Parking Now

Alessandra Fanfano, Lisa Trombetti

Università di Roma Tor Vergata

L'obbiettivo di questo report è di presentare l'implementazione ed i risultati ottenuti nella realizzazione del progetto di Sistemi Distribuiti e Cloud Computing 2019-2020.

L'implementazione si basa sul concetto di Fog Computing, sviluppatosi negli ultimi anni per gestire applicazioni data/computing intensive poiché soluzioni puramente cloud risulterebbero poco pratiche a causa dei problemi di latenza nella comunicazione. Per questo motivo nel paradigma di fog/edge computing si cerca di scaricare la computazione e lo storage su dei micro-datacenter (cloudlet) più vicini ai bordi della rete e di conseguenza agli end-users.

Parking Now sfrutta questo concetto per realizzare un'applicazione che sia in grado di monitorare i parcheggi disponibili in strutture private ed inviare i dati agli utenti il più velocemente possibile. L'applicazione si basa sull'uso di sensori IoT in grado di rilevare quando un posto auto sia occupato o meno, in modo tale da poter realizzare una stima delle ore più trafficate all'interno della struttura.

SOMMARIO

1 SPECIFICA ED ANALISI DEI REQUISITI

In questa sezione prenderemo in analisi le specifiche dei requisiti, funzionali e non, che il prodotto finale è tenuto a soddisfare.

1.1 REQUISITI FUNZIONALI

I requisiti funzionali sono utilizzati per descrivere in dettaglio i servizi da fornire o gli effetti di un sistema. Andremo quindi ad evidenziare le funzionalità principali dell'applicazione:

- deve esporre un'interfaccia utente che mostri in tempo reale la situazione del parcheggio preso in esame;
- deve fornire all'utente l'opportunità di visualizzare un grafico che mostri il numero di automobili presenti nelle 24 ore precedenti alla richiesta, così da poter definire l'orario più favorevole;

1.2 REQUISITI NON FUNZIONALI

I requisiti non funzionali, invece, sono dei veri e propri vincoli sul sistema o sul processo di sviluppo, riguardanti quindi l'implementazione stessa del sistema. Nel nostro caso sono le specifiche di assegnazione per il progetto preso in esame.

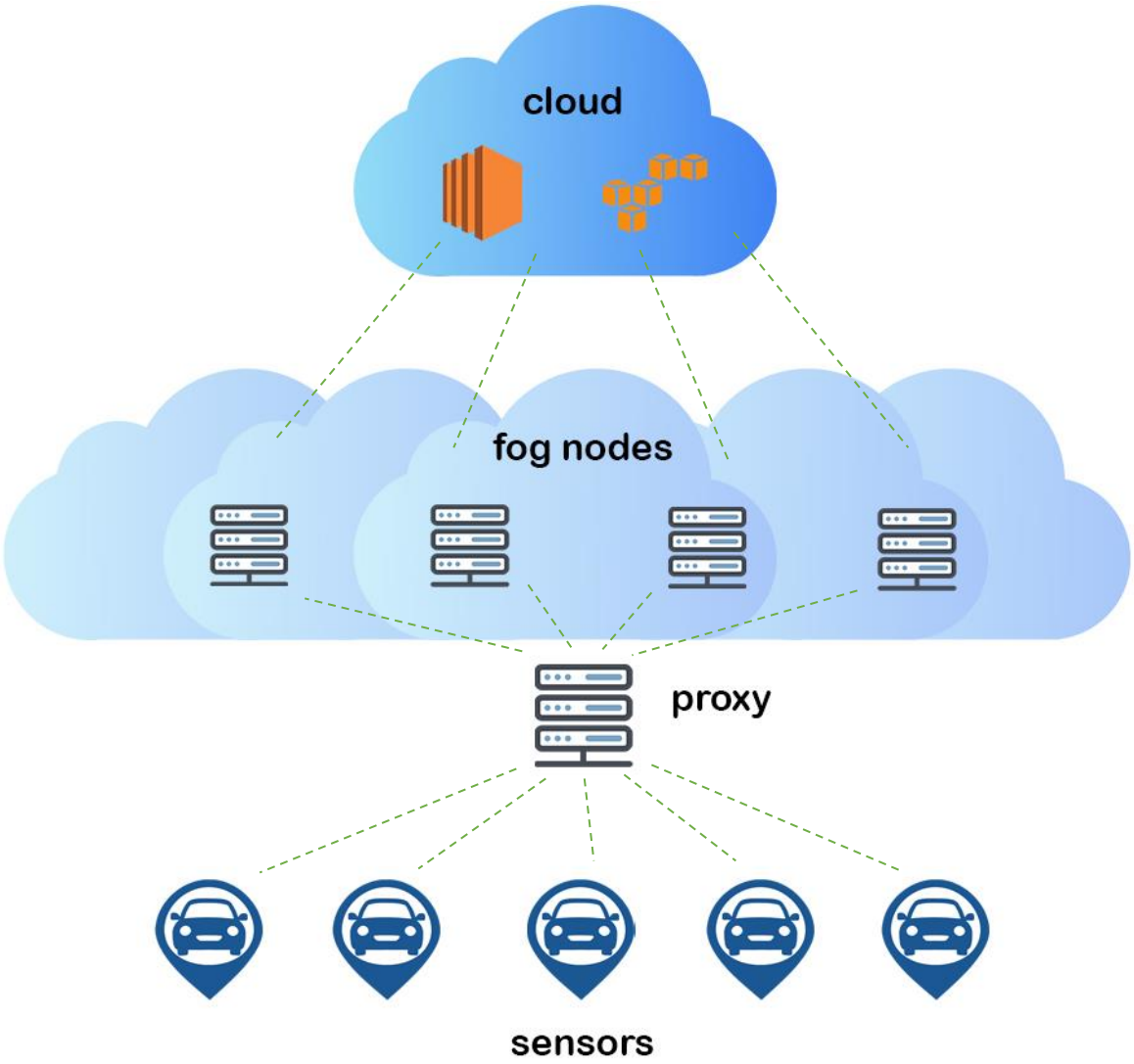
L'applicazione:

- deve essere distribuita su molteplici nodi;
- deve essere scalabile e tollerante ai crash dei nodi Fog o Cloud su cui viene eseguita;
- deve utilizzare almeno un servizio Cloud per eseguire una funzionalità dell'applicazione che sia computazionalmente onerosa;
- dovrebbe utilizzare un protocollo di messaggistica IoT.

Le scelte intraprese in fase di progettazione e sviluppo per soddisfare tali requisiti sono:

- l'utilizzo di sensori IoT, ognuno associato ad ogni singolo parcheggio per permettere così una raccolta dati accurata, fornisce l'idea di un'applicazione distribuita su molteplici nodi, fisicamente distinti;
- l'utilizzo di Containers per permettere la distribuzione del software su uno o più nodi, fornendo, inoltre, isolamento e portabilità oltre a una maggiore resistenza ai guasti: in caso di fallimento di uno o più nodi Fog ce ne sarà un altro pronto a prendere il suo posto;
- l'utilizzo di servizi Cloud di Amazon: un Server che si occuperà periodicamente della raccolta dati e della produzione delle conseguenti statistiche, e un servizio di Storage che favorirà la realizzazione di uno storico dei dati raccolti nei giorni precedenti;
- l'applicazione è strutturata in modo tale che ogni nodo Fog sia in grado di fornire gli stessi servizi, recuperando informazioni sia dagli altri nodi che dal server stesso.
  La comunicazione tra server e il singolo nodo Fog viene, quindi, realizzata utilizzando REST API, dove i parametri necessari all'esecuzione dei servizi vengono inclusi in richieste http. Mentre la comunicazione tra i diversi nodi Fog avviene tramite messaggi broadcast.

2 ARCHITETTURA DELL'APPLICAZIONE



L'architettura dell'applicazione è caratterizzata da quattro parti fondamentali:

- Cloud: si occupa di gestire tutte le funzionalità più onerose in termini di computazione e di implementare uno storage persistente dei dati raccolti giorno per giorno dai sensori
- Nodi Fog: insieme creano uno strato di "nebbia" tra l'infrastruttura cloud ed i sensori, permettendo ai client di ricevere più velocemente le informazioni
- Server proxy: un server che si occupa di ridirigere le richieste da parte sia dei client che dei sensori verso uno specifico nodo Fog, implementando anche un servizio di load balancing
- Sensori IoT: monitorano lo stato del posto auto inviando delle notifiche periodiche ai nodi Fog

La componente cloud dell'applicazione è stata realizzata utilizzando i servizi di Amazon Web Services, in particolare EC2 e S3 che verranno descritti nel particolare nelle sezioni seguenti.

La comunicazione fra le componenti del sistema si avvale dell'utilizzo di REST api messe a disposizione sia dei servizi cloud che dai nodi Fog. Tutte le richieste da parte di un client o di un sensore non vengono immediatamente gestite ma passano attraverso un server proxy, il quale implementa un meccanismo di load balancing per evitare il sovraccarico dei nodi del sistema.

I nodi che fanno parte dello strato di “nebbia” periodicamente si scambiano informazioni sui sensori locali in modo che, anche se con meno frequenza, il client possa vedere i cambiamenti relativi a posti auto più lontani dal nodo da cui riceve le notifiche.

Tuttavia, non avendo a disposizione un’infrastruttura che permettesse la realizzazione distribuita dei nodi Fog, si è deciso di simularli in locale tramite l’utilizzo di container docker.

3 IMPLEMENTAZIONE

3.1 LINGUAGGIO DI PROGRAMMAZIONE

Il linguaggio scelto per lo sviluppo dell’applicazione è *Python*. In particolare, si è fatto uso del micro-framework web *Flask* come piattaforma software di sviluppo all’applicativo. Tale scelta ha favorito l’implementazione di un’applicazione che supportasse la comunicazione stateless e che fornisse un codice uniforme e di facile lettura. Per inoltrare le richieste http, alla base delle REST API, si è fatto uso dei request object messi a disposizione del framework stesso.

L’utilizzo di Python come linguaggio di programmazione non risulta essere una scelta esclusiva: è possibile far uso di qualsiasi linguaggio si voglia, a condizione che questo supporti l’invio e/o ricezione di richieste http.

3.2 DOCKER COMPOSE

Docker Compose è uno strumento per la definizione e l'esecuzione di applicazioni Docker multi-container. Si utilizza un file “docker-compose.yml” per configurare i servizi dell’applicazione in modo tale che questi possano eseguire contemporaneamente e in maniera isolata. Quindi, con un solo comando, si è in grado di creare e avviare tutti i servizi specificati all’interno dalla configurazione.

Nel nostro particolare caso, i container utilizzati per la realizzazione dei nodi Fog presentano un’immagine costituita da *Ubuntu 20.04,* sulla quale vengono installati tutti gli strumenti necessari per lo sviluppo in Python: *python3, python3-div, pip, Flask, Flask-RESTful e requests*. Mentre il container realizzato per l’esecuzione del *Reverse Proxy* viene utilizzato *nginx:alpine* che permette la realizzazione di un reverse proxy server per la gestione dei messaggi tramite il protocollo http e del load balancer.

3.3 AWS EDUCATE

Dei molteplici servizi offerti da AWS Educate, quelli che abbiamo scelto e applicato all’interno del progetto sono:

- *EC2* (Elastic Computing): un servizio Web che fornisce capacità di elaborazione sicura e scalabile nel cloud. È concepito per rendere più semplice il cloud computing su scala Web per gli sviluppatori.
  Tale servizio viene ad essere utilizzato all’interno dell’applicativo per poter girare al suo interno il server che andrà poi ad interagire con i diversi nodi Fog.
- *S3* (Simple Storage Service): un servizio di storage di oggetti che offre scalabilità, disponibilità dei dati, sicurezza e prestazioni all’avanguardia. Può essere utilizzato per archiviare e proteggere una

qualsiasi quantità di dati per una vasta gamma di casi d'uso, come ad esempio per siti web, applicazioni, backup, ripristino, archiviazione, ecc… Tale servizio ci offre, invece, l'opportunità di mantenere un vero e proprio backup dei dati: il server presente sull'istanza di EC2 ha infatti il compito di andare a realizzare dei file, in cui sono riportati i dati raccolti un determinato momento, per poi archiviarli all'interno del bucket presente su S3.

3.4 MODULO CENTRALNODE

Il modulo CentralNode è interamente collocato sull'istanza EC2 di Amazon utilizzata all'interno dell'applicazione. Le funzionalità principali di tale modulo sono:

- La realizzazione e la gestione del server che andrà poi a comunicare con il cluster Fog
- La realizzazione e la gestione del database locale Mysql sul quale vengono memorizzati i dati relativi ai sensori, ovvero
    - il numero del sensore che si sta considerando,
    - il numero di volte in cui un determinato posto, associato a un particolare sensore, era stato occupato,
    - il valore datetime, ad indicare il giorno e l'ora
- La realizzazione dei file che verranno poi ad essere memorizzati su S3
- La gestione del bucket presente su S3

Come detto precedentemente si è sfruttato il micro-framework Flask per sviluppare l'applicativo, il server quindi utilizza le REST API per la comunicazione tramite protocollo http. Le api che sono state implementate sono:

- set_fog_info(): permette di implementare la rest api POST, ovvero il metodo http progettato per inviare carichi di dati a un server da una risorsa specificata. Tale metodo viene, inoltre, utilizzato per aggiornare i dati presi dai diversi nodi fog;
- send_stats(): permette di implementare la rest api GET, ovvero il metodo http progettato per recuperare solo la rappresentazione/informazione delle risorse e non modificarle in alcun modo. Tale metodo ci permette di restituire i dati del database che riguardano le ultime 24 ore.

All'interno del server viene, inoltre, implementato un thread che ha il compito di aggiornare periodicamente i valori. Questo si occuperà della realizzazione e gestione di S3 e del database locale Mysql. Per la loro gestione viene utilizzato anche un file di configurazione, config.json, che permetterà di definire tutti i valori necessari ad accedere alla console di AWS educate e al server MariaDB, installato sull'istanza di EC2:

Figura 1: Esempio file di configurazione utilizzato dal Server

In merito al nome del bucket di S3 da aggiungere in tale file, risulta essere una scelta più saggia andare a creare il bucket direttamente all'interno dei servizi di AWS e poi utilizzarlo nel codice, a causa delle numerose restrizioni che il servizio Amazon fornisce per l'identificazione di un nome valido. Ciò nonostante, si fornisce

l'opportunità di creare il bucket dinamicamente a runtime, ma risulta essere più probabile inciampare in errori dovuti a una scorretta formattazione di tale parametro.

Per quanto riguarda la realizzazione del file che vengono memorizzati su S3 sono state utilizzate tre librerie principali:

- Matplotlib: una libreria per la creazione di grafici per Python. Fornisce api orientate agli oggetti che permettono di inserire i grafici all'interno di applicativi. Nell'applicazione viene utilizzata per realizzare il grafico con i dati raccolti dai nodi fog.
- Fpdf: una libreria per la creazione di file pdf in Python. Utilizzata all'interno dell'applicativo per la realizzazione di un file pdf contenente una tabella, in cui sono riportati i dati rappresentati nel grafico.
- PyPDF2: una libreria Python costruita come un vero e proprio PDF toolkit. In ParkingNow questa viene principalmente utilizzata per effettuare il merge dei file pdf precedentemente creati, così da avere un unico file finale che verrà poi ad essere memorizzato all'interno del bucket S3.

3.5 MODULO FOGNODE

Non disponendo di un cluster di nodi Fog, quello che si è andato a realizzare è un modulo FogNode che verrà ad essere replicato all'interno di diversi container, in modo tale che questi possano eseguire contemporaneamente e in maniera isolata uno dall'altro. Tramite il Dockerfile è possibile realizzare i diversi container in modo che abbiano come immagine Ubuntu 20.04, sulla quale vengono installati tutti gli strumenti necessari per lo sviluppo in Python e, attraverso un file di testo, è possibile specificare tutte le librerie aggiuntive necessarie per l'esecuzione.

Anche per quanto riguarda il modulo FogNode si è sfruttato il micro-framework Flask per sviluppare l'applicativo, vengono, quindi, utilizzate le REST API per la comunicazione tramite protocollo http. Le api implementate sono:

- get_all(): permette di implementare la rest api GET, ritornando il valore di tutti i sensori;
- get_stats(): permette di implementare un'ulteriore rest api GET, ritornando il valore delle statistiche riguardanti le ultime 24 ore prese dal server presente sull'istanza EC2;
- update(): permette di implementare la rest api POST, favorendo l'update dei valori dei sensori.

In questo caso, vengono, inoltre, inizializzati tre thread. Ognuno dei quali si occupa di uno dei seguenti metodi:

- sendingThread(): si occupa di inviare periodicamente le informazioni riguardanti i sensori che si riferiscono a quello specifico fog e di verificare se i propri valori sono effettivamente aggiornati;
- listeningThread(): si occupa di ascoltare eventuali informazioni/aggiornamenti provenienti dagli altri nodi fog presenti all'interno del cluster;
- statsThread(): si occupa di inviare periodicamente le informazioni riguardanti i diversi sensori all'instanza di EC2 e di aggiornare i valori che vengono mantenuti in locale.

La comunicazione tra i diversi nodi fog, quindi per primi due metodi, avviene via broadcast. Mentre la comunicazione con il server è realizzata utilizzando REST API e richieste http. I parametri necessari all'esecuzione dei servizi vengono inclusi in un file di configurazione, config.json all'interno del modulo stesso.

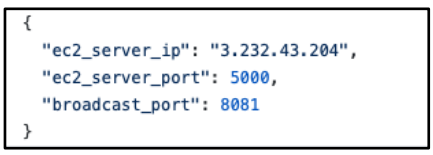


Figura 2: Esempio di file di configurazione utilizzato dal nodo fog

3.6 MODULO REVERSEPROXY

Per l'implementazione dei proxy si è scelto di usare Nginx ('engine-x'), un reverse proxy open source che permette l'utilizzo dei protocolli http, HTTPS, SMTP, POP3 e IMAP. Utilizzando Nginx è anche possibile implementare un meccanismo di load balancing, di cache http e un web server.

In particolare, l'immagine scelta per questo progetto si basa su Alpine Linux, che ha come caratteristica quella di produrre delle immagini molto piccole e snelle adatte per essere usate come base per un proprio Dockerfile.

Tutte le informazioni sulla configurazione sono presenti nel file 'nginx.conf'.

```
worker_processes 1;

events { worker_connections 1024; }

http {

    sendfile on;

    upstream fog {
        hash $arg_hash;
        server fognode:8080;
    }


    server {
        listen 5000;
        server_name findfognode;

        location / {
            proxy_pass         http://fog;
            proxy_redirect     off;
            proxy_set_header   Host $host;
            proxy_set_header   X-Real-IP $remote_addr;
            proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header   X-Forwarded-Host $server_name;
        }
    }

}
```

Figura 3: nginx.conf

La direttiva upstream permette di specificare il gruppo di server a cui verrà rediretta la richiesta da parte del proxy e di definire la politica di balancing che si intende usare. Nginx di default utilizza un round robin per distribuire le richieste, ma è anche possibile specificare:

- least_conn: la richiesta verrà inviata al server con il numero più basso di connessioni attive
- ip hash: la richiesta verrà inviata ad un server basandosi sull'ip del client che l'ha inviata
- hash: la richiesta verrà inviata ad un server basandosi su una chiave definita dall'utente

All'interno di server invece può essere specificata sia la porta su cui il proxy sarà in ascolto che il nome utilizzato per accedere al servizio. Il nome del gruppo definito in upstream viene usato nel nuovo url dalla direttiva proxy_pass che si occupa di reindirizzare la richiesta.

Avendo realizzato sia i nodi fog che i sensori in locale, si è fin da subito abbandonata l'idea di utilizzare la soluzione ip hash per il bilanciamento del carico, altrimenti tutti i sensori avrebbero fatto riferimento sempre allo stesso nodo. Per simulare al meglio una situazione reale si è quindi deciso di fornire una chiave per la funzione di hashing, $arg-hash, passata come parametro in ogni richiesta http sia da parte dei client che dei sensor.

3.7 MODULO SENSORS

Il modulo Sensors si occupa di simulare il comportamento di sensori IoT per il parcheggio. Ogni sensore corrisponde ad un thread che periodicamente sceglie il suo valore, '1' per il parcheggio occupato e '0' per il parcheggio vuoto, e lo invia al nodo fog corrisponde aggiornandone la lista locale.

Tale scelta è stata dettata dal fatto che non si dispone di una rete di sensori da poter utilizzare, per questo si è deciso di simularli localmente.

Per far si che, in assenza di fallimenti, un sensore si riferisca sempre allo stesso nodo fog, in ogni richiesta http viene passato come parametro hash il numero identificativo dello stesso.

3.8 MODULO CLIENTMAC/CLIENTWINDOWS

I due moduli relativi ai client sono essenzialmente identici, le uniche differenze riscontrate sono dovute alle librerie utilizzate che variano a seconda del sistema operativo che si sta utilizzando.

Ciò che viene ad essere implementato sono due gui:

- Nella prima vengono ad essere visualizzati i posti auto ai quali sono associati i sensori. Ad ognuno di questi viene ad essere abbinato un rettangolo che cambierà colore a seconda del suo stato: rosso in caso di posto occupato, verde in caso di posto libero. Per quanto riguarda e informazioni sullo stato dei singoli sensori e quindi del posto a cui questi fanno riferimento queste vengono ad essere reperite tramite una connessione http che i sensori instaurano con i nodi fog, che vanno a comporre il cluster.

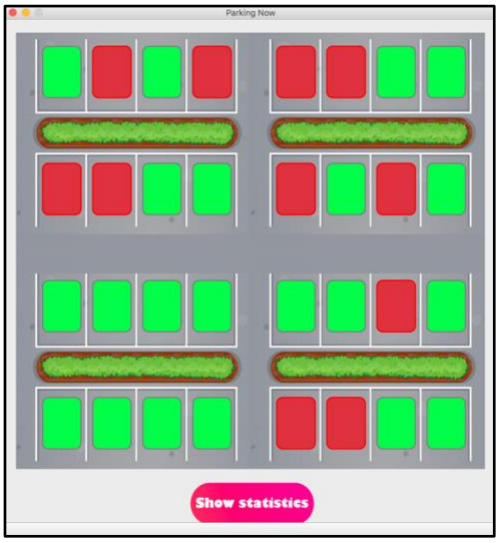


Figura 4: Esempio dell'interfaccia utente di ParkingNow(ClientMac)

- Nella seconda viene ad essere visualizzato un grafico, realizzato sempre utilizzando la libreria matplotlib esposta precedentemente, in cui sono riportati il numero di posti occupati nelle 24 ore precedenti. Per recuperare tali dati viene ad essere sfruttata la comunicazione http con il reverse proxy, che instraderà la richiesta verso il nodo fog più vicino, il quale andrà poi a comunicare direttamente con il server presente su EC2, in modo tale da reperire le informazioni necessarie alla realizzazione del grafico stesso.

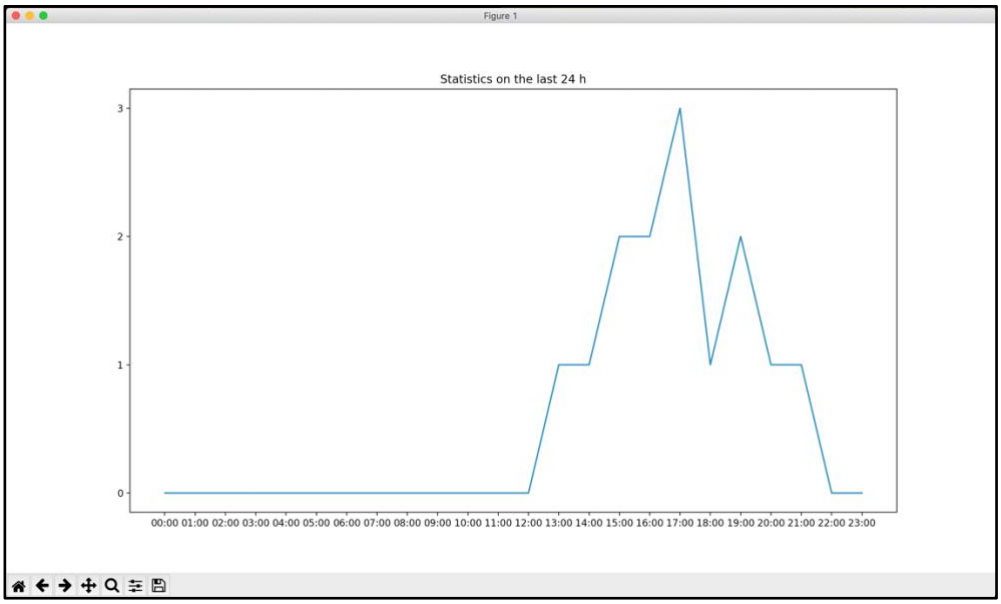


Figura 5:Esempio grafico mostrato al client

Come per gli altri moduli, anche in ognuno di questi sono stati aggiunti dei file di configurazione, config.json, nei quali è possibile individuare le informazioni riguardanti il proxy necessarie per la connessione.

```
{
  "proxy_ip": "findfognode",
  "proxy_port": 5000
}
```

Figura 6: Esempio di file di configurazione del client

3.9 MODULO TEST

Il modulo Test è stato creato appositamente per testare il tempo impiegato dalle richieste del client ad ottenere una risposta, sia essa positiva o negativa.

A questo scopo sono state realizzate una serie di funzioni:

- runTest: si occupa della gestione del caso di test richiamando le altre funzioni
- startFog: inizializza il server proxy e i nodi fog del numero specificato in input
- stopFog: elimina sia il proxy che i nodi fog correntemente attivi
- killFogNode: usato per verificate il funzionamento del sistema nel caso in cui ci siano fallimenti dei nodi fog. Periodicamente cerca i nodi dei container attivi e ne elimina uno scelto casualmente
- createTestFile: crea un file csv con i dati raccolti dal test
- createClient: simula il comportamento di un client

Ad ogni esecuzione di un caso di test viene quindi generato l'ambiente richiamando la funzione startFog, vengono effettuate un certo numero di richieste al sistema salvando i risultati nel file di output e viene richiamato stopFog per rimuovere tutti i container precedentemente creati.

I test possono essere svolti sia all'aumentare del numero di nodi fog all'interno del sistema sia all'aumentare del numero di client concorrenti, utilizzando rispettivamente le funzioni incFogNodesTest e incClientsTest, prendendo in esame i casi in cui sono presenti fallimenti all'interno del sistema e i casi in cui questi non si verificano.

Nel file di configurazione 'config.json' è possibile specificare qual è il numero di ripetizioni che si vuole utilizzare per ogni test.

4 TESTING

Per il testing sono state prese in considerazione le due richieste del client:

- Per richiedere l'aggiornamento sui valori dei sensori
- Per richiedere le statistiche sulle ultime 24 h

Per ognuna di queste i test sono stati effettuati all'aumentare del numero di nodi fog e dei client presenti all'interno del sistema:

- In presenza di fallimenti
- In assenza di fallimenti

Per effettuare tali test si è deciso di strutturate un fallimento da parte dei nodi fog che fosse casuale: in maniera randomica vengono scelti quelli non raggiungibili dal proxy. Tale scelta ci ha portato comunque a dei risulti più o meno vari in caso di fallimenti e non, ma nonostante ciò è giusto precisare che, vista la natura casuale di questi test, potremmo avere con uguali possibilità casi in cui le prestazioni in caso di fallimenti e quelle in loro assenza si equivalgano e casi in cui differiscano andando a favorire le situazioni in cui i nodi sono raggiungibili dal proxy.

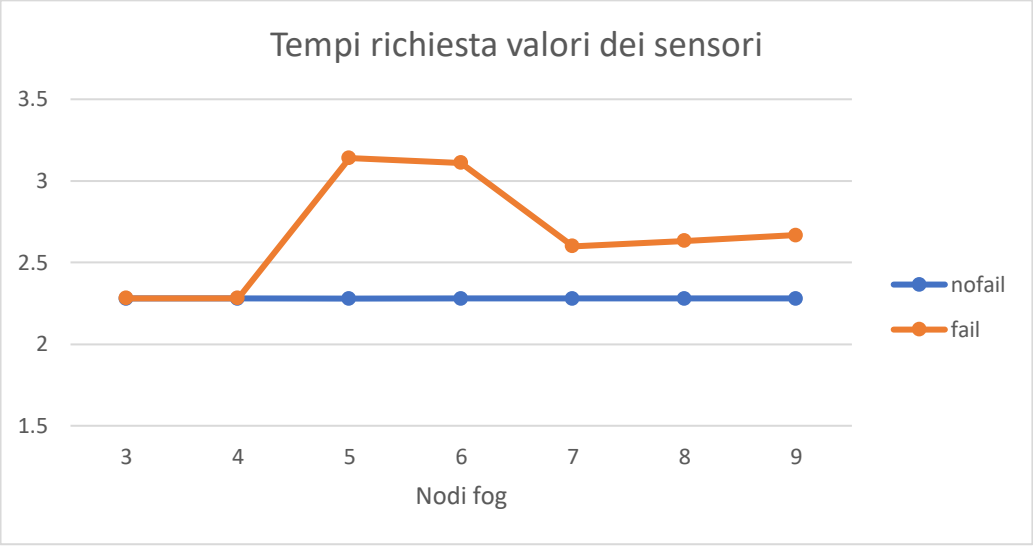


Figura 7: Grafico tempi di richiesta valori dei sensori

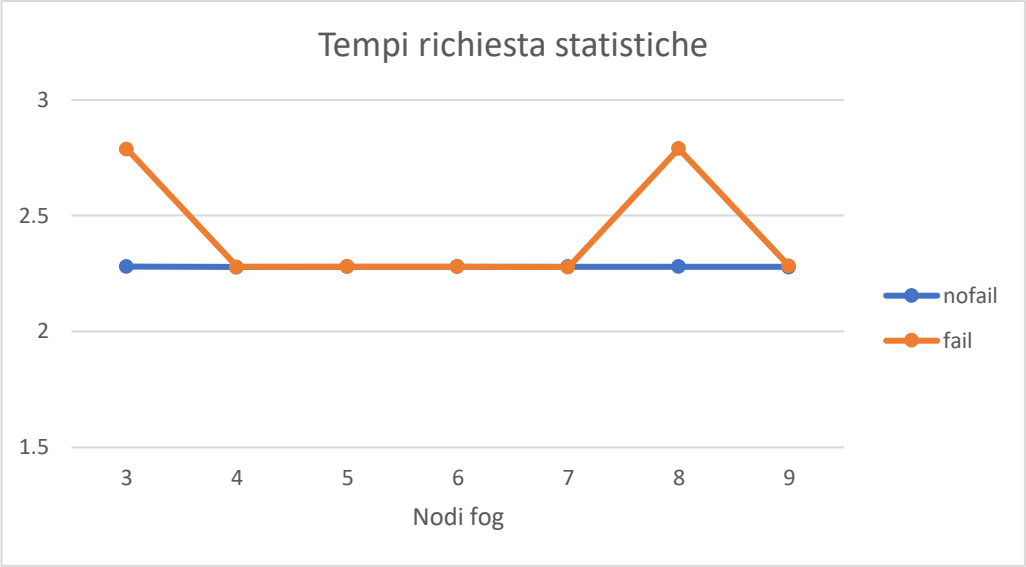


Figura 8: Grafico tempi di richiesta statistiche

In figura 7 e figura 8, sono riportati due grafici in cui vengono riportati alcuni dei risultati ottenuti svolgendo l'attività di testing all'aumentare dei soli nodi fog presenti all'interno del sistema. In questa esecuzione dei test i risultati ottenuti mostrano che in assenza di fallimenti i tempi necessari a tali operazioni risultano essere equivalenti tra di loro nonostante l'aumento dei nodi fog presenti all'interno dell'applicativo. Mentre per quanto riguarda i casi in cui si hanno fallimenti da parte di nodi appartenenti al cluster dei nodi fog, si ha un aumento dei tempi necessari per lo svolgimento delle operazioni.

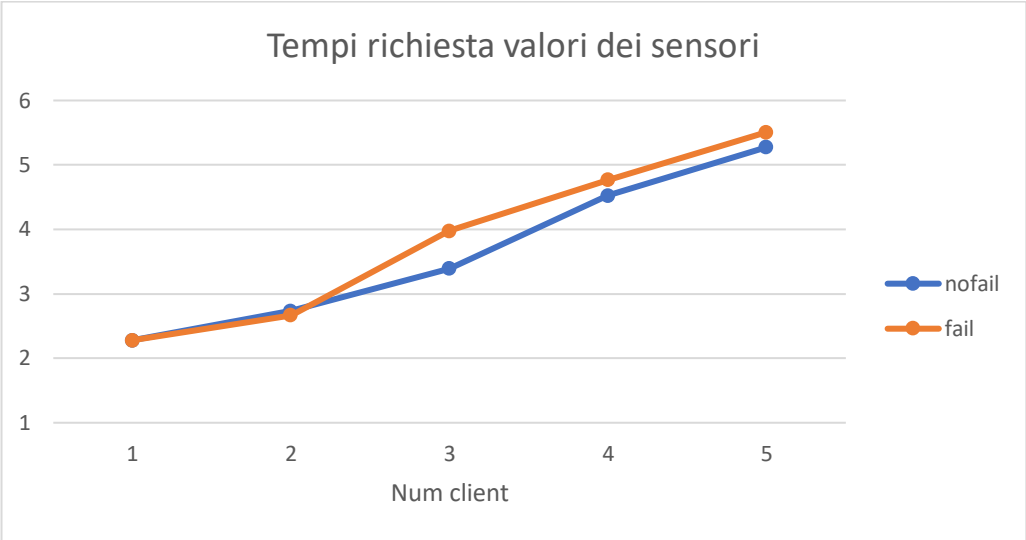


Figura 9:Grafico tempi di richiesta dei valori dei sensori

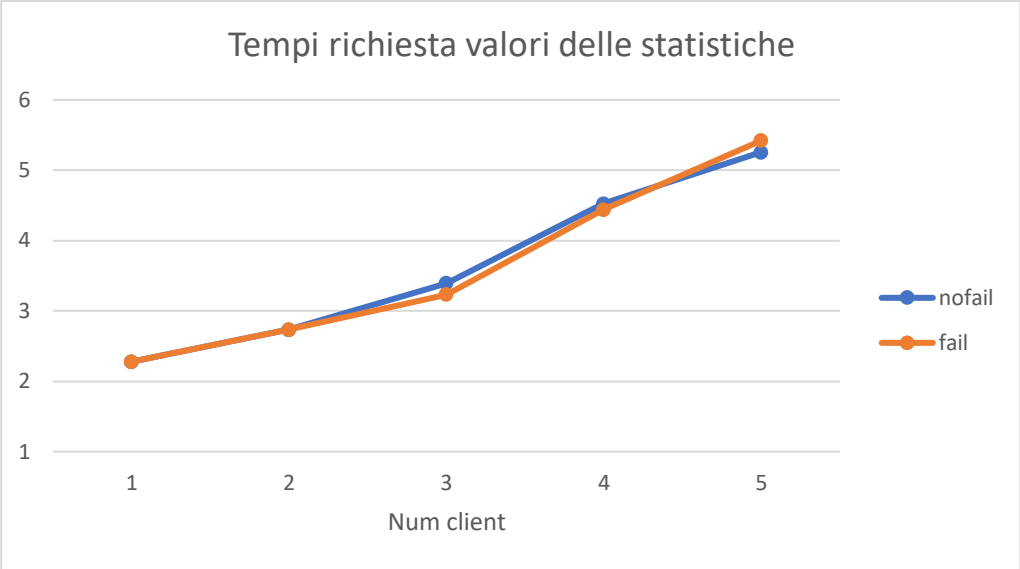


Figura 10:Grafico tempi di richiesta valori delle statistiche

In figura 9 e in figura 10, sono riportati alcuni dei risultati ottenuti svolgendo attività di testing all'aumentare del numero di client presenti all'interno del sistema. In questa esecuzione dei test, possiamo notare che la crescita del tempo di risposta è lineare e non si evidenziano cambiamenti rilevanti nei tempi di esecuzione delle operazioni prese in esame.

Da test generali, applicati all'aumentare sia del numero di fog che del numero di client, possiamo dedurre che il sistema risulta essere abbastanza stabile e resiliente ai guasti.

5 LIMITAZIONI

Come accennato precedentemente, per lo sviluppo di tale applicazione sono stati utilizzati alcuni servizi forniti da Amazon Web Services, che si è rivelato essere uno strumento molto utile, di facile comprensione e ben documentato, fornendo così un supporto alla realizzazione di un'applicazione cloud scalabile e sicura. Nonostante ciò tale servizio presenta delle limitazioni dovute, probabilmente al fatto che quello che si è utilizzato non è un account Amazon che fornisce un completo accesso a tutte le funzionalità e al potenziale di tali web services, come ad esempio la possibilità di non poter scegliere le regioni sulle quali andare a lavorare o la mancanza di un metodo che riuscisse ad utilizzare il servizio S3 senza andare di volta in volta a configurare i diversi parametri di accesso all'interno del file di configurazione.

Un'ulteriore limitazione riscontrata è stata la realizzazione dei sensori IoT tramite una libreria dummy che genera numeri randomici. Nonostante ciò, tale limitazione risulta essere solamente momentanea in quanto sensori simili a quelli che dovremmo utilizzare in tale progetto già esistono, quindi sarebbe abbastanza semplice riuscire a realizzare uno script in Python che permetta di interfacciarsi con essi.

6 CONCLUSIONI

In conclusione, il progetto Parking Now potrebbe, con ulteriori sviluppi, rivelarsi un'applicazione molto utile nella vita di tutti i giorni, in quanto apre lo scenario a più opportunità. La vera potenzialità di tale applicazione è, infatti, la possibilità di essere facilmente estesa e/o migliorata: l'architettura del cluster fog e la comunicazione REST permette uno sviluppo continuo, semplice ed intuitivo permettendo anche l'utilizzo dei più disparati linguaggi di programmazione. Anche l'utilizzo di Docker-Compose si può considerare un

vantaggio di tale applicazione in quanto permette la configurazione di un ambiente di configurazione che può variare a piacimento del programmatore, non andando a influenzare i diversi container.

Per questi motivi si è pensato a miglioramenti futuri che possono coinvolgere l'applicazione:

- L'aggiunta di nuove funzionalità, quali ad esempio la possibilità di prenotare per un periodo di tempo limitato il proprio posto all'interno del parcheggio;
- Una nuova forma di comunicazione tra i diversi nodi fog, così da limitare il numero dei messaggi presenti all'interno della sottorete e fornire una maggiore sicurezza;

Parking Now potrebbe rivelarsi uno strumento molto utile nel realizzare una vera e propria forma di "controllo" del flusso delle persone presenti all'interno della struttura a cui appartiene il parcheggio. In un periodo particolare come quello che stiamo vivendo, durante la pandemia da COVID-19, Parking Now potrebbe offrire un piccolo supporto per limitare le persone presenti all'interno di un determinato edificio. Pensando a un parcheggio di un semplice centro commerciale, grazie a tale applicazione, risulterebbe molto semplice andare ad individuare le ore più trafficate così da poter intervenire in modo repentino e appropriato per evitare assembramenti. Analogamente, in condizioni non di emergenza, potrebbe essere sfruttato per individuare le ore di punta in cui organizzare eventi nei quali si vuole cercare di attirare il maggior numero di persone possibili.