Tile38

Vlad Mattiussi
vlad.mattiussi@studio.unibo.it

April 2021

Il progetto si occuperà di eseguire uno studio su “Tile38”. Tile38 è un’applicazione open source contenente un database di localizzazione geo spaziale, e permette di fare geofencing. Un geo-fence è un perimetro virtuale associato ad un’area geografica del mondo reale. Può essere generato dinamicamente, ad esempio l’area entro un raggio da un punto prefissato, oppure può essere definito da un insieme di confini prestabiliti. Tile 38 permette quindi di memorizzare e gestire queste coordinate virtuali spaziali.

1 Goal/Requirements

La prima parte del progetto sarà dedicata sullo studio e comprensione della tecnologia Tile38, con uno sguardo generale dal punto di vista di un sistema distribuito in un reale scenario applicativo.

Si analizzeranno i principali componenti e le loro eventuali interazioni (server, client) per poi successivamente eseguire dei test pratici. Per la parte pratica verrà utilizzata anche la tecnologia Docker. Tile38 utilizza il protocollo nativo Redis RESP, di conseguenza i client supportano diversi linguaggi di programmazione. Questo progetto si concentrerà principalmente sul linguaggio Python e Java, ma non si esclude un possibile confronto fra i vari client. Si eseguiranno delle prove sia attraverso l’interfaccia da riga di comando, sia tramite l’implementazione di test in Python. Alla fine si giungerà ad un confronto tra i pro e i contro di questa tecnologia per concludere con le valutazioni finali.

1.1 Piano di lavoro

- Parte teorica: studiare e comprendere il funzionamento e le diverse applicazioni.
- funzionamento di questa tecnologia (come spiegato precedentemente).

2 Tile38

Figure 1: Tile38

2.1 Introduzione a Tile38

Tile38 [7] è un progetto open source dedicato al "realtime geospatial e geofencing": è un server che memorizza informazioni geospaziali, come ad esempio delle coordinate, ed esegue operazioni in tempo reale con esse. È utilizzata per la localizzazione fisica di utenti o oggetti entro un certo perimetro, associato a un'area geografica del mondo reale. Nell'impiego comune potrebbe essere utilizzata come applicazione per smartphone dedicata per l'utilizzo nel settore marketing. Il funzionamento e interfacciamento principale avviene tramite console di comando e client. L'utilizzo di Tile38 avviene attaverso la creazione ed esecuzione di comandi; per usarli è necessario conoscere due importanti elementi: l'operazione che vogliamo eseguire e gli oggetti che vogliamo utilizzare. Vediamo prima gli oggetti, poi i comandi veri e propri. Tile 38 supporta una varietà di tipi di oggetti suddivisi in tre categorie principali: coordinate di singoli punti, aree di delimitazione e coordinate complesse. Tutti i tipi di oggetto (ad eccezione di XYZ Tiles e QuadKeys) possono essere archiviati in una raccolta. (XYZ Tiles e QuadKeys sono riservati solo per la parola chiave SEARCH). Singoli punti:

- Lat/lon point: Il tipo di oggetto più semplice è un punto composto da una latitudine e una longitudine. Esiste un membro z facoltativo che può essere utilizzato per dati ausiliari come elevazione o timestamp.

Aree:

- Bounding box: cerca in un insieme oggetti che intersecano un'area di delimitazione specificata. Bisogna quindi specificare più coordinate rispetto al "point".

Coordinate complesse:

- Geohash: Un geohash è una rappresentazione in forma di stringa di un punto. Con la lunghezza della stringa che indica la precisione del punto; è un modo conveniente di esprimere una posizione (in qualsiasi parte del mondo) utilizzando una breve stringa alfanumerica, con una maggiore precisione ottenuta con stringhe più lunghe.
- GeoJSON: GeoJSON è un formato standard del settore per rappresentare una varietà di tipi di oggetti, tra cui un point, multipoint, linestring, multilinestring, polygon, multipolygon, geometrycollection, feature e featurecollection.
- XYZ Tile: Un riquadro XYZ è un'area di delimitazione rettangolare sulla terra rappresentata da una coordinata X, Y e un livello Z (zoom).
- QuadKey: Un QuadKey usa lo stesso sistema di coordinate di un riquadro XYZ, tranne per il fatto che la rappresentazione di stringa è un valore stringa composto da 0, 1, 2 o 3.

Per quanto riguarda i protocolli network, si utilizza una libreria client o Tile38 CLI, ma ci sono momenti in cui è disponibile solo HTTP o quando è necessario eseguire il test da un terminale remoto. In questi casi si forniscono opzioni HTTP e telnet.

- Http: Uno dei modi più semplici per chiamare un comando tile38 è usare HTTP.
- Websockets: I websocket possono essere utilizzati quando è necessario fare geofence e mantenere viva la connessione. Funziona in modo simile a HTTP, con l'eccezione che la connessione rimane attiva e i dati vengono inviati dal server come messaggi websocket di testo.
- Telnet: C'è la possibilità di utilizzare una semplice connessione telnet. L'output predefinito tramite telnet è RESP.

Il server risponderà in JSON o RESP a seconda del protocollo utilizzato all'avvio del primo comando: Http e Websockets utilizzano JSON, mentre i client Telnet e RESP utilizzano RESP.

2.2 API principali

Tile38 possiede una moltitudine di comandi, noi vedremo solo quelli più importanti, ossia quelli di inserimento, cancellazione e ricerca. La sintassi e struttura dei comandi per Tile38 è mantenuta invariata fra le varie modalità di client, ad esempio fra la console di comando docker e il client java. In seguito è espressa in ordine la struttura da conoscere per creare un comando Tile38 per l'inserimento:

- 1: comando: keyword che determina l'operazione che vogliamo compiere.
- 2: collezione: questa variabile (opzionale) determina il nome della collezione di oggetti di cui fara parte l'elemento.
- 3: elemento: variabile che individua il nome del soggetto dell'operazione.
- 4: campo: variabile opzionale da inserire se si vuole specificare ulteriori campi, come ad esempio la velocità. Verrà descritta meglio successivamente.
- 4: oggetto: keyword che assegna un tipo di oggetto all'elemento inserito prima (visto precedentemente).
- 5: coordinata: variabili numeriche da inserire per specificare le coordinate e informazioni riguardanti l'elemento.

Analizziamo ora i comandi principali di tile38, ovvero quelli di inserimento, aggiunta e cancellazione. Ci sono 6 comandi principali:

- "SET": imposta il valore di un ID. Se un valore è già associato a tale chiave/ID, esso verrà sovrascritto.

  ```
  SET fleet truck1 POINT 33.5123 -112.2693 245.0
  ```

  Esempio: creiamo un elemento "truck1" appartenente alla collezione "fleet", e assegniamo una certa coordinata singola.

```
SET fleet truck1 POINT 33.5123 -112.2693 245.0
```

Esempio di aggiunta di un Geohash:

```
SET props area1 HASH 9tbnwg
```

Esempio di aggiunta di un GeoJSON:

```
SET cities tempe OBJECT {"type":"Polygon","coordinates":
[[[-111.9787,33.4411],[-111.8902,33.4377],
[-111.8950,33.2892],[-111.9739,33.2932],
[-111.9787,33.4411]]]}
```

- "CONFIG SET": il comando CONFIG SET viene utilizzato per impostare proprietà di configurazione speciali, ad esempio per settare una password per un operazione.

```
CONFIG SET requirepass mypass
```

- "FSET": serve per impostare il valore per uno o più campi di un ID. I campi possono essere coordinate float e double. Il valore restituito è un valore intero rappresentante il conteggio del numero di campi effettivamente modificati.

```
FSET fleet truck1 speed 16 wheels 8
```

- "JSET": imposta un valore in un documento JSON, ad esempio:

```
JSET user 901 name.first Tom
JSET user 901 name.last Anderson
JGET user 901
> {"name":{"first":"Tom","last":"Anderson"}}
```

- "SETCHAN": Crea un canale Pub/Sub che punta a una ricerca geofence. Se un canale è già associato a tale nome, verrà sovrascritto. Una volta creato il canale, un client può quindi ascoltare gli eventi su quel canale con "SUBSCRIBE" o "PSUBSCRIBE".

```
SETCHAN warehouse NEARBY fleet FENCE
POINT 33.5123 -112.2693 500
```

- "SETHOOK": Crea una "webhook" che punta a una ricerca geofence. Se esso è già associato a quel nome, verrà sovrascritto.

```
SETHOOK warehouse http://10.0.20.78/endpoint NEARBY fleet
FENCE POINT 33.5123 -112.2693 500
```

In questo esempio abbiamo creato un "webhook" denominato "warehouse" che osserva le modifiche apportate agli oggetti nella raccolta "fleet". Quando si verifica una modifica, l'endpoint 'http://10.0.20.78/endpoint' riceve una notifica con un messaggio dettagliato. Il messaggio contiene informazioni sullo stato e posizione dell'oggetto in questione, ad esempio se si trova fuori o dentro l'area di geofence, oppure se ha oltrepassato i confini dell'area. Inoltre sono supportati vari "endpoint", tra cui http/https e redis per una connessione ad un server redis.

- "DEL": cancellazione di un oggetto, come ad esempio un ID.

```
    DEL fleet truck1
```

2.3 Fields

I campi (o "fields") sono dati aggiuntivi che appartengono a un oggetto. Un campo è sempre un punto mobile a doppia precisione. Non esiste un limite al numero di campi che un oggetto può avere. Per impostare un campo quando si imposta un oggetto:

```
set fleet truck1 field speed 90 point 33.5123 -112.2693
```

2.4 Ricerca

Tile38 ha il supporto per cercare oggetti e punti che si trovano all'interno o intersecano altri oggetti. È possibile eseguire ricerche in tutti i tipi di oggetto, inclusi poligoni, multipoligoni e così via.

- Within: cerca in un insieme oggetti completamente contenuti all'interno di un'area di delimitazione specificata.

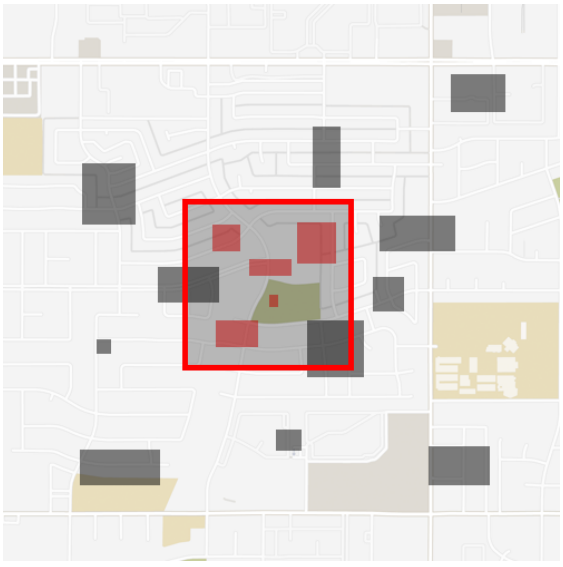

Figure 2: Within

- Intersects: cerca in un insieme oggetti che intersecano un'area di delimitazione specificata.

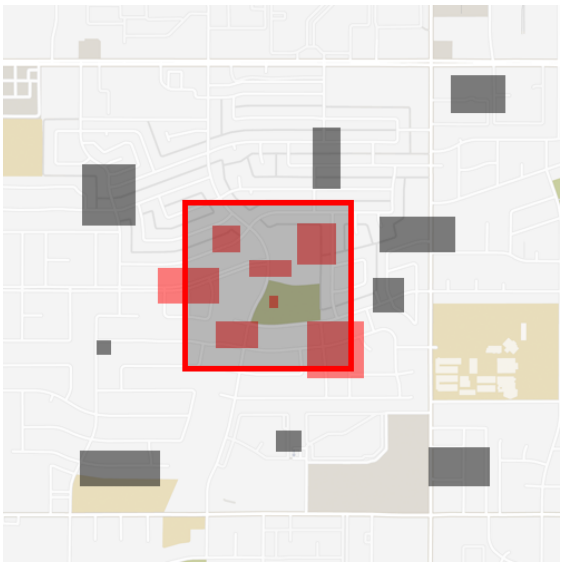

Figure 3: Intersects

- Nearby: cerca in un insieme oggetti che intersecano un raggio specificato.

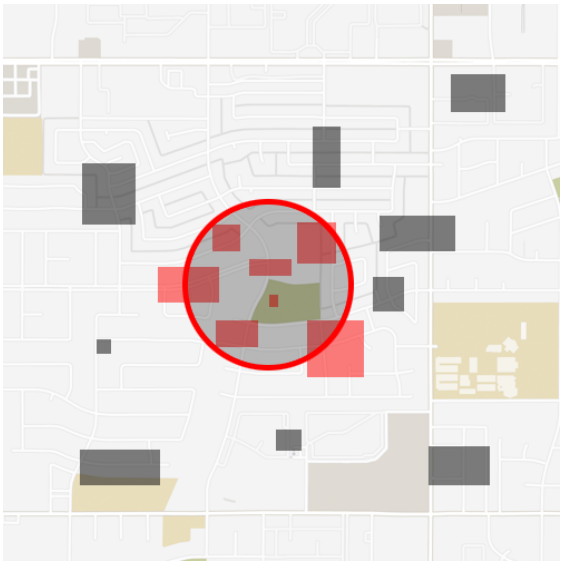

Figure 4: Nearby

2.4.1 Opzioni di ricerca

Per i comandi di ricerca e possibile esprimere alcune ulteriori opzioni.

- SPARSE: Questa opzione distribuirà i risultati di una ricerca in modo uniforme nell'area richiesta. Questo è molto utile per esempio: quando si hanno molti (forse milioni) di oggetti e non si vogliono tutti raggruppati su una mappa. SPARSE limiterà il numero di oggetti restituiti e li fornirà in modo uniforme in modo che la mappa sembri pulita. È possibile scegliere un valore compreso tra 1 e 8. Il valore 1 comporterà non più di 4 elementi. Il valore 8 si tradurrà in non più di 65536. 1=4, 2=16, 3=64, 4=256, 5=1024, 6=4098, 7=16384, 8=65536. Si nota che maggiore è il valore sparso, più lente sono le prestazioni. Inoltre, LIMIT e CURSOR non sono disponibili quando si utilizza SPARSE.

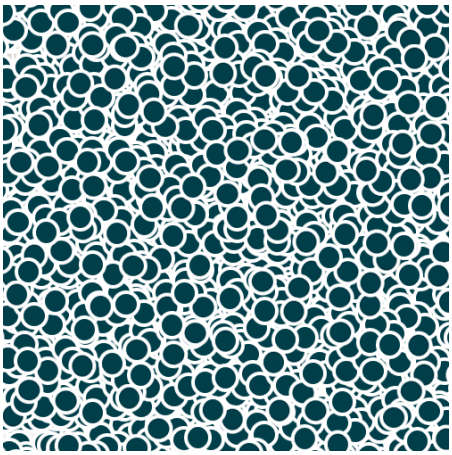

Figure 5: Sparse-none

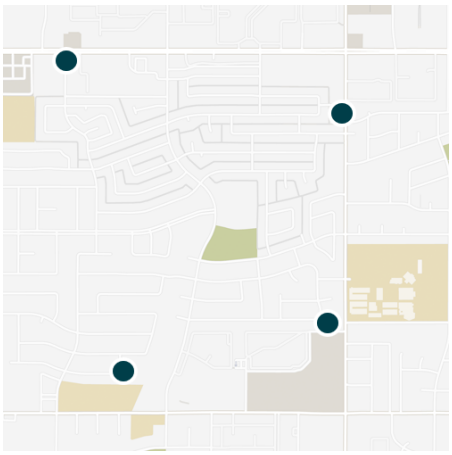

Figure 6: Sparse-1

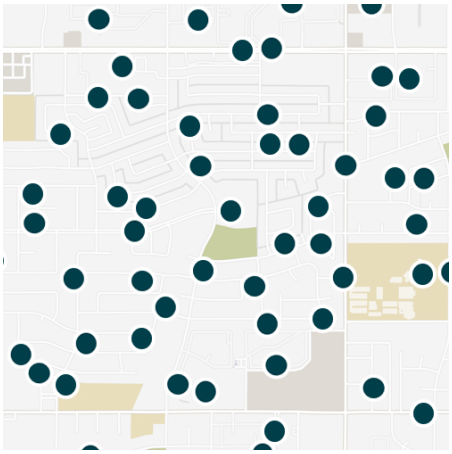

Figure 7: Sparse-3

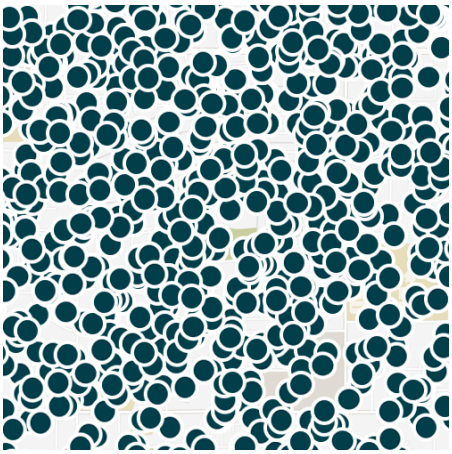

Figure 8: Sparse-5

- WHERE: Questa opzione consente di filtrare i risultati in base ai valori dei campi. Ad esempio,

```
nearby fleet where speed 70 +inf point 33.462 -112.268 6000
```

restituirà solo gli oggetti nella collezione 'fleet' che rientrano nel raggio di 6 km e hanno un campo denominato velocità superiore a 70. Molteplici WHERE possono essere concatenati:

```
WHERE speed 70 +inf WHERE age -inf 24
```

sarebbe interpretata come velocità superiore ai 70 e età inferiore a 24. Il valore predefinito per un campo è sempre 0. Pertanto, se si fa un WHERE sulla velocità

del campo e un oggetto non ha quel campo impostato, il server considererà che il valore sia 0.

- MATCH: MATCH è simile a WHERE, con la differenza che funziona sull'ID oggetto anziché sui campi.

```
nearby fleet match truck* point 33.462 -112.268 6000
```

  restituirà solo gli oggetti nella collezione 'fleet' che si trovano nel raggio di 6 km e hanno un ID oggetto che inizia con "truck". Ci possono essere più opzioni MATCH in una singola ricerca.

- CURSOR: CURSOR viene utilizzato per scorrere molti oggetti dai risultati della ricerca. Un'iterazione inizia quando cursor è impostato su 0 o non è incluso nella richiesta e viene completato quando il cursore restituito dal server è 0.

- NOFIELDS: NOFIELDS indica al server che non si desidera restituire i valori dei campi con i risultati della ricerca.

- LIMIT: LIMIT può essere utilizzato per limitare il numero di oggetti restituiti per una singola richiesta di ricerca.

2.5 Geofencing

Un Geofence (recinto virtuale) è un limite virtuale in grado di rilevare quando un oggetto entra o esce dall'area. Questo limite può essere un raggio, un riquadro di delimitazione o un poligono. Tile38 può trasformare qualsiasi ricerca standard in un monitor del recinto virtuale aggiungendo la parola chiave FENCE alla ricerca.

Figure 9: Esempio di applicazione Geofence su una mappa

2.6 Librerie dei client

2.6.1 Redis

Figure 10: Redis

Tile38 comunica con il server Redis utilizzando un protocollo denominato RESP (REdis Serialization Protocol). Mentre il protocollo è stato progettato specificamente per Redis, può essere utilizzato per altri progetti software client-server.

Redis [6] è un DBMS NoSQL di tipo "key/value storage". Esso si basa infatti su una struttura a dizionario: ogni valore immagazzinato è abbinato ad una chiave univoca che ne permette il recupero. Alcuni vantaggi di rilievo:

- Possiede una buona velocità: Redis conserva i dati in memoria RAM, salvandoli in maniera persistente solo in un secondo momento. Ciò permette di ottenere ottime prestazioni in scrittura e lettura;
- Dispone di una grande varietà di tipi di dato. Quindi nonostante l'architettura dell'archivio sia basata su una struttura a dizionario, i valori possono assumere varie forme: liste, dizionari stessi, e molto altro. Da questo punto di vista, Redis può essere visto come un gestore di strutture dati persistenti;
- Tutte le operazioni sono atomiche, quindi in caso di accessi concorrenti da parte di più client, i dati forniti risulteranno sempre aggiornati;
- Possibilità di implementare configurazioni multi-nodo, cluster e replication.

Tile38 usa il protocollo RESP Redis in modo nativo. Pertanto, la maggior parte dei client che supportano i comandi Redis di base supporterà a sua volta Tile38.

RESP può essere considerato come un insieme tra le seguenti cose: semplice da implementare, veloce da analizzare, leggibile dall'uomo. RESP può serializzare diversi tipi di dati come interi, stringhe, matrici. Esiste anche un tipo specifico per gli errori. Le richieste vengono inviate dal client al server Redis come matrici di stringhe che rappresentano gli argomenti del comando da eseguire. Redis risponde con un tipo di dati specifico del comando. RESP è binary-safe e non richiede l'elaborazione di dati di massa

trasferiti da un processo all'altro, perché utilizza la lunghezza prefissata per trasferire dati di massa.

Un client si connette a un server Redis creando una connessione TCP alla porta 6379. Mentre RESP è tecnicamente non specifico del TCP, nel contesto di Redis il protocollo viene utilizzato solo con connessioni TCP (o connessioni equivalenti orientate al flusso come i socket Unix).

Redis accetta comandi composti da argomenti diversi. Una volta ricevuto, un comando viene elaborato e una risposta viene rispedita al client.

Questo è il modello più semplice possibile, tuttavia ci sono due eccezioni:

Redis supporta il pipelining. Quindi è possibile per i client inviare più comandi contemporaneamente e attendere le risposte in un secondo momento. Quando un client Redis sottoscrive un canale Pub/Sub, il protocollo modifica la semantica e diventa un protocollo push, o cioè il client non richiede più l'invio di comandi, perché il server invierà automaticamente al client nuovi messaggi (per i canali a cui il client è iscritto) non appena vengono ricevuti. Escludendo le due eccezioni di cui sopra, il protocollo Redis è un semplice protocollo di richiesta-risposta.

3 Parte Pratica progetto

Tile38 puo' essere avviato ed eseguito tramite due principali metodi: utilizzando Docker oppure eseguendo il programma con "Go".

3.1 Docker

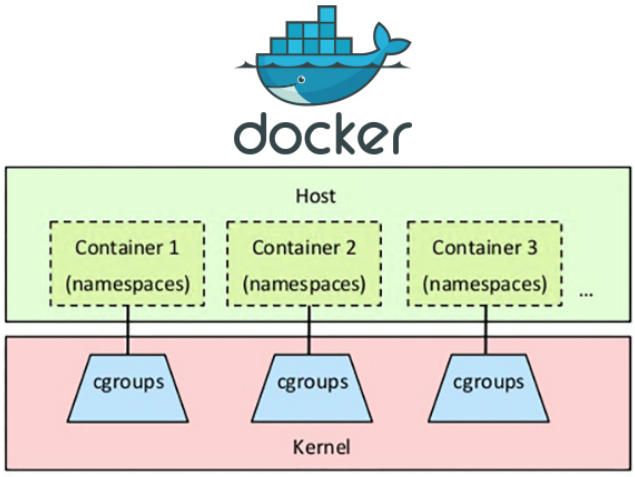

Figure 11: Docker

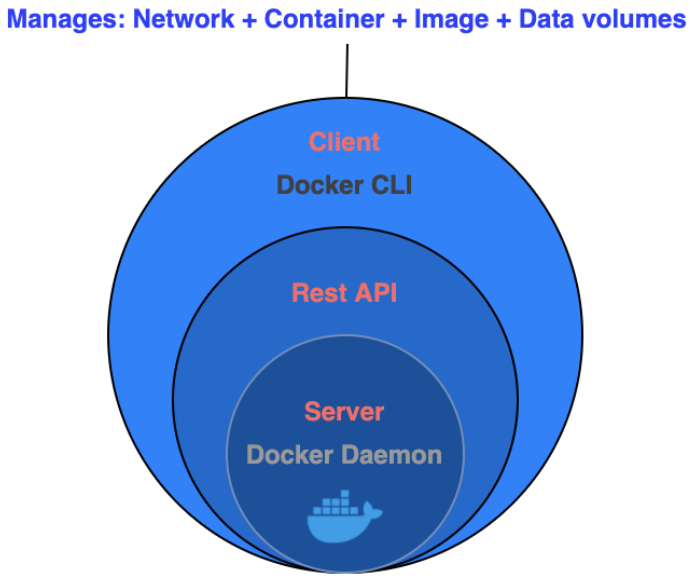

Figure 12: Docker Daemon

Docker [1] è un progetto open-source che automatizza il deployment di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Docker utilizza le funzionalità di isolamento delle risorse del kernel Linux per consentire a "container" indipendenti di coesistere sulla stessa istanza di Linux, evitando l'installazione e la manutenzione di una macchina virtuale. I container offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, e strumenti di sistema. Permette di creare, testare e distribuire applicazioni con rapidità. Docker fornisce una modalità standard per eseguire il codice. Si tratta di un sistema operativo per container. Così come la macchina virtuale virtualizza i server hardware, i container virtualizzano il sistema operativo di un server. Docker è installato su ogni server e fornisce semplici comandi con cui creare, avviare o interrompere i container.

Docker utilizza un'architettura client-server. Il client Docker parla con il daemon Docker, che esegue il build e della distribuzione dei contenitori Docker. Il client Docker e il daemon possono essere eseguiti sullo stesso sistema oppure è possibile connettere un client Docker a un daemon Docker remoto. Essi comunicano utilizzando un'API REST, tramite socket UNIX o un'interfaccia di rete. Un altro client Docker è Docker Compose, che consente di lavorare con applicazioni costituite da un set di contenitori.

Alcuni dei componenti principali di Docker:

- Immagine: un'immagine è un modello di sola lettura con istruzioni per la creazione di un contenitore Docker. Spesso, un'immagine si basa su un'altra immagine, con

alcune personalizzazioni aggiuntive. Ad esempio, è possibile creare un'immagine basata sull'immagine ubuntu, ma installa il server Web Apache e l'applicazione, nonché i dettagli di configurazione necessari per far funzionare l'applicazione. È possibile creare immagini proprie oppure utilizzare solo quelle create da altri e pubblicate in un Registro di sistema. Per creare un'immagine personalizzata, è necessario creare un Dockerfile con una sintassi semplice per definire i passaggi necessari per creare l'immagine ed eseguirla. Ogni istruzione in un Dockerfile crea un livello nell'immagine. Quando vengono modificati file docker e ricostruite immagini, vengono ricostruiti solo i livelli modificati. Questo fa parte di ciò che rende le immagini così leggere, piccole e veloci, rispetto ad altre tecnologie di virtualizzazione.

- Contenitori: un contenitore è un'istanza eseguibile di un'immagine. È possibile creare, avviare, arrestare, spostare o eliminare un contenitore utilizzando l'API Docker o CLI. È possibile connettere un contenitore a una o più reti, collegarvi spazio di archiviazione o persino creare una nuova immagine in base allo stato corrente. Per impostazione predefinita, un contenitore è relativamente ben isolato dagli altri contenitori e dal relativo computer host. È possibile controllare la posizione isolata della rete, dell'archiviazione o di altri sottosistemi sottostanti di un contenitore da altri contenitori o dal computer host. Un contenitore è definito dalla sua immagine e da tutte le opzioni di configurazione fornite quando si crea o inizializza. Quando un contenitore viene rimosso, tutte le modifiche apportate al relativo stato che non sono archiviate nell'archiviazione permanente scompaiono.

3.2 Go

Figure 13: Go programming language

Si tratta di un linguaggio di programmazione open source pensato per realizzare applicativi e software multipiattaforma. Sviluppato da Google [2], per velocizzare e semplificare la scrittura di programmi altamente concorrenti e scalabili, è fortemente tipizzato e multifunzionale. Go soddisfa le esigenze della programmazione concorrente ed è stato progettato per ottimizzare i tempi di compilazione anche per hardware modesti. La sintassi è vicina al C eccetto per la dichiarazione dei tipi e per la mancanza di parentesi tonde nei costrutti for e if. Ha un sistema di garbage collection che si occupa autonomamente

della gestione della memoria. Non include l'intercettazione di eccezioni, l'eredità dei tipi, la programmazione generica, le asserzioni e l'overloading dei metodi.

3.3 Go Benchmarking

Nel progetto sono, inoltre, presenti alcuni file di test per testare al meglio le diverse funzioni di tile38. In particolare è presente un file di benchmark contentente dei test sulle tre diverse modalita' di ricerca di Tile38 (within, intersects e nearby). Esso crea alcuni punti in posizioni random entro un certo limite spaziale e veriffica il corretto posizionamento di essi. Di seguito un esempio di codice con "Within":

```
switch strings.ToUpper(strings.TrimSpace(test)) {
case "WITHIN", "WITHIN-BOUNDS", "WITHIN-BOUNDS-1000":
minlat, minlon, maxlat, maxlon := randRect(1000)
redbench.Bench("WITHIN (within-bounds 1km)", addr, opts,prepFn,
        func(buf []byte) []byte {
                return redbench.AppendCommand(buf,
                "WITHIN", "key:bench", "COUNT", "BOUNDS",
                strconv.FormatFloat(minlat, 'f', 5, 64),
                strconv.FormatFloat(minlon, 'f', 5, 64),
                strconv.FormatFloat(maxlat, 'f', 5, 64),
                strconv.FormatFloat(maxlon, 'f', 5, 64))
                },
        )
}
```

Tramite ulteriori funzioni di controllo, vediamo come risultati finali i diversi tempi impiegati e le richieste totali fatti entro un certo intervallo di tempo:

```
====== WITHIN (within-circle 1km) ======
  100000 requests completed in 0.90 seconds
  50 parallel clients
  93 bytes payload
  keep alive: 1

79.78% <= 0 milliseconds
98.58% <= 1 milliseconds
99.51% <= 2 milliseconds
99.61% <= 3 milliseconds
99.67% <= 4 milliseconds
99.70% <= 5 milliseconds
99.71% <= 6 milliseconds
99.72% <= 7 milliseconds
99.81% <= 8 milliseconds
99.81% <= 9 milliseconds
99.83% <= 11 milliseconds
```

```
99.85% <= 12 milliseconds
99.90% <= 13 milliseconds
99.93% <= 14 milliseconds
99.96% <= 15 milliseconds
99.99% <= 16 milliseconds
100.00% <= 22 milliseconds
111234.88 requests per second
```

Da ciò possiamo notare come Tile38 possieda una buona velocità e reattività ai diversi tipi di richieste.

3.4 Docker tests

Installiamo e avviamo Tile38 tramite Docker, inserendo i comandi:

```
docker pull tile38/tile38
docker run -p 9851:9851 tile38/tile38
```

Avviamo il server con:

```
tile38-server
```

Avviamo il client in un'altra console con:

```
tile38-cli
```

Adesso abbiamo una comunicazione operativa fra server e client Tile38. Iniziamo con alcune semplici operazioni di base: aggiungiamo 2 punti chiamati "truck1" e "truck2" ad una collezione chiamata "fleet", per poi eseguire una ricerca spaziale entro una certa area e avere come risultato uno dei due punti; infine eliminiamo tutti i punti.

```
#creazione punti
> set fleet truck1 point 33.5123 -112.2693
> set fleet truck2 point 33.4626 -112.1695

# ricerca
> scan fleet
> nearby fleet point 33.462 -112.268 6000

# ritorno e cancellazione punti
> get fleet truck1
> del fleet truck2
> drop fleet
```

Di seguito una lista completa con tutti i comandi disponibili: `https://tile38.com/commands`.

3.4.1 Geofencing tests

In seguito alcuni test di geofencing tramite l'utilizzo di due terminali. Nel primo terminale ci colleghiamo al server Tile38 e creiamo il geofence utilizzando il comando SETCHAN o SETHOOK. In questo esempio utilizziamo un canale publisher/subscriber (più avanti verrà approfondito l'argomento nella parte di python)

```
SETCHAN warehouse NEARBY fleet FENCE POINT 33.462 -112.268 6000
SUBSCRIBE warehouse
```

Sottoscriviamo sul canale PubSub geofence appena creato.

Nel secondo terminale ci colleghiamo al server Tile38 e creiamo un punto nella collezione "fleet" che attiverà una notifica geofence.

```
SET fleet bus POINT 33.460 -112.260
```

Sul terminale 1 apparira':

```
"command": "set",
  "group": "5c5203ccf5ec4e4f349fd038",
  "detect": "inside",
  "hook": "warehouse",
  "key": "fleet",
  "time": "2019-01-30T13:06:36.769273-07:00",
  "id": "bus",
  "object":  "type": "Point", "coordinates": [-112.26, 33.46]
```

Il server avviserà i client sottoscritti se il comando è "del", "set" o "drop".

- DEL: notifica al client che un oggetto è stato eliminato dall'insieme recintato.
- DROP: notifica al client che l'intera raccolta è stata eliminata.
- SET: notifica al client che un oggetto è stato aggiunto o aggiornato e quando la sua posizione viene rilevata dal recinto.

Il rilevamento può essere uno dei valori seguenti;

- INSIDE: quando un oggetto si trova all'interno dell'area specificata.
- OUTSIDE: quando un oggetto si trova al di fuori dell'area specificata.
- ENTER: quando un oggetto che non era precedentemente nel recinto è entrato nell'area.
- EXIT: quando un oggetto che si trovava in precedenza nella recinzione è uscito dall'area.
- CROSS: quando un oggetto che non era precedentemente nella recinzione è entrato ed è uscito dall'area.

Tile38 1.2 introduce una funzione che consente geofencing dinamici. Ciò consente il monitoraggio in tempo reale quando uno o più oggetti in movimento si trovano l'uno nelle vicinanze.

Un paio di casi d'uso comuni sono:

- Servizi di ritiro del veicolo: ricevere immediatamente una notifica quando un veicolo si trova nelle vicinanze di qualcuno in attesa di essere prelevato o quando una persona si avvicina a un veicolo o quando un veicolo si trova nelle vicinanze di altri veicoli della fleet.
- App social di prossimità: molto utile per quando è necessario verificare se due utenti si trovano l'uno nelle vicinanze senza dover eseguire costantemente query sul database.

Un semplice esempio in seguito.

```
NEARBY people FENCE ROAM people * 5000
```

Questo aprirà una "recinzione" vagante sulla collezione di persone. La recinzione verifica quando un oggetto si trova entro 5000 metri da qualsiasi altro oggetto della stessa collezione. Per testare, aprire due terminali: Sul teminale 1 connettersi al server Tile38 ed entrare nel comando fence.

```
$ tile38-cli
localhost:9851> NEARBY people FENCE ROAM people * 5000
+OK
```

Sul teminale 2 aggiungere due punti alla collezione "people". Il secondo comando SET attiverà un evento di recinzione che apparirà nell'altro terminale.

```
$ tile38-cli
localhost:9851> SET people bob POINT 33.01 -115.01
localhost:9851> SET people alice POINT 33.02 -115.02
```

L'evento apparirà nel terminal 1 e sarà simile a:

```
{"command": "set",
  "detect": "roam",
  "hook": "",
  "key": "people",
  "id": "alice",
  "time": "2016-05-24T09:19:44.08649461-07:00",
  "object": { "type": "Point", "coordinates": [-115.02, 33.02] },
  "nearby": {
    "key": "people",
    "id": "bob",
    "meters": 1451.138152186708
  }
}
```

Il quale mostra che "alice" è stato aggiornato e "bob" è ad una distanza di 1451 metri.

3.5 Go client

Similmente all'utilizzo di tile-38 tramite la conseole di docker, il programma può essere avviato ed eseguito tramite Go a riga di comando. Le caratterisctiche del programma sono le stesse. Un altro componente presente e' "tile38-benchmark" contenente delle utilita' per eseguire dei test di benchmark; vengono creati ed eseguiti tanti comandi e vengono misurati i tempi di esecuzione (come visto in precedenza).

3.6 Python client

Per utilizzare Tile38 con python, prima di tutto, bisogna installare il pacchetto redis-py tramite "pip install redis", successivamente avviare il server Tile38 utilizzando Go e la console di comando. Infine creiamo una semplice funzione di test connessione.

```
def test_tile38():
    client = redis.Redis(host='127.0.0.1', port=9851)
    # insert data
    result = client.execute_command('SET', 'fleet', 'truck',
    'POINT', 33.32, 115.423)
    # print result
    print(result)
    # get data
    print(client.execute_command('GET', 'fleet', 'truck'))

if __name__ == '__main__':
    test_tile38()
True
b'{"type":"Point","coordinates":[115.423,33.32]}'
```

In questo caso e' stato creato con un succcesso un nuovo punto con tali coordinate.

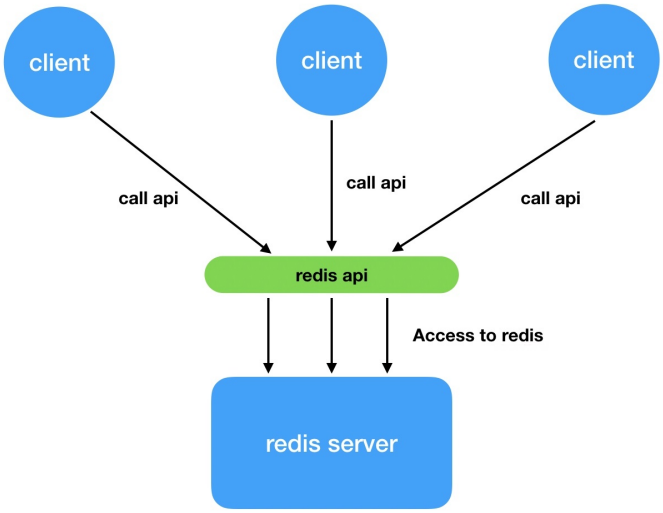

Figure 14: Esempio funzionamento Redis-Py

Redis-py utilizza un pool di connessioni per gestire le connessioni a un server Redis. Per impostazione predefinita, ogni istanza Redis creata creerà a sua volta il proprio pool di connessioni. È possibile eseguire l'override di questo comportamento e utilizzare un pool di connessioni esistente passando un'istanza del pool di connessioni già creata all'argomento del pool di connessioni della classe Redis. È possibile scegliere di eseguire questa procedura per implementare lo sharding lato client o avere un controllo granulare sulla modalità di gestione delle connessioni.

ConnectionPools gestisce un set di istanze di connessione. Redis-py viene fornito con due tipi di connessioni. L'impostazione predefinita, "Connessione", è una normale connessione basata su socket TCP. UnixDomainSocketConnection consente ai client in esecuzione sullo stesso dispositivo del server di connettersi tramite un socket di dominio unix. Per utilizzare una connessione UnixDomainSocketConnection, è sufficiente passare l'argomento unixsocketpath, che è una stringa al file socket di dominio unix. Bisogna inoltre assicurarsi che il parametro unixsocket sia definito nel file redis.conf. È commentato per impostazione predefinita.

È inoltre possibile creare sottoclassi di connessione proprie. Questo può essere utile se si desidera controllare il comportamento del socket all'interno di un framework asincrono. Per creare un'istanza di una classe client utilizzando una connessione personalizzata, è necessario creare un pool di connessioni, passando la classe all'argomento connectionclass. Altri parametri di parole chiave passati al pool verranno passati alla classe specificata durante l'inizializzazione.

Le connessioni mantengono un socket aperto al server Redis. A volte queste vengono interrotte o disconnesse per una serie di motivi. Ad esempio, gli accessori di rete, i servizio di bilanciamento del carico e altri servizi che si trovano tra client e server sono spesso configurati per terminare le connessioni che rimangono inattive per una determi-

nata soglia di tempo.

Quando una connessione viene disconnessa, il comando successivo emesso su tale connessione avrà esito negativo e redis-py genererà un ConnectionError al chiamante. Ciò consente a ogni applicazione che utilizza redis-py di gestire gli errori in modo appropriato per quell'applicazione specifica. Tuttavia, la gestione costante degli errori può essere dettagliata e ingombrante, specialmente quando le disconnessioni socket si verificano frequentemente in molti ambienti di produzione.

3.7 Publish/Subscribe

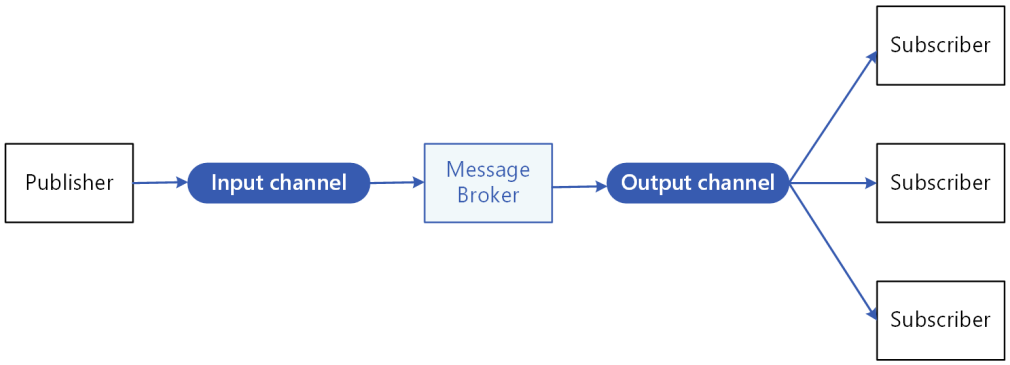

Figure 15: Publish/Subscribe pattern

Publish/Subscribe è un design pattern o stile architetturale utilizzato per la comunicazione asincrona fra diversi processi, oggetti o altri agenti. Questo pattern pattern può essere considerato un middleware per la sua caratteristica di integrazione tra diverse sorgenti software.

Redis-py include un oggetto PubSub che sottoscrive i canali e ascolta nuovi messaggi.Per creare un oggetto PubSub:

```
r = redis.Redis(...)
p = r.pubsub()
```

Una volta creata un'istanza PubSub, è possibile sottoscrivere canali e modelli.

```
p.subscribe('my-first-channel', 'my-second-channel', ...)
p.psubscribe('my-*', ...)
```

L'istanza PubSub è ora sottoscritta a tali canali/modelli. Le conferme di sottoscrizione possono essere viste leggendo i messaggi dall'istanza PubSub.

```
p.get_message()
{'pattern': None,'type':'subscribe','channel':
b'my-second-channel','data':1}
p.get_message()
{'pattern': None,'type':'subscribe','channel':
```

```
b'my-first-channel','data':2}
p.get_message()
{'pattern': None,'type':'psubscribe','channel':
b'my-*','data':3}
```

Ogni messaggio letto da un'istanza PubSub sarà un dizionario con le seguenti chiavi. tipo: uno dei seguenti:

- 'subscribe', 'unsubscribe', 'psubscribe', 'punsubscribe', 'message', 'pmessage'.
- Canale: il canale sottoscritto o il canale in cui è stato pubblicato un messaggio.
- Pattern: il modello che corrisponde al canale di un messaggio pubblicato. Sarà Null in tutti i casi ad eccezione dei tipi 'pmessage'.
- Dati: i dati del messaggio. Con il subscribe dei messaggi, questo valore sarà il numero di canali e modelli a cui la connessione è attualmente sottoscritta; con i 'pmessage', questo valore sarà il messaggio pubblicato effettivo. Mandiamo un messaggio d'esempio.

```
>>> r.publish('my-first-channel', 'some data')
p.get_message()
{'channel': b'my-first-channel', 'data': b'some data',
'pattern': None, 'type': 'message'}
p.get_message()
{'channel': b'my-first-channel', 'data': b'some data',
'pattern': b'my-*', 'type': 'pmessage'}
```

L'annullamento dell'iscrizione funziona proprio come la sottoscrizione. Se a unsubscribe non vengono passati argomenti, tutti i canali o i modelli verranno annullati dall'iscrizione.

```
p.unsubscribe()
p.punsubscribe('my-*')
p.get_message()
{'channel': b'my-second-channel', 'data': 2,
'pattern': None, 'type': 'unsubscribe'}
p.get_message()
{'channel': b'my-first-channel', 'data': 1,
'pattern': None, 'type': 'unsubscribe'}
p.get_message()
{'channel': b'my-*', 'data': 0,
'pattern': None, 'type': 'punsubscribe'}
```

Redis-py consente inoltre di registrare le funzioni di richiamata per gestire i messaggi pubblicati. I gestori di messaggi prendono un singolo argomento, il messaggio, che è un dizionario proprio come gli esempi precedenti. Per sottoscrivere un canale o un modello

con un gestore di messaggi, bisogna passare il nome del canale o del modello come argomento di parola chiave con il relativo valore come funzione di callback. Quando un messaggio viene letto su un canale o un modello con un gestore messaggi, il dizionario messaggi viene creato e passato al gestore. In questo caso, viene restituito un valore Null da "getmessage()" poiché il messaggio era già stato gestito.

Esistono diverse strategie per la lettura dei messaggi. Gli esempi precedenti hanno utilizzato "pubsub.getmessage()". "Getmessage()" utilizza il modulo 'select' del sistema per eseguire rapidamente il polling del socket della connessione. Se sono disponibili dati da leggere, "getmessage()" lo leggerà, formatterà il messaggio e lo restituirà o lo passerà a un gestore messaggi. Se non ci sono dati da leggere, "getmessage()" restituirà immediatamente Null.

```
while True:
 message = p.get_message()
 if message:
  # do something with the message
 time.sleep(0.001)
```

Un ulteriore opzione esegue un ciclo di eventi in un thread separato. "Pubsub.runinthread()" crea un nuovo thread e avvia il ciclo di eventi. L'oggetto thread viene restituito al chiamante di "runinthread()". Il chiamante può utilizzare il metodo thread.stop() per arrestare il ciclo di eventi e il thread. Questo è semplicemente un wrapper intorno a "getmessage()" che viene eseguito in un thread separato, creando essenzialmente un piccolo ciclo di eventi non bloccante. "Runinthread()" accetta un argomento facoltativo del tempo di sospensione. Se specificato, il ciclo di eventi chiamerà time.sleep() con il valore in ogni iterazione del ciclo. Poiché siamo in esecuzione in un thread separato, non è possibile gestire messaggi che non vengono gestiti automaticamente con gestori di messaggi registrati. Pertanto, redis-py impedisce di chiamare "runinthread()" se si è sottoscritti a canali a cui non sono associati gestori di messaggi.

```
p.subscribe(**{'my-channel': my_handler})
thread = p.run_in_thread(sleep_time=0.001)
# the event loop is now running in the background processing
messages
# when it's time to shut it down...
thread.stop()
```

Gli oggetti PubSub ricordano a quali canali e modelli sono sottoscritti. In caso di disconnessione, ad esempio un errore o un timeout di rete, l'oggetto PubSub si sottoscriverà nuovamente tutti i canali e i modelli precedenti durante la riconnessione. I messaggi pubblicati durante la disconnessione del client non possono essere recapitati. Al termine di un oggetto PubSub, chiamare il relativo metodo close() per arrestare la connessione.

```
p = r.pubsub()
...
p.close()
```

Sono supportati anche il set PUBSUB di sottocomandi CHANNELS, NUMSUB e NUMPAT:

```
r.pubsub_channels()
[b'foo', b'bar']
r.pubsub_numsub('foo', 'bar')
[(b'foo', 9001), (b'bar', 42)]
r.pubsub_numsub('baz')
[(b'baz', 0)]
r.pubsub_numpat()
1204
```

3.8 Java client

3.8.1 maven

Figure 16: Apache Maven

Maven [4], prodotto della Apache Software Foundation, è uno strumento di build automation utilizzato prevalentemente nella gestione di progetti Java. Simile per certi versi a strumenti precedenti, come ad esempio Apache Ant, si differenzia però da quest'ultimo per quanto concerne la compilazione del codice. Con questo strumento infatti non è più necessaria la compilazione totale del codice, ma viene fatto uso di una struttura di progetto standardizzata su template definita archetype.

Il vantaggio derivante dall'utilizzo di questo tool è da subito evidente: se generalmente per sviluppare un software sono necessarie numerose fasi, con la build automation l'intero processo viene automatizzato, riducendo il carico di lavoro del programmatore e diminuendo le possibilità di errore da parte dello stesso. Alcune delle fasi fondamentali che vengono automatizzate dal tool sono la compilazione in codice binario, il packaging dei binari, l'esecuzione di test per garantire il funzionamento del software, il deployment sui sistemi ed infine la documentazione relativa al progetto portato a termine. Esistono poi alcuni tratti comuni ai prodotti di build automation, a partire dal build file, che contiene tutte le operazioni svolte.

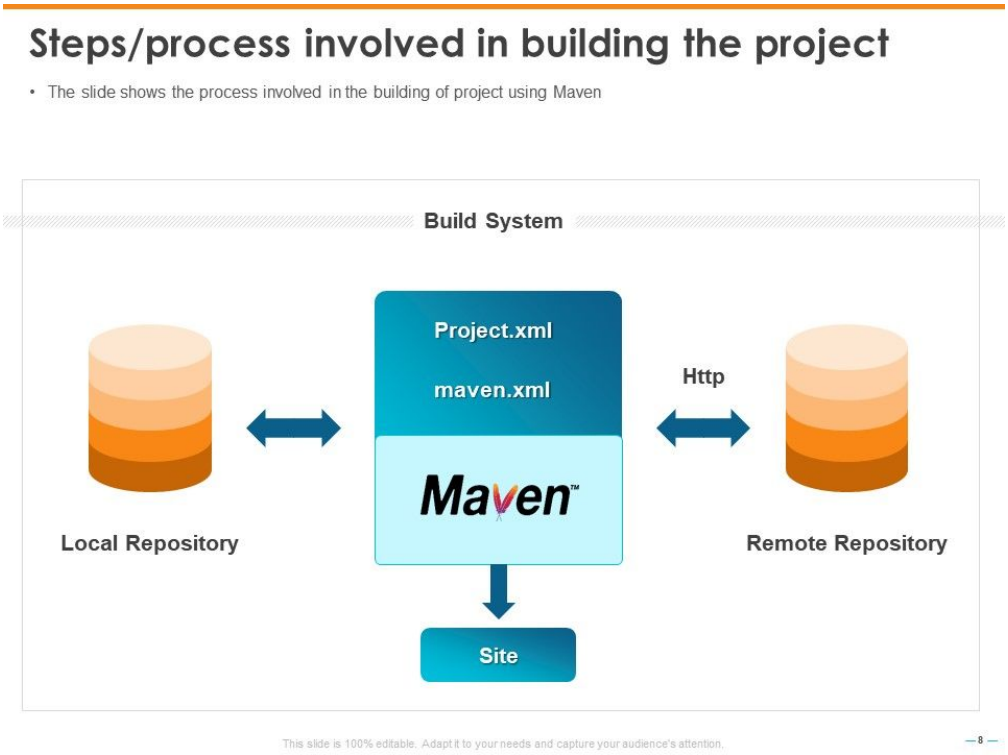

Figure 17: Esempio architettura Maven

Per creare un progetto maven e' necessario inserire il seguente comando in una console a riga di comando, in questo caso nel cmd di windows:

```
mvn archetype:generate -DgroupId=com.mycompany.app
-DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart
-DarchetypeVersion=1.4 -DinteractiveMode=false
```

- groupId: Questo elemento indica l'identificatore univoco dell'organizzazione o del gruppo che ha creato il progetto. Il "groupId" è uno degli identificatori di chiave di un progetto ed è in genere basato sul nome di dominio completo dell'organizzazione. Ad esempio org.apache.maven.plugins è il groupId designato per tutti i plugin Maven.
- ArtifactId: Questo elemento indica il nome di base univoco dell'artefatto primario generato da questo progetto. L'artefatto primario per un progetto è in genere un file JAR. Anche gli artefatti secondari come i bundle di origine usano l'artifactId come parte del loro nome finale.

Una volta premuto INVIO dopo aver digitato il comando precedente, inizierà a creare il progetto Maven.

3.8.2 lettuce

Figure 18: Lettuce

Lettuce [3] è un client Redis scalabile thread-safe per l'utilizzo sincrono, asincrono e reattivo. Più thread possono condividere una connessione se evitano operazioni di blocco e transazionali quali BLPOP e MULTI/EXEC. Lettuce è costruita con netty.

Figure 19: Netty

Netty è un framework di applicazioni di rete asincrono basato su eventi per lo sviluppo rapido di server e client. Lettuce supporta funzionalità Redis avanzate come i modelli di dati Sentinel, Cluster, Pipelining, Auto-Reconnect e Redis. Questa versione di Lettuce è stata testata rispetto all'ultima build sorgente Redis.

Lettuce copre quasi tutti i comandi Redis. Lo sviluppo di Redis è un processo in corso e il sistema Redis Module ha lo scopo di introdurre nuovi comandi che non fanno parte del Redis Core. Questo requisito introduce la necessità di richiamare comandi personalizzati o utilizzare output personalizzati. I comandi personalizzati possono essere inviati da un lato utilizzando Lua e "eval()", dall'altro Lettuce 4.x consente di attivare i propri comandi. Tale API viene utilizzata da lettuce stessa per inviare comandi e richiede una certa conoscenza di come essi vengono costruiti e inviati all'interno di lettuce.

Lettuce mette a disposizione due livelli di command dispatching:

- Utilizzo dei wrapper API sincroni, asincroni o reattivi che richiamano comandi in base alla loro natura.
- Utilizzo della sola connessione per influenzare la natura e la sincronizzazione dei comandi.

Lettuce utilizza un modello a comandi: ogni volta che viene richiamato un comando, Lettuce crea un oggetto "comando" (Command o tipi che implementano RedisCommand). I comandi possono portare argomenti (CommandArgs) e un output (sottoclassi di CommandOutput). Entrambi sono facoltativi. Invece, le due proprietà obbligatorie sono il tipo di comando (vedere CommandType o un tipo che implementa ProtocolKeyword) e un RedisCodec. Se si inviano soli comandi, non bisogna riutilizzare le istanze dei comandi per inviarli più di una volta. I comandi eseguiti una volta impostati con il flag completato non possono essere riutilizzati. La sequenza di args e parole chiave non viene convalidata da Lettuce oltre i tipi di dati supportati, il che significa che Redis segnala errori se la sintassi del comando non è corretta. Il metodo "dispatch()" serve per inviare il comando. Il gestore di risposta controlla la decodifica del comando e il suo completamento fino al termine. Lettuce lavora solamente in modalità di elaborazione asincrona.

- L'API asincrona funziona in generale con il wrapper AsyncCommand che estende CompleteableFuture. AsyncCommand può essere sincronizzato da await() o get() che corrisponde allo stile pull asincrono. Utilizzando i metodi dell'interfaccia CompletionStage (ad esempio handle() o thenAccept()), il gestore di risposta attiverà le funzioni ("listener") al completamento del comando.
- L'API sincrona di Lettuce utilizza la "future synchronization" per fornire una visualizzazione sincrona.
- I comandi reattivi vengono inviati al momento della sottoscrizione. Nel contesto di Lettuce ciò significa che è necessario iniziare prima di chiamare il metodo dispatch(). Se si desidera inviare comandi in modo reattivo, è necessario disporre i comandi con ReactiveCommandDispatcher. Il dispatcher implementa l'API OnSubscribe per creare un Observable, gestisce l'invio dei comandi al momento della sottoscrizione e può sciogliere i tipi di raccolta in elementi particolari. Un'istanza di ReactiveCommandDispatcher consente di creare più osservabili purché si utilizzi un fornitore RedisCommand. I comandi eseguiti una volta impostati con il flag completato non possono essere riutilizzati.

Proviamo a completare alcune semplici operazioni su Tile38; Ci connettiamo al nostro server avviato precedentemente da console di comando e, tramite dei comandi di SET e GET, impostiamo alcune coordinate e verifichiamo che siano state inserite correttamente.

```
RedisClient client = RedisClient.create("redis://localhost:9851");
StatefulRedisConnection<String,String>connection=client.connect();
RedisCommands<String, String> sync = connection.sync();
```

```
StringCodec codec = StringCodec.UTF8;
sync.dispatch(CommandType.SET,
        new StatusOutput<>(codec), new CommandArgs<>(codec)
                .add("fleet")
                .add("truck1")
                .add("POINT")
                .add(33L)
                .add(-115L));

String result = sync.dispatch(CommandType.GET,
        new StatusOutput<>(codec), new CommandArgs<>(codec)
                .add("fleet")
                .add("truck1"));

        System.out.println(result);
```

In questo caso la nostra stampa finale sarà:

```
{"type":"Point","coordinates":[-115,33]}
```

4 Conclusioni

Il progetto ha soddisfatto gli obiettivi posti inizialmente, sono stati toccati tutti i punti di interesse, dalle caratteristiche principali di Tile38, fino ai protoccoli network e client alla base di vari funzionamenti. E' stata vista la versatilità di questo software, soprattutto per quanto riguarda l' interoperabilità dei vari client che il protocollo Redis supporta; nel nostro caso abbiamo visto i client Python e Java.

Tile38 è un'ottima e veloce applicazione di geofencing e ricerca geo-spaziale, però, limitatamente a semplici utilizzi. Infatti, l'utilizzo e i test praticati con Tile38 riguardano mediamente semplici applicazioni; non è garantito che possieda le stesse prestazioni su sistemi più grandi e complessi. Inoltre, è necessario l'impiego di un database per archiviare tutti i dati. I client Tile38 utilizzano diversi software (Redis,lettuce, ecc...), quindi la dipendenza da essi potrebbe essere una lama a doppio taglio. In particolare, dal momento che Tile38 utilizza nativamente il protocollo Redis Resp, sono presenti alcuni difetti derivati da esso; infatti in seguito sono elencati alcuni difetti:

- Il consumo di risorse di sistema aumenta in modo scalare, infatti, man mano che l'utilizzo aumenta, costi e risorse di sistema possono iniziare a diventare elevati, soprattutto se è configurato per essere memorizzato tutto in RAM.
- I tipi di dati nelle strutture dati di un eventuale database possono essere di molteplici tipi, ma all'interno della coppia chiave-valore, il contenuto viene sempre archiviato come stringa. Redis non ha un buon supporto nativo per l'archiviazione dei dati in forma di oggetto e molte librerie costruite su di esso restituiscono i dati

come stringa, il che significa che è necessario adattare la propria architettura su di esso.

- Effetto collaterale della struttura master-slave: presenza di un solo master centrale con più slave connessi. Tutta la scrittura va al master, che crea più carico sul suo nodo. Quindi, quando il master va giù, l'intera architettura ne risente, rischiando di avere una conseguente perdita di dati.

4.1 Studi futuri

Sarebbe stato interessante vedere ed eseguire test su Tile38 direttamente applicata ad un sistema in un ambiente reale.

4.2 Cosa abbiamo imparato

Grazie allo studio su Tile38, oltre ai metodi di esecuzione e caratteristiche proprie di Tile38, ho imparato meglio i meccanismi di funzionamento e l'importanza dei sistemi distribuiti, inoltre ho appreso diversi strumenti software tra cui docker, maven, redis e lettuce. Complessivamente è stata una esperienza positiva che ha arricchito ulteriormente il mio bagaglio di conoscenze informatiche.

References

[1] Docker. Docker, 2021. [Online].

[2] Google. Go programming language, 2021. [Online].

[3] Lettuce. Lettuce, 2021. [Online].

[4] Maven. Maven, 2021. [Online].

[5] RedHat. Rest api, 2021. [Online].

[6] Redis. Redis, 2021. [Online].

[7] Tile38. Tile38, 2021. [Online].