Factory Linker – Progetto Sistemi Distribuiti

Luca Semprini - 0000854447

luca.semprini10@studio.unibo.it

Jacopo Riciputi - 00000000

jacopo.riciputi@studio.unibo.it

Giugno 2020

Indice

Sommario

Up to ∼2000 characters briefly describing the project.

1 Introduzione

L'obiettivo del progetto è di mostrare le potenzialità di Redis all'interno di un'architettura distribuita a microservizi. Redis è uno store in-memory open-source di strutture dati, usato come database, cache e message broker. Supporta vari tipi di strutture dati, dalle più semplici: stringhe, hash, liste, set alle più complesse: sorted set, bitmaps, hyperloglogs, indici geospaziali e infine gli stream. Gli stream risultano la struttura dati di riferimento all'interno del progetto in quanto, oltre a gestire la propagazione di dati, sono ripercorribili in entrambe le direzioni, perciò utilizzati anche come struttura dati di salvataggio delle informazioni raccolte, prima che queste vengano elaborate e successivamente storicizzate per le analisi a lungo termine. Gli stream in Redis offrono anche la possibilità di gestire vari gruppi di consumatori, il che permette di imbastire un'architettura distribuita basata sul pattern produttore-consumatori. L'idea infatti è di utilizzare Redis come punto di raccolta e propagatore delle informazioni.

2 Obiettivi e risultati previsti

Per rendere pratico tale utilizzo si è pensato di portare oltre all'analisi e lo studio della tecnologia un'implementazione che simuli un ambiente di produzione aziendale. L'idea sarebbe di leggere ripetutamente informazioni offerte da una serie di macchinari (tramite protocollo OPC UA o standard MTConnect, prelevando i dati da fonti realistiche: es. agent mtconnect e/o simulate) per inserirli all'interno di uno STREAM di Redis.
Si vorrebbe inoltre implementare un servizio per la visualizzazione, per l'aggiornamento in tempo reale, per la reazione ad eventi (es. allarme da parte di una macchina), e uno per il monitoraggio e il deploy automatizzato di ulteriori consumatori in caso di necessità, andando perciò a creare un'architettura a microservizi, così da utilizzare gli STREAM offerti da Redis non solo come struttura per il salvataggio dei dati ma anche come strumento di intercomunicazione tra microservizi.
Infine si vorrebbe creare una sorta di suddivisione in reparti(es. Tornitura / Assemblaggio / Logistica), così da avere più servizi attivi (uno per reparto e con la rispettiva istanza di Redis), realizzando visualizzazioni in tempo reale, veloci e dedicate, per poi, periodicamente, inviare i dati di ogni reparto (es. a fine giornata

lavorativa) ad un server centrale con la semplice funzione di storicizzare dati per l'analisi a medio lungo termine.
Il sistema sarebbe perciò riassumibile tramite i seguenti punti:

- **Produttore** Un servizio che si occupa della lettura delle informazioni da parte di uno o più macchinari e del loro inserimento all'interno dell'istanza di Redis associata.
- **Consumatori** Rappresentano i microservizi che si andranno a occupare della gestione e l'analisi dei dati inseriti nel sistema. I servizi interessati saranno i seguenti:
  - di visualizzazione: offrirà un'interfaccia grafica per la visualizzazione in tempo reale dei dati raccolti.
  - di aggiornamento: si dovrà occupare della comunicazione all'interfaccia grafica dei nuovi dati all'interno del sistema.
  - di reazione agli eventi: avrà il compito di monitorare le informazioni prodotte così da rilevare eventuali anomalie in tempo reale.
  - di monitoraggio: controllerà l'ecosistema dei gruppi di consumatori coinvolti e, all'esigenza, dovrà inserire dinamicamente nel sistema nuovi consumatori. Questo con lo scopo di rendere il sistema scalabile, andando, a fronte di un aumento dei dati prodotti, a ridondare automaticamente un servizio.
- **Un server per l'analisi a lungo termine** Si tratta di un nodo centralizzato che permetta l'analisi a medio-lungo termine dei dati raccolti dai nodi installati nei vari reparti.

2.1 Scenari di utilizzo

Lo scenario di utilizzo per il quale è stata pensata l'architettura proposta è quello indicato nella parte introduttiva del progetto: un reparto di un'azienda di produzione nel quale vengono utilizzati uno o più macchinari controllati da CNC con connessione di rete. Generalmente però l'architettura proposta si sposa bene con tutti gli ambiti riconducibile a un'architettura produttore-consumatore, generalmente quindi con molti dei progetti provenienti dal mondo IoT.

2.2 Politiche di autovalutazione

- Verrà fatta una valutazione sulla qualità del software basandosi su alcuni test unitari e di integrazione che verranno sviluppati durante tutto il percorso di crescita del progetto.
- Organizzazione e qualità del codice.
- Semplicità di deploy. Il sistema è pensato per essere installato su un dispositivo autonomo e autosufficiente per rimanere operativo per lunghi periodi, per questo motivo è anche importante che il deploy dell'infrastruttura sia il più semplice possibile.

3 Analisi dei requisiti

3.1 Requisiti funzionali

Il sistema che si andrà a creare dovrà comprendere tutti i servizi identificati e dovrà garantirne una corretta comunicazione durante tutto il ciclo di vita. I messaggi inseriti all'interno degli stream utilizzati rappresentano la fonte primaria di informazioni; è importante perciò che il sistema garantisca una *message delivery semantics* di tipo *at-least-once*. L'ecosistema dei microservizi inoltre entra in contatto con molti concetti e necessità simili, se non identiche, anche se su servizi a se stanti; è perciò importante garantire una buona organizzazione del progetto riducendo al minimo la duplicazione del codice, incrementandone quindi la qualità e offrendo quindi la possibilità a nuovi consumatori di aggiungersi al sistema senza particolari sforzi implementativi.

3.2 Requisiti non funzionali

Tra i requisiti non funzionali del sistema vi sono:

- **Prestazioni**: L'efficienza del sistema si basa principalmente sull'analisi real-time o quasi dei messaggi entrati a far parte dello stream da parte dei consumatori. Garantire perciò che i consumatori siano in grado di prendere in carico i messaggi in breve tempo e riescano ad elaborarli velocemente aumenta notevolmente la qualità e l'efficienza di tutto il progetto.
- **Scalabilità**: Il sistema implementato non deve limitare il numero di produttori, gruppi di consumatori e numero di consumatori per gruppo. Garantendo, indifferentemente dal numero di consumatori, un adeguato monitoraggio ad ogni microservizio con questo ruolo.

3.3 Requisiti tecnologici

Essendo l'infrastruttura desiderata fortemente orientata ai microservizi si è optato a un sistema di containerizzazione di ognuno di esso così da facilitarne la portabilità, l'installazione e il deploy. Per questo motivo ogni microservizio viene compilato

in un container Docker[1] e, per facilitarne ulteriormente l'avvio, si è optato per affidare il deploy al tool Docker Compose[2]. Affidare questa parte al Docker Compose permette non solo di semplificare il bootstrap dell'infrastruttura ma offre anche la possibilità di portare il sistema su più macchine attraverso Docker Swarn[3] che permette di prendere in ingresso i medesimi file di configurazione utilizzati dal Docker Compose tool.

[1] `https://www.docker.com/`
[2] `https://docs.docker.com/compose/`
[3] `https://docs.docker.com/engine/swarm/`

4 Redis

Redis è un database di strutture dati chiave-valore in memoria, rapido e open source. Redis offre una serie strutture dati mantenute in memoria molto versatili, che permettono la creazione di un'ampia gamma di applicazioni personalizzate, trovando spesso un ruolo primario in ambito caching, come database non relazionale o gestore di sessioni. Le sue qualità lo portano a essere il database NoSQL chiave-valore più utilizzato oltre che un punto di riferimento tra le implementazioni in-memory [4]. Redis, infatti, garantisce elevate prestazioni essendo scritto in linguaggio C e mantenendo interamente i dati in memoria principale, comportando perciò un enorme risparmio di tempo data l'elusione dei tempi di risposta più lenti da parte dei dischi secondari e permettendo l'adozione di algoritmi più semplici, con meno richieste di calcolo. La sua natura in-memory, tuttavia, non limita l'utente nelle strutture dati a sua disposizione: Redis offre infatti la memorizzazione dei valori associati alle chiavi secondo varie mappature, fornendo il supporto a strutture dati come: Liste, Set ordinati e non, Hash ed HyperLogLog. Tutto questo senza dimenticare la propria natura NoSQL, il team dietro al progetto, guidato dal suo ideatore Salvatore Sanfilippo [5], ha infatti costruito internamente a Redis la gestione di alcuni elementi chiave dei database non relazionali, come la replicazione master-slave, il supporto alle transazioni, la garanzia di un'elevata disponibilità grazie al progetto Redis Sentinel e la distribuzione geografica su più nodi comunicanti in rete [6].

4.1 Architettura

Redis fa della semplicità un proprio punto di forza, per questo motivo avviare un'istanza di Redis per poter iniziare a lavorare con esso risulta davvero banale e si traduce in un semplice nodo con il quale è possibile dialogare.
Questa soluzione architettonica, tuttavia, può con il tempo rivelarsi limitata per determinati sistemi: infatti è importante ricordare che, anche se Redis ha la possibilità di salvare i dati in memoria, offre le migliori performance mantenendo i dati

[4] https://aws.amazon.com/it/elasticache/what-is-redis
[5] https://github.com/antirez
[6] https://redis.io

in memoria principale, con tutte le limitazioni di spazio che ne derivano oltre a rappresentare un *single-point-of-failure* del nostro sistema.
Per ottemperare a queste problematiche, e quindi fare anche della scalabilità un punto di forza, Redis, rispettivamente alle limitazioni sopraccitate, offre la replicazione dei dati su più nodi (architettura *master-slave*) e lo *sharding* delle chiavi su nodi differenti così da ottenere un cluster di istanze Redis.
È, però, la somma delle due soluzioni a risolvere le limitazioni di scalabilità e disponibilità del sistema, in quanto è comunque possibile istanziare un singolo nodo Master al quale associare uno Slave per garantire *high-availability*, mantenendo però i problemi di poca memoria. Allo stesso modo è possibile suddividere i dati su vari nodi aumentando la disponibilità di memoria senza replicazione, con il rischio che il fallimento di uno di questi porti a disservizi del nostro sistema.

4.1.1 Replicazione

Come anticipato la replicazione dei dati su Redis avviene secondo architettura *master-slave*, infatti a una istanza master di Redis è possibile associare una o più istanze slave alle quali il master propaga, in modalità **asincrona**, le operazioni effettuate sulle chiavi così che i nodi di replica possano rimanere allineati. Nel caso in cui il nodo master rimanga improvvisamente isolato uno dei nodi di replica verrà eletto a nuovo master.
Questa è la panoramica generale che è possibile fare sulla replicazione offerta da Redis e il suo funzionamento più standard; è possibile però guidare alcuni parametri per ottenere comportamenti desiderabili per determinate situazioni di produzione.
È infatti possibile sfruttare la replicazione anche come strumento di **scalabilità** del nostro sistema, non in ottica di spazio ma di prestazioni, in quanto i vari client possono ottenere dai nodi di replica le chiavi desiderate, andando a togliere tempo di CPU all'istanza master. Allo stesso modo è possibile effettuare operazioni di scrittura sui nodi slave, comportamento però che potrebbe essere pericoloso in determinati domini, per questo motivo dalla versione 2.6 di Redis l'opzione *replica-read-only* è abilitata di default, ma nulla vieta di abilitarla per aumentare ulteriormente le performance in scrittura, rischiando però di perdere tali modifiche su altri nodi di replica.

In un ambito di replicazione, soprattutto per sistemi con determinate specifiche può essere necessario assicurarsi che le repliche vengano eseguite su almeno un numero N di istanze di slave. Questo per garantire maggior stabilità del sistema e permette anche di istituire operazioni come quella del *quorum mechanism* [7] per verificare la consistenza del dato.
Redis infatti genera uno stream di dati per propagare le informazioni ai nodi slave in modo tale che questo possa essere consumato dalle repliche che, per mantenere informato il master comunica periodicamente un *heartbeat*. Proprio tramite questo dialogo è possibile stabilire un numero minimo di repliche, infatti Redis per abilitare questa funzionalità richiede di specificare due opzioni:

- min-replicas-to-write - numero minimo di repliche sulle quali scrivere
- min-replicas-max-lag - numero di secondi entro i quali deve essere stavo inviato l'ultimo heartbeat perchè il nodo sia ritenuto attivo

Nel caso in cui queste opzioni vengano rispettate, Redis autorizzerà l'operazione di scrittura, in caso contrario non la completerà restituendo errore al client.
È inoltre importante specificare che, come definito dal CAP Theorem [8] non è possibile garantire Consistenza, Availability e Partizionamento in un sistema distribuito; allo stesso modo nello sviluppo di questa funzionalità di Redis si è dovuto sacrificare uno di questi elementi e, come spesso accade per un sistema NoSQL, viene a meno la parte di Consistenza.
Infatti, essendo la propagazione delle modifiche asincrona, è possibile che una scrittura venga autorizzata ma il nodo che ha preso in carico la richiesta dopo aver risposto positivamente al client rimanga isolato prima che possa inviare la modifica ai nodi in ascolto per le repliche che quindi non avranno la medesima consistenza del nodo master al momento dell'elezione di uno degli slave a nodo principale.
Fare comunque riferimento alla documentazione ufficiale[9] per approfondimenti e ulteriori informazioni.

[7] `https://en.wikipedia.org/wiki/Quorum_(distributed_computing)`
[8] `https://en.wikipedia.org/wiki/CAP_theorem`
[9] `https://redis.io/topics/replication`

4.1.2 Clustering

Come precedentemente indicato la sola replicazione dei dati su nodi di replica non è sufficiente a garantire un livello di scalabilità sufficiente per tutti i sistemi, infatti non si è per niente fatto riferimenti alla scalabilità in termini di spazio e geo localizzazione dei dati, aspetto che negli ultimi anni ha acquisito notevole importanza in determinati dominio applicativi. Si pensi ad esempio ad uno scenario di Smart City e di semafori intelligenti, quindi un dominio che possiamo categorizzare come *hard real-time*[10]: la possibilità di dialogare con una base dati geograficamente vicina a dove le informazioni vengono prodotte e richieste potrebbe fare la differenza in termini di tempi di risposta.
Redis permette di costruire un cluster di nodi al quale è possibile aggiungere o togliere nuove istanze dinamicamente, gestendo automaticamente lo sharding delle chiavi sui vari nodi.
Proprio sulla tecnica di sharding è importante soffermarsi un attimo. Redis implementa una nuova tecnica basata non solamente sull'hash di una chiave ma sull'*hash-slot*, il sistema infatti mette a disposizione 16384 hash slots ai quali vengono associate le chiavi inserite nel sistema con un rapporto tra chiave e hash slot *many:one*. Questo significa che ad una chiave è assegnato un hash slot e ad un hash slot possono essere associate zero, una o più chiavi. Gli hash slot possono essere rappresentati quindi come contenitori di chiavi che vengono distribuiti tra i vari nodi di un cluster Redis e all'inserimento di una nuova chiave in Redis questa viene direzionata verso il nodo con il rispettivo hash slot dove viene elaborata il comando richiesto dal client e in caso di successo salvata all'interno del proprio slot. Si prenda di esempio la messa in produzione di un cluster con tre nodi: A, B, e C; All'avvio Redis distribuirà gli slot in maniera bilanciata tra i nodi:

- da 0 a 5500 nel nodo A;
- da 5501 a 11000 nel nodo B;
- da 11001 a 16383 nel nodo C;

Il cluster è quindi ora in esecuzione e il seguente è il primo comando che viene inviato al nodo A:

[10] `https://stackoverflow.com/a/17309403/8958411`

```
> SET foo bar
```

Viene quindi richiesto di associate alla chiave 'foo' il valore 'bar'. Redis andrà a questo punto a derivare l'hash slot rispettivo alla chiave 'foo'. È possibile ottenerlo anche con il comando:

```
> CLUSTER KEYSLOT foo
```

Al quale verrà risposto con l'hash slot della chiave 'foo' che corrisponde a 12182, il nodo A perciò non può ottemperare alla richiesta e risponderà al client con un messaggio di errore indicando però l'hash slot della chiave e anche l'IP e la porta del rispettivo nodo dell'hash slot indicato.
Risposta da parte del nodo A nell'esempio di riferimento:

```
SET foo bar
–MOVED 12182 IP_NODO_C:PORT_NODO_C
```

In questo modo il client è autonomo nel re-direzionare la richiesta al nodo corretto. È interessante anche capire perché tale operazione non venga direttamente effettuata dal Nodo A, infatti questo comportamento è voluto e non risulta in una mancanza da parte di Redis, che, come indicato fa delle prestazioni un suo punto di forza.
Infatti in questo modo Redis tenta ad indurre il client a un comportamento più intelligente nelle future richieste inerenti alla chiave 'foo'. Il client viene quindi incoraggiato nel mantenere una lista di associazioni chiavi - hash slot cosicché le richieste successive alla prima vengano direttamente effettuate sul nodo corretto risparmiando perdite di tempo nel dialogo tra i nodi.
Nell'esempio di riferimento quindi il client dovrebbe mantenere in memoria la risposta da parte del Nodo A sulla chiave 'foo', in futuro un eventuale comando 'GET foo' per ottenere il valore di tale chiave dovrebbe essere presentato direttamente al Nodo C.[11]
Questo comportamento è importante, facendo ancora riferimento all'esempio del semaforo intelligente, quindi un dispositivo in ambito Smart City con vincoli, per determinate situazioni, *hard real-time* non è accettabile tutto questo *overhead* nel dialogo per ottenere una risposta.
È vero che il semaforo sarebbe in grado già da una seconda richiesta di dialogare con il nodo corretto, ma in un tale dominio applicativo ci si può permettere il rischio di

[11] `https://redis.io/topics/cluster-spec#ask-redirection`

fallire una deadline anche se solo in una casistica? Inoltre questo comunque non ci garantisce che il dato da recuperare sia geograficamente vicino alla sua produzione ed elaborazione, infatti anche nel caso in cui il semaforo conoscesse fin da subito il nodo corretto da cui prelevare il dato è possibile che tale nodo sia geograficamente dall'altra parte del mondo mentre nel cluster è presente un nodo a pochi chilometri di distanza, una tale situazione potrebbe portare a un ritardo nella risposta di diversi millisecondi, se non addirittura di ordini di grandezza più grandi.
Come può allora il sistema assicurarsi che il semaforo dialoghi con il nodo geograficamente più vicino? Per fare ciò è necessario che tutti gli hash slot associati alle chiavi utilizzate dal software risiedano su tale nodo, ma le chiavi potrebbero essere svariate e gli hash slot altrettanti è impensabile spostarli dinamicamente all'occorrenza, sia in termini di praticità che in termini di tempistiche.
Fortunatamente Redis offre un'eccezione nella computazione degli hash slot data una chiave chiamata **hash tags**[12], questi permettono che svariate chiavi vengano associate al medesimo hash slot.
Per inserire un hash tag all'interno di una chiave in Redis è sufficiente specificare un valore fra parentesi graffe, in questo modo verrà considerato solamente il valore tra parentesi nel calcolo dell'hash slot.
In questo modo si ha la garanzia che le chiavi: foo1prova e foo2prova vengano assegnate allo stesso hash slot.

```
> CLUSTER KEYSLOT foo1{prova}
7097
> CLUSTER KEYSLOT foo2{prova}
7097
> CLUSTER KEYSLOT foo2prova
  12369
```

Gli hash tags rappresentano quindi un importante funzionalità in ambito di geodistribuzione dei dati e di vicinanza dei dati, un altro use case interessante infatti è quello di utilizzare i tags per associare allo stesso slot i dati inerenti ad esempio un determinato utente o prodotto così da ottenere *data aggregation*[13].
Come poter quindi tradurre questo comportamento in lato pratico?

[12] `https://redis.io/topics/cluster-spec#keys-hash-tags`
[13] `https://bit.ly/2XD7fmL`

Si pensi a un sistema di Smart City distribuito su due città: Cesena e Bologna supportato da un cluster composto da tre nodi, uno principale del quale non ci interessa la posizione geografica, uno installato a Cesena e uno installato a Bologna. Una soluzione potrebbe essere quella di determinare gli hash slot rispettivamente corrispondenti agli hash tags: Cesena e Bologna, cosicché ogni dispositivo possa utilizzare il tag relativo alla propria posizione ed abbia la certezza che le chiavi vengano posizionate sul nodo desiderato.
Si controllino quindi gli hash slot in esame:

```
> CLUSTER KEYSLOT prova{cesena}
5859
> CLUSTER KEYSLOT prova{bologna}
7280
```

A questo punto non rimane che verificare l'effettiva posizione degli hash slots con il comando CLUSTER SLOTS.
Si faccia riferimento alla distribuzione degli slots presentata precedentemente4.1.2 e si associno i nodi A, B e C rispettivamente al nodo principale dell'esempio, Bologna e Cesena. In un situazione di questo tipo entrambi gli hash slot si troverebbero sul nodo in produzione a Bologna è necessario perciò intervenire.
Connettersi quindi con un client sul nodo di proprietario dello slot, in questo caso il Nodo B ed eseguire la seguente scaletta di comandi[14]:

> CLUSTER SETSLOT 5859 IMPORTING <**source**−node−id>

Con questo comando si dice a Redis che si sta importando lo slot 5859 proveniente dal nodo indicato.

> CLUSTER SETSLOT 5859 MIGRATING <dest−node−id>

Con questo comando si dice a Redis che si sta migrando lo slot 5859 verso nodo indicato. Questo comando deve essere eseguito sul nodo proprietario dello slot, altrimenti restituisce errore.

> CLUSTER SETSLOT 5859 IMPORTING <**source**−node−id>

Conferma dell'operazione ed effettivo spostamento dell'hash slot.

[14] `https://redis.io/commands/cluster-setslot#redis-cluster-live-resharding-explained`

Al termine di questa operazione il sistema si troverà in una situazione ottimale permettendo ai dispositivi di fare riferimento ai nodi a loro dedicati apportando un notevole aumento di prestazioni dovuto anche alla riduzione del delay da parte delle risposte fornite dalla base di dati.

4.2 Stream

Gli **stream**[15] possono essere considerata la struttura dati più complicata di Redis per quanto in realtà venga rappresentata in una modalità davvero semplice.
Infatti, uno stream in Redis è rappresentato come un struttura dati *simil* log file in quanto permette solamente operazioni di tipo *append*, creando perciò una collezione di dati percorribile in entrambi i sensi ma non modificabile se non aggiungendo nuovi elementi.
Vista in questo modo, però, gli stream sono una semplice lista ottimizzata per le letture e l'aggiunta di elementi in coda, cosa rende quindi questa struttura dati così interessante? Ad aggiungere particolare fascino agli stream è il concetto di **consumer group**[16], letteralmente gruppi di consumatori che permettono a vari client di cooperare permettendo loro di consumare differenti porzioni dello stesso stream di messaggi.
Partendo dalle basi di questa struttura dati il primo comando da visualizzare è[17]:

```
> XADD mystream * sensor-id 1234 temperature 19.8
1518951480106-0
```

Questo comando permette di aggiungere una *entry* composta da una mappa chiave-valore allo stream *mystream*. La risposta corrisponde all'ID assegnato all'elemento inserito nello stream, in questo caso autogenerato da Redis dato che nel comando è stato specificato la wildcard: *, altrimenti è possibile guidare l'ID da assegnare alla entry. In questo caso invece, visto l'utilizzo della wildcard, Redis genera autonomamente un ID secondo questo formato:

```
<millisecondsTime>-<sequenceNumber>
```

[15] https://redis.io/topics/streams-intro
[16] https://redis.io/topics/streams-intro#consumer-groups
[17] https://redis.io/commands/xadd

Dove i millisecondi rappresento il tempo locale del nodo che ha generato l'ID e il sequence number è diverso da 0 solo nel caso in cui fosse presente già una entry con il medesimo *millisecondsTime*.
A questo punto diventa possibile interrogare lo stream per ottenere le entry al suo interno, in particolare con il comando XRANGE[18] (oppure XREVRANGE[19], che semplicemente legge lo stream dall'inizio alla fine).

```
> XRANGE mystream - + COUNT 10
1) 1) 1518951480106-0
   2) 1) "sensor-id"
      2) "1234"
      3) "temperature"
      4) "19.8"
2) 1) 1518951482479-0
   2) 1) "sensor-id"
      2) "9999"
      3) "temperature"
      4) "18.2"
  ...
```

Il comando XRANGE restituire tutte le entry comprese tra gli ID specificati. In questo caso sono state utilizzate le wildcards: - e +; ma possono essere sostituiste con dei reali ID appartenenti allo stream. Questo permetterebbe ad esempio interrogazioni dello stream temporali se gli ID sono stati generati da Redis:

```
> XRANGE mystream 1519073279157-1 + COUNT 2
1) 1) 1519073280281-0
   2) 1) "foo"
      2) "value_3"
2) 1) 1519073281432-0
   2) 1) "foo"
      2) "value_4"
```

Il comando sopra infatti interroga lo stream ottenendo tutte le entry inserite successivamente a un dato momento storico.

[18] `https://redis.io/commands/xrange`
[19] `https://redis.io/commands/xrevrange`

Ad aggiungersi a questi comandi c'è XREAD[20], che permette di mettersi in ascolto di nuove entry nello stream così da poter reagire all'inserimento di nuovi elementi. Come anticipato però, per ottenere il massimo risultato dall'utilizzo degli stream è necessario fare riferimento ai gruppi di consumatori. Con il comando XREAD infatti si ha la possibilità di reagire a nuovi elementi nel sistema, ma il comando XREAD non è regolamentano, perciò alcuni messaggi potrebbero essere non elaborati mentre altri ricevuti da più client senza che ce ne sia la necessità. Ci sono però certi ambienti in cui si ha la necessità di scalabilità e controllo sui messaggi per questo è importante aggiungere un qualcosa di più. I gruppi di consumatori rappresentato infatti gruppi di messaggi ai quali per ogni gruppo viene reso disponibile tutto lo stream e le sue nuove entry, che però vengono suddivise tra i consumatori del gruppo. In questo modo è è possibile suddividere il lavoro tra workers differenti.
Un gruppo di consumatori quindi può essere visto come uno pseudo-consumatore che legge i dati dallo stream e spalma i messaggi, quindi il carico di lavoro su consumatori del gruppo differenti[21], garantendo:

- Ogni messaggio è inviato a un consumatore, quindi è impossibile che lo stesso messaggio venga inviato a più consumatori
- I consumatori di un gruppo sono identificati da un nome. Questo significa che anche dopo la disconnessione o l'isolamento di un consumatore il server mantiene comunque nel suo stato l'associazione di un dato messaggio a tale consumatore. Il messaggio perciò sarà consumabile solamente da tale consumatore, a meno che, con un comando dedicato un altro consumatore non rivendichi il possesso del messaggio.
- Ogni gruppo di consumatori ha il concetto di *primo ID mai consumato*, in questo modo alla richiesta di nuovi messaggi verranno consegnati solo messaggi non assegnati ad altri consumatori.
- Il consumo di un messaggio da parte di un consumatore richiede anche un esplicito *acknowledge* per confermare che l'elaborazione del messaggio è effettivamente andata a buon fine.

[20] `https://redis.io/commands/xread`
[21] `https://redis.io/topics/streams-intro#consumer-groups`

- I messaggi che non hanno ricevuto l'ACK vengono considerati in stato di PENDING all'interno del gruppo di consumatori. Tali messaggi possono essere quindi monitorati ed eventualmente attribuiti a consumer differenti nel caso in cui quello originario non sia in condizioni di portarne a termine l'analisi.

4.3 In una architettura a microservizi

Proprio grazie alle potenzialità dei *consumer groups*, Redis può svolgere un ottimo ruolo all'interno di un ecosistema a microservizi, soprattutto in ambito IoT, dove è forte l'architettura produttore-consumatore e dove i produttori possono essere rappresentati dai sensori che si occupano della lettura dei dati.
Redis quindi, in tale architettura, rappresenterebbe un accentratore di informazioni che dà la possibilità a vari servizi di reagire velocemente alla produzione delle informazioni da parte dei sensori.
Si faccia, ad esempio, riferimento all'architettura adottata dal progetto che si sta presentando. All'interno del progetto, infatti, si hanno i produttori: i macchinari e i dispositivi che dialogano con essi che si occupano dell'inserimento delle informazioni in stato *raw* all'interno dello stream di Redis e una serie di consumatori che reagiscono a questi inserimenti, li elaborano e producono rispettivamente un output. Nel caso del progetto infatti sono stati identificati due gruppi di consumatori: *updater* e *event-emitter*.
All'avvio del sistema ad ogni gruppo di consumatori è associato un consumatore che si occuperà di prelevare i messaggi dallo stream e di elaborarli, parallelamente viene però eseguito un ulteriore servizio di monitoring che si occupa di controllare i metadati relativi ai consumatori e ai gruppi di consumatori con lo scopo di:

- Avviare un nuovo consumatore in caso di mancanza di capacità di calcolo;
- Rilevare eventuali consumatori non più operativi, rimuoverli e sostituirli con un nuovo consumatore così da riportare il sistema in uno stato sano, eventualmente assegnando i messaggi pendenti del consumatore morente al nuovo;

Con queste modalità infatti è possibile generare un sistema dinamico capace di interagire velocemente con i dati prodotti e con garanzie che tutti i messaggi

vengano elaborati almeno una volta da parte di un consumatore senza quindi perdita di informazioni.

5 Design

In questa sezione si analizza il sistema Factory Linker sotto le tre dimensioni principali. In primo luogo si discuteranno quali sono i concetti e le entità rappresentate. Successivamente, si andrà ad analizzare il comportamento delle singole entità ed infine verranno presentate le modalità specifiche con cui queste interagiscono.

5.1 struttura

Per la realizzazione della struttura del progetto si è optato per orientarsi verso i **Micro-servizi**. In particolare, il sistema viene suddiviso in sette micro-servizi principali, utilizzati per disaccoppiare le varie funzionalità. In generale, si è pensato di includere all'interno di un package `commons` tutte le logiche e le funzionalità che potessero essere condivise tra più microservizi, quindi riutilizzabili.
Microservizi sviluppati e presenti nel sistema:

- **Reader**: Si occupa del dialogo con i macchinari controllati da CNC;
- **Web Server**: Fornisce un'interfaccia grafica sui dati di produzione registrati;
- **Updater e Event-Emitter (Consumatori)**: Svolgono entrambi il ruolo di consumatori all'interno del sistema. Dovendo ricoprire il medesimo ruolo è stata dedicata al ruolo del consumatore una definizione ad alto livello delle sue funzionalità. Questo ha permesso di implementare entrambi i consumatori con il minimo sforzo e permetterà eventualmente di aggiungerne rapidamente di nuovi (fig. 1);
- **Machine simulator**: Data la difficoltà nel reperire informazioni reali al momento del testing dell'applicazione si è pensato di creare questo progetto così da poter alimentare l'infrastruttura su richiesta;
- **Container Manager**: Il micro-servizio di Container Management ha come compito quello di gestire i container Docker associati ad ogni consumer. Esso si occupa di eliminare e creare alla bisogna i vari container per tutti quei consumer che vengono comunicati essere in difficoltà o inattivi dal Monitor.

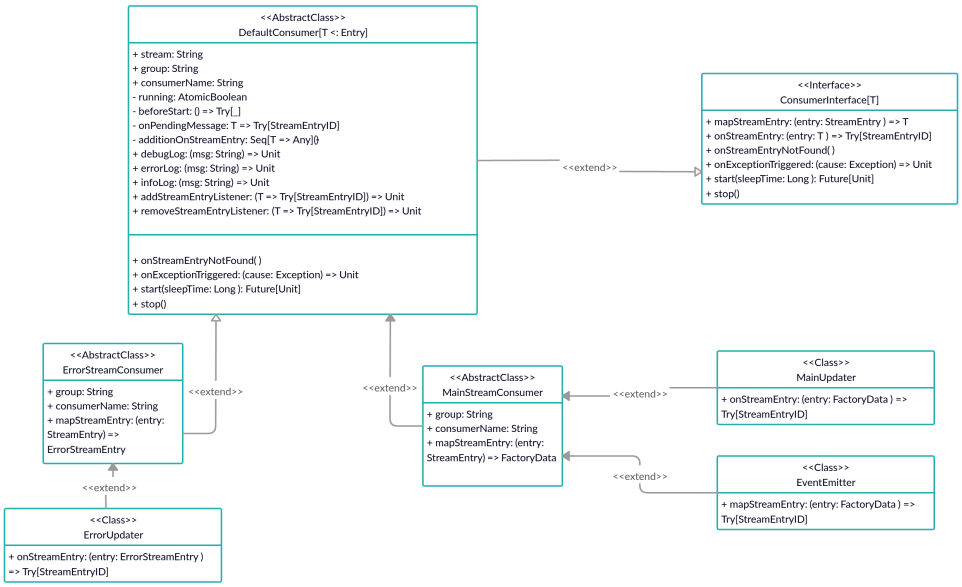

Figura 1: Diagrammi delle classi dedicato all'entità del consumatore

- **Monitoring** Il micro-servizio di Monitoring si occupa di effettuare controlli periodici sui due differenti consumer group all'interno del Redis Stream principale, per individuare eventuali consumer malfunzionanti. Il malfunzionamento viene evidenziato in particolare dall'idle time e dal quantitativo di messaggi pending esposti dal consumer stesso. Questo servizio si occupa, inoltre, di eliminare i consumer considerati malfunzionanti e trasferire tutti i messaggi pendenti ad un nuovo consumer, creato dal Container Manager, in modo tale da far tornare il sistema a regime, rimpiazzando i consumer malfunzionanti.

5.2 Comportamento

Di seguito sono descritte le attività comportamentali di ogni micro-servizio presente all'intern del sistema.

5.2.1 Micro-servizio Reader

Il micro-servizio Reader ha come unico compito il dialogo con i macchinari di produzione. Per portarlo a termine si avvale dell'utilizzo del protocollo MT Connect[22]

[22] `https://www.mtconnect.org/`

per lo scambio dei dati con la singola macchina. È possibile guidare il reader per un dialogo con uno o più macchinari tramite un file di configurazione.

5.2.2 Micro-servizio Web Server

Il micro-servizio Web Server fornisce delle API REST di dialogo con lo stream. Il suo compito è infatti quello rendere disponibili alcune interrogazioni sullo stream per permettere la visualizzazione dei dati raccolti. Il principale utilizzatore infatti di queste API è l'interfaccia grafica sviluppata in React[23] e servita direttamente dal Web Server.

5.2.3 Micro-servizio Updater

L'updater è uno dei consumatori del nostro sistema. Infatti il suo ruolo è di attendere e reagire a nuove *entry* all'interno dello stream principale, con i dati sulle macchine, e sullo stream utilizzato per il logging degli errori alimentato dall'event-emitter ed eventuali altre anomalie rilevate nel sistema Alla ricezione di un nuovo elemento, l'updater ha il semplice compito di notificare eventuali ascoltatori, nel caso del nostro progetto solamente la parte grafica, tramite messaggio su web socket[24].

5.2.4 Micro-servizio Event Emitter

L'event-emitter è uno dei consumatori del nostro sistema. Il suo ruolo, similarmente all'updater, è di attendere e reagire a nuove *entry* all'interno dello stream principale. Alla ricezione di un nuovo elemento, l'event-emitter esegue un controllo sui dati presenti nel messaggio ricevuto ed in caso di rilevazione di anomalie (es. temperatura alta, velocità del rotore elevata, allarmi macchina, ecc..) aggiunge su uno stream secondario in Redis, dedicato solamente agli errori rilevati nel sistema, una nuova *entry*.

5.2.5 Micro-servizio Machine Simulator

Il micro-servizio di simulazione è stato pensato in sostituzione o in aggiunta del reader in fase di testing. Si sono infatti riscontrati a volte problemi nella lettura di

[23] `https://reactjs.org/`

[24] `https://developer.mozilla.org/it/docs/WebSockets`

dati da sorgenti MT Connect in quanto impossibilitati ad acquistarne una fisica e l'unico servizio online disponibile spesso mostrava macchinari in stato di fermo. Il suo ruolo è di produttore di informazioni nello stream, si occupa, infatti, di generare periodicamente dati sintetici di macchine simulate, aggiungendoli allo stream di Redis.

5.2.6 Micro-servizio di Monitoring

Questo micro-servizio permette di monitorare lo stato di salute di tutti i consumatori presenti nei due gruppi di consumatori. La sua esecuzione consiste nel lanciare due istanze separate per lo stesso micro-servizio di monitoring, una per il gruppo updater e una per il gruppo event-emitter; l'istanza del servizio utilizza uno scheduler Akka per poter analizzare periodicamente i dati sullo stream. In particolare, ogni 60 secondi, il micro-servizio richiama il comando di Redis Stream `XINFO CONSUMERS` sullo stream, per ognuno dei due gruppi di consumatori; ottenute le informazioni grazie al comando, il servizio filtra solo quei consumatori che risultano **morti** o **in difficoltà**, secondo queste specifiche:

- consumatore morto: possiede un idle time superiore ad una soglia pre-stabilita, ovvero risulta irreperibile da un certo periodo di tempo, che viene considerato critico.
- consumatore in difficoltà: possiede un idle time inferiore alla soglia di criticità, ma il numero dei suoi pending messages risulta superiore ad un'altra soglia arbitraria che lo etichetterà come, appunto, consumatore in difficoltà.

Le informazioni per ogni consumatore vengono wrappate in una mappa, che viene spedita nel body di una chiamata post, per essere poi processata dal Container Manager. Ricevuto il risultato dal container manager (ovvero il nome del nuovo consumatore, creato per rimpiazzare quello risultato morente), il Monitor si occupa, tramite il comando di Redis Stream `XPENDING` di raccogliere i messaggi pending dal consumer morente, e, tramite il comando `XCLAIM` iniettarli nel nuovo consumatore creato, di modo da permettere a quest'ultimo di poter recuperare questi messaggi e riprendere il lavoro interrotto.
A conclusione del ciclo della nostra infrastruttura il servizio di monitoring ha anche il compito di scannerizzare periodicamente entrambi gli stream, pulire la sorgente

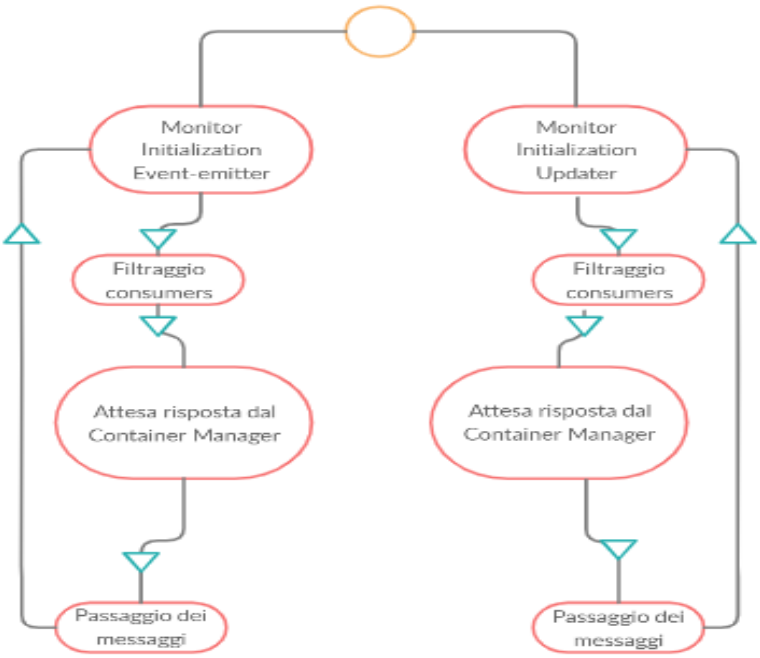

Figura 2: State diagram che illustra il comportamento interno del micro-servizio di Monitoring.

dati grezza e storicizzarla su un server S3 di Amazon in formato CSV per l'eventuale analisi con strumenti di BigData.

5.2.7 Micro-servizio Container Manager

Questo servizio garantisce la gestione dei container docker relativi ai consumatori. Esso si appoggia ad un Server, che resta in ascolto di chiamate API Rest da parte del servizio di monitoring, per prendere atto dei consumatori morti o in difficoltà e comunicarli allo stesso container manager. In primo luogo, l'esecuzione si occupa di scaricare, se non presenti, le immagini docker relative ai gruppi di consumatori event-emitter e updater; solo una volta salvate, viene lanciato il Server. Il container manager, ricevuto il nome di un consumatore e il suo stato, genera un nome univoco per il nuovo container che sta per essere creato, uccide e cancella il container morente ricevuto se esso risulta associato ad un consumer morto o se semplicemente fa parte del gruppo **updater** (in quanto di quest'ultimo gruppo si vuole solo una istanza di consumatore all'interno del sistema). Se il consumatore è semplicemente in difficoltà, non viene cancellato il container, ma si passa direttamente alla fase successiva: la creazione di un nuovo container per il consumer da affiancare a quello esistente nel caso questo sia in difficoltà, o da andarlo a sostituire nel caso risulti morto. La creazione del nuovo container avviene previa connessione alla network condivisa.

5.3 Interazione

Di seguito vengono descritte le interazioni fondamentali tra micro-servizi, accompagnate da alcuni diagrammi UML di sequenza, per rappresentare appieno le dinamiche in gioco.

5.4 Interazione Reader-Macchinari

Il servizio Reader esegue periodicamente una richiesta HTTP a tutti i macchinari configurati attendendosi per ognuna di esse una risposta in formato XML con lo stato corrente delle macchina. Una volta ottenuta la risposta tramite interrogazioni XPath ricava i dati necessari ed esegue la produzione di un elemento all'interno dello stream.

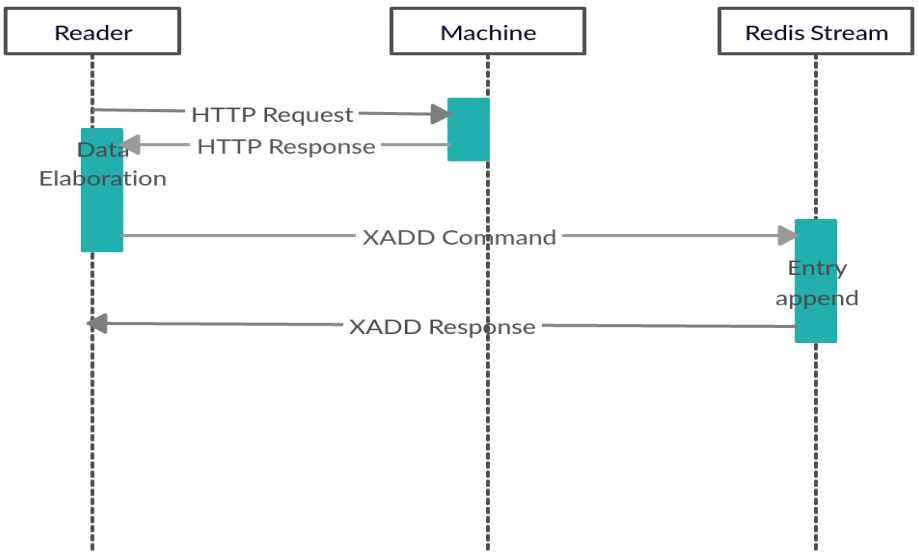

Figura 3: Diagramma di sequenza che illustra il comportamento del servizio Reader nel dialogo con il singolo macchinario

5.5 Interazione consumatori

L'interazione dei due consumatori nel sistema è accomunata nella loro fase iniziale per poi differire nella fare di reazione agli eventi. Come possibile infatti vedere dalle figure 4 e 5, rispettivamente inerenti al comportamento dell'Updater ed Event-Emitter, entrambi i servizi eseguono una richiesta bloccante di nuovi messaggi da parte dello stream con il comando XREADGROUP.
Una volta però ottenuti i nuovi elementi da elaborare i loro comportamenti si differenziano:

- Updater: Semplicemente formatta il messaggio e lo notifica tramite WebSocket, indipendentemente dal contenuto.
- Event-Emitter: Controlla il contenuto del messaggio e in caso di anomalia esegue un inserimento all'interno dello stream dedicato agli errori.

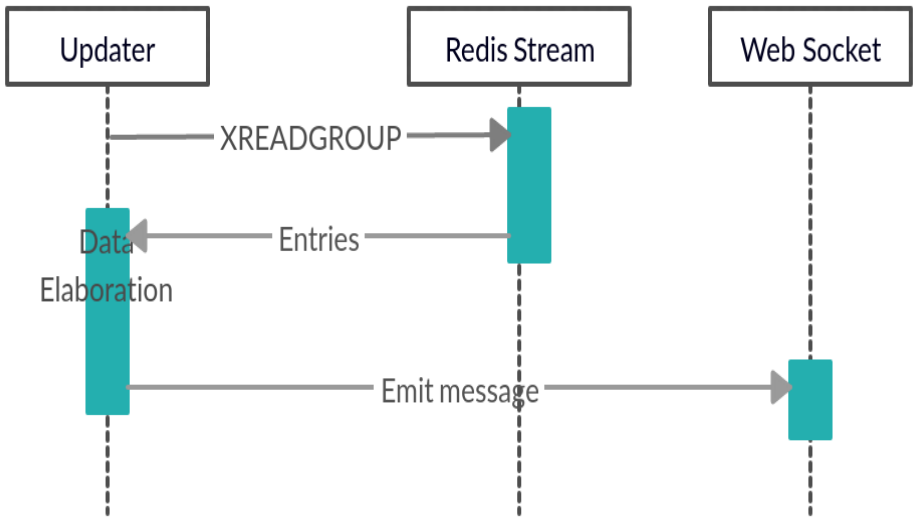

Figura 4: Diagramma di sequenza che illustra il comportamento del servizio Updater

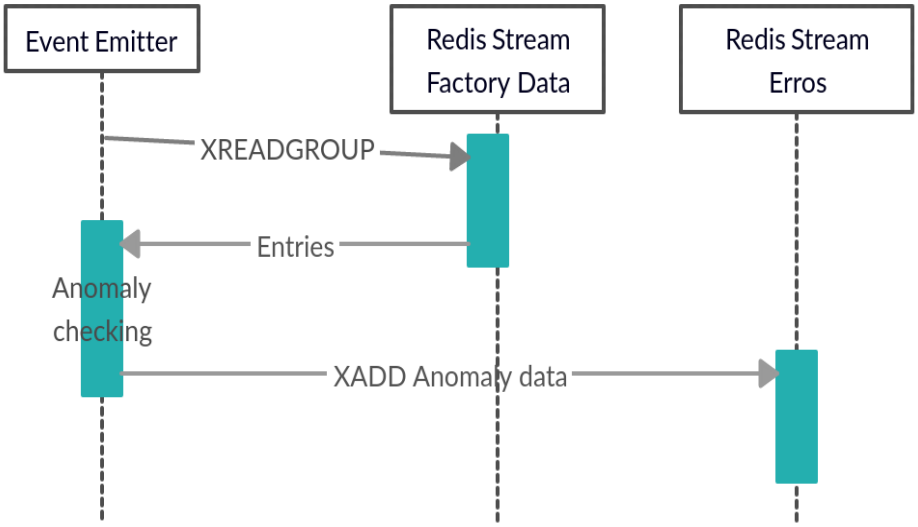

Figura 5: Diagramma di sequenza che illustra il comportamento del servizio Event-Emitter

5.5.1 Interazione Monitor-Container Manager

Come accennato in precedenza, i micro-servizi di Monitoring e di Container Managing, interagiscono costantemente, scambiandosi informazioni relative ai consumatori individuati come morti o in difficoltà. Individuato il consumatore in questione (del gruppo updater o event-emitter), il Monitor si occupa di comunicare al Server relativo al Container Manager tutte le informazioni necessarie, tramite API Rest, spedendo nei parametri della chiamata POST il nome del consumatore e i suoi parametri vitali (quindi stato di difficoltà o morte). Il container manager si appoggia al Server per ricevere le suddette informazioni, sfruttandole per gestire i container relativi al consumatore comunicato di volta in volta. Una volta creato il nuovo container, le informazioni necessarie al Monitor, ovvero il nome del nuovo container, vengono inserite all'interno della risposta alla richiesta. Ottenuta questa informazione, il Monitor può procedere al recupero dei messaggi pending e rendere operativo il nuovo container, nella maniera descritta in 5.2.6.

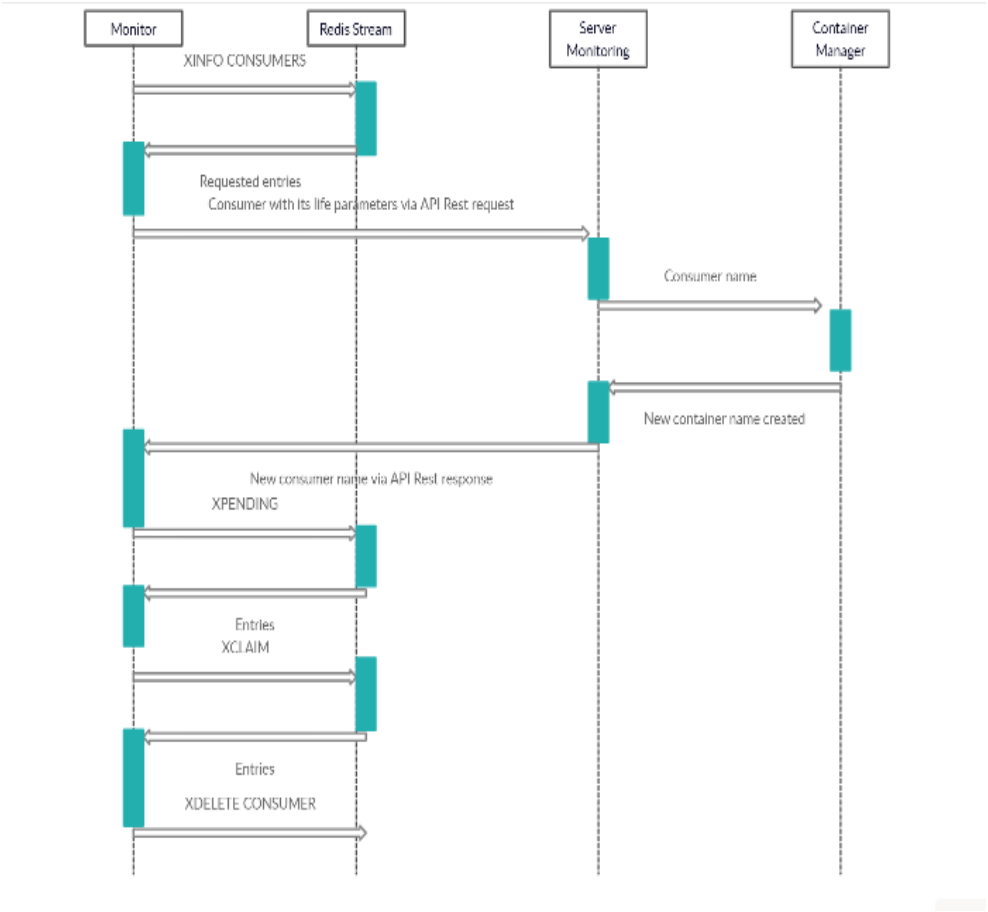

Figura 6: Sequence diagram che illustra le interazioni tra micro-servizio di monitoring, micro-servizio di container managing (con relativo Server) e Redis Stream.

6 Dettagli di implementazione

Uno degli obiettivi del progetto era di creare un ecosistema comune di funzionalità accessibili dai vari microservizi così da accomunare varie implementazioni su in un unico punto del codice, in questo caso il package *commons*.
Una particolare nota implementativa va fatta sull'utilizzo della CI di GitLab che svolge un ruolo fondamentale per il sistema creato. La CI infatti è stata configurata per, oltre che effettuare il build del progetto ed eseguire i test, compilare i container di tutti i vari servizi e posizionarli all'interno del Container Registry offerto da GitLab. I container generati dalla CI verranno poi utilizzati dal Docker Compose in fase di deploy del sistema.
Di seguito vengono riportati i principali framework e librerie utilizzate all'interno dell'applicazione, associati al loro scopo e alla motivazione del gruppo nel merito della scelta di alcune librerie specifiche.

- **Vert.x Client**: Framework utilizzato alla base dell'object RestClient nel package commons che è il riferimento dei vari microservizi per l'esecuzione delle chiamate HTTP.

- **Vert.x Server**: Si è optato per questo framework per la creazione dei servizi HTTP vista la sua velocità di apprendimento ed integrazione. Per comodità è stato *wrappato* all'interno delle classi del package *commons.api.** per semplificarne ulteriormente il riutilizzo.

- **ClientRedis con Jedis**: si è scelto di utilizzare la libreria Jedis per interfacciarsi con le chiamate a Redis tramite codice Scala. Si è scelta questa libreria in quanto possiede ad oggi la stragrande maggioranza delle funzionalità di Redis, ed, in particolare, risultava fornire le interfacce per utilizzare i comandi di Redis nell'ambito degli Stream, cosa fondamentale all'interno del nostro progetto.

- **Docker-Client**: per interfacciarsi con il sistema Docker, si è scelto di utilizzare la libreria `docker-client` di **Spotify**, in quanto è sembrata essere l'alternativa più completa ed intuitiva, nell'ambito delle operazioni che ci interessavano. Essa ci ha permesso di scaricare immagini Docker, istanziare, uccidere e cancellare container alla bisogna, trasferire variabili d'ambiente, connettere container a una network tramite codice Scala, in tutta trasparenza.

- **Akka Scheduler**: la libreria Akka è stata utile nel momento in cui ci si è trovati nella necessità di dover impostare uno Scheduler per alcune parti dell'applicazione (Monitor, Reader), che necessitavano di un controllo per svolgere operazioni periodicamente. Avendo già utilizzato Akka in altri progetti passati, e conoscendo la libreria, ci è sembrata la scelta giusta.
- **MT Connect**: Protocollo utilizzato dall'applicativo per lo scambio dei dati con le macchine
- **React per web client**: il front-end web dell'applicazione è stato implementato in React, un framework già noto dai componenti del gruppo per realizzare Single-Page-Applications: in questo caso abbiamo pensato potesse fare al caso nostro.

7 Autovalutazione / Validazione

Nella sezione 2.2 abbiamo introdotto i nostri criteri per l'autovalutazione. In particolare volevamo per creare un sistema ad alta **semplicità di deploy**, pensato per essere installato su un dispositivo autonomo e autosufficiente, e che assicurasse alcune funzionalità di base con quanti più possibili test unitari e verificare la semplicità di sviluppo del caso di studio. L'integrazione con la CI di GitLab ci ha permesso di simulare un livello di CD a nostro parere fondamentale nel caso in cui il nostro fosse un progetto realmente commissionato.
Durante la creazione, la correzione e lo sviluppo, in generale, del progetto, abbiamo riscontrato molta efficienza e semplicità d'uso nel sistema di deployment, che ci ha aiutato a testare live le modifiche apportate di volta in volta e a correggere gli errori; questo ci ha fornito in partenza un giudizio informale positivo di una delle qualità che esigevamo dal nostro software.
Per quanto riguarda i **test**, siamo riusciti ad implementare test units relative ai micro-servizi di **Updater** e **Machine Simulator**: nel primo caso c'era la necessità di verificare il corretto funzionamento del servizio, svolgendo esso un ruolo fondamentale, proprio per questo sono stati sviluppati anche dei test di integrazione con il microservizio **Event-Emitter**. Mentre i test implementati per il micro-servizio di machine simulation, sono serviti a rafforzare quanto più possibile un servizio essenziale esso stesso in fase di testing delle varie funzionalità dell'applicazione. Infatti, come accennato, molte delle funzionalità sono state testate fondamentalmente effettuando dei lanci di vari micro-servizi mirati ad evidenziare eventuali falle e mettere in luce quelle che erano le feature rilevanti in ogni momento dello sviluppo.
Infine, ma non meno importante, la **qualità del codice** è stato un principio fondamentale durante l'intero processo di sviluppo, a cui ci siamo voluti rifare dall'inizio, per rendere il lavoro il più pulito e leggibile possibile. Abbiamo cercato di riutilizzare più codice possibile, ordinando in maniera funzionale i vari files, e mantenendo un package `commons`, utile per condividere parti di funzionalità condivise tra più micro-servizi. Inoltre, abbiamo avuto cura di documentare quanto più possibile il lavoro fatto giornalmente, in modo da permettere ad ogni membro del gruppo di integrare, se necessario, con semplicità il codice scritto dall'altro.
Alla luce di tutte queste considerazioni, ci riteniamo complessivamente molto

soddisfatti di questo progetto da una prospettiva di progettazione del software, nonostante sicuramente si sarebbero potuti implementare più test unitari, che avrebbero completato il quadro del testing delle funzionalità più core.

8 Istruzioni per il deploy

Per eseguire il deploy è necessario innanzitutto installare Docker[25] e il tool Docker-Compose[26].
Eseguire poi il login all'interno del registry di GitLab:

```
docker login registry.gitlab.com
```

Una volta ottenuti, è necessario procurarsi il file *docker-compose.yml* all'interno della cartella **docker/compose/latest**, essendo configurato per eseguire i container ottenuti dalla build del ramo **master** del progetto.
Per un corretto avvio, è necessario anche creare due file nella medesima cartella in cui è posizionato il file docker-compose.yml che si sta eseguendo:

- environment.env: File con le variabili d'ambiente comuni nel sistema. Esempio di file con il minimo delle configurazioni:

  ```
  REDIS_HOST=redis
  REDIS_PORT=6378
  REDIS_PW=your_db_pw
  ```

- redis.conf: File con le configurazioni di Redis:

  ```
  port 6378
  requirepass your_db_pw
  ```

Infine è possibile avviare il sistema con il comando:

```
docker-compose -f path/to/docker-compose.yml up
```

È comunque possibile trovare maggiori dettagli per la parte di deploy all'interno del file README del progetto.

[25] https://docs.docker.com/get-docker/
[26] https://docs.docker.com/compose/install/

9 Esempi di utilizzo

Accedere tramite browser all'indirizzo: *http://localhost:4700* per visualizzare l'interfaccia grafica.
Per simulare il disservizio di un consumatore di tipo **event-emitter** è sufficiente seguire i seguenti steps:

1. Forzare lo stop del consumatore in esecuzione con il seguente comando:

   ```
   docker stop latest_event_emitter_1
   ```

2. Aprire una shell di Redis:

   ```
   redis-cli -p 6378
   ```

3. Autenticarsi:

   ```
   auth your_db_pw
   ```

4. Aggiungere una entry anomala nel sistema:

   ```
   XADD factory:stream *
   ControllerMode MANUAL
   Program PROGRAM-8W
   ToolNumber 0
   machine Texas
   RotaryVelocity 0.0
   Block 0
   Execution ACTIVE
   PartCount 7
   RotaryTemperature 1294.0
   ```

A questo punto non rimane che attendere che il servizio di monitoring si accorga della mancata connessione da parte del consumatore e ne avvii uno in sostituzione. Attendendo questo processo posizionandosi sull'interfaccia grafica ci si renderà conto della creazione di un nuovo microservizio non appena verrà notificata un'anomalia del sistema, tale segnalazione infatti viene scatenata dall'intervento del nuovo microservizio, che nel frattempo ha recuperato i vecchi messaggi e li ha scansionati.

10 Conclusioni

Possiamo ritenerci soddisfatti del risultato ottenuto con questo elaborato, in quanto siamo riusciti a raggiungere i nostri obiettivi e soddisfare i requisiti. A inizio progetto eravamo consapevoli che lo sforzo sarebbe stato notevole perché saremmo entrati nel dettaglio di un'architettura che ha al suo interno tanti elementi che entrano in contatto tra di loro ed effettivamente ci siamo a volte trovati in difficoltà nell'esecuzione dei controlli di qualità dei risultati desiderati, ma al termine dello sviluppo, come evidenziato precedentemente ci riteniamo soddisfatti del risultato ottenuto.

10.1 Lavori futuri

I primi passi di uno sviluppo futuro potrebbero essere sicuramente orientati verso l'inserimento di nuovi gruppi di consumatori e dei rispettivi consumer. Tra le varie idee potrebbero essere creati consumatori:

- con il compito di prevenire la rottura tramite algoritmi di Data Mining in *simil real-time*
- per un'integrazione con l'ERP aziendale
- per alimentare il successivo processo di produzione

10.2 Cosa abbiamo imparato

L'esperienza di questo progetto ci ha dato l'opportunità di approfondire tematiche ancora inesplorate o poco conosciute. Abbiamo avuto l'opportunità di confrontarci a tutto tondo con il paradigma architetturale a micro-servizi, molto utilizzato nelle applicazioni reali; abbiamo inoltre, potuto analizzare ed utilizzare diverse librerie e vari framework che ci sarebbero potuti essere utili, facendo vari confronti nel merito del nostro ambito di utilizzo. In particolare, è stato molto interessante confrontarsi con una tecnologia come Docker, ed apprezzarne le funzionalità; è stato, inoltre, interessante riuscire a mettere in pratica le conoscenze pregresse che il gruppo aveva nell'ambito del framework Vert.x, applicandole ad un ambito nuovo. Un altro aspetto formante dell'esperienza dell'elaborato è stata l'opportunità di

utilizzare le conoscenze nell'ambito di Redis all'interno di un'applicazione reale. Ultimo aspetto, ma non meno importante è sicuramente il fatto di aver avuto la possibilità di lavorare in team, seppur di dimensione ridotta, che ci ha spinto a cercare di ottimizzare il workflow in questo senso, lavorando in maniera agile-like, con una board di issues e riunioni molto frequenti.