LPaaS on Docker — Final Report

Niccolò Maltoni
niccolo.maltoni@studio.unibo.it

Federico Naldini
federico.naldini3@studio.unibo.it

Settembre 2019

Il progetto, sviluppato nell'ambito del corso di Sistemi Distributi, ha lo scopo di realizzare deploy e distribuzione di LPaaS (Logic Programming as a Service) con un approccio a *container*. A tal fine ci è stato richiesto di integrare la più recente versione di LPaaS con le moderne tecnologie di containerizzazione; una volta fatto ciò, avremmo dovuto valutare una piattaforma per la distribuzione dei *container* così realizzati con particolare attenzione alla gestione dello stato interno.

1 Obiettivo e requisiti

L'obiettivo principale del progetto è quello di realizzare un adattamento per Docker di LPaaS [**?**] (Logical Programming as a Service); in particolare, il progetto è volto a distribuire l'applicazione attraverso la piattaforma Swarm di Docker.

1.1 Scenari

LPaaS mette a disposizione un servizio di computazione logica remoto attraverso API, ma l'implementazione a singolo server non è facilmente scalabile: uno scenario di utilizzo dove numerosi dispositivi client interrogano una singola istanza del server (ad esempio in un contesto IoT) metterebbe seriamente in difficoltà il sistema. La possibilità di scalare orizzontalmente è dunque necessaria.

Nel contempo, è importante riuscire ad incapsulare l'applicazione in un contesto il più possibile *self-contained*, in modo da ridurre il più possibile le dipendenze esterne e favorirne la distribuzione; da qui nasce il concetto di container.

Con **container** si intende, in generale in un contesto informatico, un ambiente di esecuzione per un'applicazione isolato e contenente tutto il necessario per il funzionamento. Più nello specifico, nell'ambito della virtualizzazione, il container è un'istanza isolata

nello spazio utente, che offre, di fatto, un ambiente virtualizzato a livello di sistema operativo; è questo il caso di Docker[1].

In particolare, la tecnologia Docker mette a disposizione la struttura Swarm: è possibile inizializzare, dato un certo numero di macchine connesse da una rete e che eseguono il demone Docker, un contesto distribuito sul quale eseguire applicazioni "containerizzate".

Uno degli scenari più comuni è il cloud: la possibilità di eseguire LPaaS in un ambiente come quello messo a disposizione da piattaforme come AWS[2], Azure[3] o GCP[4] permette di risolvere buona parte dei problemi di carico sul server, anche in caso di migliaia di client.

1.2 Politica di autovalutazione

Il nostro lavoro nello sviluppo di LPaaS si configura più come un obiettivo di ricerca piuttosto che una vera aggiunta di funzionalità: si parla infatti di prendere l'attuale architettura del sistema e calarla in un contesto di effettiva distribuzione. La nostra sfida si basa quindi sulla risoluzione di problemi assolutamente noti in ambito di sistemi distribuiti (come ad esempio consistenza, resilienza, scalabilità...) e già affrontati con diverse soluzioni in letteratura; di conseguenza il nostro compito sarà quello di ricercare vari modelli teorici, trovare quello che più si addice a LPaaS e infine realizzare un effettivo prototipo.

Alla luce di ciò, abbiamo individuato i seguenti criteri di autovalutazione.

- **Coerenza con LPaaS**: Prendiamo in mano LPaaS con la consapevolezza che ci stiamo approcciando a un progetto con obiettivi già definiti e una realizzazione in linea con questi ultimi: di conseguenza qualunque nostra modifica all'architettura e/o componenti interni si deve porre in assoluta e rigorosa continuità con quanto già realizzato.
- **Efficacia dell'architettura proposta**: Punto focale del nostro lavoro sarà la progettazione e conseguente realizzazione di un'architettura distribuita per LPaaS; questo modello dovrà dimostrarsi idoneo sulla base di numerosi criteri quali scalabilità, consistenza, suddivisione del carico di lavoro e infine resilienza.
- **Implementazione reale della soluzione**: La soluzione da noi pensata dovrà essere realizzabile nel pratico, possibilmente cercando di perseguire i criteri di semplicità e efficacia.

[1] `https://www.docker.com/`
[2] `https://aws.amazon.com/ecs/`
[3] `https://azure.microsoft.com/services/kubernetes-service/docker/`
[4] `https://cloud.google.com/kubernetes-engine/`

2 Analisi dei requisiti

2.1 Ipotesi implicite

Il funzionamento del software prodotto è strettamente legato alla correttezza di funzionamento di LPaaS: in particolare, dalla logica implementativa utilizzata per il dialogo con il livello di persistenza dipende la corretta gestione dello stato in un contesto di distribuzione.

2.2 Requisiti impliciti

Si prevede di mantenere lo stato attuale sistema, eventualmente integrando nuove versioni delle librerie utilizzate (una su tutti, tuProlog), ma senza variare le funzionalità offerte al di fuori del contesto di distribuzione tramite Docker.

Collegato a ciò, è implicitamente richiesto che le scelte di progettazione e tecnologiche preesistenti, in particolare nel contesto server e persistenza, vengano rispettate: è implicitamente necessario rimanere, per quanto possibile, all'interno del contesto di Java EE e mantenere la compatibilità con RDBMS SQL.

2.3 Requisiti principali

Durante la fase di analisi dei requisiti sono stati individuati i seguenti requisiti imprescindibili:

- devono essere realizzate immagini Docker a partire dalle quali possano essere istanziati container in grado di eseguire su piattaforma Docker.
- i container Docker che eseguono LPaaS devono garantire uno stato consistente tra le repliche qualora eseguite in contesto Docker Swarm.
- i container che eseguono LPaaS devono essere resilienti agli errori e il servizio, dal punto di vista dei client, deve rimanere disponibile.
- il servizio messo a disposizione da un Docker Swarm, dal punto di vista dei client, deve essere trasparente e agire come una singola istanza del server LPaaS.
- LPaaS, una volta adattato, deve essere compatibile con l'ultima versione rilasciata di tuProlog, per quanto possibile.

2.4 Tecnologie

2.4.1 REST

LPaaS è un applicativo server che costituisce una *façade* per il motore tuProlog [**?**] attraverso API REST.

Con API REST (o ReST, Representational State Transfer [**?**]) si intende un'astrazione degli elementi architetturali di un sistema ipermediale distribuito su larga scala

(per esempio, su internet), che pone l'accento sul concetto di componenti che interagiscono tramite trasferimento di risorse in uno specifico formato e ad uno specifico percorso (l'URI [**?**]).

Punto interessante dell'architettura REST è il non prevedere il concetto di sessione ed essere, dunque, considerabili **stateless**; tale caratteristica permette un incapsulamento del codice migliore, permettendo la realizzazione di servizi *loosely-coupled* che possono essere eseguiti in ambienti separati, come ad esempio i container.

2.4.2 Docker

Il progetto open source Docker [**?**] viene rilasciato nel 2013 da una compagnia chiamata dotCloud (ora rinominata Docker), che lavorava su software di tipo *PaaS* (Platform as a service) per il cloud computing.

Docker si pone come obiettivo di automatizzare lo sviluppo di applicazioni all'interno di *container* software, sfruttando tutti i vantaggi di una virtualizzazione a livello di sistema operativo; per riuscire a fare ciò, si avvale delle funzionalità di isolamento presenti nel kernel Linux, come ad esempio `cgroups`[5], utilizzato per la gestione di risorse fisiche, e `namespaces`[6], che invece garantiscono un isolamento a livello di processo.

Grazie a diverse tecniche messe in gioco, come ad esempio l'utilizzo del filesystem *UnionFS*[7] per permettere di sovrapporre i filesystem di diverse immagini o l'utilizzo della virtualizzazione a livello di sistema operativo, Docker mette a disposizione *containers* leggeri dal punto di vista delle risorse, di facile costruzione e modifica da parte degli utenti e infine altamente compatibili con la maggior parte delle piattaforme standard di cloud computing (AWS, Google Cloud Platform, OpenStack...). Passando poi all'effettivo funzionamento del sistema Docker, il cuore di tutto può essere identificato nel modulo *Docker Engine* [**?**], la cui architettura viene descritta in Figura 1.

[5] `https://en.wikipedia.org/wiki/Cgroups`
[6] `https://en.wikipedia.org/wiki/Linux_namespaces`
[7] `https://en.wikipedia.org/wiki/UnionFS`

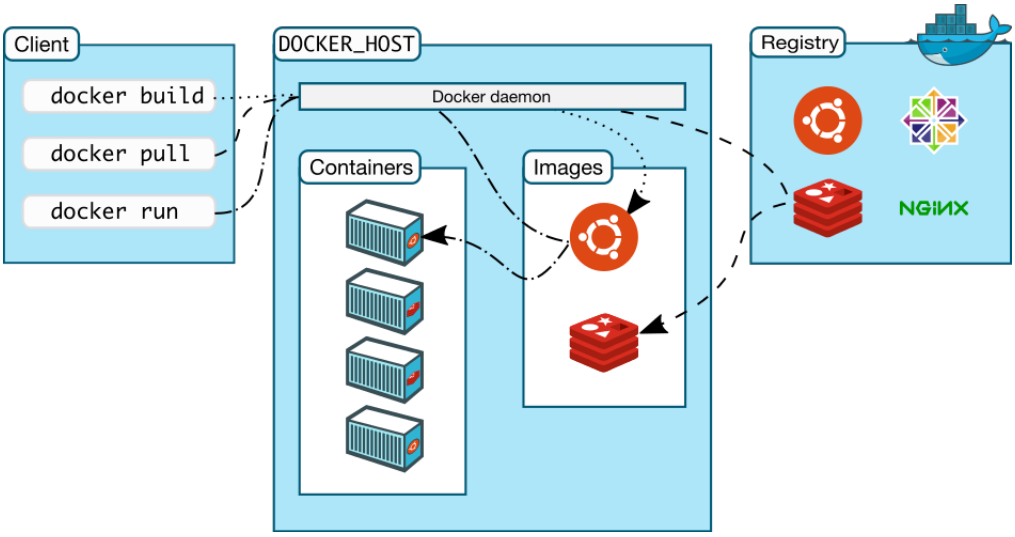

Figura 1: Architettura del sistema Docker

Le componenti principali all'interno di *Docker Engine* sono di fatto tre:

- **Docker Daemon**: demone eseguito in background che si occupa della maggior parte del lavoro, gestendo sotto comando tutti gli oggetti del sistema;
- **Docker CLI**: componente col compito di fornire un insieme di comandi all'utente per poter istruire il demone sul da farsi; comunica con *Docker Daemon* via socket se locale, o via API HTTP se remoto.
- **Docker Registry**: registro che si occupa della memorizzazione delle *Docker Images*. Docker mette a disposizione un registro pubblico, chiamato *DockerHub*, in cui ognuno può postare le proprie immagini e scaricare quelle di altri utenti. Viene contattato da *Docker Daemon* nel momento in cui quest'ultimo non disponga di una versione locale dell'immagine richiesta dall'utente.

Parlando poi del ciclo di vita dei *container* gestiti dall'architettura sopra descritta, è importante sottolineare l'approccio *stateless* nella gestione delle modifiche allo stato interno di questi ultimi: Docker infatti non prevede di default un meccanismo di conservazione dello stato tra diverse istanze della stessa immagine, al contrario ogni *container* basato su quest'ultima viene gestito indipendentemente dagli altri.

2.4.3 Java EE e Payara

LPaaS è realizzato secondo le specifiche Java EE [**?**] (o J2EE, Java Platform Enterprise Edition) e si appoggia all'*application server* Payara.

Payara[8] è un progetto derivato da Glassfish, implementazione di riferimento delle API di Java EE realizzata da Oracle stessa, che permette il deploy di applicazioni Java Enterprise compilate in formato `war`; sono disponibili 2 differenti versioni:

Payara Server Full versione completa di Payara Server, che permette configurazioni anche complesse tramite linea di comando o interfaccia web.

Payara Micro versione di Payara Server orientata ai microservizi e dunque estremamente minimale (pesa meno di 80 MB); permette esclusivamente il deploy di file `war` e supporta configurazioni tramite variabili d'ambiente ed Eclipse MicroProfile[9], non richiedendo dunque alcun tipo di installazione.

La versione preferita per la realizzazione del progetto è Payara Micro: infatti, le dimensioni contenute e la velocità di avvio la rendono perfetta per l'esecuzione all'interno di container Docker; inoltre la possibilità di prelevare il `jar` da Maven per il deploy diretto del `war` su di esso permette l'utilizzo di configurazioni Gradle per il lancio dell'applicazione in fase di sviluppo, senza richiedere l'installazione del server sulla macchina locale dello sviluppatore.

[8] `https://www.payara.fish/`
[9] `https://microprofile.io/`

3 Design

Al termine della fase di analisi dei requisiti, è risultato evidente che il progetto non richiedesse lo sviluppo di complesse strutture software, quanto più l'integrazione del progetto preesistente con uno strumento di deploy, ossia Docker.

Nelle Sezioni successive, dunque, si analizzerà anche parti della struttura preesistente rilevanti all'integrazione con Docker, oltre alle eventuali modifiche necessarie per l'integrazione con quanto realizzato.

In primo luogo occorre sottolineare che il principale problema emerso in fase di analisi riguardava la gestione dello stato interno del sistema: nel momento in cui LPaaS veniva distribuito su diverse istanze, mantenere una coerenza dello stato interno tra le varie repliche era sicuramente la sfida più grande.

Successivamente va ricordato che i *container* Docker sono progettati per essere *stateless*, quindi trascurare la gestione dello stato. Infine vale la pena spendere qualche parola sulla gestione dello stato interno di LPaaS, argomento che verrà poi approfondito nel dettaglio nella sezione riguardante l'implementazione effettiva, per ora basti sapere che l'intero stato di LPaaS è affidato a un database interno messo a disposizione dal server Payara.

L'obiettivo del nostro lavoro era quindi decisamente ambizioso: integrare tutti i vantaggi dell'approccio a *containers* e le relative piattaforme per la distribuzione con una gestione coerenete e efficace dello stato interno del sistema.

3.1 Struttura

La soluzione da noi proposta è stata progettata secondo il paradigma *Separation of Concerns*, modello che prevede la separazione delle funzionalità di un sistema in diversi moduli.

Analizzando a fondo l'implementazione di LPaaS, ci siamo resi conto che non solo il sistema in questione era stato progettato già secondo il pattern sopracitato, ma che addirittura la separazione tra *Buisness Logic* e stato interno del sistema era particamente già implementata: infatti ogni memorizzazione di dati a lungo termine veniva fatta sfruttando il database interno di Payara, senza utilizzare strutture dati *in-memory* se non in alcuni passaggi non di vitale importanza.

Alla luce di ciò, LPaaS è stato diviso nei seguenti moduli:

- **LPaaS Business Logic module**: Tutte le funzionalità esposte dal sistema LPaaS, le quali delegano al modulo database la persistenza dello stato, rendendo difatti questo modulo etichettabile come *stateless*, di conseguenza sviluppabile su *container* Docker.
- **Database module**: Database a cui il modulo di Buisness Logic affida la gestione dello stato; essendo per definizione *stateful*, mal si adatta ai *container* Docker.

Abbiamo quindi in gioco due moduli, di cui uno è naturalmente predisposto allo sviluppo su *container* mentre l'altro no; alla luce di ciò abbiamo pensato di strutturare

il sistema in due parti separate ma tra di loro comunicanti, come mostrato in figura Figura 2

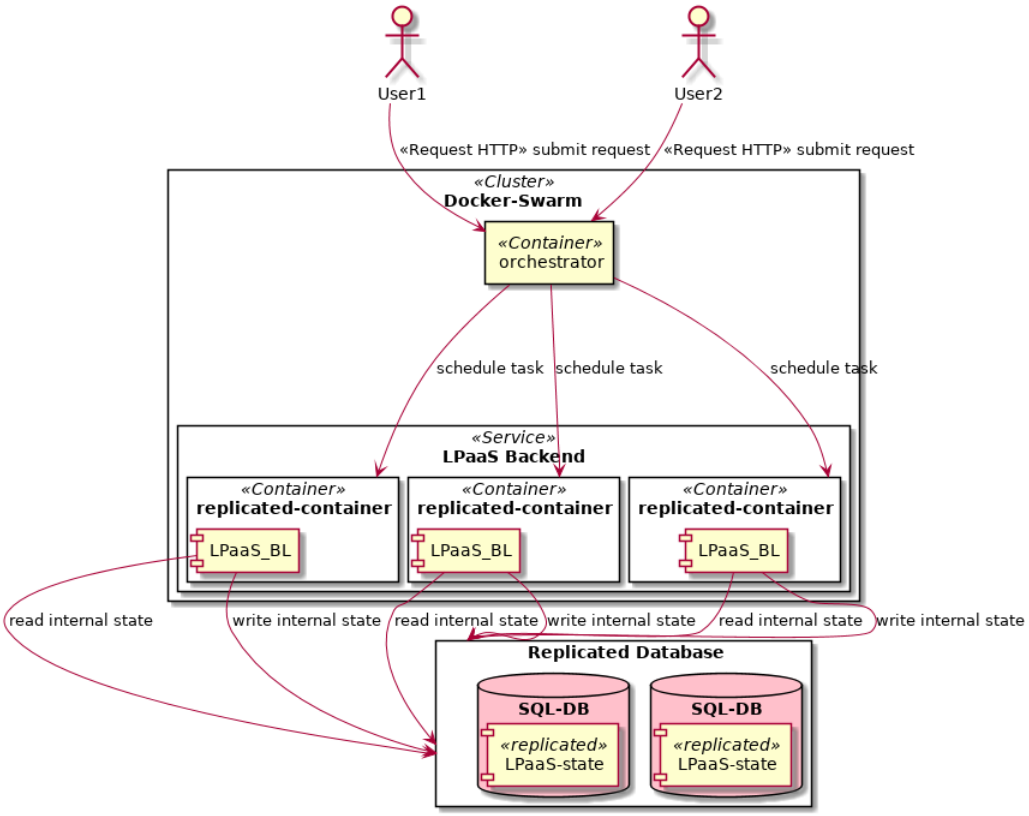

Figura 2: Architettura del sistema LPaaS distribuito

Dal diagramma in figura emerge una panoramica del sistema nella sua fase finale, in quest'architettura i moduli di Buisness Logic e Database sono stati fisicamente separati in due sottosistemi diversi. Per quanto riguarda Buisness Logic, esso viene realizzato su un insieme di *containers* gestiti mediante il tool *Docker Swarm mode*, di cui si tratterà approfonditamente nella sezione successiva; al contrario il modulo di gestione dello stato utilizza tecniche di clustering, replicazione e ridondanza proprie dei database SQL.

3.1.1 Docker Swarm Mode

Per ottenere performance ottimali anche nell'ambito distribuito, Docker mette a disposizione una modalità di funzionamento alternativa, chiamata *Swarm mode*. *Docker swarm mode* [**?**] si compone di tutto un insieme di tecniche per gestire cluster di *container* Docker distribuiti su rete, sfruttando un insieme di comandi e API, inclusi nel *Docker Engine*, che rendono trasparente al programmatore la parte di configurazione e manutenzione dei vari cluster. Le differenze di progettazione e funzionamento della *Swarm*

mode rispetto alla modalità classica sono molto marcate: scopo del *container* non è più ospitare l'applicazione nella sua interezza, ma offrire un microservizio che, combinato con altri servizi di nature differenti in esecuzione su *container differenti*, possa fornire le funzionalità dell'applicazione richiesta, seguendo quindi il paradigma di progettazione a microservizi e i conseguenti vantaggi (scalabilità, replicazione, distribuzione...).

Unità base della *Docker Swarm mode* è il nodo, ovvero un'istanza di *Docker Engine* appositamente configurata per funzionare in *Swarm mode* e collegata a un cluster di altri nodi; è fondamentale che quest'istanza possa accedere alla rete e comunicare attraverso di essa. Ogni nodo poi ha un proprio ruolo: può essere definito come `Manager` qualora si voglia assegnargli il compito di coordinare gli altri nodi del cluster, dividendo tra essi i carichi di lavoro e coordinandosi con altri elementi della stessa natura; tra tutti i `Manager` è presente un nodo `Leader`, che ha il compito di coordinare tra loro i vari `Manager`. Se invece si vuole che il nodo si occupi solo di eseguire i servizi richiesti, allora conviene assegnargli il ruolo di `Worker`, così facendo le risorse del nodo verranno interamente impiegate per la realizzazione di uno o più servizi.

Passando poi al funzionamento dell'insieme dei nodi nel cluster, occorre specificare che la gestione del sistema (o *Orchestration* che dir si voglia) avviene tramite specifiche `API` per etichettare i ruoli dei singoli nodi, uno `scheduler` che si occupa di creare i vari task e di gestirli a seconda dello stato del sistema e infine un `dispatcher`. Il compito di quest'ultimo è mandare in esecuzione sui nodi i task, o microservizi, secondo le istruzioni dello scheduler e sorvegliare l'andamento del sistema tramite una serie di controlli periodici; qualora uno dei controlli non vada a buon fine e un container e/o il nodo su cui è in esecuzione risulti compromesso, il sistema provvede automaticamente a rischedulare il lavoro del componente compromesso su quelli rimasti attivi. Per quanto riguarda infine la resilienza agli errori, ogni *container* (o `task` che dir si voglia) può essere replicato su più nodi all'interno dello *Swarm*; tali repliche sono in tutto e per tutto equivalenti e sono impiegate sia nelle procedure di gestione e ripristino dell'errore di uno o più *container* sia nella distribuzione del carico di lavoro sui nodi effettuata dall'*Orchestrator*. Tale componente si occupa a tempo di esecuzione di smistare le richieste sui nodi del cluster in base al traffico e allo stato di ogni nodo del sistema.

La Figura 3 mostra un esempio di *Swarm* composto da diversi nodi su cui sono in esecuzione diversi *container*, anche detti `task` o microservizi:

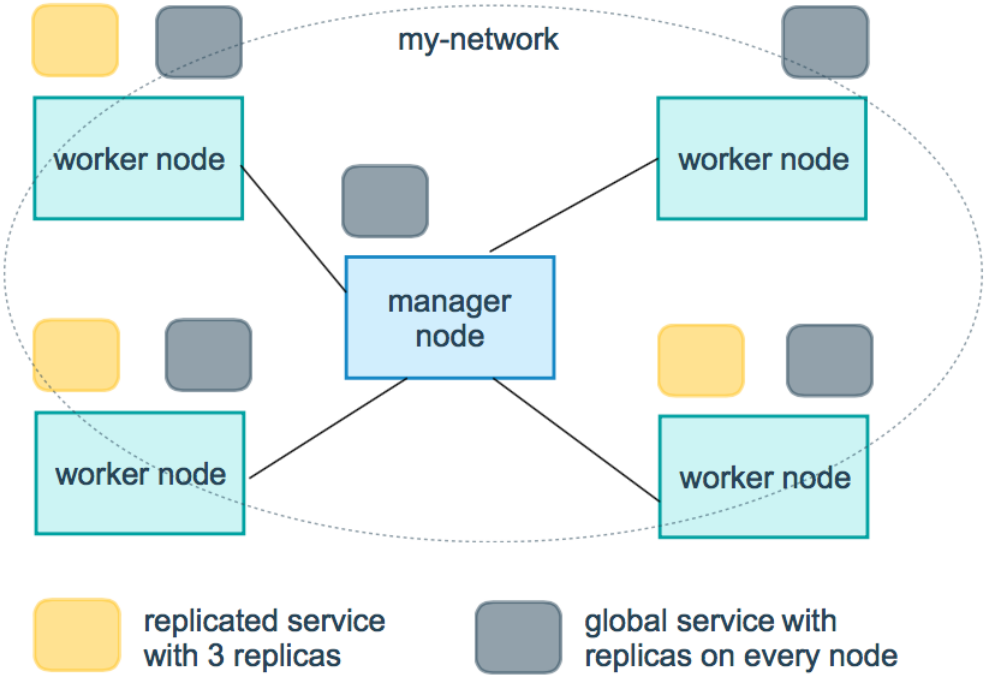

Figura 3: Esempio di Cluster Swarm con microservizi e nodi di natura differente (fonte Link)

3.1.2 Distribuzione dei database SQL

Per quanto riguarda la distribuzione dei database, abbiamo scelto di affidarci alle tecniche di replicazione messe a disposizione dai diversi database SQL. Anche qui, il problema trattato è parecchio complesso e non esiste una soluzione univoca, ma ogni database ha il proprio meccanismo di replicazione e sharding dei dati.

Microsoft SQL è sicuramente uno dei database relazionali più valido in circolazione e anche nell'approccio alla distribuzione mette in campo un'architettura efficace [**?**]: ispirato al design pattern *publish-subscribe*, il modello in questione prevede la presenza di uno o più server *publisher* che distribuiscono tramite un *distributor server* parti di uno stesso database chiamate *publications*; queste ultime non sono altro che raggruppamenti di *articles*, a loro volta collezioni di tabelle, `stored procedure` e viste; destinatari finali delle *publications* sono i server *subscribers* che mantengono una copia locale dei dati in questione. Questa architettura viene poi arricchita dalle tipologie di replicazione [**?**], che permettono di apportare modifiche di natura sostanziale all'architettura sopra descritta, ad esempio abilitando i *subscribers* a poter essere utilizzati come elementi attivi nelle scritture e non solo componenti di sola lettura.

Alternativa *open-source* alla soluzione sopracitata è il database **MySQL**: anche quest'ultimo mette a disposizione diversi modelli di replicazione [**?**], come *master-slave*,

master-master e *Group Replication*; nel caso in cui si scelga **MariaDB**, fork di MySQL, allora è a disposizione anche *Galera Cluster*. Ciascuno di questi modelli si adatta meglio a specifici contesti di distribuzione mentre arranca in altri; in ogni caso tutte le possibilità sopracitate si rifanno a modelli ben noti in letteratura, di conseguenza è più facile la scelta a seconda dello scenario di utilizzo rispetto che su Microsoft SQL e la sua architettura proprietaria.

A prescindere dallo specifico RDBMS utilizzato, è importante evidenziare che la replicazione è trasparente ai client: per quanto possano cambiare le tecniche di replicazione e gestione dei differenti nodi del DB, ciascun client può collegarsi a uno degli host e accedere ai dati come se si trattasse di un'istanza singola. In molti casi è comunque consigliato utilizzare come punto di accesso al cluster di database un *load balancer* che permetta di distribuire gli accessi nel modo più efficiente possibile.

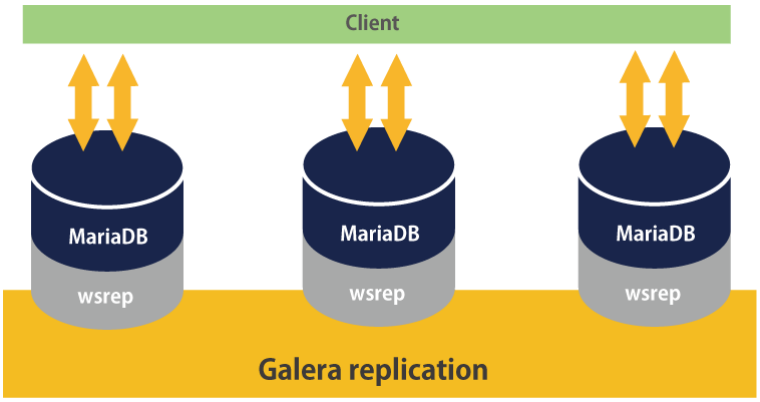

Figura 4: MariaDB Galera Cluster replication

Come è possibile vedere in Figura 4, nel quale è rappresentato il meccanismo di replicazione fornito da Galera Cluster di MariaDB, la replicazione è bidirezionale tra i vari master, di conseguenza ogni nodo consente l'accesso sia in lettura che in scrittura; il processo di replicazione, gestito dalle *write set replication API* (`wsrep`), è completamente trasperente al client, il quale può accedere dove preferisce.

3.2 Comportamento

L'architettura da noi realizzata non ha richiesto nessun cambiamento nel comportamento dei moduli in questione nel sistema LPaaS. Tutti i problemi legati alla natura distribuita del sistema, quali replicazione, distribuzione del carico di lavoro, resilienza e infine consistenza sono a carico delle piattaforme dentro cui i moduli sono inseriti, rimanendo trasparenti a questi ultimi.

3.3 Interazione

Anche qui vale lo stesso discorso fatto per il comportamento del sistema: ai moduli rimane trasparente ogni modifica necessaria a rendere il sistema distributo. Unica eccezione a ciò la necessità di configurare il modulo di Buisness Logic per l'interazione con un database remoto invece che locale, problema che però risulta più di natura implementativa e di conseguenza sarà trattato nella sezione adeguata.

3.3.1 Distribuzione

La pianificazione modulare descritta nella Sezione 3.1 e le tecnologie descritte nelle Sezioni 3.1.1 e 3.1.2 sono integrabili in maniera assolutamente trasparente a LPaaS.

Grazie alla divisione in componenti, i due moduli, Buisness Logic e Database, possono essere distribuiti tramite due piattaforme differenti che tra di loro non hanno nulla in comune.

Queste due piattaforme rimangono però tra loro assolutamente compatibili: all'applicativo in esecuzione su *Docker Swarm mode* è indifferente se il database sia un singolo host, un cluster replicato o un'immagine Docker (non replicata) nel medesimo Swarm; analogamente, al modulo database è ignoto quale replica dell'applicativo LPaaS sta scrivendo o leggendo i dati.

Il sistema quindi può essere definito come *loosely coupled*, rendendolo difatti in linea con i principi di progettazione di un'architettura a microservizi.

3.4 Soluzioni scartate

Prima di giungere a una soluzione funzionante, abbiamo effettuato diverse ricerche e sperimentato varie possibilità: in questa sottosezione tratteremo di quelle che sono state le due maggiori strade inizialmente percorse per la progettazione del nostro sistema e quali sono state le ragioni per cui abbiamo deciso di scartale. Va premesso che l'integrazione tra Docker e un approccio *stateful* è un tema decisamente dibattuto all'interno della community di Docker, di conseguenza non è stato immediato trovare una linea chiara e definita su cosa fosse e non fosse possibile realizzare e soprattutto sul come farlo.

3.4.1 Infinit

I *container* sono trattati come *stateless* da Docker, tuttavia dalle ultime versioni della piattaforma in questione è presente un supporto, se pure non ancora a tutto tondo, della gestione dello stato interno di un *container*; tale tecnologia prende il nome di *Docker Volumes* [**?**].

Il principio di funzionamento è molto semplice: utilizzando il filesystem *UnionFS* già citato si sovrappone una directory esterna a un *container* ad una al suo interno; ogni aggiunta/rimozione/modifica dei files all'interno di questa directory condivisa viene fatta sia all'interno del *container* che della directory esterna.

L'operazione di aggiunta, o *mount*, di uno o più volumi è possibile anche in modalità di esecuzione *swarm*, specificando gli opportuni paramentri di configurazione al lancio del

servizio, in via teorica sarebbe stato possibile quindi mantenere lo stato senza estrarlo dal sistema e delegarlo a un provider di database esterno, rendendo dunque il tutto più compatto e interamente gestito dalla piattaforma Docker.

Tuttavia questa soluzione comporta un problema tutt'altro che banale: nel momento in cui più repliche di una stessa immagine sono in esecuzione all'interno di uno *swarm*, l'*Orchestrator* del cluster decide a tempo di esecuzione a quale replica affidare i task ricevuti sulla base di molteplici fattori. Sta di fatto però che lo stato interno delle repliche non viene in alcun modo condiviso, di conseguenza lo stato interno del sistema non verrà mai reso consistente e per un utente che accede al sistema lo stato in questione sarà quello della replica che servirà la sua richiesta; potrebbe addirittura capitare che uno stesso utente non sia in grado di vedere le modifiche precedentemente apportate da lui al sistema per colpa di un cambio nell'instradamento delle sue richieste ad una differente replica rispetto alla precedente.

La prima soluzione che abbiamo pensato è stata quella di realizzare manualmente la sincronizzazione dello stato mappando tutte le directory contenenti i dati all'interno dei differenti *container* sulla stessa directory esterna; inutile dire però che questa tecnica presentava due grosse problematiche. In primo luogo centralizzando le operazioni di lettura e scrittura in un unica directory si sarebbero venuti a creare enormi colli di bottiglia nell'accesso ai dati e conseguenti cali di performances; successivamente il sistema così configurato sarebbe risultato assolutamente fragile dal punto di vista delle corse critiche nei processi di accesso ai dati, cosa che avrebbe reso lo stato in una continua inconsistenza.

Scartata questa proposta, abbiamo cercato quali potessero essere eventuali soluzioni al problema proposte da utenti della community Docker sicuramente più esperti di noi. Leggendo tra i vari forum, ci siamo imbattuti più volte nel nome di Infinit [**?**], piattaforma di storage software distribuito che permette di creare strutture di memorizzazione flessibili e scalabili, tra cui *volumes* integrabili all'interno dei *container* Docker[10].

Infinit sembrava offrire esattamente la soluzione al problema sopra descritto, tuttavia nel momento in cui abbiamo inziato alla configurazione del *volume* remoto che avrebbe ospitato lo stato, ci siamo imbattuti in diversi problemi a cui non trovavamo soluzione. Abbiamo quindi aperto una *issue*[11] sul repository ufficiale Git di Infinit sperando di risolvere il nostro problema. Il risultato tuttavia è stato differente: il nostro post ha ricevuto diverse risposte di utenti con il nostro stesso problema e la richiesta di una soluzione; analizzando poi la *issue* precedente alla nostra[12] abbiamo dedotto che il progetto Infinit fosse da considerarsi come non più supportato da diversi mesi, rendendo difatti il nostro problema irrisolvibile.

Alla luce di ciò, abbiamo abbandonato l'idea di volumi condivisi e abbiamo verificato altre ipotesi.

[10] `https://blog.docker.com/2017/01/docker-storage-infinit-faq/`
[11] `https://github.com/infinit/infinit/issues/55`
[12] `https://github.com/infinit/infinit/issues/54`

3.4.2 Kubernetes

Piattaforma per distribuzione e scaling di *containers* alternativa a *Docker swarm*, Kubernetes [**?**] poteva essere la tecnologia adatta alla risoluzione del nostro problema. Analizzandone l'architettura e le modalità di funzionamento, abbiamo notato come Kubernetes costituisca uno dei prodotti più completi per la gestione di applicazioni su *containers* distribuiti: in particolare per quanto riguarda l'approccio a servizi *stateful*, Kubernetes matura il concetto di *volumes* presente su Docker adattandolo al contesto distribuito; nella documentazione si parla infatti di *StatefulSets*[13], oggetto che espone API per la gestione di uno stato distribuito.

Nonostante questa soluzione ci sembrasse promettente, abbiamo scelto di non adottarla per via delle poche garanzie che abbiamo trovato sul suo effettivo funzionamento: dopo l'esperienza con Infinit descritta nella sezione sopra non ci sembrava opportuno adottare ulteriori soluzioni di natura sperimentale senza avere prima certezze sul loro funzionamento che andassero oltre le Live Demo tenute alle conferenze dove vengono presentate le tecnologie in questione.

C'è tuttavia anche un'ulteriore ragione all'accantonamento di questa soluzione, di natura più teorico/concettuale: dal momento in cui abbiamo iniziato a lavorare al progetto alla stestura di questa relazione è trascorso circa un anno e mezzo, tempo in cui abbiamo avuto modo di approfondire le nostre conoscenze nell'ambito della progettazione di architetture distribuite e delle basi teoriche che vi stanno dietro. Alla luce di ciò e della nostra analaisi su LPaaS, abbiamo ritenuto opportuno non concentrare tutto il sistema in un solo modello come avevamo pensato all'inizio, ma al contrario riconoscere che all'interno del progetto esistevano moduli diversi che dovevano essere trattati con modalità e tecniche differenti tra loro.

[13] `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/?spm=a2c65.11461447.0.0.18076370KGUqNH`

4 Dettagli di implementazione

4.1 Configurazione della persistenza

Come detto anche nella Sezione 3, LPaaS inizialmente si appoggiava sull'istanza interna a Payara di un database SQL basato su Derby; è stato dunque necessario configurare Payara per l'utilizzo di un livello di persistenza esterno ed alternativo a quello integrato.

Punto focale dei container è la portabilità delle applicazioni che incapsulano: è importante infatti che il container, al netto di eventuali configurazioni specificabili al lancio, possa essere eseguito indipendentemente da quanto installato nell'ambiente di esecuzione.

Uno dei problemi riscontrati durante la fase di implementazione è stato proprio come incapsulare tutta la procedura di configurazione sulla persistenza dei dati a livello di codice, lasciando all'utente esclusivamente l'onere di specificare credenziali ed *endpoint*. Nelle fasi iniziali del progetto, infatti, facendo riferimento alla documentazione ufficiale di Payara [**?**] si era modificato il campo `jta-data-source` del file `persistence.xml` facendo riferimento ad un *connection pool* configurato manualmente dall'interfaccia web di Payara Server; questa soluzione non era ovviamente adeguata, in quanto la configurazione manuale è inaccettabile nello scenario distribuito da realizzare.

Il passo successivo è stato identificare quali fossero le modalità di configurazione alternative alla WebGUI. La prima prevedeva l'impiego del comando `asadmin` con le opportune opzioni: la soluzione risultava accettabile per l'uso in un container Docker, ma rimaneva non immediata per un utilizzo locale all'ambiente di sviluppo. La seconda possibilità riguardava invece l'utilizzo di file di configurazione XML: `glassfish-resources.xml` o, nello specifico, `payara-resources.xml`.

```
<jdbc-resource  pool-name="LPaaSDB"
                jndi-name="java:app/jdbc/LPaaS"
                enabled="true"/>
```

Listing 1: `payara-resources.xml`: *Resource pool*

```
<jdbc-connection-pool datasource-classname="org.mariadb.jdbc.
    MariaDbDataSource"
                      is-connection-validation-required="false"
                      name="LPaaSDB"
                      res-type="javax.sql.DataSource">
    <property name="User"
              value="${ENV=LPAAS_DATABASE_USER}"/>
    <property name="Password"
              value="${ENV=LPAAS_DATABASE_PASS}"/>
    <property name="DatabaseName"
              value="${ENV=LPAAS_DATABASE_NAME}"/>
    <property name="ServerName"
              value="${ENV=LPAAS_DATABASE_SERVER}"/>
    <property name="PortNumber"
              value="${ENV=LPAAS_DATABASE_PORT}"/>
</jdbc-connection-pool>
```

Listing 2: `payara-resources.xml`: *Connection pool*

Il file di configurazione definisce una risorsa JDBC chiamata `java:app/jdbc/LPaaS` (Listing 1) e un *connection pool* (Listing 2) specifico che si appoggia al driver ufficiale di MariaDB. I valori supportano diverse opzioni di templating per l'impiego di variabili e configurazioni; si è scelto di utilizzare le variabili d'ambiente (specificatore `ENV`) in quanto più semplici da mettere a disposizione in qualsiasi contesto di esecuzione.

Nel file `persistence.xml`, riportato in Listing 3, si fa riferimento al *connection pool* e si definisce la *persistence unit* `LPaaSPU` che è poi utilizzata nel codice.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                         http://java.sun.com/xml/ns/persistence/
                             persistence_2_0.xsd">
    <persistence-unit name="LPaaSPU" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</
            provider>
        <jta-data-source>java:app/jdbc/LPaaS</jta-data-source>
        <exclude-unlisted-classes>false</exclude-unlisted-classes>
        <properties>
            <property name="eclipselink.target-database"
                      value="MySQL"/>
            <property name="eclipselink.validation-only"
                      value="false"/>
            <property name="eclipselink.ddl-generation"
                      value="create-or-extend-tables"/>
            <property name="eclipselink.create-ddl-jdbc-file-name"
                      value="createDDL_ddlGeneration.jdbc"/>
            <property name="eclipselink.drop-ddl-jdbc-file-name"
                      value="dropDDL_ddlGeneration.jdbc"/>
            <property name="eclipselink.ddl-generation.output-mode"
                      value="both"/>
        </properties>
    </persistence-unit>
</persistence>
```

Listing 3: `persistence.xml`

4.2 Autenticazione

Nel testare il sistema, ci siamo resi conto che, su più istanze distribuite dell'applicazione, si verificava un comportamento strano: infatti, il token per l'autenticazione necessario per l'accesso ad alcune risorse veniva a volte rifiutato dal sistema senza nessuna apparente motivazione.

Analizzando l'implementazione di LPaaS, ci siamo resi conto che le chiavi necessarie alla generazione, cifratura e decifratura dei token venivano generate ogni volta all'avvio. Questa implemetazione non era compatibile con il meccanismo di replicazione messo in campo da Docker, nel quale le repliche differenti avrebbero generato chiavi differenti e non compatibili l'una con l'altra; di conseguenza abbiamo scelto di modificare LPaaS

in modo da caricare le suddette chiavi da un file all'interno della folder `resources` del progetto. Tale file sarebbe risultato univoco tra le repliche, facendo sì che queste utilizzassero le stesse chiavi e di conseguenza risolvendo il problema.

Oltre a ciò abbiamo aggiunto un task Gradle per la generazione delle chiavi e il loro salvataggio nel file sopracitato, in modo da conservare la possibilità di rigenerarle automaticamente in caso di bisogno.

4.3 Dockerfile

La realizzazione di immagini Docker in grado eseguire LPaaS con un database stabilmente accessibile è stata abbastanza semplice, merito anche della documentazione ben fatta delle immagini Docker ufficiali di Payara.

In particolare, di seguito è riportato il codice del relativo `Dockerfile`:

```
FROM payara/micro

COPY lpaas/build/libs/lpaas.war $DEPLOY_DIR
```

Listing 4: `Dockerfile` dell'immagine `tirocigno/lpaas-mysql:latest`

Come descritto anche nella documentazione ufficiale[14], è sufficiente inserire il file `war` nella cartella di deploy perché questo venga caricato automaticamente da un'istanza di Payara Micro.

Discorso più complicato riguarda un contesto di esecuzione nel quale sia necessario attendere un database lanciato in contemporanea con LPaaS, come ad esempio il caso descritto nella Sezione 7; in questo caso, è stato necessario aggiungere una variante dell'immagine, accessibile aggiungendo il tag "`:safe`" all'immagine descritta sopra, che esegue uno script che attende che il database sia effettivamente accessibile prima di lanciare Payara Micro e dunque LPaaS.

[14] `https://github.com/payara/docker-payaramicro`

5 Validazione e autovalutazione

5.1 Criteri di testing

Data la natura sperimentale del nostro progetto, è sicuramente difficile definire criteri di valutazione rigorosamente formali. Abbiamo ragionato a lungo quindi su quali e quanti potessero essere i metri di valutazione idonei, giungendo alla conclusione che la risposta fosse da ricercare all'interno delle fasi del ciclo di vita di un progetto software. Non avendo avuto modo nel nostro caso di avere a disposizione un framework di supporto al testing, ma dovendo realizzare ogni interazione a mano, abbiamo pensato di includere all'interno dei nostri criteri non solo l'immediato testing successivo alla realizzazione del prodotto, ma anche le fasi di manutenzione evolutiva e adattiva.

Questa scelta può essere motivata sulla base del seguente ragionamento: non avendo al momento la possibilità né di testare il sistema in condizioni adeguate di distribuizione né di realizzare test automatici e replicabili per via della mancanza di un framework adeguato, è errato assumere che un domani questa possibilità non si concretizzi; nel caso in cui succeda bisognerà che il sistema si possa adattare alla situazione senza bisogno di stravolgere la propria struttura.

Alla luce di quanto detto, abbiamo individuato i seguenti criteri:

- **Aderenza ai requisiti in fase di analisi**: criterio fondamentale in ogni progetto, non necessita di particolari introduzioni. Nel caso del nostro progetto, i requisiti fissati in fase di analisi, presentati nella Sezione 2.3, sono stati soddisfatti; il sistema da questo punto di vista è stato testato manualmente, sia in un contesto concentrato tramite *docker-compose* che in un contesto distribuito virtuallizzato.

- **Manutenzione evolutiva** Avendo scomposto il sistema in due moduli, ciascuno può essere manipolato indipendentemente dall'altro, ciò permette di effettuare operazioni di manutenzione evolutiva in maniera sicura, scalabile, veloce e resistente agli errori.

- **Manutenzione adattiva** Qualora si abbia bisogno di adattare il sistema ad un contesto differente, ad esempio uno scenario dove la velocità e disponibilità dei dati abbiano la precedenza sulla loro consistenza, la separazione in moduli e la flessibilità delle impostazioni di *Docker swarm* rende possibile la realizzazione di diverse soluzioni adatte a differenti scenari.

5.2 Test realizzati

Come affermato nella Sezione 5.1, non ci è stato possibile utilizzare un framework di testing durante il nostro lavoro, ma ciò non implica che i test non siano stati condotti. Abbiamo utilizzato ancora una volta Docker per aiutarci a rendere il processo di testing il più automatico, indipendente e replicabile possibile: grazie al modulo *docker-compose*, che permette di creare stack di più *containers* docker comunicanti tra loro, abbiamo

realizzato un semplice stack (la cui spiegazione dettagliata viene presentata nella Sezione 7) contenente due *containers*, uno eseguiva LPaaS, l'altro metteva a disposizione un database SQL di backend, MariaDB in questo caso.

L'applicazione così realizzata presentava tutti i vantaggi del deploy su *container*: leggerezza, facilità di utilizzo e infine approccio *stateless*. Proprio quest'ultimo punto che era stato un grande svantaggio in fase di design, ora diventa una delle più interessanti feature del sistema: grazie a ciò infatti è davvero facile mettere a disposizione di ogni test un ambiente pulito e isolato, in cui non c'è alcuna traccia delle modifiche fatte nei test precedenti.

Questa applicazione è stata utilizzata nella maggior parte dei test volti a verificare che la separazione tra Buisness Logic e stato all'interno di LPaaS potesse avvenire senza problema.

Per quanto riguarda invece i test relativi alla distribuzione, abbiamo costruito un ambiente virtualizzato su hypervisor Proxmox, con tre macchine virtuali Ubuntu connesse in Swarm. Non sono state messe in pratica tecniche di replicazione sul database, bensì è stata utilizzata una singola macchina virtuale Ubuntu con una singola istanza di MariaDB in esecuzione; come detto anche nelle Sezioni 3.1.2 e 3.3.1, aggiungere una delle tecniche di replicazione di riferimento di MariaDB, come ad esempio Galera Cluster, non costituirebbe un problema per il codice di LPaaS.

6 Istruzioni per il deployment

Il repository sul quale questo progetto è andato ad appoggiarsi poteva essere importato come progetto nelle IDE Eclipse e Intellij IDEA indifferentemente e mettava a disposizione buildscript sia per Maven che per Gradle. Poiché la configurazione Gradle risultava notevolmente più aggiornata di quella Maven ed essendo Gradle maggiormente integrato con IDE moderni, si è scelto di appoggiarsi esclusivamente a Gradle per le operazioni di costruzione e gestione delle dipendenze; inoltre, come IDE di riferimento è stato scelto Intellij IDEA.

Per il corretto funzionamento di Gradle, è richiesta una versione del JDK installata, mentre non è richiesta (anche se consigliata) l'installazione di Gradle, in quanto presenti i wrapper per ambienti Windows e Unix. Per l'esecuzione delle immagini Docker, l'*engine* deve essere installato localmente o disponibile su una macchina remota.

6.1 Generazione del deployment file in formato `war`

Per la compilazione del codice a partire dai sorgenti, il task `war` messo a disposizione da Gradle è sufficiente per la generazione del file `lpaas.war` nella cartella `build/libs/`; tale file può essere utilizzato per il deploy manuale su un'istanza di Glassfish/Payara desiderata.

6.2 Deployment tramite Payara Micro in formato `jar`

Per l'esecuzione del file `war` tramite Payara Micro in formato `jar` eseguibile:

1. scaricare il file `jar` dal sito ufficiale: `https://www.payara.fish/software/downloads/`;
2. assicurarsi di avere un database **mySQL** o **MariaDB** in esecuzione su un host accessibile e con un database dedicato già creato;
3. eseguire il progetto con `./gradlew run -Pconf=<username>,<password>,<db name>,<db host>,<db port>`.

6.3 Deployment tramite Intellij IDEA Ultimate

Per l'esecuzione del progetto tramite IDE:

1. installare Payara Server Full e assicurarsi di avere un database **mySQL** o **MariaDB** in esecuzione su un host accessibile e con un database dedicato già creato;
2. importare il progetto in Intellij IDEA Ultimate come progetto Gradle;
3. accedere alla schermata di modifica delle *run configurations* tramite il tasto *Add Configurations...* in alto a sinistra;
4. selezionare il tasto `+` e selezionare *GlassFish Server* > *Local*;

5. configurare l'*Application_server* con la propria istanza di Payara installata;
6. impostare l'URL a `http://localhost:8080/lpaas/theories/default`;
7. selezionare `domain1` nel campo *Server_Domain*;
8. sotto la tab *Deployment*, selezionare il tasto `+` e selezionare *Artifact...* e poi *Gradle: it.unibo.alice: lpaas.war*;
9. sotto la tab *Startup/Connection*, aggiungere le variabili d'ambiente seguenti:
   - `LPAAS_DATABASE_USER` con l'utente del database;
   - `LPAAS_DATABASE_PASS` con la password per l'utente scelto;
   - `LPAAS_DATABASE_NAME` con il nome del database;
   - `LPAAS_DATABASE_SERVER` con l'hostname del server del database;
   - `LPAAS_DATABASE_PORT` con la porta del server del database;
10. una volta applicato, è possibile lanciare il deploy locale tramite il tasto dedicato in alto a sinistra.

6.4 Deployment tramite immagine Docker in singola istanza

LPaaS è disponibile sul Docker Hub con il nome `tirocigno/lpaas-mysql`. È comunque possibile compilare l'immagine manualmente secondo le istruzioni presenti nella descrizione del repository.

Per lanciare l'applicazione, è sufficiente eseguire:

```
docker run \
  -e LPAAS_DATABASE_USER=your_username \
  -e LPAAS_DATABASE_PASS=your_password \
  -e LPAAS_DATABASE_NAME=your_db_name \
  -e LPAAS_DATABASE_SERVER=your_db_ip \
  -e LPAAS_DATABASE_PORT=your_db_port \
  -p your_wanted_exposed_port:8080 \
  tirocigno/lpaas-mysql:latest
```

6.5 Deployment tramite immagine Docker su Swarm

Per lanciare l'applicazione, è sufficiente eseguire la versione descritta nella Sezione 6.4:

```
docker service create \
  --name LPaaSMySQLSwarm \
  --publish published=EXPOSED_OUTDOOR_PORT,target=
     LPAAS_EXPOSED_PORT \
  --replicas number_of_replicas
  -e LPAAS_DATABASE_USER=selected_username \
  -e LPAAS_DATABASE_PASS=selected_password \
  -e LPAAS_DATABASE_NAME=selected_db \
  -e LPAAS_DATABASE_SERVER=remote_mysql_server_ip \
  -e LPAAS_DATABASE_PORT=remote_mysql_server_port \
  --hostname=localhost \
    tirocigno/lpaas-mysql:latest
```

`--hostname=localhost` è necessario per la risoluzione degli indirizzi.

Ogni connessione a uno Swarm locale dovrebbe essere fatta riferendosi a `127.0.0.1`, in quanto `localhost` non è supportato da Docker Swarm.

7 Esempi di utilizzo

Nelle Sezioni 6.4 e 6.5 si è descritto diverse modalità di deploy tramite Docker; tali implementazioni sono quelle suggerite per un ambiente di produzione, nel quale il database è remoto e replicato separatamente. A scopo di test, è possibile utilizzare `docker-compose` per effettuare il deploy di entrambe le componenti necessarie all'applicazione: l'immagine di LPaaS su Payara e l'immagine del database di riferimento.

In questa sezione si riporta un esempio di implementazione di questo tipo:

```yaml
version: "3.7"
services:
  db:
    image: mariadb
    env_file:
      - "payara.env"
    ports:
      - "3306:3306"
  lpaas:
    image: tirocigno/lpaas-mysql:safe
    env_file:
      - "payara.env"
    ports:
      - "8080:8080"
    depends_on:
      - db
```

Listing 5: `docker-compose.yml`

```
MYSQL_ROOT_PASSWORD=password
MYSQL_DATABASE=test
MYSQL_USER=username
MYSQL_PASSWORD=password
LPAAS_DATABASE_USER=username
LPAAS_DATABASE_PASS=password
LPAAS_DATABASE_NAME=test
LPAAS_DATABASE_SERVER=db
LPAAS_DATABASE_PORT=3306
```

Listing 6: `payara.env`

8 Conclusioni

Per concludere, si riassumono i principali *job* portati a termine durante lo svolgimento del progetto:

- **Modularizzazione di LPaaS**. Il sistema LPaaS è stato suddiviso in due sottomoduli trattati come due servizi separati: uno riguardante la Buisness Logic di LPaaS che ha per la maggior parte conservato l'implementazione corrente e l'altro consistente in un database backend responsabile di mantenere lo stato del sistema.
- **Integrazione con Docker**. È stato aggiunto a LPaaS il supporto al deploy su *container* Docker, rendendo disponibile un immagine sul repository ufficiale Dockerhub[15]
- **Distribuzione via Docker Swarm mode**. LPaaS è stato distribuito su un cluster via *Docker Swarm mode* e testato nel suo funzionamento, utilizzando una singola istanza remota di MariaDB per la gestione dello stato interno.

8.1 Lavori futuri

Il nostro lavoro si pone come punto di partenza per diverse possibili modifiche.

In primo luogo noi abbiamo scelto di non alterare l'implementazione della Buisness Logic di LPaaS; al fine di creare un sistema orientato ai microservizi si potrebbe pensare di suddividere questo modulo in ulteriori servizi, così da rendere più efficace la sua distribuzione.

Successivamente, qualora la tecnologia per la gestione delle applicazioni *stateful* introdotta da Kubernetes dovesse risultare abbastanza matura, si potrebbe valutare di reintegrare lo stato all'interno della stessa piattaforma usata per la Buisness Logic, tuttavia questa strada va percorsa con cautela per evitare di incorrere negli stessi problemi da noi riscontrati con Infinit, senza considerare poi che questa scelta contravverrebbe, almeno in linea teorica, ai principi di buona progettazione riguardanti la *Separation of Concerns*.

Infine vale la pena far presente che i test da noi realizzati sono stati fatti in un ambiente controllato per via della mancanza delle risorse necessarie; potrebbe dunque essere opportuno mettere in funzione il sistema in un contesto veramente distribuito così da mettere il nostro modello alla prova e vedere quali possono essere i risultati effettivi ottenuti dalla nostra architettura e quali gli spunti per migliorarla.

8.2 Che cosa abbiamo imparato

La nostra esperienza con questo progetto può considerarsi assolutamente positiva.

Sicuramente la natura più di ricerca che implementativa dell'elaborato ci ha spinti a porci in modo differente rispetto a quanto fatto in altri esami, soprattutto durante le fasi di analisi e design. Più di una volta ci siamo chiesti se una determinata soluzione fosse

[15] `https://cloud.docker.com/u/tirocigno/repository/docker/tirocigno/lpaas-mysql`

innanzitutto plausibile e corretta rispetto a quanto puntavamo a realizzare e solo dopo aver risposto affermativamente a questa domanda ci concentravamo sul come implementarla; al contrario, nella maggior parte dei progetti ci siamo trovati nella situazione di dover pensare solo al come implementare una determinata soluzione piuttosto che sul comprendere appieno se quella soluzione fosse davvero ottimale ai fini dell'elaborato da realizzare.

In secondo luogo ci sentiamo di affermare che durante lo svolgimento del progetto abbiamo avuto modo di vedere nuove tecnologie e soprattutto di costruire pian piano un occhio critico verso ciò che trovavamo sul web: se nel momento in cui abbiamo iniziato ci fidavamo cecamente delle prime soluzioni che trovavamo proposte da utenti della community Docker, a fine dei lavori ci riteniamo in grado di poter esprimere un nostro parere su ciò che ci troviamo davanti, grazie alla conoscenza e esperienza maturata in questo progetto.

Infine vale la pena sottolineare che questo progetto ci ha dato modo di interfacciarci e scontrarci con il mondo dei sistemi distribuiti; abbiamo quindi avuto la possibilità di riflettere di problemi come consistenza, scalabilità e resilienza in chiave molto più pratica che teorica.

Riferimenti bibliografici

[1] Docker engine overview - docker documentation. https://docs.docker.com/engine/docker-overview/#docker-engine. Accessed on 09/23/2019.

[2] Enterprise container platform docker. https://www.docker.com/. Accessed on 09/23/2019.

[3] Infinit - decentralized software-based file storage platform. https://infinit.sh/. Accessed on 09/24/2019.

[4] Java™ EE at a glance. https://www.oracle.com/java/technologies/java-ee-glance.html. Accessed: 2019-09-22.

[5] Panoramica del modello di pubblicazione della replica - sql server - microsoft docs. https://docs.microsoft.com/it-it/sql/relational-databases/replication/publish/replication-publishing-model-overview?view=sql-server-2017. Accessed on 09/24/2019.

[6] Payara server documentation. https://docs.payara.fish/documentation/payara-server/. Accessed: 2019-09-22.

[7] Production-grade container orchestration - kubernetes. https://kubernetes.io/. Accessed on 09/25/2019.

[8] Pros and cons of mysql replication types - dzone database. https://dzone.com/articles/pros-and-cons-of-mysql-replication-types. Accessed on 09/24/2019.

[9] Swarm mode overview - docker documentation. https://docs.docker.com/engine/swarm/. Accessed on 09/23/2019.

[10] Tipi di replica - sql server - microsoft docs. https://docs.microsoft.com/it-it/sql/relational-databases/replication/types-of-replication?view=sql-server-2017. Accessed on 09/24/2019.

[11] Use volumes - docker documentation. https://docs.docker.com/storage/volumes/. Accessed on 09/24/2019.

[12] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifiers (uri): Generic syntax. RFC 2396, RFC Editor, August 1998. http://www.rfc-editor.org/rfc/rfc2396.txt.

[13] Roberta Calegari, Enrico Denti, Stefano Mariani, and Andrea Omicini. Towards logic programming as a service: Experiments in tuProlog. In Corrado Santoro, Fabrizio Messina, and Massimiliano De Benedetti, editors, *WOA 2016 – 17th Workshop "From Objects to Agents"*, volume 1664 of *CEUR Workshop Proceedings*, pages 91–99. Sun SITE Central Europe, RWTH Aachen University, 29–30 July 2016. Proceedings of the 17th Workshop "From Objects to Agents" co-located with 18th European Agent Systems Summer School (EASSS 2016).

[14] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin Heidelberg, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[15] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000.