POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Soluzione basata su Quarkus per migliorare le performance del modulo BFF su un'architettura a microservizi

Relatore

Prof. Luca ARDITO

Candidato

Carlo VITALE

Ottobre 2023

Sommario

Il progetto di tesi propone di individuare una nuova soluzione per il modulo Backend-for-Frontend (BFF) di NPRI, la Nuova Piattaforma Regionale di Integrazione gestita da Healthy Reply, azienda sotto la quale è stato portato avanti il progetto, per conto della Regione Lombardia.
NPRI si offre di servire agli enti sanitari, sia pubblici che privati, una suite di servizi che permettono la gestione di dati clinici dei pazienti tramite un repository condiviso tra le diverse cliniche. La piattaforma, oltre ad offrire una serie di servizi su misura, permette agli operatori di poter gestire, tramite un'interfaccia intuitiva, il portale clinico e di poter visionare analisi di dati avanzate che permettono di trarre informazioni rilevanti sotto il punto di vista clinico e statistico.

La nuova soluzione è necessaria in quanto il modulo BFF presenta dei difetti sotto alcuni casi particolari di stress. Considerando che questo modulo si interpone tra gli utenti e il backend si può considerare come l'unico punto di accesso al sistema, e per questo motivo può presentare alcune criticità legate allo smistamento del traffico di rete verso i server.
Più nello specifico, il problema principale riscontrato avviene quando un servizio risulta rallentato e altri utenti ne soffrono nonostante esso non sia richiesto da loro. Questo avviene perché il modulo BFF non fa distinzione tra le varie chiamate e mette in coda le richieste di tutti gli utenti indiscriminatamente, semplicemente chi viene prima viene servito. Questo meccanismo non tiene conto del fatto che alcuni utenti posti più indietro nella coda possano effettivamente essere serviti senza dare alcun fastidio agli utenti che li precedono.

Per far fronte a questa problematica è necessario l'utilizzo di tecnologie innovative come Quarkus, un framework basato su Java molto prestante, che permette la gestione di applicazioni web reactive utilizzando una quantità di memoria molto bassa rispetto ai competitors.
Il lavoro verte, quindi, sul rendere la coda più reattiva per fare in modo che tutti gli utenti che richiedono un determinato servizio, non richiesto da altri in precedenza, vengano serviti.

Questa soluzione permette di rendere il servizio di NPRI molto più efficiente evitando di causare disagi non necessari all'utenza.

Nonostante le sfide sovvenute durante i mesi di progettazione e sviluppo, i risultati ottenuti sono molto promettenti, in quanto si può verificare un notevole miglioramento delle prestazioni nei casi analoghi a quello sopracitato.
Questo lascia ben sperare sugli sviluppi futuri del progetto, in quanto può essere ampliato sotto molti aspetti, soprattutto dal punto di vista dell'efficienza e della sicurezza.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Negli ultimi anni, i servizi digitali hanno vissuto una notevole crescita in diversi settori, tra i quali quello sanitario, che riveste una certa rilevanza per il nostro progetto. La richiesta di soluzioni più scalabili ed efficienti è aumentata considerevolmente, ed è per questo motivo che i *reverse proxy* hanno guadagnato popolarità, offrendo prestazioni, sicurezza e scalabilità migliori rispetto a quelle che si possono trovare in altre soluzioni. Un *reverse proxy* è posizionato tra client e server, agisce come intermediario per gestire richieste e risposte, al fine di migliorare la qualità, la velocità e la sicurezza della comunicazione con i servizi esposti all'utente.
Nel nostro caso, tali servizi risiedono su host diversi, di conseguenza il *reverse proxy* ha anche il compito di smistare le richieste nel modo più efficiente possibile. Per far fronte a questo requisito centrale è stato adottato *Quarkus*, che fornisce dei sistemi di parallelismo molto efficaci di cui si discuterà ampiamente nei prossimi capitoli.

L'obiettivo della Tesi, quindi, è quello di presentare una soluzione basata su *Quarkus*, al fine di migliorare le prestazioni, l'efficienza e la sicurezza del modulo BFF (Backend-For-Frontend) all'interno del sistema NPRI (Nuova Piattaforma Regionale di Integrazione), garantendo così una migliore esperienza per gli operatori che utilizzano la piattaforma. *Quarkus* è un framework basato su Java che offre la possibilità di creare microservizi leggeri e veloci, inoltre la sua infrastruttura basata su Vert.x si allinea perfettamente ai requisiti di un reverse proxy ad alte prestazioni.

Il progetto è sviluppato per conto di Healthy Reply, che si occupa della gestione di NPRI, mettendo a disposizione ricerca, tecnica e infrastrutture al fine di poter garantire il miglior servizio possibile agli enti che ne fanno uso. NPRI è una piattaforma istituita da Regione Lombardia per supportare le istituzioni ed enti sanitari, fornendo loro una serie di soluzioni infrastrutturali per migliorare la produttività e l'efficienza dei servizi offerti ai cittadini.

Il flusso del progetto è suddividibile in tre fasi:

- *Stato dell'arte*: fase di ricerca, nella quale si è stabilito insieme agli interni quale potessero essere le soluzioni più vicine allo stato dell'arte, al fine di poter ottenere il risultato più in linea con i requisiti;
- *Implementazione*: ricerca delle migliori tecniche per poter sfruttare le potenzialità del framework, stesura e ottimizzazione del codice;
- *Testing*: validazione dei risultati ottenuti, confronto con le performance del sistema antecedente.

La tesi è suddivisa nei seguenti capitoli, ognuno dei quali mira ad approfondire nel dettaglio gli argomenti affrontati durante lo sviluppo del progetto:

1. **Fondamenti teorici** : in questo capitolo, viene analizzato lo stato dell'arte riguardo l'architettura sottostante, il funzionamento di un reverse proxy, nonchè di Quarkus, Vert.x e le altre tecnologie utilizzate durante lo sviluppo.
2. **Implementazione e sviluppo** : in questo capitolo, vengono affrontate nel dettaglio le decisioni progettuali, le fasi di sviluppo e vengono spiegate le difficoltà incontrate durante l'implementazione e le relative soluzioni intraprese.
3. **Testing e valutazione delle prestazioni** : in questo capitolo, sono definite le tecniche di testing adottate per poter validare le soluzioni implementate e sono presenti le valutazioni sulle prestazioni in termini di latenza, throughput e scalabilità.
4. **Sviluppi futuri** : in questo capitolo, vengono discussi i possibili sviluppi futuri del progetto.
5. **Conclusioni** : in questo capitolo, saranno riassunti e discussi i risultati, traendo le somme sul lavoro svolto.

Capitolo 2

Fondamenti teorici

All'interno di questo capitolo, avremo l'opportunità di approfondire le nozioni teoriche fondamentali che sono necessarie per ottenere una chiara comprensione del contesto e degli argomenti trattati all'interno della tesi. Attraverso questa sezione introduttiva, verrà fornito un solido fondamento teorico che ci consentirà di esplorare in modo approfondito le tematiche chiave inerenti al progetto, gli argomenti trattati spazieranno dalle reti, in quanto il progetto è strettamente legato alla rete e ai protocolli che la governano, ai linguaggi di programmazione utilizzati, frameworks, sistemi per la gestione di container.

2.1 Reti

In questa sezione verranno approfonditi gli aspetti più importanti per quanto concerne le nozioni teoriche di reti informatiche necessarie ai fini del progetto. Sarà comprensivo dei concetti di base per comprendere il funzionamento delle reti, i protocolli interessati dai nostri fini, le architetture esistenti e il funzionamento dei proxy/reverse proxy.

2.1.1 Concetti di base delle reti informatiche

Per rete informatica si intende una struttura di comunicazione tra diversi nodi, i quali possono essere rappresentati da switch, modem, router o computer/server. I nodi di una rete comunicano tra di loro al fine di condividere risorse, scambiare informazioni e cooperare tra di loro. La necessità primaria di una rete è quella di fare arrivare un pacchetto, che in gergo tecnico rappresenta l'unità d'informazione, da un punto A ad un punto B. Al fine che il pacchetto possa transitare correttamente è necessario che possieda delle meta-informazioni, come ad esempio l'indirizzo di partenza e quello di arrivo, il suo numero di sequenza (al fine di poter essere

concatenato correttamente con gli altri pacchetti del suo flusso) o la dimensione del payload (l'informazione che trasporta). Queste ultime sono fondamentali per far prendere ai router decisioni sull'instradamento del pacchetto.
All'interno di questa sezione verranno approfonditi tutti questi aspetti.

Topologia

Per *topologia* di una rete si intende la struttura fisca o logica di una rete informatica, indica come sono disposti i dispositivi di rete tra di loro e come comunicano.
Ogni topologia ha i suoi pro e i suoi contro, gli aspetti più importanti per la decisione di una topologia rispetto ad un'altra sono: affidabilità, scalabilità, prestazioni e costi

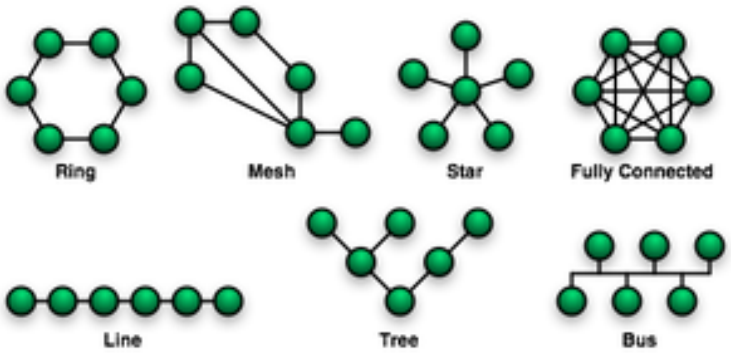

Figura 2.1: Topologie

Le topologie di rete più comuni sono:

- **Topologia a stella**: tutti i dispositivi sono collegati ad un nodo centrale (switch/hub) tramite dei cavi separati, tutte le informazioni passeranno da esso, di conseguenza la comunicazione è "centralizzata".

- **Topologia ad anello**: i dispositivi di rete sono disposti in un anello chiuso, dove ogni dispositivo è collegato direttamente ai due dispositivi adiacenti. I dati vengono trasmessi lungo l'anello da un dispositivo all'altro fino a raggiungere la destinazione.

- **Topologia a bus**: i dispositivi di rete sono collegati a un unico cavo comune chiamato "bus". I dati vengono trasmessi lungo il bus e tutti i dispositivi possono ricevere le informazioni trasmesse.

- **Topologia ad albero**: i dispositivi di rete sono organizzati in una struttura gerarchica ad albero. Ci sono un nodo radice e vari rami che si estendono dai nodi intermedi fino ai nodi finali. I dati vengono trasmessi attraverso i rami dell'albero.
- **Topologia a maglia**: in questo caso ogni dispositivo di rete è collegato direttamente a tutti gli altri dispositivi di rete. Questo crea un'interconnessione completa tra i dispositivi e consente molteplici percorsi per la trasmissione dei dati.

Tipologie di reti

Le tipologie di reti si possono suddividere in macro-categorie in base alla loro estensione e al contesto di utilizzo.
Esse sono classificate in:

- **LAN**: Le reti locali o *LAN, Local Area Network*, sono una tipologia di reti ampiamente utilizzata soprattutto in contesti domestici o aziendali, consentono la condivisione locale di risorse e offrono una velocità molto elevata, dovuto soprattutto alla presenza di un numero limitato di nodi da gestire. Per poter comunicare con l'esterno è necessaria la presenza di un gateway, ossia un dispositivo (o una serie di dispositivi) che permette la corretta gestione di ciò che entra ed esce dalla rete locale.
- **WAN**: Le reti estese o *WAN, Wide Area Network*, coprono aree geografiche più ampie rispetto alle reti locali, sono spesso usate per connettere sedi aziendali geograficamente lontane. Offrono una connessione affidabile su lunghe distanze, consentendo la condivisione di risorse e la comunicazione in contesti dove le LAN risultano poco funzionali.
- **WLAN**: Le reti wireless o *WLAN, Wireless Local Area Network*, sono delle reti locali che permettono la connessione tramite tecnologie Wi-Fi, quindi senza la necessità di cavi fisici. Esse sono principalmente utilizzate in contesti domestici e aziendali, ma a differenza delle reti locali classiche, sono sfruttate anche in contesti pubblici, dal momento che permettono maggiore mobilità e flessibilità.
- **VPN**: Le reti virtuali private o *VPN, Virtual Private Network*, permettono di generare una rete locale *"logica"* all'interno della rete pubblica, tramite la creazione di tunnel crittografati che consentono agli utenti la possibilità di usufruire di risorse remote in modo sicuro, come se fossero all'interno della stessa rete locale. Sono molto usate in contesto aziendale, per garantire ai dipendenti l'accesso alle risorse senza essere fisicamente in sede, ma anche in contesti privati per navigare in modo più sicuro.

- **Internet**: Internet, identificata anche come "rete pubblica", è una vasta rete, composta da una quantità enorme di sottoreti e singoli utenti interconnessi a livello globale, che collega miliardi di dispositivi. È un'infrastruttura di rete che permette la comunicazione e lo scambio di informazioni tra utenti e sistemi distribuiti in tutto il mondo. Internet offre una connettività onnicomprensiva e facilita l'accesso a una vasta gamma di servizi, risorse e informazioni.

Dispositivi di rete

Il motivo primario dell'esistenza delle reti informatiche è quello di favorire la comunicazione tra diversi dispositivi interconnessi.
I dispositivi di rete fondamentali sono:

- **Computer**: I computer rappresentano i nodi principali di una rete informatica, ogni computer può funzionare come client, server o entrambi, a seconda delle risorse che ha a disposizione e del suo ruolo a livello di utilizzo. Essi sono coloro che generano e processano informazioni, sono gli utilizzatori della rete, che sia dal punto di vista di usufruire di servizi/risorse (client) o da quello di fornirne (server).

- **Router**: I router sono i dispositivi responsabili dell'instradamento del traffico in rete, essi sono fondamentali per il funzionamento, ad esempio, di internet. Usufruiscono di articolati algoritmi (come ad esempio DVR, Distance Vector Routing) al fine di rendere l'instradamento più efficiente a seconda dei casi, sono i responsabili delle decisioni prese per fare arrivare il pacchetto da un punto A ad un punto B.

- **Switch**: Gli switch sono dispositivi di rete che consentono di collegare più dispositivi all'interno di una rete locale (LAN) e di instradare i pacchetti di dati tra di essi. Gli switch creano collegamenti diretti tra i dispositivi collegati, migliorando la velocità e l'efficienza della comunicazione all'interno della rete.

- **Modem**: I modem sono dispositivi che consentono la connessione di una rete locale (LAN) a una rete esterna, come Internet. I modem traducono i segnali digitali dei computer in segnali analogici compatibili con la rete esterna e viceversa, spesso questa unità è integrata all'interno dello stesso dispositivo contenente anche switch, router e access point.

- **Access point**: Gli access point sono dispositivi che consentono la connessione senza fili a una rete locale (LAN) o a una rete wireless più ampia, come Internet. Forniscono un punto di accesso per i dispositivi wireless, consentendo loro di connettersi e comunicare all'interno della rete.

- **Server**: I server sono dei computer, solitamente molto prestanti, in grado di fornire dei servizi o risorse per molteplici utilizzi, tra le tipologie più importanti si possono includere i Web Server, i server di posta elettronica, i server che mettono a disposizione degli interi file system o i server di database. Una categoria di server molto importante per il nostro studio sono i DNS, che verranno trattati in maniera approfondita successivamente.

Protocolli

I *protocolli di rete* sono fondamentali per poter permettere la comunicazione tra dispositivi di diversa natura all'interno di un mezzo di comunicazione comune. Esistono numerosi protocolli di rete molto utilizzati e variano in base agli scopi perseguiti. Alcuni esempi comuni includono il protocollo **IP** (Internet Protocol) che gestisce l'instradamento e la consegna dei pacchetti di dati su Internet, il protocollo **TCP** (Transmission Control Protocol) che garantisce una consegna affidabile dei dati attraverso connessioni di rete, e il protocollo **HTTP** (Hypertext Transfer Protocol) utilizzato per il trasferimento di risorse web. Questi protocolli verranno approfonditi di seguito in una sezione dedicata.

Indirizzamento

L'indirizzamento è un concetto fondamentale alla base delle reti informatiche, fa riferimento all'assegnazione di un identificativo univoco ad ogni dispositivo che usufruisce della rete, al fine di poterlo individuare e raggiungere. Nelle reti viene comunemente utilizzato il protocollo IP al fine di sopperire a queste necessità, esso basa le sue fondamenta sugli indirizzi IP, sequenze di "ottetti" assegnati ai vari host in rete che ne permettono l'identificazione.

Routing

Il routing è un processo fondamentale nelle reti informatiche che riguarda la selezione del percorso ottimale per inviare i dati da una sorgente a una destinazione attraverso una rete. Il routing coinvolge la presa di decisioni da parte dei dispositivi di rete chiamati router, che determinano il percorso migliore per instradare i pacchetti di dati sulla base di diversi fattori come la topologia di rete, la congestione, la larghezza di banda e altre metriche di qualità del servizio.

2.1.2 Come funziona Internet: infrastruttura, indirizzamento e routing

È ormai sotto gli occhi di tutti quanto internet sia diventato centrale all'interno della società al giorno d'oggi, ma quali sono i meccanismi che ci permettono di

poter scambiare informazioni con il resto del mondo? E soprattutto, come fa ad essere tutto così veloce? Cercheremo di dare risposta a queste domande all'interno di questa sezione.

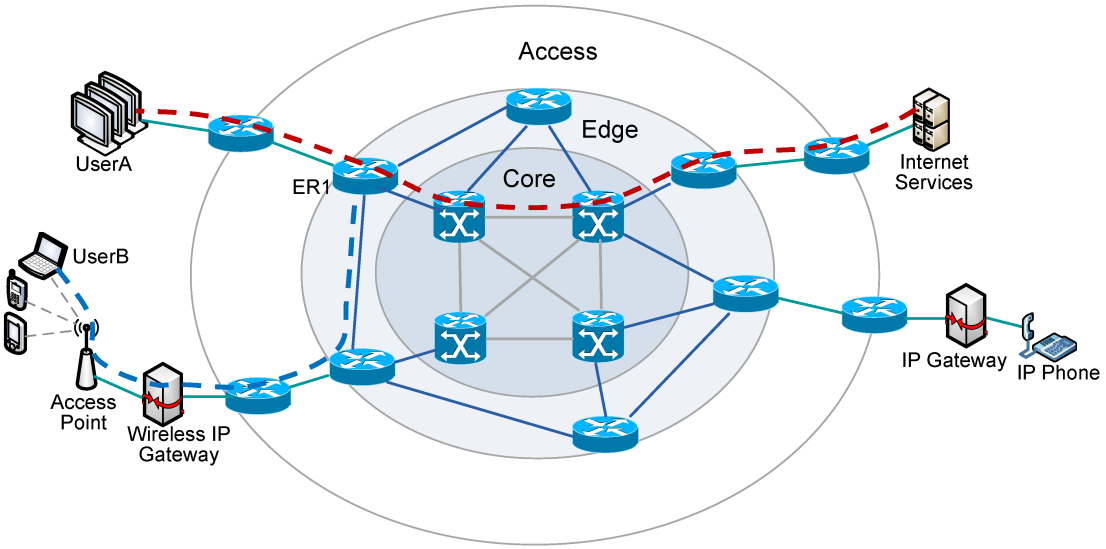

Figura 2.2: Rappresentazione grafica di Internet con Edge, Core ed Endpoints.

Come già accennato in precedenza, internet non è altro che una vastissima rete di reti, alla quale è possibile agganciarsi sia tramite dispositivi singoli, come avviene ccon gli smartphone quando utilizziamo la rete mobile, sia tramite sottoreti, come avviene ad esempio con la rete domestica.
Questi elementi costituiscono quello che viene chiamato **Edge**, ossia quella parte di internet costituita dagli "endpoint" o dispositivi finali, che siano computer, smartphone, tablet o simili. Essi si connettono ad internet tramite il proprio ISP (Internet Service Provider) che fa da ponte per permettere all'utente finale di usufruire dell'intera infrastruttura.
I vari ISP globali contribuiscono tra di loro per formare quello che viene definito **Core** della rete, ossia quella parte di rete *nascosta* nella quale viaggiano miliardi di informazioni al secondo sottoforma di pacchetti, è costituito da router, switch e vari dispositivi di rete predisposti all'instradamento del traffico.
Per poter effettuare gli instradamenti nel modo più efficiente possibile sono stati creati dei protocolli di routing, come OSPF (Open Shortest Path First) e BGP (Border Gateway Protocol), che permettono ai router di scambiarsi informazioni sulla topologia di rete, rendendo possibile la gestione del traffico in modo ottimale, con gli obbiettivi di: rendere il percorso di ogni singolo pacchetto più veloce, creare meno congestione possibile lungo determinati percorsi.

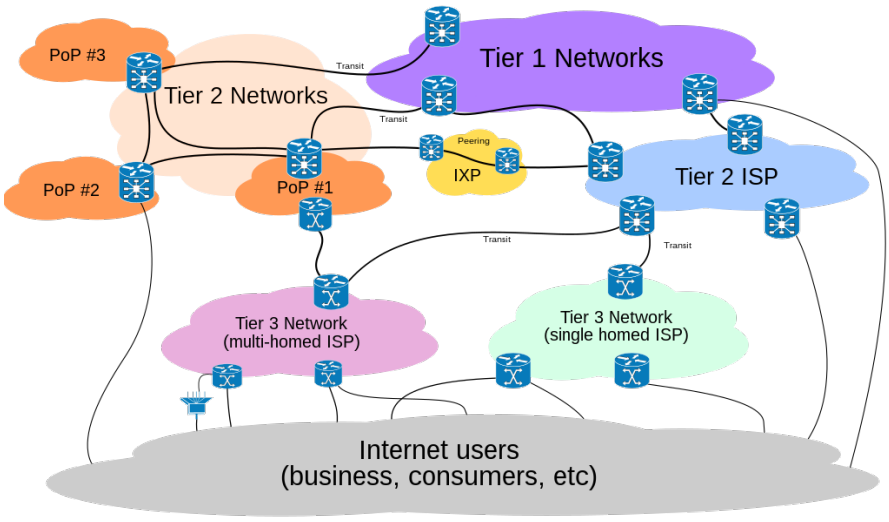

Figura 2.3: Core di Internet

È inoltre fondamentale menzionare come facciano i dispositivi ad "esistere" all'interno della rete. Come già menzionato in precedenza, ad ognuno di essi viene associato un indirizzo IP, il quale fornisce essi un identificativo univoco, il protocollo di riferimento è IPv4, che sta lasciando sempre più piede ad IPv6.
Approfondiremo questi concetti nelle sezioni sottostanti.

Protocolli di routing

I protocolli di routing sono essenziali per la consegna dei pacchetti all'interno di una rete. Essi specificano le regole e gli algoritmi che consentono ai dispositivi di comunicare tra loro e determinare il percorso migliore per inviare i pacchetti alle loro destinazioni previste.
Esistono diversi protocolli di routing, ognuno con le proprie caratteristiche e finalità:

- **RIP (Routing Information Protocol)** è un protocollo di routing a vettore di distanza. Utilizza l'algoritmo Bellman-Ford per determinare i percorsi migliori. RIP ha una metrica basata sul numero di hop tra i router e un limite massimo di 15 hop. È facile da configurare e adatto per reti di piccole dimensioni. Tuttavia, a causa delle sue limitazioni di dimensione delle tabelle di routing e di convergenza lenta, non è adatto per reti complesse o di grandi dimensioni.

- **OSPF (Open Shortest Path First)** è un protocollo di routing a stato di collegamento. Utilizza l'algoritmo di Dijkstra per calcolare i percorsi più brevi.

OSPF supporta la suddivisione della rete in aree, consentendo una maggiore scalabilità. È adatto per reti di grandi dimensioni e complesse, come reti aziendali. OSPF offre anche funzionalità di ridondanza e tolleranza ai guasti.

- **EIGRP (Enhanced Internal Gateway Routeing Protocol)** è un protocollo di routing proprietario sviluppato da Cisco. Combina elementi dei protocolli a vettore di distanza e a stato di collegamento. EIGRP utilizza la metrica di larghezza di banda e ritardo per calcolare i percorsi migliori. È efficiente in termini di larghezza di banda ed è adatto per reti di medie e grandi dimensioni. EIGRP offre anche un rapido tempo di convergenza e supporta funzionalità di ridondanza e bilanciamento del carico.

- **BGP (Border Gateway Protocol)** è un protocollo di routing utilizzato per scambiare informazioni di routing tra sistemi autonomi (AS) su Internet. È un protocollo di routing esterno che consente alle organizzazioni di prendere decisioni di instradamento basate su politiche e criteri specifici. BGP è altamente scalabile e complesso, ed è comunemente utilizzato dai provider di servizi Internet per gestire le rotte tra AS diversi.

L'utilizzo di un protocollo di routing dipende dalle dimensioni e dalla complessità della rete, nonché dai requisiti specifici di prestazioni, affidabilità e sicurezza. La scelta del protocollo di routing giusto è cruciale per garantire un'efficace gestione e un'ottimizzazione delle risorse di rete.

Indirizzamento: IPv4 e IPv6

l protocollo IP (Internet Protocol) è un protocollo di rete fondamentale utilizzato per indirizzare, instradare e trasmettere pacchetti di dati su una rete. È il principale protocollo di livello di rete nel modello di riferimento TCP/IP e svolge un ruolo fondamentale nell'interconnessione di dispositivi e reti in tutto il mondo.

L'IP fornisce un meccanismo per identificare univocamente ogni dispositivo connesso a una rete tramite un indirizzo IP. Gli indirizzi IP sono numeri che identificano in modo univoco un host o un'interfaccia di rete all'interno di una rete. In IPv4, gli indirizzi sono a 32 bit, mentre in IPv6 sono a 128 bit. Gli indirizzi IPv4 sono espressi in notazione decimale separata da punti, mentre gli indirizzi IPv6 sono espressi in notazione esadecimale separata dai due punti.

Il protocollo IP definisce anche il formato dei pacchetti dati utilizzati per la trasmissione dei dati sulla rete. Questi pacchetti, chiamati datagram IP, contengono le informazioni necessarie per instradare i dati dal mittente al destinatario. Ogni datagramma IP include l'indirizzo IP di origine e di destinazione, oltre a

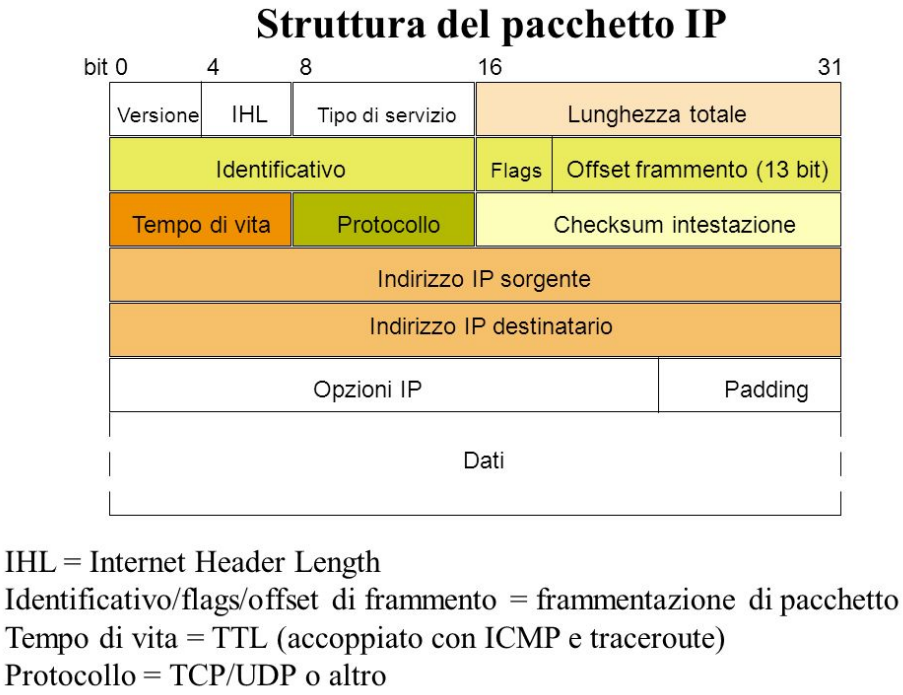

Figura 2.4: Pacchetto IP

informazioni sul protocollo di livello superiore che userà i dati.

Il routing è un aspetto chiave del protocollo IP. Il routing riguarda la scelta del percorso ottimale per instradare i pacchetti attraverso la rete. Ciò coinvolge la selezione dei router intermedi che guideranno i pacchetti lungo il percorso desiderato. I router utilizzano tabelle di routing per prendere decisioni di instradamento basate sull'indirizzo IP di destinazione dei pacchetti.

In sintesi, il protocollo IP è un fondamento essenziale per la comunicazione su Internet. Fornisce l'indirizzamento e l'instradamento dei pacchetti per consentire la trasmissione dei dati tra dispositivi e reti. L'evoluzione da IPv4 a IPv6 è stata necessaria per soddisfare le esigenze di connettività future, offrendo un numero sufficiente di indirizzi IP e funzionalità avanzate.

IPv4

L'indirizzo IPv4 è composto da 32 bit, suddivisi in quattro gruppi da 8 bit ciascuno, separati da punti. Questa notazione, chiamata "dotted decimal notation", semplifica la lettura e la memorizzazione da parte degli esseri umani. Ogni gruppo, o byte, può variare da 0 a 255, poiché 2^8 (ovvero 256) rappresenta il numero massimo di

combinazioni possibili per ciascun byte. Ad esempio, un indirizzo IPv4 potrebbe essere scritto come 172.16.254.1, che corrisponde a una notazione binaria.

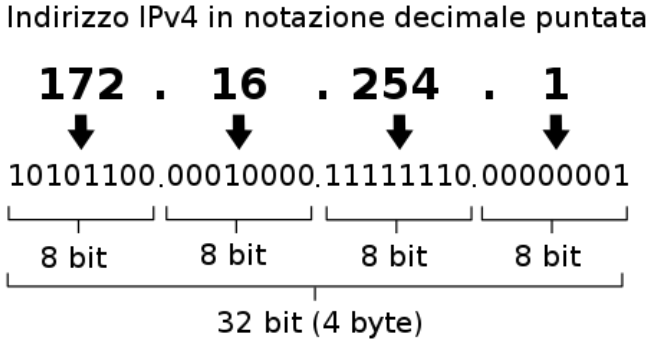

Figura 2.5: IPv4

Tuttavia, l'IPv4 ha una limitata disponibilità di indirizzi, poiché il numero massimo di combinazioni è di circa 4 miliardi. Considerando che molte persone possiedono diversi dispositivi collegati a Internet, come smartphone, computer e smart TV, la richiesta di indirizzi IP univoci supera rapidamente la disponibilità. Pertanto, sono state sviluppate soluzioni come l'utilizzo di "indirizzi privati" all'interno delle reti locali (LAN), in cui i dispositivi sono identificati univocamente ma condividono un indirizzo IP pubblico attraverso il router. Questa soluzione è comunemente adottata dalle imprese.

IPv6

L'indirizzo IPv6 è costituito da 128 bit, rappresentati da 8 gruppi di 4 cifre esadecimali, che corrispondono a 2 byte ciascuno. I gruppi sono separati da due punti (:). Ad esempio, un indirizzo IPv6 completo potrebbe essere 2001:0DB8:0000:0000:0000:0000:0370:7334, ma può essere abbreviato come 2001:0DB8::0370:7334. L'abbreviazione viene utilizzata quando ci sono consecutivi zeri in un gruppo, consentendo una notazione più compatta.
I dispositivi connessi a una rete IPv6 ricevono un indirizzo unicast globale. I primi 48 bit dell'indirizzo identificano la rete a cui il dispositivo è connesso, mentre i successivi 16 bit indicano le sottoreti specifiche a cui il dispositivo appartiene. Gli ultimi 64 bit dell'indirizzo IPv6 sono solitamente derivati dall'indirizzo MAC dell'interfaccia di rete del dispositivo.

L'IPv6 offre un enorme spazio di indirizzamento, con un totale di circa $3.4x10^38$ indirizzi disponibili. Ciò permette di assegnare un numero significativo di indirizzi IP a ogni metro quadrato di superficie terrestre. Nonostante la disponibilità di

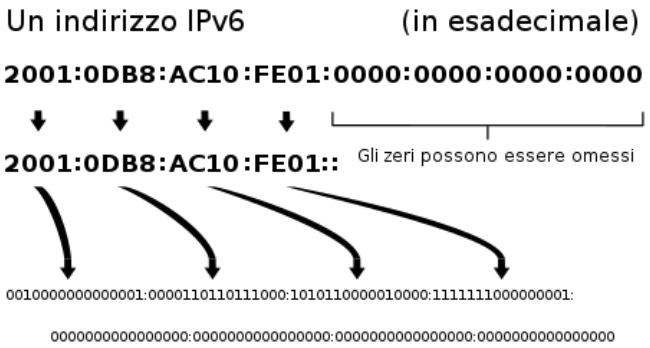

Figura 2.6: IPv6

indirizzi IPv6 sia estremamente ampia, l'adozione completa dell'IPv6 è ancora in corso, con molti sistemi che utilizzano ancora principalmente l'IPv4. Tuttavia, l'IPv6 rappresenta la soluzione a lungo termine per affrontare la crescita delle reti e la sempre maggiore richiesta di indirizzi IP.

2.1.3 Architetture di rete

Le architetture di rete sono modelli o strutture che definiscono come i componenti di una rete informatica sono organizzati e interagiscono tra loro. Una buona architettura di rete fornisce una base solida per progettare, implementare e gestire una rete efficiente, affidabile e sicura.

Le architetture di rete definiscono la disposizione dei componenti di rete, come computer, server, dispositivi di rete e cablaggi, e specificano i protocolli e le tecnologie utilizzate per consentire la comunicazione tra di essi. L'architettura di rete determina anche come i dati vengono instradati, trasmessi e gestiti all'interno della rete.

Alcune delle architetture di rete più comuni sono:

- **Architettura client-server**: Questa architettura si basa sul concetto di divisione del lavoro tra client e server. I client richiedono e utilizzano servizi o risorse fornite dai server. I server forniscono servizi, risorse o dati richiesti dai client. Questo modello consente una distribuzione efficiente delle risorse di rete e facilita la gestione e la manutenzione centralizzata.

- **Architettura peer-to-peer**:In questa architettura, i dispositivi di rete si comportano sia come client che come server, collaborando tra loro per condividere risorse, servizi o dati. Non esiste una gerarchia fissa di ruoli client-server. Ogni dispositivo può agire come peer e partecipare alla condivisione delle risorse con gli altri dispositivi della rete.

- **Architettura a strati**: Questo approccio organizza la rete in una serie di strati o livelli, ciascuno dei quali svolge funzioni specifiche. Ogni livello comunica solo con i livelli adiacenti, fornendo una separazione delle responsabilità e consentendo la sostituzione o l'aggiornamento di un singolo strato senza influire sugli altri. Il modello ISO/OSI e il modello TCP/IP sono esempi di architetture a strati.

- **Architettura cloud**: Questa architettura si basa sull'utilizzo di risorse di rete virtualizzate e distribuite su una piattaforma cloud. Le risorse, come server, storage e servizi, sono fornite on-demand tramite Internet. L'architettura cloud consente una scalabilità elastica, un'allocazione efficiente delle risorse e l'accesso da parte degli utenti da qualsiasi posizione.

Standard ISO/OSI e TCP/IP

Gli standard ISO/OSI (International Organization for Standardization/Open Systems Interconnection) e TCP/IP (Transmission Control Protocol/Internet Protocol) sono due architetture di rete ampiamente utilizzate nel campo delle telecomunicazioni e dell'informatica. Entrambi sono modelli di riferimento che definiscono un insieme di protocolli e standard per consentire la comunicazione tra dispositivi e sistemi informatici in una rete.
Essi orniscono un insieme di regole, protocolli e procedure che definiscono come i dati vengono strutturati, trasmessi, ricevuti e interpretati dai dispositivi di rete. Gli standard ISO/OSI e TCP/IP consentono l'interoperabilità tra dispositivi e sistemi diversi, consentendo alle reti di comunicare tra loro in modo affidabile e scalabile.

Inizieremo esaminando l'architettura ISO/OSI, che è un modello teorico sviluppato dall'International Organization for Standardization. Il modello ISO/OSI si basa su una stratificazione di livelli, ognuno dei quali svolge funzioni specifiche nella comunicazione di rete. Esploreremo i sette livelli del modello ISO/OSI e analizzeremo le loro caratteristiche e le interazioni tra di loro.

Successivamente, ci focalizzeremo sull'architettura TCP/IP, che è l'architettura di rete più ampiamente utilizzata e adottata su Internet. Il modello TCP/IP si basa su una divisione in quattro livelli: livello di rete, livello di trasporto, livello di applicazione e livello di collegamento dati. Approfondiremo le funzioni e le responsabilità di ciascun livello e analizzeremo come avviene la comunicazione tra di essi.

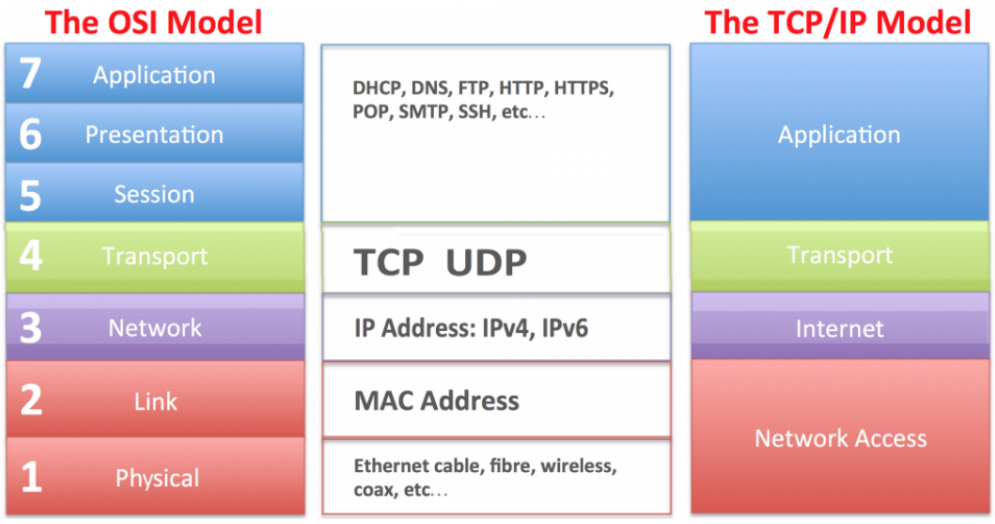

Figura 2.7: OSI vs TCP/IP

ISO/OSI

L'architettura ISO/OSI è stata sviluppata negli anni '70 dall'ISO per fornire una struttura generale per la progettazione delle reti. Il modello ISO/OSI divide il processo di comunicazione in sette livelli o strati, ognuno dei quali svolge funzioni specifiche. Questi strati vanno dal livello fisico, che gestisce i dettagli della trasmissione dei dati attraverso i mezzi fisici, al livello applicativo, che fornisce servizi di comunicazione alle applicazioni. Lo standard ISO/OSI ha avuto un impatto significativo nello sviluppo delle reti, ma la sua adozione pratica è stata limitata rispetto al modello TCP/IP.

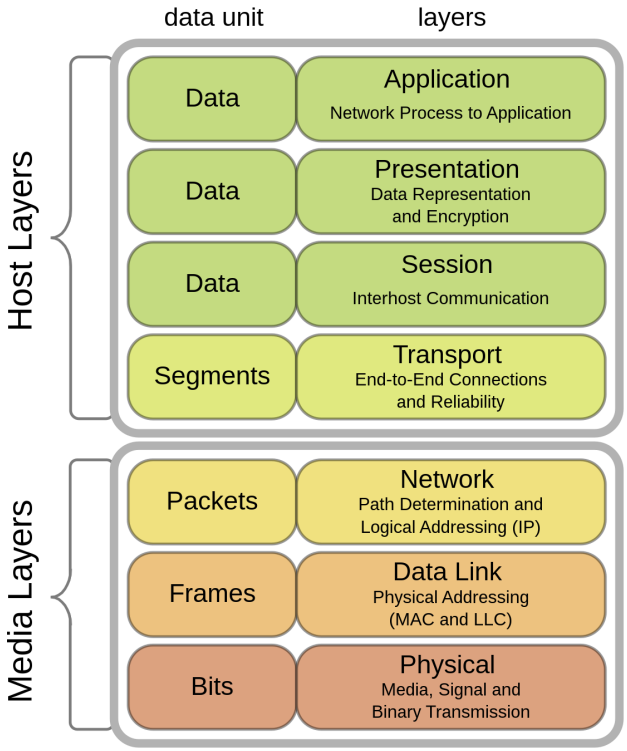

Figura 2.8: Struttura ISO/OSI

I livelli del modello ISO/OSI (ordinati dal più basso al più alto in base al livello di astrazione):

- **Livello fisico**: È il livello più basso del modello e si occupa della trasmissione dei bit grezzi tramite il mezzo fisico di comunicazione, come i cavi di rame o le onde radio. Questo livello definisce le caratteristiche elettriche, meccaniche e funzionali delle interfacce fisiche di rete.

- **Livello di collegamento**: Questo livello si occupa di trasmettere i dati tra due nodi di rete adiacenti, garantendo l'affidabilità e il controllo degli errori. Si occupa anche della suddivisione dei dati in frame e del loro invio sul mezzo di trasmissione.

- **Livello di rete**: Il livello di rete si occupa del routing dei pacchetti attraverso una rete di nodi intermedi. È responsabile della determinazione del percorso ottimale per i pacchetti e dell'instradamento dei dati in modo efficiente e affidabile.

- **Livello di trasposto**: Questo livello si concentra sulla trasmissione dei dati end-to-end tra mittente e destinatario. Fornisce servizi di segmentazione e riunione dei dati, controllo di flusso e controllo dell'affidabilità per garantire che i dati vengano consegnati correttamente e nell'ordine corretto.

- **Livello di sessione**: Il livello di sessione stabilisce, mantiene e termina le sessioni di comunicazione tra due entità di rete. Gestisce anche le attività di sincronizzazione, il controllo del dialogo e il ripristino delle sessioni in caso di interruzioni.

- **Livello di presentazione**: Questo livello si occupa della rappresentazione dei dati, inclusa la crittografia, la compressione, la traduzione dei formati e la gestione della sintassi dei dati. Assicura che i dati siano presentati in un formato comprensibile tra le diverse piattaforme e applicazioni.

- **Livello di applicazione**: Il livello di applicazione è responsabile delle interazioni tra l'utente e le applicazioni di rete. Fornisce servizi di alto livello, come l'accesso alle risorse di rete, l'elaborazione dei dati e le funzionalità specifiche dell'applicazione.

I vari strati del modello ISO/OSI *cooperano* tra loro attraverso il processo di "**incapsulamento dei dati**". Ogni strato aggiunge le proprie informazioni specifiche ai dati prima di passarli allo strato successivo. Questo processo avviene sia durante l'invio dei dati (*trasmissione*) che durante la ricezione dei dati (*ricezione*).

Durante la **trasmissione**, i dati vengono passati dallo strato superiore allo strato inferiore, che aggiunge un'intestazione specifica dello strato per fornire informazioni di controllo aggiuntive. Questo processo di aggiunta dell'intestazione continua attraverso tutti gli strati fino al livello fisico, dove i dati vengono convertiti in segnali fisici per la trasmissione.

Durante la **ricezione**, il processo avviene in modo inverso. I dati vengono ricevuti dal livello fisico e passati al livello di collegamento dati, dove vengono

estratti dall'intestazione dello strato fisico. Questo processo di rimozione dell'intestazione e di passaggio dei dati avanti e indietro tra gli strati continua fino a raggiungere lo strato superiore, dove i dati finali vengono consegnati all'applicazione di destinazione.

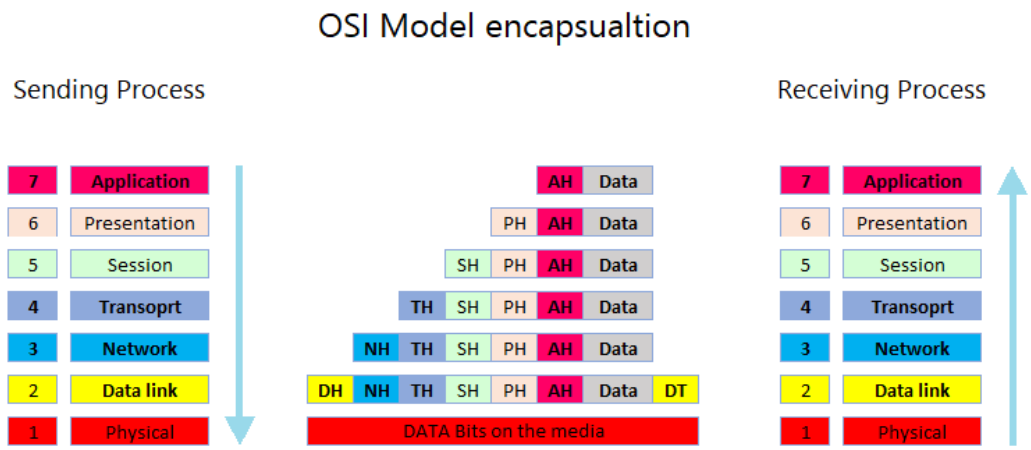

Figura 2.9: Incapsulamento ISO/OSI

La cooperazione tra gli strati avviene attraverso l'invio e la ricezione di protocolli specifici per ciascuno strato. Ogni strato ha il proprio protocollo di comunicazione che definisce le regole e le procedure per la trasmissione e la ricezione dei dati. Gli strati inferiori si occupano di dettagli più tecnici e di basso livello, come la trasmissione dei bit e il controllo degli errori, mentre gli strati superiori si occupano di funzionalità più specifiche dell'applicazione, come l'accesso ai file o la gestione delle sessioni.

Attraverso la cooperazione tra gli strati, il modello ISO/OSI permette una comunicazione efficace e standardizzata tra i dispositivi e i sistemi di rete. Ogni strato svolge un ruolo specifico e contribuisce al processo di comunicazione globale, consentendo una maggiore interoperabilità tra le diverse reti e applicazioni.

TCP/IP

TCP/IP è un insieme di protocolli di rete ampiamente utilizzati per consentire la comunicazione e lo scambio di dati tra dispositivi all'interno di una rete. Prende il nome dai suoi due principali protocolli, il Transmission Control Protocol (TCP) e

l'Internet Protocol (IP).

L'architettura TCP/IP è stata sviluppata prima del modello di riferimento ISO/OSI ed è diventata la base della struttura di Internet. È una pila di protocolli organizzata in quattro livelli che forniscono una connettività affidabile, la consegna dei dati, il routing e altre funzionalità di rete.

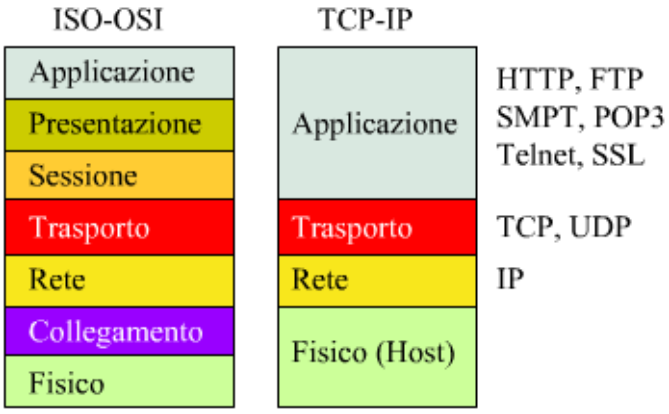

Figura 2.10: Livelli TCP/IP e protocolli

I livelli principali che compongono TCP/IP sono:

- **Livello di collegamento**: Questo livello definisce i protocolli per la comunicazione all'interno di una singola rete locale (LAN) e gestisce l'accesso al mezzo trasmissivo. Uno dei protocolli noti a questo livello è l'Ethernet.

- **Livello di rete**: Il protocollo IP è responsabile dell'instradamento dei pacchetti all'interno di una rete o tra reti diverse. Assegna indirizzi IP unici a ogni dispositivo nella rete e definisce come i pacchetti vengono indirizzati e consegnati correttamente.

- **Livello di trasporto**: Il TCP è uno dei protocolli chiave di TCP/IP. Fornisce una comunicazione affidabile e orientata alla connessione tra applicazioni su dispositivi diversi. Il TCP segmenta i dati in pacchetti, li invia e li assembla correttamente all'altro capo della connessione, garantendo l'affidabilità e il controllo di flusso.

- **Livello di applicazione**: Questo livello comprende una vasta gamma di protocolli che consentono alle applicazioni di comunicare tra loro attraverso la rete. Alcuni esempi di protocolli a questo livello includono HTTP per la comunicazione web, SMTP per l'invio di email e FTP per il trasferimento di file.

Oltre a questi livelli principali, TCP/IP include anche altri protocolli, come l'Address Resolution Protocol (ARP) per la risoluzione degli indirizzi MAC e il Domain Name System (DNS) per la traduzione degli indirizzi IP in nomi di dominio, del quale parleremo approfonditamente in seguito.

2.1.4 Protocolli di trasporto: TCP e UDP

TCP (Transmission Control Protocol) è un protocollo di livello di trasporto nella suite di protocolli TCP/IP. È ampiamente utilizzato per fornire un trasferimento affidabile dei dati attraverso una rete, come Internet.

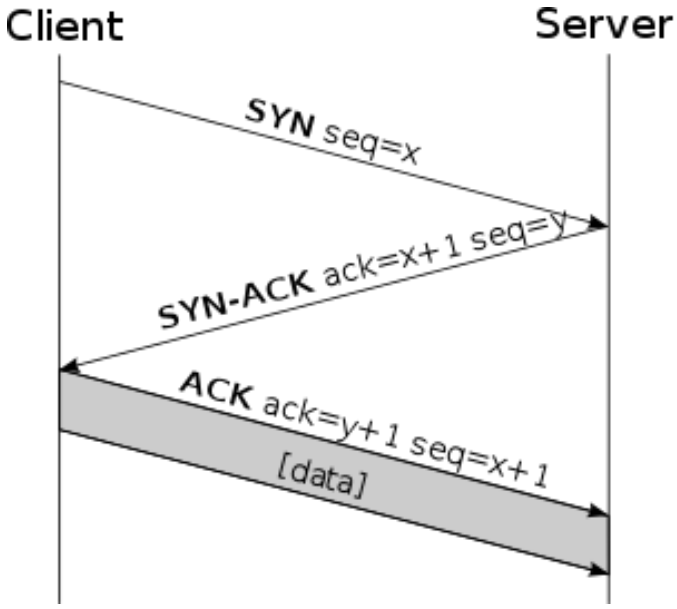

Figura 2.11: Connessione TCP

Il protocollo TCP suddivide i dati in segmenti e li trasferisce tra un mittente e un destinatario.
Offre diverse caratteristiche importanti per garantire una consegna affidabile dei dati:

1. **Connessione orientata**: TCP stabilisce una connessione bidirezionale tra il mittente e il destinatario prima del trasferimento dei dati. Questa connessione viene creata attraverso un processo di handshaking a tre vie, noto come SYN, SYN-ACK, ACK. La connessione garantisce l'affidabilità dei dati e il controllo del flusso.
2. **Affidabilità**: TCP utilizza un meccanismo di rilevamento degli errori e ritrasmissione dei pacchetti persi o danneggiati. Il destinatario conferma la

ricezione dei pacchetti inviando pacchetti di conferma (ACK) al mittente. Se il mittente non riceve la conferma entro un certo intervallo di tempo, ritrasmette i pacchetti mancanti. Questo assicura che i dati siano consegnati correttamente e in ordine.

3. **Controllo del flusso**: TCP regola il flusso dei dati tra il mittente e il destinatario per evitare l'overload del destinatario. Il destinatario indica la sua capacità di accettare i dati attraverso il meccanismo di finestre scorrevoli. In base a queste informazioni, il mittente invia solo la quantità di dati che il destinatario può gestire, evitando la congestione della rete.

4. **Controllo della congestione**: TCP è in grado di rilevare la congestione di rete monitorando il ritardo di consegna dei pacchetti e la presenza di pacchetti persi. Utilizzando algoritmi di controllo della congestione come TCP Reno o TCP Cubic, TCP riduce la velocità di invio dei dati per alleviare la congestione e mantenere la stabilità della rete.

5. **Garanzie dell'ordine**: TCP garantisce che i dati vengano consegnati all'altro estremo in ordine. Utilizza numeri di sequenza per identificare i segmenti e i pacchetti inviati, consentendo al destinatario di riordinare i dati in base al numero di sequenza.

TCP è ampiamente utilizzato per applicazioni che richiedono una consegna affidabile dei dati, come la navigazione web, l'invio di e-mail, il download di file, lo streaming multimediale e molte altre. È uno dei protocolli fondamentali della comunicazione su Internet e svolge un ruolo cruciale nell'assicurare che i dati siano trasferiti in modo affidabile e efficiente attraverso la rete.

UDP (User Datagram Protocol) è un protocollo di livello di trasporto nella suite di protocolli TCP/IP. A differenza di TCP, UDP è un protocollo senza connessione, il che significa che non stabilisce una connessione prima di trasferire i dati.

Ecco alcuni punti chiave relativi a UDP e i suoi pro e contro rispetto a TCP:

- **Senza connessione**: UDP non richiede la creazione di una connessione prima del trasferimento dei dati. I pacchetti UDP, chiamati datagrammi, sono inviati indipendentemente l'uno dall'altro. Ciò rende UDP più leggero e veloce rispetto a TCP.

- **Nessun controllo di flusso**: UDP non fornisce un meccanismo di controllo del flusso come TCP. Questo significa che non ci sono meccanismi automatici per regolare la velocità di trasferimento dei dati in base alla capacità del

destinatario. Pertanto, UDP può inviare dati a una velocità più elevata, ma senza garantire che il destinatario li riceva tutti.

- **Nessuna garanzia di consegna o ordine**: A differenza di TCP, UDP non garantisce la consegna dei dati. I pacchetti UDP possono essere persi, duplicati o consegnati fuori sequenza. Questo può essere accettabile in alcune applicazioni in cui la perdita di alcuni dati non compromette la qualità del servizio.

- **Prestazioni e latenza**: Grazie alla sua natura senza connessione e alla mancanza di meccanismi di controllo del flusso e di rilevamento degli errori, UDP ha prestazioni più elevate rispetto a TCP. È particolarmente adatto per applicazioni in tempo reale come lo streaming multimediale, le videoconferenze o i giochi online, in cui la velocità e la latenza ridotta sono cruciali.

2.1.5 Protocolli applicativi

I protocolli applicativi sono un insieme di regole e convenzioni che definiscono come le applicazioni comunicano tra loro su una rete. Questi protocolli operano nel livello più alto dell'architettura di rete, noto come livello di applicazione, e consentono lo scambio di dati, informazioni e richieste tra le varie applicazioni.

Uno dei protocolli applicativi più comuni è l'HTTP (Hypertext Transfer Protocol), che viene utilizzato per la comunicazione tra i client (come i browser web) e i server web. Grazie all'HTTP, gli utenti possono richiedere pagine web e ottenere le relative risposte dai server web. Inoltre, l'HTTP consente l'invio di dati attraverso moduli web e altre interazioni tra l'utente e il server.

Un altro importante protocollo applicativo è l'SMTP (Simple Mail Transfer Protocol), utilizzato per la trasmissione delle email. Grazie allo SMTP, gli utenti possono inviare email da un client di posta elettronica al server di posta del destinatario. Inoltre, esistono protocolli come il POP (Post Office Protocol) e l'IMAP (Internet Message Access Protocol) per il recupero delle email da un server di posta.

Il FTP (File Transfer Protocol) è un protocollo utilizzato per il trasferimento di file tra un client e un server. Consentendo agli utenti di caricare, scaricare e gestire file su server remoti. Un altro protocollo noto è il DNS (Domain Name System), che gestisce la risoluzione dei nomi di dominio in indirizzi IP, consentendo agli utenti di utilizzare nomi di dominio comprensibili per accedere a risorse su Internet invece di dover inserire gli indirizzi IP numerici.

Infine, l'SSH (Secure Shell) è un protocollo crittografato che consente agli utenti di

stabilire una connessione sicura e crittografata con un server remoto. Viene spesso utilizzato per l'accesso remoto a server o per il trasferimento sicuro di file.

Di seguito verranno approfonditi HTTP e DNS, in quanto particolarmente importanti per il nostro caso di studio.

Protocollo HTTP

HTTP (Hypertext Transfer Protocol) è un protocollo applicativo fondamentale utilizzato per la comunicazione tra i client (come i browser) e i server web. È il protocollo principale che consente agli utenti di richiedere e ottenere risposte da server web, rendendo possibile la navigazione e l'interazione con le pagine web.
Per quanto riguarda il nostro caso di studi esso è fondamentale, in quanto andiamo ad operare direttamente sulle richieste e risposte HTTP che vengono scambiate tra operatori sanitari e il server che fornisce i vari servizi di gestione.

Il suo intento originale era quello di consentire il recupero di risorse ipertestuali, come pagine web contenenti testi, immagini e collegamenti, ma nel corso degli anni è stato esteso per supportare una vasta gamma di contenuti e servizi.

Esso si basa sul modello client-server, in cui il client invia richieste al server e il server risponde con le risorse richieste o le informazioni richieste. Le richieste e le risposte HTTP sono composte da una serie di messaggi strutturati, che includono metodi, intestazioni e corpi di messaggio, per trasportare informazioni essenziali.

Le richieste HTTP possono includere operazioni come ottenere una pagina web, inviare dati di un form, inviare file al server o richiedere un'azione specifica. Le risposte HTTP restituiscono lo stato della richiesta, il contenuto richiesto o altre informazioni di controllo.

È un protocollo state-less, il che significa che ogni richiesta è trattata in modo indipendente e non conserva informazioni sulle richieste precedenti. Pertanto, è necessario utilizzare meccanismi aggiuntivi come i cookie per mantenere lo stato e consentire funzionalità avanzate come l'autenticazione e la gestione delle sessioni.

HTTP utilizza il concetto di URL (Uniform Resource Locator) per identificare le risorse richieste sui server. Ad esempio, l'URL di una pagina web è spesso visualizzato come "http://www.example.com/index.html", in cui "http://" indica il protocollo, "www.example.com" è il nome di dominio del server e "/index.html" è il percorso della risorsa specifica.

L'HTTP è un protocollo testuale e leggibile dagli umani, il che agevola la sua comprensione e il debug delle interazioni tra client e server. Tuttavia, è importante notare che HTTP può essere sicuro o non sicuro, a seconda dell'uso di connessioni criptate. L'HTTPS (HTTP Secure) è una versione sicura di HTTP che utilizza la crittografia SSL/TLS per proteggere le comunicazioni e garantire la riservatezza e l'integrità dei dati scambiati.

Pacchetto HTTP

Un pacchetto HTTP è composto da una serie di informazioni strutturate che vengono scambiate tra un client e un server durante la comunicazione HTTP. Queste informazioni sono organizzate in un formato di testo strutturato chiamato "message format". Un pacchetto HTTP è diviso in due parti principali: l'intestazione (header) e il corpo (body).

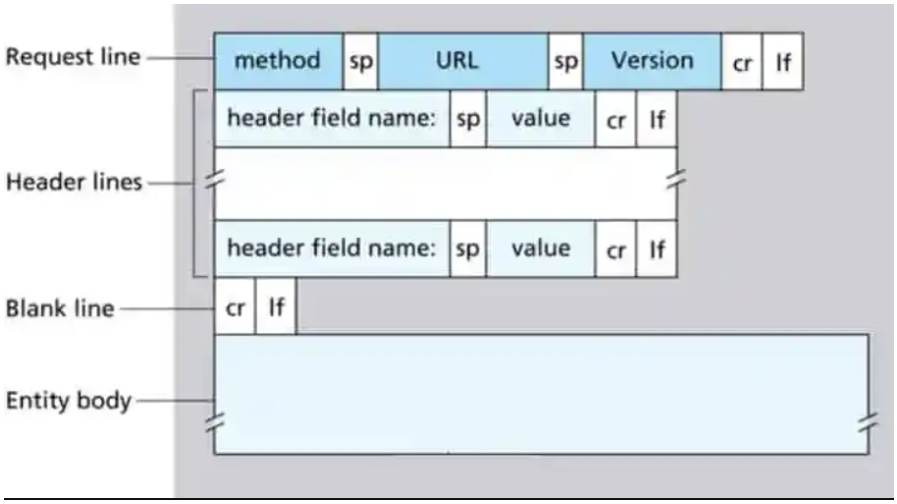

Figura 2.12: Struttura pacchetto HTTP

Request line
All'interno della request line vengono specificati:

- Il **Metodo** indica l'azione richiesta dal client, come GET, POST, PUT, DELETE.
- L'**URL** rappresenta l'indirizzo della risorsa richiesta.
- La **Versione di HTTP** indica la versione del protocollo HTTP utilizzata nella richiesta o nella risposta.

Header
L'intestazione del pacchetto HTTP contiene una serie di linee di testo strutturate che forniscono informazioni aggiuntive sulla richiesta o sulla risposta. L'intestazione può contenere informazioni come il codice di stato HTTP, nelle risposte, oppure i campi dell'intestazione che contengono informazioni riguardo il client, tipo di contenuto, token di autorizzazione, cookie, ecc.
Nel dettaglio:

1. **Content-Type**: Specifica il tipo di media (tipo di contenuto) del corpo del messaggio. Ad esempio, "Content-Type: text/html" indica che il corpo del messaggio contiene codice HTML.

2. **Content-Length**: Indica la lunghezza (in byte) del corpo del messaggio.

3. **Accept**: Indica i tipi di contenuto che il client è in grado di gestire. Viene utilizzato nelle richieste per comunicare le preferenze del client al server. Ad esempio, "Accept: application/json" indica che il client preferisce ricevere una risposta in formato JSON.

4. **User-Agent**: Identifica il software o l'applicazione client che sta effettuando la richiesta. Può essere utilizzato dal server per fornire una risposta personalizzata in base al client.

5. **Authorization**: Contiene le credenziali di autenticazione fornite dal client per accedere a una risorsa protetta. Può includere token di accesso, nomi utente e password, o altre informazioni di autenticazione.

6. **Cache-Control**: Specifica le direttive di controllo della cache per il client o il server. Può indicare se una risposta può essere memorizzata nella cache, per quanto tempo, e se deve essere convalidata con il server prima di essere utilizzata dalla cache.

7. **Location**: Utilizzato nelle risposte di reindirizzamento per indicare al client la nuova posizione a cui reindirizzare la richiesta.

8. **Cookie**: Contiene dati specifici inviati dal server al client e ritrasmessi dal client al server con le successive richieste. I cookie sono utilizzati per mantenere lo stato delle sessioni e per memorizzare informazioni personalizzate per un determinato utente.

Esistono molti altri campi header disponibili che consentono di trasmettere informazioni aggiuntive e specifiche in un messaggio HTTP.

Body

Il corpo del pacchetto HTTP è un'area opzionale che contiene i dati effettivi trasmessi dal client al server o viceversa. Il corpo può contenere testo, immagini, file o altri tipi di dati. Ad esempio, in una richiesta POST, il corpo del messaggio conterrà i dati del form inviati dal client al server.

Risposte HTTP

Nel protocollo HTTP, le risposte vengono inviate dal server al client in risposta a una richiesta. Le risposte HTTP includono una combinazione di una riga di stato (status line), gli header (header) e, opzionalmente, un corpo (body) che contiene i dati o il contenuto richiesto.

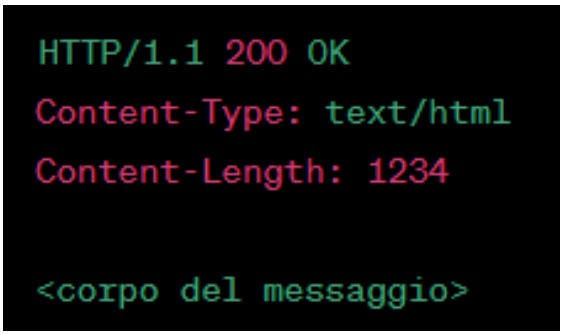

Figura 2.13: Esempio generico di risposta HTTP

Nell'esempio sopra, la riga di stato (status line) indica lo stato della risposta. È composta da tre parti: la versione di HTTP ("HTTP/1.1"), il codice di stato ("200") e una breve descrizione del significato del codice di stato ("OK" in questo caso). Il codice di stato 200 indica che la richiesta è stata completata con successo.

Gli header (header) sono presenti dopo la riga di stato e contengono informazioni aggiuntive sulla risposta. In questo esempio, l'header "Content-Type" indica il tipo di media del corpo del messaggio ("text/html") e l'header "Content-Length" indica la lunghezza del corpo del messaggio ("1234" byte).

Il corpo del messaggio (body) è opzionale e contiene i dati o il contenuto richiesto. Può essere HTML, testo, immagini, file o qualsiasi altro tipo di dati, a seconda della natura della risposta.

Le risposte HTTP possono avere vari codici di stato, che indicano il risultato della richiesta.
Alcuni esempi comuni includono:

- **200 OK**: La richiesta è stata completata con successo.

- **404 Not Found**: La risorsa richiesta non è stata trovata.
- **500 Internal Server Error**: Si è verificato un errore interno lato server durante l'elaborazione della richiesta.

Le risposte HTTP forniscono informazioni dettagliate sullo stato delle richieste e sul contenuto restituito. Il client può utilizzare queste informazioni per prendere decisioni e gestire la risposta in base al codice di stato e agli header ricevuti.

Metodi HTTP

I metodi HTTP definiscono le azioni che possono essere eseguite su una risorsa identificata da un URL. I metodi HTTP specificano il tipo di operazione che il client richiede al server per una determinata risorsa.
Alcuni dei metodi HTTP più comuni sono:

1. **GET**: Il metodo GET viene utilizzato per richiedere una rappresentazione di una risorsa specifica. Il server risponde restituendo il contenuto della risorsa richiesta. Questo metodo è generalmente utilizzato per ottenere dati da un server.
2. **POST**:Il metodo POST viene utilizzato per inviare dati al server per elaborazione o per creare una nuova risorsa. I dati inviati vengono generalmente inclusi nel corpo della richiesta HTTP. Ad esempio, un modulo di invio dati di un modulo HTML utilizza il metodo POST per inviare i dati del modulo al server.
3. **PUT**: Il metodo PUT viene utilizzato per inviare dati al server per creare o sostituire completamente una risorsa specifica con i dati forniti. Se la risorsa non esiste, viene creata. Se la risorsa esiste già, viene sovrascritta con i nuovi dati.
4. **DELETE**: Il metodo DELETE viene utilizzato per richiedere la rimozione di una risorsa specifica dal server. Dopo aver ricevuto una richiesta DELETE, il server elimina la risorsa specificata.
5. **PATCH**: Il metodo PATCH viene utilizzato per applicare modifiche parziali a una risorsa. Viene utilizzato per inviare solo le modifiche necessarie per aggiornare una risorsa, senza dover inviare l'intera rappresentazione della risorsa.
6. **HEAD**: Il metodo HEAD è simile al metodo GET, ma viene utilizzato per richiedere solo le informazioni sull'intestazione di una risorsa senza richiedere il suo contenuto. Questo metodo viene spesso utilizzato per ottenere informazioni sulle risorse senza dover trasferire l'intero corpo della risposta.

Protocollo DNS

Il sistema DNS (Domain Name System) [1] è un servizio di risoluzione dei nomi utilizzato nell'Internet per associare i nomi di dominio agli indirizzi IP corrispondenti. Attraverso il DNS, gli utenti possono utilizzare nomi simbolici, più facili da ricordare, per accedere a risorse su Internet anziché dover memorizzare gli indirizzi IP numerici.
Il DNS è un sistema distribuito che si basa su un'architettura client-server. Le richieste di risoluzione dei nomi vengono inviate dai client DNS (resolver) ai server DNS, che sono responsabili di fornire le informazioni richieste. I server DNS sono organizzati in una gerarchia di livelli, in cui i server di livello superiore (root server) forniscono informazioni sui server dei domini di primo livello (TLD server), che a loro volta forniscono informazioni sui server dei domini di secondo livello e così via.
Il funzionamento del sistema DNS prevede che il resolver, quando riceve una richiesta di risoluzione di un nome di dominio, interroghi prima il server DNS locale. Se il server DNS locale non possiede le informazioni richieste nella sua cache, inoltra la richiesta ai server di livello superiore fino a raggiungere il server DNS autoritativo per il dominio specifico. Il server DNS autoritativo restituisce quindi la risposta al resolver, che a sua volta la invia al client richiedente.
Il sistema DNS è fondamentale per il funzionamento di Internet, poiché consente agli utenti di accedere a risorse tramite nomi di dominio memorizzati in modo più intuitivo rispetto agli indirizzi IP numerici.

Implementazione DNS

L'implementazione del DNS si basa sulle specifiche tecniche descritte nel documento RFC 1035 [2] intitolato "Domain Names - Implementation and Specification". Questo documento fornisce le linee guida e le regole per l'implementazione del sistema DNS, definendo i formati dei messaggi, le strutture dei dati e i protocolli utilizzati.

RFC 1035 stabilisce che il DNS è costituito da una gerarchia di server DNS, suddivisi in zone che rappresentano porzioni del namespace dei nomi di dominio. Ogni zona è gestita da un server DNS autoritativo che mantiene i record delle risorse associati ai nomi di dominio all'interno della zona.

La comunicazione tra i client DNS e i server DNS avviene utilizzando il protocollo UDP (User Datagram Protocol) sulla porta 53. I client inviano query ai server DNS per ottenere informazioni sui nomi di dominio, come gli indirizzi IP associati o i record MX per il servizio di posta. I server DNS rispondono alle query restituendo le informazioni richieste o indicando che il nome di dominio non esiste.

I messaggi DNS seguono un formato ben definito, composto da un'intestazione e una sezione di domande (Question Section), seguite da sezioni di risposte (Answer Section), autorizzazioni (Authority Section) e informazioni aggiuntive (Additional Section). Ogni sezione contiene i record delle risorse che forniscono le informazioni richieste o ulteriori dettagli sul nome di dominio.

Funzionamento DNS

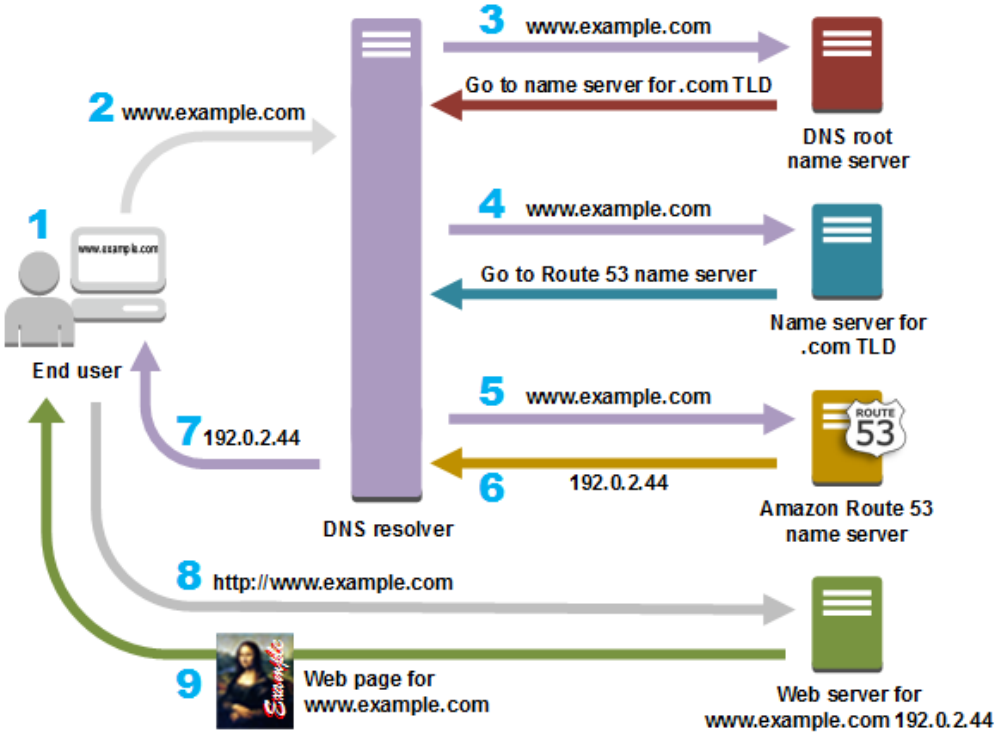

Figura 2.14: Funzionamento del DNS

Come citato all'interno di [3]:

1. Un utente ricerca tramite browser l'url *www.example.com*.
2. La richiesta per *www.example.com* viene instradata a un resolver DNS, gestito da un Internet Service Provider.
3. Il resolver DNS per l'ISP inoltra la richiesta di *www.example.com* verso un server DNS dei nomi radice, chiamato *root name server*.
4. Il resolver DNS per l'ISP inoltra nuovamente la richiesta di www.example.com, questa volta verso uno dei server dei nomi di primo livello per i domini .com.

5. Il server dei nomi per i domini .com risponde alla richiesta con i nomi dei quattro server dei nomi di Amazon Route 53 associati al dominio example.com.

6. Il resolver DNS per l'ISP sceglie un server dei nomi di Amazon Route 53 e inoltra la richiesta per www.example.com a tale server.

7. Il server dei nomi di Amazon Route 53 cerca il record www.example.com nella zona ospitata example.com, ottiene il valore associato, ovvero l'indirizzo IP di un server Web, 192.0.2.44, e lo restituisce al resolver DNS.

8. Il DNS resolver per l'ISP ha così a disposizione l'indirizzo IP richiesto dall'utente. Il resolver restituisce tale valore al browser Web. Il resolver DNS memorizza inoltre nella cache l'indirizzo IP di example.com per una quantità di tempo specificata, così potrà rispondere più rapidamente al successivo tentativo di connessione ad example.com. Per ulteriori informazioni, cerca i dettagli relativi alla funzione Time-to-Live (TTL).

9. Il browser Web invia una richiesta per www.example.com all'indirizzo IP ottenuto dal resolver DNS. Qui si trovano i contenuti, ad esempio un server Web in esecuzione in un'istanza Amazon EC2 o in un bucket Amazon S3 configurato come endpoint di sito Web.

10. Il server Web o la risorsa che si trova all'indirizzo 192.0.2.44 restituisce la pagina Web per www.example.com al browser Web, che visualizzerà così la pagina.

2.1.6 Cosa sono i server proxy: tipologie e caratteristiche

Nell'ambito delle reti informatiche, i proxy server [4] svolgono un ruolo cruciale nella gestione e nell'ottimizzazione delle comunicazioni tra i client e i server. I proxy server agiscono come intermediari tra le richieste dei client e le risposte dei server, offrendo una serie di vantaggi in termini di sicurezza, prestazioni e controllo degli accessi.

Forward proxy

Un forward proxy, chiamato comunemente proxy server, funziona come un punto di contatto tra il client e il server di destinazione, filtrando e instradando il traffico di rete. Quando un client effettua una richiesta di connessione a un server, la richiesta viene inoltrata al proxy server, che a sua volta inoltra la richiesta al server di destinazione. Il server di destinazione risponde quindi al proxy, che inoltra la risposta al client, il server finale quindi non è a conoscenza di chi sta effettuando la richiesta, lui riesce a contattare unicamente il proxy.

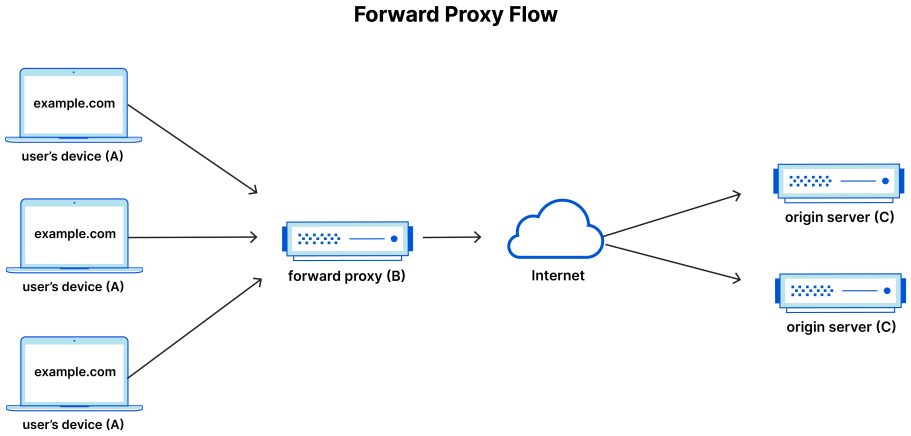

Figura 2.15: Forward proxy [4]

Ci sono diverse ragioni per utilizzare i proxy server. Innanzitutto, i proxy consentono di migliorare le prestazioni di rete mediante l'utilizzo del caching. Quando un proxy riceve una richiesta da parte di un client, può memorizzare in cache la risposta del server. Se un altro client effettua una richiesta identica, il proxy può fornire la risposta memorizzata nella cache, riducendo così il tempo di risposta complessivo e riducendo il carico sui server di destinazione.

Inoltre, i proxy server offrono un controllo granulare sugli accessi alla rete e ai servizi. Attraverso la configurazione del proxy, è possibile definire politiche di accesso che determinano quali client possono accedere a determinati siti web o servizi. Ciò consente di imporre restrizioni di accesso, limitando l'accesso a determinati contenuti o bloccando determinati siti web.

I proxy server possono anche migliorare la sicurezza delle comunicazioni di rete. Possono crittografare il traffico tra client e server utilizzando protocolli come SSL/TLS, garantendo che le informazioni sensibili siano trasmesse in modo sicuro. Inoltre, i proxy possono agire come firewall, proteggendo la rete interna filtrando e bloccando il traffico dannoso o non autorizzato.

Reverse proxy

Un tipo specifico di proxy server è il reverse proxy [5], di fondamentale interesse per il nostro caso di studi, considerando che il core del progetto è la creazione di un reverse proxy, approfondiremo l'utilizzo nel caso specifico nei capitoli successivi. Questo tipo di proxy viene posizionato dal lato del server e gestisce le richieste dei client dirette a un server web. A differenza del forward proxy, in questo caso è l'utente a non sapere nulla riguardo il server da contattare. Il reverse proxy può offrire diversi vantaggi, tra cui il load balancing, che distribuisce il traffico su più server per migliorare le prestazioni e la scalabilità. Inoltre, il reverse proxy può fornire funzionalità di caching e di gestione delle sessioni, migliorando ulteriormente le prestazioni del server.

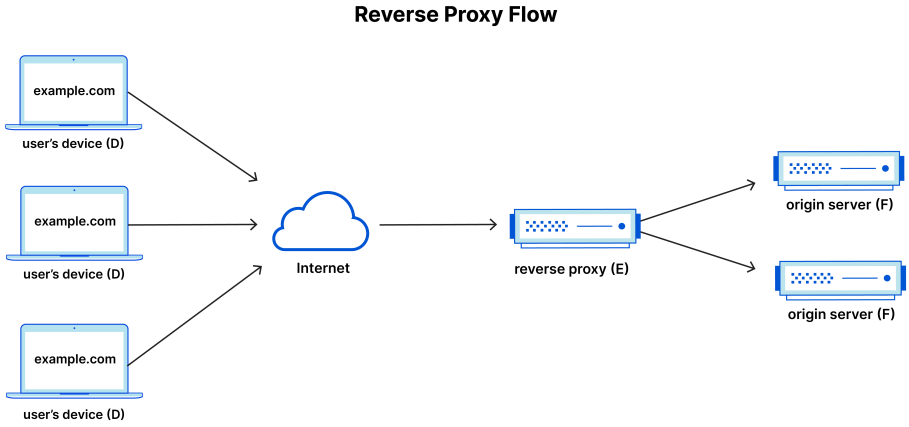

Figura 2.16: Reverse proxy [5]

I reverse proxy possono fornire grossi vantaggi, come:

- **Load balancing**: Un sito web molto popolare che riceve milioni di utenti al giorno potrebbe non essere in grado di gestire tutto il traffico con un singolo server di origine. Invece, il sito può essere distribuito su un pool di diversi server, che gestiscono le richieste per lo stesso sito. In questo caso, un reverse proxy offre una soluzione di bilanciamento del carico che distribuisce il traffico in ingresso in modo equo tra i diversi server per evitare che un singolo server sia sovraccaricato. Nel caso in cui un server dovesse fallire completamente, altri server possono intervenire per gestire il traffico.
- **Protezione da attacchi**: Con un reverse proxy, un sito web o un servizio non deve mai rivelare l'indirizzo IP dei suoi server di origine. Questo rende molto

più difficile per gli aggressori sfruttare un attacco mirato, come un attacco DDoS. Invece, gli aggressori potranno prendere di mira solo il reverse proxy, come il CDN di Cloudflare, che avrà una sicurezza più stretta e più risorse per respingere un attacco informatico.

- **Bilanciamento globale del carico**: Con questa forma di bilanciamento del carico, un sito web può essere distribuito su diversi server in tutto il mondo e il reverse proxy invierà i client al server geograficamente più vicino a loro. Ciò riduce la distanza che le richieste e le risposte devono percorrere, riducendo i tempi di caricamento.
- **Caching**: Un reverse proxy può anche memorizzare nella cache i contenuti, migliorando le prestazioni. Ad esempio, se un utente a Parigi visita un sito web con server web a Los Angeles tramite un reverse proxy, potrebbe effettivamente connettersi a un server proxy inverso locale a Parigi, che comunicherà quindi con un server di origine a Los Angeles. Il server proxy può quindi memorizzare nella cache (o salvare temporaneamente) i dati di risposta. Gli utenti successivi di Parigi che visitano il sito riceveranno quindi la versione memorizzata nella cache dal server proxy inverso parigino, con prestazioni molto più veloci.
- **Crittografia SSL**: Crittografare e decrittografare le comunicazioni SSL (o TLS) per ogni client può richiedere molte risorse computazionali per un server di origine. Un reverse proxy può essere configurato per decrittografare tutte le richieste in ingresso e crittografare tutte le risposte in uscita, liberando risorse preziose sul server di origine.

2.1.7 Cloud computing

Il cloud computing è un modello di distribuzione delle risorse informatiche, come server, storage, database, software e altri servizi, attraverso Internet. Invece di ospitare e gestire tali risorse localmente, le aziende possono usufruire di infrastrutture remote fornite da fornitori di servizi cloud.

Il cloud computing è diventato estremamente popolare per diversi motivi:

1. **Scalabilità**: Il cloud computing consente di scalare rapidamente le risorse informatiche in base alle esigenze. Le aziende possono aumentare o ridurre la capacità in base al carico di lavoro, evitando la necessità di investimenti iniziali costosi per l'infrastruttura fisica.
2. **Flessibilità**: Grazie al cloud, è possibile accedere alle risorse informatiche da qualsiasi luogo e in qualsiasi momento, a condizione di avere una connessione Internet. Questa flessibilità permette alle aziende di adattarsi facilmente a cambiamenti e richieste impreviste.

3. **Riduzione dei costi**: Il cloud computing riduce i costi associati all'acquisto, alla gestione e alla manutenzione di infrastrutture fisiche. Le aziende possono pagare solo per le risorse che utilizzano, evitando costi fissi e ottimizzando l'efficienza operativa.

4. **Aggiornamenti automatici**: I fornitori di servizi cloud si occupano della manutenzione e degli aggiornamenti delle infrastrutture, dei sistemi operativi e dei software associati. Ciò consente alle aziende di beneficiare automaticamente delle ultime funzionalità e patch di sicurezza senza dover gestire tali processi internamente.

5. **Collaborazione semplificata**: Il cloud computing offre strumenti e servizi che favoriscono la collaborazione tra team e dipendenti distribuiti in diverse sedi. Le risorse possono essere facilmente condivise e sincronizzate, migliorando l'efficienza del lavoro di gruppo.

6. **Sicurezza**: I fornitori di servizi cloud spesso investono pesantemente nella sicurezza dei dati. Hanno infrastrutture e misure di sicurezza avanzate per proteggere i dati e mitigare le minacce informatiche. Ciò offre una maggiore tranquillità alle aziende rispetto alla gestione della sicurezza internamente.

7. **Innovazione tecnologica**: Il cloud computing ha reso più accessibili le nuove tecnologie come l'intelligenza artificiale, l'apprendimento automatico e l'analisi dei dati. Le aziende possono sfruttare queste tecnologie senza dover affrontare le sfide di implementazione e gestione interne.

2.2 Sviluppo

In questa sezione, verranno esplorati i principali aspetti legati allo sviluppo del progetto, concentrandoci sul linguaggio Java, concentradoci principalmente sul framework Quarkus e Vert.x, successivamente Kubernetes come piattaforma di orchestrazione dei container e tutto ciò che gravita attorno a Quarkus. L'obiettivo è quello di rendere chiaro il motivo delle scelte intraprese dal punto di vista tecnico.

Architetture software: Monolite vs Microservizi

Architettura monolitica e architettura a microservizi sono due approcci diversi per la progettazione e lo sviluppo di applicazioni software, storicamente l'approccio monolitico è sempre stato quello utilizzato, ma con il passare del tempo, grazie alla ricerca svolta, lo sviluppo di nuove tecnologie e la crescente potenza di calcolo si è passati sempre di più ad un approccio modulare, fino ad arrivare all'approccio moderno, che si può racchiudere nell'architettura a microservizi. Analizziamo di seguito le differenze.

Architettura monolitica

Nell'architettura monolitica, un'applicazione è progettata come un unico e coerente blocco di software, detto monolite, in cui tutte le funzionalità sono sviluppate, testate, implementate e distribuite insieme. L'applicazione monolitica è generalmente implementata come un singolo processo o un'applicazione su un server e include tutte le funzioni e i componenti, come l'interfaccia utente, il business logic e il database, all'interno dello stesso codice sorgente.

Vantaggi:

- **Semplicità**: Poiché l'applicazione è sviluppata come un singolo blocco di software, la sua progettazione, sviluppo e test possono essere più semplici rispetto a un'architettura a microservizi.
- **Performance**: Poiché l'applicazione è un'unica entità, le chiamate tra i suoi componenti sono generalmente molto efficienti e veloci.
- **Facilità di deployment**: L'architettura monolitica semplifica il processo di deployment poiché richiede solo l'installazione di un'applicazione unica.

Svantaggi:

- **Scalabilità limitata**: L'intera applicazione deve essere scalata orizzontalmente, anche se solo alcune parti richiedono maggiori risorse. Questo può comportare un utilizzo inefficiente delle risorse di sistema.

- **Dipendenza dei componenti**: L'aggiornamento o il bugfix di un singolo componente richiede il deployment dell'intera applicazione.
- **Difficoltà di sviluppo e manutenzione a lungo termine**: L'architettura monolitica può diventare complessa e difficile da mantenere quando l'applicazione cresce in dimensioni e complessità.

Architettura a microservizi

Nell'architettura a microservizi, un'applicazione viene scomposta in piccoli servizi autonomi che comunicano tra loro tramite API ben definite. Ogni microservizio è responsabile di una funzionalità specifica e può essere sviluppato, testato e distribuito indipendentemente dagli altri.

Vantaggi:

- **Scalabilità flessibile**: Ogni microservizio può essere scalato individualmente in base alle esigenze di carico di lavoro, ottimizzando l'utilizzo delle risorse di sistema.
- **Modularità e flessibilità**: I microservizi consentono di aggiornare, modificare e sostituire singoli componenti senza influire sul resto dell'applicazione. Ciò favorisce una maggiore agilità nello sviluppo e nella manutenzione.
- **Isolamento dei fallimenti**: Un guasto in un microservizio non influisce sugli altri, consentendo di garantire una maggiore resilienza dell'applicazione.
- **Utilizzo di tecnologie diverse**: Ogni microservizio può essere implementato utilizzando tecnologie e linguaggi diversi in base alle esigenze specifiche, permettendo di sfruttare al meglio le caratteristiche di ciascuna tecnologia.

Svantaggi:

- **Complessità operativa**: La gestione di un'architettura a microservizi può richiedere maggiori competenze operative, come la gestione del deployment, il monitoraggio e la gestione delle comunicazioni tra i diversi servizi.
- **Overhead delle comunicazioni**: Le chiamate tra i microservizi possono comportare un certo overhead di comunicazione rispetto all'integrazione di componenti all'interno di un'applicazione monolitica.
- **Dipendenza delle infrastrutture**: L'architettura a microservizi richiede un'infrastruttura solida per garantire il corretto funzionamento e la comunicazione tra i diversi servizi.

2.2.1 Java

Il linguaggio utilizzato per lo sviluppo del progetto è Java, è stata intrapresa questa decisione perché il framework da utilizzare è nativo Java.

Java è un linguaggio di programmazione ad alto livello orientato agli oggetti. È stato sviluppato da Sun Microsystems (ora di proprietà di Oracle Corporation) negli anni '90 e si è rapidamente diffuso diventando uno dei linguaggi più utilizzati al mondo.

Ecco alcune tra le caratteristiche principali:

1. **Portabilità**: Una delle caratteristiche distintive di Java è la sua portabilità. I programmi scritti in Java possono essere eseguiti su diverse piattaforme senza la necessità di modifiche significative. Ciò è possibile grazie alla Java Virtual Machine (JVM), che interpreta il codice Java in un formato indipendente dalla piattaforma.

2. **Sicurezza**: Java ha un'architettura di sicurezza solida che protegge i programmi da potenziali minacce. La JVM implementa un sistema di sandboxing che isola l'esecuzione del codice Java e controlla l'accesso alle risorse del sistema.

3. **Orientamento agli oggetti**: Java è un linguaggio completamente orientato agli oggetti, il che significa che tutto è un oggetto, incluso il codice stesso. Supporta i principi dell'ereditarietà, dell'incapsulamento, del polimorfismo e dell'astrazione, consentendo agli sviluppatori di creare strutture di dati complesse e organizzare il codice in modo modulare e riutilizzabile.

4. **Ampia libreria standard**: Java offre una vasta libreria standard (Java Standard Edition o Java SE) che fornisce un insieme completo di classi e metodi predefiniti per svolgere una varietà di attività, come gestione delle stringhe, input/output, networking, accesso ai database, grafica e altro ancora. Questo semplifica lo sviluppo di applicazioni, poiché molte funzionalità comuni sono già implementate e pronte all'uso.

5. **Community ed ecosistema robusti**: Java ha una vasta e attiva community di sviluppatori, che fornisce supporto, risorse e librerie open source per ampliare le funzionalità del linguaggio. Inoltre, esiste un vasto ecosistema di strumenti e framework di sviluppo che semplificano la creazione di applicazioni Java complesse e scalabili, come Spring, Hibernate, Maven e molti altri.

fonti da citare: "The Java Language Environment" - Oracle: https://docs.oracle.com/javase/spe

2.2.2 Ambienti di esecuzione: GraalVM e HotSpot

Partiamo inanzitutto dal capire cos'è un ambiente di esecuzione.
Per ambiente di esecuzione si intende quell'insieme di software e risorse che forniscono il contesto in cui un programma può essere eseguito. Esso fornisce le strutture necessarie per poter interpretare ed eseguire il codice, la gestione delle risorse e l'interazione con il sistema operativo, nonchè le librerie di supporto.

Un ambiente di esecuzione include diversi componenti:

- **Runtime**: Il runtime è il nucleo dell'ambiente di esecuzione e gestisce l'esecuzione del programma. Esso comprende i componenti necessari per caricare, interpretare o compilare il codice sorgente e l'esecuzione delle istruzioni del programma.
- **Compilatore**: Il compilatore traduce il codice sorgente in un formato eseguibile dal computer. Può essere un compilatore tradizionale che traduce il codice in linguaggio macchina o un compilatore just-in-time (JIT) che traduce il codice durante l'esecuzione.
- **Gestione della memoria**: L'ambiente di esecuzione gestisce la memoria necessaria per l'esecuzione del programma. Questo include l'allocazione e la deallocazione della memoria, la gestione del garbage collection (nel caso di linguaggi con garbage collection automatica) e l'ottimizzazione dell'utilizzo della memoria.
- **Gestione delle risorse**: L'ambiente di esecuzione gestisce le risorse di sistema necessarie al programma, come file, socket di rete, accesso ai dispositivi di input/output, connessioni di database, ecc. Fornisce meccanismi per l'apertura, la lettura/scrittura e la chiusura delle risorse, mantenendo un controllo appropriato sulla loro disponibilità e utilizzo.
- **Supporto per le librerie**: L'ambiente di esecuzione può fornire librerie di supporto che contengono funzioni comuni utilizzate dai programmi, come funzioni matematiche, operazioni di input/output, manipolazione delle stringhe, operazioni di rete, criptografia e molto altro ancora.

L'ambiente di esecuzione svolge un ruolo fondamentale nell'esecuzione di un qualsiasi programma, orchestrando la gestione delle risorse e dei servizi forniti dal sistema operativo, al fine di garantire un funzionamento corretto dell'applicazione in esecuzione. Il suo obiettivo è quello di garantire un'astrazione dell'hardware sottostante e, di conseguenza, semplificare lo sviluppo per il programmatore, che non dovrà preoccuparsi dei dettagli di basso livello, ma bensì solo della logica del programma.

Nel nostro caso di studi è importante sottolineare l'utilizzo di due ambienti di esecuzione: GraalVM e HotSpot. Questi due ambienti vengono utilizzati da Quarkus, per questo motivo è necessario un piccolo approfondimento per entrambi.

GraalVM

GraalVM è un ambiente di runtime universale sviluppato da Oracle Labs. È progettato per fornire un'infrastruttura di esecuzione ad alte prestazioni per una vasta gamma di linguaggi di programmazione, tra cui Java, JavaScript, Python, Ruby, R, C++, C, Scala e molti altri.

Le caratteristiche principali di GraalVM includono:

1. **Compilazione a stato dell'arte:** GraalVM include un compilatore just-in-time (JIT) di nuova generazione chiamato Graal. Questo compilatore utilizza tecniche di compilazione avanzate per generare codice altamente ottimizzato, migliorando le prestazioni delle applicazioni eseguite su GraalVM.
2. **Esecuzione nativa:** GraalVM consente di creare eseguibili nativi a partire da programmi scritti in Java o altri linguaggi supportati. Questo significa che puoi compilare il tuo codice in un eseguibile che può essere eseguito direttamente sulla macchina senza richiedere un ambiente di runtime esterno, migliorando le prestazioni e riducendo i tempi di avvio.
3. **Interoperabilità tra linguaggi:** GraalVM offre un'interoperabilità fluida tra linguaggi di programmazione. Puoi chiamare codice scritto in un linguaggio da un programma scritto in un altro linguaggio senza dover eseguire operazioni di traduzione o chiamate esterne complesse. Ciò semplifica l'integrazione di componenti scritti in linguaggi diversi all'interno di un'applicazione.
4. **Supporto per la virtualizzazione:** GraalVM supporta la virtualizzazione, consentendo di eseguire diverse istanze di ambienti di runtime isolati su una singola macchina. Questo offre maggiore flessibilità e separazione tra le diverse applicazioni o servizi eseguiti sulla stessa macchina.
5. **Compatibilità con l'ecosistema Java:** GraalVM è compatibile con l'ecosistema Java e supporta le librerie e i framework Java esistenti. Puoi utilizzare le librerie Java standard e beneficiare di strumenti come Maven e Gradle per la gestione delle dipendenze.

GraalVM ha l'obiettivo di fornire un ambiente di esecuzione potente e flessibile per diverse applicazionei e linguaggi. Le sue caratteristiche lo rendono una scelta interessante per progetti che richiedono un'elevata efficienza e un'interoperabilità tra linguaggi.

HotSpot

HotSpot è un ambiente di runtime Java sviluppato da Oracle Corporation. È il motore di esecuzione predefinito utilizzato dalla piattaforma Java, noto per la sua scalabilità, affidabilità e prestazioni.

HotSpot comprende diverse componenti chiave, tra cui il compilatore JIT, il garbage collector e il gestore della memoria. Il compilatore JIT traduce il codice Java in codice macchina a runtime, migliorando le prestazioni dell'applicazione. Il garbage collector si occupa della gestione automatica della memoria, liberando la memoria non utilizzata dagli oggetti non più referenziati. Il gestore della memoria si occupa dell'allocazione e della deallocazione della memoria necessaria per l'esecuzione dell'applicazione.

Una delle principali differenze tra HotSpot e GraalVM è il compilatore JIT. HotSpot utilizza un compilatore JIT tradizionale, noto come C2, che è stato ottimizzato nel corso degli anni per migliorare le prestazioni delle applicazioni Java. GraalVM, d'altra parte, utilizza il compilatore JIT Graal, che è stato progettato per essere altamente performante e supportare diversi linguaggi di programmazione. GraalVM offre quindi un'opzione alternativa al compilatore JIT di HotSpot, che può fornire prestazioni migliori in determinati scenari o per specifici linguaggi.

Un'altra differenza significativa è che GraalVM supporta l'esecuzione nativa, consentendo di creare eseguibili nativi a partire da programmi scritti in diversi linguaggi supportati. Questo significa che l'applicazione può essere eseguita direttamente sulla macchina senza la necessità di un ambiente di runtime esterno, migliorando le prestazioni e riducendo i tempi di avvio.

2.2.3 Kubernetes

[6] Oggi, il concetto di big data non si limita più semplicemente al concetto di avere a disposizione una grande quantità di dataset, la prospettiva è cambiata e l'enfasi si è spostata su opzioni di archiviazione flessibili. Questo cambiamento è stato guidato dall'evoluzione della tecnologia e delle esigenze aziendali.

In passato, l'attenzione era focalizzata sulla gestione di grandi batch di dati, evitando spostamenti di informazioni attraverso la rete. Hadoop, ad esempio, è stato progettato con questa premessa, concentrandosi sulla co-locazione dei dati per un'elaborazione efficiente. Tuttavia, le condizioni sono cambiate nel tempo.

Oggi, invece, la tendenza è verso l'elaborazione dei dati in tempo reale, utilizzando un approccio più orientato allo streaming dei dati. Questo significa che i dati

vengono trasmessi in tempo reale anziché elaborati in lotti. Inoltre, i problemi di latenza di rete sono stati ridotti, con numerosi fornitori di servizi cloud disponibili e la possibilità di implementare cloud privati ibridi.

In questo contesto, i container e **Kubernetes** hanno assunto un ruolo importante. I container sono pacchetti che contengono solo le librerie necessarie per eseguire un'applicazione, consentendo un deployment più rapido e una maggiore portabilità. Kubernetes, d'altra parte, è un sistema di orchestrazione che gestisce e scala i container, garantendo che abbiano le risorse appropriate e controllando il loro ciclo di vita.

L'adozione di container e Kubernetes ha permesso di affrontare le sfide legate alla gestione dei dati e all'elaborazione in tempo reale. Le applicazioni possono essere suddivise in componenti più piccoli, chiamati microservizi, che possono essere facilmente gestiti all'interno dei container. Ciò offre maggiore flessibilità, scalabilità e agilità nello sviluppo e nella gestione delle applicazioni basate sui dati.

In sintesi, la tecnologia dei container e l'uso di Kubernetes hanno contribuito a trasformare il concetto di big data, passando dall'elaborazione batch alla gestione dei dati in tempo reale. Questa nuova approccio offre maggiore flessibilità e scalabilità, consentendo alle aziende di sfruttare appieno il potenziale dei loro dati in modo efficiente.

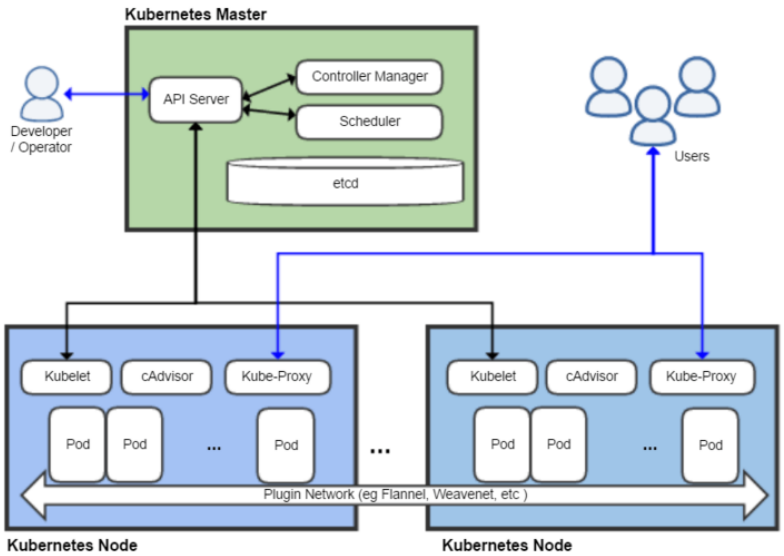

Figura 2.17: Diagramma Kubernetes

I nodi **master** di Kubernetes costituiscono fondamentalmente il cervello del cluster. Sono responsabili della gestione dell'intero cluster, dell'esposizione delle API e della pianificazione degli deployment. I nodi di Kubernetes (lato inferiore della figura precedente) contengono i servizi necessari per eseguire le applicazioni in componenti chiamati Pod. Ogni nodo master contiene i seguenti componenti:

- **API Server**: sincronizza e convalida le informazioni in esecuzione nei Pod e nei servizi.
- **etcd**: fornisce un archivio coerente e altamente disponibile per i dati del cluster. Si può pensare a *etcd* come alla memoria condivisa del cervello.
- **Controller Manager server**: controlla le modifiche nel servizio etcd e utilizza la sua API per applicare lo stato desiderato.
- **HAProxy**: può essere aggiunto quando si configurano i nodi master ad alta disponibilità per bilanciare il carico tra diversi endpoint master.

I **nodi** di Kubernetes (chiamati semplicemente nodi) possono essere considerati i cavalli di battaglia di un cluster di Kubernetes. Ogni nodo espone un insieme di risorse (come calcolo, rete e archiviazione) alle tue applicazioni. Il nodo include anche componenti aggiuntive per la scoperta dei servizi, il monitoraggio, la registrazione e i plugin opzionali. Per quanto riguarda l'infrastruttura, è possibile eseguire un nodo come macchina virtuale (VM) nell'ambiente cloud o su server bare-metal nel data center.

Ogni nodo contiene i seguenti componenti:

- **Pod**: permette di raggruppare logicamente i container e le parti delle nostre applicazioni insieme. Un Pod funge da confine logico per tali container con risorse e contesti condivisi. I Pod possono essere scalati in fase di esecuzione creando set di repliche, garantendo così che venga sempre eseguito il numero richiesto di Pod.
- **Kubelet**: è un agente che viene eseguito su ogni nodo nel cluster di Kubernetes e si assicura che i container siano in esecuzione in un Pod.
- **Kube-Proxy**: mantiene le regole di rete sui nodi per consentire la comunicazione di rete tra i Pod.
- **Container Runtime**: è il software responsabile dell'esecuzione dei container. Kubernetes supporta più runtime di container (come Docker, containerd, CRI-O, e altri).

2.2.4 Quarkus

Quarkus [7] è un framework Java, sviluppato da Red Hat in collaborazione con la comunità open-source, annunciato nel 2019.
È stato progettato specificamente per l'era dei cloud, dei container e di Kubernetes. È stato creato per consentire agli sviluppatori Java di creare applicazioni per un ambiente moderno e nativo per il cloud. Quarkus è basato su GraalVM e HotSpot ed è stato sviluppato utilizzando le migliori librerie e standard Java disponibili.

Figura 2.18: Quarkus logo [7]

L'obiettivo principale di Quarkus è quello di rendere Java la piattaforma leader in ambienti Kubernetes e serverless, offrendo agli sviluppatori un framework che permetta di affrontare una vasta gamma di architetture di applicazioni distribuite. Prima dell'avvento di Quarkus, gli stack Java tradizionali erano progettati per applicazioni monolitiche con tempi di avvio lunghi e requisiti di memoria elevati. Tuttavia, con l'evoluzione del cloud e dei container, c'è stata la necessità di adattare i framework Java a queste nuove esigenze.

Quarkus è stato creato per rispondere a questa esigenza. Grazie alla sua integrazione con GraalVM, è in grado di fornire un avvio rapido delle applicazioni e un utilizzo efficiente delle risorse. Per questo motivo sta prendendo sempre più piede nello sviluppo di microservizi. Inoltre, il framework sfrutta le migliori librerie e standard Java disponibili, consentendo agli sviluppatori di utilizzare le tecnologie già conosciute.

Essendo Kubernetes-native, Quarkus si adatta perfettamente all'ambiente Kubernetes, consentendo agli sviluppatori di sfruttare appieno le funzionalità di orchestrazione e scalabilità offerte da Kubernetes. Inoltre, il framework è anche compatibile con gli ambienti serverless, consentendo di sviluppare applicazioni senza doversi preoccupare dei dettagli dell'infrastruttura sottostante.

Grazie a Quarkus è possibile sviluppare applicazioni Java reattive e scalabili per ambienti cloud-native. Il framework offre una vasta gamma di funzionalità, strumenti e librerie che semplificano lo sviluppo di applicazioni distribuite, consentendo agli sviluppatori di concentrarsi sulla logica anziché sulle complessità dell'infrastruttura.

Caratteristiche del framework

Quarkus è stato pensato per lo sviluppo in ambiente cloud, è ottimo per lo sviluppo di microservizi, in quanto possiede delle caratteristiche che lo rendono particolarmente efficiente in quest'ottica.
Vediamo di seguito riportate quelle principali:

1. **Avvio rapido:** Quarkus offre un avvio rapido, consentendo agli sviluppatori di avviare e scalare rapidamente le proprie applicazioni. Questo è possibile grazie all'uso di GraalVM che consente la compilazione ahead-of-time (AOT) per ridurre i tempi di avvio e il consumo di memoria.

2. **Consumo di risorse ridotto:** Quarkus è progettato per avere un consumo di memoria ridotto e una minore dimensione dell'immagine rispetto alle tradizionali applicazioni Java. Ciò consente un utilizzo efficiente delle risorse e facilita il deployment su ambienti cloud e containerizzati.

3. **Supporto per lo sviluppo reattivo:** Quarkus integra nativamente il supporto per la programmazione reattiva, consentendo agli sviluppatori di creare applicazioni reattive e ad alte prestazioni. Questo è reso possibile grazie all'utilizzo di framework come Reactive Streams e **Eclipse Vert.x**.

4. **Integrazione senza soluzione di continuità:** Quarkus semplifica l'integrazione con altri framework e tecnologie. Supporta nativamente molti framework popolari come Hibernate ORM, Apache Kafka, RESTEasy, Eclipse MicroProfile e molti altri. Ciò consente agli sviluppatori di utilizzare le loro librerie preferite senza sforzi aggiuntivi di integrazione.

5. **DevOps-friendly:** Quarkus favorisce una mentalità DevOps, consentendo agli sviluppatori di creare, testare e distribuire le proprie applicazioni in modo rapido e continuo. Supporta strumenti di build moderni come Maven

e Gradle, e offre funzionalità di live coding che consentono agli sviluppatori di visualizzare immediatamente le modifiche apportate al codice durante lo sviluppo.

6. **Ecosistema attivo:** Quarkus ha una comunità di sviluppatori attiva e un ecosistema di estensioni che fornisce una vasta gamma di funzionalità aggiuntive pronte per l'uso. Questo permette agli sviluppatori di estendere facilmente le proprie applicazioni con funzionalità come la sicurezza, la gestione dei dati, la connettività con i database e molto altro ancora.

2.2.5 Vert.x e programmazione asincrona

Eclipse Vert.x è un framework per la creazione di applicazioni reattive sulla Java Virtual Machine. Le applicazioni reattive sono scalabili all'aumentare dei carichi di lavoro e resilienti in caso di malfunzionamenti. Un'applicazione reattiva viene detta tale poiché mantiene la latenza sotto controllo utilizzando in modo efficiente le risorse di sistema e proteggendosi dagli errori.
Vert.x è supportato da un ampio ecosistema di moduli reattivi che offrono tutto ciò di cui si necessita per scrivere servizi moderni: uno stack web completo, driver di database reattivi, messaggistica, flussi di eventi, clustering, metriche, tracciamento distribuito e altro ancora.

L'approccio classico alla concorrenza è quello di utilizzare i **thread**. Più thread possono vivere all'interno di un singolo processo, eseguire lavoro concorrente e condividere lo stesso spazio di memoria.

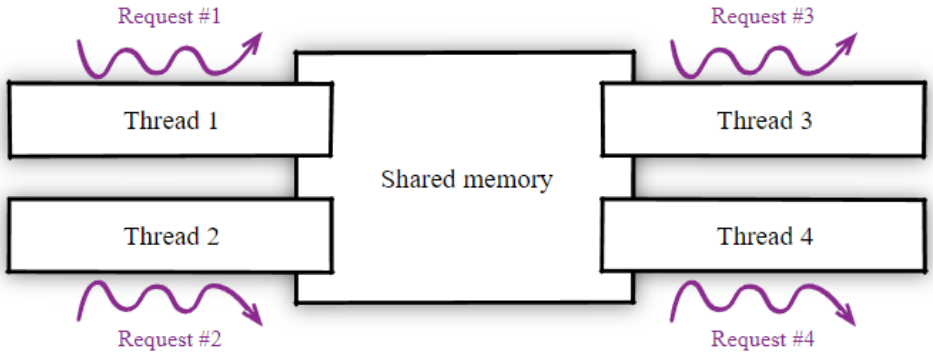

Figura 2.19: Threads [8]

La maggior parte dei framework per lo sviluppo di applicazioni e servizi si basa sul multi-threading. In superficie, il modello di avere un singolo thread per connessione

è rassicurante perché gli sviluppatori possono fare affidamento sul tradizionale codice in stile imperativo.

Il problema sorge quando il carico di lavoro cresce e il kernel comincia a soffrire a causa dell'eccessivo **context switching**. Alcuni thread resteranno bloccati a causa delle operazioni di I/O in attesa di essere soddisfatte, altri thread saranno pronti a gestire i risultati delle operazioni di I/O e altri saranno nel bel mezzo di svolgere attività in CPU.

I kernel moderni hanno degli scheduler ottimi, ma non ci si può aspettare che operino in maniera così efficiente con qualsiasi carico di lavoro, considerando anche che la creazione di ogni thread impiega alcuni millisecondi e ognuno occupa circa 1MB di memoria.

Event loop

Elaborare più connessioni concorrenti con meno thread è possibile quando si utilizza l' aynchronous I/O. Invece di bloccare un thread quando si verifica un'operazione di I/O, passiamo ad un'altra attività pronta per procedere e riprendiamo l'attività iniziale in seguito quando è pronta.

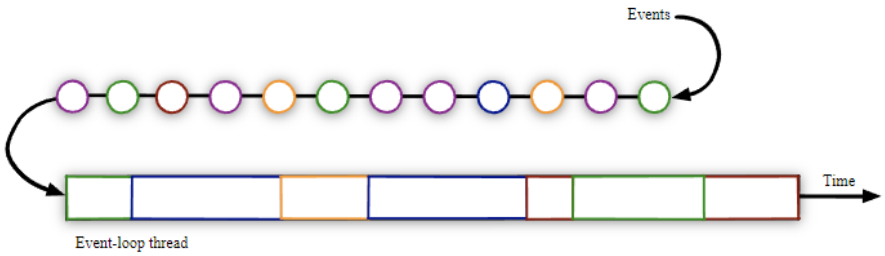

Figura 2.20: Event loop [8]

Vert.x paralleliza i carichi di lavoro concorrenti utilizzando gli **event loop**.

L'event loop è una componente fondamentale del framework, è un meccanismo che gestisce l'esecuzione di codice in modo asincrono ed efficiente. In pratica, l'event loop è un ciclo continuo che riceve e gestisce eventi in arrivo, come richieste di rete o operazioni di I/O, in modo **non bloccante**.

L'event loop di Vert.x è in grado di gestire simultaneamente molti eventi su un singolo thread, senza dover creare un thread separato per ogni richiesta o operazione. Questo approccio consente di gestire un elevato numero di connessioni

simultanee in modo efficiente, riducendo al minimo il consumo di risorse.

È importante notare che il codice eseguito nell'event loop deve essere leggero e non bloccante. Se un'operazione richiede tempo o è bloccante, come una lettura da disco o una richiesta a un database, è possibile delegarla a un thread worker per evitare di bloccare l'event loop principale.

L'utilizzo dell'event loop di Vert.x permette di sfruttare al massimo le capacità di concorrenza e scalabilità del framework, consentendo di gestire un grande numero di richieste in modo efficiente e reattivo.

Capitolo 3

Sviluppo e implementazione

Questo capitolo si concentra sullo sviluppo e l'implementazione del reverse proxy su Quarkus nel contesto del progetto. Saranno esaminate le fasi di progettazione, sviluppo e implementazione del sistema, offrendo una visione dettagliata dei processi coinvolti.

Il capitolo inizia presentando il contesto del progetto, includendo una descrizione dell'azienda coinvolta, la piattaforma nella quale verrà implementato il reverse proxy, ossia NPRI, e l'architettura di riferimento (BFF - Backend for Frontend).

Successivamente, viene esplorata la fase di progettazione, dove si descrive il problema che ha reso necessaria l'implementazione del reverse proxy, la soluzione proposta e le motivazioni dietro questa scelta. Vengono anche discusse le scelte relative al linguaggio di programmazione e al framework utilizzati.

Nella sezione successiva, viene affrontata la fase di sviluppo. Si fornisce una panoramica della struttura del progetto, inclusi i moduli e le componenti principali. Vengono esaminate le librerie e le dipendenze utilizzate e si discutono le sfide affrontate durante lo sviluppo e le strategie adottate per superarle. Vengono anche descritti il workflow di sviluppo seguito, comprese le metodologie e gli strumenti utilizzati.

Infine, viene esaminata l'implementazione del reverse proxy. Si descrive l'ambiente di sviluppo utilizzato, inclusi gli strumenti e le tecnologie coinvolte. Viene spiegata l'architettura a microservizi e il ruolo del reverse proxy nella comunicazione tra di essi. Infine, viene descritto il processo di creazione dei pod Kubernetes per il deployment del reverse proxy.

3.1 Contesto del progetto

Il progetto è stato realizzato per conto di Healthy Reply, al fine di migliorare le prestazioni del BFF già presente all'interno dell'infrastruttura di NPRI, tramite l'implementazione di un reverse proxy in Quarkus, al fine di rendere il sistema più robusto, veloce ed affidabile.

3.1.1 Azienda

Healthy Reply è una società del gruppo italiano Reply che opera nel mercato della Sanità e della Pubblica Amministrazione. La società ha consolidato esperienza nel settore attraverso la creazione e gestione di applicazioni e servizi verticali per Aziende Sanitarie Locali, Aziende Ospedaliere, Comuni e Regioni. Inoltre, possiede competenze approfondite nell'utilizzo di architetture e tecnologie innovative.

Figura 3.1: Healthy Reply logo

Healthy Reply si è specializzata nello sviluppo di telemedicina e servizi/prodotti legati alla gestione remota della salute, nella promozione di miglioramenti e innovazioni nell'uso e nella gestione dei dati socio-sanitari, nonché nella preparazione di modelli innovativi di servizi sanitari territoriali.

Oggi Healthy Reply è riconosciuta come una società del Gruppo Reply specializzata in servizi di consulenza e sviluppo di applicazioni per il settore sanitario e della pubblica amministrazione. In particolare, si rivolge alle Aziende Sanitarie Locali, agli ospedali pubblici e privati, alle RSA, alle Regioni e ai Comuni. Healthy Reply propone portali per i cittadini per mettere a disposizione servizi pubblici (ad esempio, prenotazione di appuntamenti) e, per il settore sanitario e ospedaliero,

propone un modello federato per l'integrazione di soluzioni applicative verticali (ad esempio, analisi delle attività produttive, telemedicina e gestione dei processi di accoglienza dei pazienti).

Healthy Reply vanta una significativa esperienza nella integrazione di sistemi software, acquisita nello sviluppo e nella gestione della piattaforma di integrazione del repository delle cartelle cliniche e dei referti per le regioni Valle d'Aosta e Lombardia. Inoltre, ha esperienza in progetti di progettazione e sviluppo di piattaforme tecnologiche aperte per l'accesso alle informazioni (ad esempio, Open Data), integrate con sistemi esistenti (ad esempio, IEEE, HL7, IHE, ...) orientate a future integrazioni e distribuite nell'accesso e nella fornitura di servizi.

L'esperienza di Healthy Reply nei sistemi di integrazione è iniziata nel 2005 con il progetto della Piattaforma Regionale di Integrazione [9] (Progetto CRS-SISS) che oggi vanta numerose installazioni. Nel contesto delle integrazioni interne (tramite ESB), sono state completate oltre 500 integrazioni, mentre nel contesto del progetto CRS-SISS ci sono circa 450 integrazioni di diversi tipi.

Nel repository delle Aziende Ospedaliere gestito da Healthy, sono archiviati e gestiti oltre 250 milioni di referti fino ad oggi (periodo 2009 - giugno 2023), con una media di oltre 12 milioni di referti per azienda.

3.1.2 NPRI

NPRI (Nuova Piattaforma Regionale di Integrazione) per la Regione Lombardia, una suite di soluzioni infrastrutturali messa a disposizione degli Enti Sanitari pubblici regionali. NPRI ha come base d'utenza gli operatori sanitari, i quali possono utilizzare la piattaforma al fine di gestire dati clinici dei pazienti tramite un repository condiviso tra diverse cliniche. La piattaforma offre, tra molti servizi, un touchpoint di portale clinico e dashboarding per l'analisi dei dati, per citarne tra i più importanti.

3.1.3 Architettura

L'architettura su cui si basa NPRI è mista, in quanto si sta effettuando una transizione da monolita a microservizi. I microservizi sono sviluppati in Quarkus, anche per questo motivo si è approfittato di questo progetto di tesi per poter approfondirne il suo utilizzo, e formano gran parte dei servizi a dispozione, mentre il resto è ancora inserito all'interno del monolita.
La transizione ad un'architettura completamente a microservizi è un passo molto

importante a cui l'azienda tiene a cuore, in quanto renderebbe il sistema molto più scalabile e performante.

Il sistema è organizzato su tre livelli:

- **Browser client**: sul quale è presente un'interfaccia front-end che possa permettere una gestione comoda ed intuitiva dei servizi messi a disposizione degli operatori sanitari.
- **Modulo BFF**: struttura intermedia, che funge da gateway verso la regione di back-end, all'interno del quale verrebbe inglobato il reverse proxy sviluppato in questo progetto di tesi.
- **Moduli back-end**: rappresentano i servizi effettivi, alcuni possono essere utilizzati da altri moduli, altri sono creati per fornire dei servizi diretti agli utenti. Essi rappresentano il cuore del sistema, in quanto tutta la gestione dei dati e delle informazioni è effettuata al suo interno.

3.2 Progettazione

In questa sezione verranno affrontati gli aspetti progettuali, fornendo una guida per affrontare il problema identificato e raggiungere gli obiettivi prefissati. Di seguito saranno analizzati tutti gli elementi chiave inerenti alla fase di progettazione.

3.2.1 Problema, soluzione e motivazioni

Il focus del progetto è sul rinnovamento del modulo BFF, di preciso del reverse proxy al suo interno, questo modulo è stato sviluppato molti anni fa e non risulta più allineato agli strandard attuali. La sua obsolescenza comporta diverse criticità, limitando in primis la sicurezza, l'efficenza, la scalabilità e la manutenibilità dell'applicazione nel suo complesso.
Tra i principali problemi identificati, figurano:

- **Obsolescenza tecnologica:** Il modulo BFF è stato sviluppato utilizzando tecnologie e framework ormai superati. Questo porta a una ridotta compatibilità con le nuove tecnologie emergenti, limitando la possibilità di sfruttare le loro funzionalità avanzate e di migliorare le prestazioni complessive dell'applicazione.
- **Scalabilità limitata:** Il modulo attuale non è stato progettato per gestire grandi volumi di traffico o per scalare in modo efficiente in presenza di un aumento della domanda. Ciò può causare problemi di prestazioni, rallentamenti

o addirittura interruzioni del servizio quando l'applicazione è soggetta a carichi di lavoro elevati.

- **Complessità e manutenibilità:** L'architettura del modulo BFF originale potrebbe essere complessa e difficile da comprendere, rendendo complicata la manutenzione e l'introduzione di nuove funzionalità. La presenza di codice obsoleto o poco strutturato può comportare problemi di bug, rallentamenti e una maggiore complessità di sviluppo.
- **Sicurezza:** Data la sua anzianità, il modulo potrebbe non soddisfare i requisiti di sicurezza attuali. Potrebbero essere presenti vulnerabilità o lacune che espongono l'applicazione a rischi di violazione della sicurezza e accessi non autorizzati.

La soluzione proposta consiste nel rinnovare completamente il modulo BFF, adottando un approccio basato sullo stato dell'arte e sulle migliori pratiche del settore. Ciò comporterà una completa rivisitazione dell'architettura, l'adozione di tecnologie moderne e scalabili, nonché l'applicazione di criteri di sicurezza avanzati.

Andando più nello specifico il problema principale che si vuole risolvere è la parallelizzazione a livello di BFF. Per com'era progettato in precedenza il modulo, nel caso in cui una chiamata fosse pendente, per un qualsiasi motivo (come ad esempio un microservizio particolarmente lento o malfunzionante), essa andava ad arrestare tutte le chiamate successive anche da parte di altri utenti verso altri microservizi, perchè il reverse proxy restava in attesa della risposta di quel singolo thread senza poterne servire di nuovi. Grazie all'architettura reactive di Quarkus si riesce a permettere un disaccoppiamento tra il thread che gestisce la connessione e lo stato della chiamata, in questo modo è possibile gestire più connessioni in parallelo garantendo una fruizione dei servizi più efficiente da parte di tutti gli utenti. Questa parte viene approfondita nel dettaglio più avanti all'interno della sezione "Architettura reactive di Quarkus".

L'obiettivo finale è creare un modulo BFF altamente performante, scalabile, sicuro e facile da mantenere. Questo consentirà di migliorare l'esperienza degli utenti, di garantire una maggiore efficienza nell'elaborazione delle richieste e di facilitare l'aggiunta di nuove funzionalità in futuro. Il rinnovamento del modulo BFF rappresenta un passo fondamentale per modernizzare l'applicazione e mantenerla allineata agli standard attuali del settore.

3.2.2 Linguaggio

La scelta sul linguaggio da utilizzare è ricaduta su Java. Esso offre una vasta gamma di framework e librerie che semplificano lo sviluppo, la gestione e il deployment dei

microservizi.

Java è apprezzato per la sua robustezza, la scalabilità e la capacità di gestire carichi di lavoro intensivi. Inoltre, la vasta comunità di sviluppatori di Java offre un supporto ampio e una vasta gamma di risorse e documentazione per lo sviluppo dei microservizi.

Alcuni dei framework più popolari per lo sviluppo di microservizi in Java includono Spring Boot, Micronaut e **Quarkus**. Questi framework forniscono un'infrastruttura solida per la creazione di microservizi, offrendo funzionalità come l'inversione di controllo (IoC), l'iniezione di dipendenze, la gestione delle richieste HTTP, la scalabilità e la gestione della configurazione.

Inoltre Java è una scelta popolare per le applicazioni enterprise grazie alla sua capacità di scalare in modo efficace. Le applicazioni Java possono gestire grandi volumi di traffico e dati, consentendo di gestire carichi di lavoro elevati e di crescere nel tempo. Java offre un'ampia gamma di strumenti che supportano la gestione efficiente delle risorse, la concorrenza e la distribuzione delle applicazioni su cluster o cloud. Inoltre, le tecnologie come Java EE (Enterprise Edition) forniscono funzionalità avanzate per lo sviluppo di applicazioni enterprise, come il supporto per la scalabilità orizzontale, la gestione delle transazioni e la gestione dei servizi web. Grazie a queste caratteristiche, Java consente alle aziende di costruire applicazioni robuste e ad alte prestazioni che possono crescere e adattarsi alle esigenze del business.

Java è noto per la sua portabilità, che è una delle sue caratteristiche distintive. Le applicazioni Java sono scritte una volta e possono essere eseguite su diverse piattaforme hardware e sistemi operativi. Questo è possibile grazie al concetto di "Write Once, Run Anywhere" (WORA) supportato dalla Java Virtual Machine (JVM).

JVM

La Java Virtual Machine (JVM) è un componente fondamentale dell'ecosistema Java. È un ambiente di esecuzione che interpreta e esegue il bytecode Java, un formato di istruzioni intermedie generato dal compilatore Java a partire dal codice sorgente Java.

Essa fornisce una serie di funzionalità cruciali per l'esecuzione delle applicazioni Java, inclusa la gestione della memoria, la gestione delle eccezioni, la gestione delle transazioni e la sicurezza. Inoltre, offre un'astrazione dell'hardware sottostante, consentendo alle applicazioni Java di essere eseguite in modo indipendente dalla

piattaforma di sistema operativo su cui vengono eseguite.

La JVM svolge diverse attività durante il processo di esecuzione di un'applicazione Java. Innanzitutto, carica il bytecode Java e lo verifica per garantire che sia corretto e sicuro da eseguire. Successivamente, l'interprete JVM converte il bytecode in istruzioni comprensibili per la macchina host su cui viene eseguita la JVM. Durante l'esecuzione, la JVM gestisce anche la gestione della memoria, l'allocation e il deallocation automatico della memoria per gli oggetti Java, liberando gli sviluppatori dal compito di gestire manualmente la memoria. Una delle

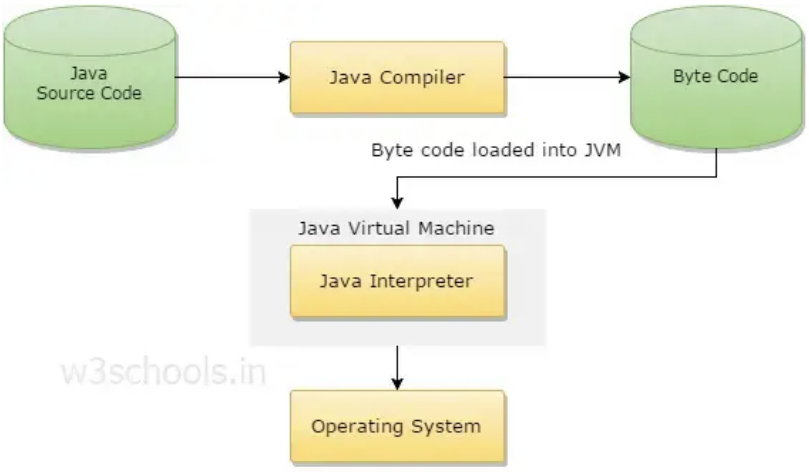

Figura 3.2: Diagramma JVM [10]

caratteristiche distintive della Java Virtual Machine è la sua capacità di eseguire applicazioni Java in modo portabile. Ciò significa che un'applicazione Java può essere scritta una volta e eseguita su diverse piattaforme senza la necessità di apportare modifiche al codice sorgente, come menzionato prima. Questo è possibile grazie all'astrazione fornita dalla JVM che si occupa dei dettagli specifici della piattaforma sottostante.

Inoltre, la JVM supporta il garbage collection, un processo automatico che recupera la memoria occupata dagli oggetti Java non più utilizzati, semplificando la gestione della memoria per gli sviluppatori.

La JVM è disponibile per diverse piattaforme, tra cui Windows, macOS e Linux, ed è stata implementata da vari fornitori, tra cui Oracle con la sua implementazione di riferimento chiamata OpenJDK.

Grazie alla JVM, l'ambiente di esecuzione Java offre portabilità, sicurezza e gestione

della memoria, rendendo Java una scelta popolare per lo sviluppo di applicazioni in diversi contesti, dall'ambito enterprise all'Internet delle cose (IoT) e ai dispositivi mobili.

JDK

La JDK(Java Development Kit) è composto da un insieme di strumenti software di diversa tipologia, al fine di supportare il programmatore in fase di sviluppo e durante l'esecuzione delle applicazioni Java. È necessario per lo sviluppo di qualsiasi applicazione, essa comprende:

- **Compilatore**: Il compilatore Java (javac) traduce il codice sorgente Java in bytecode, un formato di istruzioni leggibile dalla JVM.
- **Java Virtuale Machine**: La Java Virtual Machine (JVM) è l'ambiente di esecuzione in cui vengono eseguite le applicazioni Java.
- **Librerie standard**: La JDK include un'ampia raccolta di librerie standard Java, che forniscono funzionalità comuni per lo sviluppo. Queste librerie includono classi per la gestione delle stringhe, la gestione dei file, la manipolazione dei dati, la connettività di rete e molto altro.
- **Debug e profilazione**: La JDK offre una serie di strumenti per debug e profilazione, che permette agli sviluppatori di identificare e risolvere errori all'interno delle proprie applicazioni. Questi strumenti includono: il debugger (jdb), il profiler (JVisualVM) e altri strumenti di analisi.

3.2.3 Framework

Per quanto riguarda alcuni aspetti teorici relativi al framework sono già stati affrontati nel capitolo teorico, all'interno di questa sezione si mira a dare un contesto più preciso per quanto riguarda l'applicazione e all'utilizzo nel progetto.

La scelta del framework, in una fase iniziale si era pensato ad utilizzare Spring Boot (un altro framework molto utilizzato per lo sviluppo di applicazioni cloud-native e microservizi), è ricaduta su Quarkus, in quanto estremamente innovativo (nato nel 2019) e con delle caratteristiche molto interessanti per quanto riguarda soprattutto l'ottimizzazione nel mondo dei microservizi, grazie alla sua "filosofia" container-first [11].
Quarkus basa le proprie fondamenta principalmente su tre caratteristiche:

- **Container first**: Quarkus è stato creato attorno a questa filosofia, che permette alle applicazioni Quarkus di essere ottimizzate per un basso uso della memoria e tempi di esecuzione molto veloci.

- **Continuum**: Quarkus si lascia dietro l'architettura classica client-server. Esso cerca di fornire un ambiente perfettamente ottimizzato per quelle che sono le nuove caratteristiche delle applicazioni moderne a livello architetturale, di sviluppo, di implementazione ed esecuzione. I microservizi HTTP, applicazioni reactive, architetture event-driven e serverless sono ormai centrali nei sistemi moderni.
- **Kubernetes Native**: La combinazione di Quarkus e Kubernetes fornisce un ambiente perfetto per lo sviluppo di applicazioni scalabili, veloci e leggere. Quarkus riesce ad aumentare significativamente la produttività degli sviluppatori grazie ai suoi tool, integrazione pre-build, servizi e altro ancora.

Container first

[11] Quarkus si offre di presentare ai propri utenti una filosofia container-centrica. L'utilizzo di questi strumenti software è esplosa negli ultimi anni, grazie anche alla crescente diffusione delle architetture a microservizi. Il concetto alla base di questa architettura è quello di avere diversi moduli applicativi distinti, di piccole dimensioni, che cooperano tra di loro al fine di fornire un servizio completo. Dal punto di vista dell'utente non c'è alcuna distinzione tra questa ed un'architettura monolitica, se non per i tempi di esecuzione in alcuni contesti.

La differenza sta tutta nell'ottimizzazione del backend, in questo modo non bisogna caricare in memoria parti di codice non necessarie, il che vuol dire avere un'impronta in memoria minore (più possibilità di parallelizzazione) e tempi di esecuzione più brevi. Per esempio, se l'utente vuole effettuare il login, sarà presente all'interno del backend un microservizio atto *solo* al login, il quale effettuerà l'autenticazione garantendo il token necessario all'utente per poter navigare, il tutto senza dover mandare in esecuzione parti di codice che sarebbero inutili per quel singolo utilizzo. Se ne deduce che l'utilizzo delle risorse è di gran lunga ottimizzato.

Cosa sono i container

Da un punto di vista tecnico ed applicativo i microservizi si traducono in container, un container è un'unita eseguibile all'interno del quale viene impacchettato del software applicativo, con tanto di librerie e dipendenze, ciò significa che il software può essere eseguito in modo consistente ovunque, sia su desktop, infrastrutture IT tradizionali o ambienti cloud.

I container sfruttano le caratteristiche del kernel del sistema operativo, come i namespace e i cgroup di Linux o i silos e gli oggetti processo di Windows, per isolare i processi e controllare le risorse a cui possono accedere, come CPU, memoria

e spazio su disco.

Una delle principali caratteristiche dei container è la loro leggerezza e portabilità. A differenza delle macchine virtuali (VM), i container non richiedono di includere un sistema operativo guest completo per ogni istanza. Invece, possono condividere il sistema operativo host e utilizzare solo le funzioni e le risorse necessarie. Questo rende i container più efficienti in termini di risorse e permette loro di essere avviati e bloccati rapidamente.

Inoltre, i container sono progettati per essere piccoli e modulari, consentendo agli sviluppatori di suddividere le loro applicazioni in componenti più piccoli e indipendenti. Questo approccio favorisce la scalabilità e la flessibilità, poiché i componenti possono essere gestiti e distribuiti separatamente.

I container sono diventati molto popolari nell'ambito dello sviluppo software e delle operazioni IT, in quanto semplificano la distribuzione delle applicazioni, migliorano l'efficienza delle risorse e consentono una maggiore agilità nello sviluppo e nel deployment delle applicazioni.

Build time processing

L'idea centrale di Quarkus è di eseguire, in fase di compilazione, ciò che i framework tradizionali fanno durante l'esecuzione: parsing della configurazione, scansione del classpath, attivazione delle funzionalità in base al caricamento delle classi, e così via.

Il maggior numero possibile di elaborazioni viene effettuato durante la fase di compilazione; di conseguenza, l'applicazione conterrà solo le classi utilizzate durante l'esecuzione. Nei framework tradizionali, tutte le classi necessarie per l'avvio iniziale dell'applicazione rimangono presenti per tutta la durata dell'applicazione, anche se vengono utilizzate solo una volta. Con Quarkus, non vengono neanche caricate nella JVM di produzione. Inoltre, durante l'elaborazione in fase di compilazione, viene preparata l'inizializzazione di tutti i componenti utilizzati dall'applicazione stessa. Ciò comporta un minor utilizzo di memoria e un tempo di avvio più rapido, poiché tutte le elaborazioni dei metadati sono già state effettuate.

Architettura reactive di Quarkus

Quarkus è stato realizzato al fine di garantire una vasta scelta per quanto riguarda gli aspetti architetturali dell'applicazione che si vuole sviluppare. Il suo obiettivo primario è quello di fornire un supporto di prima classe per i nuovi paradigmi, il

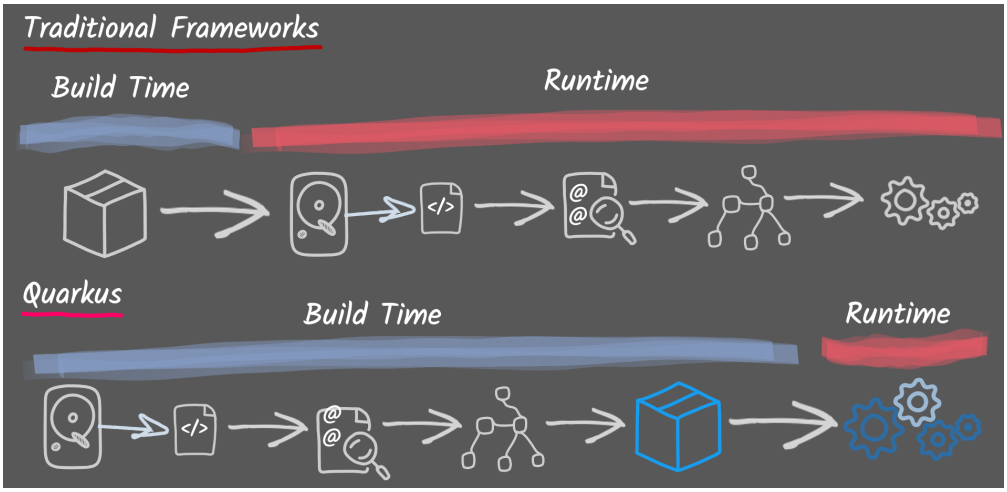

Figura 3.3: Differenza tra framework tradizionali e Quarkus [11]

che non significa che non sia possibile costruire monoliti su Quarkus. Il modello di sviluppo di Quarkus si adatta al tipo di necessità dello sviluppatore, è possibile sviluppare qualsiasi tipo di applicazione, che essa sia un monolite, un microservizio, un'applicazione reactive, event-driven, serverless o FaaS.

Ciò che interessa a noi è l'**architettura reactive** di Quarkus, in quanto permette di ovviare al problema del modulo BFF. Esso era progettato secondo una logica di **blocking I/O**, questo andava a renderlo estremamente inefficiente nel caso in cui ci fossero dei rallentamenti all'interno di singole connessioni verso dei singoli microservizi.
Per blocking I/O si intende che un singolo thread non può fare nulla fino a quando non riceve un segnale di I/O, questo meccanismo può portare a dei seri rallentamenti nel caso di malfunzionamenti a livello di backend, dal momento che se un microservizio non può rispondere per un qualsiasi motivo, il modulo BFF si ritroverà bloccato senza la possibilità di gestire altre connessioni, nonstante esse possano puntare anche ad altri microservizi perfettamente funzionanti in quell'istante di tempo.

Per ovviare a questo problema Quarkus fa ricorso all'event loop di Eclipse Vert.x, sul quale è basato tutto il comportamento reactive del framework.
Per reactive, in accordo con il Reactive Manifesto [12], si intende un'architettura in grando di garantire un sistema:

1. **Responsive**: Il sistema risponde tempestivamente, se possibile. La responsività è la base dell'usabilità e dell'utilità, ma oltre a ciò, la responsività significa che i problemi possono essere rilevati rapidamente e affrontati in modo efficace.

I sistemi responsivi si concentrano su tempi di risposta rapidi e consistenti, stabilendo limiti superiori affidabili in modo da offrire una qualità del servizio costante. Questo comportamento coerente semplifica la gestione degli errori, aumenta la fiducia degli utenti finali e favorisce ulteriori interazioni.

2. **Resiliente**: Il sistema rimane responsivo di fronte a guasti. Questo si applica non solo ai sistemi altamente disponibili e critici, ma a qualsiasi sistema che non sia resiliente diventerà non responsivo dopo un guasto. La resilienza si ottiene attraverso la replicazione, il confinamento, l'isolamento e la delega. I guasti sono contenuti all'interno di ogni componente, isolando i componenti l'uno dall'altro e garantendo così che le parti del sistema possano fallire e riprendersi senza compromettere il sistema nel suo complesso. Il ripristino di ogni componente viene delegato a un altro componente (esterno) e l'alta disponibilità è garantita dalla replicazione quando necessario. Il cliente di un componente non è gravato dalla gestione dei suoi guasti.

3. **Elastico**: Il sistema rimane responsivo con un carico di lavoro variabile. I sistemi reattivi possono reagire ai cambiamenti nel tasso di input aumentando o diminuendo le risorse allocate per gestire questi input. Ciò implica progettazioni che non presentano punti di contesa o collo di bottiglia centrali, consentendo la frammentazione o la replicazione dei componenti e la distribuzione degli input tra di essi. I sistemi reattivi supportano algoritmi di scaling predittivi e reattivi fornendo misure di prestazioni in tempo reale pertinenti. Raggiungono l'elasticità in modo economico su piattaforme hardware e software di tipo commodity.

4. **Message Driven**: I sistemi reattivi si basano su un passaggio di messaggi asincrono per stabilire un confine tra i componenti che garantisce un basso accoppiamento, isolamento e trasparenza della posizione. Questo confine fornisce anche i mezzi per delegare i guasti come messaggi. L'uso esplicito del passaggio di messaggi consente la gestione del carico, dell'elasticità e del controllo del flusso mediante la modellazione e il monitoraggio delle code dei messaggi nel sistema e l'applicazione della retropressione quando necessario. La messaggistica trasparente rispetto alla posizione come mezzo di comunicazione consente alla gestione dei guasti di funzionare con gli stessi costrutti e la stessa semantica all'interno di un cluster o in un singolo host. La comunicazione non bloccante consente ai destinatari di utilizzare risorse solo quando sono attivi, riducendo l'overhead di sistema.

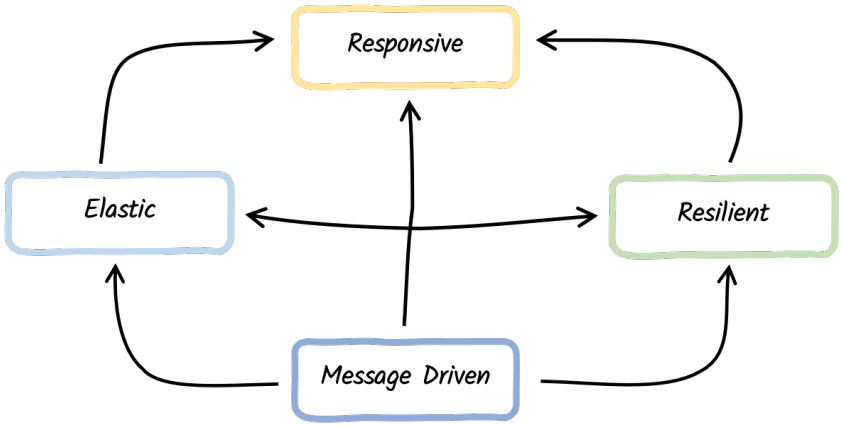

Figura 3.4: Diagramma sistemi reactive [13]

Modello di esecuzione reactive

[13] Nonostante i numerosi vantaggi del non-blocking I/O, è importante considerare che presenta delle sfide. Infatti, introduce un nuovo modello di esecuzione molto diverso da quello utilizzato dai framework tradizionali.

Le applicazioni tradizionali utilizzano un approccio di blocking I/O e un modello di esecuzione imperativo (sequenziale). Ad esempio, *come nel nostro caso di studio*, in un'applicazione che espone un endpoint HTTP, ogni richiesta HTTP viene gestita da un singolo thread. Durante l'elaborazione della richiesta, il thread rimane impegnato esclusivamente con quella specifica richiesta. Se l'elaborazione richiede l'interazione con un servizio remoto, viene utilizzato un I/O bloccante e il thread rimane in attesa del risultato. Sebbene questo approccio sia semplice da implementare (poiché segue un flusso sequenziale), presenta alcuni svantaggi. Ad esempio, per gestire richieste concorrenti, è necessario utilizzare un worker thread pool. La dimensione di questo pool influenza direttamente la capacità di gestione delle richieste simultanee dell'applicazione. Inoltre, ogni thread richiede risorse di memoria e CPU, quindi un numero eccessivo di thread può portare a un consumo eccessivo di risorse del sistema. Come osservato in precedenza, l'approccio non-blocking I/O evita tale problema. Un numero limitato di thread può gestire simultaneamente molte operazioni di I/O. Nel caso in esempio dell'endpoint HTTP, l'elaborazione della richiesta avviene su uno di questi thread dedicati all'I/O. Dato che questi thread sono in numero limitato, è fondamentale utilizzarli in modo oculato. Quando è necessario effettuare una chiamata a un servizio remoto durante l'elaborazione della richiesta, non è possibile bloccare il thread. Invece, si pianifica l'operazione di I/O e si passa una continuazione, ovvero il codice da eseguire una volta completata l'operazione di I/O.

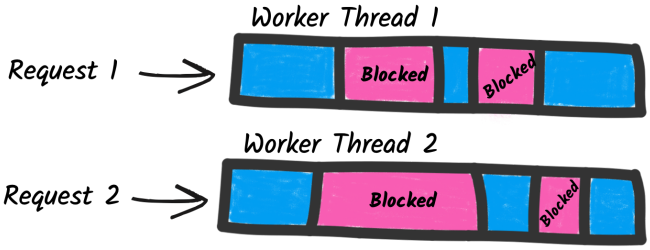

Figura 3.5: Thread bloccanti [13]

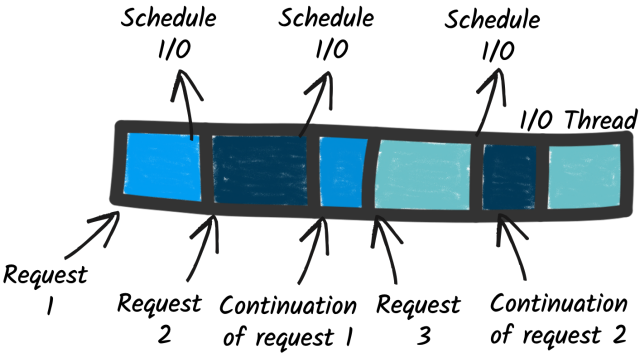

Figura 3.6: Thread reattivi [13]

3.3 Sviluppo

All'interno di questa sezione andremo ad approfondire lo sviluppo e il processo di realizzazione del progetto, fase cruciale all'interno della quale le idee prendono forma e si trasformano in soluzioni reali. Verranno affrontati aspetti chiave come la struttura del progetto, le librerie utilizzate, le sfide affrontate e il workflow adoperato.

3.3.1 Struttura del progetto

Il progetto è stato organizzato utilizzando Maven come strumento di gestione e dipendenze. Apache Maven è un tool in grado di garantire una struttura solida per progetti Java, a livello di dipendenze, build, compilazione, testing e distribuzione. Esso aiuta il programmatore a gestire efficientemente le dipendenze di un progetto, ossia le librerie esterne necessarie al fine di garantire il funzionamento corretto dell'applicazione.

Con Maven è possibile definire le dipendenze all'interno di un file di configurazione denominato `pom.xml`, si occupa di scaricare autonomamente le librerie richieste da

Figura 3.7: Maven logo

repository centralizzate, come ad esempio Maven Central Repository, all'interno del quale si possono trovare una grossa quantità di librerie. Questa funzionalità aiuta moltissimo gli sviluppatori in quanto ad essi basterà aggiungere una dipendenza all'interno del file e in automatico si avrà accesso a tutte le funzionalità messe a disposizione dalle dipendenze esterne.

Maven offre una struttura standardizzata per organizzare il codice sorgente, il file di configurazione e le risorse del progetto. Questo facilita la collaborazione per gli sviluppatori all'interno del team e la condivisione del codice sorgente. Il progetto è strutturato nel seguente modo:

- **Directory principale del progetto**: all'interno del quale il file più importante è `pom.xml`, POM sta per Project Object Model. Questo file XML è il cuore del progetto Maven e contiene tutte le sue informazioni, come il nome, la versione, le dipendenze e i plugin. Nelle sezioni successive verranno fornite le specifiche inerenti al progetto.
- **Directory di base**: `/src`, al suo interno è possibile trovare la directory `/src/java` dentro la quale si trova il codice sorgente Java del progetto; all'interno di `/src/main/resources` si trovano le risorse aggiuntive per il progetto, come il file di configurazione, di properties o le risorse statiche utilizzate dall'applicazione (come file di testo).
- **Directory di test**: `/src/test` dentro la quale si trovano le directory `/src/test/java`, che contiene il codice sorgente per il test del progetto (i moduli di unit e integration test vengono scritti al suo interno), e `/src/test/resources` contiene le risorse aggiuntive per i test, come le configurazioni per l'ambiente di testing.
- **Output del progetto**: `/target`, all'interno della quale si trovano i file di output generati da Maven. Include anche i file JAR e WAR del progetto, nonché i file di output generati durante il processo.

Maven promuove la filosofia *convention over configuration*, secondo la quale le impostazioni e le configurazioni sono imposte per convenzioni. Questo riduce da un lato la possibilità di "personalizzare" certi aspetti dell'applicaizone, ma dall'altro

fornisce una notevole semplificazione in fase di configurazione, oltre a evitare una quantità eccessiva di stesura di codice per questi scopi.

Per quanto concerne il nostro progetto la creazione della struttura è stata affidata totalmente a Quarkus, che offre un tool online per la generazione di applicazioni [14]. Esso permette di definirne il nome, lo strumento di build (nel nostro caso Maven), la versione di java e soprattutto tutte le dipendenze da inserire nel `pom.xml`. Il sito contiene una notevole quantità di dipendenze esterne da poter inserire, per la gestione dei dati, delle connessioni, del comportamento reactive, per applicazioni cloud e molto altro.

3.3.2 Librerie

Le librerie sono uno strumento essenziale per lo sviluppo, in quanto consentono l'accesso a funzionalità complesse, facendo risparmiare molto tempo e sforzi allo sviluppatore. Esse sono dei pacchetti di codice già testati, sviluppati da terzi, che offrono al programmatore molte funzionalità, che possono essere specifiche per risolvere un determinato task oppure possono fungere da mezzo per semplificare alcune operazioni.
Un aspetto fondamentale delle librerie è che possono anche migliorare notevolmente le prestazioni, se sviluppate al fine di gestire le risorse di sistema in modo efficiente. Le librerie più utilizzate sono spesso gestite da team di esperti e sono costantemente sotto monitoraggio per quanto riguarda la manutenzione e il bug fixing, aspetto fondamentale che ne garantisce una certa affidabilità.

Per gestire l'utilizzo delle librerie esterne è necessario utilizzare un sistema di dipendenze, fornito da Maven nel nostro caso, che permette il download delle librerie in maniera dinamica dalla repository di riferimento all'ambiente di esecuzione del progetto stesso.
All'interno di questa sezione andremo ad analizzare le librerie utilizzate per i fini del progetto, discutendo del motivo per il quale sono state scelte e sulla loro integrazione.

Le librerie fondamentali per lo sviluppo del codice sorgente sono fornite da: Quarkus, Vert.x e Javax.

Quarkus

Per quanto riguarda la libreria di Quarkus, è necessario inserire la dipendenza:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-vertx-web</artifactId>
```

```
</dependency>
```

all'interno del file `pom.xml`, nella sezione `dependencies`.

Essa fornisce tutte le funzionalità necessarie per l'utilizzo di Quarkus in maniera efficiente, nel nostro caso è stato utilizzato solo per le classi StartupEvent, ShutdownEvent e Route.

StartupEvent e **ShutdownEvent** sono due funzioni necessarie per stabilire un Observer Pattern all'interno della nostra applicazione, quando esse vengono istanziate tramite `injection` sono richiamate delle funzioni che vedremo successivamente nella sezione inerente al codice sorgente. Gli oggetti vengono istanziati, rispettivamente, in fase di avvio dell'applicazione e in fase di shutdown, come intuibile dai nomi, e permettono di gestire queste due fasi del processo.

Route [15] è un'annotazione che permette di creare delle route dinamiche associandole a delle funzioni che rappresentano di fatto un endpoint HTTP. Un aspetto fondamentale della nostra applicazione è quello di andare a sfruttare il meccanismo **reactive** di questa Route, secondo le politiche di Quarkus.
Come approfondito in precedenza, il motore di Quarkus è non bloccante e reattivo. Di conseguenza, ogni chiamata HTTP al modulo Quarkus viene gestita dall'event loop e successivamente mandata al codice che gestisce la richiesta.

Il metodo associato all'annotazione `@Route` viene invocato in base alle caratteristiche della chiamata HTTP.
Vediamo di seguito alcuni esempi:

```
@Route(methods = Route.HttpMethod.GET) [A]
void hello(RoutingContext rc) {
    rc.response().end("hello");
}

@Route(path = "/world") [B]
String helloWorld() {
    return "Hello world!";
}

@Route(path = "/greetings", methods = Route.HttpMethod.GET) [C]
void greetingsQueryParam(RoutingExchange ex) {
    ex.ok("hello " + ex.getParam("name").orElse("world"));
}
```

```
@Route(path = "/greetings/:name", methods = Route.HttpMethod.GET) [D]
void greetingsPathParam(@Param String name, RoutingExchange ex) {
    ex.ok("hello " + name);
}
```

[**A**] Viene intercettata qualsiasi chiamata di tipo GET, a prescindere dal path, e viene richiamata la funzione `hello(RoutingContext rc)`, che accetta come parametro un oggetto di tipo RoutingContext, il quale funzionamento verrà approfondito successivamente, e fornisce come risposta una semplice stringa "hello".

[**B**] In questo caso viene creato un endpoint sul path "/world", di conseguenza verrà richiamata la funzione `helloWorld()` che fornisce la stringa "Hello World!" in risposta.

[**C**] Endpoint sul path "/greetings" e metodo GET, in questo caso viene invocata la funzione `greetingsQueryParam(RoutingExchange ex)` che andrà a restituire come risposta "hello [name]" qualora sia presente il parametro "name" nella richiesta o "hello world" nel caso non sia presente alcun parametro.

[**D**] In quest'ultimo esempio viene fornito un endpoint richiamato in presenza di un path di tipo "/greetings/:name", dove "name" può assumere qualsiasi valore, e come metodo si accetta solo GET. In questo caso la funzione `greetingsPathParam(@Param String name, RoutingExchange ex)` andrà a fornire come risposta "hello [name]", dove name acquisisce il valore direttamente dal path.

L'annotazione `@Route` permette di configurare:

1. **path**, per il routing basato sul riscontro di un path ben preciso
2. **regex**, per il routing tramite regex matching
3. **methods**, i meetodi HTTP come GET, POST, ecc.
4. **type**, può essere "normal" (non bloccante), "blocking" (qualora siano attesi input o output) o "failure" per indicare una route richiamata in caso di errori
5. **order**, indica l'ordine di esecuzione quando sono coinvolte più route nella gestione della richiesta, può essere molto utile in caso di ambiguità (ad esempio se due route hanno come path "/example/me" e "example/:id", l'ordine diventa necessario per dare una priorità ad una chiamata piuttosto che all'altra), più è basso il valore più è alta la priorità.
6. **produces** e **consumes** vengono usati per imitare un pattern produttore-consumatore

Vert.x

La libreria di Vert.x è fondamentale per gestire le chiamate HTTP in modo reattivo, essa viene utilizzata per stabilire una connessione verso il backend all'interno della funzione @Route e per utilizzare alcune classi fondamentali per la gestione delle chiamate HTTP.
Essa è inclusa nelle dipendenze tramite:

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web-client</artifactId>
</dependency>
```

Le classi principali utilizzate sono: WebClient e RoutingContext.

WebClient [16] è un client HTTP (anche HTTP/2) asincrono. Esso è stato utilizzato per instaurare una connessione HTTP con il backend.
Come spiegato in precedenza, il modulo BFF sul quale operiamo si interpone tra il client e il server (composto da diversi microservizi), fungendo da "centro di smistamento" delle richieste verso i vari endpoint HTTP. Gli endpoint sono collocati all'interno di microservizi diversi che si trovano in ascolto in porte o addirittura server diversi, di conseguenza ognuno di essi avrà una propria coppia `[indirizzo IP - porta]` per poter essere raggiunta.
Il nostro modulo mette a disposizione un singolo dominio al client, che richiamerà il nostro servizio Quarkus senza esserne al corrente, il nostro compito è quello di inoltrare la richiesta al microservizio corretto, basandoci solo ed unicamente sul path che ha digitato l'utente, per poi restituire la risposta.
Il WebClient ci permette di creare un client momentaneo, che andrà a contattare il backend per conto del "vero" client.
La seguente istruzione:

```
WebClient myClient = WebClient.create(vertx);
```

permette di istanziare l'oggetto myClient di tipo WebClient, grazie ad esso è possibile effettuare delle chiamate a un server remoto.
Quindi, una volta ricevuta la richiesta all'interno del modulo BFF, essa verrà impacchettata in una nuova request da inviare al server.
Tramite l'istruzione:

```
myClient.request(HttpMethod method, int port, String host,
    String path)
.sendBuffer(myBuffer)
.onSuccess( res -> //success )
.onFailure( err -> //error );
```

è possibile effettuare una chiamata ad un server in ascolto su `host:port`, con con un determinato `method` e il `path` ereditato dalla chiamata dell'utente.
`.sendBuffer` permette di inserire un payload (qualora sia necessario).
`.onSuccess e .onFailure` permettono di gestire la risposta in caso di esito positivo e negativo. Vedremo più avanti l'utilizzo nel nostro caso specifico.

RoutingContext [17] rappresenta il contesto per la gestione della richiesta HTTP su Vert.x Web. La sua utilità principale è quella di permettere l'accesso a `HttpServerRequest` e `HttpServerResponse` e consente di memorizzare dati di qualsiasi genere per tutta l'attività del contesto, il quale viene eliminato non appena instradato nell'handler per gestire la richiesta.
Esso fornisce anche accesso alla sessione, ai cookie e al corpo della richiesta, a patto che siano richiamati gli handler corretti.
Il RoutingContext è fondamentale per il nostro scopo, in quanto è quell'oggetto che possiede tutti i dati necessari per la manipolazione che dobbiamo eseguire: il nostro scopo è determinare l'host corretto da contattare guardando solo ed esclusivamente il path della richiesta.
Esso possiede al suo interno i metodi `request()`, che ritorna un oggetto di tipo HttpServerRequest, il quale contiene al suo interno la richiesta HTTP in corso e `response()`, che permette di manipolare l'oggetto HttpServerResponse, ossia la risposta da fornire al client.
Per instaurare la connessione verso il servizio corretto è necessario andare ad estrapolare l'URI della richiesta da parte dell'utente, tramite:

```
String myURI = ctx.request().uri();
```

dove `ctx` rappresenta il contesto della richiesta attuale,
`.request().uri()` ritorna la stringa corrispondente all'intero URI digitato dall'utente.
Un altro esempio di utilizzo del RoutingContext:

```
ctx.response()
    .setStatusMessage(response.statusMessage())
    .setStatusCode(response.statusCode())
    .putHeader("Set-Cookie", responseCookies)
    .putHeader("content-type", response.getHeader("content-type"));
```

All'interno del quale andiamo a manipolare la risposta da fornire al client, in questo caso andando a settare varie informazioni, come lo `status message` e lo `status code`, vari header, tra cui il `set-cookie` e il `content-type`.

Javax

Javax è una libreria dello standard Java Enterprise Edition (Java EE). Essa offre delle funzionalità adibite allo sviluppo di applicazioni Java in ambito aziendale. Questa libreria ricopre una gamma molto vasta di funzioni, tra cui la sicurezza, il web, la persistenza dei dati e molto altro.
Le sottolibrerie che ci interessano sono:

- `javax.enterprise`, è la libreria principale per lo sviluppo di applicazioni aziendali. Le classi o interfacce che ci interessano sono: `javax.enterprise.context`, della quale è stato utilizzata l'annotazione `@ApplicationScoped` e `javax.enterprise.event`, all'interno della quale viene definita l'annotazione `@Observes`.

- `javax.inject`, è una libreria standard di Java che fornisce le classi ed annotazioni base per la Dependency Injection (DI). Essa contiene l'annotazione `@Inject` che indica il punto in cui deve essere effettuata un'iniezione delle dipendenze.

@ApplicationScoped [18] è un'annotazione presente all'interno di `javax.enterprise.context`, che permette di definire un bean la quale instanza resterà attiva e le cui risorse resteranno condivise per tutto il ciclo di vita dell'applicazione.
L'annotazione `@ApplicationScoped` viene tipicamente utilizzata per i bean che mantengono degli stati globali per l'applicazione. In quanto deve essere consentito a diverse parti dell'applicazione stessa di poter accede e manipolare la *stessa* istanza del bean, in modo da garantire una certa coerenza.
Tant'è che nel nostro caso viene utilizzato per la classe "main", ossia quella che deve mantenere l'endpoint (annotato da @Route) costantemente in ascolto.

@Observes [19] serve per annotare un metodo che desidera osservare gli eventi pubblicati all'interno del Contexts and Dependency Injection (CDI). Nel momento in cui una classe viene istanziata, vengon richiamati tutti i metodi che sono in "osservazione" di quel tipo.
Nel nostro caso viene utilizzato per scatenare la funzione

```
void onStart(@Observes StartupEvent ev)
```

la quale viene eseguita quando nel momento in cui viene istanziato un oggetto di tipo StartupEvent, ossia all'avvio dell'applicazione, al suo interno sono dichiarate tutte le funzioni di loading, che vedremo successivamente nella sezione relativa al codice sorgente.

@Inject [20] fa parte dello standard di Dependency Injection definito dal Contexts and Dependency Injection di Java EE. Essa viene utilizzata per indicare al framework di Dependency Injection che una determinata dipendenza deve essere iniettata all'interno di una classe o di un componente.
Quando viene applicata a un campo, un costruttore o un metodo setter, l'annotazione @Inject segnala al framework di DI che la dipendenza associata a quel campo o metodo deve essere risolta e iniettata dal container o da un'appropriato provider di dipendenze. Grazie al suo utilizzo è possibile delegare la creazione delle dipendenze al framework di DI, senza doverlo fare manualmente.

3.3.3 Sfide da superare

All'interno di questa sezione andremo ad analizzare quelle che sono le sfide principali e quelle che sono intercorse durante lo sviluppo del progetto, oltre ad essere approfondite le soluzioni intraprese.

Innanzitutto la sfida principale si incentrava sul capire come creare un reverse proxy da zero in Quarkus, considerando che di fatto il modulo BFF si comporta da reverse proxy con delle features aggiuntive.
Il nocciolo del problema consiste nel capire come reindirizzare le chiamate verso un endpoint di cui solo il reverse proxy è a conoscenza, tramite il path inserito dall'utente.

Esempio:
Un utente digita sulla barra di ricerca "http://www.hello.com/tellmehi". Questo URL porterà a ricevere come risposta "hi :)" sul proprio browser.
Di conseguenza il server DNS andrà a cercare l'indirizzo IP collegato al dominio hello.com, che sarà quello del nostro reverse proxy. Quindi la richiesta HTTP giungerà al nostro servizio, con la forma: GET www.hello.com/tellmehi (ossia una semplicissima HTTP Request GET con un URL associato).
Da questo URL noi possiamo estrapolare due informazioni: il dominio (www.hello.com) e il path (/tellmehi).
L'informazione di nostro interesse consiste nel path, in quanto ci permette di comprendere quale sia l'effettiva intenzione dell'utente, dal momento che il dominio è necessario solo al DNS per poter trovare il nostro indirizzo IP e reindirizzare il pacchetto HTTP verso di noi. Di conseguenza, *ogni* connessione che giungerà a noi avrà lo stesso dominio.
Noi sappiamo (in quanto interni al sistema) che l'endpoint HTTP che fornisce come risposta "hi :)" si trova su un server in ascolto su localhost:80/tellmehi.

Problema 1: Come contatto il server per conto dell'utente?

Primo problema che, per quanto banale, risulta essere fondamentale per il funzionamento dell'applicazione.
Per fare in modo di mandare la richiesta HTTP al server per conto dell'utente è necessario interfacciarci noi stessi come client nei confronti del server. Siamo già in possesso del pacchetto HTTP da inoltrare, di conseguenza è solo necessario cambiare l'indirizzo di partenza e destinazione, e una volta ricevuta la risposta, inoltrarla all'utente finale per com'è.
Per sopperire a questo problema, dopo una serie di tentativi tramite l'utilizzo di diverse classi, tra cui Undertow, sono giunto all'utilizzo del WebClient di Vert.x. Il quale permette di creare un client pronto a contattare un server, una volta inserite le informazioni necessarie (come host, porta, ecc).

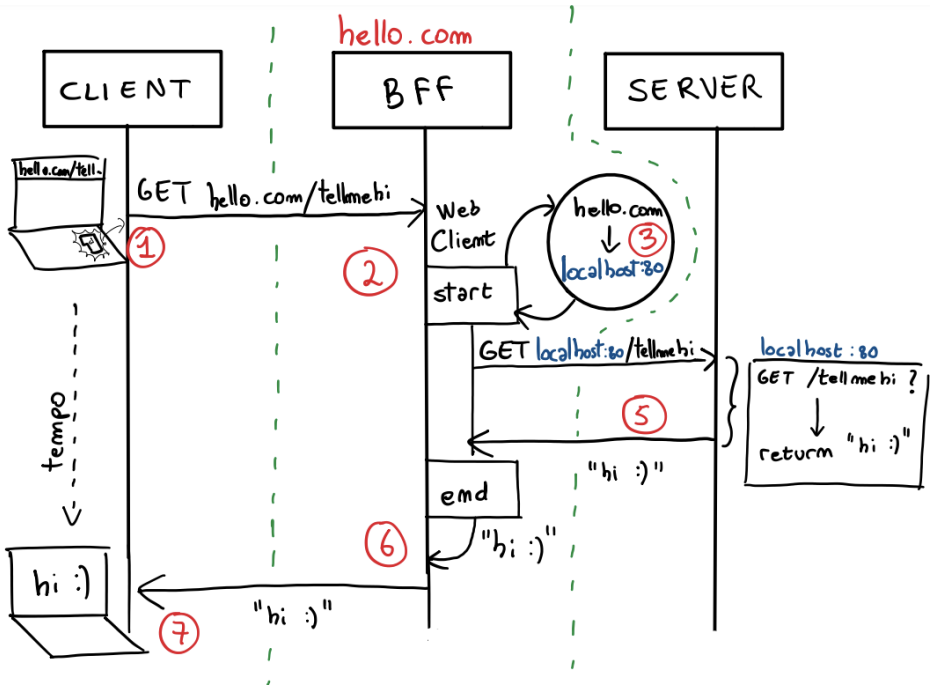

Figura 3.8: Schema chiamate HTTP con BFF

Spiegazione diagramma:

1. L'utente digita l'URL "www.hello.com/hellmehi" sulla barra di ricerca. Di conseguenza il server DNS intercetta la richiesta, cerca l'indirizzo IP associato al dominio e spedisce il pacchetto HTTP verso di noi.

2. Il pacchetto HTTP giunge al nostro servizio, viene creata un'istanza di WebClient che impacchetterà tutte le informazioni contenute nella richiesta originaria e le spedirà al server.

3. Viene sostituito il dominio, insieme alla porta, dell'URL con quelli del server di destinazione.

4. Il WebClient crea una richiesta asincrona verso il server di tipo GET su localhost:80/tellmehi

5. Il server possiede un'API in ascolto su quella porta che risponde a quel path, che manda come response una semplice stringa contenente "hi :)"

6. Il WebClient passa al nostro applicativo la risposta ottenuta e l'istanza viene distrutta

7. Il BFF inoltra all'utente finale la response "hi :)" ottenuta da parte del server, senza che esso possa accorgersi di nulla

Problema 2: Come creare un metodo che permetta di raggiungere l'endpoint su localhost:80 senza che l'utente ne venga al corrente?

Per fare in modo di soddisfare questa esigenza è necessario che l'utente non abbia alcun accesso alle informazioni in nostro possesso, come gli indirizzi dei vari server in cui si trovano le risorse richieste. Nella barra dell'URL del client deve sempre comparire il dominio esterno.
Può sembrare un problema banale, ma durante lo sviluppo sono incappato in un errore nel quale il server di destinazione mandava una redirect 301 contenente l'intero URL corretto da contattare.
Questo accadeva nel caso in cui nell'URL mancasse uno slash alla fine (esempio: www.hello.com/tellmehi invece di www.hello.com/tellmehi/). Il server correggeva l'URL sostituendo "localhost:80" a "www.hello.com" e inviava una redirect con "localhost:80/tellmehi/" (notare lo slash), di conseguenza il browser - a causa della redirect - avviava automaticamente un'altra richiesta con il nuovo URL, di conseguenza nella barra di ricerca del client non compariva più il dominio "www.hello.com" ma "localhost:80".

Per creare il client inizialmente è stato utilizzato Undertow, una classe per la gestione di connessioni HTTP, e per risolvere questo problema ho inserito un controllo sull'URL per rimpiazzare a mano lo slash, il metodo funzionava però era troppo macchinoso.

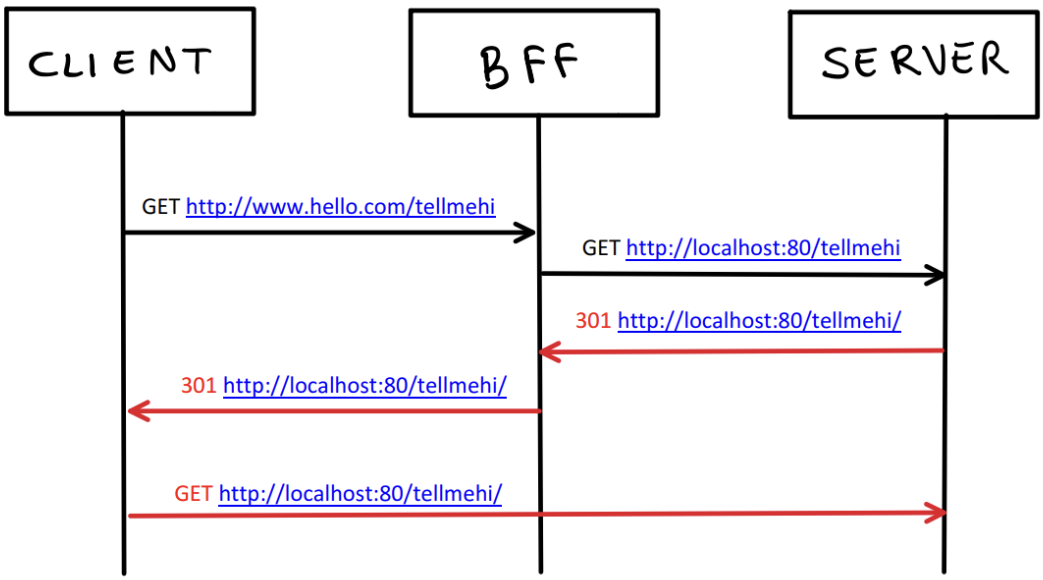

Figura 3.9: Problema redirect 301

In seguito ho scoperto l'utilizzo del WebClient di Vert.x, il quale permetteva di gestire automaticamente la presenza/assenza di slash all'interno dell'URL.
Grazie a questa implementazione è stato possibile ovviare al problema tramite un funzionamento interno del WebClient, in questo modo non è stato necessario dover correggere manualmente l'URL.

Problema 3: Come rendere generica la soluzione?

Fino a questo momento abbiamo visto come sostituire il singolo dominio e la porta di destinazione in un caso di esempio. Ovviamente la soluzione deve essere funzionale per molte tipologie di chiamate e verso diversi microservizi, che giaceranno su host e porte differenti. All'interno del progetto abbiamo a disposizione un file di configurazione, chiamato `application.properties`. Al suo interno sono trascritti tutti gli endpoint e le route con il quale è necessario fare match. Nella prossima sezione andremo a approfondire nello specifico il funzionamento di questo file.
Per rendere più generica la soluzione quindi è stato necessario creare un sistema di *matching*, che permettesse di intercettare i vari path e di conseguenza scegliere l'host corretto al quale inoltrare la chiamata.
Per fare ciò sono state create due liste che vengono caricate all'avvio dell'applicazione, una contenente i path con cui avere un riscontro e l'altra contenente tutti gli host associati. Una volta che viene effettuato il riscontro tra l'URL contenuto nella

richiesta HTTP da parte dell'utente e un elemento della lista, allora verrà sostituito l'host da raggiungere con quello corretto. In questo modo tutto il processo viene automatizzato e l'inoltro può funzionare con qualsiasi genere di chiamata, a patto che il path sia presente nella lista, in caso contrario il BFF manderà un messaggio di errore 404.

Grazie all'unione di queste soluzioni è stato possibile creare un reverse proxy affidabile, con delle prestazioni notevolmente migliori rispetto alla versione precedente e molto più scalabile.

3.3.4 Codice sorgente

All'interno di questa sezione viene analizzato il codice sorgente, approfondite le tecniche utilizzate e le soluzioni intraprese al fine di ottenere un'applicazione affine alle esigenze sopra menzionate.

All'interno del progetto il codice sorgente è disposto all'interno di un singolo file, oltre ad essere presente un'interfaccia per la gestione del caricamento della configurazione ed il file di configurazione.

File `application.properties`

All'interno del **file di configuazione** sono presenti varie informazioni, tra cui:

- **HTTP port**: la porta su cui è in ascolto l'applicazione
- **hosts**: tutti gli indirizzi IP a cui fare riferimento per contattare gli endpoint HTTP
- **match paths**: sono tutti i path con cui bisogna ricercare il match quando sopraggiunge una richiesta HTTP
- **targeturls**: una volta registrato il match con uno dei path è necessario sostituire la parte di host del dominio con il "targeturl" corrispondente. Esso è composto da [host]/[path residuo], dove host è preso dalla lista **hosts** sopramenzionata, e il path residuo varia in base all'endpoint da contattare
- **exceptions**: rappresentano delle tipologie di URL che non necessitano di essere modificate in quanto interne al sistema

Hosts

Qui possiamo vedere un esempio, a sinistra vi è il tag che viene inserito successivamente nei target urls, a destra invece vi è l'indirizzo IP e la porta corrispondente.

Figura 3.10: hosts (indirizzi IP oscurati)

Matching paths

Quando il nostro modulo riceve una richiesta HTTP, essa sarà associata ad un URL, il compito di questa lista è quello di fornire un insieme di **casi validi** per cui si può proseguire. Qualora l'url non sia affine a questi casi si restituisce un errore (a meno che non faccia parte della lista di exceptions).

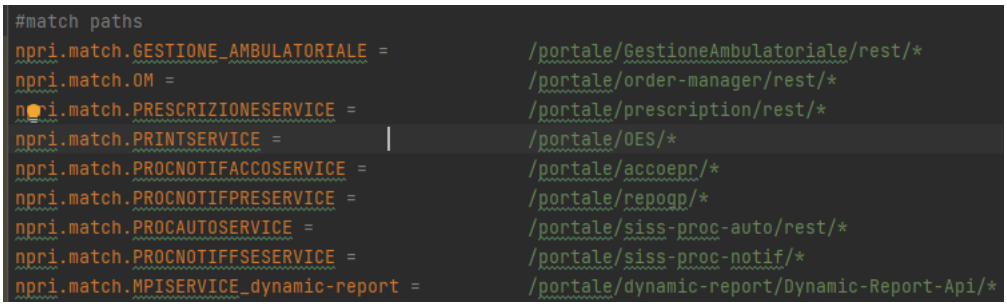

Figura 3.11: Lista dei matching paths

Nella parte sinistra è presente una denominazione puntata particolare, andremo a spiegarne il significato più avanti, che rappresenta il servizio di riferimento a cui è legato il path (a destra) da matchare, nel path è presente un asterisco, da quel punto in poi può esserci scritta qualsiasi, matcherà in qualsiasi caso.

Target urls

Una volta eseguito un riscontro con un elemento della lista di prima, sarà necessario *sostituire* la prima parte del path con la stringa corrispondente all'elemento matchato (avranno la stessa keyword).

```
#targeturls
npri.targeturl.GESTIONE_AMBULATORIALE =          ${GESTIONE_AMBULATORIALE}/GestioneAmbulatoriale/rest
npri.targeturl.OM =                              ${OM}/order-manager/rest
npri.targeturl.PRESCRIZIONESERVICE =             ${PRESCRIZIONESERVICE}/ModuloPrescrittivo/rest
npri.targeturl.PRINTSERVICE =                    ${PRINTSERVICE}/OES
npri.targeturl.PROCNOTIFACCOSERVICE =            ${PROCNOTIFACCOSERVICE}/accoepr
npri.targeturl.PROCNOTIFPRESERVICE =             ${PROCNOTIFPRESERVICE}/repogp
npri.targeturl.PROCAUTOSERVICE =                 ${PROCAUTOSERVICE}/siss-proc-auto/rest
npri.targeturl.PROCNOTIFFSESERVICE =             ${PROCNOTIFFSESERVICE}/siss-proc-notif
npri.targeturl.MPISERVICE_dynamic-report =       ${MPISERVICE}/dynamic-report/Dynamic-Report-Api
```

Figura 3.12: Lista dei target url

File `ReverseProxyReactiveRoute.class`

Questo file rappresenta la classe principale nella quale viene racchiuso tutta la logica del proxy, nelle sottosezioni successive verranno approfondite tutte le parti più importanti.

Import
La prima sezione del file riguarda l'import delle librerie, parte già ampiamente discussa in precedenza, di conseguenza verranno discussi solo dei piccoli aspetti tecnici riguardo l'utilizzo di certe classi, non menzionate in precedenza.

`io.vertx.core.buffer.Buffer` è una classe che permette di istanziare una sequenza di byte che possono essere letti o sovrascritti, permette l'espansione automatica della quantità di memoria utilizzata in base alle necessità in modo da poter facilitarne il riempimento/svuotamento.
Ne è richiesto l'utilizzo per poter conservare al suo interno il body della richiesta da inoltrare; in quanto l'inoltro del payload nella chiamata Vert.x [WebClient.request().sendBuffer(Buffer b)] richiede, appunto, un oggetto di tipo Buffer.

`io.vertx.core.Vertx` è necessaria per poter istanziare l'oggetto vertx, necessario per la creazione del WebClient.

java.util.ArrayList serve per poter caricare la lista delle eccezioni, non viene usata una mappa in quanto non sono presenti delle sulle quali effettuare dei match.

java.util.Map viene utilizzata per poter conservare tutti i target url e le match route, è necessaria una mappa in quanto la chiave serve per poter effettuare i match

```
package org.acme;

import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.event.Observes;
import javax.inject.Inject;

import io.vertx.core.buffer.Buffer;
import io.vertx.core.http.*;
import io.vertx.core.Vertx;
import io.vertx.ext.web.client.WebClient;
import io.vertx.ext.web.RoutingContext;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;
import io.quarkus.vertx.web.Route;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.regex.*;

import org.jboss.logging.Logger;
```

Figura 3.13: Librerie importate

tra le due mappe, successivamente sarà mostrata la logica.

`java.util.regex.*` ha il fine di poter effettuare il match tra alcune stringhe tramite regex, anche per questo utilizzo sarà più chiaro l'esempio più avanti.

`org.jboss.logging.Logger` si occupa di fornire i metodi necessari per i log a runtime.

Funzioni principali

void loadConfig()
All'interno di questa funzione vengono caricate tutte le liste che verranno successivamente usate per gli opportuni check e match al fine di reindirizzare correttamente

```
void loadConfig(){
    //Loading maps from application.properties
    targetURLs = config.targeturl();
    matchRoutes = config.match();
    exceptions = config.exceptions();

    LOG.info("Target URLs: " + targetURLs);
    LOG.info("Match routes: " + matchRoutes);
    LOG.info("Exceptions: " + exceptions);
}
```

Figura 3.14: Funzione loadConfig()

le chiamate, come mostrato di sotto.

URI findTargetURIByMatch(String uriToBeMatched)

```
URI findTargetURIByMatch(String uriToBeMatched) throws URISyntaxException {
    //Iterate over matches
    //LOG.info("Match entries: ");

    for(String match : exceptions)
        if(uriToBeMatched.endsWith(match)){
            LOG.info("Exception caught on " + uriToBeMatched);
            return new URI("http://10.0.0.15:9080" + uriToBeMatched);
        }

    for (Map.Entry<String, String> match : matchRoutes.entrySet()) {
        String matchValue = match.getValue();

        //Replacing "*" with ".*" for regex matching
        String regexMatch = matchValue;
        if(matchValue.contains("*"))
            regexMatch = matchValue.substring(0,matchValue.indexOf("*"))
            + "."
            + matchValue.substring(matchValue.indexOf("*"));

        //Validate uri by regex matching
```

```
        if(Pattern.matches(regexMatch, uriToBeMatched)){
            //Removing useless part from request uri to retrieve
            //only the meaningful remaining path
            String remainingPath = uriToBeMatched;
            if(matchValue.contains("*"))
                remainingPath = uriToBeMatched
                .replace(matchValue.replace("/*",""), "");
            String targetURI = targetURLs.get(match.getKey());
            LOG.info("matched" + uriToBeMatched + " - " + remainingPath);
            return new URI(targetURI+remainingPath);
        }
    }
    return null;
}
```

Questa funzione assume un ruolo cardine all'interno della logica del programma, essa acquisisce un parametro di input che corrisponde, come si può intuire dal nome, ad un URI, il quale dovrà passare una sequenza di controlli, per poter poi fornire in output quello che sarà l'indirizzo finale al quale reindirizzare la chiamata.

Vediamo di seguito i vari passaggi:

1. primo **for**: all'interno della prima iterazione andremo a verificare se l'URI presenta al suo interno (o meglio alla fine, considerando l'utilizzo del `.endsWith`) delle stringhe che permettono di catalogare a priori questo indirizzo come "eccezione"; qualora venga individuata una corrispondenza sarà ritornato un indirizzo standard già in questa fase, in modo da non proseguire con le prossime iterazioni.

2. secondo **for**: all'interno della seconda iterazione andiamo di fatto a verificare se l'indirizzo di input corrisponde a una delle righe per il match. Al fine di far combaciare le stringhe è necessaria una manipolazione per poter effettuare di conseguenza un match tramite regex, dopodichè si dovrà estrapolare il path dall'indirizzo iniziale per poi attaccarlo all'host finale.
   Un esempio per poter comprendere al meglio il passaggio:
   IndirizzoEntrante.com/remainingPath, che deve essere reindirizzato verso IndirzzoUscente.com/remainingPath. Bisogna, quindi, andare a prelevare tutto ciò che viene dopo lo "/" nell'indirizzo entrante (in questo caso remainingPath), andare a trovare IndirizzoUscente.com tramite il match, dopodichè ricostruire la stringa facendo "IndirizzoUscente.com/" + "remainingPath".
   Necessario menzionare l'istruzione:
   `String targetURI = targetURLs.get(match.getKey());`

che permette di prelevare il targetURI fornendo semplicemente la **chiave** corrispondente al match, che sarà ritrovata necessariamente dentro targetURLs grazie alla costruzione del file di configurazione (vedi figure a e **??**3.12).

void onStart() è una funzione che viene eseguita automaticamente all'avvio del programma, essa contiene semplicemente un richiamo a `loadConfig()` in modo da caricare immediatamente le liste dai file di configurazione.

void proxy(RoutingContext ctx)

Questa funzione contiene tutta la logica del proxy, di seguito verrà analizzato passo dopo passo il suo funzionamento.
Innanzitutto la funzione acquisisce come parametro un oggetto di tipo Routing-Context, il quale contiene tutte le informazioni del contesto, come la richiesta e la risposta associata, l'indirizzo, gli header, i cookies e tutti gli altri dati necessari alla connessione HTTP. Questo oggetto è fondamentale per poter orchestrare il reindirizzamento e l'impacchettamento delle informazioni, come vedremo successivamente.

```
[1]
@Route(path = "/*")
void proxy(RoutingContext ctx) throws URISyntaxException{
    [2]
    LOG.info("Request uri:" + ctx.request().uri());
    URI targetURI = findTargetURIByMatch(ctx.request().uri());
    //URI targetURI = new URI("http://localhost:8080" + ctx.request().uri());

    if(targetURI != null) {
        [3]
        WebClient myClient = WebClient.create(vertx);
        //Getting request information to be forwarded
        HttpMethod httpMethod = ctx.request().method();
        String requestCookies = ctx.request().getHeader("Cookie");
        Buffer requestBody;
        [4]
        if(ctx.request().getHeader("Content-Type") != null
            && ctx.request()
                .getHeader("Content-Type")
                .split("/")[0]
                .contentEquals("multipart")) {
            LOG.info("Multipart content-type");
            requestBody=
```

```
        Buffer.buffer(ctx.request().formAttributes().toString());
}
else
    requestBody = ctx.getBody();

[5]
LOG.info(ctx.request().formAttributes());

LOG.info("Request URI: " + ctx.request().absoluteURI() +
        " - Request Cookies: " + requestCookies +
        " - Request Body: " + requestBody +
        " - Request Headers: \n" + ctx.request().headers());

LOG.info("Target URI: " + targetURI);

[6]
myClient.request(
            httpMethod,
            targetURI.getPort(),
            targetURI.getHost(),
            targetURI.getRawPath() +
            (targetURI.getQuery() != null ? "?" +
            targetURI.getQuery() : "")
        )
        .putHeader("Cookie", requestCookies)
        .putHeader("Content-Type",
                    ctx.request().getHeader("Content-Type"))
        .sendBuffer(requestBody)
        .onSuccess(response -> {
            [7]
            List<String> responseCookies = response.cookies();
            LOG.info("Response cookies : " + responseCookies);
            LOG.info("Response status: " +
            response.statusCode() + " " + response.statusMessage());
            ctx.response()
                .setStatusMessage(response.statusMessage())
                .setStatusCode(response.statusCode())
                .putHeader("Set-Cookie", responseCookies)
                .putHeader("content-type",
                    response.getHeader("content-type"));
            if (response.body() != null)
```

```
                        ctx.response().end(response.body());
                    else
                        ctx.response().end();
                });
    }
    else{
        [8]
        LOG.info("No match found");
        ctx.response()
            .setStatusCode(404)
            .setStatusMessage("Not found")
            .end();
    }
}
```

Di seguito i vari passaggi:

[1] `@Route(path = "/*")`, questa annotazione è necessaria per Quarkus al fine di definire quale azione intraprendere nel momento in cui viene intercettata una chiamata HTTP con un determinato path. In questo caso viene intercettata qualsiasi chiamata (avendo impostato "/" come path), dal momento che dobbiamo catturare tutto il traffico in entrata al proxy e successivamente intraprendere delle azioni in base ad ogni singola richiesta.

[2] Ci troviamo all'interno della funzione, inizialmente viene settata la variabile targetURI, che sarà fondamentale per il funzionamento del proxy.
Essa contiene al suo interno la parte iniziale dell'indirizzo al quale reindirizzare la chiamata, la funzione invocata per trovarlo è stata ampiamente approfondita in precedenza.
Commentata si trova la riga di default per poter debuggare in locale.

[3] Se targetURL non è null, quindi è stato trovato un match all'interno della lista dei "matching paths", viene istanziato l'oggetto myClient di tipo WebClient, il quale richiesta l'istanza di vertx per poter funzionare. Da questo momento in poi potremo usare myClient per la creazione di un client che possa contattare un server a nostro piacimento.
Dopodiché viene catturato il metodo HTTP della richiesta corrente all'interno di httpMethod, andandolo a recuperare dall'oggetto ctx, passato al proxy come parametro, il quale contiene tutte le informazioni relative alla connessione attuale. In questo caso quindi viene richiamata la funzione ctx.request().method() che restituisce il metodo HTTP dalla request.
Stesso procedimento viene attuato per poter ottenere i cookies, sotto forma di stringa in questo caso.

Viene utilizzata la funzione `ctx.request().getHeader("Cookie")`, la quale va ad intercettare l'header "Cookie" e restituisce la stringa contenente il valore di tale header. Questa funzione è generica e può essere usata per ottenere qualsiasi header, il suo utilizzo è stato necessario in quanto la funzione apposita per ottenere i cookies messa a disposizione dall'oggetto HttpServerRequest rendeva la gestione meno fluida.
Successivamente viene dichiarata una variabile di tipo Buffer requestBody, che andrà a contenere il body della chiamata qualora sia presente. Viene dichiarata in questo punto solo per una questione di scope, in quanto verrà assegnata nel punto [4]. Questa variabili saranno necessarie per poter ricreare a nostra volta una chiamata con le stesse informazioni di quella ricevuta.

[4] Questo if è necessario in quanto la variabile requestBody deve essere assegnata tramite due modalità diverse se il tipo di content-type è "multipart" o meno. Questa escamotage è necessaria in quanto la libreria presenta un bug che rende la funzione `ctx.getBody()` non utilizzabile qualora la chiamata possieda un header ti tipo "content-type:multipart", non è chiaro a cosa sia dovuto, ma in seguito ad un confronto con Stuart Douglas [ingegnere di Red Hat specializzato in Quarkus], tramite posta elettronica, è emerso che l'unica casistica trattata diversamente dal framework dovrebbe essere quando la richiesta presenta un header di tipo `content-type:multipart`.
Di conseguenza bisgona fare distinzione tra le casistiche, content-type:

- multipart: in questo caso è possibile ottenere l'informazione del payload solo tramite l'utilizzo della funzione `ctx.request().formAttributes()`, la quale viene poi convertita in String tramite un `.toString()` e successivamente in Buffer, per motivi tecnici legati al funzionamento di determinate funzioni che utilizzeremo dopo.
- altro: per qualsiasi altro tipo di content-type è possibile usare la funzione del RoutingContext `ctx.getBody()` che permettere di catturare il body della request in un oggetto Buffer.

[5] In questa sezione vengono eseguiti dei LOG fondamentali per il debugging, nello specifico vengono stampate tutte le informazioni della request. È possibile trovare l'URI della request, i cookies al suo interno, il body (se presente) e i vari headers, oltre al targetURI e i form attributes, qualora la request abbia un header di tipo multipart.

[6] Qui è dove avviene il passaggio fondamentale del proxy. Come spiegato nei capitoli precedenti, per poter effettuare il reindirizzamento dei pacchetti HTTP verso gli indirizzi corretti è necessario creare (per poi distruggere) un client che si possa interfacciare con il server "corretto" per ogni singola connessione.

Per fare ciò è stato utilizzato un WebClient, della libreria Vert.x, che permette di creare un client al fine di connettersi a un server dove inoltrare il pacchetto. In questa sezione utilizziamo la funzione `WebClient.request(HttpMethod method, int port, String hostname, String path)` che permette di inviare una request usando un determinato metodo HTTP, specificando host, porta e path. È possibile in seguito concatenare altre funzioni che permettono di specificare ulteriori header, nel nostro caso vengono aggiunti i cookies e il content-type. Proseguendo con il method chaining andiamo ad inserire il body tramite la funzione `.sendBuffer(Buffer b)`, il quale contenuto è stato "normalizzato" in precedenza.
Successivamente viene effettuata la chiamata, che viene gestita in modo asincronto tramite `.onSuccess()`.

[7] Dentro `onSuccess` viene eseguito il reinoltro della chiamata al client, la chiamata è asincrona, di conseguenza viene messa in coda ed entra in gioco l'event-loop, quindi se la risposta subisce dei rallentamenti verrà data precedenza ad altre connessioni parallele qualora i servizi richiesti siano a disposizione. La response viene gestita tramite lambda function all'interno del quale verrà invocato ctx.response().end() per poter inviare tutto all'utente finale.
Una volta ricevuta con successo la risposta dal microservizio corretto vengono conservati i cookies dentro la lista `responseCookies`. Dopodiché viene richiamato il metodo `ctx.response()` necessario per creare la response. Vengono copiate tutte le informazioni ricevute dal microservizio, tramite method chaining vengono inseriti statusMessage, statusCode, la lista di cookies e l'header content-type. Infine viene fatto un controllo sul body qualora sia presente o meno, in quanto `ctx.response().end()` presenta due versioni, una senza alcun parametro e una con un parametro che rappresenta il body. Per questo motivo viene richiamata `ctx.response().end(response.body())` qualora sia presente un body nella response, mentre `ctx.response().end()` in caso contrario. L'utente finale riceverà così la sua response invariata, senza poter notare alcuna differenza rispetto a una connessione diretta con il server reale, solo che in questo modo è possibile controllare tutto il traffico in ingresso ed uscita. A questo punto il WebClient viene terminato e il garbage collector penserà a liberare la memoria dedicata alla sua esecuzione.

[8] All'interno di questo else, che viene invocato qualora non venga trovata alcuna corrispondenza all'interno della lista degli indirizzi autorizzati, viene semplicemente inviata come response all'utente un `Error 404: Not Found`, questo caso può sembrare banale, ma è necessario oltre che per motivi funzionali anche per questioni di sicurezza, in modo da non consentire il passaggio di chiamate non autorizzate. Tali meccanismi di sicurezza possono essere estesi in futuro per permettere una maggiore affidabilità del sistema.

Capitolo 4

Testing e valutazione delle prestazioni

All'interno di questo capitolo verrà approfondita la parte di testing e analisi delle performance del reverse proxy.

Per ricapitolare, il BFF in uso al momento utilizza un server GlassFish progettato nel 2016, per quanto non sia considerabile obsoleto presenta comunque alcuni rallentamenti, nonostante funzioni ancora con delle prestazioni accettabili è necessario far fronte ad alcuni problemi che sono sorti nel corso di questi anni.

Primo tra tutti è l'approccio di IO bloccante adoperato nei thread per gestire le connessioni, una caratteristica molto comune nei sistemi informatici in generale, che si tratta di un metodo per poter gestire in modo ordinato gli input e gli output. Sostanzialmente quando il thread è in attesa di un input/output si blocca per evitare che l'esecuzione degli stessi si intrecci causando problemi a livelli di flusso. Questa soluzione è abbastanza antiquata e poco performante, in quanto porta a bloccare tutti i thread in coda anche qualora essi non abbiano necessità di input o output, oppure anche se la risorsa dalla quale devono effettuare I/O è disponibile.

Nel nostro caso questo problema si riscontra all'interno del BFF, sostanzialmente quello che succede è che le richieste ricevute vengono assegnate ad un thread, tale thread verrà messo in coda in attesa di venire eseguito.
Quando è il suo momento ed esso presenta necessità di IO da parte di un particolare microservizio, la richiesta verrà inoltrata e tutta la coda sarà messa in attesa fin quando il thread in idle riceverà la sua risposta. In questo tempo gli altri thread richiedenti risorse da altri microservizi, che quindi sarebbero potuti essere serviti, resterebbero comunque in attesa.

Il nostro obbiettivo è quello di sopperire a questo problema utilizzando un sistema interno a Quarkus che permette di parallelizzare i thread qualora sia possibile.
Di conseguenza il testing sarà incentrato sul verificare se il sistema di parallelizzazione funziona correttamente.

Il modulo BFF dovrà di conseguenza essere in grado di effettuare un load balancing nei confronti dei microservizi a valle.
Per load balancing si intende una tecnica utilizzata nell'ambito delle reti informatiche per distribuire in modo equo e efficiente il carico di lavoro tra più risorse (come server, dispositivi di rete o servizi) al fine di migliorare le prestazioni, l'affidabilità e l'efficienza del sistema. L'obiettivo principale del load balancing è garantire che tutte le risorse coinvolte siano utilizzate in modo equo ed evitare situazioni in cui una risorsa sia sovraccaricata mentre altre sono sottoutilizzate.
In un contesto server, il load balancing coinvolge il distribuire le richieste dei client tra più server backend, nel nostro caso si tratta di microservizi. Questo viene fatto per garantire che ciascun server non sia sovraccaricato e che le richieste dei client vengano gestite in modo rapido ed efficiente.
Per fare in modo che i load balancer possano decidere su quale server instradare una richiesta è necessario che vengano utilizzati degli algoritmi di load balancing studiati apposta per poter ottimizzare il sistema.

4.1 Obbiettivi

L'obbiettivo della fase di testing è quello di verificare se sono presenti degli effettivi miglioramenti in seguito all'impiego della nuova soluzione.
Al fine di verificare se ciò è avvenuto è necessario effettuare gli stessi test di carico in entrambe le versioni e definire se gli indici sotto osservazione presentano delle differenze.
A fronte dell'esito dei test sarà successivamente valutato dall'azienda se adoperare o meno la nuova soluzione.
In caso di esiti non soddisfacenti è necessario individuare dove risieda il problema, per porre poi, insieme all'azienda, in valutazione eventuali variazioni infrastrutturali o di codice in base alla natura del difetto.

4.2 Metodologia e strumenti

La metodologia adoperata pone in risalto l'esperienza utente come parametro fondamentale da monitorare, di conseguenza come indici si prendono in considerazione principalmente throughput, latency e load balancing.

Sotto questo aspetto il worst case scenario è che l'utente debba paradossalmente subire un peggioramento nella propria esperienza, dovuto a tempi di risposta troppo elevati, che potrebbero dipendere da molteplici fattori, come un aumento del numero di controlli per ogni singola chiamata che il proxy si ritrova a gestire, ritardi non previsti dovuti alla comunicazione con il backend, un malfunzionamento dell'infrastruttura del framework, utilizzo erroneo degli strumenti messi a disposizione.

Saranno quindi osservati principalmente gli indici di latenza sotto stress, andando ad effettuare una serie di esami di carico su diversi microservizi in modo da verificare se l'event-loop funziona correttamente e se i tempi di risposta, in questi contesti, sono migliori rispetto al sistema attuale.

4.2.1 Indici

Per quanto riguarda gli indici da tenere in esame consideriamo principalmente:

- **Throughput**: serve a misurare le prestazioni di un sistema o di un processo, è utilizzato per misurare la quantità di dati che possono essere elaborati e trasferiti in un determinato lasso di tempo. In parole povere, il throughput rappresenta la capacità di gestione del flusso di dati di un sistema.

  Esso utilizza come unità di misura il byte al secondo (B/s) o bit al secondo (bps) nel caso delle reti informatiche. Ad esempio, nel caso delle connessioni internet, il throughput misura la quantità di dati che possono essere caricati o scaricati al secondo (in Mbps o Gbps), più è alto il throughput maggiore sarà la rapidità nel trasferire i dati e la banda a disposizione.
  Il throughput può essere influenzato da diversi fattori, tra cui:

  - **Larghezza di banda** a disposizione, ad esempio se un cavo permette di trasferire al massimo 20 Mbps, è inutile avere una connessione da 200 Mbps, in quanto il cavo fungerà da collo di bottiglia e il throughput sarà sempre limitato a 20 Mbps nel migliore dei casi.
  - **Congestione di rete**, in quanto in una rete particolarmente affollata è possibile che ci siano molti ritardi dovuti al fatto che i pacchetti devono attendere il loro turno per poter essere trasmessi.
  - La **qualità della connessione**, che può dipendere da diversi fattori, come i mezzi fisici a disposizione oppure la scarsa qualità dei sistemi in sé, si traduce in pacchetti persi e nella "perdita di tempo" per ritrasmetterli, di conseguenza il throughput complessivo si abbasserà.

  Prendendo in esempio una connessione Internet, il throughput misura la quantità di dati che possono essere scaricati o caricati al secondo (solitamente

in Mbps o Gbps). Maggiore è il throughput, maggiore è la capacità del collegamento Internet di trasferire dati in modo rapido.
È importante misurare e ottimizzare il throughput in modo da garantire che un sistema o una rete possano soddisfare le esigenze di prestazioni previste.

- **Latency**: Per latenza ci si riferisce al ritardo o al tempo impiegato per trasmettere dati/informazioni da un punto all'altro attraverso la rete. È spesso misurata in millisecondi (ms) ed è uno degli indicatori chiave delle prestazioni di una rete, se non il più importante. La latenza è una componente fondamentale per valutare la qualità di una connessione, nonchè il fattore principale ad influenzare l'esperienza utente.
  Per questo motivo sarà l'indice preso in esame nei nostri casi di test.

  Ci sono tre tipi principali di latenza nel contesto delle reti:

  - **Latenza di propagazione**: dovuta alla fisica della trasmissione dei dati attraverso i mezzi di comunicazione. Essa dipende dalla distanza tra i punti di origine e destinazione, nonché dalla velocità di propagazione del mezzo (elettricità in un cavo di rame oppure luce all'interno di una fibra ottica).
    Essa non può essere eliminata o ridotta se non agendo su questi due fattori.
  - **Latenza di elaborazione**: causata dai dispositivi di rete, come i router che devono processare e instradare i pacchetti seguendo precisi algoritmi di scheduling e routing che possono essere più o meno efficienti in base al contesto di applicazione.
    All'interno del calcolo complessivo della latenza viene incluso il tempo impiegato per leggere l'intestazione del pacchetto, determinare la destinazione corretta e prendere in seguito le decisioni sul routing. Essa può variare in base alla complessità dei dispositivi coinvolti e alla congestione presente nella rete.
  - **Latenza di accodamento**: è dovuta alla coda di pacchetti in attesa di essere trasmessi attraverso un dispositivo di rete. Questo avviene quando una coda è congestionata e i pacchetti devono attendere il loro turno per poter essere instradati, causando una latenza aggiuntiva che non dipende dalle singole connessioni bensì dalla congestione sui dispositivi di routing, che, trovandosi in punti trafficati della rete, fungono da collo di bottiglia causando notevoli rallentamenti. Per poter ridurre questa latenza è necessario implementare efficienti algoritmi di scheduling al fine di gestire al meglio le code.

La latenza complessiva di una connessione di rete può essere considerata come la somma si questi tre tipi di latenza. Minimizzarla è necessario per garantire un'esperienza utente ottimale e una comunicazione efficiente. Specialmente per alcuni tipi di applicazioni, come lo streaming, mantenere questi fattori ai minimi termini è fondamentale, anche per questo motivo la ricerca in questo ambito è molto florido.

4.2.2 Strumenti

Come strumento di testing è stato adoperato Apache JMeter, esso è un software open-source distribuito da Apache che permette di effettuare test sulle prestazione e test di carico per le applicazioni web, servizi, server e vari tipi di risorse di rete.

Figura 4.1: Apache JMeter Logo

Esso può essere utilizzato per effettuare:

- **Test delle prestazioni**: consente di simulare il carico di utenti su un servizio web al fine di valutarne le prestazioni, come nel nostro caso. È possibile misurare tempi di risposta, latenza, throughput e altri parametri per capire come l'applicazione si comporta in situazioni di carico variabili.

- **Test di carico**: ossia testare quanto carico un'applicazione può gestire prima che le prestazioni inizino a degradare. È possibile aumentare gradualmente il carico su un'applicazione e identificare il punto in cui raggiunge i suoi limiti.

- **Test di stress**: testare le applicazioni per verificarne la stabilità in condizioni di stress estreme. Questo può includere il superamento dei limiti di carico e la verifica di come l'applicazione gestisce le situazioni di emergenza.

- **Test di scalabilità**: può anche essere utilizzato per determinare la scalabilità verticale o orizzontale di un'applicazione all'aumento del carico. Utile per progettare e configurare sistemi in modo che possano espandersi in modo affidabile.

4.3 Test e analisi risultati

Il test è stato sviluppato al fine di verificare gli indici di latency e throughput del proxy, considerando il problema in esame è necessario strutturare il test in diverse fasi, per simulare un carico di utenti che possa mettere sotto stress il sistema di parallelizzazione è necessario salire sopra i 50 thread simultanei, ovviamente questa quantità varia in base alla macchina su cui viene eseguito il proxy.
Qualora tutte le request ricevano una response in un lasso di tempo inferiore ai 50 ms il test verrà considerato accettabile.

4.3.1 Progettazione

Il test principale viene effettuato in locale su un repository di xampp con questa struttura: Dove sono presenti due path principali:

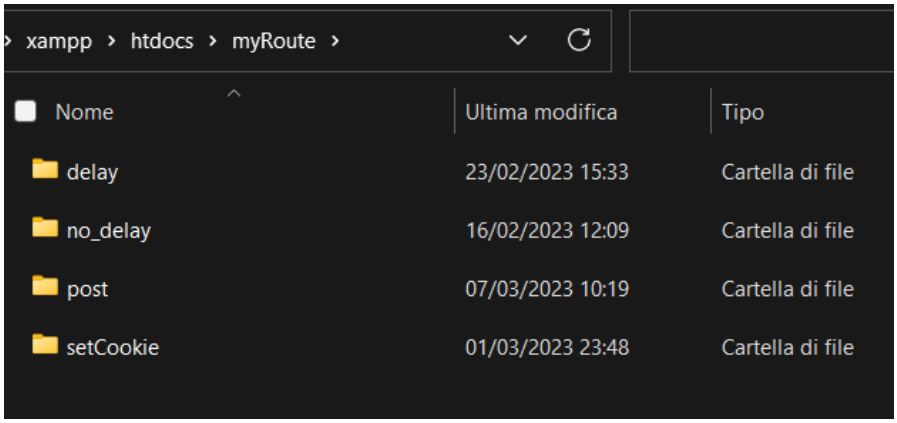

Figura 4.2: Repository xampp

- `myRoute/delay` all'interno del quale è presente un file index.php nel quale viene invocata un'istruzione di sleep per 10 secondi, dopodichè viene richiamata una echo con all'interno un timestamp

  `delay/index.php`

```
<?php
    // current time
    echo "Before " , date('h:i:s') , "\n";
    // sleep for 10 seconds
    sleep(10);
    // wake up !
    echo "After 10 seconds" , date('h:i:s') , "\n";
```

```
?>
```

- `myRoute/no-delay` contiene una semplice echo con timestamp per poter testare se il servizio è raggiungibile in qualsiasi momento

```
no-delay/index.php

<?php
    echo "Now " , date('h:i:s') , "\n";
?>
```

Il server locale di xampp è aperto sulla porta 80, di conseguenza per poter accedere alla repository 4.2 è sufficiente fare una chiamata su localhost:80/myRoute.
Ovviamente nel nostro caso è necessario attivare il proxy server in Quarkus che sarà attivo sulla porta 8082, all'interno del codice verranno reindirizzati tutti i pacchetti sulla porta 80, grazie all'istruzione statica inserita all'interno del codice java per il testing in locale:

```
URI targetURI = new URI("http://localhost:80" + ctx.request().uri());
```

I test saranno di conseguenza eseguiti su `localhost:8082` per fare in modo che tutte le chiamate vengano intercettate dal nostro proxy al fine di poterle analizzare.

JMeter setup

Il test su JMeter sarà composto da un **thread group** che sarà necessario per simulare più utenti che effettuano chiamate simultaneamente.
In questo caso sarà impostato su 100 chiamate simultanee per simulare il caso ipotetico in cui 100 utenti richiamino dei servizi in contemporanea.

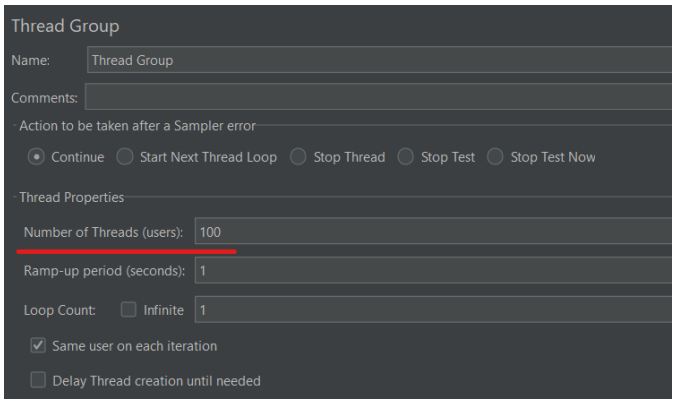

Figura 4.3: Thread group on JMeter

Successivamente vengono appese al Thread group due HTTP Request, una che farà delle chiamate su myRoute/no-delay e una su myRoute/delay.
Entrambe saranno delle chiamate GET che avranno come indirizzo da raggiungere localhost:8082, ma un path diverso.
Questo viene fatto per intrecciare le chiamate su entrambe le route al fine di verificare se la risorsa `no-delay` (fig. 4.4) viene richiamata senza ritardi dovuti all'attesa delle chiamate su `delay` (fig. 4.5).

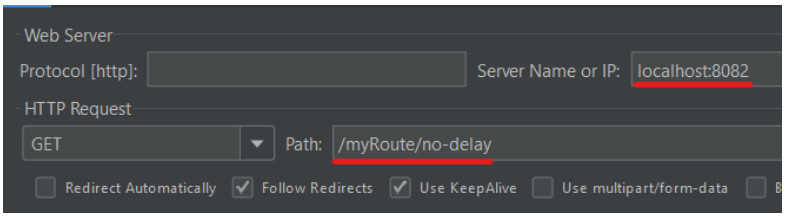

Figura 4.4: JMeter no-delay GET request

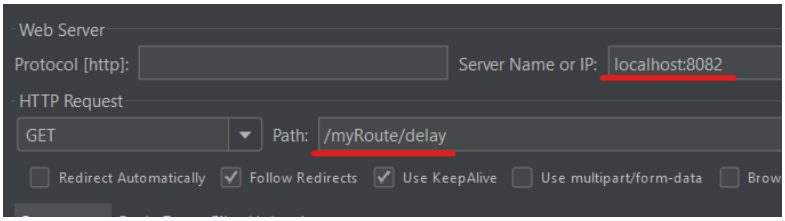

Figura 4.5: JMeter delay GET request

Infine sono presenti due listener per intercettare le response al fine di poter analizzare i risultati. La prima evidenziata in figura 4.6 è una tabella, necessaria per verificare tutti i parametri che ci interessano, mentre il secondo è un albero che permette di visionare l'interno dei pacchetti HTTP qualora la response contenga informazioni utili.

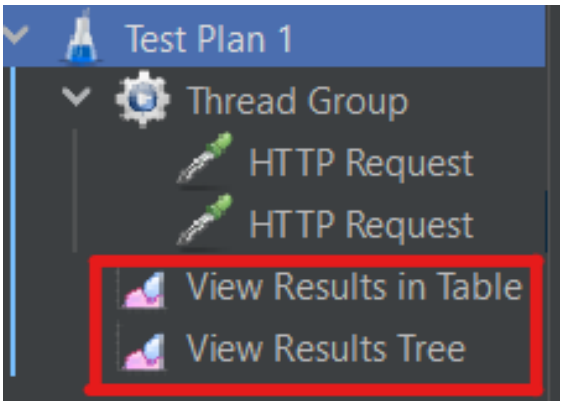

Figura 4.6: JMeter listeners

4.3.2 Risultati e analisi

I risultati sono molto promettenti, in quanto il sistema si mostra in grado di reggere il carico di test dando risposte paragonabili a quelli che si otterrebbero senza passare dal proxy.

Nel test passando per il proxy 4.7, sulla colonna Latency, si può notare un'alternanza tra una latenza corrispondente a circa 10 secondi e una di qualche millisecondo, questo comportamento è voluto, in quanto vengono effettuate delle chiamate in parallelo alla risorsa con delay di 10 secondi e alla risorsa senza delay, le chiamate con latenza di qualche millisecondo dimostrano come il sistema sia in grado di servire quelle richieste nonostante ci siano già dei thread in attesa dalla risorsa con delay.

Sample # ↑	Start Time	Latency	Thread Name
1	19:13:33.722	10026	Thread Group 1-1
2	19:13:43.748	11	Thread Group 1-1
3	19:13:33.746	10016	Thread Group 1-2
4	19:13:43.763	4	Thread Group 1-2
5	19:13:33.777	10016	Thread Group 1-3
6	19:13:43.793	5	Thread Group 1-3
7	19:13:33.793	10016	Thread Group 1-4
8	19:13:43.810	3	Thread Group 1-4
9	19:13:33.808	10016	Thread Group 1-5
10	19:13:43.825	3	Thread Group 1-5
11	19:13:33.824	10015	Thread Group 1-6
12	19:13:43.839	3	Thread Group 1-6
13	19:13:33.854	10016	Thread Group 1-7
14	19:13:43.871	3	Thread Group 1-7
15	19:13:33.870	10016	Thread Group 1-8
16	19:13:43.886	3	Thread Group 1-8
17	19:13:33.885	10016	Thread Group 1-9
18	19:13:43.901	3	Thread Group 1-9
19	19:13:33.917	10015	Thread Group 1-10
20	19:13:43.932	3	Thread Group 1-10
21	19:13:33.933	10014	Thread Group 1-11
22	19:13:43.948	2	Thread Group 1-11
23	19:13:33.948	10015	Thread Group 1-12
24	19:13:43.963	3	Thread Group 1-12
25	19:13:33.962	10016	Thread Group 1-13

Figura 4.7: Test with proxy

Sample # ↑	Start Time	Latency	Thread Name
1	19:46:11.118	1	Thread Group 1-1
2	19:46:21.127	1	Thread Group 1-1
3	19:46:11.148	1	Thread Group 1-2
4	19:46:21.157	1	Thread Group 1-2
5	19:46:11.162	2	Thread Group 1-3
6	19:46:21.173	1	Thread Group 1-3
7	19:46:11.178	1	Thread Group 1-4
8	19:46:21.188	1	Thread Group 1-4
9	19:46:11.208	2	Thread Group 1-5
10	19:46:21.220	1	Thread Group 1-5
11	19:46:11.224	1	Thread Group 1-6
12	19:46:21.236	1	Thread Group 1-6
13	19:46:11.240	1	Thread Group 1-7
14	19:46:21.251	1	Thread Group 1-7
15	19:46:11.271	1	Thread Group 1-8
16	19:46:21.282	1	Thread Group 1-8
17	19:46:11.286	2	Thread Group 1-9
18	19:46:21.297	1	Thread Group 1-9
19	19:46:11.303	1	Thread Group 1-10
20	19:46:21.313	0	Thread Group 1-10
21	19:46:11.319	1	Thread Group 1-11
22	19:46:21.329	1	Thread Group 1-11
23	19:46:11.339	2	Thread Group 1-12
24	19:46:21.345	1	Thread Group 1-12
25	19:46:11.365	1	Thread Group 1-13

Figura 4.8: Test without proxy

Nel test senza proxy 4.8 si può notare un comportamento leggermente diverso sulle latenze, questo è dovuto al fatto che mentre nel caso del proxy, esso si impegna ad attendere la risposta completa prima di rimandarla all'utente, nel caso della connessione diretta con il server xampp si andrà a ricevere una risposta istantaneamente, seppur incompleta, e una volta terminata l'attesa per l'istruzione `sleep(10)` su `index.php` verrà rimandata la seconda risposta completa, JMeter prende in considerazione solo la prima risposta e per questo motivo mostra una Latenza analoga a quella senza delay.

Nei test vengono mostrati solo 25 campioni per questioni di spazio, ma i risultati sono i medesimi anche per i restanti 75.
Altri test analoghi sono stati svolti (inizialmente con un carico minore o facendo chiamate senza delay), ma mostrano risultati identici, per questo motivo non sono stati riportati insieme agli altri.

Analisi

I risultati sono molto promettenti in quanto il carico simulato è molto alto e per la natura del sistema arrivare a questi volumi è pressoché impossibile, visto il carico di utenza.
Ricordiamo che il sistema è progettato per gestire un'utenza specializzata, tra cui medici, infermieri e operatori sanitari in genere. Di conseguenza il bacino d'utenza si limita a questi attori, che rappresentano un numero limitato di richiesta al secondo. Nel nostro caso in esame sono stati testati 100 utenti che effettuano una richiesta esattamente nello stesso istante, caso picco limite che non può verificarsi nella realtà, mostrando comunque risultati ottimi.
Tali test sono stati effettuati anche negli ambienti di sviluppo interni dell'azienda dando i medesimi risultati.
Pertanto è possibile concludere che i risultati siano più che soddisfacenti, dal momento che mostrano quanto Quarkus sia performante e scalabile, lasciando nuovi spunti per progetti futuri, al fine di poter implementare nuove funzionalità e scalare il volume di utenza se sarà necessario. Si può asserire che il sistema risulta ben preparato a gestire situazioni di stress e permette una parallelizzazione ottimale delle connessioni.

Capitolo 5

Sviluppi futuri

Poiché il sistema risulta particolarmente performante e scalabile, viste le tecnologie utilizzate, è possibile dare adito a diverse idee per sviluppi futuri.
Tra le varie idee abbiamo:

- **Miglioramento generale delle prestazioni**: si vuole puntare a rendere il sistema sempre più performante andando a studiare più approfonditamente il funzionamento dell'architettura Quarkus per sfruttarne a pieno tutte le potenzialità.
- **Sicurezza**: sicuramente il campo più importante in cui espandersi, considerando la natura del reverse proxy stesso, elemento di rete adatto ad implementare funzionalità di sicurezza come filtraggio, autenticazione, WAF e molto altro.
- **Gestione del traffico**: migliorare la gestione del traffico è un aspetto importante per un reverse proxy. Questo potrebbe comportare l'implementazione di bilanciamento del carico avanzato, la distribuzione del traffico basata su criteri specifici e il controllo del flusso del traffico.

Miglioramento delle prestazioni

Al fine di migliorare le prestazioni è necessario studiare più approfonditamente i meccanismi sottostanti a Quarkus, nonché le librerie utilizzate. Essendo un progetto open-source, infatti, è possibile adattarlo al caso di applicazione specifico al fine di rendere il sistema più performante.
Questi framework spesso utilizzano sovrastrutture abbastanza complesse, per poter adattare e rendere accessibile l'utilizzo a diverse tipologie di applicazioni, che, se da un lato rendono il framework molto più generico e scalabile, dall'altro causano un overhead non funzionale per le nostre esigenze.
Avendo la possibilità di studiare più a fondo il suo funzionamento sarebbe possibile

individuare i punti nevralgici, necessari al nostro utilizzo, e isolarli, in modo tale da migliorare le performance del nostro sistema.

Sicurezza

Un'applicazione futura necessaria per il reverse proxy è sicuramente quella in ambito di sicurezza.
Il reverse proxy è spesso utilizzato per fornire anche funzioni di sicurezza grazie alla sua collocazione a livello architetturale, che rende possibile implementare svariate funzionalità, quali: filtraggio dei pacchetti, WAF, TLS, protezione da DDoS, controllo accessi e autenticazione.

La prima tra queste su cui concentrarsi sarebbe sicuramente l'autenticazione, in quanto, per com'è progettato il sistema al momento, l'unico punto in cui avviene risulta a valle e di conseguenza non è presente alcun controllo prima di accedere alla regione di back-end.
Si potrebbe ovviare a questo problema andando ad effettuare, in fase di login, dei check di autenticazione all'interno del BFF stesso, verso il modulo di login nel back-end, ed una volta ricevuto il token di autenticazione rimandarlo all'utente che potrà inserirlo successivamente in tutte le sue richieste future. Si potrà, quindi, sfruttare la cache del proxy per effettuare il riscontro del token al fine evitare ulteriori connessioni non necessarie.

Un altro sviluppo importante, vista la natura delle informazioni di carattere sanitario, sarebbe quello di implementare un sistema di filtraggio pacchetti, tramite l'utilizzo di un Web Application Firewall (WAF), poiché il reverse proxy si pone come unico punto di accesso alla porzione interna e più sensibile del sistema.
A tal proposito sarebbe necessario creare una white list (o black list) al fine di stabilire delle regole secondo le quali il traffico verrebbe filtrato o meno. Questo può avvenire attraverso diversi criteri, come: l'utilizzo di determinati header, la presenza di elementi sospetti all'interno del payload, la connessione su una porta piuttosto che un'altra, l'utilizzo di protocolli specifici.
Queste funzionalità offrirebbero degli ottimi spunti dai quali partire per poter creare un infrastruttura di sicurezza che renderebbe il sistema molto più robusto rispetto ad ora.

Gestione del traffico

Per quanto riguarda i miglioramenti sulla gestione del traffico sarebbe possibile implementare dei meccanismi di load balancing, al fine di permettere uno smistamento dei pacchetti tale da non sovraccaricare i microservizi.

Capitolo 6

Conclusioni

Per ricapitolare, quindi, questo progetto di tesi prende vita da un'esigenza aziendale di Healthy Reply di riprogettare il modulo BFF del sistema della Nuova Piattaforma Regionale di Integrazione, ossia una piattaforma che mette a disposizione diversi servizi per gli enti sanitari regionali e privati della Regione Lombardia.
Il problema nasce dall'esigenza di parallelizzare le richieste HTTP verso i vari moduli back-end qualora ci siano dei rallentamenti su alcuni servizi che possano essere bypassati.

Il progetto propone di andare a studiare una soluzione in Quarkus che permetta di sfruttare il suo event-loop per far fronte all'esigenza di smistare i vari thread di connessione, al fine di permetterne la risoluzione anche qualora ci siano dei rallentamenti su altre connessioni.
Per com'è progettato il BFF in fase iniziale lascia spazio ad alcune problematiche legate ad una mancata gestione di soddisfacimento delle richieste qualora siano indirizzate a risorse disponibili, mentre sono presenti delle connessioni in attesa di risposta.

Il flusso di lavoro è stato orientato, in una prima fase, ad uno studio dello stato dell'arte per quanto riguarda i sistemi di reverse proxy basati su Quarkus. In seguito ad una serie di ricerche è emerso che la soluzione fosse fattibile e che il framework potesse soddisfare le nostre esigenze.

A questo punto, nella fase di sviluppo, mi sono focalizzato su:

- creazione di un server locale che potesse gestire delle chiamate HTTP;
- creazione del progetto su IntelliJ. A tal proposito, il sito di Quarkus mette a disposizione un tool che permette il download di un progetto vuoto con le

dipendenze necessarie, in modo da poter mettere subito le mani sul codice senza badare alle configurazioni;

- ricerca di progetti simili da cui prendere spunto.

Dopodiché è cominciata la fase di sviluppo vero e proprio, andando ad affrontare le varie problematiche riscontrate di volta in volta.
Uno dei problemi principali da risolvere è stato quello di capire come attuare il meccanismo di reinoltro dei pacchetti, in quanto è stato necessario attingere da altre librerie per poter generare un Client che si interfacciasse con i vari Server nel back-end in base all'indirizzo URL presente nella richiesta.
È stato quindi possibile, una volta individuato l'URL ed estratte le informazioni necessarie da esso, indirizzare le chiamate nei confronti della risorsa corretta e riceverne la risposta da reinoltrare successivamente al client corretto. Ques'ultima parte è gestita da Quarkus stesso e non ha mostrato particolari problemi.
Un'altra sfida è stata quella legata alla presenza di alcuni tipi di header nei pacchetti HTTP che non permettevano l'utilizzo di determinate funzioni. Per poter risolvere questo problema è stato cruciale aver contattato un ingegnere di Red Hat che ha fornito il suo supporto.

Una volta ovviato a queste problematiche, una versione embrionale dell'applicazione era pronta e si è potuta iniziare la fase di debug.
È stato poi necessario gestire tutta la logica per l'utilizzo del file di configurazione necessario per l'utilizzo dell'applicazione nel contesto dell'azienda.

Terminata questa parte, si è passato alla fase di testing in locale andando a simulare quello che è il comportamento del sistema reale. I test hanno portato a dei risultati soddisfacenti, nonostante siano emersi dei bachi a livello di gestione dei cookie, che si sono dimostrati essere l'ultimo ostacolo per il completamento del progetto.

Come riportato nel capitolo precedente alcuni sviluppi futuri sono stati considerati, soprattutto per quanto riguarda l'ulteriore ottimizzazione delle performance e l'implementazione di alcuni meccanismi di sicurezza.

Si può pertanto concludere che l'obbiettivo inizialmente posto sia stato raggiunto con ottimi risultati, in quanto la latenza nel caso di servizi in stallo è diminuita e il sistema risulta molto più scalabile rispetto a prima.

Bibliografia

[1] *Domain names - concepts and facilities*. RFC 1034. Nov. 1987. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/info/rfc1034 (cit. a p. 28).

[2] *Domain names - implementation and specification*. RFC 1035. Nov. 1987. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/info/rfc1035 (cit. a p. 28).

[3] *Cos'è un DNS?* URL: https://aws.amazon.com/it/route53/what-is-dns/ (cit. a p. 29).

[4] Margaret Rouse. *Proxy server*. Ott. 2012. URL: https://www.techopedia.com/definition/4200/proxy-server (cit. alle pp. 30, 31).

[5] *What is a reverse proxy?* URL: https://www.cloudflare.com/it-it/learning/cdn/glossary/reverse-proxy/ (cit. a p. 32).

[6] Francesco Marchioni. *Hands-on Cloud-native Applications with Java and Quarkus: Build High Performance, Kubernetes-native Java Serverless Applications*. Packt Publishing Ltd, 2019 (cit. a p. 40).

[7] *Quarkus*. URL: https://quarkus.io/about/ (cit. a p. 43).

[8] *Eclipse Vert.x and reactive in just a few words*. URL: https://vertx.io/introduction-to-vertx-and-reactive/ (cit. alle pp. 45, 46).

[9] *Regione Lombardia - NPRI*. URL: https://www.siss.regione.lombardia.it/wps/portal/site/siss/servizi-per-il-territorio/piattaforma-regionale-di-integrazione (cit. a p. 50).

[10] *What is JVM?* URL: https://www.w3schools.in/java/java-virtual-machine?utm_content=cmp-true (cit. a p. 54).

[11] *Container first*. URL: https://quarkus.io/container-first/ (cit. alle pp. 55, 56, 58).

[12] Jonas Bonér, Dave Farley, Roland Kuhn e Martin Thompson. *The Reactive Manifesto*. Set. 2014. URL: https://www.reactivemanifesto.org/ (cit. a p. 58).

[13] *Container first*. URL: https://quarkus.io/guides/quarkus-reactive-architecture (cit. alle pp. 60, 61).

[14] *Configure your application*. URL: https://code.quarkus.io (cit. a p. 63).

[15] *Using reactive routes*. URL: https://quarkus.io/guides/reactive-routes (cit. a p. 64).

[16] *Vert.x Web Client*. URL: https://vertx.io/docs/vertx-web-client/java/ (cit. a p. 66).

[17] *Routing Context*. URL: https://vertx.io/docs/apidocs/io/vertx/ext/web/RoutingContext.html (cit. a p. 67).

[18] *Annotation Type ApplicationScoped*. URL: https://docs.oracle.com/javaee/6/api/javax/enterprise/context/ApplicationScoped.html (cit. a p. 68).

[19] *Annotation Type Observes*. URL: https://docs.oracle.com/javaee/6/api/javax/enterprise/event/Observes.html (cit. a p. 68).

[20] *Annotation Type Inject*. URL: https://docs.oracle.com/javaee/6/api/javax/inject/Inject.html (cit. a p. 69).