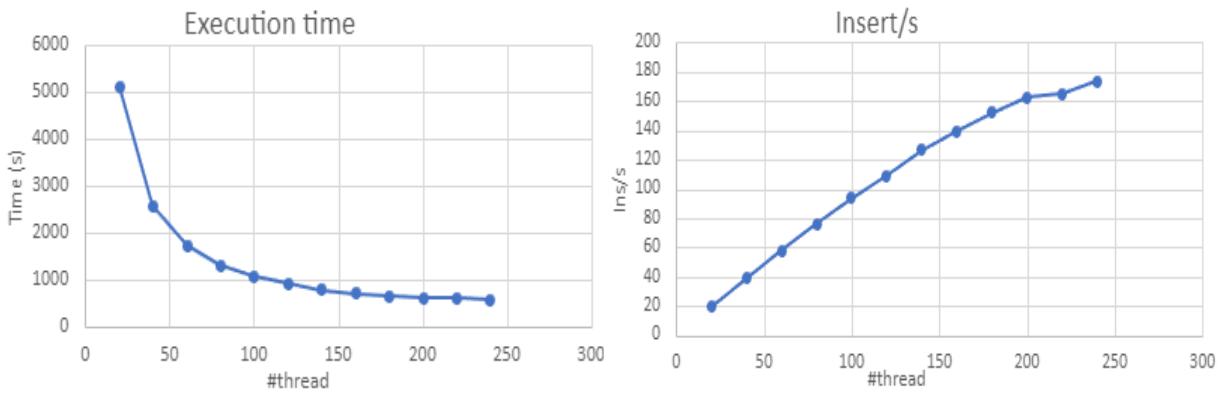


Figura 4.6: *Tempo impiegato dallo script al variare del numero di thread*

Figura 4.7: *Inserimenti al secondo al variare del numero di thread*

I grafici in figure 4.8 e 4.9 derivano dalla rilevazione ogni 2 minuti dei dati di percentuale di CPU e Current Disk Write.

Effettuando un confronto con i grafici in figure 4.3 e 4.4 relativi alla sperimentazione con inserimento su Orion Filter, risulta che la percentuale di CPU si è abbassata, come i valori del Current Disk Write. Questo è dovuto al fatto che non vengono eseguite le operazioni di autenticazione per i dati che impegnava la CPU e che ha bisogno di scrivere sul disco.

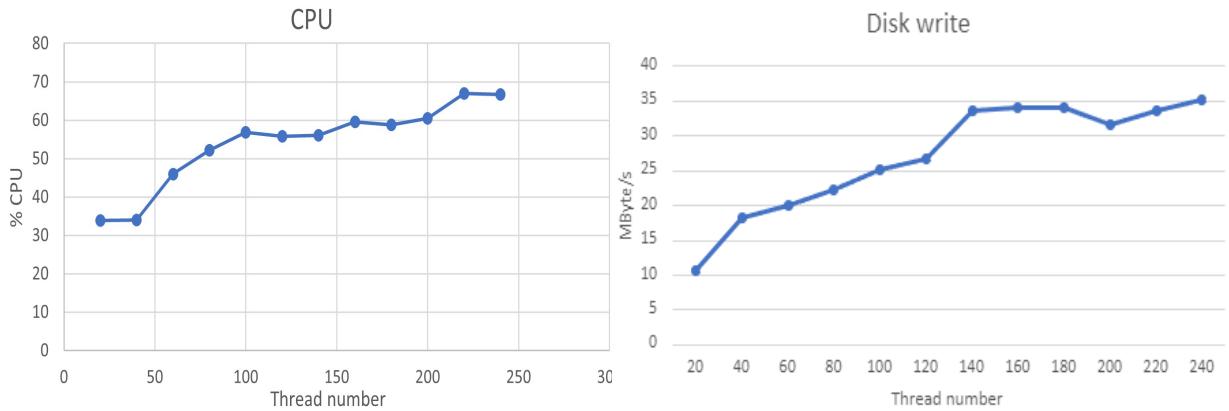


Figura 4.8: Percentuale di CPU in uso al variare del numero di thread

Figura 4.9: Transfer rate al variare del numero di thread

Il grafico in figura 4.10 viene ricavato calcolando la media dei tempi di inserimento di ognuno dei 100800 dati inviati al variare del numero di thread. Il grafico non supera gli 0.3 secondi ad inserimento e ha il gomito in relazione ai 100 thread.

Per gli inserimenti su Orion Filter il grafico (figura 4.5) supera gli 0.6 secondi ad inserimento che sono il doppio rispetto agli inserimenti su Orion Broker.

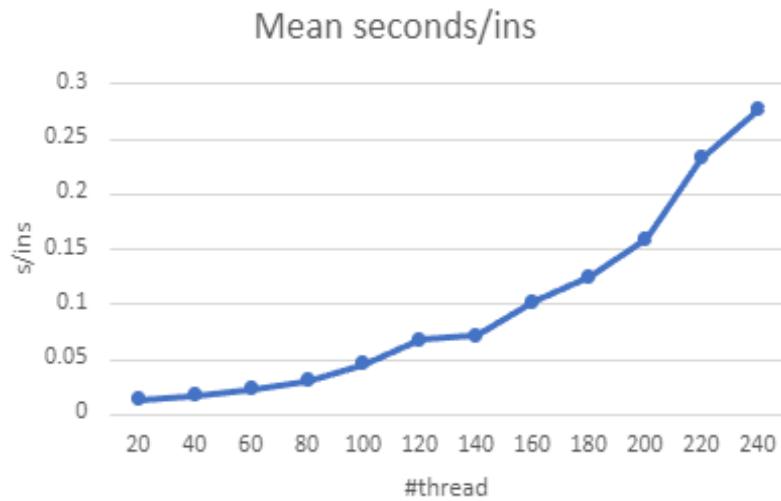


Figura 4.10: Media del tempo di inserimento al variare del numero di thread

4.1.3 Saturazione degli inserimenti

Aumentando progressivamente il numero di thread, il grafico degli inserimenti al secondo tende verso un limite orizzontale che, per gli inserimenti sul Orion Filter corrisponde a 140 Ins/s (figura 4.2), mentre per gli inserimenti su Orion Broker a 175 Ins/s (4.7). Grazie a questa informazione è possibile concludere che, superata una certa soglia di thread in esecuzione, una parte dell'architettura si satura e limita quindi gli inserimenti.

Le figure 4.11 e 4.12 sono il risultato del rapporto fra il CDW ed il numero di inserimenti al secondo al variare del numero di thread, rispettivamente per inserimenti su Orion Filter e su Orion Broker. Grazie a questi grafici è possibile studiare l'efficienza delle due versioni di inserimento in relazione al transfer rate su disco. Il rapporto minimo nel grafico in figura 4.11 si trova con 180 thread, il che significa che, oltre questo valore, inizia la saturazione. Per quanto riguarda la figura 4.12, il minimo corrisponde a 200 thread, infatti anche dai grafici sugli inserimenti al secondo (figure 4.2 e 4.7) si osserva che, inserendo direttamente su Orion Broker, la curva inizia a flettere quando è utilizzato un numero maggiore di thread.

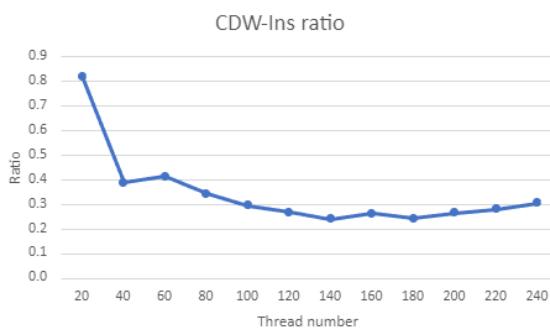


Figura 4.11: Rapporto fra CDW e inserimenti al secondo su Orion Filter

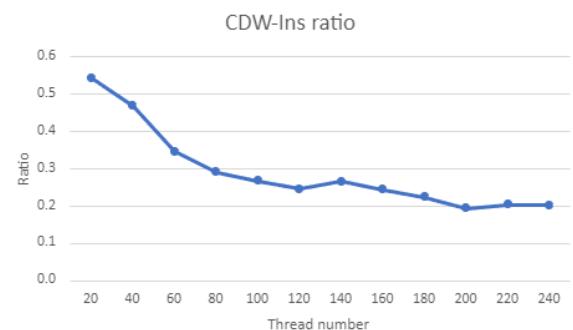


Figura 4.12: Rapporto fra CDW e inserimenti al secondo su Orion Broker

4.1.4 Tempi di attesa

Per cercare una spiegazione al fenomeno della saturazione, viene ripetuta la sperimentazione degli inserimenti, sia su Orion Filter che su Orion Broker, abbassando il tempo di attesa, dopo ogni send, da 1s a 0.5s.

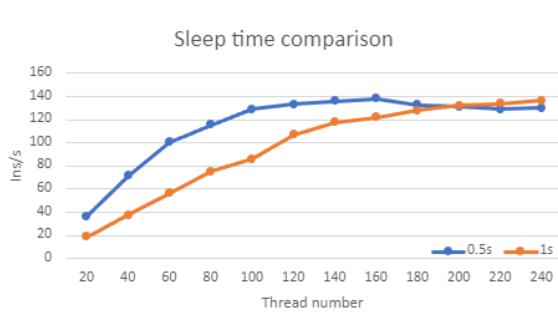


Figura 4.13: Confronto fra i grafici degli inserimenti al secondo su Orion Filter con tempi di sleep differenti

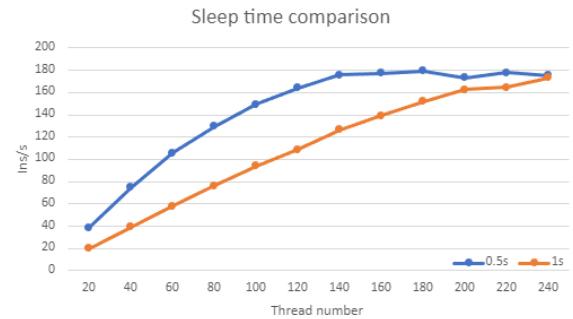


Figura 4.14: Confronto fra i grafici degli inserimenti al secondo su Orion Broker con tempi di sleep differenti

Come si vede nelle figure 4.13 e 4.14, le curve saturano in corrispondenza dello stesso valore, 140 ins/s per gli inserimenti su Orion Filter e 175 ins/s per gli inserimenti su Orion Broker. Attraverso questa analisi è possibile escludere il fatto che la saturazione sia dovuta al tempo atteso da ogni thread fra un inserimento ed il successivo.

4.1.5 Stato della VM micro 1 al termine dell'inserimento

Nella tabella 4.1 sono raccolte le informazioni sullo stato della VM micro 1 al termine dell'inserimento di tutti i dati. Da questa configurazione è patita la seconda sperimentazione.

#Device	3001
#Messaggi complessi per tipologia	50.165.100
#Variabili medie per messaggio	10
#Righe in Open Search	552.003.500
#byte in Open Search	152.686.087.372,8
#GByte Open Search	142,20 GByte
Macchina occupa	220 GByte

Tabella 4.1: *Stato VM micro 1*

4.2 Lettura

Come già anticipato nella Sezione 3.3, la sperimentazione in lettura è ripetuta aumentando il numero di device caricati sulla Knowledge Base (KB):

- 3.000
- 300.000

Per ognuna di esse vengono ricavati grafici sul numero di letture con successo al minuto e la percentuale di CPU utilizzata, sia per le query spaziali che per quelle temporali. Infine vengono raccolti in una tabella i risultati ricavati durante la fase di lettura e inserimento simultanei.

Ciascun esperimento dura 5 minuti, nel quale ogni thread in esecuzione richiede una query e attende 0.1s dopo aver ricevuto la risposta.

4.3 KB con 3.000 device

4.3.1 Query spaziali

I grafici in figure 4.15 e 4.16 rappresentano rispettivamente il numero di query spaziali soddisfatte al minuto e la percentuale di CPU utilizzata sulla VM micro 1, al variare del numero di thread. Il numero di successi al minuto, rappresentato nel grafico in figura 4.15, è ottenuto sommando il numero di query eseguite con successo da ogni thread.

In figura 4.15 si osserva che inizialmente la curva sembra assumere un andamento lineare, ma una volta raggiunte le 5.500 query al minuto si ha una saturazione. La saturazione inizia con 50 thread e in corrispondenza di tale valore, la CPU raggiunge l'80%, e mantiene valori simili anche con un numero maggiore di thread (come mostrato in figura 4.16).

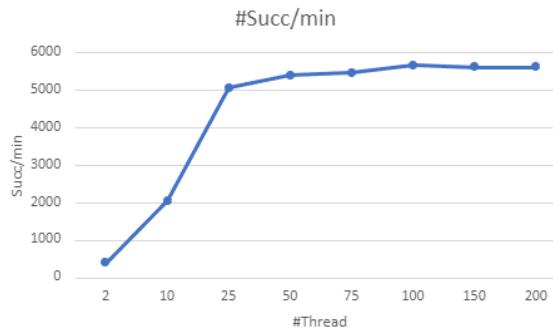


Figura 4.15: Richieste spaziali con successo al minuto al variare del numero di thread

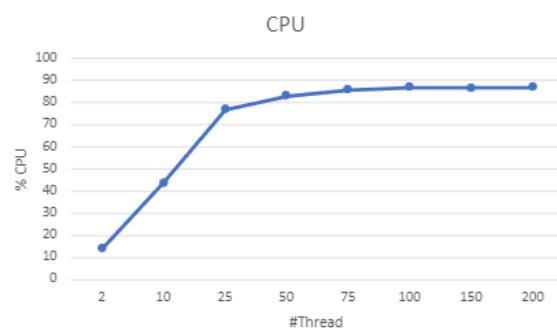


Figura 4.16: Percentuale di CPU utilizzata dalle richieste spaziali al variare del numero di thread

Query temporali

Le condizioni con cui sono eseguite le sperimentazioni sulle query temporali sono del tutto analoghe a quelle per le query spaziali.

Come si vede dalle figure 4.17 e 4.18, le query temporali hanno prestazioni peggiori di quelle spaziali, infatti queste saturano intorno alle 2000 query al minuto e la percentuale di CPU supera l'85%.

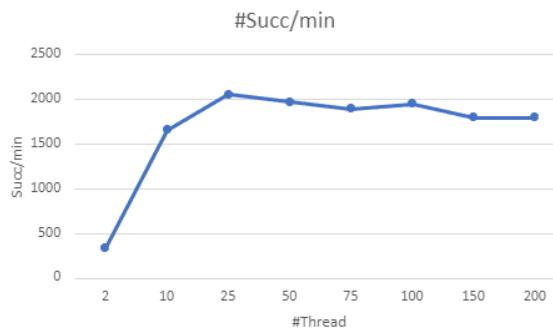


Figura 4.17: Richieste temporali con successo al minuto al variare del numero di thread

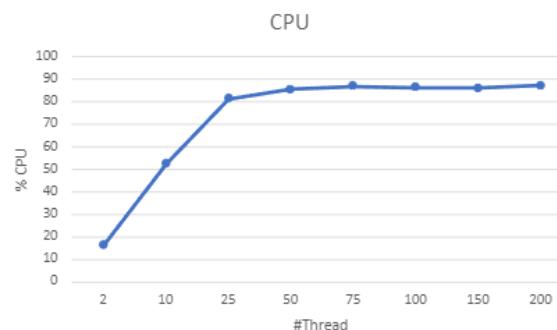


Figura 4.18: Percentuale di CPU utilizzata dalle richieste temporali al variare del numero di thread

4.3.2 Lettura e scrittura

La tabella 4.2 riporta i risultati della sperimentazione con lettura e scrittura contemporanee. Per 5 minuti vengono eseguiti entrambi gli script, impostati in modo tale che l'utilizzo di CPU sia equamente spartito fra le operazioni di scrittura e di lettura. Il valore di "% CPU totale" viene rilevato ogni 30 secondi e rappresenta il valore raggiunto dalla CPU con entrambi gli script in esecuzione.

Per scegliere il numero di thread usati nei due script, vengono utilizzati i grafici della CPU ricavati durante le varie fasi di sperimentazione, andando a trovare il numero di thread che corrisponde alla percentuale di CPU di interesse.

Si prenda come esempio la riga 40% - 40%. Durante gli esperimenti precedenti si vede che le operazioni di scrittura su Orion Filter occupano il 40% della CPU con 50 thread (figura 4.3), mentre le query temporali con 8 thread (figura 4.18). Le colonne "Scrittura da solo" e "Lettura da solo" riportano il numero di inserimenti al secondo e di letture al minuto con quel dato numero di thread quando gli script vengono eseguiti da soli. Nelle colonne "Scrittura insieme" e "Lettura insieme" sono riportati i risultati ricavati durante l'esecuzione simultanea degli script.

Scendendo nel dettaglio, nel caso 40%-40%, le prestazioni sono simili a quando gli script sono eseguiti indipendentemente l'uno dall'altro. Nei casi 50%-50% e 60%-60% invece si notano dei peggioramenti delle prestazioni di inserimento e di lettura. Il numero di inserimenti e di letture diminuisce di un 4.5% nel caso 50%-50% e di un 8% nel caso 60%-60%.

Spartizione %	Scrittura singola (ins/s)	Scrittura insieme (ins/s)	Lettura singola (succ/min)	Lettura insieme (succ/min)	% CPU totale	Differenza scrittura	Differenza lettura
40-40	46	45.4	1631	1650	65.2	-1.5%	+1.2%
50-50	75.2	71.7	1658	1589	69.3	-4.7%	-4.2%
60-60	134	123.2	2055	1902	82	-8.1%	7.4%

Tabella 4.2: *Prestazioni lettura e scrittura contemporanee*

4.4 KB con 300.000 device

4.4.1 Query spaziali

Nelle figure 4.19 e 4.20 sono riportati i risultati della sperimentazione con query spaziali sulla KB con 300.000 device. Confrontando i grafici con quelli relativi alla sperimentazione con 3.000 device in figure 4.15 e 4.16, si nota non solo che hanno forme molto simili, ma anche il valore massimo di successi al minuto è lo stesso.

Per quanto riguarda il grafico in figura 4.20, con più di 75 thread in esecuzione, il valore della CPU supera il 90%, mentre nella sperimentazione con 3.000 device arriva al massimo all'85%.

Passando da 3.000 device a 300.000 device nella KB, viene effettuata una moltiplicazione per un fattore 100 sul numero di device nella KB. Il numero di device (di qualsiasi tipologia) nel cerchio di raggio 1 km dal centro di Firenze passa quindi da 419 a 23.025, e nonostante questo non è presente un calo delle prestazioni.



Figura 4.19: Richieste temporali con successo al minuto al variare del numero di thread

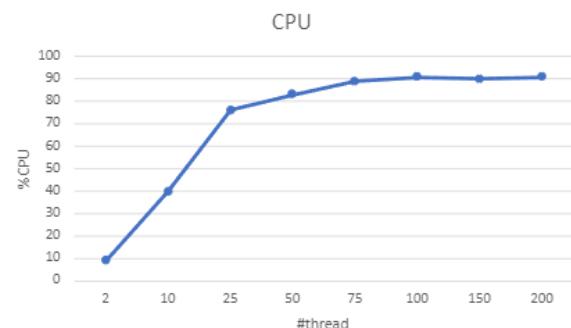


Figura 4.20: Percentuale di CPU utilizzata dalle richieste temporali al variare del numero di thread

4.4.2 Query temporali

Nelle figure 4.21 e 4.22 sono riportati i risultati della sperimentazione con query temporali sulla KB con 300.000 device.

Per quanto riguarda la figura 4.21, il numero di query al minuto si abbassa rispetto all'esperimento con 3.000 device nella KB (figura 4.17). Questo passa da un massimo di 2.000 successi al minuto a 1.600 successi al minuto, con un calo del 20%.

Per soddisfare una query temporale, l'architettura inizialmente ricerca nella KB il device a cui la query fa riferimento e in seguito ricerca i dati in OpenSearch, filtrandoli per la data di rilevazione. Siccome il numero di dati in OpenSearch non è variato, è possibile che il calo di prestazioni nelle query temporali sia dovuto alla ricerca del device nella KB, nella quale sono memorizzati 100 volte il numero di device rispetto alla prima sperimentazione.

Per quanto riguarda la figura 4.22, la percentuale di CPU utilizzata è aumentato fino a superare il 90% con 25 thread in esecuzione. Anche questo dato potrebbe contribuire al calo delle prestazioni rispetto all'esperimento con 3.000 device.



Figura 4.21: Richieste temporali con successo al minuto al variare del numero di thread

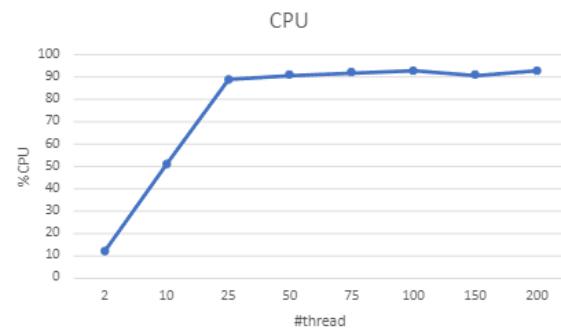


Figura 4.22: Percentuale di CPU utilizzata dalle richieste temporali al variare del numero di thread

4.4.3 Lettura e scrittura

I risultati ottenuti durante l'esperimento di inserimento e lettura simultanei con 300.000 device sulla KB, sono riportati nella tabella 4.3.

A differenza della prova a 3.000 device, in questo caso il calo delle prestazioni non è uniforme fra lettura e scrittura. Le operazioni di lettura risultano molto meno efficienti durante la scrittura e lettura simultanee. Per tutte e tre le prove la lettura peggiora più del 45%, mentre la scrittura risulta migliorata rispetto ai risultati ottenuti nella tabella 4.2 con 3.000 device.

Il valore della CPU totale è più alto, questo può far parte delle possibili cause del calo delle letture, come visto anche nella figura 4.21.

Quando il valore di CPU è troppo alto, viene quindi data precedenza alle operazioni di scrittura rispetto a quelle di lettura, che ne risultano largamente penalizzate.

Spartizione %	Scrittura Singola (ins/s)	Scrittura Insieme (ins/s)	Lettura singola (succ/min)	Lettura insieme (succ/min)	% CPU totale	Differenza scrittura	Differenza lettura
40-40	46	46.4	1307	483	84.2	+0.8%	-63%
50-50	75.2	73.5	1349	534	86.2	-2.3%	-60%
60-60	134	127.9	1598	870	91.8	-4.6%	-45.6%

Tabella 4.3: *Prestazioni lettura e scrittura contemporanee*

4.5 Device mobili

La sperimentazione sui device mobili è incentrata sul determinare il numero massimo di messaggi che Kafka e WebSocket riescono a gestire in *real time*. Per determinare ciò, un primo script python invia per 2 minuti messaggi di aggiornamento dei valori di longitudine, latitudine, value1 e value2, mentre un secondo script è in ascolto su WebSocket e tiene il conto di tutti i messaggi di aggiornamento in ingresso. Lo script in ascolto su WebSocket inoltre misura il tempo che intercorre fra l'arrivo del primo e dell'ultimo messaggio, in modo tale da verificare quando il servizio perde la caratteristica di essere *real time*.

Il grafico riportato in figura 4.23 mostra il delay fra il momento in cui viene inviato l'ultimo dato e il momento in cui esso arriva su WebSocket. Superati i 30.000 messaggi in ingresso su WebSocket, il tempo di delay non è più trascurabile. Questo significa che all'interno dell'architettura si generano delle code che si svuotano solo quando i messaggi smettono di essere inviati. Questo fa sì che il servizio fornito dalla dashboard non sia più *real time*

Continuando ad aumentare il numero di messaggi, la curva dei tempi continua a salire e con 45.000 messaggi la differenza di tempo fra l'invio dell'ultimo messaggio ed il suo arrivo nel WebSocket è di circa 1 minuto.

In figura 4.24 è riportato il grafico della percentuale di CPU utilizzata durante la sperimentazione. In corrispondenza dei 30.000 messaggi in ingresso su WebSocket, il grafico inizia a

salire più velocemente, sintomo del fatto che i messaggi non sono più gestiti in *real time*.

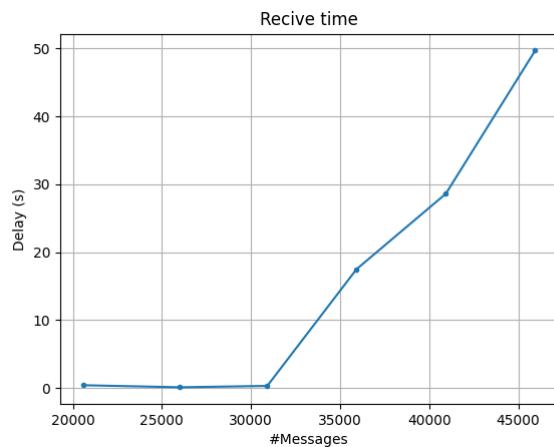


Figura 4.23: *Tempo necessario all’arrivo su WebSocket di tutti i messaggi*

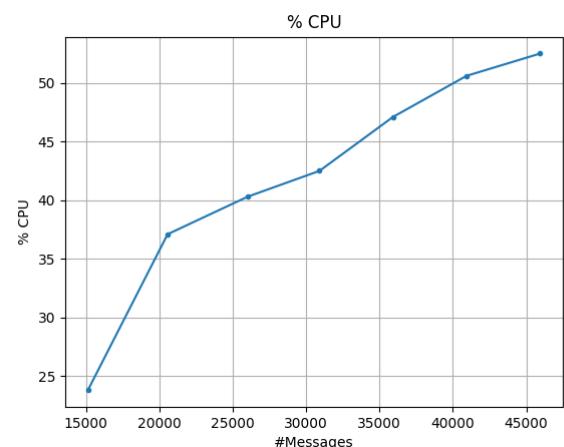


Figura 4.24: *Percentuale di CPU utilizzata all’aumentare del numero di thread*

5

CONCLUSIONI

Dalla ricerca condotta è emerso che tutte le operazioni effettuate portano alla saturazione delle prestazioni dell'architettura.

5.1 Scrittura

Dagli esperimenti di scrittura dei dati nei device è emerso che il numero massimo di inserimenti al secondo è 140. Attraverso gli inserimenti direttamente su Orion Broker è però possibile aumentare questo numero fino a 175, si può quindi concludere che la fase di autenticazione e autorizzazione degli inserimenti impatta per un 20% le prestazioni.

Attraverso lo studio sul Current Disk Write è possibile trovare il numero ottimale di thread da utilizzare per le operazioni di scrittura (figure 4.11 e 4.12), ovvero 180 per la scrittura tramite Orion Filter e 200 per la scrittura tramite Orion Broker.

Grazie alla sezione 4.1.4 dedicata allo studio dei tempi di attesa, è possibile confermare che il fenomeno della saturazione non dipende in alcun modo dal tempo atteso fra un inserimento ed il successivo.

5.2 Lettura

Lo studio sulle operazioni di lettura ha sottolineato che il numero massimo di query spaziali con 30 device nella zona di interesse è 5.500 query al minuto. Ripetendo l'esperimento con 300.000 device nella Knowledge Base (sezione 4.4) il numero di query al minuto non varia. È quindi possibile concludere che le prestazioni dell'architettura non variano se si effettua un filtro per tipologia di device, anche se aumenta il numero totale di device nella zona circoscritta dalla query.

Per quanto riguarda le query temporali, si è passati da un massimo di 2000 query al minuto per la versione con 3.000 device sulla KB a 1.600 con 300.000 device sulla KB. Siccome il numero di dati nei device non è variato, si può quindi assumere che il peggioramento delle prestazioni sia dovuto alla fase di ricerca nella KB del device specificato nella query.

5.3 Scrittura e Lettura contemporanee

I risultati riscontrati durante la sperimentazione di lettura e scrittura contemporanee sono molto diversi con 3.000 device e 300.000 device. La prima sperimentazione ha riportato una diminuzione uniforme delle prestazioni di scrittura e lettura, mentre nella seconda la lettura ha avuto un calo decisamente più significativo. Analizzando le percentuali complessive di CPU utilizzate, si può ipotizzare che quando la CPU viene stressata eccessivamente, le operazioni di scrittura sono privilegiate rispetto a quelle di lettura.

5.4 Device mobili

Attraverso lo studio effettuato sui device mobili è possibile identificare il numero massimo di messaggi di aggiornamento della posizione di device mobili, tali che il servizio risulti comunque *real time*.

È da sottolineare che, anche quando il numero di messaggi inviati è superiore a quelli che l'architettura riesce a processare in *real time*, questi comunque vengono ricevuti da WebSocket (anche se in ritardo) e quindi non si ha una perdita di dati.

5.5 Sviluppi futuri

Come prosecuzione di questo studio, si può pensare di effettuare sperimentazioni mirate a determinare quali siano le componenti che portano alla saturazione delle letture e delle scritture. Una volta individuata tale componente sarebbe possibile migliorarne le prestazioni e ripetere la sperimentazione per osservare i cambiamenti rispetto a questo caso base.

Per quanto riguarda l'inserimento su device mobili, si potrebbe pensare di analizzare quali siano i limiti di WebSocket in relazione al numero di utenti connessi alle dashboard; infatti WebSocket apre una connessione diversa per ogni utente che visualizza la stessa dashboard e questo può determinare un limite all'aggiornabilità della dashboard stessa.

Tutta l'analisi effettuata risulterà utile nel momento in cui verrà richiesto un miglioramento delle prestazioni di Snap4City, difatti si saprà esattamente quali siano le componenti che maggiormente limitano le prestazioni della piattaforma, le quali saranno le prime a dover essere migliorate.



SITOGRAFIA

- Documentazione NiFi:
<https://nifi.apache.org/docs.html>
- Documentazione OpenSearch:
<https://opensearch.org/>
- Documentazione MongoDB:
<https://www.mongodb.com/docs/>
- Snap4City:
<https://www.snap4city.org/>
- ServiceMap:
<https://servicemap.disit.org/WebAppGrafo/>
- Apache Kafka:
<https://kafka.apache.org/20/documentation.html>