POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Informatica

Tesi di Laurea Magistrale

Architettura a Microservizi On-Demand

Soluzione basata sulle specifiche di archiviazione ADI

Relatore
Prof. Giovanni Malnati

Candidato
Federico Lecchi
Matr. s267627

Anno Accademico 2021-2022

Summary

Negli ultimi anni la tecnologia ha permesso di aver diversi tipi di intercettazioni legali. Tra queste si annoverano tre macro-classi: telefoniche, ambientali e infine telematiche. Le intercettazioni telefoniche si basano sulla registrazione a distanza delle conversazioni telefoniche tramite la duplicazione della linea telefonica dell'intercettato da parte dei provider telefonici. Quelle ambientali si riferiscono alla registrazione di tutto ciò che si può udire all'interno di un ambiente tramite l'utilizzo di microspie (in gergo cimici). Infine, quelle telematiche intercettano tutto ciò che passa tramite il canale internet e non solo, sfruttando l'iniezione di *spyware,* software che vengono installati sul *device* dell'utente a sua insaputa e che vanno a registrare tutto ciò che avviene al suo interno. Questo rende possibile visualizzare tutti i processi in esecuzione sulla CPU, la copia degli album multimediali, della rubrica e lo storico della navigazione del browser.

Per la giurisdizione italiana queste prove devono essere raccolte e mantenute all'interno di un archivio digitale, presente all'interno di ogni procura (ADI, Archivio Digitale Informatico). Tutte le prove raccolte vengono salvate all'interno di dispositivi eterogenei (CD, HDD, DVD) e devono sottostare a requisiti di crittografia e visualizzazione ben specifici. E' necessario, infatti, che i dati salvati all'interno dei supporti fisici sopraelencati siano cifrati con opportuni algoritmi e chiavi di una certa lunghezza, come descritto dalle specifiche ADI. La visualizzazione, invece, può avvenire solamente in appositi locali all'interno delle procure attraverso elaboratori che rispettino standard rigidi. (es. non permettano la connessione a Internet).

Come si può immaginare, le intercettazioni hanno dimensioni assai grandi, anche per il fatto che nella maggior parte dei casi si prolungano per settimane se non mesi, portando ad accumulare Terabyte (1 Terabyte = 1000 Gigabyte) di informazioni. Queste intercettazioni fatte dalla polizia giudiziaria devono poi essere conferite alle procure su dei supporti (CD, DVD, ecc.). Sarà poi compito delle singole procure mantenere i supporti dentro all'archivio delle intercettazioni (camere blindate), in attesa che avvocati o pubblici ministeri vogliano visionarle.

La procedura di conferimento delle intercettazioni alle procure si articola principalmente nella fase di cifratura e di validazione. Oggetto della tesi è stata l'esaminazione e la rivisitazione dell'architettura che svolge le fasi appena citate. Al giorno d'oggi questi

software, che debbono elaborare ingenti quantità di file, si trovano a dover lavorare per giorni e giorni, a volte settimane, senza sapere quando verrà terminato il processo.

Tramite il lavoro di tesi si è voluta trovare una soluzione alternativa per migliorare le tempistiche dei conferimenti e avere informazioni sull'andamento del processo in tempo reale. Per fare ciò è stata utilizzata un'architettura distribuita che utilizzasse microservizi e non più una soluzione monolitica come quella utilizzata tutt'ora. Questo ha permesso di avere un'architettura i cui servizi abbiano una certa resilienza e possano essere pilotati dall'esterno. Gli orchestratori di container come Kubernetes e i *message broker* come NATs hanno permesso di ottenere buoni risultati nella separazione delle procedure su applicativi autonomi, gli uni rispetto agli altri. Inoltre, si è riusciti ad ottenere una parallelizzazione delle procedure scalando orizzontalmente, con l'utilizzo di più *container* in parallelo. Per scalamento orizzontale viene intesa la divisione del carico di lavoro su più *container* (o tecnologie similari) in parallelo, mentre con lo scalamento verticale si va a indicare l'incremento prestazionale dell'hardware del singolo server.

Indice

Elenco delle figure

Capitolo Primo

Capitolo 1

1 Ambito del lavoro

1.1 Introduzione

Le intercettazioni sono da sempre un importante mezzo per ottenere informazioni riguardanti attività illecite che non si sarebbero potute acquisire in altri modi. Le intercettazioni, come vedremo nei paragrafi successivi, possono essere di vario tipo: ambientale, telematico e telefonico. In particolare, possiamo osservare come quelle telematiche abbiano acquisito una certa rilevanza con l'utilizzo sempre più diffuso di pc, tablet e smartphone.

E' uso comune oramai scambiarsi una grossa quantità di dati, da fotografie a video anche in alta risoluzione. Anche il mondo della telefonia si sta spostando sulla rete Internet, questo perché, avendo connessioni sempre più performanti e con una banda maggiore, si riesce ad avere una qualità pari a quella di una chiamata cellulare.

Tutte le informazioni che vengono intercettate devono essere poi gestite e mantenute in modo tale da poter essere visionate durante periodi seguenti dagli enti preposti. Questo procedimento che, partendo dallo svolgimento delle intercettazioni porta poi alla loro scrittura su supporti fisici (CD, DVD, HDD), prende il nome di conferimento. L'aumento delle evidenze e delle loro dimensioni ha avuto un effetto di rallentamento nel processo di creazione dei conferimenti. Infatti, sono venuti alla luce i problemi che presenta il software utilizzato tutt'ora quando ha a che fare con grandi moli di dati.

Con questo lavoro di tesi si è cercato di trovare delle soluzioni architetturali che, sfruttando i nuovi paradigmi di programmazione distribuita, di divisione in microservizi, permettessero lo svolgimento del processo di conferimento in modo più rapido e diramato su più entità.

Per ottenere un ambiente distribuito sono state utilizzate diverse tecnologie, dai *message broker* agli orchestratori di *container*. Nei seguenti capitoli verranno analizzate le tecnologie utilizzate e l'infrastruttura creata. Infine, verranno elencati i risultati ottenuti con diversi approcci a seconda delle risorse computazionali che vengono messe a disposizione.

1.2 Procedura di conferimento

Di seguito verranno descritti i passi precedenti che portano al conferimento, menzionati per avere un'idea del flusso e del contesto in cui si opera, oltre alle fasi del conferimento stesso, che andranno calate all'interno della nuova architettura.

1.2.1 Intercettazioni legali

Le intercettazioni, autorizzate dal giudice su richiesta del PM[1] per le indagini preliminari, possono essere di tre tipi: ambientali, telefoniche e telematiche.

Quelle telefoniche vengono attuate da parte del provider telefonico, dopo che sotto la richiesta del PM, duplica la linea telefonica in maniera totalmente trasparente a colui che è sotto intercettazione. Questo metodo è sempre meno utilizzato, infatti con il progredire della tecnologia sono sempre di più le chiamate che avvengono tramite applicazioni VoIP[2] (WhatsApp, Viber, ecc.) a discapito di quelle sulle reti fisse e cellulari.

Vi sono poi le intercettazioni ambientali, che necessitano la previa disposizione di microspie negli ambienti che si vogliono monitorare. Infine, quelle telematiche dove viene messo sotto osservazione tutto ciò che viene scambiato tramite Internet e altri mezzi informatici.

È opportuno sottolineare che le intercettazioni telematiche hanno iniziato a prendere piede con la diffusione esponenziale dei *device* tra la popolazione. Per queste vengono utilizzati programmi chiamati *spyware* [1]. Come già detto all'interno dell'abstract, questi programmi riescono a copiare una grande quantità di informazioni dal *device*. Tutto ciò è permesso dai bug di sicurezza che presentano il sistema operativo oppure le applicazione scaricate al suo interno e che quindi vanno a fornire un punto d'ingresso per gli stessi.

1.2.2 Archiviazione ADI

Dopo l'esecuzione dell'intercettazione da parte della PG[3], inizia il vero e proprio processo di conferimento che terminerà con l'archiviazione delle stesse all'interno dell'ADI. Prima di passare a spiegare le fasi del conferimento, è bene definire cosa sia l'ADI e quale sia la sua funzione.

[1] Pubblico Ministero.
[2] Voice over Internet Protocol, è il protocollo che gestisce il passaggio della voce sulla rete Internet.
[3] Polizia Giudiziaria.

In Italia le intercettazioni e tutti i contenuti a esse collegati, come ad esempio le trascrizioni sommarie, i verbali ecc. confluiscono all'interno dell'ADI, acronimo di Archivio Digitale Informatico, all'interno di supporti di memorizzazione.

Nello specifico, in ogni procura della Repubblica è presente un locale adibito alla conservazione di questi supporti e uno utilizzato per la fruizione degli stessi. Quando un avvocato, un PM o un ente terzo farà richiesta di visionare un insieme di prove, gli verrà fornito il supporto richiesto da visionare all'interno della procura stessa.

I supporti che vengono creati devono rispettare dei vincoli stringenti per mantenere la privacy di chi viene intercettato. Infatti, una parte del procedimento di conferimento si articola proprio nella cifratura dei dati presenti all'interno rendendone impossibile l'utilizzo, se non durante il periodo di fruizione, dove i dati rimangono in chiaro.

1.2.3 Conferimento

Il conferimento è composto principalmente da due fasi: cifratura e validazione. Per la creazione del codice che implementa queste fasi e per le caratteristiche che devono possedere i vari algoritmi di cifratura mi sono basato sulle specifiche che vengono fornite dal Ministero di Giustizia.

1.2.3.1 Fase di cifratura

Prima di entrare nella descrizione dei passaggi che compongono la procedura di cifratura, è bene spiegare il motivo per cui verranno combinati assieme due algoritmi di cifratura, quello simmetrico e quello asimmetrico, lasciando al capitolo seguente l'onere di entrare nei dettagli di come funzionino i singoli algoritmi.

Per capire il motivo della scelta è necessario sapere che gli algoritmi a chiave asimmetrica sono computazionalmente molto pesanti, mentre quelli a chiave simmetrica sono più leggeri. Nel caso delle comunicazioni su rete internet (quando si deve creare un canale sicuro[4]), succede che i dati da cifrare siano veramente tanti e l'utilizzo di una chiave asimmetrica andrebbe ad appesantire in modo significativo il carico degli *end-point* che devono cifrare/decifrare. Quindi il *trade-off* che si utilizza è quello di cifrare i dati, che sono tanti, sempre con la stessa chiave simmetrica, mentre lo scambio di questa chiave simmetrica verrà

[4] Il canale sicuro è quello che viene creato da due end-point che a livello trasporto cifrano i dati che passano, si contrappone alla cifratura che viene fatta a livello applicativo.

fatto cifrandola con la chiave asimmetrica (in particolare con quella pubblica di chi si trova all'altro estremo).

Lo stesso accade nella procedura di conferimento, dove i dati dei file iso e zip (che verranno descritti al termine di questa spiegazione preliminare) sono cifrati con la chiave simmetrica, mentre questa stessa chiave simmetrica viene cifrata a sua volta in modo asimmetrico tramite la chiave pubblica della procura in cui andrà a finire il conferimento. In questo modo solamente la procura stessa, che possiede la chiave privata, potrà decifrare e ottenere la chiave simmetrica con cui poi tornerà ad avere i file in chiaro. In Figura 1 la parte evidenziata di arancione indica la fase di cifratura con la chiave simmetrica, mentre la fase di cifratura e decifratura con la chiave asimmetrica è evidenziata in rosso.

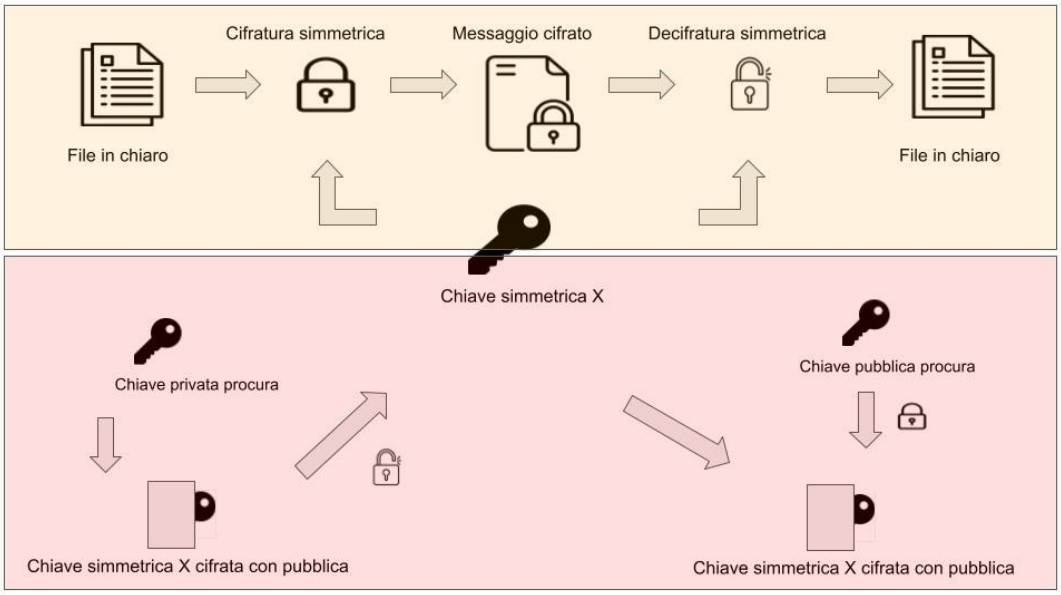


Figura 1: Processo di combinazione di algoritmi di cifratura

Prima ancora di descrivere i file che troveremo all'interno della cartella, verrà brevemente spiegata la tipologia di questi file: il file con estensione .iso e quello con estensione .zip.

I file ISO [2] sono dei file immagine che contengono l'intero contenuto di un archivio e possono essere montati come i dischi. I file ZIP [3] sono archivi compressi che contengono al loro interno altri file.

Dopo questa delucidazione iniziale, si può passare alla descrizione vera e propria della prima fase del conferimento, ovvero quella della cifratura. In questa fase si lavora sulla singola *folder* (così viene chiamata all'interno del documento che riguarda le specifiche ADI che è l'unità di conferimento). Questa *folder,* che per comodità chiameremo cartella, presenta al suo interno quattro file in chiaro:

- Un file ".iso", che contiene al suo interno il visualizzatore e i dati da visualizzare.
- Un file ".zip", che conterrà i file xml[5] che contengono i metadati che si riferiscono ai dati all'interno del file ".iso".
- Un file contenente una passphrase[6] (precedentemente scelta e che rispetti dei canoni di sicurezza).
- La chiave pubblica della procura (la procura possiede una chiave asimmetrica, che è composta da una pubblica e una privata, come verrà spiegato nel capitolo successivo inerente alle tecnologie utilizzate).

Il frutto dell'elaborazione, quindi i vari file cifrati, verranno salvati all'interno di una cartella di output, creata dal software prima dell'inizio di tutta l'elaborazione. Di seguito, la descrizione verrà accompagnata da pseudocodice per una migliore comprensione dei vari passaggi in cui si svolge la procedura di cifratura.

I. Il primo file a essere cifrato è la passphrase (utilizzata per la cifratura simmetrica dei file iso e zip) che viene cifrata con la chiave pubblica della procura a cui fa riferimento il conferimento.

```
1: user_password.txt = validatePassword( userPassword )
2: str = fileRead( user_password.txt )
3: K_sim.enc = Encrypt( K_pub, str )
```

Inoltre, per evitare la manomissione dei file si utilizza la tecnica del *digest* (spiegata nel capitolo seguente) andandolo a calcolare sul file K_sim.enc

```
1: string_hash = hash_256(K_sim.enc)
```

II. Calcolo dell'hash sui file iso e zip in chiaro

```
1: file_stream_iso = open_file(file.iso)
2: string_hash_iso = hash_256(file_stream_iso)
3: file_stream_zip = open_file(file.zip)
4: string_hash_zip = hash_256(file_stream_hash)
```

cifratura simmetrica dei file iso e zip

```
5: file_stream_iso_encrypt = Encrypt( K_pub, file_stream_iso)
6: file_stream_zip_encrypt = Encrypt( K_pub, file_stream_zip)
```

calcola dell'hash sui file iso e zip cifrati

```
7: string_hash_iso_enc = hash_256(file_stream_iso_encrypt)
8: string_hash_zip_enc = hash_256(file_stream_zip_encrypt)
```

[5] eXtensible Markup Language è un metalinguaggio che consente di definire e controllare ciò che viene scritto fra due tag.

[6] La differenza tra passphrase e password risiede nel numero di caratteri che vengono utilizzati. Nella prima i caratteri dovrebbero essere non meno di 20/30, nella seconda i caratteri dovrebbero essere non meno di 6/8. Le specifiche ADI indicano che il numero di caratteri debba essere maggiore uguale di 40.

III. Creazione del file info.json[7] che conterrà i vari *digest* calcolati precedentemente. Di seguito abbiamo un esempio del file info.json con la sua tipica indentazione

```
{

    "conferimento": "identificativo-01",
    "subiso": "true",
    "digests": [
        {
            "object": "pwd",
            "path": "passphrase",
                                                                    "hash":
"e3e2a8ce620b959215fdd77227ca1eac352a84f5ef3d55f727db5db32b9120e0"
        },
        {
            "object": "iso",
            "path": "audio.iso",
                                                                    "hash":
"2b6249bbb9952bfe12b791c87423748cec08dcfb05bd103da34deefb8bdd29e5"
        },
        {
            "object": "iso.enc",
            "path": "audio.iso.enc",
                                                                    "hash":
"048f22a713019dd0ee89954440481f3cf9d41b192fd813e23304cd310e40ed77"
        },
        {
            "object": "zip",
            "path": "audio.zip",
                                                                    "hash":
"e47d7d53c4e858cea0419a65ad5164855cd24a128a4d0f558131b62b9462d9e9"
        },
        {
            "object": "zip.enc",
            "path": "audio.zip.enc",
                                                                    "hash":
"e5e620987a3703f48dc60ce48b99a22460719739ccb5af1ab7f6ef5bb7a51c4c"
        }
    ]
}
```

IV. Infine, si rinomina la cartella contenente tutti questi file con il *digest* che si ottiene dalla funzione di hash calcolata sul file info.json

```
1: string_name = hash_256(info.json)
```

[7] JavaScript Object Notation è un formato standard per lo scambio di dati, è leggero e human readable.

```
2: new_folder_name = new Folder(string_name)
```

Il risultato finale con i file cifrati è possibile visionarlo in Figura 2, dove è rappresentata la cartella e i file al suo interno.

```
141bbdaf88c1670f5746393014a387e951a992c38ee4c346e208fd7560f6e532
  |
  +-- passphrase
  |
  +-- audio.iso.enc
  |
  +-- audio.zip.enc
  |
  +-- info.json
```

Figura 2: Risultato della fase cifratura [8]

1.2.3.2 Fase di validazione

La fase di validazione ha come scopo quello di controllare che i file cifrati siano stati cifrati in modo corretto prima che il processo vada a scrivere il conferimento sul supporto di memorizzazione. Inoltre, viene fatto anche un controllo dei file contenenti i metadati tramite l'utilizzo dei file XSD. Il codice, che andrà a verificare la validità di ciò che è appena stato cifrato, agirà in modo inverso rispetto al precedente. Prima andrà a controllare che gli hash dei file cifrati corrispondano a quelli presenti nel file info.json (per quanto riguarda i file .iso e .zip) e successivamente, decifrata la chiave pubblica attraverso la chiave privata della procura, si decifreranno i file .iso e .zip. Su questi verrà ricalcolato il *digest* per verificarne l'uguaglianza con quello presente nel file info.json.

E' importante menzionare la procedura di controllo dei file xml, tramite i file xsd. Questi file, il cui nome è l'acronimo di Xml Schema Definition [4], definiscono i vincoli che un certo file xml deve avere. In altre parole, possiamo noi definire dei vincoli e poi controllare se i file xml da noi utilizzati rispettino o no questi stessi vincoli.

[8] Fonte: Archivio_Digitale_Multimediale-Specifiche_Tecniche_v2.1.pdf

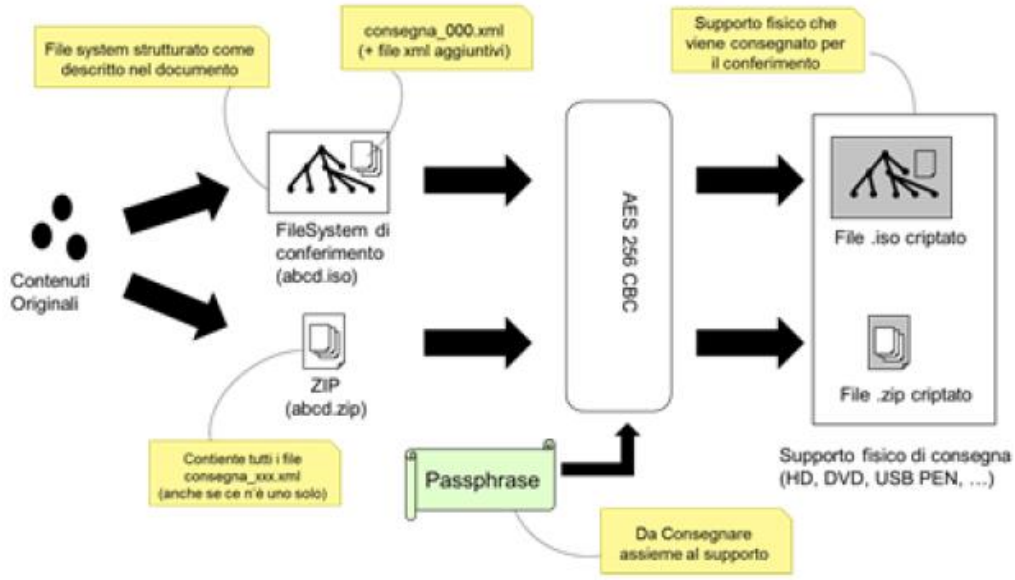


Figura 3: Schema del processo di cifratura[9]

1.3 Architettura a microservizi su richiesta

Dopo aver spiegato le fasi in cui si articola il conferimento nei paragrafi precedenti, andremo a spiegare il perché sarebbe necessaria una nuova architettura, sottolineando i problemi riscontrati dall'aumento delle dimensioni dei conferimenti. Si terminerà il paragrafo con la spiegazione dei vantaggi portati da questa nuova architettura.

I problemi legati alla lunghezza dell'elaborazione del conferimento sono dovuti in primis alle dimensioni dei conferimenti sempre più ingenti e, in secondo luogo, all'impossibilità di parallelizzare i *task* di lavoro a causa di un software i cui servizi sono molto accoppiati l'uno all'altro.

L'idea alla base della nuova architettura è quella di utilizzare un orchestratore di container per gestire la creazione e l'amministrazione di microservizi. Questo pattern si discosta leggermente dal normale utilizzo che viene fatto dell'orchestratore. Infatti, solitamente lo si utilizza per avere il *load balancing* delle richieste sulle diverse repliche della stessa applicazione, permettendo di avere uno scalamento orizzontale a seconda della quantità delle richieste.

In questo caso, invece, l'orchestratore è utilizzato non solo per la creazione dei container a scopo di dare resilienza, ma più come un servizio che al suo interno abbia della logica

[9] Fonte: Archivio_Digitale_Multimediale-Specifiche_Tecniche_v2.1.pdf

applicativa e che quindi crei dei container con cui poi si aspetta d'interagire. Rispetto all'interazione standard che si basa semplicemente (nella maggior parte dei casi) sulla conoscenza dello stato di salute degli stessi, qua l'interazione viene utilizzata per capire a che punto sia la procedura che deve svolgere tale contenitore, se è stata terminata e in che modo è stata terminata.

Si è pensato di creare dei microservizi per il fatto che la fase di cifratura e quella di validazione sono due flussi completamente indipendenti, o meglio, necessitano di essere eseguiti in sequenza, ma non hanno dei punti in comune che ci obblighino a svolgerli in maniera accoppiata.

Avremo quindi un'entità (colei che gestisce il flusso di elaborazione) che chiameremo *supervisor*. A fronte di una richiesta di elaborazione pervenuta creerà una prima istanza del microservizio che si occupa di cifrare tramite le client API[10] messe a disposizione dall'orchestratore e, quando questo avrà terminato, creeremo l'istanza dell'altro microservizio: il validatore, che lavorerà sui dati ricevuti dal precedente microservizio.

Nei paragrafi che seguono si accennerà ad alcune caratteristiche dei pattern utilizzati e i benefici che si potrebbero avere, andando a fare un eventuale *deploy* dell'architettura su di un'infrastruttura cloud esterna in termini di costi monetari.

1.4 Pattern a Microservizi

All'interno del paragrafo verrà spiegato il passaggio che ha permesso di spostarsi da una programmazione monolitica a una basata su microservizi. Il passaggio per arrivare a questa architettura non è stato un passaggio unico; infatti, si è passato attraverso delle fasi intermedie [5].

Come si evince dalla Figura 4, all'inizio vi era questo software monolitico, tipicamente un singolo progetto che veniva distribuito tramite un unico pacchetto. Gli inconvenienti si registravano con l'evoluzione del progetto che, diventando troppo grosso, portava a problemi in fase di sviluppo, test e implementazione. Dal monolite si è passati poi a una scomposizione a strati dell'applicativo. Gli strati vanno a identificare i *tier* dell'applicazione, uno strato di dati , uno dedicato alla logica applicativa e infine un livello di presentazione.

[10] Application Programming Interface. Sono interfacce che vengono messe a disposizione per interagire con l'applicazione/sistema e le sue funzionalità.

Tuttavia, anche con questa nuova divisione si formava una sorta di collo di bottiglia nel layer della logica applicativa nel momento in cui venivano implementate nuove caratteristiche dell'applicazione.

Infine, si è cercato di avere una scomposizione più legata alle funzionalità di business, in altre parole ai servizi che vengono offerti dall'applicazione: il servizio che riguarda la gestione dell'autenticazione, il servizio che riguarda la gestione delle notifiche ecc. Si è cercato di fare lo stesso discorso all'interno del lavoro di tesi, separando il servizio dedicato alla gestione della cifratura da quello per la gestione della validazione.

Sempre rispetto al pattern in questione, è da sottolineare la necessità da parte di questi servizi di comunicare tra di loro. Nelle applicazioni a servizio singolo, gli eventuali *processi*[11] che debbono comunicare possono farlo tramite l'utilizzo di *lock* da acquisire per poter scrivere messaggi su file condivisi. Nel caso dei microservizi, i processi che operano possono trovarsi a essere elaborati da processori diversi, su macchine distinte. E' quindi necessario l'utilizzo di elementi esterni che abbiano l'onere di permettere la comunicazione.

Per tale motivo le architetture a microservizi fanno proprio uso dei *message broker*, che sono in grado di garantire affidabilità nella ricezione e nell'invio dei messaggi, permettendo inoltre l'asincronicità dei servizi.

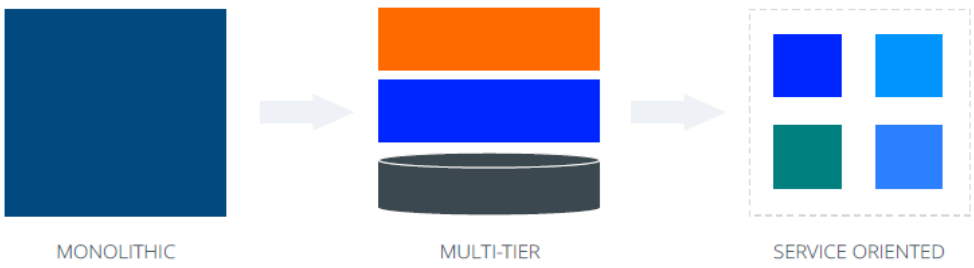


Figura 4: Evoluzione architetturale[12]

[11] I thread a meno che non siano di processi diversi (e quindi comunicherebbero come se fossero due processi diversi) possono appartenere allo stesso processo, e quindi hanno lo spazio di indirizzamento condiviso per cui non necessitano di mezzi per la comunicazione esterna come accade per i processi, possono accedere alle stesse variabili.

[12] Fonte: http://losviluppatore.it/wp-content/uploads/2015/09/arch_evol.png

1.5 Cloud

Lo sviluppo di hardware sempre più performanti e i passi avanti nel campo della virtualizzazione hanno portato a un cambiamento radicale: il ribaltamento della regola "*one application per server*". Precedentemente si era costretti a ospitare una singola applicazione per server mentre ora, tramite la virtualizzazione, si riesce a far coesistere più applicativi sullo stesso server. Tutto ciò ha portato anche a un abbattimento dei costi e dell'hardware da reperire. Il cloud permette quindi l'utilizzo di server remoti sfruttando la rete internet.
I servizi che può offrire il cloud sono molteplici e possiamo distinguerli in quattro diverse categorie, a seconda del tipo di servizio offerto. La Figura 5 mostra un breve sommario delle tipologie.

- HaaS (On-Premises), Hardware As A Service. Si ha la possibilità di affittare l'hardware, lo storage, "è un po' come avere un server a casa". Si ha la massima flessibilità possibile, ma è necessario gestirsi tutta la parte della virtualizzazione.

- IaaS, Infrastructure As A Service. Si affittano le machine virtuali, si possono attaccare volumi per lo storage e infine creare delle reti virtuali. Con questa soluzione si ha ancora molta flessibilità senza doversi preoccupare di gestire la virtualizzazione.

- PaaS, Platform As A Service. Vengono messi a disposizione software quali database, programmi di messaggistica, servizi di *map-reduce*, che per l'appunto sono *hostati* all'interno del cloud. Non bisogna più preoccuparsi della gestione del software (installazione, aggiornamenti ecc.), ma ci si limita al suo utilizzo.

- SaaS, Software As A Service. In questo caso l'utente può fruire dell'applicazione già pronta, senza dover scrivere alcun programma o doversi gestire alcuna configurazione.

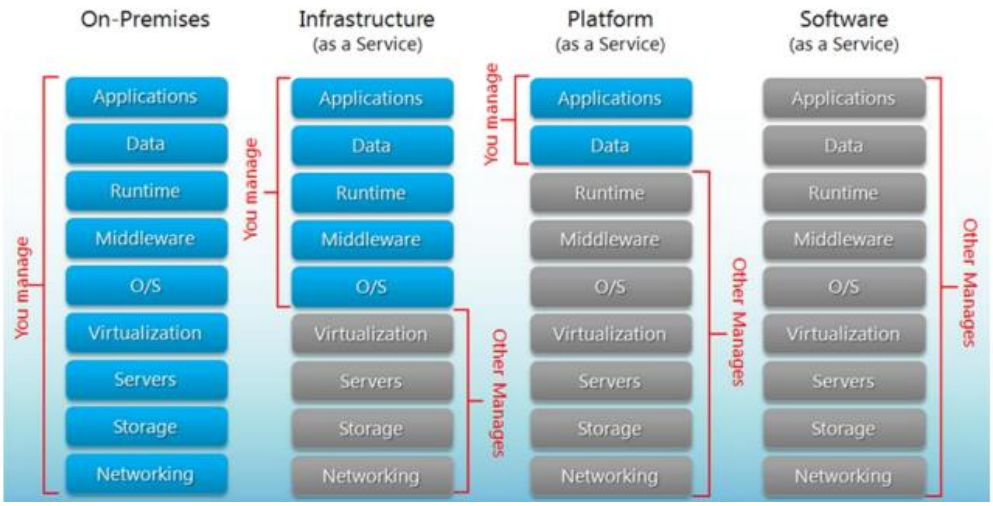


Figura 5: Modelli di servizio[13]

La possibilità del deploy di questa architettura su di un cluster di terze parti è il motivo per cui si è scelto di accennare al cloud. Nonostante la consapevolezza dei vincoli stringenti in termini di privacy relativi alla tematica trattata, non posso comunque esimermi dall'affermare che la flessibilità del cloud permetterebbe di poter "comprare" solo quello che effettivamente ci serve e pagarlo per il tempo che è necessario. In questa maniera le singole procure potrebbero evitare di sobbarcarsi la spesa onerosa dell'hardware e poter aumentare / diminuire le risorse sul cloud in base al carico di lavoro

[13] Fonte: https://drive.google.com/file/d/17rMXX1OBfPR_vzhroqnnbMbhzoQ8Z7gW/view

Capitolo Secondo

Capitolo 2

2 Tecnologie

All'interno del seguente capitolo verrà fatta una panoramica generale delle tecnologie utilizzate durante la progettazione e lo sviluppo della soluzione architetturale. Per la creazione dei microservizi è stato utilizzato principalmente il linguaggio Java, a eccezione delle parti legate al *deployment*[14] dei componenti Kubernetes, per cui è stato utilizzato il linguaggio Yaml e delle parti legate alla costruzione dei *container*[15] per le quali sono stati utilizzati linguaggi di *scripting*[16].
Sono stati impiegati alcuni *framework,* tra cui Spring e Logback, mentre dall'ambiente *devops*[17] è stato fatto largo uso di applicativi quali Docker, Kubernetes e NATs. Infine, per la gestione del progetto e delle librerie che lo compongono, è stato utilizzato Maven, mentre per la scrittura del codice l'*ide*[18] della Jetbrains IntelliJ IDEA.

2.1 Framework

Nell'ambito dello sviluppo del software, per *framework* si intende un'architettura logica che serve a supportare il design e la creazione di progetti.

2.1.1 Spring

Spring è un *framework* gratuito per lo sviluppo delle applicazioni basate su Java [6]. Fornisce diverse librerie a seconda del contesto in cui viene utilizzato: dalle applicazioni web al cloud, fino all'automatizzazione di *task*. Nel caso della programmazione a oggetti fornisce supporto nella connessione degli oggetti tra di loro, permettendo di passare da una programmazione procedurale a un tipo dichiarativo tramite l'utilizzo di apposite annotazioni.

Figura 6: Logo Spring framework

[14] Indica l'azione di schieramento delle risorse all'interno del cluster.
[15] Contenitore che ospita il ciclo di vita di un'applicazione.
[16] Sono linguaggi interpretati a differenza di linguaggi come il C che è compilato.
[17] Nell'ambiente devops ci si occupa dei processi e delle metodologie che riguardano il ciclo di vita di un software, dalla messa in campo al mantenimento fino agli update.
[18] Integrated Development Environment. E' l'ambiente di sviluppo del codice.

Vi sono poi alcuni concetti su cui si appoggia Spring che è necessario menzionare:

- *Inversion of Control (IOC)*. È un paradigma che inverte il flusso di controllo della programmazione tradizionale. Lo sviluppatore prima si occupava della creazione, inizializzazione e chiamata dei metodi dei vari oggetti mentre con questo nuovo paradigma è il *framework* che, sotto opportuni stimoli, se ne occupa. Dalla Figura 7 possiamo comprendere che nell'approccio tradizionale, sarebbe stato necessario istanziare l'oggetto *engine* all'interno della classe *vehicle*. Grazie all'Ioc basta inserire il nome dell'oggetto con l'annotazione *@Autowired* e Spring stesso si prenderà l'onere di andare a cercare un *bean* a cui collegare l'oggetto.
- *Dependency Injection (DI)*. Questo nuovo approccio, al contrario di ciò che avviene tradizionalmente dove è il punto di ingresso del programma a dover creare tutte le dipendenze, permette di non badare più all'inizializzazione delle dipendenze di oggetti che si andranno ad utilizzare. Quindi possiamo dire che la *dependency injection* è il processo mediante il quale spring va a cercare i *bean* necessari per il funzionamento di un particolare *bean* e li inserisce come dipendenza.
- *Aspect Oriented Programming (AOP)*. È un paradigma che permette di definire dei comportamenti trasversali a tutte le applicazioni in modo da non dover ripetere la logica applicativa. Spring fornisce diversi *advice* (così sono chiamati gli *aspect*) tra cui *Before*, *After*, *After-Running,* ecc.

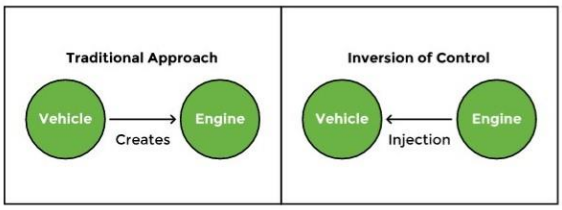


Figura 7: Approccio tradizionale vs IOC

2.1.2 Logback

Logback, *framework* che aiuta nella creazione e gestione del log[19], è un software *open source*[20] creato dalla compagnia QOS.ch [7]. L'architettura si compone di tre classi principali: il *logger*, l'*appender* e la classe *layout*.

[19] Il log fornisce la "storia" dell'applicazione / sistema, ovvero ciò che è avvenuto al suo interno.
[20] Software libero, non soggetto a copyright.

Il *logger* è il contesto per la creazione del log, gli *appender* si occupano di collegare e aggiungere i nuovi log ai file di log eventualmente già presenti, infine il *layout* permette di customizzare e formattare l'*output* come si vuole.

2.2 Container

I *container* nascono come alternativa alla *full virtualization*, che viene fornita dalla *Virtual Machine* (VM) tramite l'*hypervisor*. Osservando la Figura 8 possiamo capire come la virtualizzazione dell'*hardware* e quella del *Guest OS*[21] possano abbassare le performance andando a introdurre un *overhead* dovuto alla virtualizzazione del numero chiamate di sistema. Al contrario, l'uso dei *container*, e in particolare l'uso di Docker, garantisce delle buone performance mantenendo un buon isolamento tra le varie applicazioni grazie alla possibilità di poter condividere il sistema operativo e non doverne avere un altro da virtualizzare. E' da sottolineare l'introduzione del concetto di *namespace* che ha permesso questo salto tecnologico.

Il *namespace* è una caratteristica del *kernel*[22] Linux che partiziona le risorse del kernel in modo tale che un insieme di processi veda un certo numero di risorse mentre un altro insieme di processi veda un set di risorse differenti. Quindi possiamo dire che i vari container sono isolati tra di loro e isolati dal sistema operativo host grazie ai *namespace*. Le recenti versioni del *kernel* di Linux forniscono ben sette differenti tipi di *namespace*, tra questi i più importanti sono *process namespace*, *network namespace* e *mount namespace*.

All'interno del sistema operativo tutti i processi sono figli del processo *init*, ovvero quello con *pid*[23] uguale ad uno. Tramite il *namespace* dei processi si riesce ad avere un nuovo processo che diventa radice di tutti i processi al di sotto di lui e che non conosce il processo che lo ha originato.

Il *network namespace* permette la creazione di reti differenti nonostante la scheda di rete rimanga singola. I *namespace* creati hanno associati uno stack di rete, un ip e un interfaccia diversi.

[21] Sistema operativo ospitato.
[22] Il kernel è il nucleo del sistema operativo, e si occupa della gestione dei processi per l'esecuzione dei programmi, della gestione della memoria e della gestione delle comunicazioni di input e output.
[23] Process Identifier, è un numero che va ad identificare in modo univoco un processo.

Infine, il *mount namespace* permette la creazione di un nuovo filesystem completamente staccato da quello dell'host. E' sulla base dei vari *namespace* che nasce la *light virtualization* tramite i container.

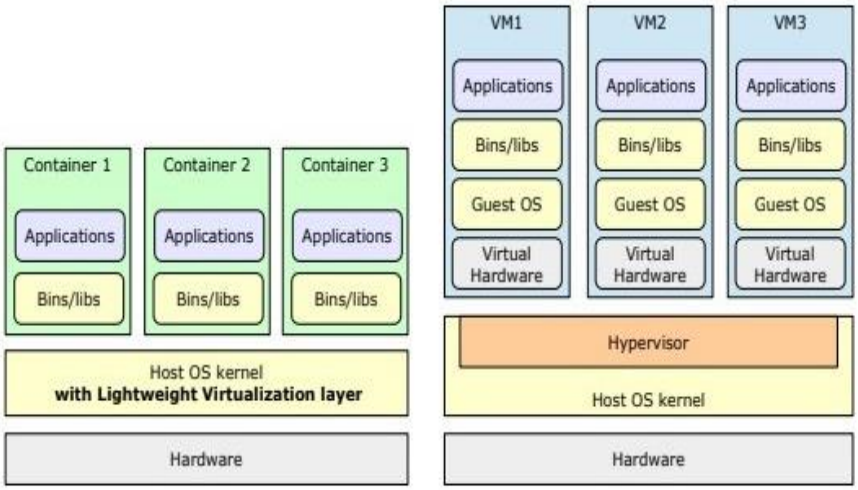


Figura 8: Lightweight Virtualization vs Full Virtualization

2.2.1 Docker

Docker è un progetto *open source*, multipiattaforma ed è composto da diversi elementi tra cui [8]:

Figura 9: Logo Docker

- *Docker Engine*. È il processo in grado di creare i *container*, fermarli e rilanciarli. Gira all'interno di un processo in *back-ground* comunicando con i servizi tramite una serie di API.
- *Docker Image.* È un *template read-only* che contiene un set di istruzioni per creare il *container*. L'immagine può essere generata a partire dal *Dockerfile* oppure scaricata dal *DockerHub*: una repository remota su cui si possono trovare i *layer* che faranno da base per i nostri *container* (es. Linux Alpine, OpenJDK).
- *Docker Container.* È l'istanza di un'immagine. Può essere fatto partire, fermare e ripartire mantenendo i cambiamenti all'interno del *filesystem.*
- *Docker Client.* Permette di accedere alle funzionalità offerte dal processo *docker-engine* attraverso le API. Questo componente è fruibile sia da linea di comando che da diverse GUI[24], tra cui *Docker Desktop* oppure *Portainer*.

[24] Graphic User Interface. E' la parte grafica dell'applicazione, quella con che permette l'interazione all'utente.

2.3 Orchestratore

L'utilizzo sempre più intensivo dei *container* ha portato alla necessità di studiare un'entità terza che potesse permettere di "amministrare" nel modo più automatico possibile il ciclo di vita dei contenitori. E' così che nasce il termine "orchestratore" e tra i più conosciuti abbiamo Kubernetes, Mesos, Swarm e OpenStack.

Tra le caratteristiche che questi orchestratori devono avere, c'è la possibilità di schedulare *container* seguendo metriche di diverso tipo, essere in grado di avviare più repliche in parallelo, gestire i fallimenti e i *crash* delle macchine, poter scoprire i servizi forniti da altri *container* sulla stessa rete.

2.3.1 Kubernetes

Kubernetes probabilmente è uno, se non il più utilizzato tra gli orchestratori di *container*. Nasce dall'esperienza di Google e attualmente è mantenuto dalla Cloud Native Computing Foundation. Kubernetes, conosciuto anche come K8s, è un progetto *open source* e questo ne ha permesso la sua rapida diffusione e utilizzo all'interno del mondo degli sviluppatori.

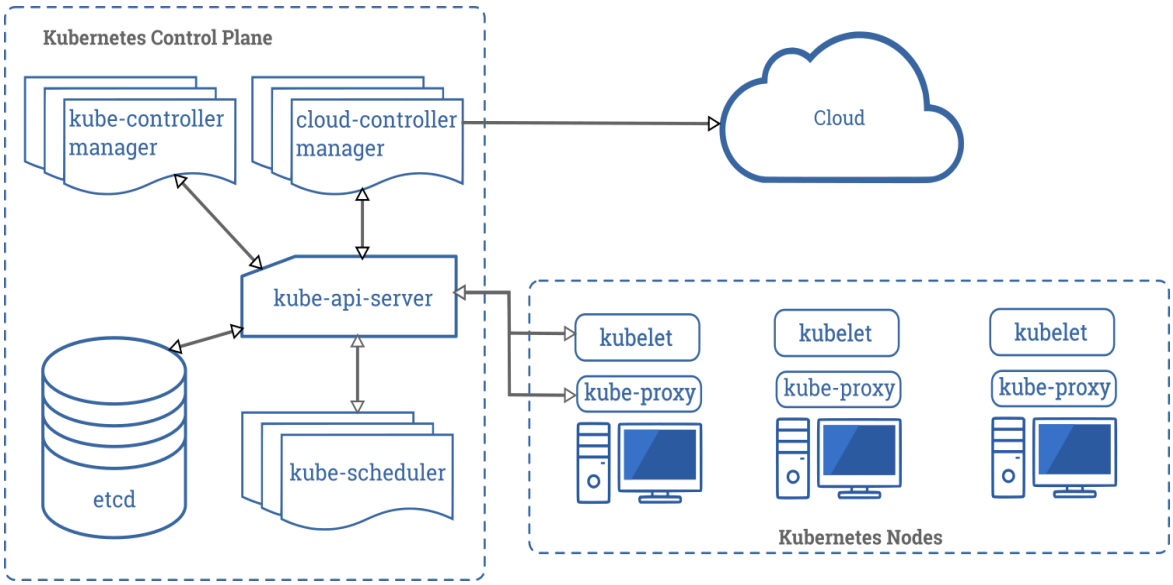


Figura 10: Architettura Kubernetes[25]

[25] Fonte: https://kubernetes.io/blog/2019/04/17/the-future-of-cloud-providers-in-kubernetes/

2.3.1.1 Concetti principali

Tra i concetti alla base di K8s [9] vi è quello di *Cluster*, un insieme di nodi che viene coordinato al fine di poter garantire una elaborazione distribuita.
I *Nodi* sono le "macchine" a cui viene demandato il carico di lavoro, possono essere fisiche o virtuali a seconda che girino nativamente sul server oppure siano macchine virtuali all'interno di un *datacenter*[26]. Esistono due tipi di nodi: i *Master Node*, che generalmente controllano il cluster e solitamente vi sono più copie in modo da garantire ridondanza e i *Worker Node*, che invece sono quelli atti all'elaborazione.

2.3.1.2 Componenti control plane

Dallo schema in Figura 10 si possono notare i componenti più importanti che formano il *Control Plane* di Kubernetes:

- *Kube-apiserver*. Sono le API che vengono esposte da Kubernetes e che possono essere interrogate da un *http-client* qualsiasi come ad esempio Curl.
- *Etcd.* È un database di tipo chiave-valore in cu vengono salvate le informazioni del *cluster.*
- *Kube-scheduler*. Decide a quale nodo attribuire il carico di lavoro basandosi su diversi fattori per fare la scelta migliore. I vincoli dell'hardware e i requisiti in termini di risorse sono alcuni dei fattori che vengono presi in considerazione.
- *Kube-controller-manager*. Serve per monitorare il *cluster* e il suo stato tramite le API k8s. Infatti, controllando ciclicamente i diversi processi, riesce ad apportare le modifiche che vengono richieste.
- *Kube-controller-cloud*. È presente solamente nel caso in cui si stia lavorando attraverso un *cloud provider* e permette di eseguire controlli specifici all'interno dell'infrastruttura di quest'ultimo.

2.3.1.3 Componenti worker nodes

All'interno di ogni *worker node* troviamo:

- *Kubelet*. È adibito a monitorare lo stato dei pod e dello stesso nodo, montare i volumi condivisi e controllare lo stato di salute dei singoli *pod*.

[26] Luoghi in cui si concentrano numerosi server al fine di erogare servizi.

- *Kube-proxy.* È un proxy che viene eseguito all'interno di ogni nodo e si occupa della gestione dei *Service*.
- *Container-runtime*. Serve a eseguire i *container* e, considerato il carattere *open source* di k8s, supporta diversi *container runtime* tra cui Docker, containerd, rktlet

2.3.1.4 Risorse k8s. Deployment, Pod, Service

Le risorse sono gli oggetti logici che k8s può creare. Hanno tutti la stessa struttura che deriva dal *template* utilizzato per la loro creazione, che prende il nome di Manifest.
All'interno è specificato il tipo di risorsa che si andrà a creare, se un Pod piuttosto che un Deployment tramite il campo *kind*. Vengono poi definiti i dettagli della risorsa all'interno dei campi *label* e *annotation* e, infine, vengono fornite nozioni più specifiche riguardo al *container* che ospiterà il carico di lavoro.
Kubernetes mette a disposizione diversi tipi di risorse, ma solo alcune di queste sono state prese in considerazione durante lo svolgimento della tesi. In particolare, è stato utilizzato l'oggetto di tipo *pod* e quello di tipo *deployment*.
I *pod* sono l'unità base dell'esecuzione delle applicazioni Kubernetes, come paragone potremmo dire che i pod stanno a k8s come i processi stanno al sistema operativo. Un *pod* a sua volta è costituito da uno o più *containe*r al suo interno, dove effettivamente prende posto l'applicativo.
I *deployment* sono simili ai *pod*, ma danno inoltre la possibilità di gestire il numero delle repliche. Questo tipo di oggetto è stato utilizzato per quanto concerne il *message broker*[27], fornendo resilienza a quest'ultimo.

I *service,* infine, sono l'ultima risorsa utilizzata all'interno del progetto di tesi. Sono molto importanti perché permettono di esporre i vari *pod / deployment* con diverse configurazioni di rete. Nonostante si sia fatto uso di un singolo tipo di configurazione (il nodePort), si ritiene opportuno menzionarli tutti e tre:

- *ClusterIP*, dove la porta del *container* che viene esposta è solamente accessibile dall'interno del *cluster*.

[27] Server che permettono la comunicazione tra applicativi diversi, alle volte situati anche in zone fisiche differenti.

- *NodePort*, che permette al *container* di essere contattato dall'esterno. L'indirizzo che ne deriva avrà come ip l'ip del nodo su cui gira il *container* mentre come porta quella esposta all'interno del Manifest.
- *LoadBalancer*, che permette di raggiungere il servizio dall'esterno come nel caso del NodePort, ma fornisce anche il servizio di bilanciamento.

Un altro aspetto assai importante che riguarda il *service* è la maniera in cui gli oggetti comunicano tra di loro all'interno del *cluster*. A causa della loro natura effimera, quando i *pod* si rompono non vengono aggiustati, ma vengono distrutti e creati dei nuovi.
Considerato quanto appena detto, è molto probabile che il nuovo *pod* che verrebbe tirato su non avrebbe lo stesso indirizzo ip avuto in precedenza, ma probabilmente uno nuovo. Tutto ciò non comporta problemi di connessione poiché i *pod* possono anche comunicare tra di loro grazie a delle etichette, rendendo non più necessaria una configurazione statica dell'indirizzo ip.

2.3.2 Client Kubernetes API

Come già accennato nei paragrafi precedenti, Kubernetes mette a disposizioni delle API per poter comunicare. Infatti, è lo stesso controller che le utilizza per schedulare i servizi. Anche nel progetto di tesi è stato necessario utilizzare queste API all'interno del codice dei servizi creati.

Siccome è necessario dover chiamarle all'interno del codice dei vari microservizi, si è cercata una libreria client che implementasse queste chiamate tramite delle funzioni. La libreria client che è stata trovata, però, non è supportata da Kubernetes stesso ma è un progetto open source che prende il nome di *fabric8* ed è supportato da RedHat.

2.3.3 Minikube

Per il lavoro di tesi è stata utilizzata una versione di Kubernetes che prende il nome di Minikube, è una versione leggera e che può girare in locale anche su macchine non così recenti e performanti. L'utilizzo di questa versione permette di utilizzare le caratteristiche principali di Kubernetes senza perder troppo tempo con varie configurazioni.

Minikube può lavorare come se fosse una *virtual machine*, un *container* oppure direttamente sull'hardware (*bare metal*) [10]. Quando viene lanciato si possono decidere i driver con cui avviarlo, se quelli di Docker oppure di VirtualBox (servizio di *full virtualization* messo a disposizione dalla compagnia Oracle attraverso una licenza gratuita). Per quanto riguarda la

tesi, dopo un iniziale sviluppo con Docker Desktop, si è passati all'utilizzo della VM con i driver di Virtual Box. Questo perché con la soluzione iniziale si avevano problemi a far funzionare alcuni servizi di Kubernetes. Il cluster che Minikube ci mette a disposizione è composto da un singolo nodo.

2.4 Crittografia

La crittografia occupa una parte importante all'interno del progetto, considerando i dati sensibili che sono toccati. Per tale ragione, si ritiene necessaria una breve introduzione delle due tecniche di cifratura utilizzate, lasciando poi a successivi capitoli la spiegazione del modo in cui vengono usate.

A identificare una *suite* di crittografia vi sono quattro parametri: azione (cifratura/decifratura), algoritmo, numero di bit della chiave digitale[28] e, infine, la modalità. Di seguito declineremo tali caratteristiche su due principali categorie di cifratura: quella a chiave simmetrica e quella a chiave asimmetrica.

2.4.1 Crittografia a chiave simmetrica

La cifratura simmetrica utilizza una singola chiave che viene condivisa tra mittente e destinatario. Sia la cifratura che la decifratura avvengono con la medesima chiave. Ci sono principalmente due modalità: *stream* e *a blocchi (CBC, chyper block chain* ed *ECB electronic chyper block)*. Gli algoritmi che seguono fanno uso di uno dei metodi citati precedentemente: DES, RC e AES. Infine, la lunghezza della chiave gioca un ruolo importante nel determinare la maggior o minor sicurezza di una certa suite crittografica.

Essendo la chiave simmetrica una modalità che richiede un'elaborazione non eccessiva, si tende a prediligerla nel momento in cui si ha una grossa mole di dati da cifrare.

2.4.2 Crittografia a chiave asimmetrica

La cifratura a chiave asimmetrica permette di avere una maggior sicurezza, ma allo stesso tempo apporta un *overhead* computazionale. In questo caso vi sono due chiavi: una pubblica e una privata. Per cifrare e decifrare non si può utilizzare la stessa chiave, come accade invece per quella simmetrica. L'utilizzo della stessa per decifrare (o cifrare a seconda di quale era stata l'azione iniziale) porterebbe ad avere un' ulteriore cifratura. Tra gli algoritmi

[28] File contenente una sequenza di caratteri. Per comodità il nome verrà abbreviato con chiave.

utilizzati tutt'ora e anche nel contesto della tesi abbiamo RSA, che si basa sulle proprietà dei numeri primi e DSA, che si basa sul problema dell'algoritmo discreto.

La cifratura asimmetrica può essere sfruttata in diversi contesti, tra cui la firma di un documento (firma digitale). Firmando un documento con la chiave privata diamo la sicurezza matematica che chiunque riceva il documento e voglia essere sicuro dell'autenticità (ovvero che non sia stato modificato) possa farlo utilizzando la chiave pubblica che lo stesso firmatario espone per decifrare. La lunghezza dei bit delle chiavi arriva fino 4072 bit, garantendo una ragionevole sicurezza.

Questo tipo di cifratura offre anche la riservatezza senza dover scambiare, in maniera telematica (non sicura) o fisica, la chiave precedentemente come invece è necessario nel caso di cifratura simmetrica. Andando a cifrare un contenuto con la chiave pubblica di chi si vuole contattare saremo certi che solamente il possessore della chiave privata riuscirà a decifrare il messaggio.

2.4.3 Key Derivation Function

Questo tipo di algoritmo viene utilizzato per la creazione di chiavi simmetriche a partire da una *passphrase (o password)*, un *salt* e un certo numero di iterazioni. Il *salt*, che in italiano sarebbe sale, è una sequenza di caratteri che viene aggiunta a una password prima di subire il calcolo dell'*hash*[29]. Quasi sempre questa tecnica viene utilizzata all'interno delle basi dati per salvare gli *hash* delle password degli utenti e non le password in chiaro, in modo tale da rendere difficile un attacco di tipo dizionario[30], nel caso in cui venisse rubato il database con le password.

2.4.4 Calcolo del digest

Il calcolo del *digest* nasce dal fatto che tutto ciò che è cifrato magari non può essere decifrato e letto da terzi, ma può essere modificato in modo imprevedibile. Per mettersi al riparo da questo problema, soprattutto nel caso delle comunicazioni, si è adottata la strategia del *digest*.

[29] L'hash è una funzione non invertibile che mappa una stringa a lunghezza variabile, su di un'altra stringa che ha lunghezza fissa.
[30] L'attacco di tipo dizionario mira a scoprire una password cercando all'interno di un file (chiamato dizionario) che contiene un gran numero di parole e password note (es: 12345, abcdef ecc.). La presenza dell'hash costringe a creare dei dizionari che contengano password *hashate* con cui poi fare il confronto. Questo porta ad un aumento dei tempi per l'attaccante.

Questa procedura porta a calcolare una sorta di riassunto del messaggio che si vuole proteggere e, nel caso in cui si voglia controllare che il file originario non sia stato manomesso, basterà ricalcolare il digest e confrontarlo con quello originario. Nello specifico della tesi, i digest che vengono calcolati e che devono essere controllati sono quelli che si trovano all'interno del file info.json. In questo modo si ha la certezza che i file che vengono cifrati non possano essere manomessi.

Per completezza citiamo alcuni degli algoritmi più utilizzati: la famiglia MD (MD2, MD4, MD5) anche se oramai in disuso perché considerati poco sicuri e quelli della famiglia SHA (SHA256, SHA512,) molto più sicuri e utilizzati nel contesto della tesi.

2.5 Message Broker

Con lo sviluppo dei microservizi e quindi il disaccoppiamento di componenti che prima viaggiavano uniti, è stato necessario studiare dei nuovi metodi per permettere ai servizi di parlarsi tra di loro. Questa necessità ha portato alla nascita dei *message broker*. Il funzionamento si basa sulla registrazione, da parte di chi riceve, a un certo canale e, allo stesso modo, chi avrà intenzione di mandare un messaggio, farà la medesima registrazione al canale prima di inviare il messaggio. In questo modo anche altri servizi interessati allo stesso messaggio potranno iscriversi al medesimo canale e riceverlo.

La comunicazione tra microservizi può avvenire in due maniere: sincrona e asincrona. Nello specifico abbiamo :

- approccio sincrono dove il microservizio invia un messaggio e poi attende una risposta prima di andare a eseguire altre operazioni. In altre parole, questa attesa si esplicherà nell'interrogare la coda per farsi restituire un'eventuale risposta.
- approccio asincrono dove dopo l'invio del messaggio da parte di un microservizio, questo si disinteresserà della risposta per andare ad eseguire altre operazione. Sarà poi la coda a notificargli l'arrivo del nuovo messaggio sfruttando una *callback* [31]

[31] E' una funzione o comunque un blocco di codice che viene passato ad un'altra funzione come parametro. Un esempio è rappresentato dagli EventHandler (gestori di eventi) nel caso della programmazione web quando viene ad esempio cliccato un bottone.

2.5.1 NATs

Figura 11: Logo NATs

Sul mercato vi sono numerosi *message broker*. Tra questi possiamo citare alcuni dei più famosi come *Kafka*, *KubeMQ* e *RabbitMQ*. Tuttavia, per il progetto di tesi, si è preferito sperimentare un nuovo competitor: NATs. L'acronimo sta per "*Neural Autonomic Transport System*", dove Derek Collison, il creatore, ha voluto richiamare alla mente lo scambio di messaggi che vi è all'interno del sistema nervoso centrale.

Tra le caratteristiche che rendono interessante questo *message broker*, anche se non è così famoso, abbiamo: il supporto di ben 48 *client* (piattaforme e linguaggi), mantenuti in parte dalla *community*, tre tipi di garanzie nell'inoltro dei messaggi e il supporto allo standard del TLS[32].

2.5.1.1 *Pattern* supportati

NATs supporta pressoché tutti i *pattern* più conosciuti [11]. Tra questi troviamo il pattern *publish/subscribe*, dove il mittente e il destinatario si sintonizzano su di un canale. In questa situazione il *message broker* ha la responsabilità di mantenere il messaggio, utilizzando come supporto una coda FIFO dal quale verrà estratto il messaggio da parte del *subscriber.*

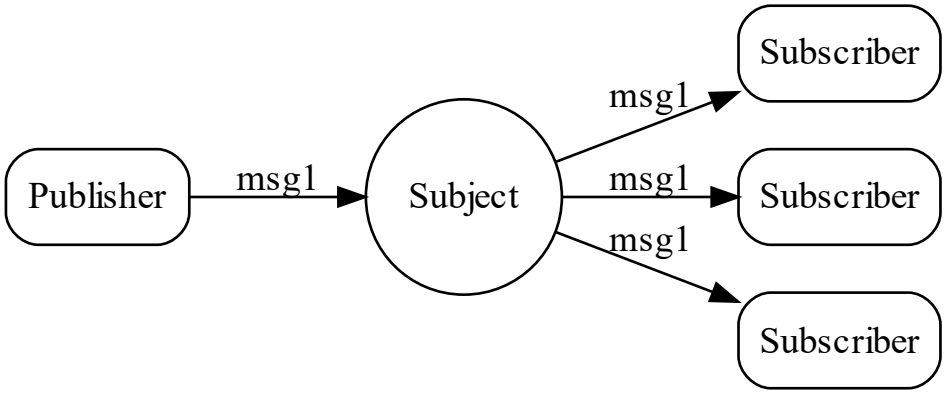


Figura 12: Schema del pattern publish-subscribe.[33]

Un altro *pattern* incluso è il *pattern request-reply*, nel quale il destinatario si iscrive al canale che vuole con una risposta già pronta. Nell'ambito di applicazioni che spesso interrogano sensori questo *pattern* può tornare utile.

[32] Transport Layer Security. Protocollo di sicurezza.
[33] Fonte: https://docs.nats.io/nats-concepts/core-nats/pubsub

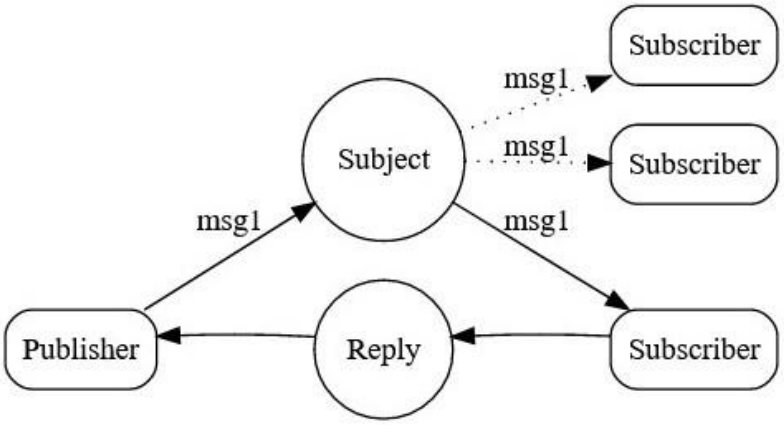
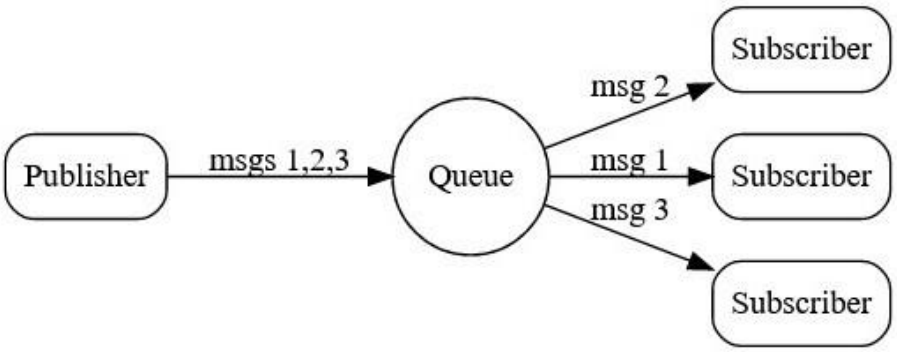


Figura 13: Schema del pattern request-reply[34]

Infine, vi è il *queue-group* che ci fornisce una *load-balancer* completamente gratuito. Con questo *pattern* abbiamo un rapporto 1:N tra *publisher* e *subscriber*, permettendo che il messaggio venga ricevuto da un solo *subscriber* alla volta. NATs fornisce questa opzione con una particolare iscrizione da parte del *subscriber*, senza dover far nulla su chi pubblica.



[35]

Figura 14: Schema del pattern queue-group

2.5.1.2 Garanzie di inoltro

Come detto inizialmente, NATs supporta tre tipi di garanzia di inoltro: *at most one*, *at least one* e, infine, *exactly one*. Nel primo caso potrà essere ricevuto al massimo un messaggio dal subscriber. Tuttavia, trattandosi di garanzia *at most one*, è contemplata anche la possibilità di non ricevere alcun messaggio nel caso in cui il ricevente non faccia per tempo l'azione di subscribe (ovvero mettersi in ascolto sul canale).

[34] Fonte: https://docs.nats.io/nats-concepts/core-nats/reqreply
[35] Fonte: https://docs.nats.io/nats-concepts/core-nats/queue

At least one assicura che almeno un messaggio arriverà a destinazione, mentre nella maggior parte dei casi la garanzia sarà *exactly one*. Infatti, tramite la possibilità di salvare il messaggio su file / memoria, il ricevente avrà la possibilità di ricevere il messaggio anche nel momento in cui non fosse stato pronto a leggerlo.

È necessario sottolineare il funzionamento della garanzia *exactly one*, infatti grazie a questa garanzia si riesce a evitare che vengano inviate più copie dello stesso messaggio. Alcune volte capita che dei malfunzionamenti, difficili da riprodurre, portino a far pensare a chi pubblica che il messaggio non sia stato ricevuto in modo corretto e quindi a rinviarlo. Tramite questa garanzia viene inserito un id all'interno dell'header (da parte di chi pubblica), questo fa si che il broker sappia quali siano i messaggi ricevuti e quelli che debbono essere "inoltrati". Quindi, mantenendo questi id all'interno di una tabella per un certo periodo, riesce a distinguere se un messaggio è stato già ricevuto oppure no e di conseguenza non va a svegliare i consumer per un messaggio già inviato in precedenza. Questo lasso di tempo in cui il broker può mantenere gli id all'interno della tabella è variabile e può essere deciso tramite la modifica del parametro *duplicate window*.

```
Information for Stream my_stream created 2021-10-12T08:42:10-07:00

Configuration:

             Subjects: foo
     Acknowledgements: true
            Retention: File - Limits
             Replicas: 1
       Discard Policy: Old
     Duplicate Window: 2m0s
     Maximum Messages: unlimited
        Maximum Bytes: unlimited
          Maximum Age: 0.00s
 Maximum Message Size: unlimited
    Maximum Consumers: unlimited


State:

             Messages: 0
                Bytes: 0 B
             FirstSeq: 0
              LastSeq: 0
     Active Consumers: 0
```

Figura 15: Parametri configurazione stream

2.5.1.3 JetStream

JetStream è il sistema di persistenza dei messaggi che mette a disposizione NATs [12], e che si aggiunge alle funzionalità base messe a disposizione da "Core NATs" che è la versione base del server. E' necessario abilitare JetStream all'avvio del server tramite l'opzione "-js", altrimenti questo avrà solamente le caratteristiche base.

Come possiamo comprendere dal paragrafo precedente per avere determinate garanzie è necessario dover eseguire NATs con l'ausilio di JetStream. Con tale funzionalità abbiamo anche due tipi di consumatori: *push based* e *pull based*. Sulla base di queste due tipologie veniamo ad avere anche strategie differenti di *acknowledge*.

L'*acknowledge*, molto spesso abbreviato con *ack,* è un messaggio che viene inviato per confermare l'avvenuto arrivo dei messaggi, in altre parole una garanzia per chi invia dei dati sul fatto che il destinatario li abbia ricevuti. Questa stessa tecnica viene utilizzata nel caso del protocollo TCP, dove si ha la garanzia della consegna dei segmenti tramite l'utilizzo dell'*ack*, ovvero all'interno di un a certa finestra di ricezione il mittente deve ricevere un ack per ogni segmento inviato. Nel caso di mancata ricezione di un singolo ack, quest'ultimo rinvierà i segmenti all'interno della finestra.

Nel caso del consumatore *pull based* il servizio va a richiedere il messaggio dalla coda e per questa tipologia è necessario rispondere con l'ack tutte le volte. Nel caso del *push based*, siccome è il message broker a fare il *pushing* del messaggio verso il servizio, si possono adottare due strategie di *acknowledge*: *AckAll* e *AckNone*. Il primo caso ci porterebbe a pensare di dover mandare l'ack per ogni messaggio ricevuto, mentre al contrario si può inviare l'ack anche solo dell'ultimo effettivamente ricevuto su di un gruppo di messaggi. In questo modo si riesce a risparmiare diversa banda e a incrementare le performance. Nel secondo caso non viene inviato alcun *ack*, quindi nessuna garanzia di arrivo effettivo.

2.5.1.4 Comparazione con altri message broker

All'interno del paragrafo si farà una breve comparazione con il *message broker* Kafka, che rimane probabilmente il più conosciuto. Il funzionamento è simile a quello di NATs, tranne per il modo in cui vengono salvati i messaggi. I *topic*, che equivarrebbero ai *subject* in NATs, vengono partizionati all'interno di più *broker*. Questo permette ai client di leggere e scrivere sullo stesso *topic* nel medesimo istante. I vari messaggi (nel mondo Kafka sono chiamati eventi) che hanno la stessa chiave vengono inseriti all'interno della medesima partizione. I consumatori che leggono i messaggi li leggeranno sempre nello stesso ordine in cui sono stati scritti.

Dopo questo preambolo sul funzionamento di Kafka, possiamo sottolineare le differenze che, in un ambiente di produzione, potrebbero far propendere per l'utilizzo di NATs. Kafka è scritto in java e il contenitore occupa più di 600 Mb, mentre NATs occupa all'incirca 15

Mb. Infine, NATs, essendo scritto in Golang, permette di avere delle performance migliori nelle gestione dei messaggi.

2.6 Build automation, Maven

I software di *build automation* sono strumenti per la gestione del codice all'interno dei progetti. Nello specifico, lo sviluppo del software è composto da numerose fasi come la compilazione del codice binario, il collegamento delle risorse, il packaging, l'esecuzione di test per garantirne la correttezza e la messa in campo. Tramite i software di *build automation* queste fasi vengono automatizzate, nel caso di Maven attraverso dei *plug-in*, andando a prevenire errori e facendo risparmiare tempo.

Per il progetto di tesi è stato utilizzato Maven, la cui caratteristica base è la presenza del file *pom.xml*. All'interno del *pom* vengono inserite le dipendenze, ovvero le librerie esterne che un software utilizza. Di seguito un esempio di dipendenza (libreria lombok).

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
    <version></version>
</dependency>
```

All'interno della dipendenza vengono specificati i nodi xml groupId e artifactId, che servono a individuare in maniera univoca una libreria, il cui codice verrà scaricato dalle repository in modo automatico.

2.6.1 Multi-Module Project

Maven fornisce supporto anche nel lavorare con i moduli[36]. Questa caratteristica, infatti, è stata sfruttata all'interno del progetto dove i vari servizi sono stati creati come moduli a se stanti.

[36] I moduli vengono supportati anche dalle versioni ≥ Java 9 che dallo stesso ambiente di sviluppo IntelliJ oltre che per l'appunto da Maven. Il supporto ai moduli da parte di queste tre entità è comunque diverso. I moduli di Maven permetto di controllare le versioni dei moduli e le dipendenze tra essi, infatti ogni modulo produce un artefatto. I moduli in Java sono semplicemente un modo di incapsulare le classi. Infine, con IntelliJ si ha la possibilità di far coesistere framework e tecnologie diverse all'interno dello stesso progetto tramite l'uso dei moduli (progetto React con un progetto Spring).

```
<modules>
    <module>common</module>
    <module>cifratore</module>
    <module>viewer</module>
    <module>validatore</module>
    <module>osservatore</module>
    <module>supervisor</module>
</modules>
```

Il progetto *multi-module* fa si di avere un file *pom* all'interno del progetto più esterno (che ingloba tutti i moduli) e poi altri file *pom* all'interno dei singoli moduli. I vantaggi che ci offre l'utilizzo di inglobare i sotto-moduli all'interno di un progetto *multi-module* è quello di poter fare la build dell'applicazione con un singolo comando. Infatti, come si può vedere dall'appendice 5.3, è possibile creare il file Jar andando a fare il packaging del progetto più esterno, quello dell'Archiviatore, senza dover andare a fare il packaging dei singoli sotto-moduli. All'interno del file *pom* di ogni sotto-modulo è necessario andare a specificare il *parent project* tramite il tag apposito.

Inoltre, è stata utilizzata un'altra caratteristica di Spring che prende il nome di *Spring Boot Starter Parent*. Questa permette una migliore gestione delle dipendenze all'interno di un progetto; infatti, dà la possibilità di non dover più inserire le versioni delle dipendenze che andiamo ad utilizzare. E' lo stesso Maven che andrà a scaricare le dipendenze sulla base della versione che viene specificata all'interno del tag presente nel file *pom* più esterno (il *parent* file) .

Tutti i sotto-moduli avranno il seguente gruppo di tag.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.5</version>
    <relativePath/>
</parent>
```

Capitolo terzo

3 Architettura e sviluppi

3.1 Architettura complessiva

Per un focus generale sull'architettura possiamo osservare la Figura 16, la quale mostra tutti i componenti. Si può vedere come l'intera struttura sia divisa in due parti: ciò che è all'interno del nodo (i componenti che si trovano entro e sulla riga tratteggiata) e tutto ciò che sta al di fuori del nodo. All'esterno del nodo è presente un singolo componente: il *supervisor*. Come già citato nel paragrafo che riguarda Minikube, il 2.3.3, viene fatto il deploy di un singolo nodo su cui andare a schedulare i pod; è bene sottolineare che rimane comunque il control plane a decidere su quale nodo schedulare il pod.

La decisione di tenere il *supervisor* al di fuori del nodo è stata dettata dalle scelte implementative. Questo fa si che il *supervisor* necessiti di essere innescato come un normale programma java. Tuttavia, sarebbe anche possibile portarlo all'interno del nodo e da qui poterne fare il deploy come accade per gli altri componenti. Come si vede in Figura 16 il *supervisor* si interfaccia con il control plane di Kubernetes tramite API.

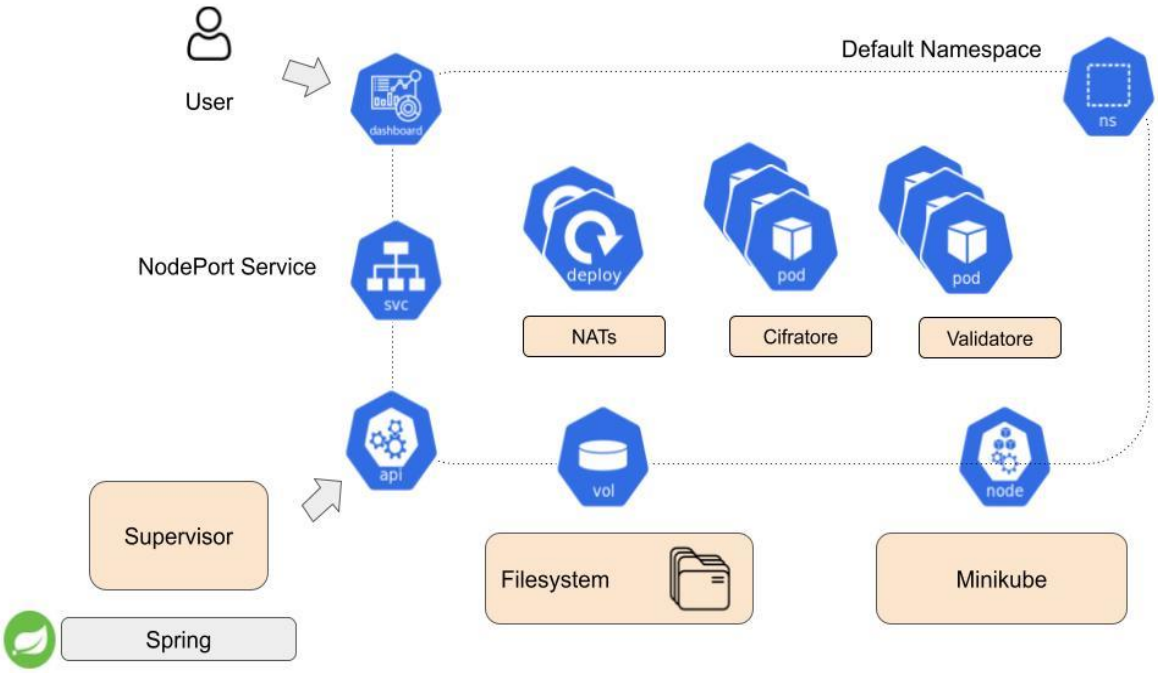


Figura 16: Architettura finale

Osservando sempre la Figura 16 si potrebbe pensare che all'interno del nodo vi siano già dei servizi schedulati e attivi quando viene avviato il *supervisor*. In realtà, l'unico servizio che viene avviato con l'avvio del supervisor è quello di NATs.

Le repliche dei vari pod che sono state inserite, sono solamente per dare un'idea dell'architettura generale. E' comunque bene sottolineare che una situazione simile, ovvero avere già un certo numero di repliche attive dello stesso servizio, avviene quando si utilizza la risorsa del *replicaSet* di Kubernetes, dove viene mantenuto un certo numero di repliche del medesimo servizio attive nello stesso momento per poter fare il *load balancing* e servire le richieste tramite repliche diverse dell'applicazione.

Nel progetto di tesi, invece, si hanno repliche dello stesso servizio in flussi distinti e paralleli. Per rendere meglio l'idea possiamo osservare la

Figura 17: Gestione richieste nella quale viene selezionato un riquadro temporale (da *t5* a *t5+x*) ovvero un'istantanea dell'infrastruttura. In questo lasso di tempo si nota come all'interno del nodo sia stato fatto il deploy di un pod che svolge il servizio di cifratura e due pod che svolgono il servizio di validazione a fronte di tre richieste pervenute agli istanti t2, t3, t4. Questi pod appartengono a tre elaborazioni distinte, in altre parole, non si ha la replica del singolo pod, ma la replicazione del flusso di elaborazione. Infatti, al termine di tutte le elaborazioni dei conferimenti, il numero di pod tornerà ad essere pari a zero, tranne per quello che ospita il *message broker*.

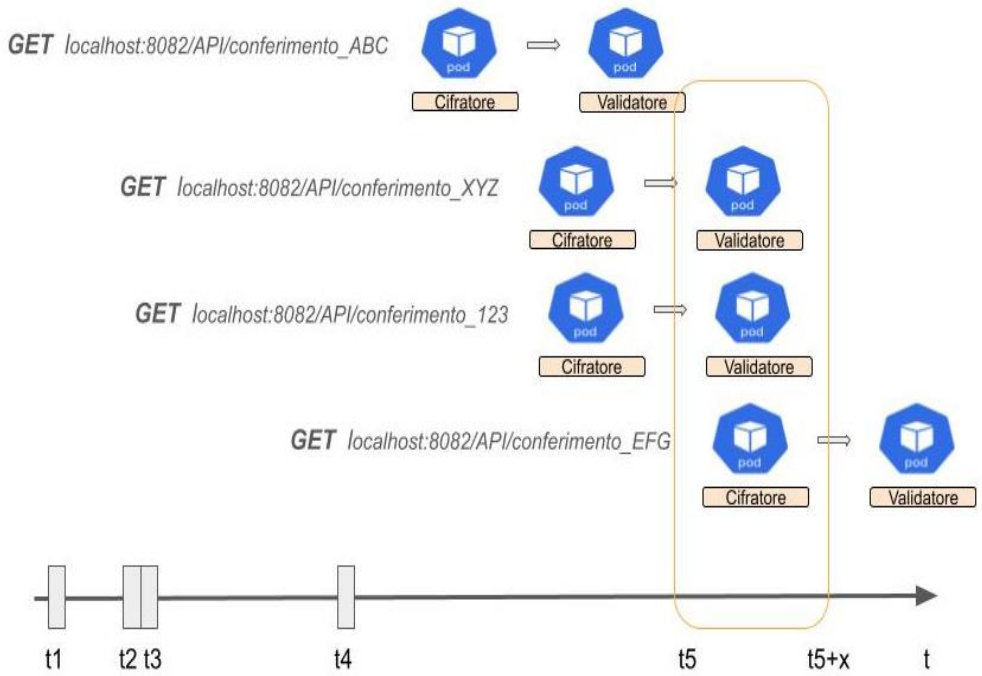


Figura 17: Gestione richieste

L'esempio appena riportato rende più comprensibile ciò che il titolo della tesi vuole sottolineare: la creazione di microservizi *on-demand*, ovvero su richiesta. E' proprio il *supervisor* che, man mano che arrivano le richieste, crea i pod e, quando non ce n'è più bisogno, li "distrugge".

E' da sottolineare anche il lavoro svolto da Spring nella gestione delle interrogazioni REST. All'interno del codice vi è una classe chiamata *controller* che è stata mappata con l'annotazione *@RestController*. Questa particolare annotazione è una specializzazione della annotazione *@Controller*. Con l'utilizzo dell'annotazione *@RestController* non è più necessario mappare i metodi interni con l'annotazione *@ResponseBody*. Detto ciò, all'interno della classe appena citata vi è l'API *"/API/{conferimentoId}"* che permette l'avvio del processo di conferimento.

Quando arrivano più richieste in parallelo il framework Spring, in modo totalmente trasparente al programmatore, utilizza il *multi-thread* per gestire il richieste. Ogni richiesta che arriva è servita da un *thread* differente e se il server su cui lavora Spring è Tomcat, possono essere gestite fino a 200 richieste simultanee.

Nei paragrafi seguenti verranno dettagliate le motivazioni che hanno portato a scegliere un determinato tipo di oggetto Kubernetes piuttosto che un altro, per la schedulazione dei vari microservizi.

Infine, in questa panoramica generale, la dashboard è un altro importante elemento poiché permette di osservare lo stato dei vari oggetti senza dover utilizzare i comandi da linea. Inoltre, ci fornisce un rapido accesso al log del programma e alla shell[37] del contenitore in cui gira l'applicativo. Per il progetto di tesi si è scelto di utilizzare la dashboard standard fornita da Kubernetes, che è possibile schedulare tramite pod all'interno del nodo (in appendice vengono riportate più informazioni riguardanti la dashboard)

Non ultimo, sempre in Figura 16, viene indicato il servizio NodePort che ci permette di contattare i servizi interni al cluster. Il pod in questione verrà rimappato su di una nuova porta, e potrà essere contattato tramite l'indirizzo ip del nodo su cui giace e la nuova porta assegnatagli. Per concludere, sempre nella medesima figura, in alto a destra è stato inserito il simbolo del namespace per indicare che i componenti giacciono all'interno dello stesso Namespace.

3.1.1 I volumi

Entrambi i microservizi necessitano di svolgere azioni di scrittura e lettura su filesystem. Kubernetes ci mette a disposizione la possibilità di montare e smontare delle cartelle presenti sul file system dell'*host* all'interno del container. Le cartelle che vengono utilizzate per l'elaborazione sono:

- una per contenere i conferimenti da elaborare (plain)
- una contenente i file che verranno elaborati (encrypt)
- una in cui è presente la copia privata della chiave asimmetrica della procura (key)
- una in cui vi sono i file xsd (xsd)

Vi sono ancora due cartelle che vengono sempre montate. La prima cartella, il cui nome è config, contiene il file di configurazione di spring (l'*application.yaml*) e il file l'xml che descrive il *layout* e l'output *appender*[38] del log.

La seconda cartella, a cui è stato dato il nome di *logs*, possiede al suo interno i veri e propri file di log. Infatti, tramite il framework Logback si riesce a raccogliere in modo ordinato questi file contenenti i log relativi alle singole elaborazioni su file divisi in base alla data in cui la stessa è avvenuta.

[37] E' il componente che permette di interagire con il sistema operativo in ambienti in cui non è presente un'interfaccia grafica, come il caso dei container.
[38] Il "supporto" su cui viene scritto il log, ovvero se verrà scritto sullo standard output oppure su di un file.

3.1.2 Protocollo della procedura di conferimento

Dopo aver fatto un accenno agli elementi che costituiscono l'architettura e alla caratteristica dei volumi, mi accingo ora a descrivere quello che è il flusso dell'applicazione.

I messaggi che si scambiano i vari servizi formano un vero e proprio protocollo di comunicazione che verrà descritto successivamente. Questo protocollo ha subito diverse variazioni durante la sua stesura e altri miglioramenti potrebbero ancora essere fatti relativamente ad alcuni aspetti. Di seguito avremo un *sequence diagram*[39] corredato da una descrizione su come si articola in termini di scambio di messaggi, mentre nei paragrafi successivi avremo una descrizione più dettagliata di come operano i vari servizi che interagiscono all'interno del protocollo e dei costrutti di sincronizzazione che è stato necessario utilizzare.

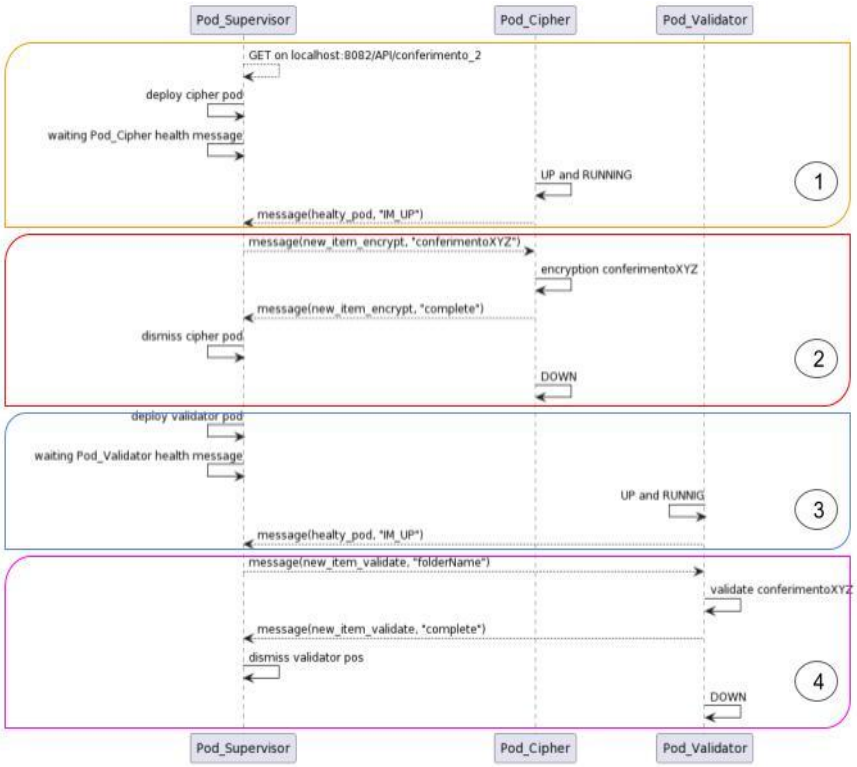


Figura 18: Sequence Diagram[40]

Come possiamo notare dalla Figura 18, l'intero processo inizia quando viene fatta una richiesta di tipo GET[41] all'*endpoint* del supervisor, all'interno della richiesta vi è il nome del

[39] Il diagramma di sequenza è un diagramma previsto dall'UML (Unified Modeling Language) in cui vengono descritti gli scambi di messaggi tra due servizi.
[40] Per la creazione del diagramma è stato fatto uso della fonte: https://plantuml.com/sequence-diagram
[41] E' uno dei verbi utilizzati all'interno dell'architettura REST (Representional State Transfer), inventata all'inizio degli anni 2000 da Roy Fielding. Con questo metodo ci si scambia lo stato degli oggetti tramite i verbi CRUD: POST, GET, UPDATE, PUT.

conferimento da elaborare. Per motivi di chiarezza non è stato possibile inserire anche il *message broker* all'interno dell'immagine; tuttavia, bisogna tenere presente che ogni messaggio passi tramite quest'ultimo.

Nel riquadro giallo possiamo vedere lo scambio di messaggi che porta al *deploy* del servizio di cifratura. In una prima stesura del protocollo è stato necessario aggiungere lo scambio di messaggi che riguardano la "salute" del pod, ovvero quando il servizio di cifratura è *up* e potrà effettivamente ricevere il messaggio con il nome del conferimento. Nel caso in cui il *supervisor* lo avesse inviato senza che il servizio di cifratura fosse stato pronto con la subscribe a riceverlo, il messaggio sarebbe andato perso. Come vedremo all'interno del paragrafo 4.2 dedicato agli sviluppi futuri, questo scambio di messaggi inerente allo stato di salute del pod potrà essere rivisto se non addirittura eliminato.

Nel riquadro rosso viene ricevuto il messaggio con il nome del conferimento e il servizio parte con la cifratura dello stesso mentre il supervisor rimane in attesa di leggere il nuovo nome della cartella che andrà ad essere validata dal secondo servizio. Nel caso in cui sopraggiungano errori nella fase di cifratura (Figura 19) il pod segnalerà al supervisor l'eventuale errore e, prima di essere eliminato, libererà la memoria da eventuali elaborazioni corrotte.

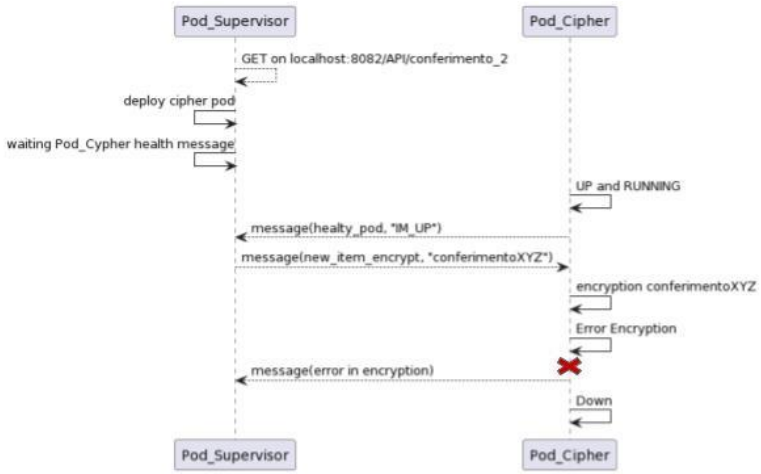


Figura 19: Errore nel servizio di cifratura

Nel riquadro blu abbiamo il deploy del pod che si occuperà del servizio di validazione. Anche in questo caso verrà scambiato un messaggio che riguarda l'avvenuta correttezza del deploy.

Infine, il riquadro fucsia riporta il messaggio con il nome della cartella da cifrare da parte del *supervisor* e come nel caso precedente, un messaggio di errore verrà segnalato al *supervisor* nel momento in cui vi fossero problemi con la validazione dei contenuti. Al termine del flusso verranno cancellati i messaggi dalla memoria del *message broker*.

3.1.3 Oggetti Kubernetes

Durante lo studio dell'architettura è stato necessario decidere quale oggetto utilizzare per schedulare i microservizi. Come già accennato all'interno della paragrafo dedicato alle tecnologie sviluppate 2.3.1.4, Kubernetes mette a disposizioni vari tipi di astrazioni (chiamate anche oggetti o risorse): *pod*, *deployment*, *job*, *replicaSet*. Queste hanno caratteristiche diverse che possono andare a plasmare le varie situazioni di utilizzo.

Il deployment è un'astrazione che gestisce il ciclo di vita di un pod. Permette di rilasciare nuove versioni dell'applicazione, che gira all'interno del pod, senza che la stessa debba essere fermata; permette anche di rischedulare la vecchia versione nel caso in cui la nuova abbia degli errori senza mai interrompere il servizio. Questo oggetto permette la rischedulazione automatica del pod non appena quest'ultimo cade. Mi è sembrato utile sfruttare questo comportamento per la schedulazione del servizio del *message broker*. Infatti nel caso in cui quest'ultimo dovesse cadere automaticamente verrebbe rischedulato, permettendo di mantenere *up* lo scambio tra microservizi.

Per i microservizi, invece, si è optato per l'utilizzo della risorsa fondamentale: il pod. Questa scelta è stata motivata dal fatto che il servizio del cifratore come quello del validatore non hanno la necessità di rimanere attivi una volta terminati e posson quindi essere eliminati.

Il pod, tra le sue funzionalità, permette di scegliere tra diverse policy di *restarting*: *always*, *onFailure*, *never*. Settando come policy quella *never* riusciamo a eliminare il pod una volta terminata la sua procedura senza che questo provi ad essere automaticamente rischedulato. All'interno del capitolo che riguarda gli sviluppi futuri, verrà ampliata la tematica della caratteristica del *restarting* del pod, che permetterà di inserire della resilienza all'interno dei vari servizi.

3.2 I componenti

Dopo la breve introduzione a quella che è l'architettura generale, passo a descrivere i vari attori in gioco: NATs, il cifratore, il validatore e il *supervisor*, contestualizzando il tutto con quello che è il flusso di elaborazione.

3.2.1 Il message broker

In questa prima versione dell'architettura è il *supervisor* che va a fare il *deploy* del *message broker*. Tramite le annotazioni che ci fornisce Spring ho mappato la classe *MessageBrokerDeployer* con l'annotazione *@Service*. Tramite questa annotazione (che è simile al lavoro svolto da *@Component*) Spring si va a istanziare un oggetto di questo classe, senza che vi sia la necessità di farne la creazione con un costruttore (con ciò possiamo cogliere il paradigma dell'IOC, citato nel paragrafo sulle tecnologie 2.1.1).

Successivamente, è necessario che l'oggetto istanziato vada a schedulare NATs tramite il Manifest apposito. In questo caso viene utilizzata anche la risorsa *service* di Kubernetes, al fine di configurare il servizio *nodePort* per poter permettere la comunicazione dall'esterno.

Con questa configurazione si vanno a mappare due servizi: il primo si trova sulla porta 4222 ed è il servizio di broker, mentre sulla porta 8222 si ha un servizio basilare di monitoring, che ci permette di osservare le connessioni attive e il numero di messaggi scambiati sotto forma di pagina Html.

Per utilizzare JetStream è necessario fare delle configurazioni aggiuntive, infatti non basta creare la variabile di classe *Connection* e connettersi al server NATs passandogli l'indirizzo ip e la porta. Per la creazione dello stream bisogna aggiungere alcune informazioni sia obbligatorie che opzionali. Tra quelle obbligatorie abbiamo:

- il nome dello stream
- il nome del / dei *subject*, ovvero i canali (*topic* quando si utilizza Kafka) su cui ci si va a mettere in ascolto per poter scambiare i messaggi.
- il mezzo su cui salvare i messaggi che vengono inviati al *message broker*, che può essere la memoria, oppure un file.

Tra quelle non obbligatorie abbiamo il numero massimo di messaggi che può essere salvato, la policy che porta a scartare il messaggio più vecchio, la dimensione massima che un messaggio possa avere e anche il tempo massimo in cui possa risiedere all'interno della coda, tutte queste caratteristiche le possiamo ritrovare nella Figura 15 riassunte sottoforma di richiesta al server NATs.

Inoltre, NATs per essere tollerante ai guasti ci permette di avere un certo numero di repliche del server garantendoci fino a cinque copie del singolo messaggio (salvato all'interno delle

differenti repliche). Con questa *feature*, NATs creare una sorta di database distribuito che è totalmente trasparente a chi lo utilizza.

3.2.2 Microservizio di cifratura

Il servizio di cifratura, come già detto all'interno del paragrafo 3.1.2 (protocollo della procedura di conferimento), viene attivato dal *supervisor* tramite la schedulazione di un Pod. Prima di addentrarmi nella spiegazione di come funziona il servizio, ritengo sia opportuno fare un breve accenno a quello che è l'iter per la creazione del container, utilizzato poi da Kubernetes per il deploy dei pod.

Come già spiegato nel paragrafo 2.3.1, Kubernetes fa il *deploy* delle applicazioni all'interno dei *container* tramite l'utilizzo dei Pod. Per la creazione dei container a loro volta si ha la necessità di avere delle immagini da cui poter partire. Docker ci permette di avere un *registry*[42] in locale con cui lavorare, mentre quando si è in produzione è necessario avere dei server da cui Kubernetes possa scaricare le immagini nel caso in cui non le possa trovare in locale (essendo l'ambiente di produzione, quasi sempre, un infrastruttura cloud esterna di tipo SaaS come Azure o AWS, le immagini necessarie ai container non sono mai presenti in locale).

All'interno del Manifest è presente un campo con il nome *imagePullPolicy* che può assumere il valore *ifNotPresent*. Questo valore ci porta a scaricare l'immagine da un server nel caso in cui non sia presente l'immagine in locale. Un altro valore che può assumere è *always*, portandoci a scaricare l'immagine nonostante sia presente già in locale. Quest'ultima tecnica potrebbe essere utile nel momento in cui le repliche che si vogliono utilizzare debbano essere le ultime versioni presenti.

Aggiungo ancora che vi sono *image registry* gratuiti e servizi cloud a pagamento per il *management* delle immagini su piattaforme cloud come Azure o AWS.

Per terminare questo accenno, elenco brevemente i passaggi per la produzione di queste immagini. Il primo passaggio è la creazione del pacchetto Jar tramite Maven, contenente tutto il codice e le librerie utilizzate. Successivamente, tramite un Dockerfile, si creerà l'immagine specificandone la versione e la posizione del file Jar da cui partire per la creazione della stessa. Nel caso in cui fossimo in produzione, sarebbe necessario fare l'upload dell'immagine sul *cloud registry*.

[42] Registro

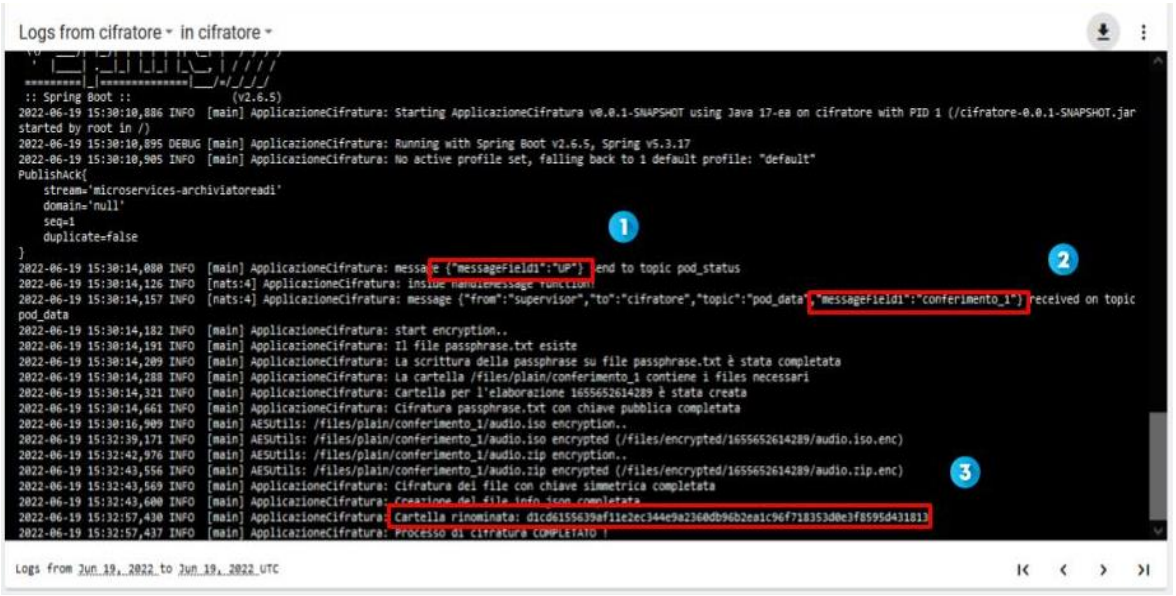


Figura 20: Log servizio di cifratura

In Figura 20 possiamo vedere il log che viene generato dal programma, tramite la dashboard di Kubernetes durante il suo funzionamento. Vediamo inizialmente lo scambio dei messaggi che riguarda l'effettivo avvio del pod con il messaggio "up" (1), poi leggiamo il messaggio con il nome del conferimento da elaborare (2). Infine, il messaggio con il nuovo nome della cartella che contiene i dati cifrati (3).

All'interno del funzionamento del microservizio è da sottolineare l'utilizzo dei costrutti di sincronizzazione; infatti, quest'ultimi sono necessari durante il periodo in cui si attende che venga ricevuto il messaggio con il nome del conferimento da elaborare. Come primitiva di sincronizzazione ho optato per la classe CountDownLatch.

La classe CountDownLatch ci permette di bloccare un o più thread finché non sia stato completato un determinato *task*. Quando si inizializza un oggetto di questo tipo bisogna specificare un contatore che solitamente corrisponde al numero dei *thread* con cui si sta lavorando. Quindi si andrà a chiamare la funzione *countDown()* al termine di ogni *thread* garantendo che un thread che aveva chiamato la *wait()* rimarrà bloccato fin tanto che tutti i thread non saranno terminati (avendo chiamo *countDown()* al loro interno).

All'interno del servizio dedito alla cifratura è stata inizializzata la variabile *latch* di tipo *CountDownLatch* con il counter pari ad uno. Successivamente è stata chiamata la funzione *await()* dopo aver istanziato il *Dispatcher* e il *MessageHandler*. In questo modo il thread principale rimane in attesa che qualcuno lo sblocchi per poter iniziare la procedura di cifratura. La chiamata alla funzione *countDown()* avverrà all'interno del *MessageHandler*,

ovvero nel momento in cui verrà ricevuto il messaggio con le informazioni per proseguire la procedura.

3.2.3 Microservizio di validazione

Il microservizio di validazione lavora in modo simmetrico a quello che viene fatto dal servizio di cifratura, anche in questo caso viene fatta la schedulazione del pod da parte del *supervisor*. All'interno della Figura 21, avviene in modo speculare quello che accade per l'altro microservizio, infatti possiamo notare le medesime fasi: quella in cui viene mandato un messaggio che riguarda l'attività del pod, quella dell'attesa di un certo dato per poter iniziare la validazione del conferimento ed infine l'esito dell'operazione.

Come già spiegato all'interno nel capitolo uno, il processo della validazione serve a controllare che ciò che è stato cifrato sia stato effettivamente cifrato correttamente prima di venire scritto in modo definitivo sul supporto di memorizzazione. Oltre ai controlli sulla cifratura dei dati vengono fatti dei controlli sulla conformità dei metadati che si trovano all'interno degli xml. Infatti, su questo microservizio viene montata anche una cartella contenente i file xsd per operare i controlli descritti precedentemente oltre alla cartella che andrà a contenere i dati cifrati.

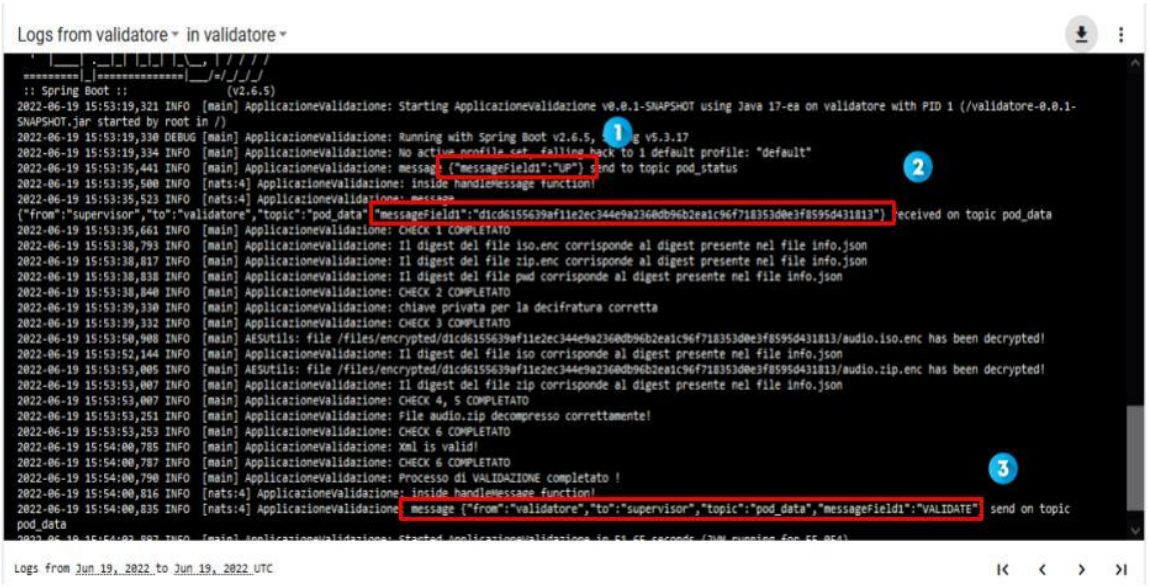


Figura 21: Log servizio di validazione

Al termine della validazione verrà notificato al supervisor se la procedura è andata a buon fine oppure no. Il fatto di aver disaccoppiato i due servizi ci permette anche di poter decidere quale delle due procedure rischedulare a seconda degli errori riscontrati.

E’ importante ancora menzionare come i vari microservizi si connettano al server NATs. Infatti, essendo questi dei pod, hanno come loro comportamento insito quello dell’essere effimeri, e quindi il fatto che possano terminare inaspettatamente e successivamente essere rischedulati senza che sorgano problemi di connettività dovuti al cambio degli indirizzi ip. Questo si ottiene grazie all’utilizzo delle label assegnate ai container, che svolgono un lavoro simile ad un DNS all’interno del cluster. In questo modo i vari servizi si contattano tra di loro per mezzo di nomi e non più di indirizzi ip che potrebbero cambiare nel momento in cui il servizio tornasse attivo. Questa caratteristica viene sfruttata nel contattare il *message broker*, rendendo non più necessario dovergli assegnare un indirizzo statico.

3.2.4 Servizio del supervisor

Il *supervisor* contiene a tutti gli effetti la logica dell’applicazione. E’ proprio lui che, man mano che arrivano le richieste, schedula i pod e reagisce con eventuali interruzioni nel caso in cui sopraggiungano errori.

Anche in questo caso è necessario l’utilizzo della sincronizzazione, in particolare è stato necessario aspettare i due servizi in tempi diversi. Nel caso in cui si fosse deciso di utilizzare di nuovo l’oggetto di tipo *CountDownLatch* ne avremmo dovuti istanziare due. Infatti, per come è stato pensato, non è possibile resettarlo assegnando un nuovo valore alla variabile *count*.

Inoltre, andando a utilizzarne due, si sarebbe anche dovuto distinguere all’interno dell’ *handler* di messaggi il momento in cui si stava attendendo i messaggi da parte di un servizio oppure da parte dell’altro, in modo tale da poter agire sul *countdown* corretto e non incorrere nel blocco del programma se si fosse agito sul *latch* sbagliato.

Per evitare quindi di dover inserire altra logica all’interno dell’*handler* si sono utilizzate le *CycleBarrier*. Le barriere cicliche permettono a un certo numero di *thread* deciso a priori di attendersi l’un l’altro e, solamente nel momento in cui tutti i *thread* avranno chiamato la funzione *await(),* verranno sbloccati. Questo comportamento è simile a quello del CountDownLatch, con la differenza che, un volta che si sono sbloccati tutti i *thread,* questi possono tornare ad aspettarsi un numero infinito di volte, senza la necessità di dover fare alcun reset della variabile, ne doverne istanziare una nuova. Da questa caratteristica deriva l’aggettivo cicliche.

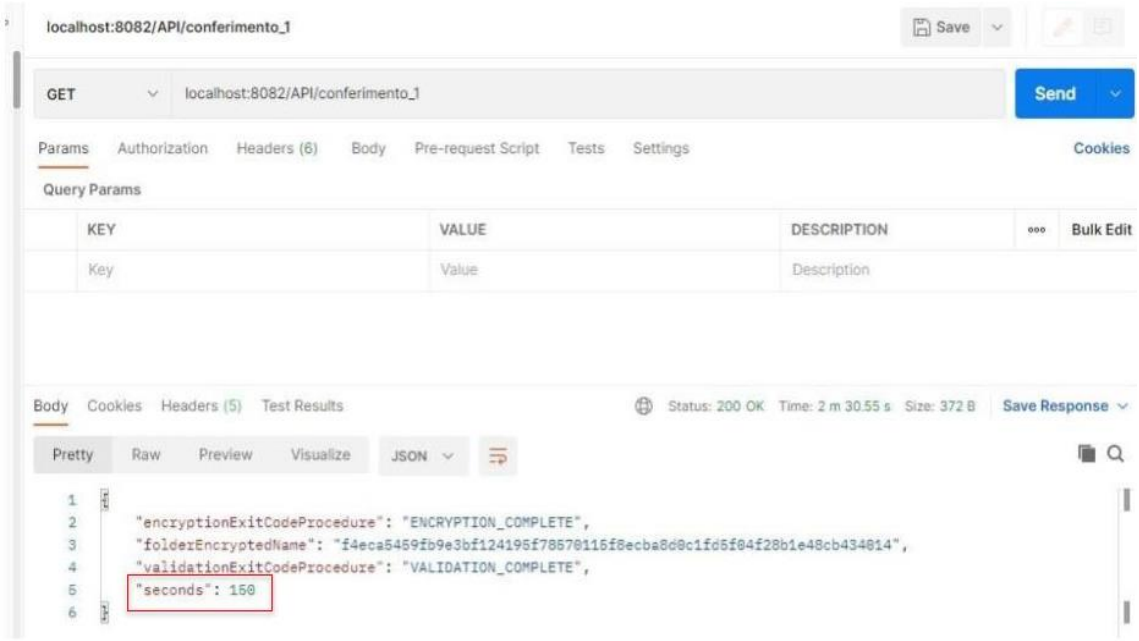


Figura 22: Dati formato Json che vengono ritornati

In Figura 22 è stata riportata la risposta alla Get che si riceve nel momento in cui viene terminata l'intera elaborazione del conferimento. All'interno della richiesta pervengono, oltre alla stringa che contiene il nuovo nome della cartella cifrata, il tempo di durata dell'elaborazione ovvero 150 secondi. Questo tempo indica la durata da quando arriva la richiesta fino alla terminazione di quando viene dismesso il pod per il servizio della validazione.

Capitolo quarto

4 Conclusioni

Il capitolo conclusivo della tesi è volto ad analizzare il sistema architetturale che si è creato, a verificare se gli obiettivi raggiunti combacino con quelli che ci si era prefissati e ad analizzare eventuali sviluppi futuri dell'architettura.

4.1 Obiettivi raggiunti

Tra gli obiettivi vi era sicuramente la necessità di disaccoppiare il software e far in modo che il risultato che ne fosse uscito potesse essere impiegato all'interno del pattern a microservizi. Questo obiettivo, che possiamo dire di aver raggiunto, si aggiunge a quello dell'utilizzo di Kubernetes come pilota di servizi a richiesta.

Durante il periodo di studio prima dello sviluppo del codice, si è analizzato il problema e studiata una possibile soluzione per capire se l'utilizzo di Kubernetes in questa maniera fosse qualcosa di già visto oppure no. La scarsità delle informazioni trovate è stata denotata anche dal fatto che la libreria *client*, paragrafo 2.3.2, che fornisce le funzioni per interfacciarsi con le API Kubernetes non è supportata dallo stesso Kubernetes ma da enti esterni.

Nonostante la poche informazioni a riguardo di questa metodologia, si è riusciti comunque ad ottenere dei buoni risultati riuscendo ad utilizzare l'orchestratore come pilota di servizi.

Infine, è da sottolineare l'utilizzo di NATs, un *message broker* un po' di nicchia che però possiede diverse potenzialità.

4.2 Sviluppi Futuri

Questo progetto di tesi è un prototipo per lo studio del funzionamento di microservizi attivabili e disattivabili on-demand; tuttavia, la fattibilità di tale prototipo può essere un trampolino di lancio per degli sviluppi futuri. Infatti, una caratteristica che non è stata trattata in modo esaustivo e che invece è fondamentale nei progetti che fanno uso degli orchestratori è quella della resilienza dei vari componenti.

Per il momento l'architettura non dispone di fattori di resilienza per i servizi che offrono la funzione di cifratura e quella di validazione, solamente il server che ospita il message broker ha resilienza. Questa, infatti, viene perseguita tramite l'astrazione del deployment e il

salvataggio dei messaggi su file, per cui nel caso in cui terminasse in modo improvviso verrebbe rischedulato e i messaggi salvati sarebbero di nuovo fruibili.

Per avere una caratteristica di resilienza completa bisognerebbe aggiungere ridondanza al server NATs. Infatti, si potrebbero schedulare più repliche dello stesso riuscendo a mantenere il servizio funzionante anche nel caso della caduta di uno dei server.

Per quanto riguarda i due microservizi si potrebbe andare a modificare il protocollo e far in modo di sfruttare la funzionalità automatica di restarting dei pod, andando a settare *onFailure*. In questa maniera, in caso di errore, il pod ripartirebbe e la presenza dei messaggi in memoria permetterebbe un nuovo tentativo di elaborazione.

Appendice

5 Appendice

Di seguito andrò ad elencare i passi per la riproduzione dell'infrastruttura su cui poggia l'architettura ottenuta dal lavoro di tesi, andando anche a spiegare nel dettaglio il procedimento di creazione dei vari container utilizzati da Kubernetes in modo da rendere riproducibile l'ambiente di lavoro.

5.1 Minikube on Windows 10 Educational

L'architettura a microservizi si poggia completamente sull'utilizzo dell'orchestratore Kubernetes, nello specifico viene utilizzata la tecnologia di Minikube.

Minikube può essere configurato per lavorare con diversi tipi di driver, tra cui Docker e VirtualBox che sono quelli più noti.

Nel caso specifico ho utilizzato i driver di VirtualBox (Oracle), siccome la funzione di nodePort non è stato possibile farla funzionare con i driver Docker. Minikube di default fa il deploy di un singolo nodo ma è possibile schedularne più di uno.

Come si legge dal titolo, l'intera infrastruttura lavora su Windows 10 la cui versione è quella Educational, che supporta l'opzione *Hyper-V* per la virtualizzazione. Altra versione che fornisce questa opzione è quella *Enterprise*, mentre la versione *Home* non la supporta.

Per l'installazione di Minikube è necessario seguire i passi qui elencati:

1. Dal Pannello di *Controllo di Windows -> Programmi e Funzionalità -> Attiva o Disattiva Funzionalità di Windows*
2. Si apre un pannello dalla quale è necessario spuntare la checkbox *Hyper-V*, poi riavviare il sistema

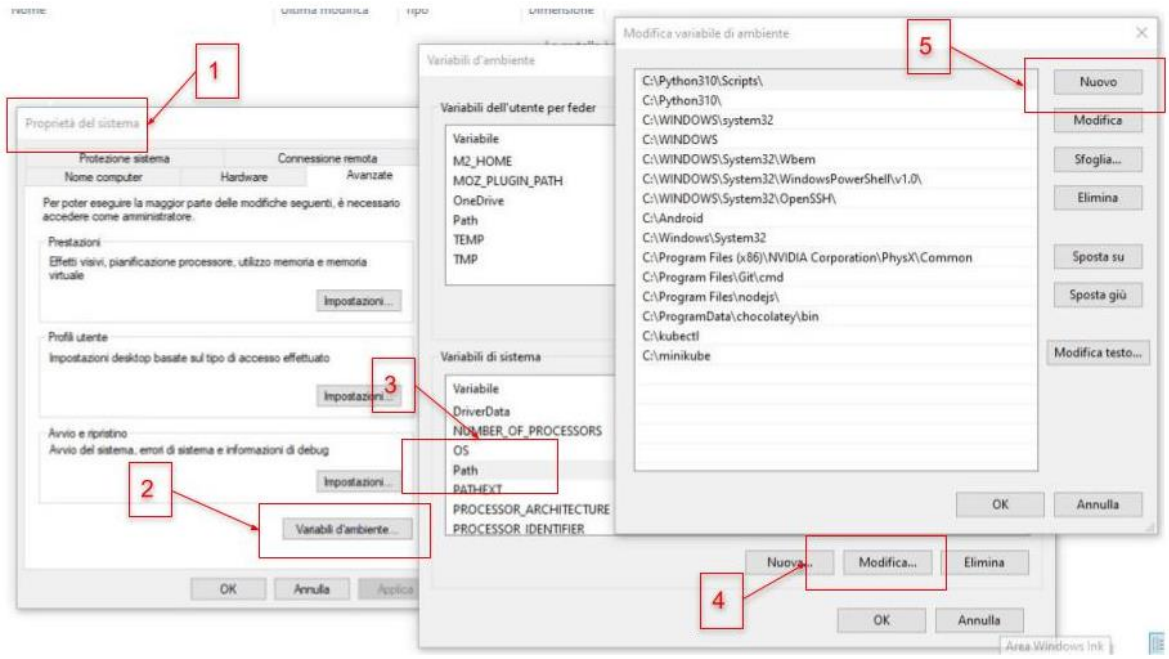
3. Fare il download e l'installazione di Oracle VM VirtualBox (https://www.virtualbox.org/wiki/Downloads)
4. Fare il download e l'installazione dell'applicazione Docker Desktop (https://docs.docker.com/desktop/windows/install)
5. Fare il download di *minikube.exe* e *kubectl.exe* dal sito di Kubernetes. Porre i downloads secondo lo schema all'interno della cartella minikube (da creare) sotto il disco c:

   c:\minikube

   _ *minikube.exe*

   _ *kubectl.exe*

6. Aggiungere la variabile di sistema seguendo *Pannello di controllo -> Sistema e Sicurezza -> Proprietà di Sistema -> Variabili di Sistema* (in basso a destra) e poi aprire path e aggiungere la seguente variabile: *C:\minikube.* Successivamente riavviare.



7. Dalla *prompt* di comando di Windows, con i privilegi amministrativi si dia il seguente comando che segue e riavviare il sistema.

   bcdedit /set hypervisorlaunchtype off

8. Sempre dal *prompt* dei comando con privilegi amministrativi lanciare il seguente comando, che andrà a creare la macchina minikube (visibile anche dall'home page dell'applicazione VM VirtualBox)

   minikube start --driver=virtualbox

5.1.1 Comandi Minikube

Di seguito vengono elencati alcuni comandi utili per capire lo stato del nodo

minikube start -> per avviare minikube.

minikube stop -> per terminare minikube .

minikube ip -> per conoscere l'ip del nodo (utile per la funzione NodePort).

minikube status -> per conoscere lo stato dei componenti (kubelet, apiserver, docker-env).

minikube ssh -> per entrare all'interno della *shell* della macchina virtuale.

5.2 Docker Engine

Siccome si andrà ad utilizzare i driver di Minikube e non quelli di Docker Desktop per la creazione del nodo Kubernetes è necessario dare il comando che segue dalla shell, nel momento in cui si voglia utilizzare l'*engine* Docker per la creazione dei contenitori. Tramite questo comando si va a dire all'engine di lavorare nel contesto di minikube.

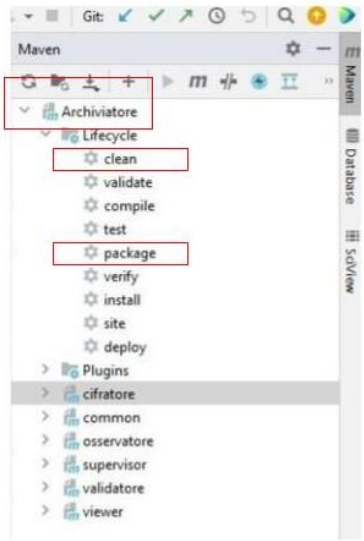
minikube docker-env | Invoke-Expression

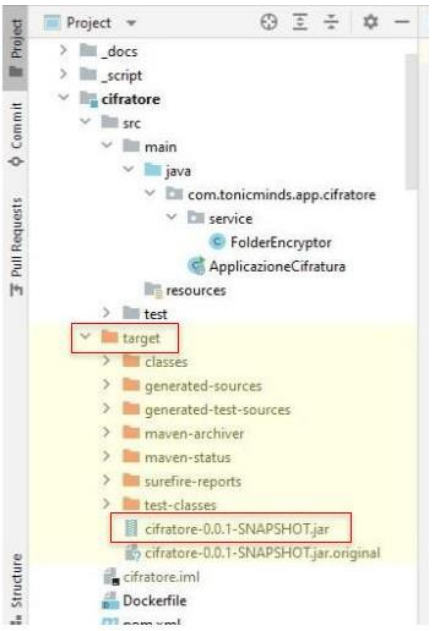
E' necessario darlo su ogni linea di comando nuova che viene aperta e dalla quale si vogliano creare dei container.

5.3 Creazione dei file Jar

Dopo aver aperto il progetto tramite IntelliJ si creino i file Jar che conterranno tutte le librerie utilizzate dall'applicazione tramite Maven.

Il progetto è formato a moduli, anche se alcuni moduli non vengono adoperati è necessario fare il packaging di tutto il progetto. Dalla barra laterale di IntelliJ si apra Maven e si clicchi su *package*, nel caso di eventuali errori dare il comando *clean* prima di ripetere il *packaging*. Nell'immagine di sinistra che segue, si possono vedere i comandi Maven che IntelliJ mette a disposizione sottoforma di bottoni.





Al termine dell'operazione si controlli che siano stati creati i files Jar sotto la cartella target all'interno di ogni modulo. La figura in alto a destra mostra la cartella *target* con all'interno il file Jar, frutto dell'elaborazione.

5.4 Creazione delle Immagini

I moduli che necessitano la creazione delle immagini sono Cifratore, Validatore. Il Supervisor, per il momento viene lanciato al di fuori del nodo.

1. Spostandoci all'interno della sottocartella cifratore si faccia la build del modulo tramite il docker file. Si lanci da *command line* il seguente comando

   docker build -t cifratore .

2. Spostandoci all'interno della sottocartella validatore si faccia la build del modulo tramite il docker file. Si lanci da *command line* il seguente comando

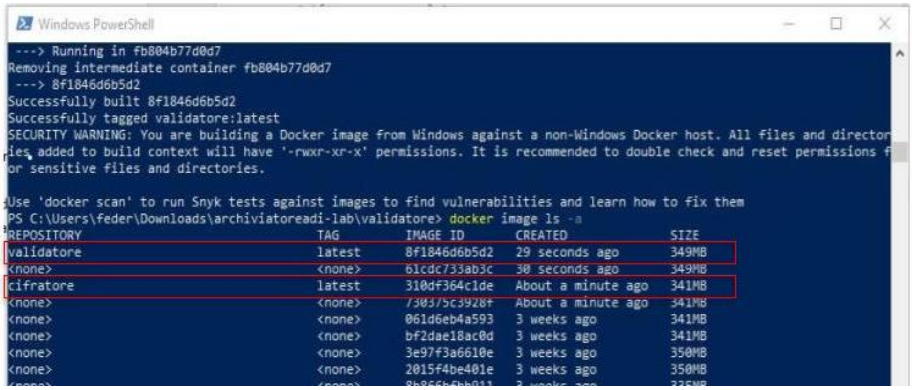
   docker build -t validatore .

5.4.1 Comandi Docker

Di seguito vengono elencati alcuni comandi utili per la gestione delle immagini in locale

docker image rm <image-name> -> per eliminare l'immagine di un container

docker image ls -a -> per controllare l'effettiva creazione delle immagini



5.5 Mounting di una cartella Minikube

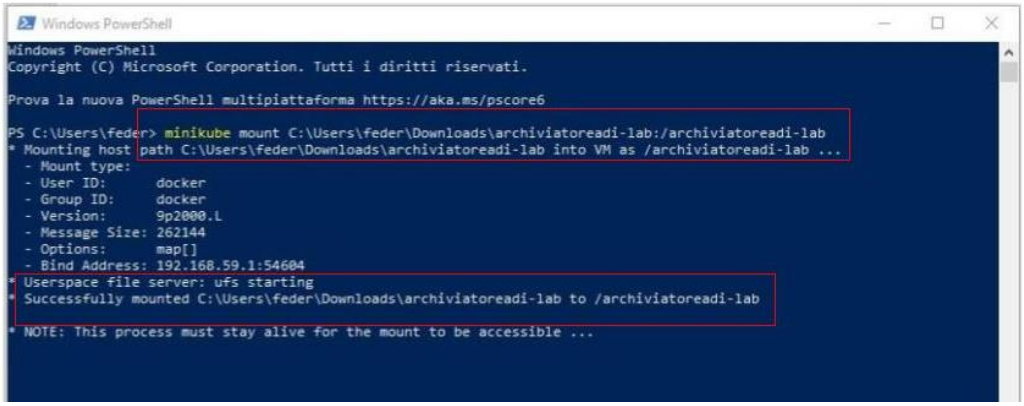
Siccome Kubernetes non gira direttamente sul pc ma vi è Minikube che virtualizza l'infrastruttura su cui appoggiarsi, è necessario fare il mounting della cartella (di Windows) che poi verrà utilizzata dall'applicazione. Verrà fatta una specie di by-pass, ovvero si deve fare in modo che all'interno del filesystem di Minikube vi sia la cartella desiderata.

Nel mio caso, siccome il progetto si trova sotto la cartella dei Downloads C:\Users\feder\Downloads\archiviatoreadi-lab si lancerà il comando seguente.

minikube mount <FOLDER_PATH_HOST>:<FOLDER_PATH_CONTAINER>

Il path risultante è *C:\Users\feder\Downloads\archiviatoreadi-lab:/archiviatoreadi-lab* .

Se tutto funziona il processo indicherà che la shell deve rimanere attiva per poter funzionare. Nel caso in cui si volesse controllare l'effettiva presenza della cartella montata, entrando con il comando *minikube ssh* all'interno della *shell* del container e lanciando il comando *ls -la*, non verrà visualizzato nulla. Nonostante ciò, il comando *cd /archiviatoreadi-lab* andrà a buon fine e dall'interno della cartella si potranno vedere tutti i file ripetendo il comando *ls -la* .



5.6 Avvio dell'applicazione

L'avvio si può fare tramite il tasto start a fianco dalla classe SupervisorApplication, nel file SupervisorApplication.java. In questo modo si evita di dover andare a creare una cofigurazione per avviare il programma.

Prima di avviare l'applicazione controllare che sia presente almeno un conferimento (all'interno della cartella */files/plain*) che possa essere elaborato. L'avvio del supervisor porta al deployment di nats con un service che permette il servizio NodePort.

5.7 Avvio elaborazione

L'applicazione dopo essere stata avviata, espone l'endpoint

/API/<nome-conferimento>

All'interno della cartella *files/plain* vi è il *conferimento_1*. La url a cui fare una GET è

http://localhost:8082/API/conferimento_1

Per fare la richiesta GET è stato utilizzato il programma *PostMan*.

5.8 Termine dall'applicazione / crash

Il server NATs, una volta lanciato il *supervisor*, rimarrà sempre attivo. Quindi se si volesse far ripartire l'applicazione sarebbe necessario che si eliminassero sia i pod (nel caso in cui si fossero rotti durante l'esecuzione), i deployment (quindi il server Nats) ed il service con la configurazione di rete. Nel caso in cui si facesse ripartire il *supervisor* senza aver eliminato i vari oggetti, il programma si interromperà con l'emissione di vari errori.

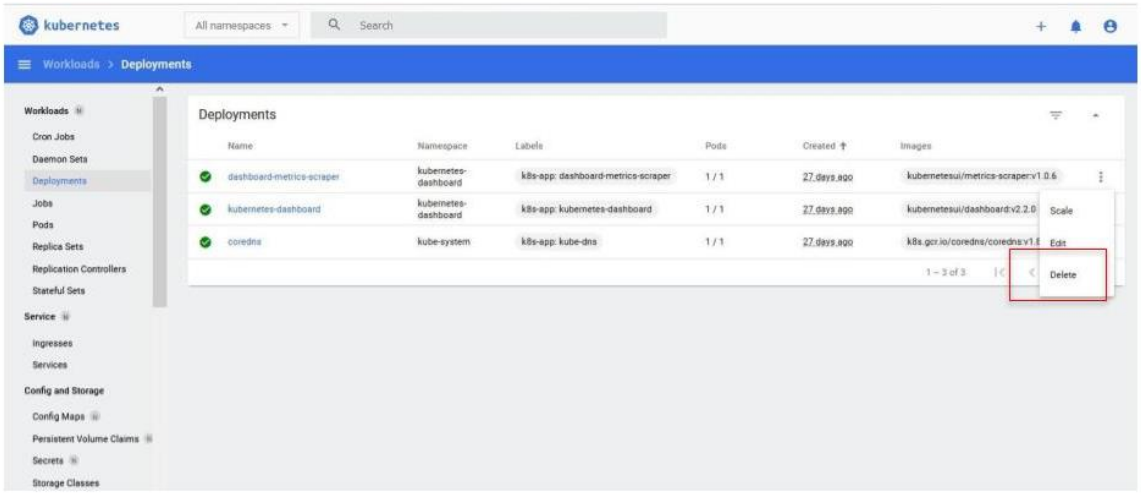
Per eliminare gli oggetti è possibile agire sia da linea di comando tramite i comandi seguenti che debbono essere lanciati dalla cartella i cui sono presenti i deployment.

kubectl delete -f nats-deployment.yaml

kubectl delete -f nats-service.yml

kubectl delete -f cifratore-pod.yaml

Il secondo metodo per l'eliminazione è quello tramite la dashboard



5.9 Dashboard UI

Kubernetes mette a disposizione una propria interfaccia grafica, Dashboard UI. Questa interfaccia ha il formato di una pagina web (viene usato il pattern della *single page application*). Tramite questa dashboard si può fare il deploy, la cancellazione e la modifica delle risorse senza dover andare ad utilizzare continuamente i comandi da *command line*.

Questa interfaccia ci permette di avere una panoramica generale delle risorse che vengono utilizzate e di quelle che stanno girando all'interno del cluster. Di seguito le operazioni per la schedulazione all'interno dei pod

1. Avviare il proxy tramite il comando che segue, questo permette di accedere alla dashboard dall'esterno del nodo.

   kubectl proxy

2. Il comando *kubectl* permette anche di schedulare risorse tramite Manifest che si trovano hostati su server remoti.

   kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.2.0/aio/deploy/recommended.yaml

Cliccando sul seguente link si verrà reindirizzati alla pagini di login della dashboard, dove si potrà entrare fornendo un token oppure un certificato.

localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/error?namespace=default

Per la creazione di un'utenza che permetta l'accesso tramite token utilizzare i due Manifest presenti all'interno della cartella *deployments*:

1. *kubectl apply -f dashboard-adminuser.yaml*
2. *kubectl apply -f dashboard-clusterrolebinding.yaml*

Tramite il comando che segue sarà possibile vedere l'elenco dei segreti, tra questi il token di default per l'accesso.

kubectl get secrets

Ringraziamenti

Giunto al termine del mio percorso universitario ci tenevo molto ringraziare tutte le persone che mi sono state vicine e mi hanno aiutato lungo questo cammino, a tratti non sempre facile.

In primis mia mamma Chiara che mi ha sempre supportato credendo in me, mio papà Giulio e le sorelle Marta e Francesca che mi hanno fornito sempre una spalla su cui appoggiarmi.

Un grazie speciale va ai miei nonni RosaMaria, Anna, Paolo e Luigi una grande esempio di amore altruistico e di dedizione al lavoro.

Gli amici dell'università con cui le giornate di studio sono sembrate meno pesanti: Robin, Rostislav, Michele e Alessio e quelli di sempre che mi hanno accompagnato fin dall'inizio: Dario, Emanuele, Samuele e Eugenio.

Un grazie particolare va anche a mia cugina Martina, ad Alice e a tutti gli altri parenti che mi sono stati a fianco.

Infine, intendo ringraziare il professor Giovanni Malnati per l'opportunità datami e David per i preziosi consigli durante lo svolgimento della tesi.

Bibliografia

[1] Wikipedia (2012), «Spyware - Wikipedia», [Online]. Available: https://it.wikipedia.org/wiki/Spyware. [Consultato il giorno 22 Febbraio 2022].

[2] Wikipedia (2011), «.iso - Wikipedia», [Online]. Available: https://it.wikipedia.org/wiki/.iso. [Consultato il giorno 5 Maggio 2022].

[3] Wikipedia (2010), «ZIP - Wikipedia», [Online]. Available: https://it.wikipedia.org/wiki/ZIP_(formato_di_file). [Consultato il giorno 5 Maggio 2022].

[4] FileTypeAdvisor (nd), «XSD - FileTypeAdvisor», [Online]. Available: https://www.filetypeadvisor.com/it/extension/xsd. [Consultato il giorno 10 Maggio 2022].

[5] LoSviluppatore (2018), «Microservices Architecture – LoSviluppatore», [Online]. Available: http://losviluppatore.it/microservices-architecture-il-pattern-architetturale-emergente-per-le-grandi-applicazioni-moderne/. [Consultato il giorno 15 Marzo 2022]

[6] Spring (nd), «Spring - Spring», [Online]. Available: https://spring.io/. [Consultato il giorno 20 Febbraio 2022]

[7] LogBack (nd), «LogBack Project - LogBack», [Online]. Available: https://logback.qos.ch/. [Consultato il giorno 10 Marzo 2022]

[8] Docker (nd), «Get Started - Docker », [Online]. Available: https://docs.docker.com/get-started/. [Consultato il giorno 2 Marzo 2022]

[9] Kubernetes (2018), «Kubernetes Component - Kubernetes», [Online]. Available: https://kubernetes.io/docs/concepts/overview/component. [Consultato il giorno 4 Aprile 2022]

[10] Minikube (2020), «Minikube start - Minikube », [Online]. Available: https://minikube.sigs.k8s.io/docs/start/. [Consultato il giorno 10 Maggio 2022]

[11] Nats (2019), «CoreNats - Nats», [Online]. Available: https://docs.nats.io/nats-concepts/core-nats. [Consultato il giorno 16 Maggio 2022]

[12] Nats (2019), «JetStream - Nats», [Online]. Available: https://docs.nats.io/nats-concepts/jetstream. [Consultato il giorno 16 Maggio 2022]