**Carlo Francesco Pellegrino**

O55000378

# Relazione Progetto APL

## Proposta

La proposta concordata per il progetto di APL prevedeva:

1. Implementazione di un modello di DHT in C++ per bilanciare il carico di un MQTT Broker scritto in C# per un progetto precedente, il quale effettua il publishing dei messaggi sulla base della locazione (latitudine longitudine) dei dispositivi sottoscritti.
2. Data Analysis in R per delle statistiche sull'utilizzo dell'applicazione Android scritta in Java per un progetto precedente.
3. Porting dell'applicazione Android da Java a Scala.

## Revisioni alla proposta

Dopo aver concordato ciò che doveva essere sviluppato per questo progetto, si sono incontrate alcune difficoltà che vengono qui riportate:

- Scala Build Toolt (sbt) nella versione utilizzata per lo sviluppo Android (0.13.16) incontra numerosi problemi con il JDK di default dei sistemi Linux: richiede necessariamente quello fornito da Oracle.
- Scala Build Tool non supporta le librerie AndroidX utilizzate per sviluppare l'applicativo in Android.
- Tutti gli IDE che vengono comunemente utilizzati per lo sviluppo su Android (IntelliJ, Android-Studio) hanno smesso dal 2017 di supportare lo sviluppo di applicazioni Scala su Android.

Oltre a problemi di natura tecnica, sono sorte delle esigenze per poter simulare l'applicazione e osservarne il funzionamento in una demo. Inoltre il precedente utilizzo di Apache Kafka rendeva pesante l'ambiente (dovendo effettuare in precedenza il deploy di ZooKeeper, di un connettore personalizzato tra Kafka e MQTT e di un altro connettore tra Kafka e MongoDB), aprendo lo scenario per la scrittura di un componente in Scala che utilizzasse il modello ad attori e che eseguisse questa stream analysis in maniera personalizzata.

Per fare ciò sono stati aggiunti i seguenti componenti:

1. **MQTT to Mongo Writer:** Un componente scritto in Scala con l'ausilio della libreria Akka, ed in particolare il suo modello ad attori, di cui effettuare il deploy su Alpine Linux per avere un container leggero che non consumi troppe risorse in una macchina Amazon Educate.
2. **Bot Manager:** Un componente scritto in Go per utilizzare la libreria docker/client, molto utilizzata nell'ambito dei container per gestirli comunicando con il DockerD.
3. **Bot:** Un componente scritto in Go che funge da Client per il Broker MQTT e interagisce con esso come fosse un normale utente. Questo può generare dati finti relativi a temperatura, sensore magnetico di una finestra e immagini.

## Sorgenti

I codici sorgenti consegnati con questa relazione sono quelli che sono stati sviluppati per il progetto. Il progetto completo, invece, può essere trovato al seguente link.

Inoltre la documentazione mira a fornire una descrizione funzionale ed introduttiva, ma per ulteriori informazioni sui singoli componenti, si rimanda al repository git, di cui sarà disponibile il link in ogni paragrafo.

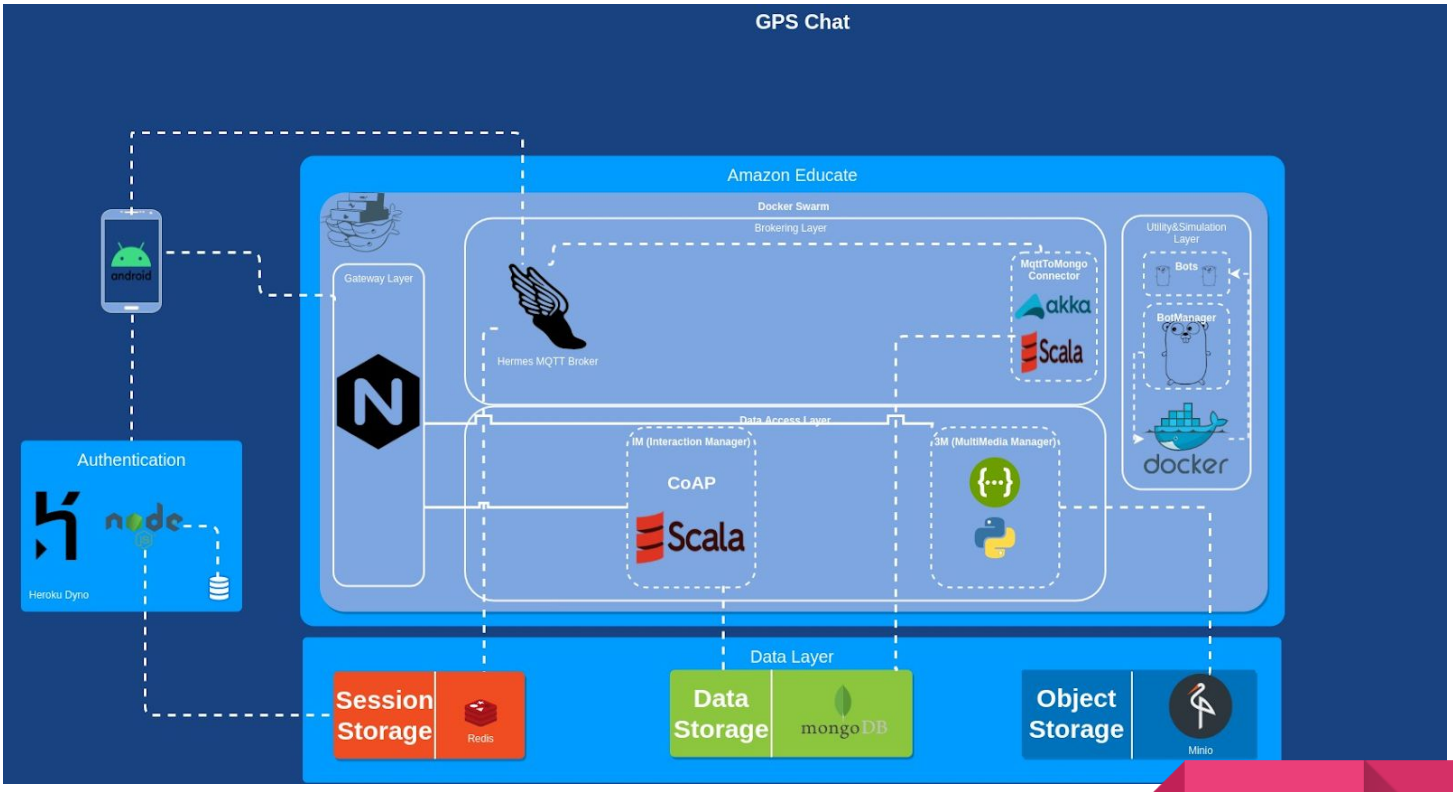
## Utilizzo

Il deploy di questo progetto è stato fatto su delle macchine Amazon Educate negli Stati Uniti con 1 GB di RAM e 1 vCPU. Per questi requisiti è stato scelto di effettuare il deploy tramite Docker Swarm e non Kubernetes. L'applicazione Android non è disponibile sul Play Store per essere utilizzata, tuttavia, se si possiedono delle Chiavi Google Maps si possono scrivere in un file xml e avviare l'applicazione con Android Studio.

Eccezion fatta per l'applicazione Android, tutti i componenti possono essere avviati in maniera separata tramite l'uso di Docker e delle immagini pubbliche (per ulteriori informazioni, consultare i vari README.md dei file).

## Architettura

Nella figura sottostante, che può essere trovata nel file [README.md] del progetto, è presente l'architettura complessiva. Il componente che si occupa del load balancing del Broker MQTT



(DHT scritte in C++) non è riportato perché nell'idea iniziale sarebbe stato embedded nel codice C# del Broker MQTT.

Di seguito vengono riportati i singoli componenti con le loro responsabilità e le motivazioni dei linguaggi di programmazione scelti e delle scelte implementative che sono state fatte.

## C++

Si è scelto di implementare le DHT in C++ perché si preferiva utilizzare un linguaggio di basso livello per interagire direttamente con le Socket e in particolare con le Socket UDP.

Il deploy di questo componente è stato fatto infine su container, scegliendo di inserire un numero di repliche pari a quello dei Broker MQTT: in questo modo e con **constraints** sui deployment si ottiene che ogni DHT sia nella stessa macchina di un Broker MQTT.

L'implementazione delle DHT è ispirata al protocollo Chord, senza implementare la parte relativa al ripristino della consistenza della visione dell'anello da parte di ogni Peer dopo una operazione di Join completata.

Ogni nodo ha un proprio ID, generato univocamente tramite l'uso di Poco/UUID (Universal Unique ID) con cui si fa conoscere nella rete. Quando viene inviata una comunicazione verso un altro Peer della rete, si invia anche il proprio **GeoHash**.

Per ottenere questo, anzitutto si utilizza una API IP-API, la quale tramite l'uso di un protocollo STUN a qualche livello, ottiene informazioni approssimative circa la latitudine e la longitudine di un Host.

Con queste informazioni invia una Query presso GeoHash.org, endpoint presso il quale, una volta forniti latitudine e longitudine si ricava il proprio GeoHash, con un livello di precisione personalizzabile tramite il query parameter *maxlen*. Quest'ultimo parametro si dovrebbe scegliere sulla base del numero di Peer appartenenti alla DHT, ma al momento non si è effettuato questa miglioria.

### Join

Per ogni messaggio di Join ricevuto, il Peer controlla se vi sono state altre richieste di Join in precedenza senza che esso abbia notificato agli altri peer della rete dell'aggiunta di un altro nodo (parametro booleano *ringUpdated*): se questo parametro risulta vero, allora il Peer

risponde al Client tramite uno status *NOT OK (50)*, altrimenti il Peer procede all'aggiunta del Client nella rete proprio come in Chord.

Viene aggiunto il Client come predecessore del Peer ricevente e viene inviata una risposta *OK (20)* presso il Client. Quest'ultimo aggiunge il Peer che lo ha fatto entrare nell'anello come successore e si mette in ascolto di richieste FIND o JOIN.

### Find

Ogni Peer tiene in memoria una mappa di Peer, sulla base della quale risponde alle richieste che arrivano sulla sua porta (default 4242): in particolare per ogni *FIND* message ricevuto, il Peer ricava il GeoHash dell'indirizzo da cui è stato contattato e prende dalla memoria un Peer di quelli che dovrebbero essere più vicino al Client, restituendone l'indirizzo IP presso quest'ultimo.

### Librerie

Per velocizzare l'implementazione si è utilizzata la libreria PoCo (Portable Components), una libreria statica che viene installata direttamente sul sistema host.

Inoltre si è utilizzata la libreria (header-only) SPeeDLog per effettuare il logging delle informazioni di debug e non che ogni Peer riporta.

# Go

Si è scelto di utilizzare Go come linguaggio di programmazione per il BotManager dato che le librerie per l'interazione con DockerD più complete sono scritte in questo linguaggio; la semplicità con cui scrivere un HTTP Server lo rende una buona scelta anche per i Bot che devono ascoltare la ricezione di comandi via HTTP.

## BotManager

Questo componente ha il compito di ascoltare le richieste HTTP in arrivo, di interpretarle utilizzandone i dati contenuti o per creare/distruggere Bot (avviandoli tramite docker client) oppure nel caso contengano un BotID (UUID che identifica univocamente ogni Bot), passarli al Bot indirizzato tramite HTTP per istruirlo su comandi da eseguire.

In particolare questo componente, una volta avviato crea un Controller che effettua la creazione di una goroutine, attendendone la sua terminazione tramite WaitGroup. Questa goroutine avvia il server HTTP sulla porta (default 9450) e resta in attesa di comandi, come soprascritto.

## Bot

Questo componente ha il compito di sottoscriversi al topic MQTT che gli viene fornito tramite CL, e di pubblicare i risultati dei comandi (periodici e non) che riceve tramite un altro topic fornitogli sempre da CL.

Una volta avviato, crea un Controller che attende due goroutine tramite WaitGroup. La prima è relativa ad un Server HTTP avviato per ascoltare eventuali comandi provenienti o dal BotManager o da client della rete; la seconda invece schedula periodicamente l'esecuzione di ogni comando che viene mandato al Bot con un parametro *Interval*.

Al momento attuale, i messaggi ricevuti dal topic cui il Bot si sottoscrive vengono solo loggati a schermo.

## Librerie

### BotManager

Le librerie utilizzate per il BotManager sono:

- docker/client per la comunicazione con il DockerD, per la gestione dei container (tramite i quali vengono avviati i Bot)
- kubernetes/klog per il logging delle informazioni a schermo.
- google/uuid per la generazione di ID univoci per l'identificazione dei Bot da parte dei client che istruiscono questo componente per la creazione di Bot.

### Bot

- kubernetes/klog per il logging delle informazioni a schermo.
- paho.mqtt per la connessione presso il broker MQTT sia per la pubblicazione che per la ricezione dei messaggi.
- google/uuid per la generazione di ID univoci per i messaggi inviati presso il Broker MQTT.

# Scala

Si è scelto di utilizzare Scala come linguaggio di programmazione per la possibilità di parallelizzare il lavoro in maniera efficiente tramite l'uso del modello di concorrenza ad attori. In particolare si è scelto di utilizzare il modello di concorrenza di Akka poiché grazie al modulo Akka Cluster vi è la possibilità di distribuire il carico di lavoro semplicemente connettendo più nodi e scalando il numero di attori.

In questo modo, aumentando il numero di Broker MQTT nel sistema, basta incrementare il numero di attori che fungono da *subscriber* per scalare il carico di lavoro di ogni nodo.

Vi sono quattro attori per questo componente:

1. **Main Guardian**: l'attore che effettua lo *spawn* di tutti gli altri e che ne gestisce il ciclo di vita ed eventuali failure.
2. **MQTTActor**: l'attore che si sottoscrive al topic che gli viene passato come parametro nel costruttore e gestisce la sottoscrizione e la riconnessione presso il Broker MQTT. Di ogni messaggio ricevuto, viene passato al MsgAnalyzer il payload come stringa.
3. **MsgAnalyzer**: l'attore che riceve i messaggi dal precedente MQTTActor e li passa al MongoWriter dopo aver effettuato una computazione. Al momento la computazione consiste nell'effettuare l'unmarshalling da stringa a JSON tramite la libreria Jackson.
4. **MongoWriter**: l'attore che riceve il dato dal MsgAnalyzer e lo scrive su MongoDB. Non è stato implementato un meccanismo di caching perché si è scelto di utilizzare un Atlas Cluster (soluzione managed per hosting MongoDB)

## Librerie

Le librerie utilizzate sono:

1. FasterXML.Jackson per la manipolazione di oggetti JSON.
2. Scala Mongo Driver per la connessione con MongoDB.
3. Paho MQTT per la connessione con il broker MQTT.

# R

Si è scelto di utilizzare R come linguaggio di implementazione di questo script perché risulta essere molto completo per ciò che riguarda l'analisi dei dati.

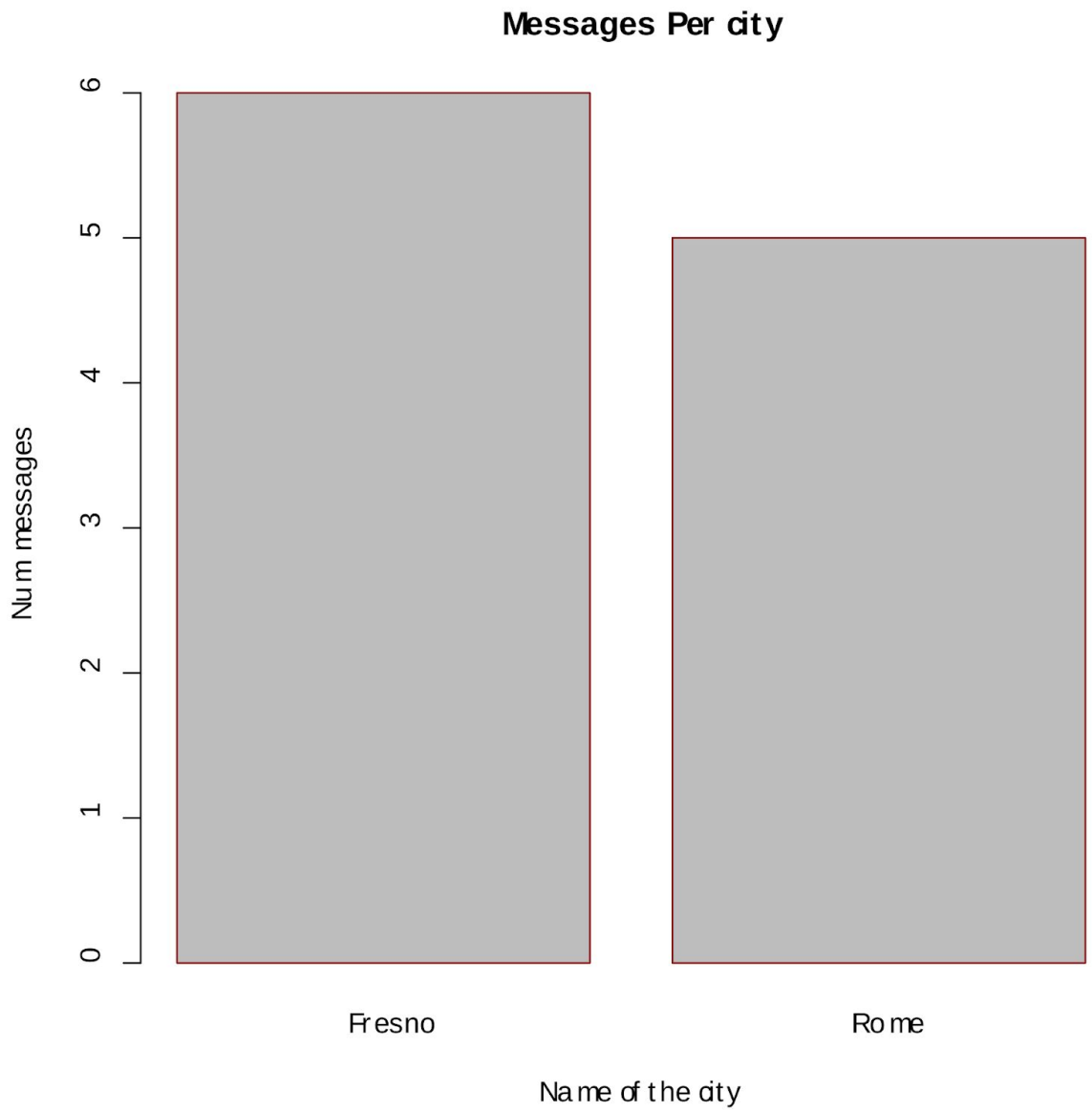
Lo script sviuppato Analyze.r contiene inizialmente come commenti i comandi da utilizzare per installare le librerie necessarie.

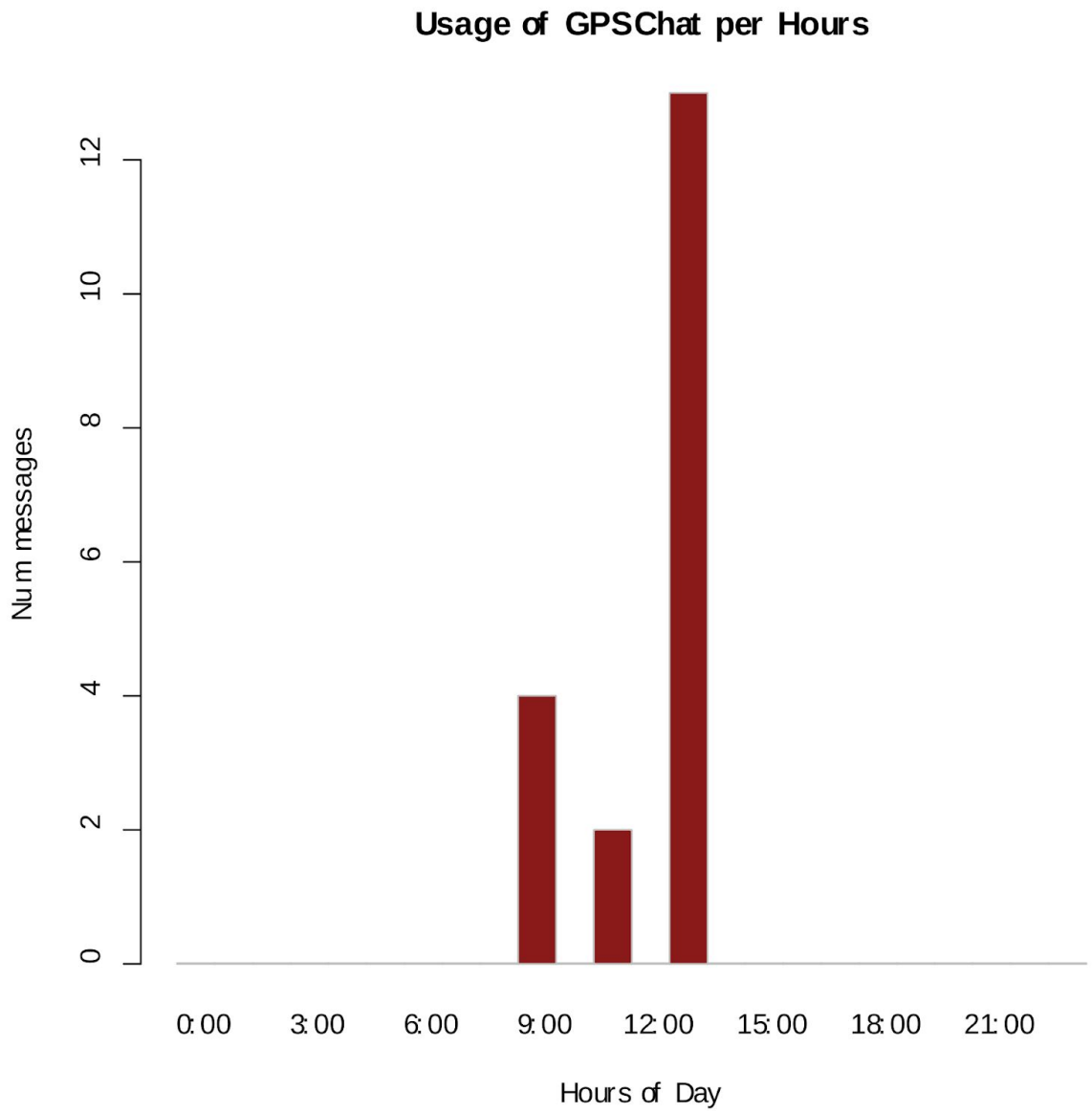
Si è scelto di effettuare come analisi, anzitutto il numero totale di  messaggi di ogni utente per osservare quali sono gli utenti più attivi, successivamente quali sono le fascie orarie in cui gli utenti sono più attivi, e infine quali sono le zone del mondo in cui l'applicazione viene più utilizzata. Nelle pagine a seguire sono riportati alcuni dei grafici tracciati: per completezza si rimanda al repository.

## Librerie

Le librerie utilizzate sono:

1. MongoLite per la connessione al database MongoDB per ottenere i BSON relativi ai messaggi inviati tramite l'applicazione.
2. HTTR per effettuare chiamate GET presso le OpenStreetMaps API per il Reverse Geocoding.
3. Stringr per la semplificazione dei timestamp inviati dai client dell'applicazione per le analisi di dati che vengono fatte dallo script.


Messages Per city
Num messages
0
1
2
3
4
5
6
Fresno
Rome
Name of the city


Usage of GPSChat per Hours
Num messages
0
2
4
6
8
10
12
0:00
3:00
6:00
9:00
12:00
15:00
18:00
21:00
Hours of Day