O2.2

Prototipo di piattaforma cloud per il supporto delle funzionalità AI/ML per big data advanced analytics su larga scala

Code	O2.2
Data	21/07/2020
Type	Confidential
Participants	CIRI-ICT
Authors	Lorenzo Civolani (UNIBO)
Corresponding authors	Luca Foschini

SBDIO I4.0
Big Data
for Industry

Indice

SBDIO I4.0
Big Data
for Industry

Introduzione

Questo documento descrive le attività svolte e i risultati raggiunti dal CIRI-ICT nell'ambito del progetto SBDIOI40 relativamente allo sviluppo della piattaforma cloud destinata ad ospitare applicazioni intelligenti per l'industria digitale. Nel progetto SBDIOI40, le applicazioni per l'industria digitale sono strutturate secondo un'architettura a multiservizi, e i servizi possono essere realizzati mediante macchine virtuali gestite tramite l'interfaccia di OpenStack, secondo un approccio più consolidato, oppure attraverso l'incapsulamento all'interno di contenitori software (anche noti come *container*) gestiti con Docker. Il lavoro svolto dal CIRI-ICT in questa fase ha dunque affrontato il prototipo della piattaforma cloud mediante lo studio e l'implementazione di soluzioni utili da entrambi i punti di vista.

- Per quanto riguarda le applicazioni composte da macchine virtuali, si è elaborata una soluzione per la migrazione di applicazioni tra due piattaforme OpenStack pertinenti al progetto SBDIOI40.
- Per quanto concerne le applicazioni implementate a container, si è lavorato a un sistema per la velocizzazione dei tempi di avvio di applicazioni a microservizi Docker Compose. Velocizzando l'avvio, è quindi possibile anche ridurre il tempo necessario per la migrazione di un'applicazione a microservizi tra due piattaforme Docker.

La restante parte di questo documento è organizzata intorno a due sezioni principali che presentano le soluzioni menzionate, discutendone rispettivamente l'idea, i criteri che ne hanno guidato lo sviluppo, le scelte effettuate, i punti salienti dell'implementazione ed infine alcune prospettive per estensioni future.

Migrazione di applicazioni OpenStack

All'interno di SBDIOI40, la piattaforma cloud federata che supporta l'esecuzione delle applicazioni intelligenti è verosimilmente costituita da molteplici sistemi OpenStack, ciascuno sotto la supervisione di un partner del progetto. Considerata questa caratteristica distribuita della piattaforma cloud, diventa importante disporre di un sistema in grado di spostare (o migrare) un'applicazione da un terminale all'altro dell'infrastruttura. L'obiettivo di questo primo lavoro è quello di realizzare un sistema che semplifichi tale trasferimento.

Dopo un breve richiamo a OpenStack, analizzeremo un caso d'uso specifico di applicazione SBDIOI40 su OpenStack. A partire da tale esempio, verranno stabilite alcune convenzioni circa la struttura delle applicazioni e dei rispettivi componenti. Poi, si giustificheranno le scelte implementative e si mostreranno i punti più significativi del codice. Infine, si documenterà l'esperienza d'uso dello strumento durante la migrazione di un'applicazione di prova.

OpenStack

OpenStack[1] è una piattaforma software per il cloud computing che ha l'obiettivo di offrire all'utente un punto di accesso per la gestione e l'uso di risorse virtuali. Le risorse virtuali gestibili tramite la piattaforma includono le macchine, reti e router virtuali. OpenStack non è un software monolitico, ma consiste piuttosto in una serie di moduli (principalmente scritti in Python) che lavorano in modo coordinato e che possono controllare un bacino hardware eterogeneo, come ad esempio un cluster di computer convenzionali. Grazie ai servizi offerti dalla piattaforma, uno sviluppatore può assegnare, revocare e utilizzare risorse virtuali senza l'intermediazione di un amministratore di sistema.

Gli utilizzatori della piattaforma hanno essenzialmente 3 possibilità per accedere alle risorse:

1. utilizzare la *dashboard*, un pannello di controllo semplificato che è accessibile con un semplice browser web;
2. impartire comandi testuali tramite l'interfaccia a riga di comando (CLI);
3. inviare richieste alle API REST esposte dai singoli servizi, meccanismo che abilita l'interazione anche da parte di un software.

Applicazioni SBDIOI40 su OpenStack

Internamente al progetto SBDIOI40, la piattaforma OpenStack viene utilizzata per ospitare le applicazioni di interesse per l'industria digitale. Come primo passo verso la migrazione di un'applicazione, occorre definire che cosa si intende, nell'ambito di questo lavoro, con il termine "applicazione". Si è scelto di cominciare attraverso l'analisi di un caso d'uso concreto già implementato da parte di UNIFE e T3LAB: l'applicazione per Carpigiani. Come mostrato nei Deliverable O1.1 e O5.1, l'applicazione per Carpigiani ha l'obiettivo di costituire uno

[1] OpenStack, https://www.openstack.org/

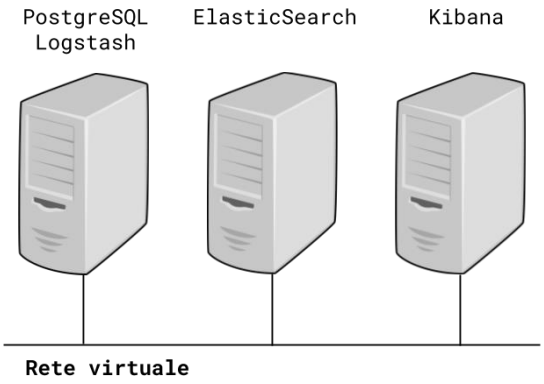
strumento flessibile per l'analisi dei dati raccolti dalle macchine per gelato. In particolare, l'applicazione è realizzata tramite l'uso dei seguenti software:

- un database PostgreSQL
- uno stack ELK, composto a sua volta da
  - Logstash
  - Elasticsearch
  - Kibana

Nonostante l'applicazione faccia uso di 4 diversi software, complessivamente, e per ragioni che esulano dallo scopo di questo documento, si è scelto di organizzare l'applicazione in tre macchine virtuali:

- una prima VM ospita il database PostgreSQL e Logstash
- una seconda VM ospita Elasticsearch
- una terza VM ospita la dashboard di Kibana

Le tre VM che compongono l'applicazione per Carpigiani sono collegate ad una stessa rete virtuale che consente la comunicazione interna fra i vari componenti software. Inoltre, la VM che ospita il database e quella che ospita la dashboard sono configurate in modo da avere accesso alla rete pubblica, rispettivamente per consentire il caricamento dei dati grezzi e l'accesso al pannello di visualizzazione.



Convenzioni e denominazione dei componenti

Il primo passo verso il progetto del sistema di migrazione attraverso è stato la scelta di alcune convenzioni sulla configurazione delle piattaforme OpenStack e sulla struttura delle applicazioni SBDIOI40 in esse contenute. Come si vedrà nel seguito, per la denominazione dei componenti si è preso spunto dal caso Carpigiani.

Stabilire queste convenzioni ha comportato molteplici vantaggi. In primo luogo, si è semplificato lo sviluppo del software, dal momento che i componenti virtuali delle applicazioni sono reperibili più facilmente rispetto al caso in cui i suoi nomi siano scelti arbitrariamente. Inoltre, il software finale risulta più semplice anche da utilizzare: se i componenti fossero chiamati in modo arbitrario, all'atto di ciascuna invocazione l'utente dovrebbe indicare esplicitamente la struttura dell'applicazione tramite parametri, file di configurazione o variabili d'ambiente, cosa invece non necessaria se il nome dei componenti è deducibile. In ultimo, risulta immediato riconoscere e gestire le risorse virtuali anche manualmente, perché le regole di denominazione contribuiscono a produrre un ambiente di lavoro "ordinato".

La prima delle convenzioni stabilite riguarda il **dominio** di lavoro. In OpenStack, un dominio è un contenitore astratto e di alto livello per tutti i tipi di risorse che è possibile creare e gestire all'interno della piattaforma. All'interno di un dominio è possibile creare utenti, gruppi e progetti, ed assegnare autorizzazioni di accesso e di modifica. I domini sono uno strumento che permette di suddividere l'ambiente di computazione in partizioni logiche ciascuna assegnata, ad esempio, a un'organizzazione o a un dipartimento. Per semplicità, supponiamo che nelle piattaforme OpenStack del progetto SBDIOI40 esista sempre e ovunque un solo dominio: "Default". Generalmente, questo dominio esiste già per preimpostazione ed è l'unico utilizzato.

ID	default
Name	Default
Enabled	true
Description	The default domain

All'interno di un dominio convivono generalmente diversi progetti. Un **progetto** è un insieme isolato di risorse virtuali gestite da OpenStack. Tra le risorse virtuali comprese in un progetto figurano le VM, le immagini per la loro creazione (ad es. snapshot), le reti, i router e così via. Il progetto è l'unità base di appartenenza, perché, in OpenStack, ogni risorsa appartiene a un progetto. Per SBDIOI40, supponiamo che su ogni piattaforma esista un progetto destinato a contenere tutte le applicazioni per l'industria digitale, in modo da costituire un ambiente separato rispetto ad altri eventuali usi della piattaforma.

Name	sbdioi40
Domain Name	Default
Domain ID	default
Description	Servizi per l'industria digitale

Per gestire le applicazioni SBDIOI40, è necessario che esista un **utente** con diritti di amministratore sul relativo progetto. Supponiamo che su ciascuna piattaforma esista almeno un utente con tali prerogative.

Con il termine **applicazione** si intende un insieme di risorse virtuali che realizzano un insieme di funzionalità destinate a un partner SBDIOI40. Il caso Carpigiani rappresenta un esempio di applicazione e, nell'ambito di questo lavoro, è stata scelta come modello per la definizione delle caratteristiche generali.

In generale, un'applicazione è scomposta in N servizi secondo un'architettura a multiservizi. Più in dettaglio, un'applicazione è costituita da:

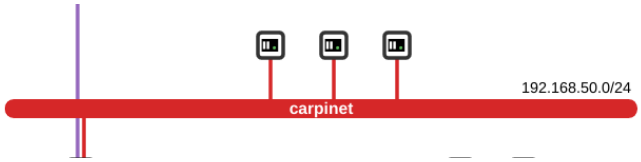
- una rete virtuale
- N macchine virtuali (una per ogni servizio)
- N porte, ciascuna che collega una macchina virtuale alla rete

Gestire le porte in modo esplicito, piuttosto che delegare tale responsabilità agli automatismi di OpenStack, è fondamentale per essere in grado di assegnare un indirizzo IP statico a ciascuna VM, circostanza verosimilmente comune nella configurazione di applicazioni a multiservizi.

La maggior parte delle applicazioni SBDIOI40 hanno verosimilmente la necessità accedere a Internet. Per sopperire a questo requisito, all'interno del progetto sbdioi40 sarà presente un **router** virtuale di nome "router" che, essendo collegato alle reti virtuali di ciascuna applicazione, funge da punto di accesso condiviso verso la rete pubblica.

Name	router
Description	Router per le applicazioni SBDIOI40
Status	Active
Admin State	UP

Il diagramma seguente mostra come si presenta una piattaforma SBDIOI40 che ospita tre applicazioni: "test", "carpi" e "sacmi".

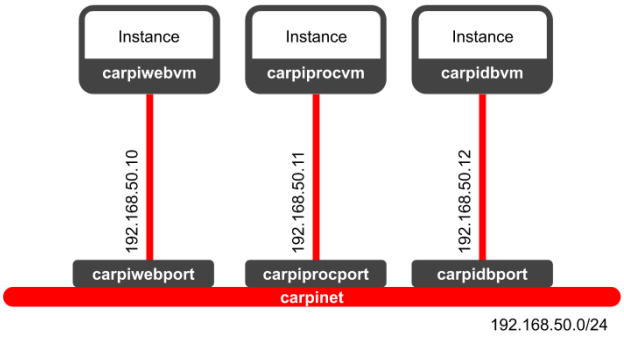


Sono state definite delle convenzioni circa la denominazione dei componenti virtuali di ciascuna applicazione SBDIOI40. Supponendo che l'applicazione "carpi" sia costituita dai servizi "db", "proc" e "web":

- il nome della rete sarà "carpinet" (e quello della sottorete "carpisubnet")

- il nome delle VM sarà “carpidbvm”, “carpiprocvm” e “carpiwebvm”
- il nome delle porte sarà “carpidbport”, “carpiprocport” e “carpiwebport”

Il diagramma seguente mostra la denominazioni dei componenti dell’applicazione “carpi” (il router è omesso):



Più in generale, in un’applicazione di nome <app>:

- il nome della rete sarà <app>net
- il nome delle VM sarà <app><serv>vm, dove <serv> rappresenta il nome di ciascun servizio
- nome delle porte sarà <app><serv>port

Grazie a queste convenzioni, sarà possibile scrivere un software che non abbia bisogno di essere "istruito" riguardo le caratteristiche e i componenti dell’applicazione che vogliamo migrare (ad esempio, attraverso un file di configurazione). L’unica informazione essenziale che contraddistingue un’applicazione, e permette di distinguerne una dall’altra, è il suo nome; il resto può essere dedotto in modo sostanzialmente automatico.

Un insieme di **flavor** predefiniti. Per ora, di default, viene sempre applicato il flavor “m1.tiny”.

Name	**VCPUs**	**RAM**	**Disk**
m1.tiny	1	512MB	1GB
m1.small	1	2GB	20GB
m1.medium	2	4GB	40GB
m1.large	4	8GB	80GB
m1.xlarge	8	16GB	160GB

Un **security group** di predefinito di nome “default” che consente il passaggio del traffico di rete senza applicare alcun filtro. Per ora viene applicato sempre quello.

Name	default
Description	Default security group

I **floating IP**, ove necessari, devono per ora essere assegnati manualmente dall’amministratore della piattaforma di destinazione della migrazione. In effetti, non si tratta di un’operazione facilmente automatizzabile dal momento che i floating IP da assegnare sono verosimilmente diversi tra una piattaforma e l’altra.

La gestione delle **chiavi SSH** per l’accesso remoto alle VM non è per ora implementata e deve essere eseguita manualmente.

Scelte implementative

Per migrare un’applicazione tra due piattaforme OpenStack occorre senza dubbio interagire con la piattaforma di partenza e con quella di destinazione. Come già visto, esistono sostanzialmente tre alternative per interagire con OpenStack: la dashboard accessibile da browser web; l’interfaccia a riga di comando (CLI) utilizzabile sui nodi fisici del cluster; le interfacce applicative (REST API) esposte dai servizi OpenStack. Seguendo il secondo metodo, T3LAB ha già preparato un insieme di script shell che eseguono la migrazione dell’applicazione Carpigiani e che mostrano con successo la fattibilità dell’approccio. Gli script interagiscono con OpenStack collegandosi tramite SSH alle due macchine fisiche delle rispettive piattaforme e poi impartendo comandi al CLI. Trattandosi di un *proof-of-concept*, lo script presenta alcuni svantaggi: risulta dipendente dalle modifiche dell’interfaccia del CLI; richiede credenziali di accesso alla macchina, anziché soltanto accesso a OpenStack; non risulta pienamente conforme alla prassi d’uso, visto che le API OpenStack sono pensate proprio per interazione da parte di software applicativo. In aggiunta, gli script sono ritagliati su una specifica applicazione (quella di Carpigiani) e non possono essere riutilizzati con applicazioni diverse rispetto a quella prestabilita. È proprio per affrontare queste questioni ancora aperte che abbiamo deciso di lavorare a un nuovo software per la migrazione.

Si è scelto di implementare il prototipo software in linguaggio Go[2]. Si tratta di un linguaggio moderno, che dispone di una libreria standard assai ampia e ordinata, e che semplifica lo sviluppo di software complesso. Un software scritto in Go risulta più robusto rispetto a script shell, grazie, ad esempio, alla peculiare centralità della gestione degli errori nel linguaggio e alla compilazione statica che rende ciascun eseguibile autocontenuto e dunque indipendente dalle librerie installate sul sistema ospitante. In definitiva, Go abilita quindi la scrittura di software estendibile e affidabile per la gestione di una generica applicazione SBDIOI40.

[2] The Go Programming Language, https://golang.org/

Si è scelto, inoltre, di interagire con OpenStack invocando direttamente le sue API anziché attraverso l'emulazione di comandi sul CLI. Questa decisione rende il software dipendente soltanto dalle funzionalità "core" offerte dalla piattaforma, e non, indirettamente, anche dalle peculiarità d'uso del CLI. Inoltre, risulta sufficiente disporre di credenziali per l'accesso a OpenStack, senza il bisogno di autorizzare alcun accesso al sistema operativo delle macchine fisiche del cluster. Per l'utilizzo delle REST API si è deciso di sfruttare Gophercloud[3], un software development kit (SDK) per Go che fornisce allo sviluppatore gli strumenti per collegarsi a un cloud OpenStack ed eseguire ogni possibile operazione senza dover manipolare direttamente gli scambi di messaggi REST.

Anziché costruire un unico eseguibile monolitico, si è scelto di sviluppare una libreria e poi un semplice programma console che utilizza le funzionalità della libreria. Si spera, in questo modo, di rendere il codice più semplice da utilizzare, da manutenere e, eventualmente, estendere.

Il package sbdioi40

La libreria (*package* in Go) ha nome "sbdioi40" e import path "github.com/lorciv/sbdioi40". Il codice è consultabile al repository https://github.com/lorciv/sbdioi40/. Per utilizzare il package all'interno di un programma, occorre prima di tutto scaricare e compilare il codice sorgente con il seguente comando:

```
$ go get github.com/lorciv/sbdioi40
```

Il package contiene una funzione globale **Connect** che consente di connettersi a una piattaforma OpenStack del progetto SBDIOI40. Connect prende in ingresso le credenziali di accesso (indirizzo IP della piattaforma, utente e password) e restituisce una struttura[4] di tipo Platform (o, eventualmente, un errore).

```
func Connect(url, user, pass string) (*Platform, error)
```

Platform è una struttura che rappresenta una connessione stabilita con una piattaforma OpenStack facente parte del progetto SBDIOI40. Internamente, un oggetto Platform contiene le connessioni a ciascuno dei servizi fondamentali della piattaforma OpenStack (Neutron, Nova, Glance, e così via), tuttavia queste informazioni sono nascoste all'utente, il quale interagisce con OpenStack solamente attraverso le funzionalità della libreria sbdioi40. Una Platform, infatti, espone una serie di metodi che implementano funzionalità utili in particolare per la migrazione delle applicazioni.

Il metodo **Application** ritorna una struttura di tipo Application che contiene le informazioni sulla applicazione con quel nome (o un errore se l'applicazione non esiste). Da notare come, grazie alle convenzioni stabilite, il semplice nome sia sufficiente per individuare l'intera applicazione e i suoi servizi.

[3] Gophercloud, https://github.com/gophercloud/gophercloud
[4] Una struttura (struct) in Go è simile per funzionalità a una classe Java.

```
func (p *Platform) Application(name string) (Application, error)
```

Il metodo ListApplications ritorna la lista di tutte le applicazioni.

```
func (p *Platform) ListApplications() ([]Application, error)
```

Il metodo **Snapshot** crea uno snapshot della applicazione con quel nome, lo scarica in locale e ritorna un oggetto di tipo Snapshot. Naturalmente, si tratta di un metodo da invocare sull'oggetto Platform collegato alla piattaforma di partenza.

```
func (p *Platform) Snapshot(appname string) (Snapshot, error)
```

Una chiamata al metodo Snapshot esegue grossomodo questi passi:

- Recupera informazioni sull'applicazione (tramite il metodo Application)
- Per ogni servizio di cui l'applicazione è costituita:
  - Crea lo snapshot della V.M.
  - Esegue il download in locale dell'immagine risultante in un file temporaneo
  - Quando il download è completato, rimuove l'immagine dalla piattaforma
- Ritorna un oggetto di tipo Snapshot

L'oggetto Snapshot restituito dal metodo contiene informazioni riguardo l'applicazione di cui è stata scattata l'istantanea e anche la posizione dei file temporanei dove è stata archiviata la copia locale.

Il metodo **Restore** esegue l'upload di uno snapshot sulla piattaforma e riavvia l'applicazione. Si tratta, quindi, di un metodo da invocare sulla piattaforma di destinazione.

```
func (p *Platform) Restore(snap Snapshot) error
```

Una chiamata al metodo Restore esegue grossomodo questi passi:

- Crea la rete sulla piattaforma e la configura in modo opportuno
- Collega la rete al router
- Per ogni servizio:
  - Esegue l'upload dell'immagine dello snapshot
  - Crea la porta di rete assegnandole l'indirizzo IP della V.M. originale
  - Crea la V.M. collegata alla porta a partire dall'immagine
  - Lancia la V.M.
  - Quando l'avvio è concluso, rimuove l'immagine

Di seguito riportiamo un esempio d'uso della libreria. Il codice esegue la migrazione di un'applicazione chiamata "carpi".

```
srcPlat, err := sbdioi40.Connect("url", "user", "password")
if err != nil {
   log.Fatal(err)
}
dstPlat, err := sbdioi40.Connect("url2", "user2", "password2")
if err != nil {
   log.Fatal(err)
}
log.Print("connected successfully to both platforms")

snap, err := srcPlat.Snapshot("carpi")
if err != nil {
   log.Fatal(err)
}
log.Print(snap)

err := dstPlat.Restore(snap)
if err != nil {
   log.Fatal(err)
}
log.Print("done")
```

Il comando migra

Una volta preparata la libreria, lo sviluppo dell'eseguibile da invocare per lanciare una migrazione risulta immediato. Dal momento che le funzionalità principali sono già implementate, l'eseguibile si limita solamente ad invocare quelle necessarie nell'ordine opportuno. Omettiamo quindi il codice, che si occupa soprattutto della gestione e interpretazione dei parametri da riga di comando.

Il programma si chiama "migra" e accetta una serie di parametri fra cui:

- src, l'indirizzo IP della piattaforma di partenza (più precisamente, del suo endpoint di autenticazione)
- dst, l'indirizzo IP della piattaforma di destinazione
- user, il nome utente OpenStack
- pass, la password

Di seguito riportiamo l'output di una esecuzione d'esempio del programma.

```
$ ./migra -src "http://X.X.X.X:5000/v3" -dst "http://Y.Y.Y.Y:5000/v3" -user "luciodalla" -pass "domenicosputo" carpi
sbdioi40: connected successfully to both platforms
sbdioi40: found application "carpi" with [service "proc" service "web" service "db"]
sbdioi40: image carpiprocsnap not ready; waiting...
sbdioi40: image carpiprocsnap not ready; waiting...
sbdioi40: service snapshot of proc completed (40173568 bytes)
```

```
sbdioi40: image carpiwebsnap not ready; waiting...
sbdioi40: image carpiwebsnap not ready; waiting...
sbdioi40: service snapshot of web completed (40239104 bytes)
sbdioi40: image carpidbsnap not ready; waiting...
sbdioi40: image carpidbsnap not ready; waiting...
sbdioi40: service snapshot of db completed (40239104 bytes)
sbdioi40: snapshot of app carpi (2020-06-30 17:02:28.98108)
sbdioi40: uploaded service snapshot for proc
sbdioi40: server carpiprocvm not active yet; waiting...
sbdioi40: server carpiprocvm not active yet; waiting...
sbdioi40: server carpiprocvm not active yet; waiting...
sbdioi40: server carpiprocvm not active yet; waiting...
sbdioi40: restored service proc
sbdioi40: uploaded service snapshot for web
sbdioi40: server carpiwebvm not active yet; waiting...
sbdioi40: server carpiwebvm not active yet; waiting...
sbdioi40: server carpiwebvm not active yet; waiting...
sbdioi40: server carpiwebvm not active yet; waiting...
sbdioi40: server carpiwebvm not active yet; waiting...
sbdioi40: restored service web
sbdioi40: uploaded service snapshot for db
sbdioi40: server carpidbvm not active yet; waiting...
sbdioi40: server carpidbvm not active yet; waiting...
sbdioi40: server carpidbvm not active yet; waiting...
sbdioi40: server carpidbvm not active yet; waiting...
sbdioi40: restored service db
sbdioi40: carpi snapshot removed from local storage
sbdioi40: done
```

Avvio veloce di applicazioni a container

Come già visto, le applicazioni SBDIOI40 sono in generale strutturate secondo uno schema a multiservizi. I singoli servizi delle applicazioni potrebbero essere implementati attraverso delle macchine virtuali gestite da OpenStack oppure con dei container gestiti da Docker. Nella prima parte di questo documento abbiamo descritto un sistema per la migrazione di applicazioni a macchine virtuali tra due piattaforme OpenStack.

In questa seconda parte descriveremo un sistema che velocizza la migrazione di applicazioni a container tra due piattaforme Docker. Precedentemente al progetto SBDIOI40, è stato sviluppato un sistema chiamato "FogDocker" che contribuisce a velocizzare l'avvio di singoli container. L'obiettivo di questo lavoro è estendere e adattare l'approccio anche per applicazioni a microservizi, che sono composte da container multipli. Il resto della sezione è organizzato come segue: dopo un breve richiamo alle tecnologie di base, sarà illustrato il problema dell'avvio lento delle applicazioni a microservizi; poi, sarà presentato a grandi linee FogDocker, sottolineando il principio di funzionamento, la modalità d'uso, i vantaggi prodotti in termini di performance ma anche i limiti nell'utilizzo con applicazioni complesse; infine, si descriverà FogCompose, l'estensione di FogDocker pensata per l'uso con applicazioni a microservizi anche nel progetto SBDIOI40.

Docker container

Un contenitore software, meglio noto come *container*, è un'unità software che realizza un ambiente virtuale in cui un'applicazione può eseguire senza il rischio di interferire con le altre. Più precisamente, un container consiste in un tradizionale processo del sistema operativo che però opera all'interno di un ambiente isolato. In Linux, tale isolamento si ottiene impiegando opportunamente due potenti funzionalità del kernel: i *namespace* e i *control group*. I namespace consentono di partizionare le risorse del kernel in modo che un processo abbia accesso soltanto ad un sottoinsieme delle risorse. Ad esempio, è possibile partizionare un punto di mount del filesystem, lo spazio di nomi dei processi (PID), lo stack di rete (indirizzi IP, porte, tabelle di routing, ecc.) e così via. I control group, invece, consentono di limitare la quantità di risorse che un processo può impiegare (ad esempio CPU, memoria, disco, rete, ecc.). Ad esempio, si può limitare al 20% la capacità di calcolo fruibile da un processo. Quando namespace e control group sono utilizzati per isolare un processo, allora il processo prende il nome di container.

L'efficienza e la semplicità dell'approccio a container hanno contribuito al forte incremento della sua popolarità, e tra i vari strumenti di gestione disponibili Docker è di gran lunga quello più diffuso. Docker è una piattaforma che abilita la costruzione, il deployment, l'esecuzione, e, più in generale, la gestione dei container. Docker semplifica l'uso dei container offrendo all'utente la possibilità di interagire con un'interfaccia semplice e immediata. Ad esempio, un semplice comando come docker run -it alpine recupera la distribuzione Alpine Linux, inizializza il filesystem, prepara i namespace e i control group necessari per l'isolamento del container, avvia il container e ritorna una shell interattiva all'utente.

La combinazione della tecnica di virtualizzazione a container e di uno strumento che ne agevola la gestione genera un sistema che si rivela adatto per l'applicazione in diversi scenari, fra i quali menzioniamo *fog*

computing e *applicazioni a microservizi*. Il fog computing è un'estensione del paradigma cloud computing che si propone di distribuire geograficamente le funzionalità di calcolo e archiviazione portandole vicino all'utente finale che produce i dati e consuma il risultato. In un'architettura fog, la virtualizzazione è auspicabile perché permette di ospitare contemporaneamente servizi diversi sui nodi senza il rischio di incorrere in interferenze fra gli stessi; allo stesso tempo, le risorse computazionali sono limitate, dal momento che i nodi fog sono verosimilmente costituiti da apparecchi piccoli ed economici (come ad esempio un Raspberry Pi). Di qui la popolarità dei Docker container nei sistemi fog.

Docker consente di amministrare in modo semplice e uniforme i singoli container. Tuttavia, la gestione di applicazioni a microservizi, spesso composte da un notevole numero di container che interagiscono tra loro, rischia di diventare un'attività tediosa e incline ad errori. Per questo motivo è stato introdotto un nuovo strumento chiamato **Docker Compose**. Docker Compose permette di gestire in modo semplice un'intera applicazione a microservizi e tutti i componenti virtuali che la realizzano. Si tratta di uno strumento a riga di comando che lavora *on-top-of* Docker e automatizza la costruzione, l'avvio e l'arresto di tutti i container che costituiscono l'applicazione. Oltre ai container, Docker Compose si occupa anche di creare e gestire le reti virtuali che li mettono in comunicazione e i volumi per l'archiviazione dei dati permanenti. Ogni volta che si invoca un'operazione su Docker Compose, questo deve essere istruito circa le caratteristiche e la configurazione dell'applicazione e di tutti i suoi servizi attraverso la redazione di un apposito file di configurazione chiamato *Compose file*. Il Compose file è un file YAML[5] che definisce i servizi, le reti e i volumi che costituiscono l'applicazione. Se non altrimenti specificato, Docker Compose cerca un file di nome "docker-compose.yml" all'interno del direttorio corrente.

FogDocker

In Docker, un'immagine è un'istantanea a partire dalla quale molteplici container dello stesso tipo possono essere istanziati. Un'immagine non è un oggetto monolitico; piuttosto, il suo contenuto è organizzato in diversi livelli chiamati *layer*. I layer di un'immagine sono impilati uno sull'altro a formare uno stack che globalmente rappresenta il contenuto iniziale del filesystem di un container. Sfortunatamente, a causa delle loro grandi dimensioni, il processo di *deployment* di un'immagine da un registro remoto verso una macchina locale tende a richiedere tempi lunghi. La lentezza di questa operazione è particolarmente svantaggiosa in un'architettura fog, dove i servizi devono muoversi velocemente da un nodo all'altro in risposta alla mobilità degli utenti, ma anche nel caso delle applicazioni a microservizi, dove i container da avviare sono molteplici. Per porre rimedio a tale fenomeno, è stato presentato FogDocker.

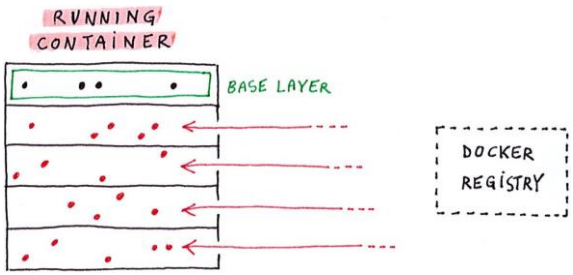
FogDocker[6] è un sistema che propone un approccio originale all'operazione di download delle immagini Docker con l'obiettivo di ridurre il tempo necessario per avviare un container. FogDocker fa leva sul fatto che,

[5] YAML: Yaml Ain't Markup Language, https://yaml.org/
[6] Lorenzo Civolani, Guillaume Pierre, and Paolo Bellavista. 2019. *FogDocker: Start Container Now, Fetch Image Later*. In Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'19). Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3344341.3368811

nonostante le immagini tendono ad avere dimensioni importanti, solo una parte dei file è effettivamente necessaria nelle prime fasi dell'esecuzione. L'idea centrale del lavoro è dunque quella di scaricare soltanto il contenuto essenziale per l'esecuzione dei container e procedere immediatamente con l'avvio; poi, in un secondo momento, mentre l'applicazione è già al lavoro, il sistema può proseguire col recupero della restante parte dell'immagine. Il progetto della soluzione menzionata ha richiesto di dirimere una serie di questioni, tra cui:

- scegliere un criterio per identificare i file essenziali per l'esecuzione;
- riorganizzare il contenuto di un immagine in modo da isolare tali file in un layer dedicato (il *base layer*);
- introdurre modifiche nel codice di Docker per implementare la nuova politica di deployment;
- assicurare il corretto funzionamento del sistema anche quando il container prova ad accedere a un file il cui recupero è stato posticipato.

I risultati delle prove sperimentali mostrano comunque che il sistema è in grado di ridurre i tempi di avvio fino ad un fattore 6.



Per una discussione dettagliata dell'implementazione di FogDocker, che esula dallo scopo di questo documento, si rimanda all'articolo corrispondente. Tuttavia, è importante sottolineare che nonostante FogDocker sia adatto per singoli container, esso presenta degli inconvenienti che rischiano di ostacolarne l'uso in un contesto reale. Le procedure di profiling, analisi dei risultati, costruzione del base layer e della versione finale (fog) dell'immagine, sebbene ben definite attraverso una sequenza precisa di passi incrementali, sono da eseguirsi manualmente. Tale approccio è stato sicuramente vantaggioso in fase di sviluppo e può considerarsi tollerabile per container singoli, ma rischia di diventare oggettivamente impraticabile quando si ha a che fare con applicazioni a multiservizi. In questo caso, infatti, la procedura di profilamento va ripetuta per ogni servizio, il che rischia di diventare un'attività tediosa e soprattutto prona ad errori. È dunque chiara la necessità di sviluppare uno strumento che renda possibile utilizzare FogDocker anche con un caso d'uso più articolato, e dunque usufruire dei vantaggi in termini di performance al momento dell'avvio.

L'obiettivo di questa fase del lavoro è dunque quello di estendere la possibilità di utilizzare FogDocker alle applicazioni a microservizi. Il compito si compone di due aspetti principali:

- verificare la compatibilità di FogDocker con Docker Compose;

- automatizzare la fase del profilamento.

Nella restante parte di questa sezione affronteremo con un certo grado di dettaglio la descrizione di ciò che è stato svolto.

Compatibilità di FogDocker con Docker Compose

Il primo passo del lavoro consiste nel verificare che il funzionamento di Docker Compose quando in esecuzione su FogDocker corrisponda alle aspettative. Come già visto, Docker Compose è uno strumento che semplifica la gestione di applicazioni a microservizi composte da container multipli. Docker Compose interagisce con Docker attraverso la sua interfaccia di programmazione (API) che, nel caso di esecuzione sul computer locale, viene esposta su una socket di dominio locale Unix. Nonostante FogDocker introduca dei cambiamenti nel comportamento di alcune operazioni (principalmente, l'operazione *pull*), esso non altera le API. Di conseguenza, Docker Compose può funzionare anche nel caso in cui sul PC locale si trovi in esecuzione FogDocker al posto della controparte originale. Naturalmente, è necessario che ciascun container che compone l'applicazione coinvolta in sia stato oggetto di profilamento e sia quindi dotato del base layer addizionale, che è fondamentale per l'avvio rapido.

Convenzioni e struttura del workspace

Un'applicazione a microservizi è composta da container multipli che comunicano fra di loro attraverso una rete virtuale. Ogni container viene istanziato a partire dal rispettivo modello che in Docker viene indicato col termine *immagine*. Per agevolare lo sviluppo del software di profilamento automatico sono state stabilite delle convenzioni circa la denominazione delle immagini che compongono le applicazioni. In particolare, il nome delle immagini deve essere composto in questo modo:

utente/applicazione-servizio:versione

Ad esempio, l'immagine in versione originale del servizio "web" appartenente all'applicazione "gbook" costruita dall'utente "lorciv" sarà denominata come segue:

lorciv/gbook-web:orig

Una volta definite le convenzioni sulla denominazione delle immagini dei servizi, ci si è concentrati sulla struttura dello spazio di lavoro (workspace). La radice del workspace coincide con una cartella nel filesystem che porta il nome dell'applicazione. Ad esempio, il workspace relativo all'applicazione "gbook" sarà contenuto in una cartella di nome "gbook". La struttura interna del workspace è organizzata intorno all'esistenza di molteplici versioni dell'applicazione:

- Versione originale (orig)
- Versione profilamento (prof)
- Versione minimale (lean)

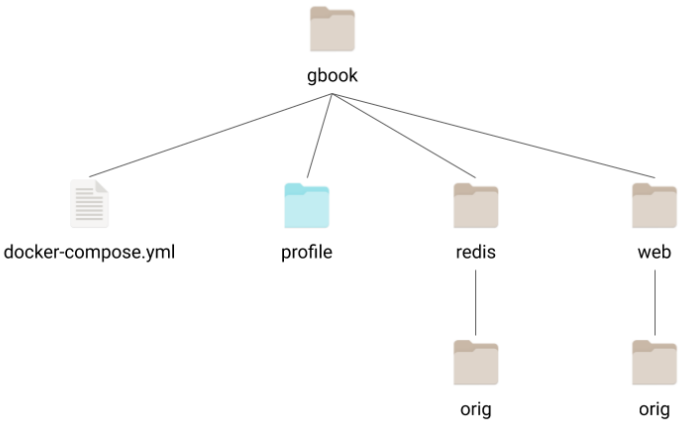
- Versione finale (fog)

Naturalmente, quando ci apprestiamo ad eseguire il profilamento di una applicazione l'unica versione esistente sarà quella originale. Sarà compito dello strumento di profilamento automatico produrre le restanti versioni, fino a quella finale.

Dunque, all'interno del workspace ci sono:

- N sottocartelle, una per ogni servizio che compone l'applicazione e ciascuna con il nome del servizio. Ad esempio, se l'applicazione è composta da due servizi "redis" e "web", il workspace conterrà due sottocartelle rispettivamente con lo stesso nome.
- I file di configurazione per Docker Compose (detti anche compose file), uno per ogni versione dell'applicazione. Il compose file per la versione originale deve essere denominato semplicemente "docker-compose.yml", in accordo alle convenzioni di Docker Compose. Delle altre versioni si parlerà più avanti.
- Una speciale cartella di nome "profile" che contiene gli strumenti necessari per la fase di profilamento. Anche del contenuto di questa cartella si parlerà più avanti.

Ciascuna delle N cartelle dedicate ai servizi è internamente organizzata in ulteriori sottocartelle che individuano le varie versioni per il singolo container. Ognuna di queste cartelle contiene il contesto di compilazione (build context) del servizio nella relativa versione, il quale include sempre un Dockerfile. Dal momento che, inizialmente, è presente solo la versione originale dell'applicazione, per ogni servizio sarà sufficiente predisporre soltanto la cartella "orig". Riprendendo l'esempio precedente, in posizione gbook/redis/orig dovremmo trovare la versione originale del servizio "redis" per l'applicazione "gbook".



Il file docker-compose.yml sarà strutturato in modo da specificare per ciascun servizio il build context della versione originale. Anche la denominazione delle varie immagini Docker segue le condizioni prestabilite. Ad esempio, nel caso dell'applicazione gbook, il compose file potrebbe essere quello mostrato di seguito.

file: docker-compose.yml

```
version: "3"
services:
  redis:
    image: lorciv/gbook-redis:orig
    build: ./redis/orig
  web:
    image: lorciv/gbook-web:orig
    build: ./web/orig
    ports:
      - "8080:80"
```

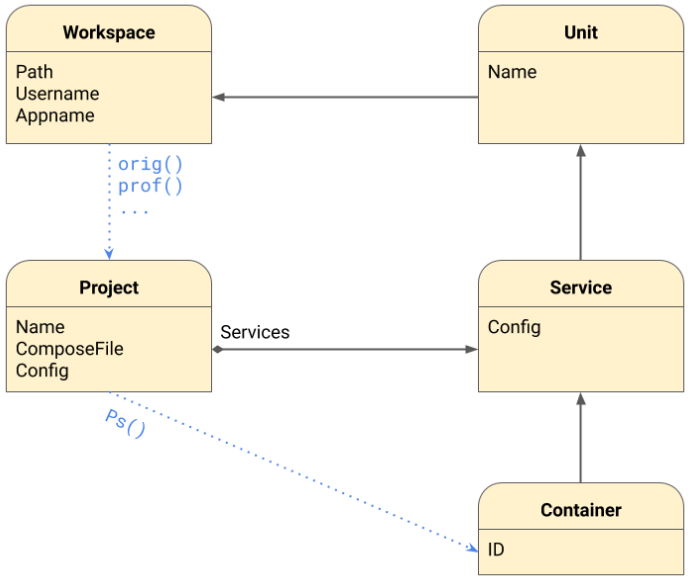
Definita la struttura dell'applicazione in versione originale, si è passati alla preparazione dello strumento per automatizzare le fasi principali del profilamento.

Struttura del software

Si è scelto di realizzare lo strumento di automatizzazione del profilamento sotto forma di un tool a riga di comando. Il suo nome è "autoprof". Come linguaggio di programmazione per l'implementazione è stato scelto Go dal momento che, oltre ai vantaggi già enumerati nel corso della prima parte di questo documento, esso dispone di librerie semplici e funzionali per interagire con il sistema operativo, con Docker, per l'interpretazione dei file YAML nonché per la gestione di argomenti e parametri da riga di comando. Inoltre, la compilazione statica offerta di default dall'ambiente di sviluppo del linguaggio consente di produrre un eseguibile affidabile in quanto non dipendente da alcuna libreria preinstallata sul sistema operativo ospitante.

L'obiettivo di autoprof è quello di offrire un insieme di semplici comandi per: eseguire il setup dell'ambiente di lavoro in previsione del profilamento; analizzare i risultati ottenuti dal profilamento; generare una versione minimale (lean) dell'applicazione; generare la versione finale (fog) dell'applicazione.

A livello di implementazione, il software è stato organizzato intorno a una serie di strutture (simili per funzionalità alle classi Java) che consentono di decomporre il codice in sottoparti distinte. La figura seguente mostra uno schema UML semplificato delle entità che interagiscono nel sistema autoprof. Successivamente, si forniranno dettagli aggiuntivi per la sua interpretazione.



La struttura **Workspace** punta alla radice dello spazio di lavoro e costituisce il punto di accesso a tutte le operazioni relative al profilamento delle applicazioni. Workspace permette di ottenere un riferimento a una specifica versione dell'applicazione sotto forma di strutture Project. Ad esempio, tramite il metodo orig è possibile ottenere la versione originale dell'applicazione.

La struttura **Project** rappresenta un'applicazione che può essere gestita con Docker Compose. Nel contesto di questo lavoro, Project individua una specifica versione dell'applicazione (ad esempio, la versione originale) ed è quindi associato a un preciso compose file contenuto nel workspace. Gli oggetti di tipo Project offrono una serie di funzionalità che emulano l'emissione di un comando a Docker Compose. Ad esempio, attraverso Project è possibile creare un'istanza del progetto, avviare o interrompere la sua esecuzione, oppure ottenere informazioni sulla configurazione dei relativi container.

Ogni Progetto è costituito da una serie di **Service**, ciascuno dei quali rappresenta un servizio dell'applicazione in una specifica versione. Ogni servizio contiene un riferimento a un oggetto di tipo **Unit**, il quale a sua volta rappresenta un servizio in ogni sua possibile versione. Un oggetto Unit corrisponde sostanzialmente a una delle sottocartelle del workspace (ad esempio, riprendendo il caso dell'applicazione gbook, la sottocartella "web" oppure "redis").

Tra le operazioni che è possibile invocare su un progetto, Ps consente di ottenere informazioni sull'istanza in esecuzione. Il metodo Ps ritorna al chiamante una lista di oggetti di tipo **Container**, ciascuno dei quali

rappresenta naturalmente un container Docker che compone l'istanza. Ogni container contiene un identificatore ID che è possibile utilizzare per invocare operazioni Docker su di esso.

Funzionalità di autoprof

Il punto di partenza di autoprof è una funzione main che esegue il parsing dei parametri a riga di comando e poi avvia l'operazione richiesta. autoprof può essere invocato con una sola opzione "workspace" che consente di specificare il percorso della directory radice dello spazio di lavoro (se si omette il parametro, autoprof assume che la radice del workspace coincida con la directory corrente). Dopo le opzioni, è necessario specificare il comando da eseguire. Nel momento in cui scriviamo, sono possibili quattro comandi: "setup", "process", "lean" e "fog". Una volta eseguito il parsing di parametri e comando, autoprof costruisce un oggetto Workspace e poi invoca la funzione corrispondente al comando inserito dal cliente.

Setup. A partire da un nuovo workspace contenente l'applicazione originale, la prima operazione è quella di abilitare il profilamento.

```
$ ./autoprof -workspace path/to/dir setup
```

Abilitare il profilamento consiste sostanzialmente nel preparare una nuova versione dell'applicazione. La nuova versione, che possiamo chiamare come versione di "profiling", differisce da quella originale prima di tutto per il comando iniziale che viene eseguito al momento del lancio di ciascun servizio. Nella terminologia di Docker, tale comando iniziale è normalmente suddiviso in due parti chiamate *entrypoint* e *command*, e la loro combinazione stabilisce qual è il comando da eseguire al momento dell'avvio di un container. Nella versione originale dell'applicazione, il comando iniziale di ogni container avvia semplicemente il servizio corrispondente. Nella versione di profiling, la questione è più articolata: prima di tutto occorre avviare lo strumento di profilamento che intercetta tutte le operazioni sul filesystem; poi, una volta che il profilamento è in esecuzione nel background, può finalmente essere avviato anche il servizio vero e proprio. Conformemente a quanto fatto in FogDocker, anche in FogCompose intercetta le operazioni sul filesystem tramite fatrace[7], un programma che internamente sfrutta le fanotify API[8] offerte dal kernel Linux. fatrace è incluso nello spazio di lavoro all'interno della speciale cartella profile. Oltre a fatrace, la cartella include anche uno script shell chiamato "do.sh" il cui unico compito è quello di avviare il profilamento in background e poi eseguire quanto passato come argomento. Tale script aiuta a semplificare la preparazione del Compose file per la nuova versione dell'applicazione. Naturalmente, la cartella profile deve essere disponibile all'interno di ciascun container, e questo può essere ottenuto per mezzo di un *bind-mount*.

Di seguito riportiamo il Compose file relativo alla versione di profilamento dell'applicazione "gbook". Esiste una serie differenze tra questa versione del file e quella dell'applicazione originale. Le modifiche introdotte in accordo a quanto discusso nel paragrafo precedente sono importanti; viceversa, negli altri casi si tratta

[7] fatrace: report system wide file access events, https://piware.de/2012/02/fatrace-report-system-wide-file-access-events/
[8] fanotify - monitoring filesystem events, https://man7.org/linux/man-pages/man7/fanotify.7.html

semplicemente di alterazioni di forma ma non di sostanza, introdotte dal parser al momento dell'esportazione della configurazione di Docker Compose dalle strutture dati Go verso il formato YAML.

file: docker-compose-profile.yml

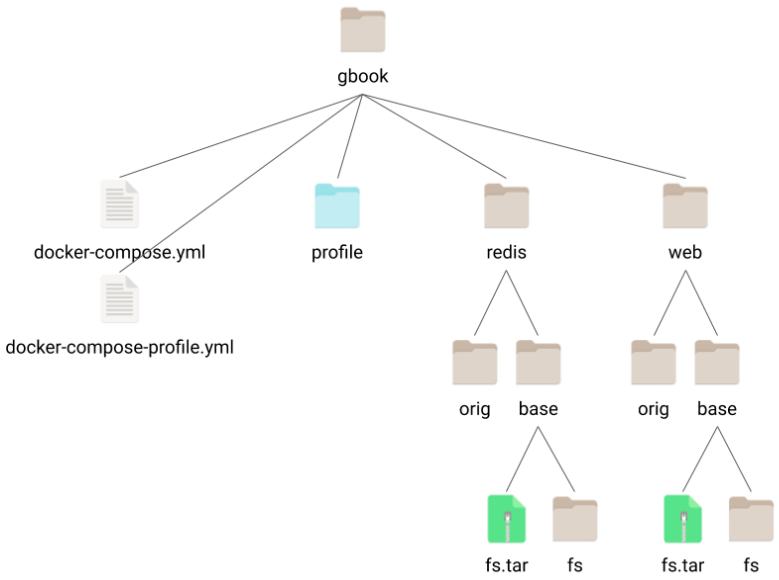
```
version: "3.2"
services:
  redis:
    build:
      context: ./redis/orig
    cap_add:
     - SYS_ADMIN
    entrypoint:
     - /profile/do.sh
     - redis
     - docker-entrypoint.sh
     - redis-server
    image: lorciv/gbook-redis:orig
    volumes:
     - type: bind
       source: ./profile
       target: /profile
  web:
    build:
      context: ./web/orig
    cap_add:
     - SYS_ADMIN
    entrypoint:
     - /profile/do.sh
     - web
     - docker-php-entrypoint
     - apache2-foreground
    image: lorciv/gbook-web:orig
    ports:
     - mode: ingress
       target: 80
       published: 8080
       protocol: tcp
    volumes:
     - type: bind
       source: ./profile
       target: /profile
```

Il comando setup esegue nell'ordine le seguenti operazioni:

- Controllo delle precondizioni
  - "docker-compose.yml" deve essere presente
  - la struttura di base del workspace deve essere presente

- Creazione di un'istanza dell'applicazione in versione originale
- Analisi del compose file
- Per ogni servizio dell'applicazione, regola la configurazione per il profilamento
  - trova il comando iniziale (entrypoint e command)
  - imposta il nuovo comando iniziale
  - conferisci privilegi di amministratore (necessario per utilizzare le fanotify API)
  - configura *bind-mount* della cartella profile all'interno del container
- Salva la configurazione all'interno di un nuovo compose file chiamato "docker-compose-profile.yml"
- Per ogni servizio, esporta ed estrai il filesystem
  - esporta il filesystem nel file service/base/fs.tar
  - estrai all'interno di service/base/fs
- Rimuovi l'istanza dell'applicazione originale

Al termine del processo di setup, lo spazio di lavoro dell'applicazione di esempio "gbook" si presenta come nello schema seguente.



Avvio del profilamento. Per avviare il profilamento, occorre semplicemente avviare la versione "profile" dell'applicazione. Non è stato giudicato necessario incorporare questa funzionalità all'interno di autoprof dal momento che è sufficiente utilizzare il comando "up" di Docker Compose. Naturalmente, occorre indicare a

Docker Compose che il compose file da interpretare non è quello di default ma bensì quello relativo alla versione di profiling: è possibile farlo per mezzo del parametro -f.

```
$ docker-compose -f docker-compose-profile.yml up
```

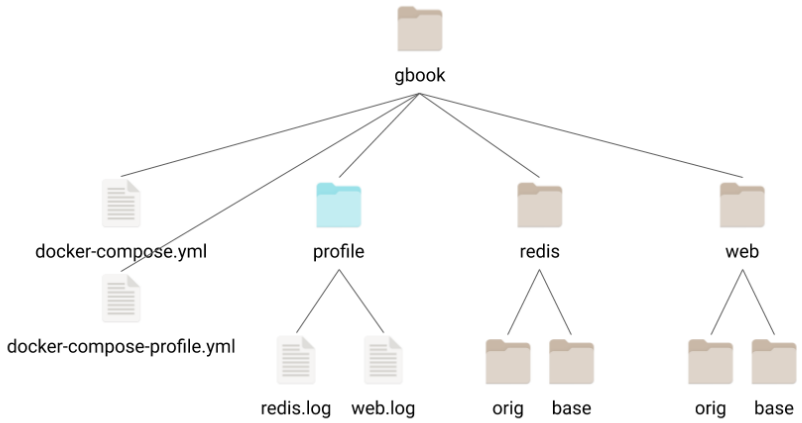
Il profilamento è configurato in modo da produrre all'interno della speciale cartella "profile" dello spazio di lavoro le informazioni riguardo il log degli accessi ai file. In particolare, verrà prodotto un file di log diverso per ciascun servizio.

Simulazione. Una volta avviato il profilamento, l'applicazione è in esecuzione e ogni accesso ai file viene rilevato e registrato automaticamente negli appositi file di log. A questo punto occorre dunque simulare un caso d'uso il più possibile rappresentativo delle normali condizioni di lavoro del software. Ad esempio, nel caso di un'applicazione web, potrebbe essere ragionevole simulare l'utilizzo della piattaforma da parte di cliente. Questa fase risulta difficilmente automatizzabile perché dipende intrinsecamente sia dalla tipologia di applicazione che si vuole profilare sia dal contesto in cui essa sarà verosimilmente impiegata. Ciononostante, nell'ambito di un eventuale sviluppo futuro si potrebbe estendere autoprof in modo da ricevere in ingresso un modulo software (ad esempio, uno script shell) che incapsula la logica di simulazione, e poi chiedere al cliente di fornirlo al sistema contestualmente all'applicazione in versione originale. In questo modo si potrebbe compiere un passo importante verso l'automazione completa del procedimento.

Arresto del profilamento. Specularmente all'avvio del profilamento, anche la sua interruzione è un'operazione semplice: è sufficiente fermare l'esecuzione dell'applicazione. Per farlo, basta utilizzare comando "down" di Docker Compose:

```
$ docker-compose -f docker-compose-profile.yml down
```

Il risultato del profilamento, come già anticipato, sarà disponibile nella cartella "profile". In particolare, il risultato consiste in una serie di file di log, ciascuno dei quali riporta lo storico degli accessi ai file relativi a ciascun servizio.



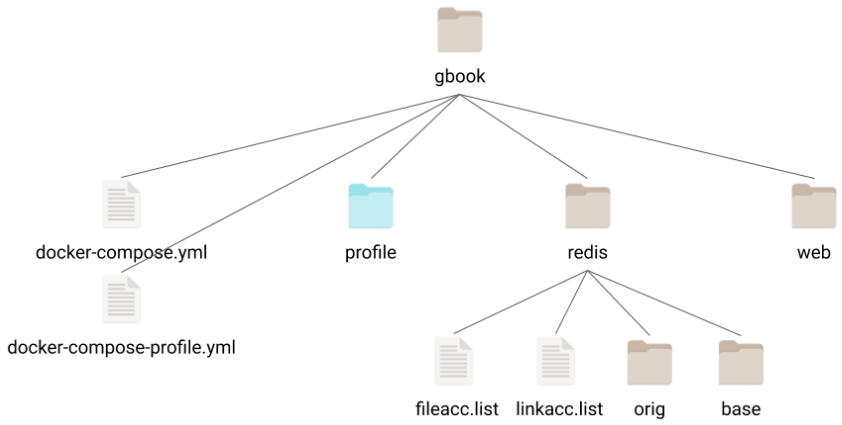
Elaborazione del risultato. Una volta conclusa la simulazione occorre elaborare il risultato. In effetti, il log degli accessi ai file non tiene conto del fatto che un file sia stato acceduto tramite un link; al contrario, solo l'oggetto finale viene rilevato. Una parte fondamentale dell'elaborazione del risultato consiste nell'identificare, all'interno del filesystem del container, tutti i link che conducono alla lista di file essenziali per l'esecuzione. Per avviare l'elaborazione è sufficiente invocare autoprof con il comando "process".

```
$ ./autoprof -workspace path/to/dir process
```

Il comando process esegue le seguenti operazioni:

- Sposta i file di log nelle cartelle dei rispettivi servizi (attribuendo a ciascuno il nome "fileacc.log")
- Per ogni servizio, elabora i risultati
    - a partire dal log, ottieni una lista di file ordinata e senza ripetizioni (genera "fileacc.list")
    - a partire dalla lista di file, trova, all'interno del filesystem del container originale esportato in precedenza, tutti i link che conducono a quel file (genera "linkacc.list")

Il diagramma seguente mostra la struttura del workspace al termine dell'elaborazione del risultato. Il contenuto della cartella "web" è stato omesso perché assolutamente speculare a quanto contenuto nella cartella "redis".



Generazione della versione lean. L'elaborazione del risultato del profiling ci consente di ottenere tutte le informazioni riguardo ciò che è necessario includere all'interno del base layer di ciascun servizio per garantire il funzionamento dell'applicazione in versione minimale. Prima di procedere con la costruzione della versione "fog" dell'applicazione, che è il passo finale, occorre quindi costruire i base layer e assicurarsi che essi siano in grado di supportare l'esecuzione in un contesto d'uso simile a quello che è stato simulato. La costruzione dei base layer è un processo automatizzabile, e per questo fa parte delle funzionalità offerte dallo strumento autoprof.

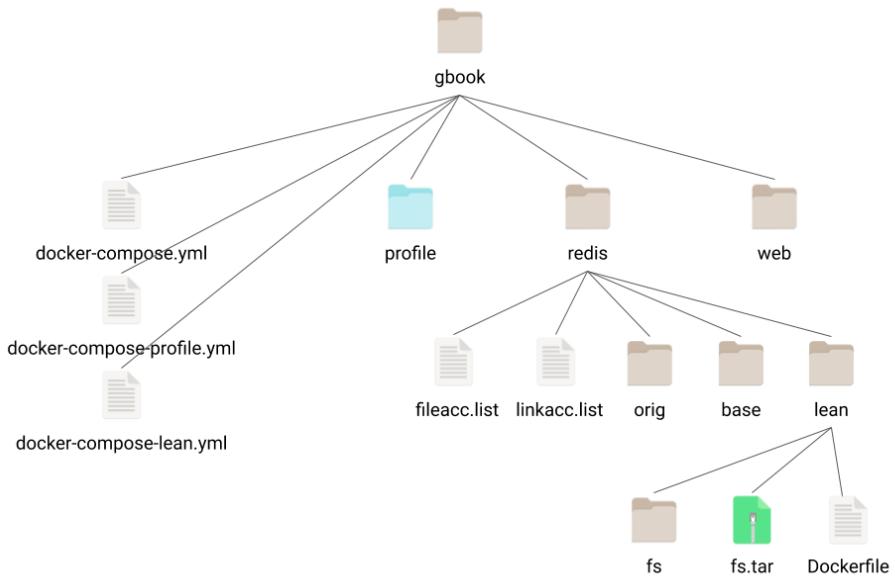
```
$ ./autoprof -workspace path/to/dir lean
```

Il comando esegue nell'ordine i seguenti passi:

- Per ogni servizio, costruisci il base layer e prepara la versione minimale (lean)
  - Costruzione del base layer
    - crea la cartella <servizio>/lean/fs
    - clona albero dei direttori del filesystem originale
    - clona file essenziali (grazie a fileacc.list)
    - clona link verso quei file (grazie a linkacc.list)
    - comprimi contenuto della cartella fs nell'archivio fs.tar
  - Generazione del Dockerfile
    - ispeziona la *history* di creazione dell'immagine originale
    - estrai i comandi ENV, VOLUME, WORKDIR, ENTRYPOINT e CMD
    - crea il Dockerfile per l'immagine lean in modo da includere la parte necessaria di filesystem, e poi impostare il resto della configurazione conformemente alla versione originale (ad esempio, preservando le variabili d'ambiente)

    - Generazione del dockerignore
        - escludere dal contesto di build la cartella fs, perché è sufficiente l'archivio fs.tar
- Genera il Compose file per la versione minimale (lean) dell'intera applicazione
    - imposta il contesto di build alla cartella <servizio>/lean
    - imposta nome immagine a <user>/<app>-<servizio>:lean
- Salva il Compose file nel workspace con nome "docker-compose-lean.yml"

Terminata l'esecuzione del comando, i base layer di ciascun servizio saranno disponibili nelle rispettive sottocartelle "lean". Inoltre, una nuova versione minimale dell'applicazione sarà disponibile sotto forma di un compose file di nome "docker-compose-lean.yml".



Con un semplice comando Docker Compose (il comando "up" già visto in precedenza) è possibile avviare l'esecuzione dell'applicazione in formato minimale e verificarne il corretto funzionamento. Il corretto funzionamento della versione minimale è imprescindibile per assicurare il funzionamento anche della versione finale.

Generazione della versione fog. Una volta appurato che l'applicazione in versione minimale è in grado di eseguire correttamente, diventa possibile generare la versione finale. La versione finale dell'applicazione è costruita in modo che, per ciascun servizio, lo stack di layer che ne compone il filesystem contenga un layer aggiuntivo che chiamiamo *layer di base*. Il layer di base coincide con il contenuto dell'archivio in posizione

Commented [1]: Inserire

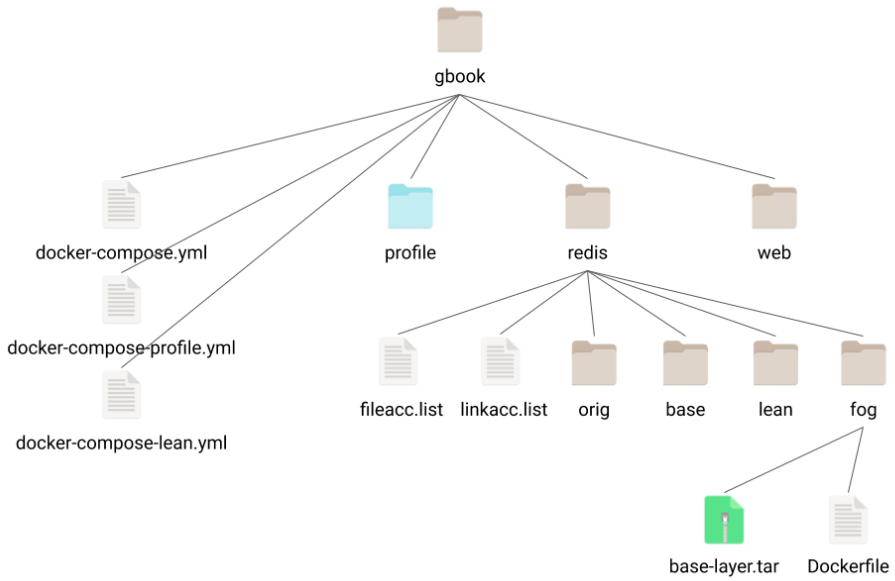
lean/fs.tar e corrisponde quindi a quel sottoinsieme del filesystem del container la cui presenza è imprescindibile per l'esecuzione. Se l'applicazione sarà lanciata su un computer che utilizza FogDocker, allora soltanto il layer di base verrà recuperato via rete prima dell'esecuzione, contribuendo alla riduzione del volume di dati da trasferire e dunque accorciando i tempi di avvio. Il resto dell'immagine sarà scaricato e installato solo più tardi, quando l'applicazione sarà già al lavoro.

Per generare la versione finale (fog) dell'applicazione, è possibile invocare autoprof in questo modo:

```
$ ./autoprof -workspace path/to/dir fog
```

Dal punto di vista dell'implementazione, l'operazione è piuttosto semplice:

- Per ogni servizio, prepara la versione finale (fog)
  - crea la cartella <servizio>/fog
  - copia <servizio>/lean/fs.tar in <servizio>/fog/base-layer.tar
  - genera il Dockerfile per la versione fog del servizio
- Genera il Compose file per la versione finale (fog) dell'intera applicazione
  - imposta il contesto di build dei servizi alla cartella <servizio>/fog
  - imposta il nome immagine dei servizi a <user>/<app>-<servizio>:fog
- Salva il Compose file nel workspace con nome "docker-compose-fog.yml"



Per ciascun servizio, il Dockerfile della versione fog è estremamente semplice. In particolare, consiste di due istruzioni: la prima dichiara l'immagine originale come base di partenza; la seconda aggiunge, in cima allo stack, il nuovo layer di base. Ad esempio, nel caso del servizio "web":

```
FROM lorciv/gbook-web:orig
ADD base-layer.tar /
```

Naturalmente, l'applicazione in versione fog può essere avviata con Docker Compose, avendo cura di specificare il Compose file appropriato.

Considerazioni finali

Nel corso dell'attività oggetto di questo documento, abbiamo lavorato alla piattaforma destinata ad ospitare applicazioni intelligenti per l'industria digitale del progetto SBDIOI40. Abbiamo sviluppato due soluzioni che contribuiscono all'uso della piattaforma nelle principali modalità secondo cui le applicazioni possono essere implementate: con macchine virtuali o con software container. Nel caso delle macchine virtuali, abbiamo semplificato la migrazione di un'applicazione tra due cloud OpenStack di pertinenza del progetto attraverso lo sviluppo di una libreria che ne consente la gestione con grande facilità; nel caso dei container, invece, abbiamo lavorato a un sistema per accelerare la fase di avvio dell'applicativo, e quindi ridurre la latenza anche nel caso della migrazione di un'applicazione a microservizi.

Restano alcuni punti aperti per eventuali perfezionamenti. Per quanto riguarda la migrazione di applicazioni tra piattaforme OpenStack, ad esempio, sarebbe sicuramente utile implementare la gestione dei flavor, dei security group, dei floating IP e delle chiavi SSH. Dal punto di vista tecnico, dovrebbe trattarsi di estensioni relativamente semplici da adottare visto il buon grado di modularità del software che è stato sviluppato. Ciononostante, occorrerà dirimere alcune questioni dal punto di vista delle convenzioni. Ad esempio, relativamente ai flavor: sarebbe opportuno scegliere un insieme di configurazioni predefinite che devono essere presenti su tutte le piattaforme? Oppure sarebbe meglio conservare la possibilità di assegnare a una VM un qualunque flavor personalizzato? E, in questo caso, che conseguenze potrebbe avere la scelta di assicurare denominazioni univoche ai flavor attraverso molteplici piattaforme?

Per quanto riguarda, invece, l'avvio veloce di applicazioni a microservizi, potrebbe essere utile automatizzare ulteriormente la creazione della versione personalizzata dell'applicazione. Come già visto, la creazione di tale versione personalizzata a partire da quella originale passa attraverso un processo di profilazione che ha come obiettivo quello di individuare, per ciascun servizio, i file essenziali per le prime fasi di esecuzione. La profilazione prevede una fase di simulazione in cui l'applicazione in oggetto viene sollecitata per simulare un caso d'uso tipico. Allo stato attuale, autoprof consente di avviare e fermare la profilazione, ma, tra queste due operazioni, lascia all'utente la responsabilità di eseguire la simulazione. Si potrebbe estendere autoprof al fine di accettare in ingresso un modulo aggiuntivo che incapsuli la logica di simulazione (come ad es. un eseguibile o uno script shell). In questo modo l'automazione del profilamento (e quindi, della generazione della versione personalizzata dell'applicazione) risulterebbe molto più completa.