Corso di Laurea Magistrale in Ingegneria Informatica – Anno Accademico 2022/2023

DISTRIBUTED SYSTEMS AND BIG DATA

Membri gruppo del progetto:

- Marco Giuseppe Biondo 1000014390
- Giuseppe La Vecchia 100037548

Abstract

Per lo sviluppo del progetto si è deciso di utilizzare cAdvisor come applicazione da cui prelevare metriche usando il Prometheus fornitoci. E' stato deciso di utilizzare il metodo USE con particolare attenzione sull'Utilization delle memorie di alcuni container.

Le metriche definite all'interno dello SLA Manager sono prelevate dal primo micro-servizio (ETL Data Pipline) e successivamente saranno prelevati i dati delle rispettive metriche esposte da Prometheus e successivamente processati. Si è deciso di usare due approcci differenti per le predizioni, tra cui ARIMA ed Exponential Smoothing. Si è deciso di optare per un sistema di monitoraggio interno per la visualizzazione delle tempistiche computazionali visibili tramite log. I dati processati verranno strutturati e poi inviati come messaggio ad un topic Kafka chiamato "prometheusdata" tramite un producer asincrono.

Il secondo micro-servizio è composto da un consumer che si occuperà di prelevare i dati dal topic "prometheusdata" e di inserirli all'interno di un db mySql.

Nel terzo micro-servizio si è deciso di utilizzare un'interfaccia di tipo REST per la visualizzazione delle informazioni generate dal primo micro-servizio che saranno prelevate da un db mySql.

Anche nel quarto micro-servizio si è deciso di utilizzare un'interfaccia di tipo REST per la definizione dello SLA Set in base alle metriche da noi selezionate che sono state definite e successivamente inserite all'interno del db.

Schema architetturale dei micro-servizi e delle relative comunicazioni

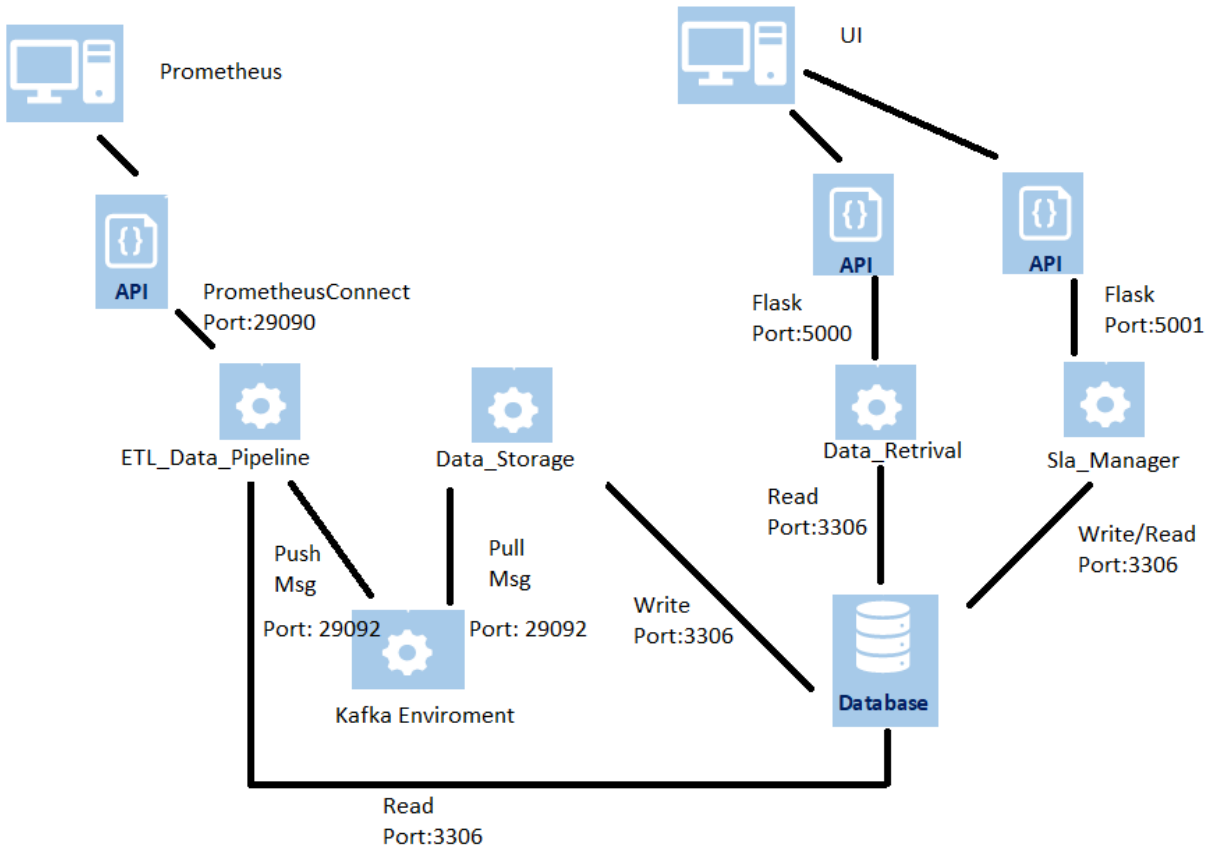

Figura 1.1 Schema microservizi e relative comunicazioni

```
.
├── docker-compose.yml
└── dump
    ├── data_retrival
    │   ├── Dockerfile
    │   ├── microServ3.py
    │   └── requirements.txt
    ├── data_storage
    │   ├── Dockerfile
    │   ├── microServ2.py
    │   └── requirements.txt
    ├── etl
    │   ├── Dockerfile
    │   ├── microServ1.py
    │   └── requirements.txt
    ├── mysql
    │   ├── Dockerfile
    │   └── initdb.sql
    └── sla_manager
        ├── Dockerfile
        ├── microServ4.py
        └── requirements.txt
```

Figura 1.2 Architettura progettuale

Descrizione dell'applicazione

Etl Data Pipeline:

microServ1.py

```
29  kafka_server=(str(os.environ["KAFKA_SERVER"]))
30
31  db_server=(str(os.environ["DB_SERVER"]))
32  prom = PrometheusConnect(url="http://15.160.61.227:29090", disable_ssl=True)
```

Figura 2.0.1Prelievo environment variables e connessione Prometheus

Nelle variabili kafka_server e db_server viene salvato l'indirizzo del container di Kafka e del Database. Con la PrometheusConnect avviene la connessione al Prometheus e si definisce una variabile che permette di eseguire le operazioni.

```
121     try:
122         myresult=[]
123         mydb = mysql.connector.connect(host=db_server,user="root",password="root",  database="dsbd")
124     except mysql.connector.Error as err:
125       if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
126         print("Something is wrong with your user name or password")
127       elif err.errno == errorcode.ER_BAD_DB_ERROR:
128         print("Database does not exist")
129
130     else:
131         mycursor = mydb.cursor(dictionary=True)
132         mycursor.execute("SELECT * FROM Sla;")
133         myresult = mycursor.fetchall()
134         mycursor.close()
135
136     for i in myresult:
137         print("Inizio Logs:")
138         dict_dati={}
139         dict_dati["metric"]=i["metric"]
140         dict_dati["metric_id"]=str(i["ID"])
```

Figura 2.0.2 Prelievo metriche da db

Viene eseguita una connessione al db tramite connector e viene eseguita una query per prelevare le varie metriche all'interno della tabella Sla (all'interno di questa tabella saranno presenti il nome della metrica con il relativo label e vi sarà un campo che definisce se la metrica è nello SLA_Set che sarà di tipo booleano: 0 se non è nel Set, 1 se lo è. Vi saranno anche dei campi di range min e max che serviranno per le violazioni, tali dati sono presenti all'interno di un file chiamato initdb.sql che verrà chiamato dall'entrypoint che verrà eseguito quando verrà fatto il comando docker-compose up che effettuerà l'inserimento all'interno della tabella Sla). I vari dati che dovranno essere salvati dal secondo microservizio nella tabella Metrics vengono inseriti in un dizionario dict_dati.

```
162                 media=vm.mean()
163                 varianza=vm.var()
164                 dati_normalizzati=metric_range_data_df['value']-media
165                 acorr=numpy.correlate(dati_normalizzati,dati_normalizzati,'full')[len(dati_normalizzati)-1:]
166                 acorr = acorr / varianza / len(dati_normalizzati)
167                 print(compute_time(start_time,"calcolo autocorrelazione ",i["metric"]))
168                 acorr = pd.Series(acorr)
169                 c=acorr.to_string(index=False)
170                 arr = c.split()
171                 accdic={}
172                 z=0
173                 for ar in arr:
174                     accdic["value"+str(z)]=ar
175                     z=z+1
176                 dict_dati["autocorr"]=accdic
177                 start_time=time.time()
178                 sd=seasonal_decompose(vm,model="additive",period=6)
179                 print(compute_time(start_time,"calcolo seasonal ",i["metric"]))
180                 fg=sd.seasonal.index
181                 fg=fg.to_series().to_string(index=False)
182                 fg=fg.replace('timestamp','')
183                 fg=fg.replace(' ','')
184                 fg = fg.split()
185                 arr=sd.seasonal.to_string(index=False)
186                 arr=arr.replace('timestamp','')
187                 arr = arr.split()
188                 dictionary = dict(zip(fg, arr))
189                 dict_dati["seasonal"]=dictionary
```

Figura 2.0.3 Funzione di autocorrelazione e stagionalità

Con queste funzioni verranno calcolate la funzione di autocorrelazione e la stagionalità.

```
58  def adfuller_test(isStationary):
59      result=adfuller(isStationary)
60      if result[1] <= 0.05:
61          return True
62          #print("strong evidence against the nu
63          #vai a funzione per ARIMA
64
65      else:
66          return False
67          #print("weak evidence against null hyp
68          #vai a funzione per expsmoothing
```

Figura 2.0.4 Stazionarietà tramite Augmented Dickey-Fuller

Tramite adfuller_test determiniamo la stazionarietà della serie temporale e, se stazionario, si passerà alla funzione del calcolo delle predizioni con ARIMA in quanto l'algoritmo richiede che la serie sia stazionaria. Se non è stazionaria, si utilizzerà la funzione di Exponential Smoothing per le predizioni.

```python
def arima(forPrediction):
    forPrediction=forPrediction.resample(rule='T').mean()
    train = forPrediction.iloc[0:len(forPrediction)-10]
    test = forPrediction.iloc[len(forPrediction)-10:]
    errori=[]
    start_ts=len(train)
    end_ts=len(train)+len(test)-1
    for p in range(7):
        for q in range(7):
                model=ARIMA(train,order=(p,0,q))
                results=model.fit()
                predictions = results.predict(start=start_ts,end=end_ts,dynamic=False,typ="levels")
                error = rmse(test, predictions)

                arima_object=tipo_arima(p,q,error)
                errori.append(arima_object)
    oggettominimo=tipo_arima(0,0,0.00000000)
    for i in range(len(errori)):
        if(errori[i].errore<oggettominimo.errore or i==0):
            oggettominimo=errori[i]

    model=ARIMA(train,order=(oggettominimo.p,0,oggettominimo.q))
    results=model.fit()
    predictions = results.predict(start=start_ts,end=end_ts+9,typ="levels")
    return predictions
```

Figura 2.0.5 ARIMA

Dopo aver effettuato il ricampionamento dei dati prelevati da Prometheus relativi ad 1h, la serie risultante viene suddivisa in train e test. Dopo aver iterato su possibili valori di p e q, viene scelto il modello con l'rmse minore, comparando i valori di test (dunque i valori attesi) con i risultati predetti. Il parametro d di ARIMA viene posto uguale a 0 perché la serie è stazionaria, dunque non necessita di differenze. Infine, viene effettuata la predizione dei valori nei successivi 10 minuti utilizzando il modello con il minore errore.

```python
def exponential_smoothing(forPrediction):
    forPrediction=forPrediction.resample(rule='T').mean()
    model=ExponentialSmoothing(forPrediction,trend="additive",seasonal="mul",seasonal_periods=12)
    model=model.fit()
    predictions=model.predict(start=len(forPrediction),end=len(forPrediction)+9)
    return predictions
```

Figura 2.0.6_Exponential Smoothing

Si è scelto di utilizzare seasonal_periods=12 in quanto vengono prelevati chunk di dati ogni 5 minuti per 1h (60 minuti / 5 minuti =12).

```python
def compute_time(start_time,control_type,metric_name):
    end_time=time.time()
    total_time_exe=end_time-start_time
    return metric_name+":Tempo esecuzione per "+control_type+" ="+str(total_time_exe)
```

Figura 2.0.7 Calcolo tempo di computazione

Funzione che restituisce il tempo di computazione che verrà mostrato nei log. Start_time viene calcolato all'inizio di ogni funzionalità.

```
230         result = str(dict_dati)
231         conf = {'bootstrap.servers': kafka_server+':29092'}
232         topic="promethuesdata"
233         # Create Producer instance
234         p = Producer(**conf)
235         p.produce(topic, result)
236         p.poll()
```

Figura 2.0.8 Producer

Viene instanziato un Producer che invia il messaggio contenente la conversione del dict_dati in string al topic “prometheusdata”.

```
151             if(i["sla_set"]):
152                 incr=0
153                 for med in vm:
154                     if (med<i["range_min"] or med>i["range_max"]):
155                             incr=incr+1
156                 dict_dati["violazione_"+j]=str(incr)
157             else:
158                 dict_dati["violazione_"+j]=0
```

Figura 2.0.9 Counter violazioni in 1h,3h,12h

Data Storage:

microServ2.py

```
20  try:
21          while True:
22              msg = c.poll(1.0)
23              if msg is None:
24                  # No message available within timeout.
25                  # Initial message consumption may take up to
26                  # session.timeout.ms for the consumer group to
27                  # rebalance and start consuming
28                  print("Waiting for message or event/error in poll()")
29                  continue
30              elif msg.error():
31                  print('error: {}'.format(msg.error()))
32              else:
33                  # Check for Kafka message
34
35                  record_key = msg.key()
36                  record_value = msg.value()
37
38                  record_value=record_value.decode("utf-8")
39                  dict_dati=ast.literal_eval(record_value)
40                  mydb = mysql.connector.connect(host=db_server,user="root",password="root",  database="dsbd")
41                  mycursor = mydb.cursor(dictionary=True)
42                  sql = """INSERT INTO metrics (metric, max_1h, min_1h, avg_1h  , dev_std_1h ,max_3h , min_3h , avg_3h ,dev_std_3h , max_12h , min_12h , avg_12h
43                  val = (dict_dati["metric"], float(dict_dati["max_1h"]), float(dict_dati["min_1h"]), float(dict_dati["avg_1h"]), float(dict_dati["dev_std_1h"]),
44                  mycursor.execute(sql,val)
45                  mydb.commit()
46                  mycursor.close()
47                  print(mycursor.rowcount, "was inserted.")
```

Figura 2.1.1 Data Storage

Dopo aver inizializzato un gruppo di Consumer (1 solo membro), ogni messaggio prelevato viene convertito in un formato adatto al db. Si è deciso di usare 1 solo Consumer in quanto nel messaggio inviato dal Producer sono contenute tutte le informazioni necessarie all'inserimento nella tabella metrics.

Data Retrival:

microServ3.py

Il Data Retrival utilizza un'applicazione Flask che permette di visualizzare tramite l'URL localhost:5000 le metriche definite all'interno dello SLA Manager, vi è inoltre un button che permette di accedere tramite il metodo GET alla funzione my_view_func.

```
151  @app.route('/<name>')
152  def my_view_func(name):
153      mydb = mysql.connector.connect(host=db_server,user="root",password="root",  database="dsbd")
154      mycursor = mydb.cursor(dictionary=True)
155      sql="Select * from metrics where metric_id="+str(name)+" Order by time_stamp DESC;"
156      mycursor.execute(sql)
157      myresult = mycursor.fetchall()
158      mycursor.close()
```

Figura 2.2.1 Query visualizzazione risultati metriche generate da ETL

Tale funzione permetterà la visualizzazione dei risultati delle metriche generati dall'ETL che saranno ordinate in base al timestamp . Vi saranno due bottoni che, tramite un metodo GET permetteranno di accedere ad una funzione per visualizzare graficamente la funzione di autocorrelazione e la stagionalità.

```
114      if(metric=="autocorr"):
115          sql="Select metric_id,autocorr from metrics where Id_metric="+str(name)+";"
116          mycursor.execute(sql)
117          myresult = mycursor.fetchall()
118          mycursor.close()
119          d=""
120          p=0
121          for asa in myresult:
122              p=asa["metric_id"]
123              fig = Figure()
124              res = ast.literal_eval(asa["autocorr"])
125              serie=pd.Series(res)
126              axis = fig.add_subplot(1, 1, 1)
127              axis.locator_params(axis='y',tight=True, nbins=2)
128              axis.grid()
129              c=[]
130              for ada in serie.values:
131                  c.append(float(ada))
132
133              axis.set_title("Autocorrelazione")
134              axis.set_xlabel("Campioni")
135              axis.set_ylabel("Valori")
136              axis.plot(c)
137              pngImage = io.BytesIO()
138              FigureCanvas(fig).print_png(pngImage)
139              pngImageB64String = ""
140              pngImageB64String += base64.b64encode(pngImage.getvalue()).decode('utf8')
141              d=pngImageB64String
```

Figura 2.2.2 Plot autocorrelazione

Tramite il metodo GET arriveranno 2 valori tra cui l'id della metrica selezionata precedentemente e una stringa che permetterà di visualizzare o la stagionalità o la funzione di autocorrelazione.

Dopo che è stata scelta la stagionalità o la funzione di autocorrelazione viene prelevata dal database i valori relativi alla funzione da calcolare. Tramite questi valori vengono calcolati i plot che verranno inseriti all'interno di una stringa che sarà visualizzabile tramite il template.

SLA Manager:

microServ4.py

Lo SLA Manager utilizza un'applicazione Flask che permette di visualizzare tramite l'URL localhost:5001 le metriche definite all'interno della tabella Sla, vi sono inoltre 4 button:

- Visualizza predizione: permette di visualizzare se ci sono state delle violazioni nei dati predetti
- Update: permette di modificare il range di min e max e imposta la metrica all'interno dello SLA Set
- Remove: rimuove la metrica all'interno dello SLA Set
- Possibili violazioni: visualizza il numero di violazioni su 1h, 3h, 12h

I button Visualizza predizione, Remove e Possibili violazioni saranno visualizzabili solamente dalle metriche presenti nello SLA Set. Il button Update sarà visualizzabile dalle metriche presenti all'interno dello SLA Set e anche dalle altre metriche se il numero di metriche all'interno dello SLA Set risulta essere minore o uguale a 5.