# Utilizzo di un'architettura di Fog Computing in ambito sanitario

Paolo Melissari
Università di Roma "Tor Vergata"

Gabriele Tummolo
Università di Roma "Tor Vergata"

***Il progetto consiste nello sviluppo di un'applicazione in ambito Fog Computing con dispositivi IoT. Questo paradigma si basa sul processamento dei dati ad un livello intermedio tra dispositivi IoT e il data center Cloud, spostando dunque parte della computazione nelle vicinanze dei dispositivi IoT con l'obiettivo di ridurre la latenza e il traffico di rete verso il data center Cloud. Abbiamo applicato questa architettura in ambito sanitario, raccogliendo dati provenienti da sensori quali battito cardiaco e pressione sanguigna per individuare eventuali dati anomali e avvertire un familiare oppure l'ospedale più vicino.***

## I. Introduzione

L'idea del progetto è stata quella di implementare un applicazione in ambito sanitario che permette di rilevare dati da sensori che monitorano battito cardiaco, pressione sanguigna, ossigenazione sanguigna, posizione e cadute di una persona. I dati vengono inviati ad un nodo di processamento intermedio (nodo Fog) il quale a sua volta li analizza e determina se è necessario intervenire inviando un ambulanza o avvertire un parente tramite email. Nel caso in cui il nodo Fog stabilisce che il paziente ha bisogno di un ambulanza invia la posizione di quest'ultimo al data center Cloud, che determina l'ospedale più vicino da cui far partire l'ambulanza. La panoramica delle ambulanze disponibili e occupate sul territorio verrà poi mostrata in una pagina web.

L'applicazione è stata pensata per ricoprire il servizio di soccorso di 11 ospedali della città di Roma, con l'ausilio di due nodi Fog intermedi.

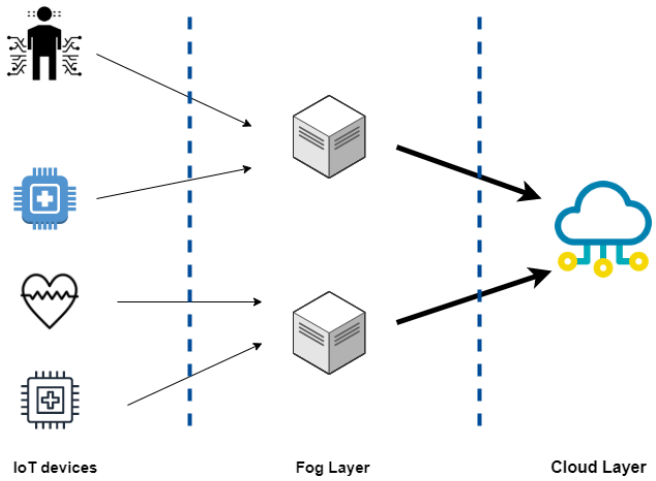


Figure 1. General Architecture

### *1.Simulazione dispositivi IOT*

Per simulare i dispositivi, è stato necessario realizzare dei software scritti in linguaggio Python che generino i dati da inviare al Fog Node per ogni sensore considerato. I dati per ogni sensore hanno i seguenti intervalli:

- Sensore per il battito cardiaco: valori da 50 e 115 che rappresentano i battiti per minuto.
- Sensore per la pressione sanguigna: valori tra 50 e 170 che rappresentano i millimetri di mercurio (mmHg)
- Sensore per l'ossigenazione sanguigna: valori da 40 a 140 che rappresentano i millimetri di mercurio (mmHg)
- Sensore di movimento: valore *True* se il sensore ha rilevato una caduta, valore *False* altrimenti

Inoltre, viene anche simulato lo spostamento di un paziente cambiandone le coordinate di Latitudine e Longitudine. Questo perché ogni nodo Fog è responsabile di una certa area geografica e, se un paziente si muove da una zona all'altra, deve essere connesso al nodo Fog più vicino in modo da avere la minima latenza possibile nelle comunicazioni.

Successivamente sono stati creati 3 dataset contenenti rispettivamente 10, 100 e 1000 pazienti. Ogni paziente ha i seguenti attributi, ognuno scelto in modo randomico: Nome, Cognome, Codice Fiscale, Posizione di partenza, Zona di partenza, Sensori posseduti.

Dunque, per ogni persona nel dataset, sarà possibile generare una sequenza di valori che simuli i dati dei sensori posseduti.

Tra le possibili scelte del protocollo di comunicazione tra i sensori ed i nodi Fog, abbiamo scelto **MQTT** (Message Queuing Telemetry Transport). Questo perché, essendo un protocollo molto leggero, permette l'invio e la ricezione dei dati con un utilizzo di banda molto inferiore rispetto all'utilizzo di richieste HTTP consentendo , in situazioni reali, di utilizzare molta meno batteria al sensore. MQTT utilizza un'architettura di tipo publish/subscribe tramite l'utilizzo di un broker. I sensori possono inviare i dati al broker, che si occuperà di inoltrarli a tutti i componenti (in questo caso il nodo Fog) che sono interessati a riceverli, come mostrato in Figura 2:

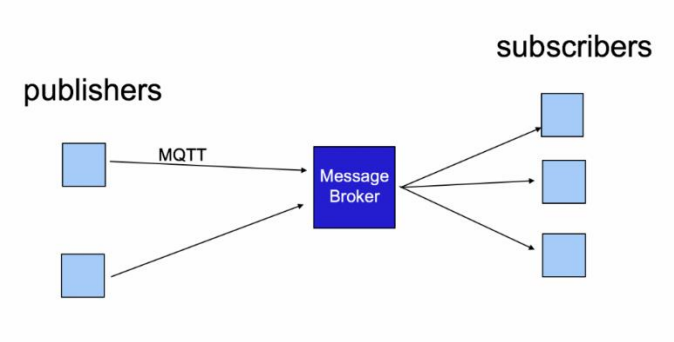


*Figure 2. MQTT Protocol*

Ogni sensore invierà i propri dati sul topic "*pazienti/_nome_del_sensore"* del broker presente nel nodo Fog più vicino, che potrà ricevere i dati per poi processarli.

Una delle caratteristiche principali di MQTT è la scelta del Quality Of Service (Qos) sull'invio e ricezione dei dati.

Sono disponibili 3 livelli di Qos:

- At most once (Qos = 0): in questo scenario non c'è alcuna garanzia di consegna. Il mittente invia al broker e non attende alcuna risposta né provvede a un eventuale re-invio.
- At least once (Qos = 1): in questo scenario, il mittente (che può essere sia il sensore che invia il dato, sia il broker che invia un messaggio di corretta ricezione) mantiene in memoria il messaggio finché non riceve dal destinatario l'ACK di avvenuta ricezione. In caso non lo riceva entro un tempo congruo, lo invia nuovamente, una seconda volta, e attende nuovamente risposta, continuando allo stesso modo finche la risposta non viene ricevuta. A questo livello di Qos potrebbero quindi verificarsi ricezioni multiple dello stesso pacchetto, che avvengono quando il broker riceve correttamente il messaggio e lo rigira ai subscriber, ma l'ACK di corretta ricezione non viene ricevuto dal sensore che quindi invia nuovamente il dato.
- Exactly once (Qos = 2): in questo scenario, viene effettuato un ulteriore scambio di messaggi successivo a quelli necessari per il Qos At least once. Questo scambio di messaggi serve a certificare che il destinatario ha correttamente ricevuto e processato i dati, evitando quindi che sia necessario un secondo invio.

Nel nostro caso, abbiamo scelto di utilizzare il Qos 0 per i sensori di battito cardiaco, pressione sanguigna ed ossigenazione sanguigna, in quanto questi dati vengono inviati molto spesso e non è necessario preservarne l'ordine. Dunque abbiamo preferito scegliere il Qos più basso per avere minor traffico possibile ( ed un minor spreco di batteria in casi reali).

Abbiamo usato il Qos 1 invece per il sensore di caduta, questo perché, se ne osserva una, il nodo Fog deve inviare immediatamente un'email ad un familiare (dunque se un sensore la rileva, dobbiamo essere sicuri che la invii al fog). Un eventuale doppione viene gestito dal nodo Fog, che terrà traccia delle email inviate ed eviterà di inviarne una uguale per lo stesso paziente.

Infine, i dispositivi IoT possono cambiare nodo Fog a cui inviare i dati se, analizzando i valori prodotti dal sensore di posizione, si rendono conto di essere più vicini ad un altro nodo Fog rispetto a quello a cui stanno mandando attualmente i dati.

Inoltre, possono decidere di cambiare nodo Fog a cui mandare i dati se il nodo a cui li stanno mandando attualmente non è più disponibile. In questo modo, è possibile quindi gestire i crash dei nodi Fog: se un nodo Fog si guasta e non è più disponibile, il dispositivo invia i dati ad un secondo nodo più vicino e, quando tornerà disponibile, tornerà ad inviarli al nodo originario.

Il broker utilizzato è stato il progetto open source *Mosquitto* [1], mentre la libreria *paho-mqtt* [2] è stata utilizzata per inviare i dati utilizzando il protocollo MQTT.

### *2.Fog-layer*

Il Fog Layer è il livello intermedio tra quello dei sensori e quello del Cloud. Il suo ruolo è quello di fare un processamento iniziale dei dati, in modo da ridurre la grande quantità di traffico di dati e la latenza che altrimenti sarebbe presente se la comunicazione tra dispositivi IoT e Cloud fosse diretta.

Nella nostra applicazione sono presenti due nodi Fog, che simulano la gestione di due zone diverse di Roma.

Ogni funzionalità di un nodo Fog è stata sviluppata come un'applicazione Flask, un framework web che utilizza il linguaggio Python. Abbiamo poi scelto la tecnologia Docker per avere una migliore distribuzione, portabilità ed esecuzione delle applicazioni. In questo modo, ogni applicazione Flask viene eseguita all'interno del suo container, in modo che ogni servizio possa essere aggiornato indipendentemente dagli altri.

I servizi di un nodo Fog sono i seguenti:

- Servizio *Broker*: container contenente il framework Mosquitto che, come visto in precedenza, svolge la funzione di broker per il protocollo MQTT
- Servizio *MqttSubscriber*: container che si occupa della sottoscrizione al Broker, in modo da poter inoltrare tutti i dati ricevuti agli altri servizi del nodo
- Servizio *HealthChecker*: container che si occupa dello stabilire, in base ai dati ricevuti, se un determinato paziente ha bisogno di assistenza. In particolare:
    - Se negli ultimi 2 minuti ha ricevuto almeno 4 pacchetti contenenti dei dati fuori dalla norma, ma non così anomali dal dover chiedere l'intervento di un'ambulanza, invia un'email ad un parente. Inoltre, invia un'email se il sensore di cadute ne rileva una
    - Se negli ultimi 2 minuti ha ricevuto almeno 4 pacchetti contenenti dei dati che sono di molto fuori dalla norma (che potrebbero ad esempio indicare la presenza di un infarto), richiede al servizio *CloudConnector* di

inviare i dati e la posizione del paziente per poter inviare un'ambulanza

- Servizio mailSender: container che si occupa dell'invio di email ad un parente del paziente per notificarlo dell'anomalia di uno dei sensori (ed eventualmente se è stata chiamata l'ambulanza). Questo servizio utilizza la libreria yagmail [3], che consente l'invio automatico di email da parte di un indirizzo di posta elettronica. Nel nostro caso, abbiamo creato un account Gmail *service.ehealth@gmail.com*
- Servizio mysql : container nel quale viene installato un database Mysql per tenere traccia degli ultimi dati ricevuti e delle email inviate (per evitare di inviare email doppie o di fare due richieste al Cloud per lo stesso paziente)
- Servizio DbConnector: container che si occupa della gestione delle operazioni CRUD richieste dagli altri servizi sul database
- Servizio CloudConnector: container che si occupa dell'invio dei dati al data center Cloud

Per eseguire un nodo Fog è necessario prima di tutto effettuare il build dell'immagine di ogni container, per poi avviarli tutti insieme con docker-compose (che di default genererà una rete che collega tutti i container presenti nel file docker-compose.yml).

La Figura 3 sintetizza la struttura di un nodo Fog:

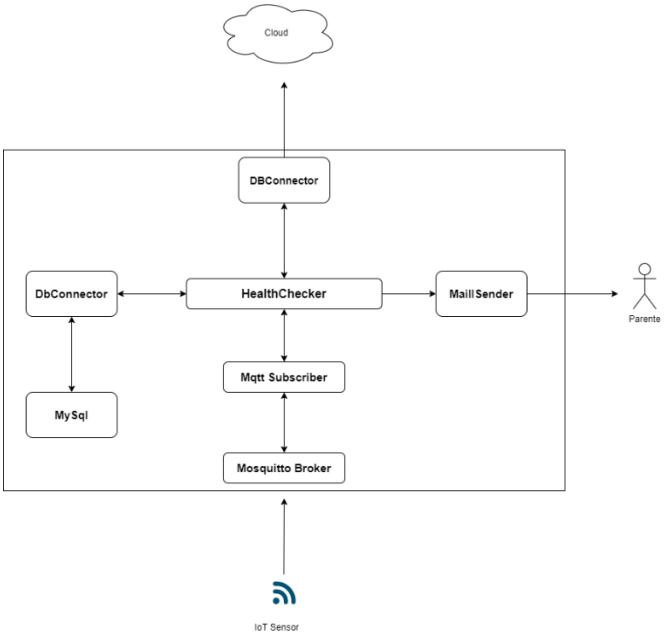


*Figure 3. Fog Node*

*3.Cloud*

Il Cloud è stato realizzato come un applicazione Flask, permettendo di realizzare una semplice pagina web per avere una panoramica generale della situazione degli ospedali e dei pazienti in soccorso, facilitando anche la comunicazione con il Fog-layer che tramite richieste GET/POST può inviare dati o ricevere informazioni. Le richieste di soccorso vengono processate individuando l'ospedale più vicino (data la posizione ) utilizzando la libreria Python "geopy" [4] e calcolando approssimativamente il tempo di trasporto dal punto di richiesta di soccorso all'ospedale; tutte le richieste di soccorso vengono salvate in una tabella prima di calcolare ed individuare l'ospedale più vicino. E' stato utilizzato un database DynamoDB utilizzando il servizio AWS. Prima di procedere con il calcolo della distanza si controlla che la richiesta di soccorso non è già stata presa in carico, evitando di assegnare due più ambulanze ad uno stesso paziente, dopo questi controlli preliminari si procede a calcolare la distanza minima degli ospedali, successivamente vengono inseriti in una tabella il cui contenuto è visibile all'indirizzo: *http://cloud-service.us-east1.elasticbeanstalk.com/.*

In condizioni di saturazione degli ospedali e quindi con una disponibilità non immediata di ambulanze per il soccorso, la richiesta di soccorso viene ritardata di 30 secondi per attendere un eventuale rilascio e quindi disponibilità di un ambulanza, tutto questo viene effettuato dopo aver controllato l'effettiva non disponibilità delle ambulanze al momento dell'arrivo della richiesta. Per implementare l'architettura del Cloud-layer è stato utilizzato il servizio AWS Elastic Beanstalk.

AWS Elastic Beanstalk [5] è un servizio per distribuire applicazioni e servizi web sviluppati in molteplici linguaggi di programmazione, tra cui python che è stato utilizzato come linguaggio per la scrittura del codice Cloud, gestendo l'implementazione di un servizio di auto scaling e definendo un servizio di bilanciamento del carico per la gestione delle richieste. Inoltre crea automaticamente l'istanza EC2 per il lancio dell'applicazione o del servizio web che si vuole offrire ( le cui impostazioni posso essere a loro volta modificate ). È stato deciso di utilizzare questo servizio per la sua semplicità nella definizione delle varie impostazioni e anche per la sua facilità per l'aggiornamento dell'applicazione. La pagina web utilizzata permette di avere una panoramica della situazione corrente degli ospedali, aggiornandosi automaticamente ogni 30 secondi.

Nelle varie impostazioni di Elastic Beanstalk abbiamo aggiunto un trigger basato sulla somma delle richieste, in questo modo viene effettuato auto scaling quando il numero di richieste arriva a 100 richieste in 3 minuti, aumentando fino ad un massimo di 3 istanze EC2; la diminuzione delle istanze avviene nel momento in cui il numero di richieste in 3 minuti e minore o uguale a 50.

La pagina web principale fornisce tutte le informazioni necessarie per poter individuare il paziente e a quale ospedale sta per essere trasportato ( un esempio è mostrato in figura 1), tra le informazioni in tabella le più rilevanti sono il tempo di trasporto e il tempo di arrivo, questo perché con

l'aggiornamento automatico della pagina dalla tabella vengono eliminati i pazienti che sono arrivati in ospedale.

| Nome Ospedale | Id Ospedale | Numero ambulanze |
|---|---|---|
| OSPEDALE SAN CARLO DI NANCY | 7 | 0 |
| OSPEDALE SANDRO PERTINI | 8 | 0 |
| POLICLINICO CASILINO | 10 | 0 |
| OSPEDALE CRISTO RE | 3 | 0 |
| AZIENDA POLICLINICO UMBERTO I | 2 | 0 |
| OSPEDALE SANT'ANDREA | 9 | 0 |
| OSPEDALE S. EUGENIO | 4 | 0 |
| OSPEDALE S. SPIRITO | 6 | 0 |
| OSPEDALE BAMBINO GESU' | 1 | 0 |
| POLICLINICO GEMELLI | 0 | 0 |
| OSPEDALE S. GIOVANNI BATTISTA | 5 | 0 |
| POLICLINICO TOR VERGATA | 11 | 0 |

| Nome | Cognome | Codice Fiscale | Ospedale | Tempo in Minuti | Richiesta inviata alle ore | Ora Arrivo Ospedale |
|---|---|---|---|---|---|---|
| Prova | ooo | hsufqmc | 0 | 5 | 17:56:40 | 18:01:40 |
| Prova | ooo | zuxuozs | 11 | 39 | 17:57:25 | 18:36:25 |
| Prova | ooo | dddigqj | 11 | 39 | 17:57:26 | 18:36:26 |

*Figure 4*

## II. DEPLOY

Per il deploy è stato utilizzato Terraform [6], uno strumento che consente di costruire, modificare e aggiornare l'infrastruttura in modo semplice ed efficiente consentendo il controllo di diverse risorse ed infrastrutture tra cui AWS. E' stato usato per creare automaticamente l'infrastruttura del Cloud, creando uno script bash che avvia le tre fasi di terraform:

- terraform init, comando usato per inizializzare la working directory che contiene i file di configurazione Terraform
- terraform plan, comando usato per creare un piano di esecuzione
- terrafrom apply, comando che applica i cambiamenti descritti nello stato di configurazione che è stato precedentemente pianificato

Successivamente viene caricata l'applicazione su Elastic Beanstalk attraverso comandi AWS.

Questa scelta ha fatto si che la creazione dell'architettura del Cloud viene creata automaticamente e resa operativa lanciando un singolo comando dalla shell.

E' possibile eliminare tutte le risorse create con i comandi sopra indicati attraverso il comando terraform destroy.

## III. TEST

Sono stati implementati diversi tipi di test per valutare la correttezza dell'applicazione nonché la sua stabilità, scalabilità e tolleranza. Si è proceduto prima con la valutazione della correttezza delle funzionalità di Fog e Cloud. Per il Cloud sono stati sviluppati alcuni casi di test per valutare la correttezza del calcolo della distanza tra paziente e ospedale ed assicurarsi che ad ogni paziente venga assegnato l'ospedale più vicino. Nel caso in cui una singola istanza del Cloud-layer termina questa viene ripristinata automaticamente nel giro di pochi minuti, questo avviene in quanto è stato impostato come numero minimo di istanze EC2 = 1.

I vari test sono risultati positivi in quanto la distanza e il tempo effettivo stimato per il trasporto del paziente rispecchiano approssimativamente la realtà e viene sempre individuato l'ospedale con ambulanze disponibili più vicino alla posizione del paziente .

Infine si sono testate le prestazioni del Cloud valutando il tempo di elaborazione delle richieste al variare del numero di richieste al secondo. Si è potuto notare che le prestazioni del Cloud rimangono quasi invariate , in quanto al variare delle richieste ricevuto il tempo di elaborazione delle richieste rimane nell'intorno di 0.06 secondi. Le figure 5 e 6 mostrano i dati nel particolare.


*Figure 5*

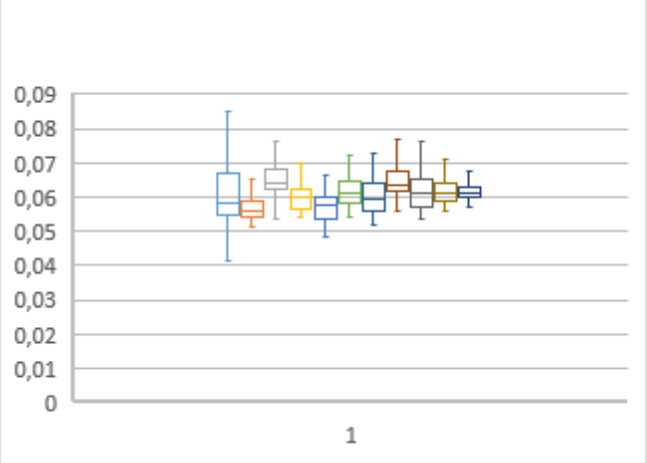


*Figure 6*

Il nodo Fog è stato testato per analizzare i tempi medi di risposta al variare del carico di richieste. Abbiamo simulato l'invio di richieste da parte di 10 utenti, 100 utenti e 1000 utenti (ognuno possedente tutti i possibili sensori). I risultati sono mostrati in tabella 1:

| *Numero utenti* | *Tempo medio di risposta (ms)* |
|---|---|
| 10 | 133.24 |
| 100 | 150.45 |
| 1000 | 175.10 |

*Tabella 1*

Come si può notare, i tempi medi di risposta subiscono un incremento (seppure minimo).

Probabilmente questo è dovuto al fatto che il servizio mqttSubscriber ed il broker mosquitto agiscono da collo di bottiglia. Infatti, non è possibile avere delle subscriptions condivise tra più servizi, ovvero non è possibile fare in modo che, nel caso in cui fossero presenti più servizi mqttSubscriber, il broker invii il messaggio ricevuto solo ad uno di essi permettendo la gestione in parallelo di più messaggi.

## SVILUPPI FUTURI

Tra i possibili sviluppi dell'applicazione sono stati individuati:

- Utilizzo di sistemi che stabiliscano il percorso stradale migliore che l'ambulanza deve seguire considerando anche il traffico invece dell'attuale sistema che considera la distanza in linea d'area.
- Utilizzo di cluster di broker MQTT invece che un singolo broker per ogni nodo Fog.
- Utilizzo di un architettura gerarchica all'interno del Fog-layer che consenta una migliore scalabilità, tolleranza ai guasti e performance.
- Utilizzo di modelli di machine learning che permetta di effettuare una prima diagnosi del paziente a partire dai dati raccolti.

## RIFERIMENTI


[1] https://mosquitto.org/

[2]. https://pypi.org/project/paho-mqtt/#description

[3] https://pypi.org/project/yagmail/

[4] https://geopy.readthedocs.io/en/stable/

[5]https://docs.aws.amazon.com/elastic-beanstalk/index.html

[6] https://www.terraform.io/docs/index.html