Distributed Gathering Detection System

Monitor and Control COVID-19 Pandemic

Andrea Paci
Ingegneria dell'Informazione
Università degli studi Roma Tor
Vergata
Roma Italia
andrea.paci1998@gmail.com

ABSTRACT

Questo documento è atto a presentare il progetto elaborato nel corso di Sistemi Distribuiti e Cloud Computing tenuto dalla professoressa Cardellini Valeria. Lo scopo dell'applicativo sviluppato è quello di fornire un sistema distribuito e decentralizzato per il monitoraggio e la rilevazione di assembramenti nelle strutture publiche come ospedali e ristoranti. Grazie ai dispositivi di tutti i giorni è possibile verificare la presenza di assembramenti con una precisione molto alta: un semplice smartphone può rilevare il numero di dispositivi grazie ad un comune hardware Bluetooth, e quindi il numero di persone nel raggio di ~10 metri. Il sistema ideato si basa sul pattern a **code di messaggi publish/subscribe basato su topic** con l'aggiunta della **context awareness:** il *contesto* delle entità influenza l'interazione e lo scambio di informazioni con il sistema. Un esempio di contesto è la **posizione**: è ragionevole pensare che una determinata informazione, fortemente correlata ad una certa posizione, possa essere di interesse solo per un numero più ristretto di *subscribers*, cioè per quelli che si trovano nelle vicinanze; e quindi limitare l'invio ad una cerchia più stretta di destinatari. I principali attori di questo scenario sono due:

- **Publisher**: l'entità che si occupa di raccogliere le informazioni dell'ambiente esterno e di inoltrarle verso il sistema
- **Subscriber:** l'entità che ha un interesse verso le informazioni condivise dai vari publisher

In un contesto reale questo si potrebbe applicare come segue: ogni struttura installa al suo interno un *modulo hardware con tecnologia Bluetooth* allo scopo di trovare il numero di persone all'interno dell'edificio, segnalando opportunamente le variazioni di quest'ultimo, insieme ad altre informazioni che possono essere usate per monitorare l'ambiente più efficientemente. Questo verrà definito come **Publisher**: saranno le varie strutture che misureranno e comunicheranno in maniera *asincrona* gli eventi di interesse nel contesto preso in esame. I possibili usufruitori di queste informazioni sono sia i cittadini sia enti speciali appositi per monitorare la situazione relativa alla diffusione del virus.

I cittadini hanno interesse ad apprendere informazioni quali:

- *concentrazione* di persone in un edificio (espresso come *persone/m*^{2})
- presenza di persone risultate positive al COVID-19 che hanno frequentato il luogo nel recente periodo

Così come I cittadini, anche gli *enti speciali* possono usufruire di questi dati per agire in maniera più tempestiva nelle situazioni di possibile emergenza. Facendo un esempio, è nell'interesse di entrambe le categorie sopracitate sapere che un luogo presenta un alto numero di clientela: il cittadino potrà fare una scelta informata su dove è più sicuro recarsi, evitando posti particolarmente affollati. D'altra parte, anche un ente istanziato per contere la pandemia ha interesse nel sapere la presenza di assembramenti per poter intervenire. Chiaramente, questi due attori rappresentano i **subscribers** della nostra infrastruttura infromatica.

In questo scenario particolare, è logico dedurre che un sistema di questo tipo è utile solo nel momento in cui le informazioni veicolate sono filtrate in base all'interesse che può avere chi le riceve. Un primo filtro è quello del **Topic**: ogni individuo è più o meno interessato a determinate categorie, quali ad esempio *ristorazione*, *strutture sanitarie* e *farmacie*. Quindi se l'utente in questione non ha interesse nel ottenere infromazioni riguardanti *circoli sportivi*, non è necessario riportargli degli aggiornamenti.

Un altro filtro è la **Posizione**: se una nuova notifica proviene da un luogo distante all'utente, lui non avrà guadagno/interesse nel riceverla.

Questi due filtri, oltre che a portare ad una esperienza col sistema migliore, evita l'utilizzo ingiustificato dell'infrastruttura per la comincazione di notizie non richieste dagli utenti finali. Come si vedrà successivamente, applicare come criterio la *posizione* porta dei guadagni prestazionali non trascurabili.

KEYWORDS

Sistemi, distribuiti, AWS, Publisher, Subscriber, Golang, Code, Fifo, Decentralizzato, Covid-19, Monitoraggio

ACM Reference format:

Andrea Paci 2019. Distributed Gatherings Detection System: Monitor and Control Covid Pandmeic.

1 Architettura di sistema e requisiti

Il sistema da realizzare si può schematizzare come di seguito:

1) **Publishers** che inviano aggiornamenti riguardanti la situazione che gli compete
2) **Subscribers** che ricevono gli aggiornamenti citati sopra e li usano per compiere una scelta informata
3) Un **mediatore** che mette in comunicazione le due figure presentate sopra, cioè quello che nell'architettura PUB/SUB viene definito come **Broker**.

Il nocciolo del sistema è il **broker**: sarà lui che si occupera di raccogliere ed elaboare i messaggi ricevuti dai *publishers* e di veicolarli verso i *subscribers*. Il broker può essere modellato come un insieme di *requisiti funzionali e non*:

- Deve permettere una **comunicazione affidabile** tra due enti che non necessariamente devono entrare in contatto tra loro (*space decoupling*)
- Deve essere in grado di memorizzare i messaggi ricevuti e di renderli disponibili finchè i *subscribers* non li processano opportunamente.
- Deve permettere all'intera infrastruttura di comunicare in modo *time decoupled*.
- Deve saper utilizzare il *contesto*, in questo caso quello contesto geografico, in maniera opportuna
- Deve permettere la *personalizzazione* delle preferenze di chi riceve i messaggi. cioè iscrizione ai topic
- Deve garantire che i destinatari ricevano *almeno una volta* le notifiche di loro interesse
- Il sistema deve essere in grado di gestire un grosso quantitativo di informazioni e utenti, deve quindi essere in grado di *scalare* le proprie risorse al bisogno
- Deve poter fornire dei parametri di configurazione da modificare all'occorrenza tale che non occorrà un riavvio del sistema.
- Deve essere fruibile alla maggioranza dei cittadini.
- Deve tollerare *guasti* e malfuzionamenti senza l'intervento di supporto tecnico esterno.

1.1 Implementazione

Mettendo insieme i requisiti descritti nel paragrafo precedente sono state fatte le seguenti scelte implementative:

- I **publishers** rappresentano le strutture pubbliche, le quali installeranno al loro interno degli appositi moduli in grado di *captare* gli ingressi e le uscite. Inoltre essi inviano le notifiche con un determinato **contesto** e **topic**, cioè la posizione geografica dell'edificio e la descrizione del tipo di attività esercitata nello stesso (es. attività di intrattenimento, vendita al dettaglio, ecc.). Devono anche poter inoltrare segnalazioni *personalizzate*: possono notificare, oltre che ai dati raccolti dall'apposito hardware, eventi di interesse riguardanti il settore (o topic) di appartenenza.
- I **subscribers** rappresentano l'utenza che fa uso delle segnalazioni effettuate. Essi possono esprimere la preferenza a determinati *topics*, quindi devono avere la possibilità di *aggiungere o rimuovere preferenze*. Inoltre devono essere in grado di comunicare la propria posizione geografica, la quale può variare nel tempo, e di disiscriversi completamente dal sistema.
- Il **Broker** deve essere *distribuito* per fornire un'alta capillarità e raggiungibilità del sistema. Questo deve essere il punto di incontro tra tutti gli utenti che interagiscono con l'infrastruttura, deve quindi essere in grado di gestire la comunicazione tra due o più utilizzatori e di analizzare il contenuto dei messaggi ricevuti per poter estrarre ed usare le informazioni di contesto (posizione) e di preferenza (topic), per veicolare correttamente i messaggi. Deve poter gestire un *livello di comunicazione* semplice e accessibile ai più, per esempio con le **API REST**. Infine il broker deve poter **scalare** le proprie capacità a seconda della richiesta e del carico di lavoro (gestione delle risorse).

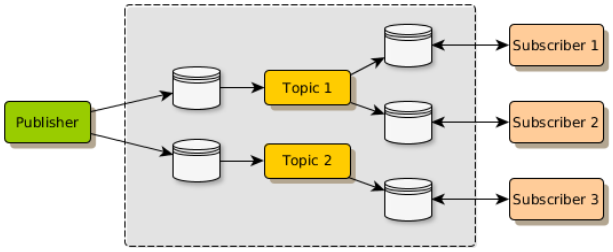


Figure 1: Una rappresentazione schematica di un sistema publish/subscribe

- La **comunicazione** deve avvenire attraverso un sistema basato su **code** che garantiscano un ordinamento temporale e che i messaggi vengano effettivamente recapitati. Non è necessaria la compresenza dei due comunicanti per far si che lo scambio di messaggi avvenga: il broker gestisce una comuncazione **asincrona e** scollegata da **vincoli di tempo e spazio**.

- La **posizione** è utilizza il concetto di **geofence:** la topografia di una città è suddivisa in *quadranti*, identificati tramite cordinate X,Y, di dimensione arbitraria. Si è ritenuto valido utilizzare come dimensione per la geofence un lato di 20 metri. La posizione notificata dai publishers e dai subscribers non sarà quindi espressa come *latitudine e longitudine*, ma piuttosto come una coppia di valori che identificano il *quadrante* su dove essi si trovano

Figure 2: Una rappresentazione schematica del concetto di Geofence

2 Framework e tecnologie utilizzate

In questo paragrafo verranno presentate nel dettaglio le teconologie e i framework utilizzati per lo sviluppo del sistema.

2.1 Broker

Il broker è stato realizzato utilizzando:

- **Golang** come linguaggio di programmazione: golang offre un alta semplicità di utlizzo, ma allo stesso tempo si rivela un linguaggio molto efficace e leggero. Caratteristica fondamentale è la sua *portabilità*. Con Go è stata implementata l'intera logica del broker insieme alla configurazione di un server **HTTP** per la gestione di chiamate ad **API REST**. La scelta di utlizzare l'architettura **Restful** si basa sulla sua *semplicità* e *leggerezza* oltre che alla sua *versabilità* rispetto al numero di sistemi software e hardware lo supportano. Inoltre, basandosi sul protocollo HTTP, è possibile utilizzare anche un *semplice browser Internet* per interagire con il sistema. Le API REST portano un incremento di prestazioni *significativo* rispetto ad altri protocolli di comunicazione basati su TCP. Come supporto a Go sono state usate la libreria **gorilla/mux,** che facilita la creazione di un server HTTP con supporto ad API REST, e le librerie offerte da **AWS** per l'interazione con l'infrastruttura cloud.
- **DynamoDB** come databse *NoSQL* offerto da **Amazon Web Services** per mantenere le informazioni neccessarie alla corretta esecuzione del sistema:
  - **Subscribers:** una tabella contenente tutte le informazioni rilevanti per identificare e comunicare con i subscriber, tra cui i **topics** di interesse e la **posizione** (espressa come coppia X,Y rappresentativa della quadrante a cui appartiene ad un certo istante).
  - **Configurazione**: una tabella contenente alcuni parametri di configurazione modificabili a *run time* attraverso l'opportuna interfaccia esposta da AWS o con quella sviluppata ad-hoc. I parametri di configurazione possono essere anche essere aggiornati mediante delle opportune chiamate API REST

  Con DynamoDB non è necessario preoccuparsi di eventuali problemi di *consistenza* e di *guasti*: grazie alla sua natura *replicata* e *scalata su molteplici nodi* è un database estremamente affidabile e reattivo.
- **Elastic Beanstalk:** per il deployment del broker si è scelto di utilizzare il **PaaS** offerto da Amazon per la sua alta accessibilità. Garantisce caratteristiche come: *tolleranza ai guasti, scalabilità* e *alta configurabilità* dell'ambiente di esecuzione. Grazie a questa tecnologia è possibile utilizzare un insieme di risorse di AWS (come macchine EC2, bucket S3, ecc.) in maniera automatizzata, ottimizzando al meglio il funzionamento del sistema. L'impiego di Elastic Beanstalk comporta come conseguenza l'utilizzo indiretto degli altri servizi proposti da AWS:
  - **Amazon S3** per la memorizzazione del **codice sorgente** su un bucket
  - **Amazion EC2 e ELB** come macchine virtualizzate per eseguire l'applicativo del broker.
  - **Virtual Private Cloud** e **Security groups** per la creazione di una rete *virtualmente privata* dove le macchine che vi sono dentro possono comunicare tra loro isolate dal resto della rete.
  - **Elastic Load Balancing e Auto scaling** i quali hanno permesso di creare un'applicazione altamente scalabile e resiliente alle fluttuazioni del carico di lavoro a cui è sottoposto il *broker*. A seguito di un aumento/dimunuizione delle richieste al broker, l'Auto scaling può scegliere se **aumentare/diminuire** il numero di *istanze EC2* su cui far girare l'applicazione, e reindizzare automaticamente il traffico alla macchina meno sottoposta a stress.
  - Tramite I **Cloudwatch alarms** è possibile scegliere le politiche con cui effettuare il

resize e lo scaling del cluster di macchine EC2

- **Amazon SQS** per la creazione di code **FIFO** utilizzate nello scambio di messaggi tra *publisher-borker* e *broker-subscribers*. Inoltre SQS supporta vari formati per l'invio di messaggi (es. JSON) e permette di utilizzare tramite chiamate API REST per immettre o ricevere messaggi nella coda. I messaggi possono essere con e senza **attributi**. SQS consente sia la semantica **at-least-once** che **exaclty-once** con ordinamento **FIFO** o **out-of-order**. SQS si poggia su molteplici repliche rendendolo un prodotto altamente affidabile e scalabile. SQS offre anche le semantiche **Transaction based delivery** e **Timeout based delivery**, entrambe implementate nel sistema. Infine, offre la modalità **Long polling:** l'interrogazione alla coda per il *fetch* di nuovi messaggi interroga tutte le *istanze replicate* della coda SQS per *limitare il numero di chiamate alla stessa* e *garantire di ricevere tutti i messaggi accodati* ma non ancora elaborati. L'utilizzo che è stato fatto di SQS emula le caratteristiche di **Amazon Simple Notification Service**:
  - per il broker esiste una un'unica coda per la **ricezione dei messaggi** dall'esterno (il che non genera un collo di bottiglia dato che le sue risorse sono distribuite e replicate attraverso la rete, anche se vi si accede come se *virtualmente fosse un'unica risorsa*)
  - una coda per ogni **subscriber** su cui quest'ultimo effettuerà le interrogazioni per ottenere un eventuale messaggio.

  Mentre la coda SQS del **broker** è presente sin da subito, le code dei vari subscribers *vengono create* al momento della registrazione e vengono eliminate succcesivamente ad una richiesta di disiscrizione dal sistema.
- **Sistema di monitoring** integrato in AWS per l'analisi delle performance e del carico a cui l'infrastruttura è sottoposta.
- Il formato **JSON** per scambiare le informazioni nelle chiamate API REST.

2.2 Publisher e Subscriber

I publisher ed i subscriber utilizzano le chiamate **API REST** per interagire con il broker per le operazioni di **registrazione**, **aggiunta/rimozione di topic**, **deregistrazione** e **aggiornamento** della **posizione**, mentre per l'invio e la ricezione di messaggi su SQS vengono usate le API fornite da Amazon. Queste due entià sono completamente configurabili e non contengono niente di *hard-coded*: da un file denominato **config.json** recuperano dei parametri basilari per l'interazione con il sistema.
Successivamente alla fase di registrazione da parte di uno dei due, il Broker fornisce ulteriori parametri per il corretto funzionamento dei due applicativi. Entrambi sono stati sviluppati con il linguaggio GO, ma dato l'impiego di API SQS e REST, non c'è un vincolo su quale linguaggio/tecnologia utlizzare per la loro implementazione. Caratteristiche come l'aggiornamento della posizione e il conteggio di persone attualmente presenti nell'edificio sono *mocked,* cioè simulate, data l'impossibilità di ricreare questo scenario. Entrambi forniscono anche una versione **interattiva**, cioè dove l'utente ha pieno controllo su cosa mandare, a chi mandarlo e a quale distanza nel caso del publisher, e la coda utilizzare nel caso del subscriber.
E' stato fornito anche un **modulo extra** che funge da logger **remoto** hostato su un Beanstalk separato: le 3 principali entità coinvolte nel sistema comunicheranno al logger le operazioni effettuate, e poi quest'ultimo le inoltra a tutti i dispositivi in ascolto. Questo modulo è usato per monitorare il comportamento del sistema sia per motivazioni di **testing** sia come **dashboard** dove vengono riportati **eventi importanti** come *alta concetrazione di persone in un luogo* o *segnalazione di persone positive al Virus*.

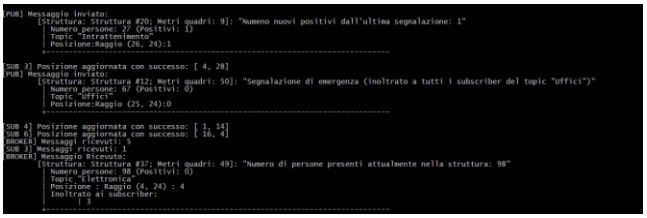


Figure 3: Output del logger remoto catturato con netcat

E' presente anche un modulo per il **logging in locale** che permette di estrarre informazioni utili, anche a runtime, sul flusso di esecuzione.

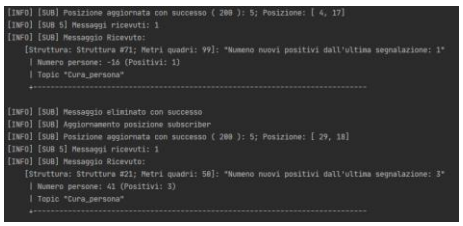


Figure 4: Output del logger locale

Broker, subscriber e publisher utilizzano **Docker** sia per l'esecuzione degli applicativi che per fare il *deployment* sul Cloud AWS. Docker isola il workspace dal resto del filesystem e permette di scegliere tra una ampia selezione di environments, quello che più si adatta di più all'applicazione sviluppata, sia per una esecuzione locale che per una sul cloud. L'immagine scelta per l'applicativo è **alpine**: una *distro UNIX* molto leggera, perfetta per l'esecuzione tramite *container*.

3 Motivazioni delle scelte architetturali

L'aggiornamento della posizione del subscriber è stato sviluppato come *evento notifica*: invece di avere il broker che interroga il subscriber ad intervalli regolari per capire se è variata la sua posizione, sarà il subscriber a segnalare al broker quando la sua posizione cambia.

Si è scelto di usare una coda **SQS** con **ordinamento FIFO**, la quale tenta di garantire la sematica **exactly once** ritardando l'invio di messaggi duplicati o tentativi di rinvio a 5 giorni. Per garantire la coda FIFO è stato necessario l'impiego di **groupID**, dove tutti i messaggi con lo stesso ID saranno ordinati cronologicamente. Per offrire la semantica *exactly once* si usa il ***deduplication ID***, utilizzato per marcare i messaggi con un certo ID al fine di riconoscere eventuali duplicati. Una valida alternativa poteva essere **SNS** (Simple Notification Service) invece che SQS, il quale fornisce nativamente il supporto ai topic, però non da garanzie ne di ordinamento ne di semantiche. Si è scelta una soluzione ibrida tra SQS e SNS sviluppata ad-hoc: le code SQS vengono create dinamicamente per dare la possibilità di inviare i messaggi a destinazioni mutiple come con SNS (cosa non concessa con una singola coda SQS) ma comunque garatendo l'affidabilità offerta dalle code SQS ma non garantita da SNS. Si è ritenito più corretto in questo contesto utilizzare un ordinamento **FIFO** piuttosto che casuale: data la natura dei messaggi che si intende scambiare sul sistema, il loro ordinamento risulta una caratteristica di primaria importanza, anche se a discapito di un incremento della latenza. Per limitare il numero di richieste alla coda con risulato nullo è stato utilizzato il long polling, cioè le interrogazione alla coda perdurano per un intervallo di tempo più esteso. Inoltre, grazie alla sua infrastruttura replicata, gestisce automaticamente la perdita dei messaggi, togliendo una responsabilità importante allo sviluppatore.

E' stata adottata la soluzione proposta da **Elastic Beanstalk** piuttosto che l'impiego coordinato dei servizi proposti da AWS come EC2, S3, VPC, Security Groups, Load Balancing e Autoscaling. **Elastic Beanstalk** è stato fatto per rispondere a numerose necessità diverse, infatti sono proposti dei *setup guidati* a seconda del tipo di applicazione, linguaggio di programmazione e framework utilizzati. Per ogni configurazione proposta, EB propone un piano di scaling che si adatta al sistema che si vuole esportare sul cloud, rendendo minima e spesso inutile la configurazione dell'infrastruttura e dei parametri nel caso di applicazioni più semplici. Un'ulteriore motivazione per cui si è usato EB è il **deployment**: con EB il deployment è molto veloce è intuitivo, non richede la configurazione dell'enviroment di esecuzione e fornisce delle *facility* che automatizzano fino a renderlo un semplice comando da CLI.

Si è scelto di supportare 3 tipologie di messaggi diversi:

- **Variazione di numero delle persone in una struttura:** il publisher che rileva una variazione nel numero di persone, notifica tutti i subscriber entro un raggio definito nella segnalazione stessa. Per notificare i subscriber, il publisher manda il messaggio alla **coda globale** su cui il broker lo riceve; successivamente il broker analizza gli attributi del messaggio che si riferiscono alla posizione del publisher ed al raggio di pubblicazione. Il messagio viene inoltrato a tutti i subscriber che sono interessati al topic del publisher e rientrano nell'area deisngata di interesse.
- **Riscontro di persona/e positiva**: I publisher possono segnalare se nei giorni antecedenti l'edificio sia stato frequentato da una persona risultata positiva. Questo messaggio viene inoltrato a tutti i subscriber indipendentemente dall'interesse nel settore di operazione del publisher (il *topic*) per un raggio specificato globalmente (impostato di default a ~5km nel file di configurazione).
- **Messaggio di emergenza relativo ad un topic:** Se il broker riceve un messaggio con *raggio di interesse* pari a 0, lo inoltra a tutti i subscriber iscritti al topic contenuto nel messaggio.

Utilizzando altri attributi presenti nel messaggio, il broker può segnalare al *logger remoto* **situazioni di emergenza**:

- **concentrazione di persone/m^2 superiore al consentito**: se il rapporto tra persone presenti nel luogo e i metri quadri è superiore ad una certa soglia (specificata nei file di configurazione) manda una segnalazione al logger remoto con allegato il problema riscontrato
- **Riscontro di persone positve**: oltre ad inoltrare a tutti i subscriber il messaggio contenente la segnalazione di positività di alcuni soggetti, lo inoltra anche al logger remoto.

Causa l'impossibilità di ricreare un ambiente reale su cui provare il sistema, alcuni elementi sono stati "simulati":

- **Posizione e variazione di posizione del subscriber:** per simularlo è stato sufficiente avere una *goroutine* che dopo un determinato intervallo di tempo manda il comando di *aggiornamento posizione*, impostando come valori per la nuova posizione dei numeri randomici
- **Posizione del publisher:** la posizione dove è ubicato il locale relativo a quel determinato publisher è generato in maniera casuale
- **Eventi segnalati dal publisher:** gli eventi che il publisher notifica sono simulati con una scelta random.

Gli applicativi di esempio sviluppati per simulare publisher e subscriber gestiscono *input a riga di comando* per definire le proprietà dei due:

Subscriber:

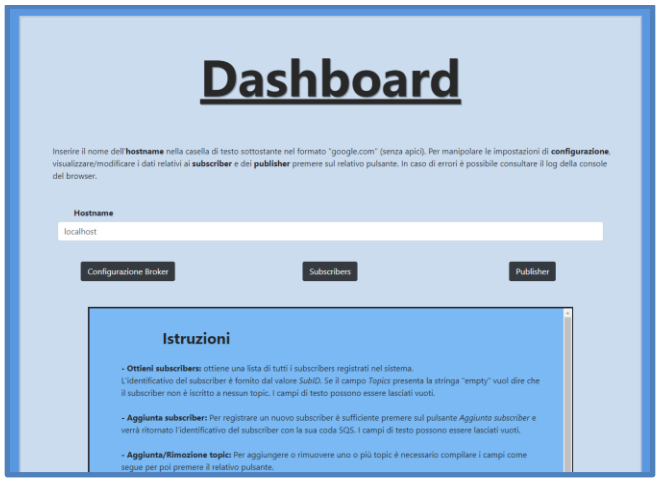
- **Interattivo:** se il primo parametro è ugale alla stringa **"i"** allora l'applicativo viene eseguito in modalità interattiva, altrimenti viene simulato un comportamento casuale
- **Subscriber ID** identificativo per distinguersi da altri subscriber
- **Coda di ricezione:** specifica l'URL della coda SQS creata dal broker e usata per ricevere messaggi dal subscriber
- **Posizione X e Y** identificatore del *quadrante* dove è presente il subscriber
- **Topics:** I topic a cui ci si vuole iscrivere

Publisher:

- **Interattivo:** uguale al subscriber
- **Nome** nome identificativo della struttura
- **Topic** topic a cui mandare il messaggio
- **Numero persone:** il numero di persone presenti nell'edificio
- **Posizione X e Y** identificatore del *quadrante* dove è presente il publisher
- **Raggio:** Raggio entro il quale i subscribers vengono notificati
- **Metri quadri:** metri quadri della struttura

4 Ulteriori dettagli implementativi

E' stata sviluppata anche una interfaccia Web come **console di amministrazione**: da lì si può accedere ai parametri di configurazione, simulare operazioni del publisher e del subscriber:



La dashboard è stata fatta usando HTML e Javacript e si limita a mandare richieste al broker mediante API REST. L'output viene poi scritto su di una *textarea* per essere analizzato dall'utlizzatore. Questa è una dashboard che con l'aggiunta di ulteriori features e l'accoppiamento con il *logger remoto* descritto prima può diventare uno strumento per controllare e gestire prontamente situazioni che richiedono l'intervento delle autorità.

Per ricevere i messaggi inoltrati dal **logger remoto** è suffciente un qualsiasi client che gestisce **stream TCP** come per esempio **netcat**. Il logger distingue le sorgenti da cui ricevere le segnalazioni e i destinatari a cui vanno recapitati con uno scambio iniziale di messaggi:

- Se il logger riceve il carattere **"l"** marca quella connessione come *in ascolto*, e gli inoltrerà tutti i dati ricevuti dalle sorgenti
- Se il logger riceve una qualsiasi altra combinazione di caratteri, segna la connessione come *sorgente*.

5 Testing e analisi dei risultati

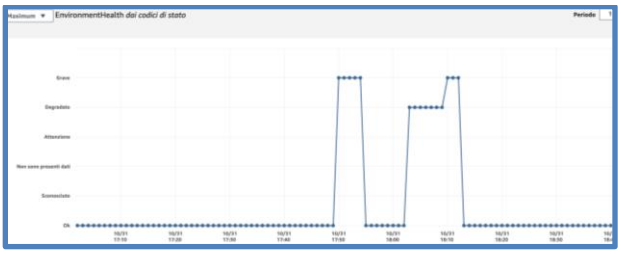
Per comprendere l'efficienza del sistema sviluppato sono stati eseguiti due tipi di test:

- **Test sulla scalabilità dell'infrastruttura Beanstalk**
- **Test sull'efficienza dell'introduzione della contex awareness**

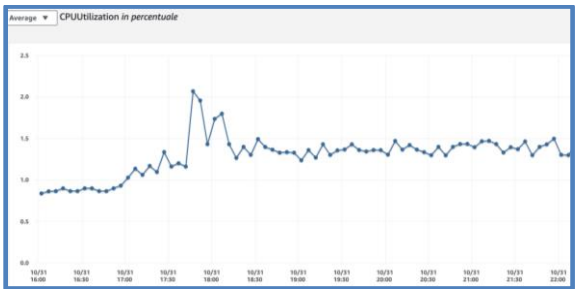
Per testare la scalabilità del sistema sono stati cambiati i parametri di configurazione degli applicativi dei publisher e dei subscriber per generare un grande numero di richieste con API REST ad alta frequenza, che pone sotto stress l'infrastruttura e permette di verificare se e come viene gestito il carico in situazioni di picco. Per analizzare il comportamento del sistema sotto stress ci si è avvalsi del **monitor integrato di AWS** e del *logger remoto e locale*

Questo grafico mostra l'andamento della **latenza del sistema** nel momento in cui viene messo sotto stress. Il livello di carico a cui è stato sottoposto è rimasto costante in tutto l'intervallo di tempo mostrato. E' evidente come all'inizio il sistema fatica a reagire al picco di richieste, introducendo un ritardo non indifferente nel sistema fino a scartare le richieste diventando inattivo per un periodo di tempo limitato. Questo può essere visto anche nel grafico sottostante che riporta la

situazione del sistema da un punto di vista generale (viene infatti classificato come *in salute* o *grave*). E immediato vedere come l'intervallo in cui il sistema segna latenza 0 quando scarta le richieste in ingresso, che coincide col momento di massima criticità.



Successivamente a quella fase, il sistema reagisce e alloca più risorse, aggiungendo una macchina EC2 al cluster. Dopo il secondo picco viene aggiunta un altra macchina EC2, arrivando ad un totale di 3 macchine. L'intervento dell'autoscaling di Beanstalk, nonostante non fosse stato **proattivo**, è stato in grado in pochi minuti di istanziare le risorse necessarie per gestire il traffico in entrata. Beanstalk è stato in grado, senza previa configurazione, di adattare le risorse al carico da gesitre nella maniera più opportuna. Chiaramente la perdità di richieste, anche se per pochi minuti, potrebbe non essere indifferente in alcuni contesti, specialmente quelli più critici dove non è ammissibile *down* del sistema.

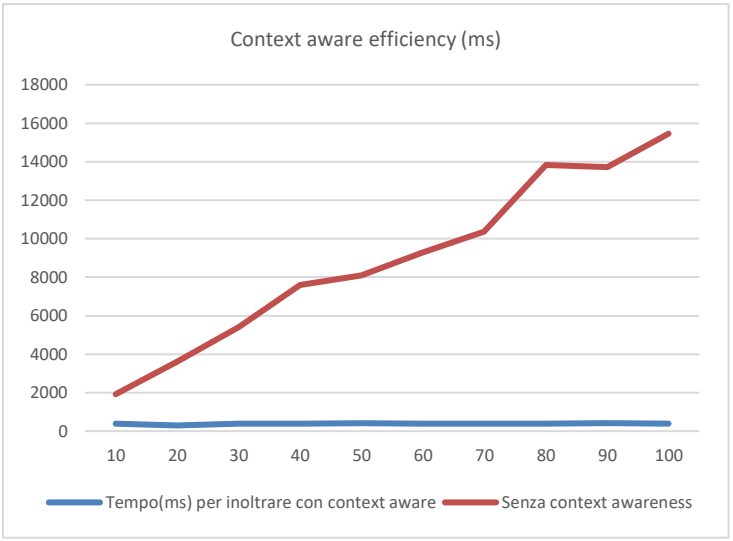


L'ultimo grafico riporta l'andamento percentuale dell'utilizzo della CPU sempre nel periodo di picco delle richieste. E' immediato osservare come ad eccezione di un iniziale sbilanciamento dai valori, il sistema si porta in poco tempo verso una situazione più stabile grazie all'aggiunta di ulteriori istanze EC2.

Per quanto riguarda l'**efficienza introdotta con la context awareness** si è preso in esame esclusivamente il tempo impiegato da quando il broker riceve un messaggio a quando finisce di inoltrarlo a tutti i subscriber a cui è destinato. L'operazione di invio dei messaggi da parte del broker è **bloccante**, quindi più tempo impiega ad inviare I messaggi e più tempo il sistema sarà in *stallo*. La scelta di non tenere conto di tutto il percorso del messaggio (cioè da quando viene inviato dal publisher fino a quando è ricevuto dal subscriber) è motivata dal fatto che si effettuerebbero misure inaccurate a causa della forte componente aleatoria a cui è sottoposta la rete, rendendo così difficile l'interpretazione dei dati. Gli ambienti di test messi a paragone sono i seguenti:

- Il publisher *non utilizza il contesto geografico* e invia un messaggio destinato a tutti i subscriber interessati al topic in oggetto.
- Il publisher *utilizza il contesto geografico* e invia un messaggio destinato solo ad un subscriber: l'unico che è sufficientemente vicino da renderlo uno dei destinatari.

Queste due ambienti di test sono stati iterati ogni volta con un numero sempre crescente di subscribers.



Il grafico mostra sull'asse delle ascisse il numero di subscribers registrati nel sistema e sulle ordinate il tempo in millisecondi per effettuare l'inoltro del messaggio.

Dal grafico si evince che il contesto migliora drasticamanete le performance: con il context aware l'inoltro del messaggio rimane abbastanza stabile e costante nel tempo, mentre senza context awareness, i tempi crescono linearmente. Questa forte disparità e da ricercare nel fatto che, per come sono stati formulati i test, nel caso di *context-awareness* il messaggio viene inoltrato solo una volta (cioè solo al subscriber sufficientemente vicino), mentre *senza context-awareness* viene inoltrato per *n volte*, dove però $n - 1$ sono da considerarsi operazioni *inutili* visto che quel sottoinsieme di subscribers non avrà interesse a ricevere quel messaggio.

Con questi dati si evince che in situazioni analoghe a questa, l'utilizzo della posizione per determinare l'interazione con una risorsa in un sistema sottoposto ad alto carico e fortemente distribuito, è una strategia vincente.

Da notare che gli alti tempi di latenza mostrati sono da ricercare principalmente nell'impiego di risorse ubicate negli USA, ma che comunque non modifica l'andamento dei grafici

6 Limitazioni e sviluppi futuri

Lo sviluppo del sistema è stato limitato a causa di molteplici fattori.

Sotto il punto di vista della **sicurezza:** il sistema non fornisce un sistema di autenticazione o di crittografia, quindi per un attaccante è possibile manipolare le informazioni scambiate, prendere il controllo di publisher e subscribers arbitrariamente e generare un numero di richieste tale da mandare il sistema in stallo. E' possibile arginare il problema con l'impiego di protocolli più sicuri, come HTTPS, basati su TLS, e un sistema di autenticazione basato su **API Keys**, collaudato già ampiamente nei sistemi odierni che implemetano l'architettura REST, che permette di autenticare il messaggio, verificandone la fonte, e limitare il numero di messaggi che si possono ricevere.

Ulteriore limitazione è stata quella di non poter utilizzare **IAM Roles**: con questa utility è possibile limitare le possibilità di interazioni degli utenti con il sistema, definendo a grana fine quali operazioni sono permesse e quali no.

Una limitazione importante è stata quella di non avere **un environment reale** su cui testare il funzionamento del sistema, dovendo così simulare quegli eventi e quelle funzionalità correlate con l'ambiente in una situazione reale, come ad esempio la posizione geografica di un'entità.

Infine, sarebbe sarebbe stato opportuno utlizzare risorse AWS più vicini all'ambiente di utilizzo, anche se non si tratta di un sistema *time critical*, avere risosre replicate in più siti permette di avere una distribuzione del carico migliore e un tempo di risposta più adeguato.