Simulazione di una rete di sensori nell'Edge-Cloud Continuum

Simone Bauco
Università di Roma Tor Vergata
Rome, Italy
simone.bauco@students.uniroma2.eu

Simone Staccone
Università di Roma Tor Vergata
Rome, Italy
simone.staccone@students.uniroma2.eu

***Sommario*—Questo documento descrive lo sviluppo del progetto del corso di Sistemi Distribuiti e Cloud Computing. In particolare, il sistema realizzato rappresenta una simulazione di una rete di sensori distribuita in 22 nazioni europee, che raccoglie dati meteorologici, e li invia a dispositivi edge (implementati in container, uno per nazione), che si occupano di formattare i dati e di inviarli al cloud, dove delle funzioni serverless raccolgono i dati e calcolano delle statistiche. L'obiettivo del sistema sviluppato è quello di creare un ambiente in cui garantire availability, sca scalabilità, fault tolerance e consistenza dei dati, rispettando sempre il principio di minimizzazione dei costi e quindi cercando di utilizzare risorse solo quando effettivamente necessarie.**

I. Introduzione

Nell'ambito dell'Edge-Cloud Continuum, molte applicazioni riguardano il mondo dell'**Internet of Things**. In particolare, la combinazione tra la capacità - spesso ridotta - di elaborazione dei dispositivi edge e la potenza di calcolo scalabile e flessibile dei dispositivi cloud, attraverso l'implementazione di funzioni serverless, consentono di ottenere applicazioni più disponibili, tolleranti ai guasti ed economicamente convenienti. Inoltre, la prossimità dei dispositivi edge ai sensori IoT consente di ridurre la latenza di comunicazione.

II. Motivazioni

A. Edge-Cloud Continuum

Nel contesto di applicazioni localizzate e latency-sensitive, soluzioni cloud-only sono impraticabili per motivi di latenza, di conseguenza si cerca di scaricare parte della computazione e dello storage a risorse presenti sui bordi della rete (edge). Questa soluzione è nota come **Edge-Cloud Continuum**.
In particolare, i dispositivi edge (e.g., sensori) comunicano con i nodi edge (e.g., dispositivi di raccolta dati), che sono fisicamente molto vicini, e questi comunicano con il cloud, dopo aver effettuato parte della computazione. I vantaggi più evidenti sono una maggiore velocità ed efficienza nel processamento dei dati, sfruttando le potenzialità offerte dal cloud.

B. Serverless Computing

Per Serverless Computing, si intende un modello di cloud computing che punta ad astrarre dall'utente aspetti di gestione del server e decisioni sull'infrastruttura di basso livello. Di conseguenza, l'utente è in grado di produrre codice applicativo definendo **funzioni serverless**, che vengono eseguite su un ambiente totalmente gestito dal Cloud provider.
In questo progetto, le applicazioni nel cloud sono state realizzate come funzioni serverless.

III. Framework e linguaggi utilizzati

Come linguaggio di programmazione, è stato utilizzato Python, ritenuto adatto e flessibile per le esigenze del progetto.
I nodi edge sono stati realizzati all'interno di container **Docker**, che consentono di realizzare ambienti isolati ed indipendenti.
La comunicazione - sia tra container che tra container e Cloud - si basa totalmente su HTTP, ed è stata realizzata tramite il framework Flask.
Lato cloud, sono stati utilizzati i seguenti servizi offerti da Amazon Web Services:

- **AWS Lambda**, per realizzare funzioni serverless che ricevono e gestiscono i dati.
- **AWS API Gateway**, per realizzare degli API gateway e collegarli alle funzioni di AWS Lambda, allo scopo di ricevere i dati tramite comunicazione basata su HTTP.
- **AWS DynamoDB**, per realizzare un database dove salvare i dati ricevuti, e su cui effettuare delle query.
- **AWS S3**, al fine di salvare statistiche su base giornaliera, a partire dai dati salvati sul database.
- **AWS CloudWatch**, utilizzato come strumento di logging per verificare il corretto funzionamento dell'infrastruttura cloud.

Tutti i dati meteorologici sono stati ottenuti dal portale OpenWeatherMap.

IV. Obiettivi

Le specifiche di progetto per la realizzazione di questo sistema sono le seguenti:

- Gestire le funzioni nei container.
- Mantenere i container "warm", lato edge, per evitare il fenomeno di Cold Start.
- Effettuare l'offload delle funzioni edge verso un servizio Cloud serverless (e.g., AWS Lambda), oppure verso un framework FaaS open-source (e.g., OpenWhisk, OpenFaaS) in esecuzione su un'istanza EC2.

V. Progettazione

A. Descrizione ad alto livello del progetto

Questo progetto vuole simulare una rete di sensori dislocata in tutta Europa, in particolare in 22 paesi europei. Per ogni nazione, sono presenti dei **sensori** (dispositivi edge) in una o più città, ed un singolo **nodo edge**, che ha la responsabilità di ricevere dati dai sensori, processarli, ed inviarli al **cloud**. In particolare, periodicamente i sensori inviano dati al nodo edge della nazione corrispondente; una volta ricevuti i dati, il nodo edge effettua delle conversioni su questi (e.g., converte la temperatura dalla scala Kelvin alla scala Celsius), per poi inviarli al cloud.
I nodi edge non sono sempre disponibili: infatti, se non ricevono dati per un determinato intervallo di tempo, vanno in shutdown; di conseguenza, deve essere prevista una strategia di contrasto al fenomeno di cold-start.
Lato cloud, è prevista una funzione serverless che riceve i dati, salva questi nel log di ricezione e nel database, e ne calcola delle statistiche, che vengono salvate e rese disponibile on-demand all'edge.
L'**offload** da edge a cloud, dunque, riguarda due aspetti:

- **Storage**: per scelta progettuale, i nodi edge non mantengono alcuno stato, in quanto non sarebbe scalabile mantenere una quantità arbitrariamente grande di dati lato edge, in singoli dispositivi; risulta necessario, di conseguenza, adottare soluzioni cloud.
- **Computazione**: il calcolo delle statistiche sui dati, chiaramente, si basa sulla disponibilità dei dati stessi; perciò, queste statistiche sono calcolate direttamente lato cloud, dove i dati sono direttamente disponibili e più rapidamente accessibili. L'unica computazione che viene effettuata sull'edge si limita al pre-processing dei dati di cui sopra.

B. Architettura di sistema

In figura, è mostrata un'architettura ad alto livello dell'applicazione. In particolare, periodicamente (il ciclo di vita

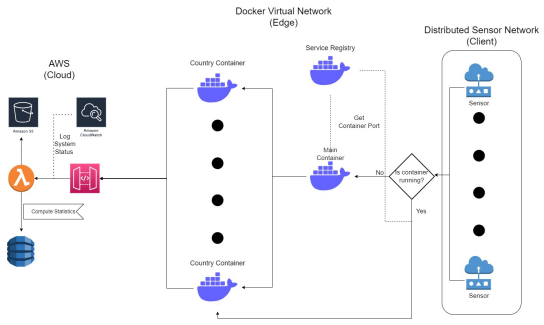

Figura 1. Architettura

dell'applicazione è strutturato in **round**) i sensori presenti in ogni nazione cercano di inviare dati verso il dispositivo edge della nazione corrispondente. Se questo è attivo[1], i dati vengono inviati normalmente, altrimenti viene contattato un dispositivo edge centralizzato (i.e., Main Container), il quale ha le seguenti responsabilità:

- Nel caso in cui un dispositivo edge non sia raggiungibile da uno o più sensori, deve scatenare l'accensione del dispositivo, e poi inviargli i dati in attesa. Questo è il meccanismo adottato contro il fenomeno di **cold start**.
- Nel caso in cui un nodo edge non viene contattato per due round consecutivi, il Main Container deve effettuare lo shutdown di quest'ultimo, mantenendo quindi un algoritmo di check periodico per la gestione di tutti i container.

Per quanto riguarda i dispositivi edge, una volta ricevuti i dati, questi devono effettuare conversioni dei dati ad un formato comune, che è il seguente:

- Temperatura: scala Celsius
- Pressione: hPa
- Umidità: %
- Data: formato YYYY-MM-DD

Effettuate le conversioni, il dispositivo edge deve inoltrare i dati al Cloud: effettua una richiesta di POST HTTP verso un AWS API Gateway.
L'API Gateway è collegato, come trigger, ad una funzione AWS Lambda. Questa funzione deve salvare i dati ottenuti su un database ad hoc, relativo alla nazione specificata dai dati, implementato in AWS DynamoDB.
Una volta al giorno, inoltre, una diversa funzione AWS Lambda calcola, se ci sono dati disponibili, delle statistiche relative al giorno precedente, mediando tutti i dati a disposizione, e salva queste statistiche su un AWS S3 Bucket.

VI. Implementazione

A. Container Docker

I container **Docker** hanno vita locale e servono a rappresentare il core dell'infrastruttura distribuita alle foglie della rete e nell'edge. L'idea è quella di simulare una rete di sensori IoT, quindi all'avvio, vengono lanciati tre container diversi:

- **Data Generation Container**: simula la generazione dei dati da parte dei sensori.
- **Main Container**: dispositivo centralizzato di controllo.
- **Service Registry**: mantiene indicazioni sui numeri di porta esposti da tutti i container.

L'entry point del sistema è rappresentato da uno script bash che crea ed esegue questi tre container, tramite dei Dockerfile differenti che creano diversi layer specifici per ogni container. Inoltre, prima di lanciare i container, viene effettuato il set-up di una **Docker network**, che verrà utilizzata per la comunicazione tra container, demandando di fatto a Docker la responsabilità di effettuare il set-up di un infrastruttura di rete locale per la comunicazione.

1) Data Generator Container: La responsabilità di questo container è quella di simulare i sensori situati geograficamente in zone distanti.

[1]I dispositivi edge vanno in shutdown se non ricevono dati per due round consecutivi.

In particolare, esso presenta un layer Python, che implementa le seguenti operazioni:

- **Richiesta di dati** al portale OpenWeatherMap: in ogni round, viene generato un sottoinsieme casuale, di dimensione fissa[2], dell'insieme delle 55 città presenti, rispetto al quale devono essere generati i dati meteorologici. Per ogni città presente in questo sottoinsieme, viene effettuata una richiesta HTTP GET a OpenWeatherMap per ottenere le coordinate geografiche della città, necessarie per la richiesta successiva, ancora di tipo HTTP GET, per ottenere i dati effettivi, in formato JSON.


```
{
  "coord":{
    "lon":12.4754,
    "lat":41.8933
  },
  "weather":[
    {
      "id":802,
      "main":"Clouds",
      "description":"scattered clouds",
      "icon":"03d"
    }
  ],
  "base":"stations",
  "main":{
    "temp":293.78,
    "feels_like":293.49,
    "temp_min":291.94,
    "temp_max":295.05,
    "pressure":1010,
    "humidity":61
  },
  "visibility":10000,
  "wind":{ },
  "clouds":{ },
  "dt":1698741139,
  "sys":{ },
  "timezone":3600,
  "id":3169070,
  "name":"Rome",
  "cod":200
}
```


Figura 2. Esempio di dati generati da OpenWeatherMap

- **Packing delle informazioni** da inviare al nodo edge, includendo nome della città, nazione di appartenenza, i dati in formato JSON generati in precedenza ed altre informazioni secondarie.
- **Invio dei dati** al container di destinazione, cioè quello relativo alla nazione di appartenenza della città: se questo è attivo, riceve correttamente i dati, altrimenti il Data Generator Container invia i dati al Main Container, che si occuperà di gestire la richiesta.
- **Invio del messaggio di fine round** al Main Container, una volta inviati tutti i dati generati per il sottoinsieme delle città. Questo messaggio deve comprendere la lista di tutti i nodi edge che hanno ricevuto dati direttamente dal Data Generator Container, per fare in modo che il timer di shutdown di questi container sia azzerato, come è spiegato meglio nella prossima sezione.

[2]Sia la frequenza dei round (default: uno ogni 60 secondi) che la dimensione del sottoinsieme di città (default: 10 città) sono configurabili, tramite file config.json.

2) Main Container: Questo container è realizzato utilizzando un layer Linux, in quanto deve essere in grado di lanciare altri container a sua volta (quindi, tra le varie installazioni di start-up, è presente anche l'installazione del Docker engine sul container stesso).
La responsabilità del main container è quella di **controllare** il **ciclo di vita** dei container (nodi edge). In questo contesto, esso implementa le seguenti funzionalità:

- **Shutdown dei nodi edge**: il Main Container mantiene timer di shutdown per ogni nodo edge (container) attivo, per effettuare lo shutdown di quei nodi che sono inattivi per n round di fila, dove n è un parametro configurabile tramite il file di configurazione, e per default vale 2. Quando un nodo edge riceve dati (o dal Data Generator Container, o direttamente dal Main Container), il contatore corrispondente viene azzerato, per cui il relativo container resterà attivo per almeno altri due round.
- **Startup dei nodi edge**: come già accennato, quando il Data Generator Container non riesce a contattare il container del nodo edge, invia i dati al Main Container. In questo caso, il Main Container deve effettuare lo startup del nodo edge corrispondente, per poi inoltrare i dati ricevuti dal Data Generator Container. Questo meccanismo consente di mitigare il fenomeno di **Cold Start**, come meglio spiegato in una sezione successiva.

3) Service Registry: Questo container ha la responsabilità di mantenere le informazioni relative ai **port number** esposti e ai nomi dei container che sono running, al fine di aumentare il disaccoppiamento dei dati nel sistema.
In particolare, il container è costruito utilizzando un layer Python che lancia un server Flask che rimane in ascolto di richieste di HTTP GET, relative al numero di porta per un determinato container istanziato a run time.

4) Edge Container: Questi container, uno per nazione, vengono costruiti utilizzando un layer Python.
Essi hanno le responsabilità di **ricevere** i dati, come già detto o dal Main Container oppure dal Data Generator Container, effettuare un **pre-processing** dei dati, cioè conversioni a formato comune, ed infine **inviare** dati al cloud, in particolare ad un AWS API Gateway. Inoltre, questi dispositivi possono effettuare delle **query** al cloud, ad esempio per ottenere i dati medi relativi ad una certa data; in questo caso, si effettua una richiesta HTTP verso un ulteriore AWS API Gateway.

B. AWS

La componente cloud del sistema è stata implementata utilizzando servizi di Amazon Web Services. Durante lo sviluppo, a scopo di debugging, è stato utilizzato il servizio di logging offerto da AWS CloudWatch.

1) API Gateway: Questa componente è necessaria per permettere al sistema cloud di essere contattato dall'esterno tramite richieste HTTP. Rappresenta quindi l'**entry point** del cloud.
In particolare, sono stati realizzati due API Gateway: uno relativo alla funzione di salvataggio dei dati e uno relativo alla funzione di query.

2) AWS Lambda: Le funzioni serverless sono state implementate tramite AWS Lambda. Sono state realizzate tre funzioni:

- `cloudWeatherDataCollector()`: collegata ad un trigger API Gateway che la rende raggiungibile dall'esterno tramite comunicazione HTTP, implementa le funzionalità di **salvataggio dei dati su database**. In dettaglio, la funzione riceve in input dati JSON formattati nel seguente modo:

```
{
  "name": "Madrid City Center",
  "country": "es",
  "temp": 28.120000000000005,
  "feels_like": 28.07000000000005,
  "temp_min": 26.120000000000005,
  "temp_max": 29.680000000000007,
  "pressure": 1025,
  "humidity": 44,
  "date_time": "2023-10-04 15:46:31"
}
```

  A partire da questi dati, estrae la nazione corrispondente e crea, se non già esistente, una tabella DynamoDB chiamata table-sdcc-{country} (nell'esempio, table-sdcc-es), avente come chiave la coppia (name, date), in quanto non interessa salvare dati duplicati relativi alla stessa città e alla stessa data e ora. Successivamente, vengono salvati tutti i dati in input su questa. Se non ci sono errori, viene ritornato il codice HTTP 200.

- `computeAndSaveDailyStatistics()`: questa funzione ha lo scopo di calcolare delle **statistiche medie giornaliere**, partendo dai dati salvati nelle tabelle DynamoDB. In particolare, la funzione ha come trigger tutte le tabelle DynamoDB relative alle 22 nazioni, per cui qualsiasi evento su questa tabella (inserimento, modifica, cancellazione, ...) triggera l'esecuzione di questa funzione. L'evento di interesse, in questo caso, è l'inserimento di un nuovo record, per cui il primo controllo effettuato dalla funzione riguarda la tipologia di evento: se questo non è 'INSERT', allora la funzione ritorna senza effettuare operazioni.
  L'idea alla base di questa funzione è di calcolare le statistiche medie relative alla giornata precedente a quella corrente **una sola volta**: per realizzare ciò, le statistiche sono calcolate solamente all'inserimento della prima tupla relativa alla giornata corrente, motivo per cui all'inizio della funzione viene effettuata una query, sulla tabella della nazione corrispondente, per verificare se già esistono record nella tabella che hanno come data quella corrente e, in caso affermativo, la funzione non effettua ulteriori operazioni.
  Se, invece, il record inserito è il primo della giornata, la funzione calcola le statistiche medie relative alla giornata precedente (se sono presenti dati) formattandoli come file JSON, che conterrà i dati medi per ogni città presente e la media nazionale, per poi salvare questi dati in un **AWS S3 Bucket**, unico per nazione.


```
{
  "Bremen":{
     "avg_temp":9.629999999999995,
     "avg_feels_like":8.340000000000032,
     "avg_temp_min":8.930000000000007,
     "avg_temp_max":10.550000000000011,
     "avg_pressure":995.0,
     "avg_humidity":96.0
  },
  "Dortmund":{
     "avg_temp":9.670000000000016,
     "avg_feels_like":9.670000000000016,
     "avg_temp_min":8.240000000000009,
     "avg_temp_max":11.379999999999995,
     "avg_pressure":995.0,
     "avg_humidity":81.0
  },
  "Hamburg":{ },
  "Frankfurt am Main":{ },
  "Altstadt Nord":{ },
  "Kreisfreie Stadt Essen":{ },
  "National Average":{
     "avg_temp":10.440000000000012,
     "avg_feels_like":9.817500000000024,
     "avg_temp_min":9.381666666666689,
     "avg_temp_max":11.69083333333335,
     "avg_pressure":995.3333333333334,
     "avg_humidity":89.91666666666667
  }
}
```


Figura 3. Esempio di file di statistiche giornaliere salvato sui bucket

- `queryWeatherData()`: questa funzione ha come trigger un API Gateway, per renderla invocabile dall'esterno. L'input della funzione, che ha lo scopo di **effettuare query** sulle tabelle DynamoDB, deve ricevere in input dati in formato JSON che rispettino le seguenti indicazioni:

```
{
    'city': 'string',
    'country': 'string',
    'date': 'string',
    'full_data': 0/1,
    'only_national_average': 0/1
}
```

  dove 'city', 'country' e 'date' sono i parametri della query, mentre 'full_data' specifica se vengono richiesti tutti i dati, quindi relativi a tutte le città e la media nazionale, mentre 'only_national_average' indica se si vuole ricevere solamente la media nazionale (in entrambi i casi, se l'attributo vale 1 non c'è necessità di specificare il valore di 'city').
  In base alla configurazione di input, la funzione recupera i dati dal bucket S3 corrispondente e li ritorna al chiamante in formato JSON.

3) AWS S3: Come già detto nella descrizione delle funzioni AWS Lambda, viene utilizzato come servizio di persistenza per i dati medi giornalieri. In dettaglio, esiste un bucket per ognuna delle 22 nazioni supportate dall'applicazione.

4) AWS DynamoDB: Componente di persistenza che viene utilizzato per salvare le tuple generate dai container Docker sul cloud, attraverso la funzione AWS Lambda. Il DB è strutturato in più tabelle, una per ognuna delle 22 nazioni. Inoltre, in ogni tabella è stato creato un indice, chiamato DayIndex, che consente di effettuare query sulla tabella per data, in quanto la data non è l'unica chiave della tabella.

VII. Meccanismo di contrasto a Cold Start

Come già accennato in una sezione precedente, il meccanismo di contrasto al fenomeno di Cold Start è stato implementato all'interno del Main Container. In particolare, qualora uno dei container edge non sia raggiungibile, in quanto spento, il Data Generator Container contatta direttamente il Main Container, il quale riattiva il container edge corrispondente per inoltrarvi i dati.
Questo meccanismo, congiuntamente al meccanismo di shutdown dei container edge, consente di ottenere due risultati:

- Se un container edge viene contattato spesso, è molto probabile che sia sempre disponibile, dal momento che questo viene sottoposto a shutdown soltanto dopo un certo numero (configurabile) di round di inattività. Seppure questo container venisse sottoposto a shutdown, in seguito a inattività, sarebbe riattivato dal Main Container, azzerando di conseguenza il timer di inattività.
- Se un container edge viene contattato raramente, è plausibile che ogni volta il Data Generator Container trovi il container di destinazione spento. Tuttavia, in questo caso, non è di interesse che la richiesta sia soddisfatta nell'immediato, dal momento che la tupla successiva, con tutta probabilità, sarà generata dopo un tempo significativo dall'istante corrente.

Di conseguenza, l'unione di questi meccanismi consente di mantenere ***warm*** i container edge, mitigando il fenomeno di Cold Start.

VIII. Risultati

Questa sezione contiene una valutazione del meccanismo di mitigazione di Cold Start descritto nella sezione precedente. In particolare, sia il Data Generator Container che i vari container edge salvano in un file, rispettivamente, i tempi di generazione di ogni dato e i tempi, corrispondenti agli stessi dati, in cui sono stati ottenuti i risultati dal cloud, in relazione alla funzione di salvataggio delle statistiche.
Ogni tempo, inoltre, mantiene un'annotazione che afferma se, durante il processing del dato corrispondente, è stato riscontrato oppure no il fenomeno di Cold Start. Questi dati, salvati in file .csv presenti nella cartella time_data, sono stati analizzati tramite il file compute_time_stats.py, ottenendo i seguenti risultati[3]:

- Tempo medio nel caso di Cold Start: 3.818279631578948
- Tempo medio nel caso in cui non c'è Cold Start: 1.2049860710382525

Si nota quindi come, nel caso in cui i dati non incontrino il fenomeno di Cold Start, le prestazioni siano decisamente superiori. Chiaramente, come descritto prima, i tempi minori saranno sperimentati nel processing dei dati relativi alle nazioni che generano più dati.

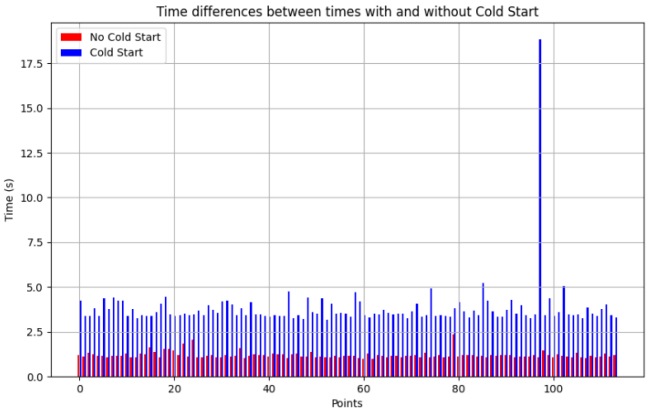

Figura 4. Rappresentazione grafica dei risultati

IX. Conclusioni

A. Miglioramenti Possibili

Alla fine dello sviluppo del nostro progetto, possiamo dire che questo è stato realizzato per gestire scenari medi, andando a simulare il sistema con carichi di traffico non troppo estesi. Proprio per questo ci sarebbe un ulteriore possibile miglioramento prima di un deploy in una rete reale. In particolare le componenti critiche del sistema sono:

- Main Container: Questa componente centralizzata non presenta alcun meccanismo di recovery automatico dopo un possibile fault. Per aumentare la fault tolerance di questo componente, si potrebbero realizzare più repliche di questo, implementando un meccanismo di load balancing condividendo lo stato dei container attivi in quel momento, per garantire anche maggiore disaccoppiamento tra sensori e sistema locale.
- Discorso analogo si può fare per quanto riguarda il Service Registry, essendo una componente centralizzata potrebbe risultare in un SPOF nel discovery della rete di un nuovo sensore.

Inoltre, si potrebbe fare uno studio più esteso su questioni di pricing in base al carico medio atteso del sistema, per gestire meglio l'utilizzo delle risorse nei servizi AWS (On premise o on demand). Nel nostro scenario, infatti, tutti i costi sono

[3]Sono stati considerati 114 tempi con Cold Start e 366 tempi senza Cold Start

on demand, dal momento che il carico di lavoro è piuttosto ridotto; tuttavia, al crescere del carico di lavoro, potrebbe essere più conveniente passare a costi fissi, che garantiscono determinati requisiti di qualità.