UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"Architetture a Micro Servizi in ambito Industriale: un esempio d'uso"

Relatore: Prof. Mauro Migliardi

Laureando: Fabio Ruscica

ANNO ACCADEMICO 2021 – 2022

Data di laurea 20/07/2022

Indice

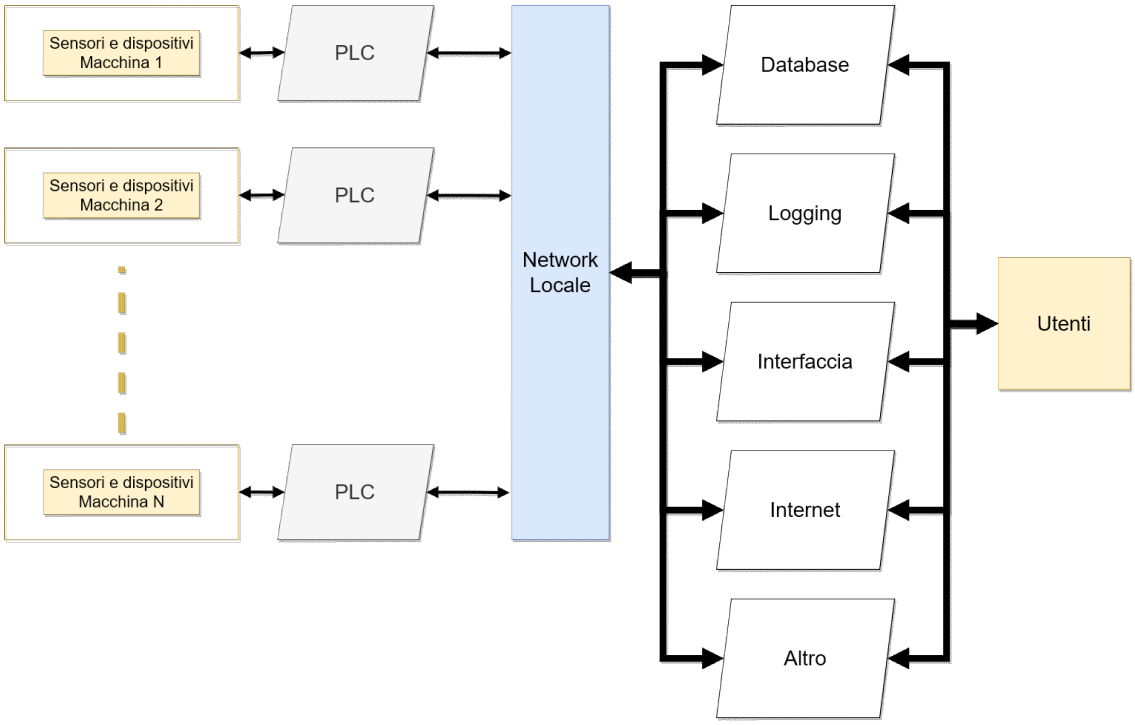
Capitolo 1

Introduzione

L'obiettivo di questa tesi è quello di analizzare la programmazione a microservizi e compararla alla programmazione tradizionale e all'utilizzo di macchine virtuali in ambienti industriali. Durante il tirocinio è stato possibile esplorare queste tecniche tramite due progetti.

1.1 Software industriale

Figura 1.1: *Struttura generica di un insieme di macchine industriali.*



Il grafico esposto nella figura 1.1 mostra un'architettura generica nel mondo del software industriale. Dei "Programmable Logic Computer" gestiscono i componenti meccanici presenti nelle macchine e comunicano con il software tramite appositi protocolli di comunicazione. Usualmente la parte di software che fa da intermediario tra utente e PLC risiede in uno o più server locali. Il passo successivo e l'introduzione di macchine virtuali per isolare i programmi tra di loro e avere più controllo sull'ambiente dove il singolo software viene utilizzato. I microservizi accoppiati a container vanno a migliorare parti critiche di questi sistemi fornendo ulteriori vantaggi rispetto alle macchine virtuali. Prima di introdurre tali vantaggi è importante definire due concetti:

- Scalabilità [1] - Un software sviluppato per fornire un servizio deve essere in grado di funzionare per una durata di tempo indeterminata e per un numero variabile di consumatori. L'esempio più comune è quello del motore di ricerca, che deve essere in grado di gestire un numero sempre più elevato di utenti e indicizzare un numero in costante crescita di pagine web. Tipicamente la scalabilità del software viene divisa in due categorie:

  - Scalabilità verticale - Ovvero l'aumento delle risorse hardware utilizzabili dal software su un singolo dispositivo. L'operazione permette di contenere il software nel minor numero di dispositivi possibili però può risultare molto costoso. Il software è generalmente monolitico e ogni dispositivo ne contiene una copia.

    Vantaggi

    - L'incremento della potenza di calcolo non altera il software, di conseguenza può risultare più facile da implementare e gestire.
    - I dati elaborati dal software si trovano principalmente sullo stesso dispositivo, facilitando la condivisione di informazioni in eventuali sotto processi.
    - Meno risorse impiegate per mantenere il singolo sistema anziché più dispositivi.

    Svantaggi

    - La quantità limitata di dispositivi impiegati risulta pericolosa qualora sia necessario effettuare un aggiornamento o ci sia un malfunzionamento.
    - Esiste un limite economico e fisico oltre il quale non è possibile aumentare in modo notevole la potenza di calcolo.
    - La rottura dell'hardware può avere effetti catastrofici per il servizio fornito dal software.

- Scalabilità orizzontale - Ovvero l'utilizzo di tecniche di load balancing che suddividono il carico di lavoro su più dispositivi con lo stesso livello di potenza hardware. I dispositivi vengono attivati o disattivati in base alle necessità.

  Vantaggi

  - Garantisce agli utenti prestazioni costanti, indipendentemente dalla quantità di richieste che il servizio sta gestendo (entro certi limiti fisici e/o economici).
  - La presenza di tanti dispositivi dove risiede il software, suddiviso in sottoparti, fornisce un'alta tolleranza ai guasti.

  Svantaggi

  - Il software deve essere progettato tenendo in considerazione la sua implementazione.
  - Spesa costante (Overhead) introdotta dall'utilizzo di componenti network per la comunicazione tra tutti i dispositivi che offrono il servizio.

- Manutenibilità - Qualsiasi tipologia di software deve, prima o poi, essere sottoposto a test e aggiornamenti per risolvere o rilevare problemi all'interno del proprio codice. Un software con una buona manutenibilità permette agli sviluppatori di aggiornare il codice senza dover incorrere in problemi come la regressione o l'introduzione di bug.

  - Manutenzione correttiva - Il tipo più comune, ovvero l'aggiornamento del software per ovviare ad un problema o errore scoperto durante l'esecuzione di esso.
  - Manutenzione adattiva - Consiste nell'aggiornamento del software introducendo o rimuovendo funzionalità in base alle necessità dell'utenza.
  - Manutenzione perfettiva - Consiste nel miglioramento del software già presente per ottimizzare operazioni o supportare aggiornamenti futuri.
  - Manutenzione preventiva - Consiste nel prevenire problemi che al momento non esisto ma che in futuro potrebbero presentarsi con conseguenze più o meno dannose.

I vantaggi introdotti dai microservizi sono:

- Scalabilità orizzontale, Manutenibilità - Per definizione l'architettura a microservizi promuove questi concetti suddividendo il software in sotto parti più facilmente gestibili, questo è vero anche per le macchine virtuali, però in questo caso si ha una granularità maggiore. In una macchina virtuale viene generalmente inserito l'intero programma mentre un microservizio rappresenta una parte atomica, cioè indivisibile, di un programma più grande.

- Sicurezza - La comunicazione tra microservizi o altri ambienti avviene solamente tramite rigide interfacce definite dagli sviluppatori, questo riduce la quantità di errori e problemi che potrebbero verificarsi e facilità la risoluzione di quelli che vengono scoperti;

- Utilizzo di linguaggi di programmazione differenti - Parti diverse di un programma traggono determinati vantaggi con particolari linguaggi: il Frontend è generalmente associato con linguaggi come CSS, JavaScript, HTML; mentre nelle parti che necessitano particolare attenzione alle prestazioni è preferito sviluppare con linguaggi come C o C++. La flessibilità dei microservizi permette agli sviluppatori di utilizzare vari linguaggi senza complicare la struttura del codice. In un software monolitico questo invece non è garantito.

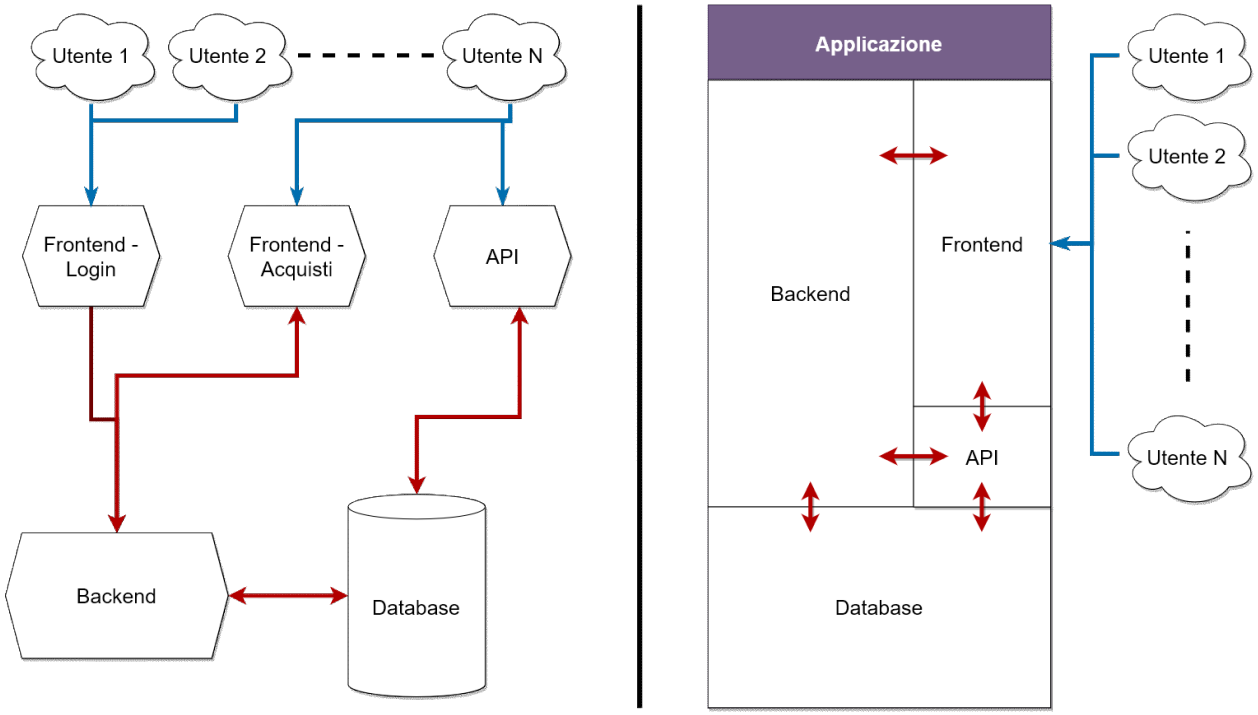
- Dimensioni - Un software monolitico può comportare dimensioni notevoli non solo per il programma in sè ma anche per tutte le dipendenze che esso potrebbe avere. I microservizi invece risultano essere autonomi e contenuti riducendo al minimo le dipendenze necessarie e quindi lo spazio occupato.

1.2 Architettura a microservizi

Introduzione

La progettazione a microservizi[9] è una filosofia di design che impone delle regole per lo sviluppo di applicazioni, principalmente destinate a offrire un servizio ad un'utenza di dimensione non definita. Questa filosofia esclude l'idea di un'applicazione monolitica, infatti, come implicato dal nome, il software viene suddiviso in parti più piccole ognuna delle quali ricopre un compito preciso.

Figura 1.2: *Esempio di una applicazione concettualizzata con filosofia monolitica (destra) e con filosofia a microservizi (sinistra).*



L'idea di distribuire un'architettura su più macchine nasce attorno agli anni ottanta. Il concetto di microservizio moderno non esisteva ancora, il termine fu coniato negli anni duemila, però l'idea di separare un'applicazione monolitica in sottoparti per semplificarne la gestione era già presente da diversi anni[4]. Il primo tentativo di separare il software su più sistemi fu attuato tramite il concetto delle "Remote Procedural Calls". L'obiettivo era quello di introdurre un'astrazione tale da permettere agli sviluppatori di poter invocare metodi remoti senza doversi preoccupare dell'infrastruttura necessaria, per ovviare ai problemi introdotti dal limite di memoria utilizzabile su un singolo dispositivo.

Questa soluzione risultò inefficiente per vari motivi, i principali:

- Il costo a livello di hardware e di prestazioni introdotto nell'invocazione di procedure presenti su un numero diverso di macchine risultava spesso oneroso anche per le operazioni più semplici, questo era soprattutto dovuto alla frequente frammentazione di software monolitico senza ulteriori modifiche, causando la dispersione (o duplicazione) di codice frequentemente utilizzato su più dispositivi;

- Introdurre nuovi metodi in questo sistema significava complicare ulteriormente l'interfaccia di comunicazione rendendo i punti d'ingresso di sistemi remoti estremamente verbosi. Una soluzione proposta per migliorare la situazione fu quella di introdurre il concetto di Façade: un design pattern che descrive l'utilizzo di un'interfaccia strutturata secondo rigide regole che prende il ruolo di intermediario nelle comunicazioni tra sistemi, centralizzando tutte le operazioni di comunicazione;

- La comunicazione era spesso sincrona: con l'aumentare dei numeri di dispositivi e le distanze che i dati dovevano attraversare, spesso il software doveva fermare tutte le sue operazioni per attenderne la ricezione.

Questo tipo di approccio venne eventualmente abbandonato in favore di una nuova filosofia di design agli inizi degli anni duemila. Con l'introduzione della "Service Oriented Architecture" (SOA) vennero introdotte linee guida più precise per la progettazione di software a servizi. In primo luogo i servizi, per essere tali, dovevano essere asincroni e comunicare tra di loro attraverso interfacce condivise. Come appena menzionato, la comunicazione sincrona tra microservizi introduce ritardi e blocchi del software con l'aumentare della distanza da percorrere e la quantità di dati da trasmettere tra ogni dispositivo. La mancanza di punti d'ingresso strutturati rendeva la gestione di essi estremamente complicata.

Due metodi, usati ancora oggi, per ovviare a quest'ultimo problema sono SOAP e REST. Entrambe le architetture si basano su messaggi standardizzati che viaggiano in rete. Ciononostante le due architetture non si escludono a vicenda.

- SOAP - Simple Object Access Protocol - SOAP è un protocollo con delle regole particolarmente rigide. Permette la comunicazione tra sistemi che condividono un network attraverso il protocollo HTTP utilizzando il formato XML per la rappresentazione delle informazioni;

- REST - REpresentational State Transfer - REST descrive l'architettura attraverso delle regole più flessibili, lasciando allo sviluppatore l'effettiva implementazione. Le risorse devono essere identificate da link URI (Uniform Resource Identifiers), e la loro rappresentazione deve essere distaccata dall'effettiva implementazione attraverso diversi formati, tra cui: XML, HTML, testo e JSON. In un secondo luogo client e server comunicano scambiandosi queste rappresentazioni attraverso protocolli standardizzati, HTTP non è un requisito di REST ma è comunque il protocollo più utilizzato.

Questo ci porta alla definizione attuale dell'architettura a microservizi: Un insieme di software indipendenti e autonomi che che ricoprono un ruolo ben definito, e tramite comunicazioni di rete standard collaborano per svolgere uno o più compiti.

Prendendo d'esempio un'applicazione monolitica: scalarla significa copiare interamente il programma ed eseguirlo su un nuovo server o istanza di una virtual machine. Un'architettura a microservizi permette di avere un controllo con granularità maggiore su quali parti del software ottengono più o meno risorse.

Comunicazione tra microservizi

Generalmente due o più microservizi comunicano tramite la rete (internet oppure locale). Uno degli svantaggi più importante che questo modo di progettare introduce è l'overhead dovuto all'elevato utilizzo di protocolli di rete e da tutto quello necessario a gestirli e utilizzarli. Lo sviluppatore deve tenere in considerazione non solo l'hardware necessario, ma anche tutte le complicazioni che una comunicazione tramite rete può introdurre.

Vengono definiti due tipi di protocolli di comunicazione tra microservizi:

- Protocollo sincrono - Le richieste effettuate dai client devono essere completate per permettere al programma di proseguire con il proprio compito. Un esempio di protocollo sincrono è il protocollo HTTP/S;
- Protocollo asincrono - Le richieste effettuate dai client non bloccano il programma e vengono processate il prima possibile. Un esempio di implementazione di protocollo asincrono è l'utilizzo di sistemi di intermediazione (Broker) oppure componenti built-in presenti nei sistemi operativi;

I protocolli sincroni o asincroni possono essere:

- Protocollo a singolo ricevitore - Il protocollo impone che una richiesta di un client venga processata da un singolo ricevitore;
- Protocollo a ricevitori multipli - Il protocollo permette a più ricevitori di processare la richiesta, operando in modo asincrono;

È importante ricordare che un'applicazione a microservizi non farà uso di un unico protocollo, bensì ne utilizzerà una combinazione in base alle necessità. È consigliato l'utilizzo di protocolli strettamente asincroni cosi da assecondare l'autonomia dei servizi.

La figura 1.2 mostra concettualmente come la stessa applicazione potrebbe essere sviluppata nei due modi. Le quattro separazioni principali di un'applicazione possono essere le seguenti, ognuna delle quali potrebbe essere ulteriormente scomposta in altrettante sottoparti:

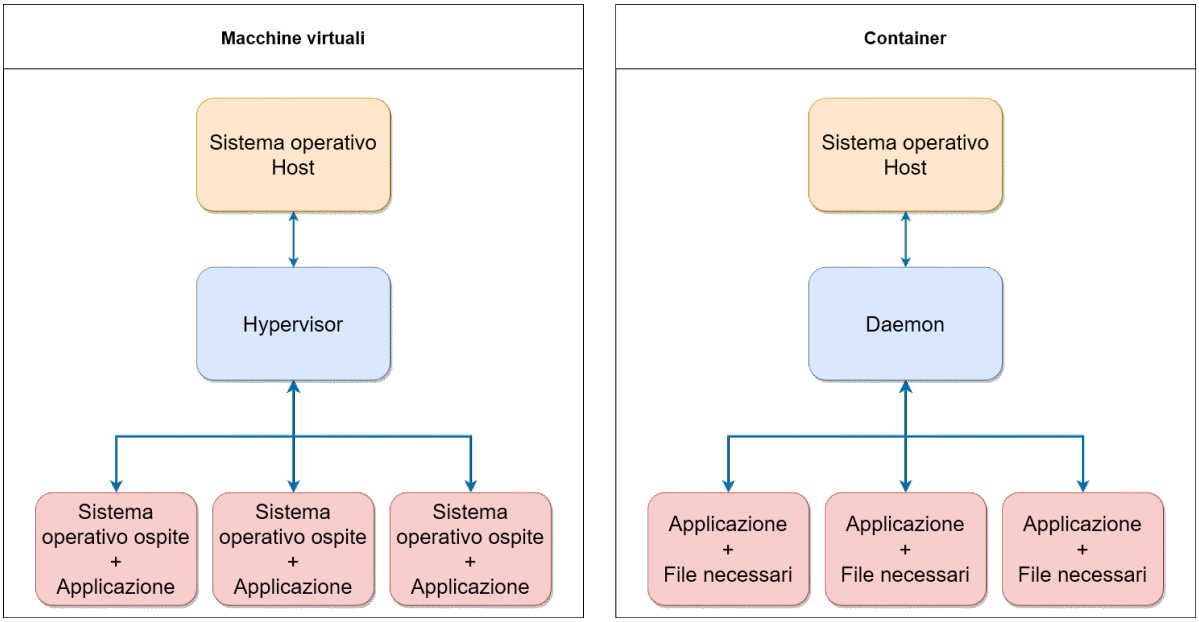
- Frontend - Servizio che permette all'utente finale di utilizzare il software;
- Backend - Servizio che gestisce tutte le operazioni interne dell'applicazione;
- Broker Service - Servizio che si occupa della comunicazione tra container e servizi astraendo i protocolli impiegati.
- Database access - Servizio che gestisce le operazioni di Read/Write sui database impiegati (MongoDB, PostgreSQL, ...).

Capitolo 2

Container e virtualizzazione

I container sono uno strumento molto potente che permettono di eseguire software in modo sicuro e isolato con il minimo spreco di risorse hardware. Per poter discutere di container è necessario conoscerne l'evoluzione e i motivi per cui sono stati adottati, i loro vantaggi e svantaggi.

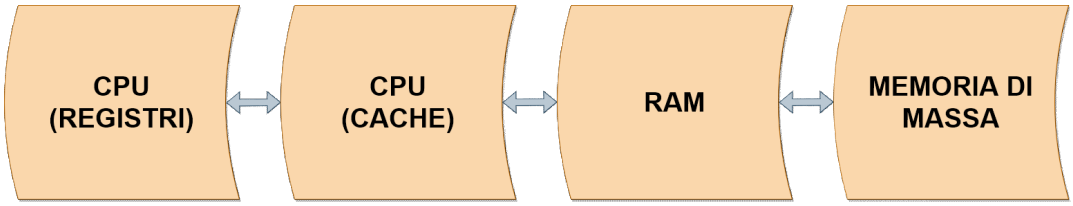
Figura 2.1: *Grafico semplificato Virtual Machine & Container*



2.1 Virtualizzazione

La nascita della virtualizzazione nei sistemi commerciali viene attribuita a IBM[2] negli anni sessanta con il primo sistema in grado di sfruttare il concetto di memoria virtuale automatica. Tale concetto nacque all'incirca dieci anni prima quando all'università di Manchester fu introdotto il cambio di pagine della memoria all'interno di un mainframe per spostare dati all'interno della gerarchia della memoria senza l'intervento di un operatore.

Figura 2.2: *Diagramma generico per memoria all'interno di un sistema informatico moderno. A partire da sinistra è presente la memoria più veloce all'interno del sistema. Ad ogni spostamento verso destra il tempo di latenza aumenta.*



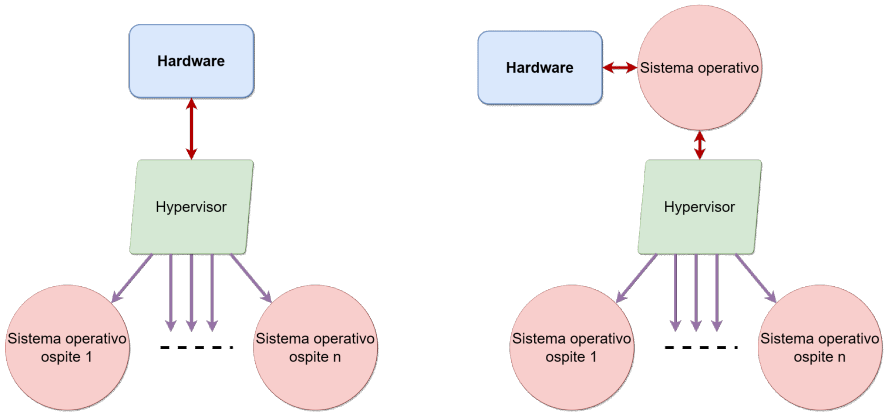
Questo fu un evoluzione dell'hardware e del software estremamente importante, infatti prima di questo evento era necessario effettuare l'operazione manualmente rendendo il dispositivo inutilizzabile per tutta la durata di essa. Negli anni successivi, il concetto di virtualizzazione si è evoluto ulteriormente introducendo la possibilità di eseguire molteplici sistemi operativi su sistemi a singolo processore, permettendo ad ogni sistema operativo di eseguire tutto il set di istruzioni del processore in modo isolato dal resto degli altri sistemi operativi. La necessità di virtualizzare i sistemi operativi è nata dal costo eccessivo dei sistemi informatici che rendeva impossibile l'idea di fornire ad ogni utente il proprio dispositivo, soprattutto, in ambienti lavorativi. Fu anche introdotto dell'hardware specializzato per permettere al software gestore delle virtual machine di effettuare il suo compito in un ambiente isolato e protetto con maggiore efficienza.

Quando il costo economico di questi sistemi iniziò a diminuire, tra gli anni ottanta e novanta, la necessità di tecniche di virtualizzazione diminuì anch'essa. Successivamente, con l'avvento di internet, la quantità di server e dispositivi informatici attivi ha continuato a crescere. Fin da subito fu determinato che nella maggior parte degli impieghi non venivano utilizzate tutte le risorse messe a disposizione dai server, e quindi l'utilizzo di un server per una una singola applicazione era spesso uno spreco.

Fu così che la necessità di consolidare più applicazioni in un unico dispositivo con ambienti protetti e isolati tra loro fece riemergere l'utilizzo di tecniche di virtualizzazione per migliorare le prestazioni e massimizzare l'utilizzo dell'hardware. Al giorno d'oggi, nel mondo industriale, con la necessità di monitorare un gran numero di dispositivi informatici e sensori le principali due tecniche utilizzate per ovviare allo spreco di hardware mantenendo un livello di sicurezza appropriato sono: le macchine virtuali e i container. Le prime eseguono un sistema operativo e tutte le applicazioni al suo interno in modo isolato dal resto del sistema ospitante su cui sono eseguite e altri eventuali sistemi operativi attivi con lo stesso metodo. I container, eseguono un compito simile, per implementazioni differenti, senza la necessità di eseguire un ulteriore sistema operativo, riducendo quindi le risorse necessarie.

2.2 Hypervisor & Container Daemon

Figura 2.3: *Diagramma concettuale per tipi di hypervisor. Bare metal hypervisor (sinistra) e Hosted hypervisor (destra).*



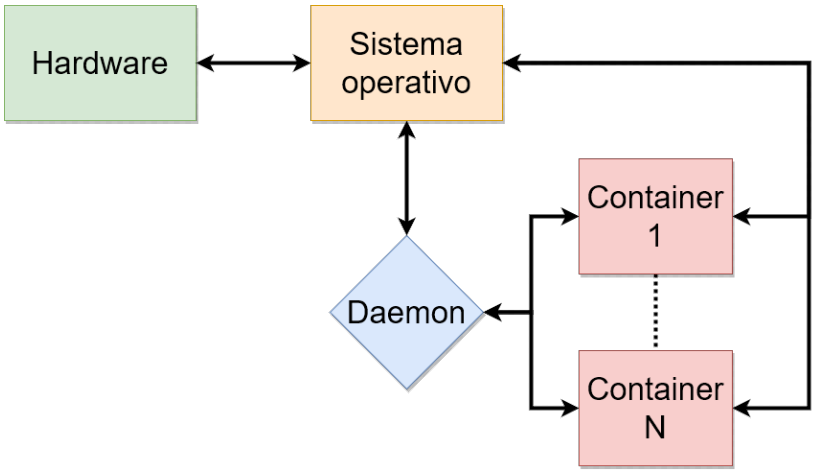
È importante conoscere cosa sono e le differenza tra l'Hypervisor e Daemon poiché spesso vengono erroneamente eguagliati.

Gli hypervisor hanno un ruolo estremamente critico nell'esecuzione di macchine virtuali. Essi fanno apparire ai sistemi operativi virtualizzati le risorse hardware come un unico insieme di risorse disponibili. Di conseguenza devono costantemente fare da interprete tra macchina virtuale e hardware per garantire l'accesso alle risorse limitando le collisioni tra operazioni. Sono definite due tipologie di hypervisor:

- Tipo 1 , anche detti '**Bare metal hypervisor**', vengono installati direttamente sull'hardware, non necessitano di un sistema operativo ospitante per la loro esecuzione. La mancanza del sistema operativo host presenta vantaggi dal lato delle prestazioni, però risulta abbastanza complesso da impostare e quindi non adatto a progetti "piccoli" dove la quantità di virtual machine non è tale da giustificare l'uso di questo hypervisor.

- Tipo 2 , anche detti '**Hosted hypervisor**', vengono installati su un sistema operativo esistente come un generico programma. Un lato positivo può essere la facile distribuzione dei driver necessari per le varie VM che vengono recuperati direttamente dal sistema operativo host. Però questo potrebbe non essere sufficiente a giustificare le perdite di prestazione dovuta alla presenza del sistema operativo principale che utilizza una quantità di risorse non irrisoria.

Figura 2.4: *Diagramma concettuale delle relazioni tra container e macchina*

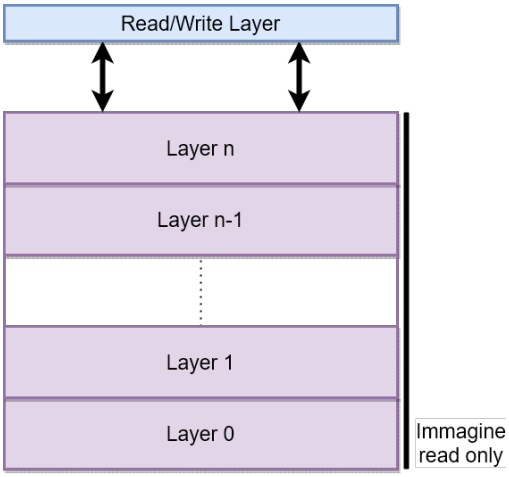


I daemon invece agiscono sulle risorse e i container in modo totalmente diverso. Alla base di ogni sistema operativo risidede il Kernel, il daemon sfrutta operazioni messe a disposizione da esso per isolare ed eseguire i container, in particolare i namespace ovvero identificatori che permettono di distinguere insiemi di risorse. Al momento dell'esecuzione il daemon fornisce al singolo container le informazioni necessarie a identificare le proprie risorse, da quel momento fino alla fine della vita del container esso potrà interagire solamente con tali risorse attraverso il sistema operativo host e nessun altro container sarà in grado di manipolare tali risorse senza averne il permesso. L'intervento del daemon è, quindi, necessario solamente nelle fasi di avvio e terminazione dei container per allocare o liberare porzioni di risorse. Per questo motivo, la figura 2.1 potrebbe trarre in inganno, facendo pensare che il daemon e l' hypervisor svolgano un compito simile, ma in realtà i container, durante normale esecuzione, sono liberi di eseguire il proprio codice nella loro parte di hardware/software isolata dal resto senza ulteriori astrazioni, un diagramma più accurato potrebbe essere quello presente nella figura 2.4.

2.3 Immagini & Container: Cosa sono?

I container e le immagini sono un evoluzione della virtualizzazione che fin da subito si differenziano dalle macchine virtuali per via della loro flessibilità. Le immagini sono oggetti immutabili composti da un insieme di strati (*layers*) che contengono dati (immagini, file, codice, ecc...) e metadati. I metadati contengono le istruzioni necessarie per generare ed eseguire i container e ulteriori informazioni identificative. La sequenza di questi strati e i loro metadati identificano l'immagine.

Figura 2.5: *Composizione di un Container*



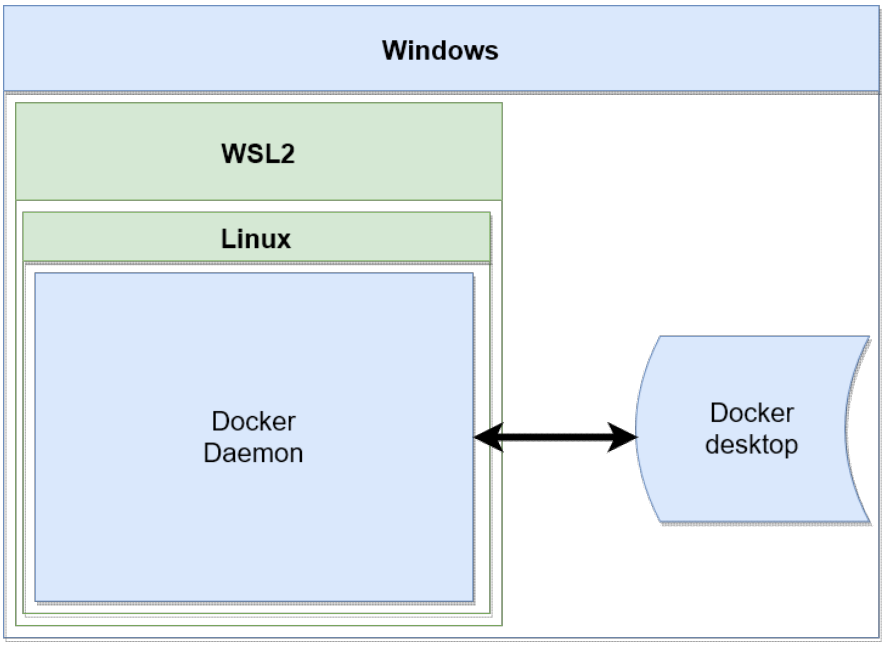
Al momento dell'esecuzione di un container viene generato un nuovo layer sopra a quelli forniti dall'immagine che permette la lettura e scrittura e conserva ogni cambiamento effettuato durante l'esecuzione[8]. Di conseguenza l'immagine non viene copiata per ogni container, bensì ognuno di essi comporta solamente la creazione di un nuovo strato di lettura/scrittura indipendente dagli altri. Docker non introduce particolari cambiamenti al concetto di Container generico appena introdotto. Esso prende parte all'iniziativa Open Containers[3] che definisce come container e immagini devono essere strutturate permettendo a più daemon di eseguire il container senza dover ricostruire l'immagine.

Come accennato in precedenza i container vengono isolati tramite namespace, si potrebbe vedere questo come un livello inferiore di sicurezza rispetto alle macchine virtuali tradizionali, ma nonostante questo, esso introduce grandi miglioramenti dal lato prestazioni e occupazione di spazio. Eseguendo molteplici Container isolati tra loro attraverso un unico sistema operativo si riduce l'overhead generale e la quantità di hardware necessario.

Una distinzione importante è la differenza tra Docker e Docker desktop:

- Docker Desktop - Applicazione necessaria per Windows e MacOS che permette di eseguire il Docker daemon all'interno di una distribuzione Linux e di conseguenza abilita l'esecuzione di container e la creazione di immagini sui rispettivi sistemi operativi. La distribuzione Linux eseguita risulta comunque essere occupare meno spazio rispetto ad una installazione completa di una macchina virtuale Windows.
- Docker - Rappresenta il daemon e tutto l'insieme di comandi e strumenti che permettono di interagire con esso.

Figura 2.6: *Rappresentazione generica della relazione tra Docker Desktop e Docker daemon*



Capitolo 3

Perchè Container?

3.1 Container & Virtual Machine

Generalmente le dimensioni di un container sono dell'ordine dei megabyte[6], proprio perché i file contenuti nell'immagine sono quelli strettamente necessari all'esecuzione del programma. Questa dimensione, inoltre, diminuisce ulteriormente se si sviluppa il software attraverso un'architettura a microservizi. Le virtual machine invece hanno dimensioni dell'ordine dei gigabyte oltre al programma devono portare con se tutti i file necessari al sistema operativo per funzionare. Nonostante questa differenza, le virtual machine ricoprono ancora un ruolo molto importante.

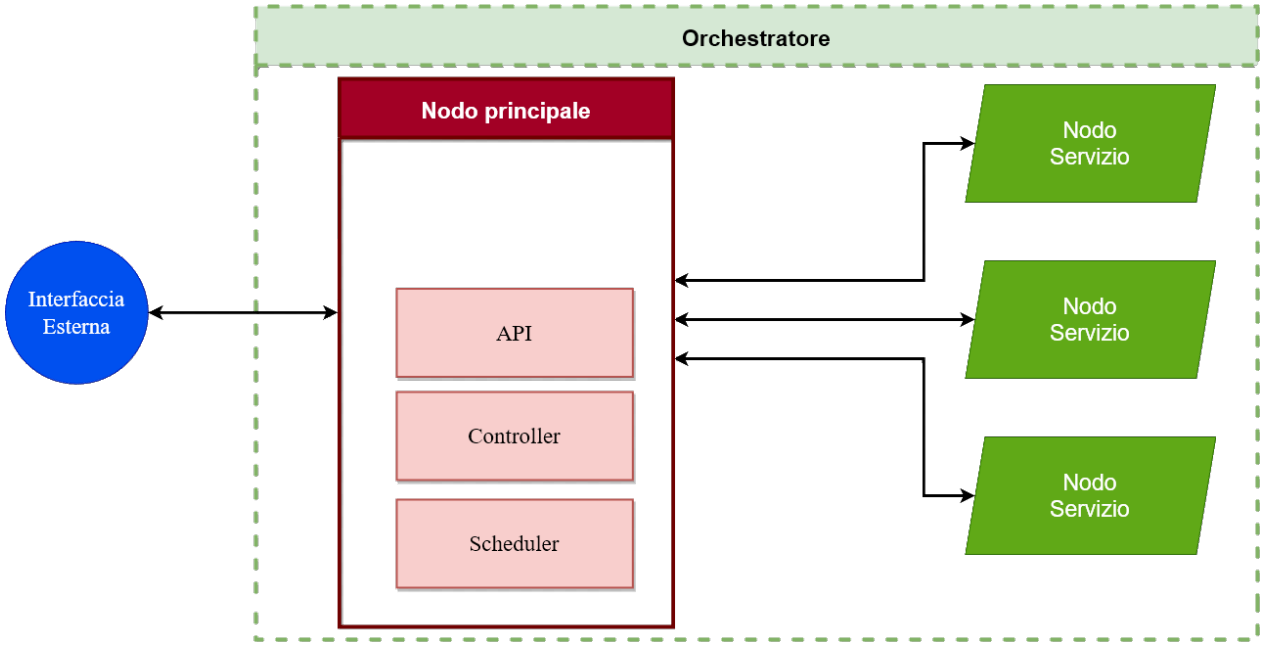
Per citare un esempio, in un possibile ambiente lavorativo è possibile permettere di avere molteplici sistemi operativi in esecuzione sullo stesso dispositivo è quindi offrire a più utenti un'ambiente lavorativo riducendo i costi e il numero di dispositivi da mantenere.

È importante saper scegliere quale delle due tecnologie è necessario impiegare per risolvere un determinato problema. Il punto di forza di container e immagini sono la grande portabilità dovuta alle dimensioni ridotte e all'autonomia del software contenuto in essi. Anche le virtual machine offrono tale possibilità, però è bene ricordare che tale operazione è piuttosto costosa dal punto di vista delle risorse per via della necessità di dover copiare tutti i dati e essenzialmente ricostruire tutta la virtual machine qualora sia necessario distribuire il software. Il forte legame tra virtual machine e l'hardware su cui risiede rendono questa tecnica più adatta a impieghi semi permanenti di risorse hardware. È importante ricordare che i container sono piuttosto limitati per certi aspetti, per esempio, su Windows non è possibile utilizzare il display per l'interfaccia utente e bisogna ricorrere ad altre alternative come una pagina web oppure terminale di comandi; mentre i software all'interno di una virtual machine hanno a disposizione l'intero sistema operativo in cui risiedono.

3.2 Orchestrazione container

Un altro grande punto di forza dei container è la relativa semplicità con cui e possibile gestire insiemi (cluster) di container tramite applicazioni specializzate. L'impiego di questi software permette di automatizzare la distribuzione di microservizi gestendo tutto ciò che è necessario, a partire dal monitoraggio costante di impiego di risorse e stato dei container fino ad arrivare ad avvii/riavvii di istanze qualora sia necessario.

Figura 3.1: *Grafico riassuntivo di un generico orchestratore*



Generalmente un orchestratore permette di scalare orizzontalmente sia in modo manuale che automatico in base all'impiego di risorse; e in caso di fallimento di specifici container (software non responsivo, morte improvvisa del programma, ecc..) essi vengono automaticamente riavviati e presentati all'utente solamente quando il sistema e pienamente funzionante. La figura 3.1 riassume come un orchestratore è concettualmente suddiviso al suo interno. Un nodo principale contenente tutti i sistemi di orchestrazione assume il ruolo di tramite tra il mondo esterno utilizzando un'interfaccia (interfaccia grafica, terminale comandi, ecc..) e i nodi contenenti i container impiegati. Di conseguenza i container vengono ulteriormente astratti dal servizio di orchestrazione che nasconde ciò che accade ai singoli container e dirottando le operazioni ad istanze funzionanti del servizio qualora dei specifici container incontrino problemi.

3.3 Alternative a Docker

Alcune alternative a Docker sono le seguenti, molte, anche più complesse, sono principalmente diffuse su Linux, mentre si trova molto meno varietà su Windows, e anche quelle che ci sono (anche Docker stesso) utilizzano un layer di compatibilità chiamatosi "Windows Subsystem Layer 2" per poter eseguire, senza overhead di virtual machine o dual-boot, ambienti GNU/Linux[7].

- Minikube - Permette la creazione di immagini e la gestione di Container attraverso un cluster locale, può eseguire software tramite container, virtual machine, oppure direttamente su "bare metal". Necessita di software aggiuntivo per la virtualizzazione software: Docker (Windows), HyperKit (MacOs), HyperV (Windows), KVM (Kernel-based Virtual Machine) (Linux), Parallels (Mac), Podman (Windows/Linux), Virtualbox, VMWare.
- Podman - Necessita di WSL2 e una distribuzione Linux installata su Windows.
- Rancher - Permette di gestire singoli container oppure gruppi (cluster) in modo automatico;
- Singularity - Permette di sostituire Docker completamente (Sia eseguire container che creare immagini), come Podman richiede una distribuzione Linux con WSL2 se si vuole utilizzare tramite Windows.
- Hyper-V Può eseguire container come una Virtual Machine.
- Packer - Supporta diverse immagini (AWS, Docker, HCP Packer)

Capitolo 4

Progetto formativo

4.1 Introduzione al progetto

Il lavoro svolto durante il tirocinio è suddivisibile in due parti: Nella prima è stato possibile ricercare e studiare i concetti fondamentali di container, immagini e Docker. Programmando tre applicazioni ognuna piú complessa della precedente, introducendo nuovi concetti. Nella seconda sono stati applicate le conoscenze acquisite per produrre un microservizio in grado di comunicare con una macchina industriale, questa comunicazione include sia una comunicazione diretta con i meccanismi automatici della macchina e comunicazioni con altre applicazioni esistenti che ricoprono ruoli differenti.

4.1.1 Ambiente di lavoro

Il software è stato creato con C# e compilato tramite Visual Studio 2019. Strumenti utilizzati:

- DOCKERFILE - Contiene le istruzioni necessarie per la creazione dell'immagine finale, specificando come ogni layer dell'immagine deve essere creato. Per esempio si può inserire nell'immagine il software gia compilato oppure si possono specificare le operazioni necessarie alla compilazione per rendere la creazione dell'immagine più flessibile permettendo la ricompilazione ogni volta che si rigenera l'immagine.

- DOCKER-COMPOSE.YAML - Il file non è necessario per la creazione dell'immagine però può appoggiarsi al *dockerfile* per orchestrare l'avvio di uno o più container specificando da dove recuperare le immagini (creazione al momento, localmente, download da un server esterno), opzioni aggiuntive, e in aggiunta, la creazione automatica di un network locale condiviso per ogni container generato dal *dockercompose*.

- GRPC E RABBITMQ - Sono i due metodi di comunicazione tra applicazioni che sono stati utilizzati durante questi progetti.

  - RABBITMQ - È un broker, ovvero ricopre il ruolo di mediatore tra trasmettitori e ricevitori, offre diverse opzioni per la gestione dei messaggi trasmessi tra cui una coda dove i messaggi vengono conservati qualora il ricevitore sia occupato.

  - GRPC - Permette a client e server di comunicare attraverso una comunicazione bi-direzionale costante astraendo tutto il processo di trasmissione dietro ad una semplice invocazione di un metodo all'interno del codice.

- MONGODB - MongoDB é stato scelto come database per il primo progetto per la sua semplicità di implementazione e di utilizzo.

La rete locale condivisa è fondamentale per la comunicazione locale tra container.

4.2 Primo progetto

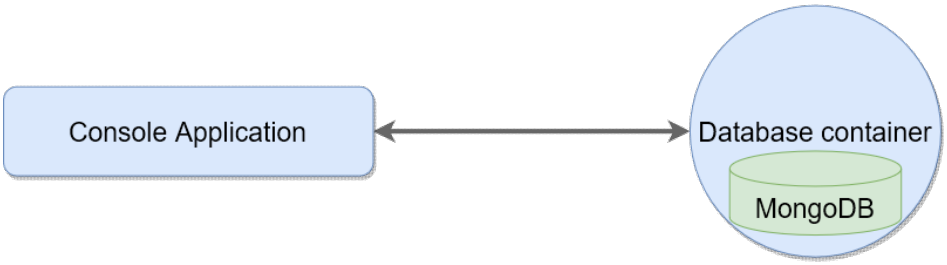
Nei primi mesi del tirocinio ho avuto modo di familiarizzarmi con container, immagini e Docker. Questa applicazione ha visto cinque variazioni, ognuna delle quali ha introdotto un nuovo concetto.

4.2.1 Applicazione base

Inizialmente è stata creata un'applicazione in .NET Core (framework base per applicazioni multi-piattaforma fornito da Microsoft) in grado di ricevere comandi da un terminale per eseguire delle operazioni base **(Create, Read, Update, Delete, List)** su un database gestito in un container separato. Il container MongoDB e la libreria che abilità la comunicazione tra applicazioni e container viene fornito direttamente dagli sviluppatori e perciò non ha necessitato di particolari modifiche.

È importante sottolineare come le applicazioni che vengono eseguite in un container non possono avere un'interfaccia utente o, più in generale, aprire finestre di dialogo. Come vedremo a breve, bisogna utilizzare una soluzione alternativa per poter permettere all'utente di interagire con il software al di fuori del terminale dei comandi.

Figura 4.1: *Definizione base dell'applicazione*



Per connettersi al database è necessario fornire un link che può essere un URL, URI o un indirizzo IP. Questo link può anche contenere credenziali per poter effettuare il login automaticamente o comunicare al database particolari impostazioni. La classe che gestisce le operazioni necessarie per effettuare la connessione al database segue il **Factory Design Pattern**' per poter testare se la connessione è effettivamente avvenuta; Ovvero, un metodo apposito crea l'oggetto che gestisce la connessione al database utilizzando la stringa di connessione fornita e lo restituisce al chiamante se la connessione risulta valida.

È stato scelto questo pattern per non dover verificare lo stato della connessione ogni volta che l'oggetto viene creato.

```
public static DBConnection CreateConnection(string conString)
{
  DBConnection obj = new DBConnection(conString);
  return obj.IsValid ? obj : null;
}
```

Mentre la classe che si occupa di gestire i comandi ricevuti dal terminale segue un **Singleton Pattern** attraverso le proprietà di C#. L'utilizzo del singleton permette di ridurre il numero di allocazioni per il processore di comandi e ne facilita l'utilizzo permettendo a qualsiasi parte del programma di accedere al processore, di fatto, sarebbe uno spreco di risorse dover allocare e inizializzare il processore per ogni comando ricevuto.

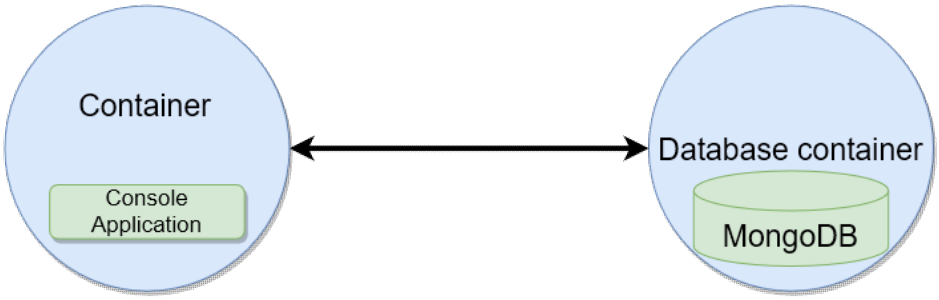
```
private static CLIProcessor SharedInstance;
public static CLIProcessor Instance
{
  get
  {
    if (SharedInstance is null)
    {
       SharedInstance = new CLIProcessor();
    }
    return SharedInstance;
  }
  private set
  {
    SharedInstance = value;
  }
}
```

Il codice salva in una variabile privata della classe un riferimento all'istanza di *CLIProcessor*, quando la proprietà statica viene invocata essa controlla se l'istanza è stata definita, se non lo è viene alloca un nuovo oggetto di tipo *CLIProcessor* altrimenti viene restituito quello già esistente.

4.2.2 Inserimento in un Container

Il passo successivo è stato quello di creare un'immagine contenente l'applicazione e poterla eseguire in un container.

Figura 4.2: *Inserimento dell'applicazione in un container*



L'obiettivo è quello di isolare completamente l'applicazione e il database dal sistema operativo host e di permettere la comunicazione tra i tre solamente tramite una rete locale e operazioni definite da interfacce rigide. Il *Dockerfile* è composto da:

```
1  FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
2  WORKDIR /app
3  EXPOSE 80
4  EXPOSE 443
5
6  FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
7  WORKDIR /src
8  COPY ["DockerDBContainer/DockerDBContainer.csproj", "DockerDBContainer/"]
9  RUN dotnet restore "DockerDBContainer/DockerDBContainer.csproj"
10 COPY . .
11 WORKDIR "/src/DockerDBContainer"
12 RUN dotnet build "DockerDBContainer.csproj" -c Release -o /app/build
13
14 FROM build AS publish
15 RUN dotnet publish "DockerDBContainer.csproj" -c Release -o /app/publish
16
17 FROM base AS final
18 WORKDIR /app
19 COPY --from=publish /app/publish .
20 ENTRYPOINT ["dotnet", "DockerDBContainer.dll"]
```

In questo caso si è scelto di compilare l'applicazione durante la creazione dell'immagine spostando tutti i file di progetto al suo interno. L'alternativa è quella di compilare il programma normalmente e successivamente, con comando "*COPY*" nel *Dockerfile* copiare solamente i file strettamente necessari all'esecuzione del programma. Questa opzione permette di scegliere esplicitamente cosa includere ma l'operazione è comunque sconsigliata. Compilare all'interno del processo di generazione dell'immagine permette di verificare il corretto inserimento di tutte le dipendenze dato che il processo avviene in un'ambiente isolato, e perciò non include tutto quello che potrebbe essere installato all'interno di un computer utilizzato da uno sviluppatore.

Gli step definiti nel *Dockerfile* seguono questa logica:

- Linee 1-4 - Il primo layer viene chiamato "base" ed è composto da una copia dei file base necessari per eseguire applicazioni .NET. All'interno dell'immagine viene impostata come directory base "app", infine vengono esposte le porte 80, 443 per permettere la comunicazione di rete;

- Linee 6-12 - Il secondo layer viene chiamato "build" ed è composto, inizialmente, da tutti i file necessari per l'SDK di .NETCore. Vengono copiati tutti i file del progetto all'interno di una cartella chiamata src, infine viene invocato il comando dotnet per compilare il progetto.

- Linee 14-15 - Il terzo layer viene chiamato "publish", viene invocato il comando "dotnet publish" per spostare in una cartella solo i file necessari all'esecuzione dell'applicazione.

- Linee 17-20 - Vengono copiati i file generati dal terzo layer in una cartella chiamatasi app e infine viene impostato il punto d'ingresso dell'applicazione, ovvero il punto iniziale per poter eseguire il software che verrà utilizzato automaticamente all'avvio di un container.

L'ordine e la presenza di questi layer determinano il codice identificativo finale che identifica l'immagine, per esempio, la creazione di immagini con nomi diversi, ma con dockerfile identico produrrebbe un codice identico.

Il file *docker-compose* effettua istruzioni più complesse non realizzabili nel *dockerfile*. Con l'invocazione del comando "*docker-compose up*" viene avviato il processo di creazione dei container:

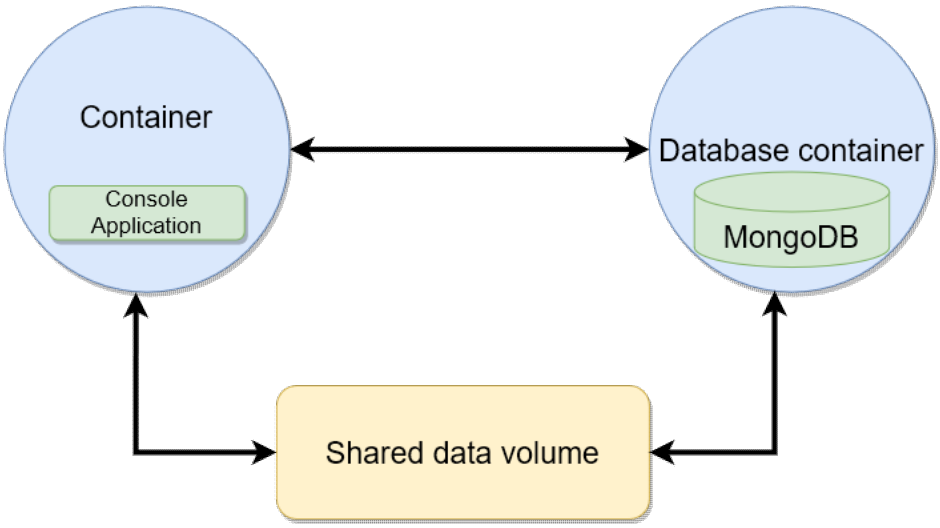
```
version: '3.4'
services:
  dockerdbcontainer:
    image: ${DOCKER_REGISTRY-}dockerdbcontainer
    build:
      context: .
      dockerfile: DockerDBContainer\Dockerfile
  mongo:
    image: mongo
    restart: always
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example
```

Esso si occupa di avviare le istanze dei due container (applicazione e database), impostare eventuali variabili d'ambiente (in questo caso viene impostata la password e nome dell'account admin del database). Inoltre, con l'utilizzo di *docker-compose* tutti i container generati da esso vengono inseriti in un network locale separato da altri eventuali container. Questo è anche realizzabile manualmente però *docker-compose* ne facilità la creazione.

La presenza di network separati per ogni gruppo di container è un ulteriore livello di sicurezza che può essere introdotto all'interno di un sistema con più servizi. In questo modo si evitano possibili collisioni qualora più servizi utilizzassero la stessa porta e/o indirizzo IP per la comunicazione.

4.2.3 Container e Filesystem Host

Figura 4.3: *Comunicazione tra container attraverso una cartella condivisa.*



Per capire come i container interagiscono con il file system, è stato aggiunta una cartella condivisa disponibile ad entrambi. Tramite un costrutto fornito dal linguaggio di programmazione (*"FileWatcher"*) vengono rilevate modifiche al contenuto di una cartella predefinita presente nel sistema operativo ospitante e se viene inserito un nuovo file questo provoca l'invocazione di un evento nel container principale. Un oggetto rileva questo evento, analizza il file e in base al contenuto invoca i comandi necessari per manipolare il container. Altre applicazioni potrebbero depositare in questa cartella una sequenza di comandi (sequenza **First In First Out**) che verranno eseguiti il prima possibile.

Il sistema operativo utilizzato (Windows) ha introdotto un problema: gli eventi possono essere ricevuti prima dell'effettivo completamento della scrittura pertanto si potrebbe creare un conflitto di permessi. Come soluzione si è deciso che subito dopo l'inserimento del file contenente i comandi sia creato un ulteriore file con nome identico, ma con estensione "*.ready*", e solo dopo aver rilevato quest'ultimo di procedere con la lettura e eliminazione del file di comando.

Rispetto alla verisone precedente del programma la logica principale non è cambiata, prevede solo l'aggiunta del seguente metodo, che permette di attivare o disattivare il FileWatcher:

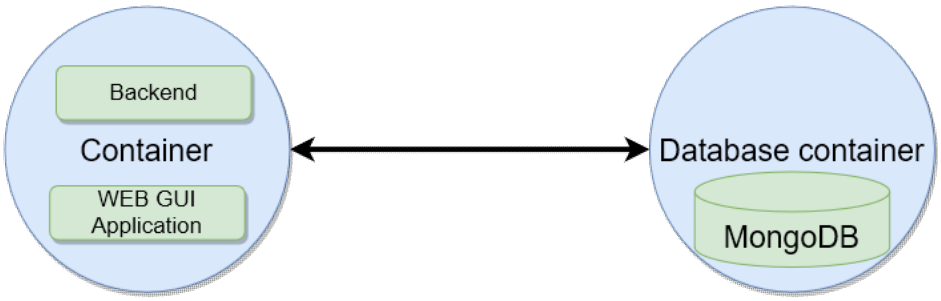
```
public void ToggleFileWatcher(bool bToggle){
if (FWatcher is null){
      FWatcher = new FileSystemWatcher(ConfigurationManager.AppSettings["FWATCH_PATH"]);
      FWatcher.NotifyFilter =    NotifyFilters.LastAccess |
                                                NotifyFilters.LastWrite |
                                                NotifyFilters.FileName |
                                                NotifyFilters.DirectoryName;

      FWatcher.Filter = "*" + ConfigurationManager.AppSettings["READY_EXT"];
      FWatcher.IncludeSubdirectories = false;
      FWatcher.EnableRaisingEvents = true;
}
  if (bToggle){
      FWatcher.Changed += OnFileCreated;
      ThreadSafePrint($"Starting folder watch on: { FWatcher.Path }");
      return;
}

  FWatcher.Changed -= OnFileCreated;
  ThreadSafePrint($"Stopping folder watch on: { FWatcher.Path }");
}
```

La cartella controllata viene impostata tramite un file di configurazione. Vengono impostati dei filtri per ricevere solamente eventi che riguardano la creazioni di nuovi file con una particolare estensione ("*.ready*"), non viene controllata la creazione dei file di testo perchè si assume che l'esistenza di un file "*.ready*" implica l'esistenza di un file "*.txt*" come menzionato in precedenza.

4.2.4 GUI da Container

È stata aggiunta una GUI (Graphical User Interface) per permettere all'utente di interagire con il container. Un web server presenta la pagina web nella rete locale permettendo all'utente di connettersi attraverso un qualsiasi browser.

Figura 4.4: *Aggiunta di una pagina Web per l'utente*



La pagina web permette di offrire all'utente un metodo più semplice per interagire con il database. In alternativa, è possibile aprire la pagina al pubblico e rendere accessibile il database a chiunque abbia le credenziali necessarie. La pagina agisce da intermediario tra utente e container e tramite i metodi di comunicazione visti nei punti precedenti manipola il database.

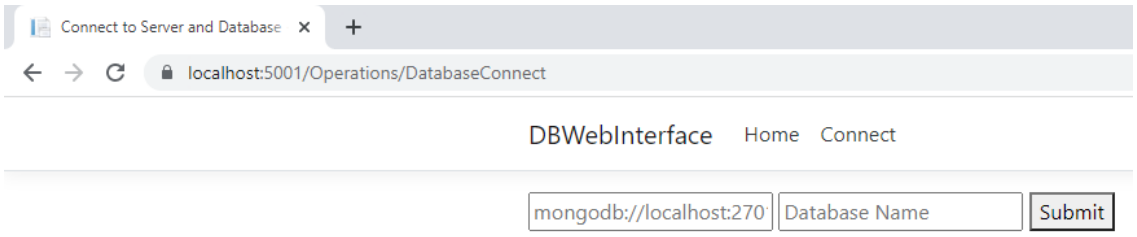
Il tipo di framework per l'applicazione è stato cambiato da *.NET Core* a *.NET Web* per poter usufruire dei componenti Web-hosting forniti da Microsoft. Il codice necessario per la creazione dell'host non verrà mostrato dato che si tratta di codice standard generato automaticamente e quindi poco interessante ai fini di questo testo.

Il codice sottostante mostra come il metodo di connessione al database non sia cambiato molto rispetto ai punti precedenti, salvo la gestione dei dati "permanenti" che devono esistere per tutta la durata della sessione. Viene utilizzato un dizionario accessibile da qualsiasi classe per leggere/scrivere dati che non devono essere cancellati al caricamento di una nuova pagina. È importante specificare che questo dizionario non contiene dati sensibili per via dell'assenza di garanzie sui dati.

```
private void AttemptConnection(DatabaseModel DBModel)
{
  // Codice omesso ...
      string ConLink = DBModel.ConnectionLink;
  // Codice omesso ...
      ApplicationData.Items["DatabaseConnection"] = DBConnection.CreateConnection(ConLink);
  // Codice omesso ...
}
```

Con questa applicazione, non solo i due processi sono completamente isolati a livello di risorse, ma l'interfaccia risulta anche "universale" per un qualsiasi database MongoDB.

Figura 4.5: *Come appare la pagina web una volta connessi tramite browser.*

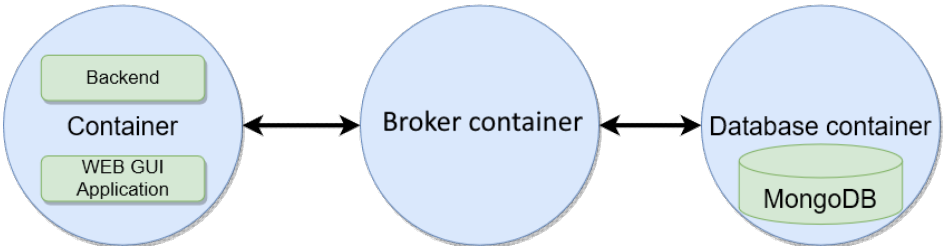


4.2.5 Messaggistica con Broker

Un sistema broker è un software che supporta uno o più metodi per la comunicazione attraverso messaggi tra applicazioni, astraendo il processo di trasmissione e utilizzando interfacce standardizzate per la gestione dei dati. Generalmente un sistema broker garantisce l'arrivo di un messaggio e rispetta l'ordine con cui essi vengono inviati qualora il ricevitore non sia in grado di consumare nuove trasmissioni per un qualsiasi motivo.

Per creare un sistema più simile alla realtà è stato aggiunto un container contenente un software broker (RabbitMQ) tra il container interfaccia e il container database.

Figura 4.6: *Comunicazione tra container gestita da un broker.*



Questo Broker riceve e invia messaggi tra interfaccia web e database, aggiungendo un ulteriore livello di astrazione tra le due applicazioni. In questo modo, il broker è in grado di gestire il carico dei messaggi in modo autonomo.

Sono stati aggiunti due nuovi componenti per permettere questo tipo di comunicazione: la classe "MessageHandler" che contiene tutto il necessario per inviare/ricevere messaggi al/dal Broker, e l'interfaccia "IAppMessage" con i metodi:

Source Code 4.1: *Metodi dell'interfaccia IAppMessage estendibili da una qualsiasi classe*

```
public byte[] GetSender();
public byte[] GetContents();
```

che permettono la definizione di nuovi messaggi senza dover implementare un metodo apposito in MessageHandler.

Quando l'oggetto di tipo MessageHandler viene creato la connessione al Broker viene gestita automaticamente tramite il costruttore, l'indirizzo IP deve essere definito tramite variabile d'ambiente.

Di seguito, vengono spiegati brevemente tutti i metodi della classe per dare un'idea generale di come comunicano l'interfaccia web e il database. I due container possono agire come ricevitori e trasmettitori, segnalando al Broker, tramite metodi appropriati, l'intenzione di iscriversi alla coda dei messaggi ricevuti/trasmessi definita in questo modo:

Source Code 4.2: *Definizione di una coda generica per messaggi con RabbitMQ.*

```
Channel.QueueDeclare(queue: QueueName,
             durable: true,
             exclusive: false,
             autoDelete: false,
             arguments: null);
```

Il metodo sopraesposto inizializza una coda all'interno del broker ed è sufficiente per permettere la trasmissione di messaggi tramite la chiamata alla funzione:

Source Code 4.3: *Invio di un messaggio tramite RabbitMQ con la queue definita in precedenza.*

```
public void PublishMessage(IAppMessage Message)
{
// Transform the Message contents and name into a byte array
  var Bytes = Message.GetSender()
    .Concat(ByteMessageSeparator)
    .Concat(Message.GetContents())
    .ToArray();
// Send data
  Channel.BasicPublish(exchange: "",
    routingKey: QueueName,
    basicProperties: Properties,
    body: Bytes);
}
```

Essa riceve un oggetto che implementa l'interfaccia IAppMessage, genera un array di byte e li invia al broker per essere processati.

La trasmissione invia pacchetti di bytes quindi è necessario convertire i dati del messaggio, in questo caso *sender* e *contents*, in byte e raggrupparli in un unico array codificato. Grazie all'utilizzo di un'interfaccia la responsabilità di convertire i dati in byte è lasciata allo sviluppatore.

Se si vuole abilitare la ricezione di messaggi è necessario invocare due ulteriori funzioni:

```
void AsReceiver()
```

La quale segnala al Broker che l'oggetto vuole ricevere i messaggi presenti nella queue, e la funzione:

```
void SubscribeToReceiveEvent(EventHandler eventHandler)
```

Che permette all'oggetto di ricevere l'evento che viene invocato dal broker quando un messaggio viene ricevuto.

In questo caso, sia la GUI (Client) sia il database (Server) agiscono come trasmettitori e ricevitori. Ogni volta che una trasmissione viene ricevuta il codice che la gestisce deve inviare un messaggio di "Acknowledge" per confermare la ricezione del messaggio originale. In questo modo, qualora si dovesse presentare un errore, il broker provvedere a inviare nuovamente il messaggio.

Il server impiega lo stesso codice usato fino ad ora per processare i comandi da terminale, e i messaggi inviati in rete contengono una stringa (criptata in invio e decriptata in ricezione) che rappresenta il comando da processare.
Per esempio, il comando *"add databaseA CollezioneB A,B,C"* istruirà il database di aggiungere gli elementi A, B e C alla collezione "CollezioneB" presente nel database nominato "databaseA".

4.3 Secondo progetto

In questo progetto l'obiettivo è stato quello di applicare i concetti imparati nel primo applicando una filosofia a microservizi per una macchina taglio laser. Il computer affiancato alla macchina è incaricato di eseguire un'applicazione grafica che permette all'utente di controllare lo stato dei componenti, modificare parametri e iniziare nuovi lavori. Alcuni sotto menù di questa applicazione sono un'ulteriore servizio gestito separatamente che viene avviato al momento per eseguire un compito specifico.

In questo caso l'incarico consisteva nel convertire una di queste applicazioni, seguendo con molta attenzione la filosofia dei micro-servizi (autonoma, contenuta, senza dipendenze) e di creare l'interfaccia utente in grado di comunicare con esso. L'applicazione è un server che contiene tutti i dati associati ai vari ugelli attualmente presenti nella macchina, alcuni di questi dati possono essere: la posizione attuale, il diametro, e le ore totali d'uso dell'ugello. Gli obiettivi posti erano:

- Modificare il codice sorgente per l'esecuzione in un container Docker;
- Programmare un'interfaccia tramite il framework ASP.NET MVC;
- Generare un programma d'installazione in grado di eseguire il deploy dei container in modo autonomo e rapido;

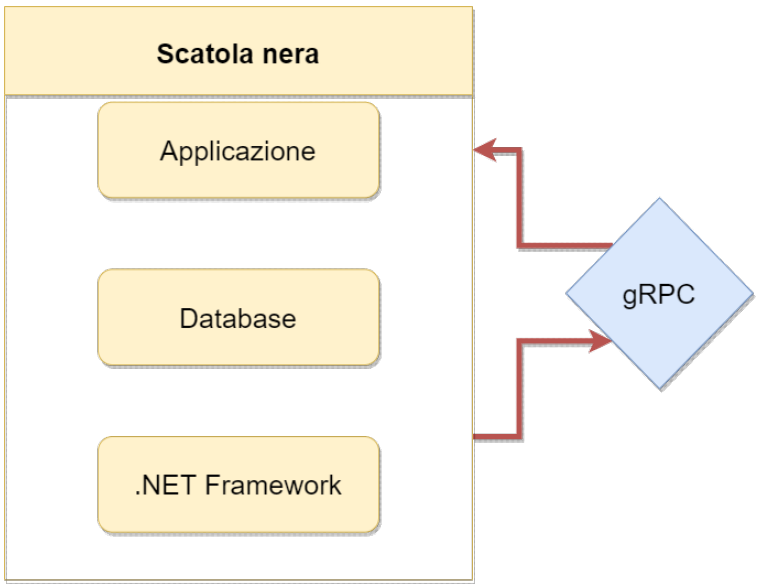
4.3.1 Inserimento di un Server .NET Framework in un container

In primo luogo il software è stato convertito da ".NET Framework" a ".NET Core", questo perchè è disponibile un'immagine base estendibile per container che include tutte le dipendenze strettamente necessarie per eseguire programmi .NET Core. Con questa operazione è stato possibile rimuovere parti in eccesso e ridurre la dimensione complessiva del programma.

Le parti rimosse includono "WindowsForms", un framework per interfacce utenti per gestire le operazioni base dell'applicazione, e una varietà di dipendenze proprietarie non supportate da .NET Core. Questo ha permesso di includere nel software solamente le parti strettamente necessarie ai fini di questo progetto semplificandone la struttura.

Sfruttando le operazioni messe a disposizione dai file di set-up di Docker visti in precedenza (dockerfile e docker-compose) è stato possibile gestire la fase di distribuzione del software tramite un singolo comando.Il comando esegue il container esponendo le porte di comunicazione di rete e collegando determinate cartelle esterne per permettere la scrittura e lettura di file particolarmente importanti come file di log e di impostazione, di conseguenza si introduce la possibilità di utilizzare questi file al di fuori del container.

Figura 4.7: *Composizione del container con server.*



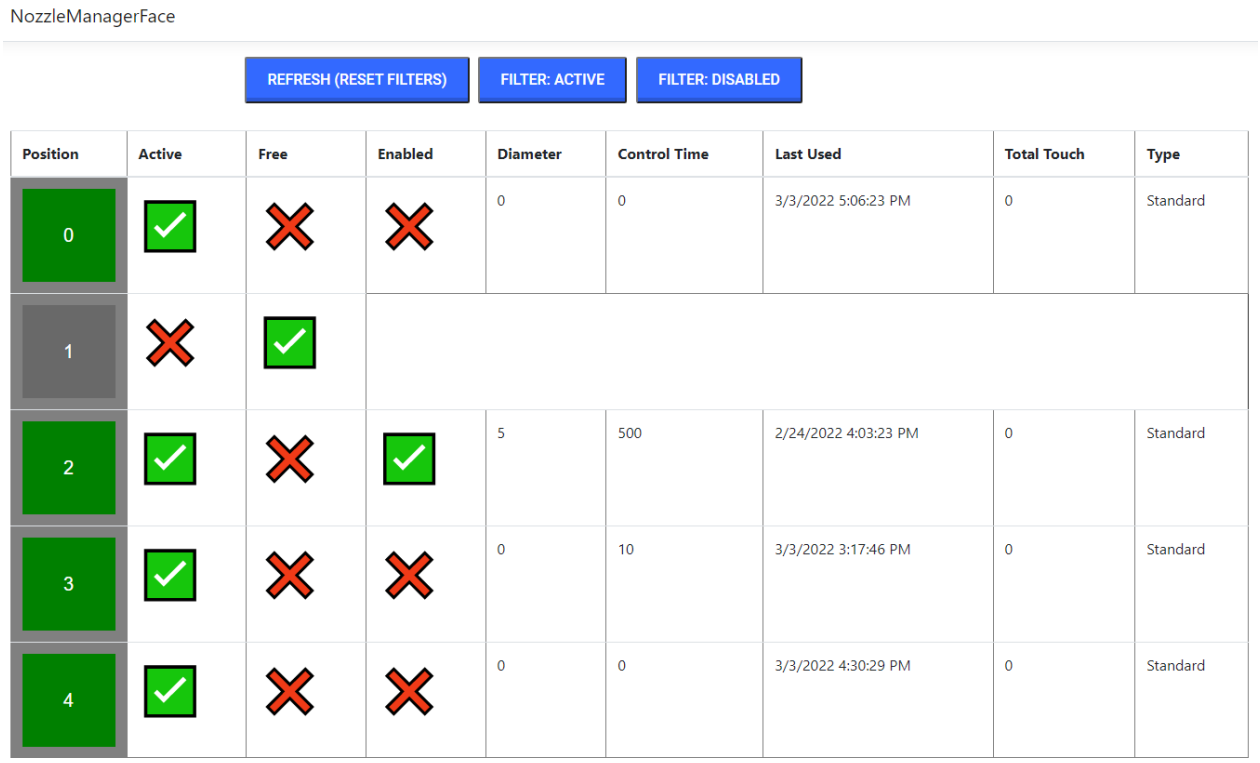
4.3.2 Client con interfaccia utente real-time

È stata utilizzata una combinazione di C#, JavaScript, CSS, e HTML con il framework ASP.NET Web per la creazione del frontend dedicato all'interazione tra applicazione e utente. Il compito principale dell'interfaccia è quello di informare l'utente dello stato attuale della macchina taglio laser e di permettere l'aggiornamento dei dati riguardanti gli ugelli in utilizzo.

Tramite un protocollo di rete sicuro (web socket) è stato possibile garantire la sincronizzazione dei dati tra client e server. L'utente è in grado di apportare modifiche alla tabella presente nella figura 4.8 e la macchina si adatta di conseguenza in tempo reale.

Siccome è anche possibile aggiornare il database attraverso ulteriori applicazioni il web socket notificherà la pagina web di eventuali cambiamenti e comporterà un aggiornamento della tabella automatico.

Figura 4.8: *Interfaccia utente connessa al server attivo in un container.*



4.3.3 gRPC vs Broker

Una grande differenza tra il primo e il secondo progetto è la metodologia scelta per comunicare tra client(s) e server. Nel progetto precedente era stato utilizzato il broker RabbitMQ, mentre in questo caso il server e i messaggi che esso utilizza erano già stati progettati con tecnologia gRPC[10], un protocollo sviluppato da Google per comunicazioni bi-direzionali tra servizi.

L'utilizzo di gRPC in alternativa ad un broker comporta differenze nel protocollo di comunicazione sostanziali:

- RabbitMQ necessita di un container a sè per fare da intermediario tra client e server, con gRPC il protocollo viene compilato assieme al programma e viene utilizzata una libreria per gestire la comunicazione;
- RabbitMQ utilizza una coda per gestire i messaggi mentre gRPC utilizza un'architettura a eventi dove il codice che vuole ricevere informazioni si sottoscrive all'evento specifico che verrà invocato quando un messaggio sarà ricevuto;

Perchè utilizzare uno o l'altro? Dipende dal contesto e dal tipo di protocollo a cui si mira. Se non è importante chi processa il messaggio allora un broker può essere una soluzione adatta per coordinare più applicativi. Invece se è necessario avere una corrispondenza 1:1 tra client e server gRPC fornisce un'architettura più adatta.

Per poter utilizzare gRPC bisogna seguire dei passaggi particolari:

1) Definire il protocollo, gli eventi e i messaggi che client e server utilizzeranno per comunicare;
2) Creare e inizializzare il canale di comunicazione a lato server;
3) Sottoscriversi a lato client agli eventi interessati;

Dopo aver effettuato questi passaggi sarà possibile utilizzare i metodi definiti all'interno del protocollo gRPC per inviare istruzioni e dati.

Nel progetto è stato necessario programmare nuovamente il metodo con cui il canale e la comunicazione venivano gestiti per permetterne il funzionamento all'interno di un container.

Se non strettamente impostato dallo sviluppatore porte e indirizzi IP risultano dinamici per i container, quindi per permetterne la connessione in quel particolare caso è stato introdotto del codice (4.5).

Anche in questo progetto sono state definite le operazioni **Create, Read, Update e Delete** nel protocollo gRPC.

Source Code 4.4: *Implementazione parziale del protocollo utilizzato nel programma con esempio di definizione di messaggio e trasmissione*

```
 syntax = "proto3";
option csharp_namespace = "Data.GrpcProto";
service DataGrpcProto
{
        // ....
        //UPDATE_DATA_REQUEST
        rpc UpdateDataRequest(SetDataRequest)  returns (Message){}
        // ...
}
message SetDataRequest
{
        int32 Position        = 1;
        int32 Enabled         = 2;
        double Diameter       = 3; // (mm)
        int32 LifeTime        = 4;
        int32 UsedTime        = 5;
        int32 Type            = 6;
}
```

Un messaggio è composto da variabili di dati primitivi (interi, float, double, boolean, ...) oppure altri messaggi possono essere utilizzati come tipi.

Nel protocollo vengono definiti i metodi da invocare a lato server/client per l'invio/ricezione di dati con la seguente sintassi:

```
rpc <NomeMetodo>(<NomeMessaggioTrasmetto>) returns (Message){}
```

Sarà compito di chi invia il messagio di popolare la struttura dati con le informazioni da trasmettere.

4.3.4 Browser Embedded

Per completare il progetto è stata creata una semplice applicazione per avere più controllo sulla presentazione e poter mascherare l'uso di un browser per l'interazione con il software.

È stata creata una libreria con cui è possibile interagire con Docker astraendo l'invocazione di comandi powershell tramite metodi appositi, in questo esempio, fornendo il nome del container si è in grado di ottenere l'indirizzo IP esterno. L'utilizzo di questa libreria permette all'applicazione di connettersi al server senza richiedere azioni particolari dall'utente. Nonostante questo, si è scelto di permettere la comunicazione tra i container e/o host tramite l'indirizzo "localhost" come previsto dal progetto originale per evitare di dover modificare il software PLC properietario che necessita di quel particolare punto d'ingresso per comunicare con il server.

All'avvio di questa applicazione il browser cerca di connettersi al container e se questa operazione termina con successo mostra all'utente i dati presenti nel database ugelli.

Source Code 4.5: *Implementazione parziale del protocollo utilizzato nel programma con esempio di definizione di messaggio e trasmissione*

```
private static string PowershellQueryContainerIP(string ContainerName){
    using (var pShell = PowerShell.Create())    {
        pShell.AddCommand("docker")
    .AddParameter("inspect")
    .AddParameter("-f", "'{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'")
    .AddArgument(ContainerName);
        var Result = pShell.Invoke();
        if (Result is null || Result.Count == 0)
        {
            return null;
        }
        var Output = Result[0].ToString();
        if (Output == string.Empty || Output == "\'\'"){
            // Return original if no results where found
            return ContainerName;
        }
        return Output.Replace("\'", string.Empty);
    }
}
```

Capitolo 5

Conclusioni

L'applicazione finale è stata completamente inserita in un container senza perdite di funzionalità. La conversione dell'applicazione a .NET Core ha comportato un miglioramento nelle prestazioni di avvio e chiusura del server, soprattutto dovuta alla rimozione di componenti come WindowsForms.

L'immagine generata non necessità di dipendenze esterne quindi i container generati da essa risultano in linea con la filosofia dei micro servizi, ovvero l'applicazione contenuta in essi è completamente autonoma e può essere eseguita senza il supporto di ulteriori programmi. In aggiunta, è stato mantenuto lo stesso protocollo di comunicazione precedentemente definito, inserendo solamente un livello di astrazione dovuto all'utilizzo di un container. Per questi motivi rimpiazzare l'applicazione originale sarebbe triviale.

Avendo isolato il programma all'interno di un container si ha la garanzia che tutte le comunicazioni avvengono attraverso l'interfaccia di rete definita tramite gRPC e quindi di poter apportare cambiamenti anche sostanziali al programma fintanto che l'interfaccia di comunicazione rimane identica senza causare particolari problemi a tutti gli applicativi che dipendono da essa.

Bibliografia

1. “Overview of Scaling: Vertical And Horizontal Scaling”. www.geeksforgeeks.org/overview-of-scaling-vertical-and-horizontal-scaling. Accessed 30 March 2022.

2. Campbell, Sean, and Michael Jeronimo. ”An introduction to virtualization.” Published in “Applied Virtualization”, Intel (2006): 6-8. Accessed 31 March 2022.

3. “Open Containers Initiative”. www.opencontainers.org. Accessed 2 December 2021.

4. “A brief history of microservices patterns”. https://developer.ibm.com/articles/cl-evolution-microservices-patterns. Accessed 11 April 2022.

5. ”Red Hat”. www.redhat.com/topics/virtualization/what-is-virtualization. Accessed 22 December 2021.

6. Red Hat. www.redhat.com/topics/containers/containers-vs-vms. Accessed 22 December 2021.

7. Windows Subsystem for Linux. www.docs.microsoft.com/windows/wsl. Accessed 2 December 2021.

8. “Docker”. www.docs.docker.com/storage/storagedriver. Accessed 2 December 2021.

9. “Microservizi”. [Ebook version].www.docs.microsoft.com/dotnet/architecture/microservices. Accessed 22 December 2021.

10. gRPC, Google, www.grpc.io. Accessed 30 January 2022.