UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI INGEGNERIA
ELETTRICA ELETTRONICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

RELAZIONE FINALE DEL PROGETTO IN ITINERE DELL'INSEGNAMENTO DISTRIBUTED SYSTEMS AND BIG DATA

ANNO ACCADEMICO 2022-2023

STUDENTI:

Luigi Fontana N. 1000037171

Giuseppe Testa N.1000022605

Sommario

1. Introduzione

1.1 Abstract

L'elaborato prevede la creazione di un'applicazione formata da più microservizi che permetta di monitorare le metriche esposte da un server Prometheus. Il server Prometheus utilizzato è stato fornito dal professore G. Morana ed è raggiungibile all'Url *http://15.160.61.227:29090*. Questo server fa uno scraping delle metriche esposte dell'exporter. Dopo aver identificato le metriche da analizzare, sono stati prodotti i vari dati tramite ETL Data Pipeline, inviati su un topic Kafka di nome "prometheusdata", ed una volta consumati dal Data Storage, sono stati inviati al database. È stato implementato in ETL Data Pipeline, tramite REST API, un sistema di monitoraggio interno.
I dati salvati nel database sono disponibili tramite REST API implementate nel Data Retrieval che tramite delle query comunica con il database.
Infine, è stato sviluppato un meccanismo, nell'SLA Manager, per la verifica delle violazioni passate e future. Tramite REST API è possibile visualizzare i vari dati ottenuti.

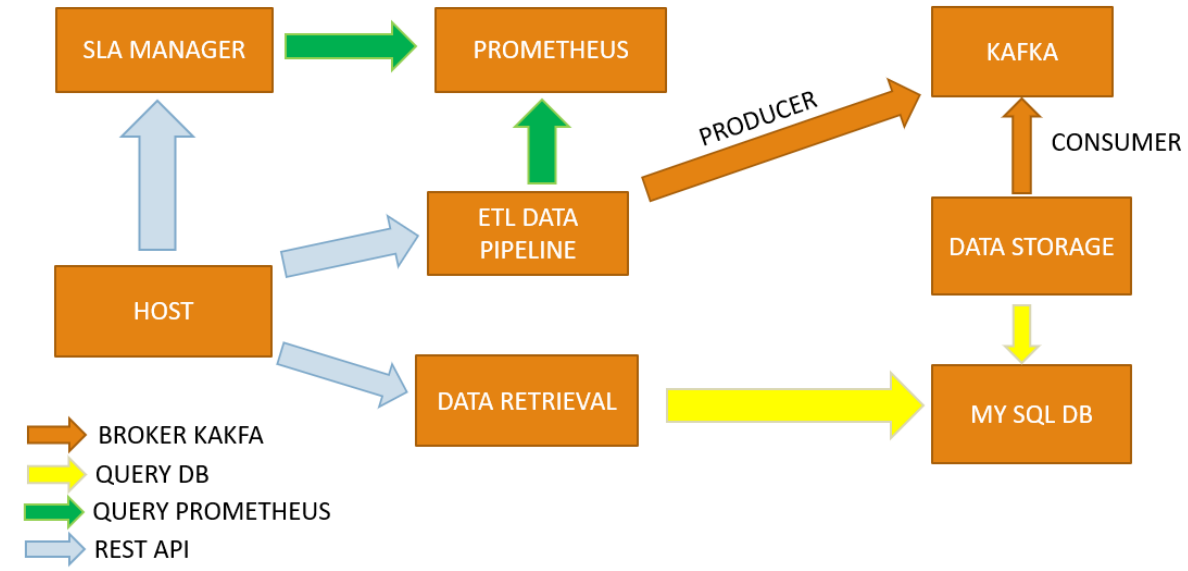

Figura 1 - Diagramma dei micro-servizi

1.2 Guida all'uso

nota:
- per sistemi windows based eliminare sudo dai comandi;
- i comandi devono essere lanciati nella cartella in cui sono presenti i vari docker compose;
- i file sono all'interno della cartella fontana_testa.

Passi:
- Avviare Docker o installare Docker
- Tramite terminale creare una network con il comando:

-> sudo docker network create monitoring

Per verificare la presenza della network:

-> sudo docker network ls

Per poter funzionare correttamente bisogna avviare due diversi docker-compose.

Andare nella cartella in cui sono presenti i file. E poi:

- Avviare il docker-compose all'interno della cartella KafkaDB tramite comando:

->sudo docker-compose up -d

All'interno di questo docker-compose sono presenti zookeeper, kafka e mysql.

Su un terminale eseguire il comando sudo docker ps. In questo modo verifichiamo che i container sono attivi.

- Prima di far partire il secondo compose, bisogna creare le tabelle all'interno del DB. Dunque, eseguire i seguenti comandi:

->sudo docker exec -it ID_CONTAINER_SQL bash (ID_CONTAINER si legge dal docker ps)

->mysql -u root -p metrics (La password è toor. In questo modo entriamo già dentro il DB)

Dunque dentro il DB creiamo le varie tabelle con i comandi:

- CREATE TABLE metrics (ID INT AUTO_INCREMENT, metric varchar(255),max DOUBLE, min DOUBLE, mean DOUBLE, dev_std DOUBLE, duration varchar(255) ,PRIMARY KEY (ID));
- CREATE TABLE autocorrelation (ID INT AUTO_INCREMENT, metric varchar(255),value DOUBLE, duration varchar(255),PRIMARY KEY(ID));
- CREATE TABLE seasonability (ID INT AUTO_INCREMENT, metric varchar(255),value DOUBLE,duration varchar(255), PRIMARY KEY(ID));
- CREATE TABLE stationarity (ID INT AUTO_INCREMENT, metric varchar(255),p_value DOUBLE,critical_values varchar(255),duration varchar(255), PRIMARY KEY(ID));
- CREATE TABLE prediction_mean (ID INT AUTO_INCREMENT, metric varchar(255), timestamp varchar(255), value varchar(255), duration varchar(255),PRIMARY KEY(ID));
- CREATE TABLE prediction_min (ID INT AUTO_INCREMENT, metric varchar(255), timestamp varchar(255), value varchar(255), duration varchar(255),PRIMARY KEY(ID));
- CREATE TABLE prediction_max (ID INT AUTO_INCREMENT, metric varchar(255), timestamp varchar(255), value varchar(255), duration varchar(255),PRIMARY KEY(ID));

Nel caso qui presente, il topic è già creato nel compose. In alternativa si può creare il topic tramite comando sul terminale:

-> sudo docker exec -it kafkdb_kafka_1 kafka-topics --create --replication-factor 1 --partitions 1 --topic prometheusdata --bootstrap-server localhost:9092.

- Eseguire il docker-compose presente nella cartella principale:

->docker-compose up -d

In questo modo avviamo tutti i microservizi. È necessario aspettare qualche minuto poiché devono essere scaricati ed installati.

Per poter vedere i vari risultati, aspettare qualche minuto per permettere il processamento dei dati.

-Tramite estensione di google chrome talend api tester, postam o su qualsiasi applicazione che permette di fare richieste get e post inserire i vari url per osservare i risultati.

Elenco Metriche:

- availableMem
- cpuLoad
- cpuTemp
- diskUsage
- inodeUsage
- networkThroughput
- push_time_seconds
- realUsedMem

Elenco comandi REST API (La descrizione si trova nelle sottosezioni):

ETL

GET -> http://localhost:5000/metrics/1h
GET -> http://localhost:5000/metrics/3h
GET -> http://localhost:5000/metrics/12h
GET -> http://localhost:5000/regen_data
POST -> http://localhost:5000/forecasting
GET -> http://localhost:5000/all_data

DataRetrieval

GET: http://localhost:5005/metrics/metric/<nome_della_metrica>
GET: http://localhost:5005/metrics/metadati/<nome_della_metrica>
GET: http://localhost:5005/metrics/forecasting/<nome_della_metrica>
GET: http://localhost:5005/metrics

SLA Manager

POST: http://localhost:5002/SLA
GET: http://localhost:5002/assess_Violations
NOTA: Vanno eseguite per prime queste due richieste per poter far funzionare tutto.
GET: http://localhost:5002/get_Violations
GET: http://localhost:5002/get_Violations_Num
GET: http://localhost:5002/get_SLA_status
GET: http://localhost:5002/get_SLA_pred
GET: http://localhost:5002/get_SLA_pred_status

Nota: La descrizione di tutte le richieste è nelle relative sottosezioni nelle pagine successive.

1.3 Scelte implementative iniziali

La prima operazione svolta è stata la scelta delle metriche da monitorare. Le metriche da noi scelte appartengono al job:'summary' ed all'istance:'106'.
Nella figura 1 è presente uno snapshot del risultato della query effettuata su Prometheus.

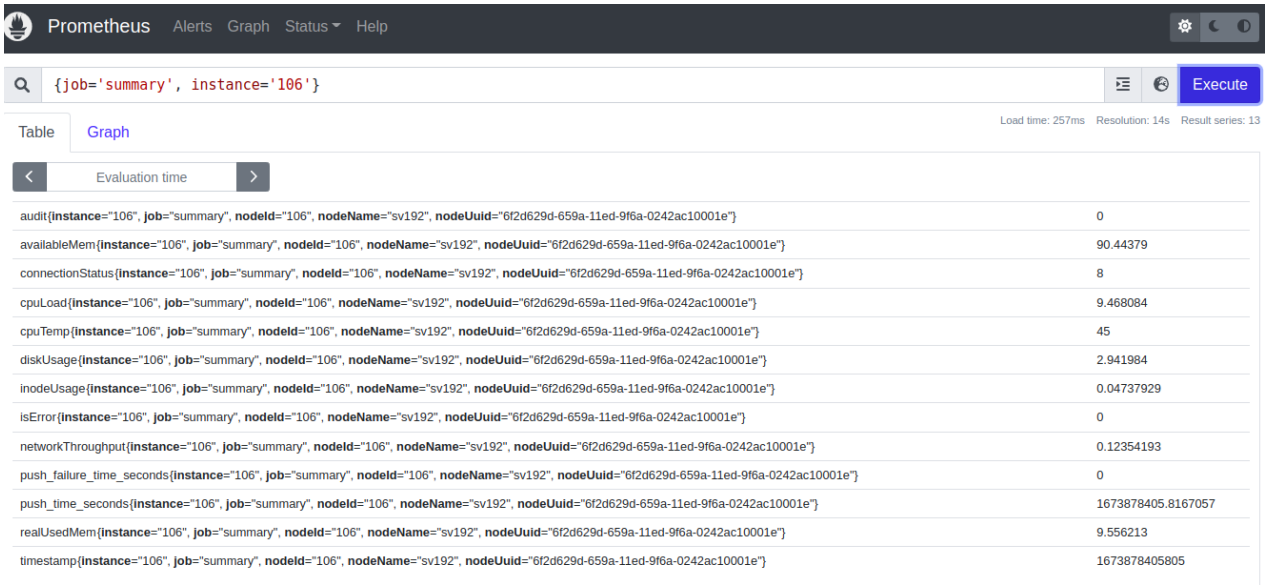

Figura 2 - Metriche esposte alla query {job='summary', instance='106'}

Una volta identificate le metriche da voler analizzare, sono stati sviluppati i vari microservizi. I microservizi sono stati implementati all'interno di container separati utilizzando la piattaforma Docker. Sono stati creati due differenti docker compose:

- Un docker-compose in cui verranno creati i container per kafka, zookeeper e mysql. All'interno di questo docker compose è stato creato il topic "prometheusdata", che verrà sfruttato dal producer e dal consumer, ed il database "metrics";

- Un docker-compose in cui verranno creati i container contenenti i microservizi rimanenti. Questi eseguiranno i Dockerfile necessari ad avviare il codice.

Questa scelta permette di far funzionare il tutto tramite docker o facendo partire i vari file python da terminale.
Questi compose contengono gli strumenti e la configurazione necessaria per il funzionamento dell'applicativo.
In entrambi i compose è indicata una network creata appositamente per lo scambio di informazioni tra i vari container.

2.ETL Data Pipeline

Viene implementato nel file ETL_dataPipeline.py. Questo microservizio si connette al broker di kafka come producer.
La prima azione effettuata dal microservizio è chiamare la funzione *metric_scapring()* che permette di fare una query all'indirizzo prometheus e recuperare le metriche di cui voler fare il monitoraggio. Viene fatto un filtraggio basato sul label name e poi viene fatto un ulteriore filtraggio per eliminare le metriche di cui non si aveva bisogno. Con questo filtraggio sono tetrovate 8 metriche di cui effettuare il monitoraggio:
- availableMem
- cpuLoad
- cpuTemp
- diskUsage
- inodeUsage
- networkThroughput
- push_time_seconds
- realUsedMem

Dopo aver fatto la query, il risultato è stato convertito in dataframe e sono stati calcolati:

- Valori di massimo, minimo, media e deviazione standard in un'ora, tre e dodici ore di funzionamento con i metodi di pandas;
- Un set di metadati comprendente autocorrelazione, stazionarietà e stagionalità.
  - L'autocorrelazione in python è stata calcolata utilizzando la funzione "acf" del pacchetto "statmodels".
  - la stazionarietà è stata calcolata utilizzando il test di Dickey-Fuller tramite la funzione "adfuller".

- la stagionalità è stata calcolata tramite la funzione "seasonal_decompose" della libreria "statsmodels.tsa.seasonal".

Sono stati inviati tramite kafka tutti i valori che sono stati generati in modo da poterne prevedere in futuro un riutilizzo.

- Calcolata una predizione relativa al massimo, minimo e media per i 10 minuti posteriori alla chiamata. Le metriche di cui fare la predizione sono definite all'inizio in una lista, ma possono essere cambiate o tramite una chiamata REST oppure quando viene modificato l'SLA set nell'SLA Manager. La predizione è stata effettuata utilizzando la funzione 'ExponentialSmoothing' della libreria 'statsmodels.tsa.holtwinters'.
  Per poter fare la predizione, la dataframe è stata ricampionata, con la regola rule = "2T", e la predizione è stata fatta di 5 campioni in modo da risultare nei 10 minuti successivi.
- I valori calcolati vengono inoltrati, in formato JSON, su di un topic Kafka denominato "prometheusdata".

Infine, è stato creato il sistema di monitoraggio per calcolare i tempi con cui sono stati generati i vari valori all'interno della funzione. Questi valori sono esposti creando delle REST API sfruttando il framework di python Flask. Le API create sono:

- *http://localhost:5000/metrics/1h:* funzione GET che ritorna il JSON con i valori temporali di monitoraggio relativi ad un'ora di computazione.
- *http://localhost:5000/metrics/3h:* funzione GET che ritorna il JSON con i valori temporali di monitoraggio relativi a tre ore di computazione.
- *http://localhost:5000/metrics/12h:* funzione GET che ritorna il JSON con i valori temporali di monitoraggio relativi a 12 ore di computazione.
- *http://localhost:5000/regen_data:* funzione GET per ricalcolare tutti i dati presenti nella funzione *metric_scraping.*
- *http://localhost:5000/forecasting:* funzione POST che permette di aggiornare la lista di metriche di cui voler fare la predizione. Il formato JSON da inviare è del tipo:

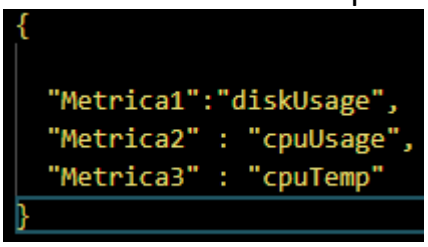

Figura 3 - Formato Json Post forecasting

- *http://localhost:5000/all_data:* funzione GET per restituire tutti i dati di monitoraggio con una sola chiamata.

- *http://localhost:5000/test_forecast:* funzione GET per testare il corretto funzionamento dell'aggiornamento della lista da parte dell'SLA Manager.

3.Data Storage

Viene implementato nel file DataStorage.py. Questo microservizio si connette ad un database mysql e a Kafka come consumer, facendo una subscribe al topic "prometheusdata". È stato configurato il parametro auto.offset.reset a latest, in maniera tale da prendere solamente i dati che sono stati inviati successivamente all'avvio del consumer. Successivamente, dopo la connessione al database, tramite l'oggetto Consumer della libreria confluent_kafka viene effettuato un polling continuo per leggere i dati che vengono inseriti nel topic.
I dati prodotti vengono inviati al database chiamato "metrics" e alle rispettive tabelle, che sono state create precedentemente. Le tabelle create sono:

- metrics: in cui sono inseriti i valori di massimo, minimo, media e deviazione standard per ogni metrica;
- autocorrelation: in cui sono inseriti i valori di autocorrelazione calcolati per ogni metrica;
- seasonability: in cui sono inseriti i valori di stagionalità calcolati per ogni metrica;
- stationarity: in cui sono inseriti il valore di p-value ed i valori critici calcolati per ogni metrica;
- prediction_max: in cui sono inseriti i valori di predizione del massimo nei 10 minuti successivi per ogni metrica;
- prediction_min: in cui sono inseriti i valori di predizione del minimo nei 10 minuti successivi per ogni metrica;
- prediction_mean: in cui sono inseriti i valori di predizione della media nei 10 minuti successivi per ogni metrica.

4.Data Retrieval

Viene implementato nel file DataRetrieval.py. Questo microservizio permette di recuperare le informazioni generate dall'ETL Data Pipeline e salvate nel database.

Il microservizio sfrutta il framework di python Flask per poter offrire una interfaccia REST che con le varie funzioni effettua delle query al database "metrics".

Le query create sono:

- *http://localhost:5005/metrics:* funzione GET effettua più query in modo da recuperare tutti i valori di tutte le metriche presenti all'interno del database.
- *http://localhost:5005/metrics/metric/<nome_della_metrica>:* funzione GET in cui inserendo il nome della metrica, vengono effettuate delle query al database per restituirne i valori di massimo, minimo, media e deviazione standard nelle ultime 1, 3, 12 ore di quella singola metrica.
- *http://localhost:5005/metrics/metadati/<nome_della_metrica>:* funzione GET in cui inserendo il nome della metrica, vengono effettuate delle query al database per restituire i valori dei metadati di quella singola metrica.
- *http://localhost:5005/metrics/forecasting/<nome_della_metrica>:* funzione GET in cui inserendo il nome della metrica, vengono effettuate delle query al database per restituire i valori delle predizioni di quella singola metrica.

I nomi delle metriche da poter inserire sono:

- availableMem
- cpuLoad
- cpuTemp
- diskUsage
- inodeUsage
- networkThroughput
- push_time_seconds

- realUsedMem

5.SLA Manager

Viene implementato nel file SLA_Manager.py. Questo microservizio, dopo essersi connesso al Prometheus che è stato fornito, permette di andare a verificare se ci sono delle violazioni in 1 ora, 3 ore e 12 ore di computazione e di fare una predizione su una possibile violazione nei successivi 10 minuti.
Le metriche su cui vengono effettuate queste analisi sono determinate tramite una funzione POST; questo microservizio, infatti, sfrutta il framework di python Flask per poter offrire una interfaccia REST.
Il set di metriche, SLA set, che viene inserito, va anche ad aggiornare la lista di metriche da predire nell'ETL data pipeline.
Le REST API create sono:

- *http://localhost:5002/SLA:* funzione POST che tramite un formato JSON permette di riempire la lista delle metriche su cui fare le operazioni ed i range ammissibili. Un esempio di formato è presente nella figura sottostante.

```
Formattazione del messaggio json per la richiesta post
{
  "availableMem": [0, 90.44],
  "cpuLoad"     : [0, 10.00],
  "cpuTemp"     : [0, 44.00],
  "diskUsage"   : [0, 2.9487],
  "realUsedMem" : [0, 9.62]
}
```

Figura 4 - Formattazione del messaggio json per la POST

- *http://localhost:5002/assess_Violations:* funzione GET che richiama la funzione metric_scraping per generare i dati. Generati i dati vengono chiamate due funzioni:
    - *range_violation*: questa funzione permette di calcolare il numero di violazioni che ci sono state in 1 ora, 3 ore e 12 ore.
    - *future_violations*: questa funzione permette, tramite la funzione ExponentialSmoothing, di effettuare la predizione di una possibile violazione nei successivi 10 minuti. In questo caso, la dataframe è stata ricampionata con la rule = "1T", pertanto, i

campioni calcolati sono 10 campioni a rappresentare i 10 minuti nel futuro.

Queste prime due REST devono essere eseguite per prime in modo da poter permettere il corretto funzionamento delle successive funzionalità:

- *http://localhost:5002/get_Violations:* funzione GET che mostra quando sono avvenute le varie violazioni.
- *http://localhost:5002/get_Violations_Num:* funzione GET che mostra il numero di violazioni che sono avvenute in 1 ora, 3 ore e 12 ore.
- *http://localhost:5002/get_SLA_status:* funzione GET che mostra il numero di violazioni che sono avvenute in 1 ora, 3 ore e 12 ore divise per nome metrica.
- *http://localhost:5002/get_SLA_pred:* funzione GET che mostra l'eventuale lista di violazioni nei successivi 10 minuti.
- *http://localhost:5002/get_SLA_pred_status:* funzione GET che mostra il numero di possibili violazioni divise per il nome della metrica.

Un esempio di nomi di metriche e relativi range possibili da poter inserire sono:
- availableMem [0, 90.44];
- cpuLoad [0, 10.00];
- cpuTemp [0, 44.00];
- diskUsage [0, 2.9587];
- inodeUsage [0, 0.04738];
- networkThroughput [0, 0.1226];
- push_time_seconds [0, 1673963905.81]; (è consigliabile non fare la predizione di questo poiché è un valore che va sempre a crescere)
- realUsedMem [0, 9.62];

I range delle metriche dipendono dal server Prometheus che è stato fornito; pertanto, si consiglia di effettuare un'analisi dei dati prima di aggiornare i range poiché questi ultimi sono stati esemplificati a tempo di stesura del documento.