Sistema con coda di messaggi di tipo Context-aware in Go per la gestione di un magazzino

Alessandro Amici
Corso di laurea magistrale in Ingegneria Informatica, Università degli studi di Roma "Tor Vergata"
alessandro.amici@alumni.uniroma2.eu

Cecilia Calavaro
Corso di laurea magistrale in Ingegneria Informatica, Università degli studi di Roma "Tor Vergata"
cecilia.calavaro@alumni.uniroma2.eu

Roberto Pavia
Corso di laurea magistrale in Ingegneria Informatica, Università degli studi di Roma "Tor Vergata"
roberto.pavia@alumni.uniroma2.eu

***Abstract*—Sistema che gestisce e monitora la comunicazione tra dei sensori e dei robots presenti in diverse aree di un magazzino, tramite una coda di messaggi di tipo context aware che implementa il pattern publish-subscribe.**

1. Architettura di Sistema

E' stata realizzata un'applicazione software per il monitoraggio dei parametri in un magazzino tramite un sistema a code di messaggi basato su topic e di tipo context-aware. In particolare, si è pensato di utilizzare l'applicazione in uno scenario in cui la superficie di un magazzino viene suddivisa in delle aree chiamate **settori**, all'interno dei quali sono presenti dei **sensori** che effettuano delle misurazioni circa parametri quali **umidità, temperatura e movimento** rilevati nel settore in cui si trovano. In ogni settore possono essere presenti dei robot, detti **bots**, che ricevono i dati prelevati dai sensori appartenenti allo stesso settore. Il magazzino può essere organizzato anche in più piani ed ognuno di essi può essere gestito da una componente dell'applicazione, ossia il lato front-end che verrà analizzato dettagliatamente più avanti nel paragrafo 3.2. Alla base del sistema è stato sviluppato un pattern di tipo **publish-subscribe**: la comunicazione tra i sensori ed i bots avviene attraverso un **broker di messaggi**, contenenti i dati rilevati dalle misurazioni dei sensori che vengono consegnati a più riceventi: i **publishers**, ovvero le componenti che pubblicano i messaggi, sono identificabili nei sensori, mentre i **subscribers**, ossia i bots, ricevono messaggi inerenti ai topic a cui sono iscritti ed eventualmente al proprio contesto, inteso come "area geografica". Inoltre, da riga di comando è possibile specificare se eseguire l'applicazione con o senza contesto tramite l'inserimento del parametro "ctx", in modo da poter poi fornire un'analisi prestazionale più accurata del software e studiare le dinamiche dei cambiamenti che avvengono a livello di performance.

L'architettura del sistema prevede due applicazioni, una per il **back-end** ed una per il **front-end**. In entrambi i casi è stato sviluppato sia un lato **server** che uno **client**, in modo da poter interagire l'una con l'altra tramite l'invio di messaggi con il protocollo http **REST API**. Come accennato precedentemente, la comunicazione tra bots e sensori viene coordinata da un broker, che si occupa di ricevere i messaggi dai sensori, di salvarli e poi di pubblicarli, inoltrandoli ai bots in accordo con le sottoscrizioni effettuate. Per merito della supervisione di quest'intermediario, è stato possibile sviluppare un sistema flessibile e che possa sfruttare al meglio la distribuzione e la scalabilità, grazie al **disaccoppiamento** che vi è tra le componenti dell'applicativo, sia nel tempo che nello spazio. Nel primo caso, il sistema permette che i sensori ed i bots non debbano essere presenti insieme nello stesso istante quando la comunicazione ha luogo; nel secondo, i componenti non devono conoscersi l'un l'altro in quanto è il broker stesso che si occupa di smistare i messaggi tra i corretti destinatari.

Infine, l'applicazione è stata distribuita ed eseguita su **AWS** utilizzando il servizio di **Elastic Beanstalk**. Un altro servizio cloud utilizzato è quello di Amazon **DynamoDB**. Quest'ultimo è un servizio di database NoSQL interamente gestito che combina prestazioni elevate e prevedibili con una scalabilità ottimale. E' quindi una base di dati distribuita che gestisce direttamente il provisioning dell'hardware, installazione e configurazione, replica, applicazione di patch al software e dimensionamento del cluster. Elastic Beanstalk invece permette di effettuare il **deploy** e di gestire rapidamente l'applicazione nel cloud AWS; quindi una volta che quest'ultima è stata caricata, il servizio coordina automaticamente i dettagli di provisioning delle risorse AWS, quali le istanze di Amazon EC2 per l'esecuzione dell'applicazione, S3, Simple Notification Service, CloudWatch, autoscaling, and Elastic Load Balancer. In particolare, per effettuare il deploy le impostazioni sono state configurate nelle seguenti modalità: **us-east-ohio** per la regione, come nome dell'applicazione "**default**", come load balancer "**classic**", come DNS canonical name dell'applicazione "**default**"; in più, è stato utilizzato **docker** come containter, quindi il servizio effettua il detect automatico del dockerfile e lo lancia da una macchina chiamata "**64bit bit amazon linux**".

2. Scelte del design d'implementazione

Per la realizzazione del sistema sono state sviluppate **due applicazioni**, come accennato nel paragrafo 1.1: una di **front-end** ed una di **back-end**. Il lato front-end è stato codificato in **Python** e ha lo scopo di fornire un'interfaccia grafica che prevede un duplice utilizzo, ovvero sia quello di poter manipolare e gestire un piano del magazzino attraverso la GUI sviluppata, sia quello di mostrare in tempo reale ciò che sta accadendo all'interno del sistema. Per quanto riguarda la manipolazione del sistema, la GUI dispone di widgets, quali i bottoni, che permettono di effettuare le seguenti operazioni:
- **publish**: si possono creare uno o più sensori associati ad un determinato topic e settore, che pubblicano così nuovi valori sul broker;
- **subscribe**: si possono creare uno o più bots iscrivendoli ad un determinato topic all'interno di un settore, così che ricevano messaggi dal broker. Ogni bot può essere iscritto solo ad un singolo topic;

- **unsubscribe**: annulla l'iscrizione di un bot al suo topic;
- **resubscribe**: reiscrive un bot al topic a cui era precedentemente iscritto;
- **bot/sensor killing frequencies**: simula che un bot o un sensore vadano offline per un certo lasso temporale, quindi un crash di una componente del sistema.

D'altra parte, l'interfaccia grafica mostra anche ciò che sta accadendo nel sistema in ogni istante, in quanto riporta quali bots sono presenti in un determinato settore, a quale topic sono iscritti e quali messaggi ricevono. I sensori ed i bots vengono quindi rappresentati dal front-end, mentre nel lato back-end, scritto nel linguaggio di programmazione **Go**, sono stati sviluppati il broker di messaggi e tutte le attività necessarie al proprio compito di orchestrazione della comunicazione.
Poiché tra i requisti è stato richiesto un modello d'interazione con il sistema tramite **REST API**, la comunicazione tra le diverse componenti applicative è stata realizzata interamente sfruttando il protocollo **http**. Difatti è tramite richieste http che il front-end comunica al broker quali operazioni compiere tra quelle sopra elencate in base a quanto selezionato dall'utente. Il broker inserisce i sensori ed i bots, secondo quanto comunicatogli dal front-end, all'interno di liste separate che vengono impiegate nell'implementazione delle attività di publish e di subscribe. La semantica di comunicazione utilizzata è quella dell'**at-Least-Once**, che prevede che il broker si assicuri che i messaggi vengano consegnati con successo aspettando un ack per ogni messaggio inviato, e ritrasmettendolo se non riceve l'ack. La descrizione dell'implementazione del sorgente dell'applicativo comunque è riportata in maniera approfondita più avanti, nel paragrafo 2. Inoltre, per garantire **persistenza** nel sistema è stato utilizzato il servizio offerto da Amazon di **DynamoDB**. In particolare, dal back-end vengono create tre tabelle nel database:
- "**bots**": in cui vengono salvati i bots creati e quindi presenti nel sistema;
- "**resilience**": tiene traccia del forwarding di ogni richiesta di pubblicazione da parte dei sensori verso ogni bot iscritto a quel topic e contesto. Quando il broker è sicuro che l'inoltro del messaggio sia avvenuto, cancella la entry relativa a quel messaggio. Questa tabella è stata creata per assicurarsi che nel caso in cui il sistema subisca un crash, i messaggi vengano poi comunque consegnati;
- "**sensorsRequest**": in cui vengono salvate le singole richieste di pubblicazione da parte dei publishers.

3. Descrizione dell'implementazione

Il codice sorgente del sistema è suddiviso in due directories, chiamate **WBMQSystemProject**, in cui è stato sviluppato il lato back-end, e **WBMQ-Controller** per il lato front-end.

3.1. WBMQSystemProject - Back-End

All'interno della directory WBMQSystemProject, il sorgente è suddiviso in tre files chiamati:

- **http-api.go**, in cui è presente la main func dell'applicazione e che contiene le funzioni che gestiscono le chiamate REST che permettono la comunicazione tra il back-end ed il front-end, ossia tra broker e bots o sensori;

- **dynamo-db-repository.go**, in cui sono state sviluppate le funzioni che effettuano operazioni di lettura e scrittura nel database;

- **pubsub-system.go**, in cui viene implementato il pattern publish-subscribe e il broker e le operazioni eseguite da quest'ultimo.

http-api.go. All'interno del main func viene inizialmente definito un **router** facendo uso della libreria **mux**, con l'obbiettivo di gestire le richieste http in entrata nell'applicazione. Viene poi chiamata la go func *checkCli* per determinare se ci si trova in uno scenario d'esecuzione di tipo context-aware o meno. In seguito si controlla se nel database sono eventualmente presenti le tabelle citate nel paragrafo 1.1 attraverso *ExistingTables*; qualora non lo siano, le tabelle vengono create chiamando la funzione *createTables*. Tramite un'operazione di read sulla tabella "bots" in DynamoDB, vengono poi recuperati i bots dall'applicazione ed inseriti in un'apposita lista attraverso la funzione *checkDynamoBotsCache*. Con la go routine *checkResilience* si vanno a servire le richieste presenti nella tabella "resilience" del database, che corrispondo a quelle che non sono state servite in caso di un eventuale precedente crash del sistema.
Come accennato all'inizio di questo paragrafo http-api.go, l'applicazione gestisce le chiamate REST che riceve grazie al router definito precedentemnte. Il router va ad instradare le richieste che riceve in un **handler** con la funzione *HandleFunc* della libreria **server**; ogni handler definisce un path URL specifico ed è associato ad determinata go func che svolge uno specifico task. Ad esempio, l'istruzione:

router.HandleFunc("/unsubscribeBot", unsubscribeBot).Methods("POST")

indica che quando il router riceve una richiesta all'indirizzo IP della macchina, avente come url path "/unsubscribeBot", viene eseguita la go func "unsubscribeBot".
In particolare, sono state sviluppate le seguenti le seguenti funzioni per servire le richieste in arrivo dal front-end:

- **heartBeatMonitoring**: effettua il ping dell'applicazione;

- **spawnBot**: riceve in formato json l'id, il settore, il topic e l'ip address di un bot attraverso una richiesta http. Se il campo dell'id è vuoto, signfica che il bot è appena stato creato; se invece non lo è, significa che dal front end si è effettuata una resubscribe. Nel primo caso, l'id viene assegnato ad ogni bot in modo randomico ed univoco facendo utilizzo della libreria **shortuuid**. Il bot viene poi aggiungo alla lista di bots, inserito nell'apposita tabella del database con la funzione *AddDBBot* e poi viene chiamata la func *Subscribe*.

- **spawnSensor**: con le analoghe modalità in spawnBot, l'applicazione riceve l'id, il settore, il topic, il messaggio del sensore e un campo Pbrtx che indica se il messaggio del sensore è una ritrasmissione o meno. Se il campo id è vuoto, viene generato con la libreria shortuuid. Il campo **Pbrtx** serve per indicare al broker se deve riprocessare il messaggio oppure è già presente nel sistema in quanto ricevuto in precedenza. Se è impostato a true, significa che il messaggio è già stato trasmesso e quindi il broker deve semplicemente rimandare un ack al front-end, in accordo con la semantica di comunicazione At-Least-Once. Al contrario, se è impostato a false deve effettuare la routine di gestione dei messaggi, quindi lo inserisce nella lista "**SensorRequest**", contenente i messaggi da inoltrare ai

subscribers interessati. Infine, viene inviato l'ack al front-end.

- **unsubscribeBot**: annulla l'iscrizione di un bot dal topic andando ad eliminarlo dalla lista di bot nell'applicazione. Il bot non viene rimosso dalla tabella bots del database per un'eventuale resubscribe futura. Una volta eseguita l'operazione con successo, viene inviato l'ack al front-end.

Per impostare l'applicazione in perenne ascolto di richieste http, viene avviata una go routine tramite la funzione *ListenAndServe* della libreria **server** sulla porta 5000 sull'indirizzo del router creato inizialmente tramite la libreria mux. In questo modo, una richiesta che arriva dal front-end del sistema, che ha necessariamente uno tra gli url path gestiti dall'hadler del router HandleFunc, viene instradata nel path corretto, così da portare in esecuzione una delle go func sopra elencate. L'utilizzo di una go routine permette di eseguire quest'attività su un thread separato e concorrente. Infine, l'applicazione entra in loop infinito che controlla la lista delle richieste da parte dei publishers, quindi dei sensori: per ogni richiesta nella lista viene avviata una **goroutine** che effettua la pubblicazione del messaggio tramite la funzione *Publish* e che infine rimuove quella richiesta dalla lista sensorRequest.

pubsub-system.go. In questo file è sviluppato il vero e proprio **broker** e la **coda di messaggi** secondo il pattern publish-subscribe. In particolare, il broker è implementato come una **struct** avente come fields più significativi la mappa "**subscribersCtx**" (dichiarata come *map[key]BotSlice* dove key è una coppia *[Topic, Sector]*) che contiene informazioni sui subscribers, ovvero i bots, interessati ad un certo topic e che si utilizza quando l'applicazione viene eseguita **con contesto**; la mappa "**subscribers**" (dichiarata come *map[string]BotSlice* dove la chiave è la stringa del topic) che ha lo stesso utilizzo della precedente ma in uno scenario **senza contesto**; una lista di sensori chiamata" **sensorsRequest**" per poter accedere ai messaggi inviati dagli oggetti di tipo Sensor. Sono state quindi sviluppate le seguenti funzioni:

- **Subscribe (bot Bot)**: prima di ogni altra cosa, si controlla se si sta lavorando in modalità context aware o meno. Nel primo caso, si va a cercare in "subscribersCtx" la coppia [Topic, Sector] del bot di cui si vuole effettuare un'iscrizione, ovvero quello del parametro di input. Se tale coppia viene trovata, allora il bot viene inserito in append nella rispettiva slice avente la coppia come chiave; altrimenti, viene creata una slice di bots per tale chiave ed inserito il bot in input all'interno di essa. Nel caso senza contesto, le operazioni eseguite sono le stesse ma si ricerca il topic del bot nella mappa "subscribers".

- **Unsubscribe**: anche qui si distingue il caso con contesto da quello senza. Il meccanismo è opposto a ciò che avviene nella go func *Subscribe*: scorrendo la mappa dei subscribers, si rimuove il bot presente nella slice avente come chiave il topic del bot in input, o la coppia [topic,sector] nel caso context-aware.

- **Publish (sensor Sensor)**: La funzione riceve come parametro di input un sensore e anche qui si distingue il caso con contesto da quello senza. Nel primo caso, si cerca nella mappa "subscribersCtx" la slice di bots avente come chiave il topic ed il settore del sensore in input "sensor". Se vi sono bots iscritti a quel topic di tale settore, questi vengono messi in una slice di bots chiamata "**myBots**". In seguito, tramite la funzione *writeBotIdsAndMessage* vengono inseriti nella tabella "**resilience**" in DynamoDB l'id di ogni bot presente nella slice ed l'attuale messaggio del sensore, che il bot si aspetta di ricevere a causa della sua iscrizione. Dopo quest'operazione, **per ogni bot** della slice viene avviata una **go routine** *publishImplementation* che si occupa d'inviare il messaggio al bot stesso e da cui aspetta di riceverne l'ack. Le routines sono state sincronizzate tramite una **WaitGroup**. Una volta che tutte le routine hanno ricevuto gli ack da parte dei bots interessati a quel messaggio, attraverso la funzione *removePubRequest* vengono rimosse dalla tabella "sensorsRequest" le entries inserite precedentemente. Nel caso in cui ci si trova in uno scenario senza contesto, le attività svolte sono le stesse ma utilizzando la slice "subscribers".

- **PublishImplementation**: qui viene implementata la semantica di comunicazione **at-least-once**. La funzione riceve come parametri in input il **bot** a cui deve inviare il messaggio, il **sensore** che pubblica il messaggio ed il **WaitGroup**. Tramite la funzione *newRequest*, viene generata una nuova richiesta http per notificare il bot con il messaggio. Nel caso in cui il bot risponda con l'ack, dalla tabella "resilience" in DynamoDB viene eliminata l'entry relativa al bot, sensore e messaggio in questione. Se invece l'applicazione back-end non riceve l'ack da parte del bot a causa di un **timeout**, si riprova ad effettuare la trasmissione con le stesse modalità di prima.

dynamo-db-repository.go. In quest'ultimo file sono contenute tutte le go func per le operazioni di **scrittura** e **lettura** dal database, come per creare le tabelle, rimuovere, aggiungere e leggere entries da esse.

3.2. WBMQ-Controller - Front-End

Il front-end è stato sviluppato in modo da rappresentare i bots ed i sensori come delle richieste al sistema. Il modo in cui poi questi vanno ad interagire con applicazioni esterne, ad esempio i tools con cui vengono rilevate le misurazioni da parte dei sensori, o come potrebbero essere processati i dati da parte dei bots per un qualsivoglia utilizzo, dipende da un'eventuale futura implementazione dei bots e dei sensori; un esempio potrebbe essere la geolocalizzazione per individuare la posizione dei bots nei diversi settori.
Dal punto di vista implementativo, il front-end è stato realizzato come un'applicazione **multithread**, le cui funzionalità sono definite (come "def") all'interno di una classe chiamata "**class one**". Il main thread dell'applicazione quindi inizializza la classe e per eseguire ogni funzione di quest'ultima utilizza un thread separato. Il main della classe è dato dalla funzione "**main form**", in cui vengono definite ed inizializzate tutte le variabili globali della classe. Inoltre, viene effettuato anche il setup della **grafica** dell'applicazione, sviluppata con la libreria **Tkinter**. Come è possibile vedere al momento dell'esecuzione, la **GUI** è composta da una **console** per visualizzare lo **stato dei bots** nel sistema e le loro informazioni, e da una lista di **bottoni** che permettono d'**interagire** con il sistema stesso. I bottoni consentono all'utente di scegliere un'attività da compiere, che quindi viene comunicata al back-end e gestita tramite una delle go func implementate nel file *http-api.go*. Le operazioni presenti sul GUI si distinguono in quelle relative ai sensori e quelle ai bots. Le prime sono:
-"**Publish a new value**": per la creazione di un nuovo sensore e la pubblicazione di un nuovo valore;
- "**Publish random values**": va semplicemente ad effettuare

nel back-end una *publish* creando non un solo sensore ma molteplici (il numero viene inserito interattivamente dall'utente), appartenti a settori randomici tra quelli a disposizioni e con topic anch'esso randomico;
- "**Sensor killing frequencies**": per impostare la frequenza con cui i sensori simulano un crash, andando ad effettuare delle sleep più o meno lunghe a seconda della frequenza scelta dall'utente.

Le operazioni dei bots invece sono:
- "**Subscribe a new bot**" : per l'iscrizione di un bot ad un topic in un certo settore;
- "**Subscribe random bots**": ha lo stesso funzionamento di "publish random values" ma riguarda l'iscrizione di molteplici bots;
- "**Unsubscribe bot**" : dopo aver selezionato il bot dalla console, effettua l'annullamento dell'iscrizione di un bot al suo topic. Sulla console appare semplicemente che il bot è in stato di "waiting" per la ricezione di nuovi messaggi;
- "**Resubscribe bot**" : effettua nuovamente l'iscrizione di un bot al suo topic;
- "**Bot killing frequencies**" : ha lo stesso funzionamento di "Sensor killing frequencies" ma per i bots.

Per quanto riguarda le operazioni *Sensor killing frequencies* e *Bot killing frequencies*, si è deciso d'introdurre la possibilità d'inserire più bots o sensori contemporaneamente nel sistema, senza sceglierne topic o settore, per facilitare l'attività di testing. Infine, il thread grafico avvia una **pool di threads** di cardinalità pari a 100. Appena viene effettuato lo spawn della pool di threads, infatti, ad ognuno di essi è associata una routine ("**task**") che utilizza **Flask**. Quest'ultimo è un framework che viene utilizzato per realizzare una web-application in Python. Pertanto, ognuno di questi threads attende di ricevere richieste http dal broker che contengono i messaggi pubblicati dai sensori, e segnala al thread grafico di aggiornare i valori ricevuti sull'interfaccia della console.
Inoltre, quando l'utente clicca sul bottone *publish new value*, ad ogni sensore viene associato un thread (che esegue la funzione *runBackground*) che si occupa di mandare periodicamente richieste http al broker con dentro un nuovo valore del messaggio da pubblicare, cioè quello che simula la misurazione rilevata, andando a scegliere dei valori random compresi tra 25.5 e 50.9. Una volta effettuate queste operazioni, si attende l'ack.
Anche quando si clicca il bottone *subscribe new bot* viene creato un nuovo thread che esegue la routine *bot task background*, il quale svolge l'operazione di subscribe inviando un messaggio http al broker che contiene i dati del bot ed attende di ricevere l'ack da quest'ultimo.

4. Limitazioni

1) La **console** di monitoraggio delle **risorse** di **Elastic Beanstalk** (macchina EC2, load balancer...) ha fornito valori inconsistenti a volte, andando a renderne l'utilizzo leggermente difficoltoso;
2) Quando il sistema scala su un'altra macchina EC2, vi è una probabilità prossima allo zero ma comunque non nulla che l'applicazione generi un altro bot o sensore con un **id** già utilizzato;
3) Per quanto riguarda la configurazione di "**platform**" in Elastic Beanstalk, non è stata utilizzata l'opzione "64 bit amazon linux **2**" (ma quella "64 bit amazon linux") poiché sono stati rilevati e anche segnalati dei **bugs** che non permettevano di usare un docker container su quella macchina.
4) L'implementazione del Front-End basata sul multi-threading è ovviamente una limitazione importante in quanto come unicum gestisce il workload sia dei publisher sia dei subscriber, limitandoci a livello sia di rete sia che di gestione del TLS (Thread Local Storage). Si è deciso comunque di realizzare un **secondo Front-End** che espone le funzionalità base per interagire con il sistema e che può essere deployato e **scalato** dinamicamente allo stesso modo del sistema stesso. Riguardo la GUI è stato trovato un trade-off tra gestione delle performance e la corretta visualizzazione del funzionamento del sistema.

5. Piattaforme Software

Per la realizzazione del sistema sono stati utilizzati come IDE i servizi offerti da JetBrains di GoLand e PyCharm, oltre a VisualStudio per lo sviluppo del sorgente in Python. I servizi AWS utilizzati invece sono DynamoDB come database, ElasticBeanstalk per la gestione dell'applicazione nel cloud AWS. Di seguito si riporta un elenco delle librerie utilizzate.

Go. - **dynamodb**: per usufruire del servizio Amazon DynamoDB.
- **mux**: per implementare il router e le HandleFunc.
- **shortuuid**: per generare numeri random.
- **aws session**: per creare nuova sessione con cui utilizzare i servizi di aws.
- **dynamodbattribute**: per utilizzare l'API di dynamo DB.

Python. - **flask**: framework per la creazione di una web application.

6. Testing

Sono riportati in questa sezione i risultati dell'attività di testing ottenuti secondo tre modalità definite:

1) Death Frequency Standard (DFS). Bots e sensori condividono **una distribuzione di probabilità** centrata su un valore di sleep pari ad 1 secondo. Con una probabilità dell'1% simulano un rallentamento dovuto ad una possibile disconnessione e/o riavvio del dispositivo.

2) Death Frequency Slow Sensors (DFSS). In questo caso bots e sensori sono associati a **due distribuzioni diverse**. In particolare, i bots tendono a simulare un funzionamento corretto nel 67% delle volte mentre per i sensori sarà molto più probabile simulare una disconnessione o un rallentamento maggiore o uguale a due secondi.

3) Death Frequency Long Sensors (DFLS). E' stato creato un **nuovo frontend** esclusivamente per la fase di test, con le stesse funzionalità di quello originario ma **senza grafica**, per poter testare i limiti dell'applicativo in modo da dedicare tutte le risorse allo scambio di messaggi. Così come nel front-end originario, vi sono 100 threads che **simulano** ognuno il comportamento di un **subscriber**: quando arriva un messaggio dal broker con il valore pubblicato, poiché in questa nuova versione non sono presenti i bots a causa dell'assenza della grafica, è responsabilità dei threads rispondere ai messaggi andando ad inviare un ack. A differenza del frontend originario,

tuttavia, ogni thread può servire una richiesta di un qualsiasi bot. Allo stesso modo vi sono anche altri threads che invece simulano il comportamento dei sensori, implementati secondo **gruppi di publishers** che pubblicano nuovi valori ogni 10 secondi. Viene aggiunto un nuovo gruppo ogni 5 minuti, pertanto nell'istante iniziale in cui avviene l'esecuzione si hanno 10 publishers che pubblicano per 5 minuti ogni 10 secondi l'uno. Parallelamente alla loro attività, al quinto minuto viene introdotto un ulteriore gruppo di 10 publishers che pubblicheranno a loro volta ogni 10 secondi, e così via. Per rendere più chiaro il funzionamento di questo nuovo front-end si potrebbe fare un paragone con una **classe di test**, ad esempio come avviene in Java utilizzando il framework JUnit: in quel caso verrebbe realizzata una classe contenente esclusivamente i casi di test che servono a testare le funzionalità di una parte del sistema. Il front-end secondario ha quindi una funzionalità analoga, e, contrariamente a quello precedente, non permette che sia l'utente ad interagire con il sistema stesso in quanto ha un flusso d'esecuzione già pre impostato. I grafici relativi a quest'ultima modalità sono quelli annotati con **G5**.

I test marcati come DFS e DFSS sono stati realizzati seguendo una politica di scaling con i seguenti parametri configurabili dalla console di monitoraggio dell'applicazione di cui è stato effettuato il deploy:

- **Tipo d'istanza**: t2.micro
- **Metrica**: Latenza
- **Statistica**: Media
- **Unità**: Secondi
- **Periodo di valutazione**: 1 minuto
- **Periodo di breach**: 5 minuti
- **Soglia di scale-out**: 2 secondi con incremento di 1 macchina
- **Soglia di scale-in**: 0.5 secondi con decremento di 1 macchina

I test marcati con DFLS seguono una politica di scaling cosi fatta:

- **Tipo d'istanza**: t2.micro
- **Metrica**: Request Count
- **Statistica**: Somma
- **Unità**: Conteggio
- **Periodo di valutazione**: 1 minuto
- **Periodo di breach**: 1 minuti
- **Soglia di scale-out**: 500 requests con incremento di 1 macchina
- **Soglia di scale-in**: 200 requests con decremento di 1 macchina

Queste politiche di scaling sono il risultato ottenuto da attente valutazioni svolte con il tempo. Non essendo particolarmente precise le indicazioni fornite dalla documentazione AWS, in modo del tutto sperimentale il sistema è stato inizialmente valutato tramite la metrica di **CPU-Utilization**. Questa metrica si è rilevata troppo stringente nel caso di quest'applicazione che ha un periodo di startup assolutamente non trascurabile. La seconda metrica presa in considerazione è stata **Latency**,con la quale sono stati effettuati e riportati alcuni casi di test.

E' importante sottolineare il fatto che, a livello prestazionale, la differenza principale tra "**con contesto**" e "**senza contesto**" è che nel primo caso, quando il sistema deve processare le richieste, deve tenere conto sia del settore (contesto geografico) sia del topic; nel secondo invece si tiene conto **esclusivamente del topic**. Vi sarà quindi una differenza sostanziale nel **numero di messaggi** inoltrati ai subscribers, in quanto in uno scenario senza contesto il broker dovrà occuparsi d'inviare i messaggi a tutti i singoli bots iscritti ad un determinato topic, senza discriminarli per il settore a cui appartengono.
Questa differenza fondamentale viene riscontrata in ognuna delle seguenti tabelle riportate.

(1) DFS without Context

Bots	Sensors	Host	CPU-U	Latency	Requests
10	10	1	54,9	48,3	1786
20	20	1	95,4	55,4	2900
30	30	1	98,1	48,3	16100
40	40	1	99,7	41	21800

(2) DFS with Context

Bots	Sensors	Host	CPU-U	Latency	Requests
10	10	1	99.8	31.7	1200
20	20	1	99.9	32.5	2600
30	30	1	99.7	55.6	14200
40	40	1	9.8	139.1	16600
50	50	1	9.9	138.7	26100
60	60	2	24.4	250.4	27100

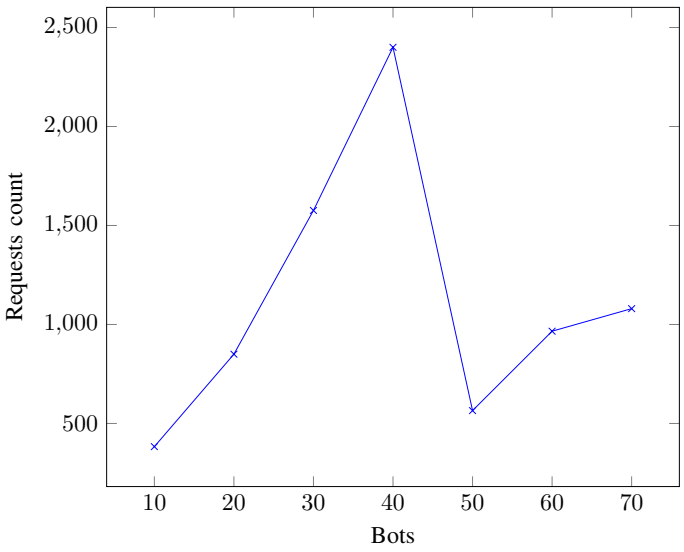
Nelle tabelle **(1)** e **(2)** sono riportati i dati estrapolati dalle attività di test con modalità **DFS**, ovvero con una frequenza di morte **standard** dei sensori e dei bots. Questa modalità, rispetto alla DFSS ha appunto una frequenza molto più elevata, e questo fa sì che il sistema subisca un crash in tempi brevi, a causa di una **crescita delle richieste** molto più **rapida**.
In tabella 1 il frontend subisce un crash a **40 bots**, arrivando ad un numero di richieste pari a **21800** (più di 12 volte tanto quelle relative a 10 bots). In tabella 2, invece, il sistema arriva a sostenere fino a **60 bots**, ovvero 1/3 in più dei precedenti, ma con **27100** richieste, un dato basso in proporzione all'aumento del numero di bots. Infatti, quando si hanno in **entrambi i casi 40 bots**, in tabella 1 si hanno **21800** richieste, mentre in tabella 2 sono **16600**, circa **5000 in meno**. Questo dato è molto rappresentativo per evidenziare la differenza di workload smaltita dal sistema a seconda se si è in uno scenario con contesto o meno.

(3) DFSS without Context

Bots	Sensors	Host	CPU-U	Latency	Requests
10	10	1	96,4	19,6	383
20	20	1	36,8	154	850
30	30	1	9,9	672,9	1576
40	40	2	12,9	759,7	2400
50	50	3	30,9	1500	565
60	60	3	68,7	238	966
70	70	2	62.8	202,8	1080

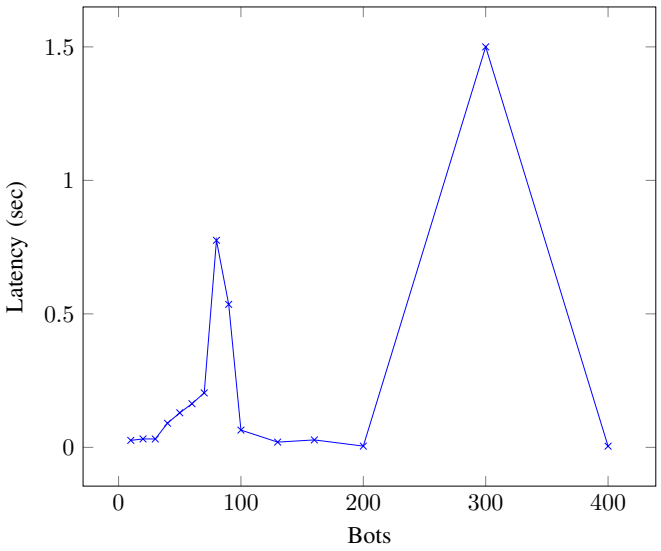
(4) DFSS with Context					
Bots	Sensors	Host	CPU-U	Latency	Requests
10	10	1	94	26,2	127
20	20	1	95,9	31,6	482
30	30	1	96,5	31,4	872
40	40	1	87,7	90,4	1700
50	50	1	78,7	129,4	2600
60	60	1	71,5	163,5	3900
70	70	1	66	204	5200
80	80	1	60.9	775,5	7800
90	90	2	9,8	535,5	13200
100	100	1	48,9	64,7	3100
130	130	1	99,7	19,7	4200
160	160	1	99,6	27,9	4700
200	200	2	98	4,7	5100
300	300	1	87	1500	1700
400	400	2	94	4,4	2890

(G3) Requests/Bots relationship

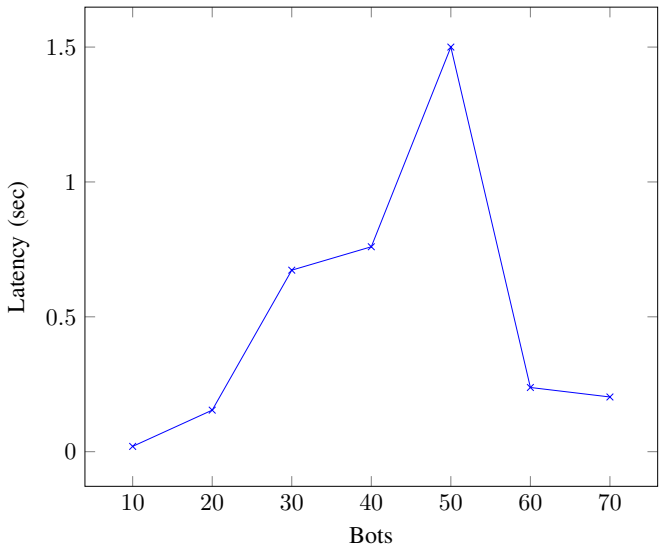


I grafici (**G3**) e (**G4**) sono relativi rispettivamente alle tabelle (**3**) e (**4**), che riportano i dati relativi ai test in modalità **DFSS**. A differenza di prima, la **frequenza di morte** dei **sensori** è molto più **alta**, quindi si avranno delle **pubblicazioni** molto **meno frequenti**. In entrambe le coppie di grafici sono presenti sulle ascisse i bots e sulle ordinate una volta la latenza e una il numero di richieste.

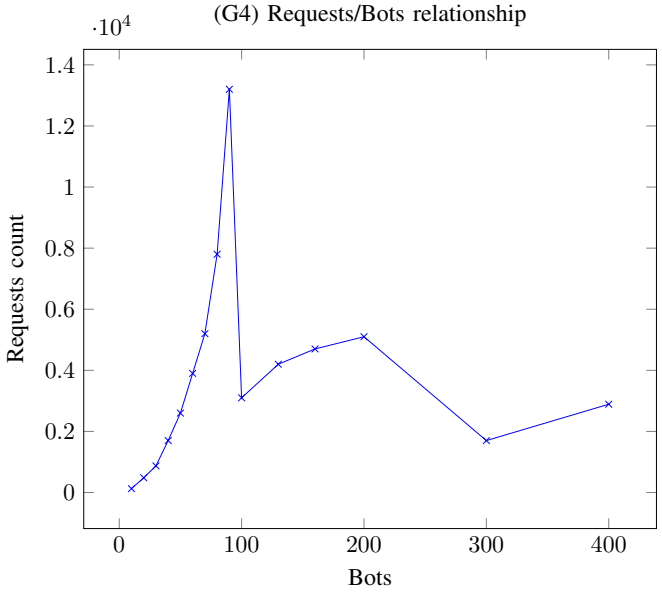
(G4) Latency/Bots relationship



(G3) Latency/Bots relationship



(G4) Requests/Bots relationship



Andando ad analizzare la coppia di grafici **G3**, si può vedere come l'andamento dei grafici sia coerente con ciò che accade nel sistema, che si può evincere dalla tabella 3: con l'aumentare del numero di bots, aumentano sia la latenza che il numero di richieste fin quando si arriva a **40 bots**: a questo punto, come riportato anche nella tabella 3, viene effettuato uno **scale-out** e quindi vengono utilizzate due macchine per la computazione. Lo scaling comporta quindi un **numero di richieste più basso all'aumentare dei bots**, perché il workload è distribuito tra più istanze EC2 e mentre viene effettuato lo startup della seconda macchina le richieste in entrata vengono gestite dal **load balancer**. Al contrario, la **latenza continua ad aumentare** fino a 1.5 secondi quando si arriva ai 50 bots. Questo significa che lo scale è avvenuto nel momento in cui si aveva un solo host, altrimenti la latenza sarebbe diminuita: questa decrescita non è immediata perché vi è un lasso di tempo in cui viene effettuato lo **startup** della seconda macchina e del suo ambiente, mentre la prima macchina continua ad accettare le richieste. Solo dopo fatto ciò inizia a rispondere anche la seconda istanza di EC2 e così la latenza si abbassa.
Inoltre, avviene nuovamente uno scaling quando si raggiungono 50 bots: la latenza qui ha una curva che continua a decrescere fino alla fine. Il numero di richieste invece ricomincia a crescere a partire da 50 perché i bots aumentano ed il numero di host resta costante; da 60 in poi invece si ha un trend comunque crescente ma con meno velocità.

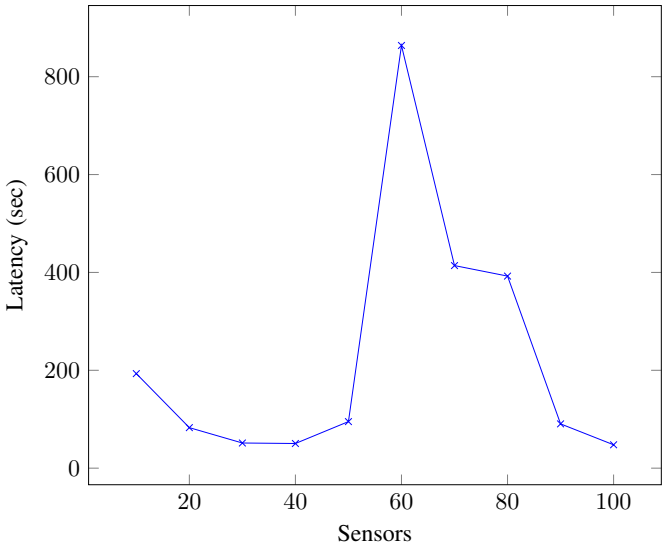
La coppia di grafici **G4** invece rappresenta l'esecuzione con contesto, infatti si arriva a 400 bots contro i 70 del caso precedente: all'aumentare dei bots crescono sia la latenza che il numero di richieste fino a **90 bots**, quando si ha uno **scale-out**. Da qui entrambi i parametri decrescono fino a 100 bots, dove avviene uno scale-in, da cui poi le richieste riprendono a salire e la latenza alterna brevi andamenti crescenti e decescenti, anche se con un trend meno inclinato rispetto a prima: ciò avviene per le stesse ragioni esposte nel paragrafo precedente dove vi è l'analisi dei grafici G3. A 200 bots avviene di nuovo uno scale out, e ciò comporta un picco di crescita molto ripido della latenza, fino a 1.5 secondi, che poi diminuisce quando a 300 bots si passa nuovamente ad una sola istanza d'esecuzione. Per quanto riguarda il numero di richieste, invece, si ha invece prima una decrescita fino a 300 bots perché vengono smaltite da due macchine, e poi aumentano da 300 in poi in concomitanza con lo scale-in.

Considerando infine l'ultima metrica **Request Count** usata per le valutazione dei test marcati con **DFLS**.

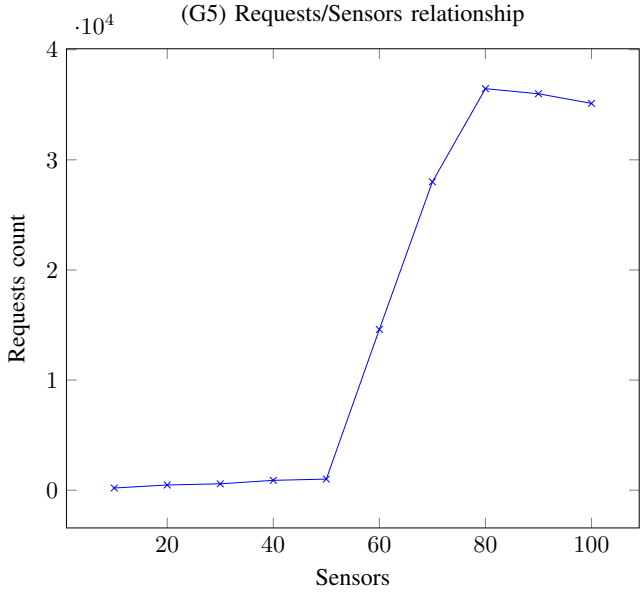
(5) DFLS with Context - Fixed Bots

Bots	Sensors	Host	CPU-U	Latency	Requests
100	10	1	99.6	193,2	199
100	20	1	99,6	82,8	473
100	30	1	99,7	51,4	580
100	40	1	99,5	50,4	894
100	50	1	95,6	95,15	1008
100	60	3	64,5	863,8	14600
100	70	5	48,8	414	28000
100	80	7	82,6	329,5	36460
100	90	9	79,6	90,5	36000
100	100	11	89,7	47,75	35125

(G5) Latency/Sensors relationship

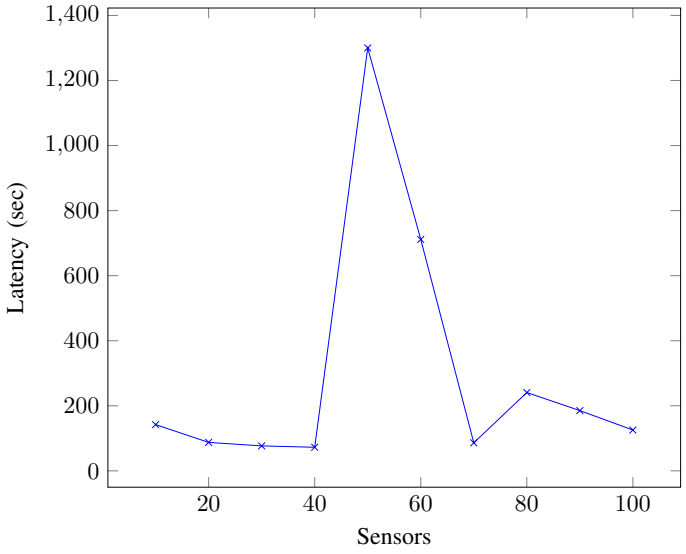


(G5) Requests/Sensors relationship

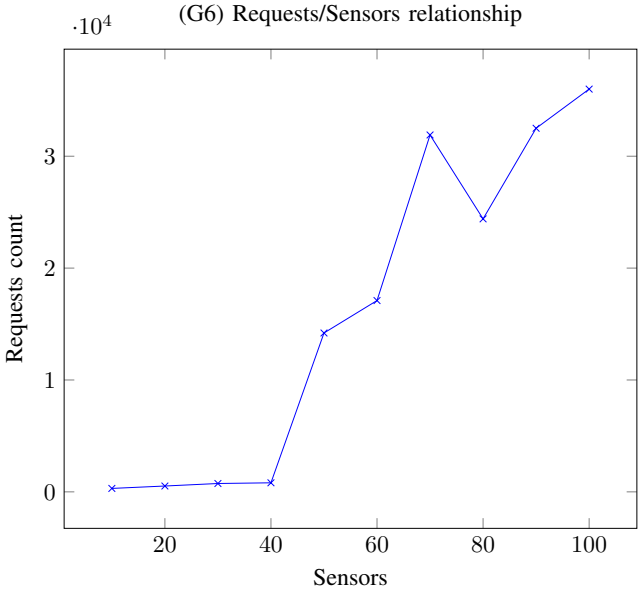


(6) DFLS without Context - Fixed Bots					
Bots	Sensors	Host	CPU-U	Latency	Requests
100	10	1	99,8	142,3	307
100	20	1	99,8	87,5	522
100	30	1	97,9	76,7	747
100	40	1	99,0	72,5	809
100	50	1	76,9	1300	14200
100	60	4	44,9	711,4	17100
100	70	6	60,6	86,2	31900
100	80	8	92,6	240,4	24400
100	90	12	83,5	185,2	35200
100	100	14	92,2	125,5	36000

(G6) Latency/Sensors relationship



(G6) Requests/Sensors relationship



Le considerazioni riguardandi l'utlima coppia di grafici (**G6**) sono riportate nella sezione conclusiva per maggiore chiarezza.

7. Conclusione

Di seguito si riportano tutte le spiegazioni, osservazioni e commenti relativi a tutti i grafici delle 3 diverse modalità di testing effettuate.
In merito alla **DFSS**:
Questo è stato il secondo approccio al testing del sistema e si è ritenuto interessante utilizzare come metrica di scaling la **latenza**, in quanto si voleva garantire all'utente un tempo di servizio non superiore ai 2 secondi per request. E' importante notare che questa scelta si è rivelata fallace, in quanto prima che il sistema raggiungesse la soglia necessaria allo scaling, cioè 2 secondi di latenza, questo già presentava profondi **rallentamenti** che avrebbero a breve portato all'impossibilità di un processamento continuo delle richieste da parte del sistema. Anche la **statistica media** non si è rivelata ottimale, in quanto eventuali periodi di picco isolati non venivano catturati dal sistema poiché non eccedevano la media. A peggiorare le performance ha contribuito anche il **periodo di breach** di 5 minuti, che causava al sistema lunghi periodi di lavoro in sovraccarico, mentre il tempo di cooldown di 6 minuti ha profondamente rallentato l'instanziazione di nuove macchine. La concomitanza di questi tre fattori ha reso impossibile quindi lo scaling dell'applicativo in tempi utili a prevenirne il crash, come testimoniato dalla **scarsa variazione del numero di hosts**. Pertanto si può affermare che questo ha portato ad un'**inconsistenza** dei valori misurati, dovuta ai continui sovraccarichi del sistema mal gestiti. Inoltre, come visto in (**G3**) e (**G4**), i comportamenti agli antipodi presentati dai parametri "latenza" e "request", rispettivamente i picchi di massimo e di minimo, che sono stati registrati in presenza dello stesso numero di bots, vanno proprio ad indicare l'incapacità nella gestione del sovraccarico. Al contrario, invece, in **DFLS** i picchi sono concordi a testimoniare l'istante di forte stress del sistema, che però mantiene il servizio attivo grazie ad uno scaling molto più tempestivo ed efficiente. La configurazione presentata in questi grafici non fa più affidamento sulla latenza, in quanto si è notato che anche nei periodi più critici per il sistema il massimo valore raggiunto erano le poche unità di secondi. La configurazione di scaling in questi test infatti prevede l'uso della metrica **Request-Count** secondo la statistica di somma, cosi da monitorare il massimo numero di richieste presenti nel sistema in quell'istante di tempo. In concomitanza a questa vengono utilizzati un periodo di breach di 1 minuto, che limita il tempo di utilizzo del sistema fuori dalle condizioni pre-impostate ed un periodo di cooldown di 90 secondi, che rende immediata e tempestiva l'istanziazione dei nuovi hosts. Inoltre, la soglia di scale out è stata impostata a 500 requests, benchè il sistema potesse gestirne di più per prevenire periodi di rallentamento e smaltire quelli eventualmente già presenti. La politica di scaling riportata in questo ultimo caso di test risulta essere molto aggressiva, ma è servita per capire i limiti di scaling del sistema in presenza di test mirati a stressarne l'utilizzo (fino a 600 nuove requests per minuto). Sia in caso di contesto che non il sistema ha sperimentato al massimo dei rallentamenti che però sono sempre stati evasi con l'utilizzo di istanze aggiuntive. Sarebbe stato interessante esaminare il comportamento del sistema in presenza di un'altra politica di scaling altrettanto aggressiva ottenuta rilassando i parametri sopracitati, ad eccezione del numero di requests per minuto che sarebbe dovuto diminuire drasticamente, eventualmente in una futura implementazione. Si conclude evidenziando le differenze tra la presenza e l'assenza del contesto. Nel caso dei test **DFSS** la presenza del contesto ha permesso un'introduzione molto superiore della quantità di bots e sensori nel sistema. Tuttavia, si ricorda che questi valori sono molto instabili. Nel caso dei test **DFLS**, raggiunti 70 bots si ha avuto un **rallentamento** del sistema che ha portato ad un accumulo delle requests, per cui il sistema ha assunto comportamenti analoghi sia in presenza che in assenza di contesto ad eccezione del parametro "traffico di rete" (non riportato nelle tabelle), che in presenza di contesto era di **qualche unità di MB**, mentre in assenza era di diverse

decine di MB, ad evidenziare un palese carico di pacchetti sul network. Tale parametro è rimasto indicativo anche sotto ai 70 sensori, dove però la differenza nella gestione del sistema con e senza contesto viene evidenziata anche dagli attributi ”latenza”, ”request” e ”numero di hosts”, tutti riportanti **valori maggiori** nel caso **senza contesto**. Si può dunque affermare che, in qualsiasi scenario, una politica senza contesto presenta un **risparmio in termini di operazioni** da compiere per il broker (ricerche più rapide), ma presenta **costi superiori** in termini di numero di messaggi da inoltrare e quindi traffico di rete. D'altro canto, la presenza del contesto si traduce in un maggior **numero di azioni** da compiere da parte del broker per il forwarding del singolo messaggio (ricerche a grana più fine), ma selezionando in modo più preciso i destinatari dei messaggi ha guadagni prestazionali in termini di **minor numero di messaggi** inviati a cui fa seguito un minor traffico di rete.

References

[1] libreria ”dynamodb”: ”github.com/aws/aws-sdk-go/service/dynamodb”

[2] libreria ”mux”: ”github.com/gorilla/mux”

[3] libreria ”shortuuid: ”github.com/lithammer/shortuuid”