Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Implementazione di un sistema di monitoraggio grey-box per microservizi e cluster Kubernetes

Relatore:
Chiar.mo Prof.
DAVIDE ROSSI

Presentata da:
FABIO TRIGARI

Sessione III
Anno Accademico 2023-2024

Introduzione

Negli ultimi anni con la diffusione dell' architettura a microservizi si è creata sempre più l'esigenza di avere un sistema di monitoraggio che permetta di verificare in tempo reale lo stato dell'intero sistema. Il monitoraggio è fondamentale in un contesto enterprise perché con le giuste metriche è possibile risolvere e persino anticipare gli incidenti in tempi brevi. La tipologia di monitoraggio che negli ultimi anni ha preso il sopravvento è denominata white-box; questo recupera da ogni microservizio tutte le informazioni che riguardano lo stato dell'applicativo. Inoltre, con l'uso di altri sistemi che si integrano perfettamente con questa tipologia di monitoraggio, è possibile mostrare tutti i microservizi richiamati da una pagina web con il relativo tempo impiegato di risposta per ciascuno di essi.

È presente una seconda tipologia di monitoraggio denominata black-box. Questa effettua chiamate periodiche che simulano l'attività dell'utente e ad ogni chiamata viene verificata la correttezza della risposta. Questa tipologia è meno utilizzata poichè non prevede l'insorgere di problemi ma verifica che in un dato periodo temporale l'intera infrastruttura funzioni correttamente. Tuttavia, in un'architettura a microservizi, utilizzare un approccio black-box potrebbe essere molto complicato. Alcuni microservizi potrebbero non interagire mai con il monitoraggio dovuto alla natura stessa dell'architettura utilizzata e potrebbero insorgere problemi di performance, dovuti al numero eccessivo di chiamate.

Il progetto svolto sfrutta una metodologia ibrida conosciuta come grey-box monitoring, che concede la libertà sia di monitorare il sistema esattamen-

te come definito dalla metodologia black-box, sia di monitorare direttamente le applicazioni senza sovvraccaricare il sistema.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

SRE

Site reliability engineering (SRE) è una disciplina nata nel 2003 all'interno di Google sotto la direzione di Ben Treynor Sloss, che aveva il bisogno di rendere affidabili i servizi messi a disposizione a fronte delle innumerevoli richieste da tutto il mondo, nonostante all'epoca fossero presenti connessioni dial-up da 56 kbit/s [1, 2]. Inoltre, Google è stata la prima azienda con un'infrastruttura distribuita massiva e gli necessitava un modo per automatizzare e mantenere monitorato su larga scala tutti i server. Treynor ha così creato il SRE, un modello che condivide molti concetti con il DevOps.

1.1 Che cosè SRE

Sono presenti notevoli definizioni di site reliability engineering, ma ogni definizione presente sul web è riconducibile a quella di David N. Blank-Edelman, rendendola così la più completa:

"*Site reliability engineering è una disciplina dedicata ad aiutare le organizzazioni a raggiungere in modo sostenibile il livello appropriato di affidabilità nei loro sistemi, servizi e prodotti*"[3].

Tuttavia, le altre definizioni evidenziano un aspetto chiave che in quella di Taylor risulta meno esplicito, ovvero l'automazione. Per Amazon Web Services (AWS)[4], SRE è la pratica di utilizzare strumenti software per au-

tomatizzare le attività dell'infrastruttura IT come la gestione del sistema e il monitoraggio delle applicazioni, mantenendo le applicazioni affidabili ai rilasci da parte del team di sviluppo. Questa definizione si avvicina notevolmente al concetto di DevOps, ma sono presenti delle differenze che permettono di mantenere separati i due modelli.

Le parole chiavi in queste definzioni sono `affidabilità` e `sostenibilità`. Nonostante possano sembrare concetti distinti, in realtà sono molto legati tra loro. Di seguito verrà mostrata la loro correlazione.

1.2 Affidabilità

L'`affidabilità` non garantisce solamente la disponibilità di un servizio, ma anche la sua raggiungibilità da parte di un utente e la correttezza della risposta fornita. Quindi, anche se un servizio risulta disponibile, potrebbe non essere affidabile [5]. Avere a che fare con problemi con l'affidabilità comporta gravi conseguenze:

1. **Reputazione**: gli utenti necessitano un servizio affidabile e, in caso contrario, cercano alternative;

2. **Tempo**: le risorse che impiegano tempo a risolvere i problemi riscontrati stanno utilizzando il loro tempo non come previsto, di conseguenza è un costo per l'azienda

3. **Salute**: essere sotto pressione ogni ora del giorno comporta gravi problemi di salute come il burnout inoltre, le risorse con la reperibilità impiegano il loro tempo privato a risolvere problemi;

4. **Licenziamenti e nuove assunzioni**: le risorse se sono costantemente in allarme, cercheranno altre aziende. Inoltre, la reputazione che l'azienda si crea dal punto di vista lavorativo, andrà a inficiare negativamente sulle nuove assunzioni.

1.3 Sostenibilità

Il termine `sostenibilità` nel SRE indica tutti i modi per rendere un sistema affidabile nel tempo. La sostenibilità è garantita quando viene eliminato o minimiazzato lo sforzo umano e per ridurlo è necessario automatizzare il più possibile tutto il processo[6].

Un processo operativo, affinchè funzioni correttamente, deve essere sostenibile. Se così non fosse, il risultato ottenuto nel lungo periodo comporta gravi rischi per la persona. Quindi, quando un sistema funziona correttamente affinchè sia sostenibile? Questa domanda trova risposta nella citazione di Carla Geisser:

"*Se un operatore umano deve mettere mano nel tuo sistema durante il normale processo, hai un bug. La definizione di normale cambia all'aumentare del sistema*".

Il fatto che una persona debba modificare costantemente qualcosa per portare a termine le operazioni di routine, porta a svolgere del lavoro inutile e nel lungo periodo si trasforma in `sforzo`. Si definisce `sforzo` tutto il lavoro manuale e ripetitivo che può essere automatizzato. È doveroso eliminare il più possibile lo sforzo umano, perchè:

- **Mancata crescita della carriera**: non dedicare tempo nella creazione e gestione dei progetti, comporta un rallentamento nella carriera;
- **Costo aziendale**: spendere tempo su un processo automatizzabile comporta un costo per l'azienda e tale costo aumenta proporzionalmente alla crescita di quest'ultima;
- **Creazione di attriti**: produrre molto sforzo comporta anche la generazione di attriti tra i colleghi, poichè nessuno vuole svolgere attività ripetitive.
- **Scarsa motivazione**: un lavoro ripetivo risulta noioso e questo inficia negativamente sul morale delle persone.

1.4 Affidabilità in pratica

Il risultato atteso da tutte le persone è un sistema affidabile al 100%, quindi che sia sempre disponibile e che risponda correttamente. Tuttavia, anche se un servizio può essere altamente affidabile, esistono dei fattori esterni che possono causare disservizi, come interruzione della rete elettrica, guasti sulla rete internet e disastri naturali. Avere un sistema affidabile al 100% è impossibile quindi è necessario andare incontro al rischio e rendere un sistema affidabile ad una percentuale inferiore seppur alta. La disponibilità è la metrica chiave per l'affidabilità, infatti un sistema è affidabile quando ha un'alta percentuale di disponibilità e non sono presenti errori sul software.

Per avere un livello di disponibilità elevato è necessario avere un sistema ridondato, quindi acquistare macchine su cui duplicare l'esecuzione delle applicazioni. Per avere una maggiore disponibilità è necessario che i server siano anche dislocati. I colossi digitali del cloud, mettono a disposizione dei piani per avere sistemi ridondanti su più zone e region. Purtroppo, nel caso cloud e in quello on-premise, il costo è elevato.

Inoltre, per avere un sistema affidabile, è necessario che anche lo stesso codice lo sia. In media uno sviluppatore crea 70 bug ogni 1000 righe di codice[7]. Per ovviare a questo problema è necessario rallentare lo sviluppo di nuove feature per risolvere i problemi esistenti e prevernirli. Tuttavia, questo significa che nel medio lungo periodo potrebbe causare problemi di mancata innovazione e perdita di clienti.

Per Google, produrre un software affidabile ha un costo che cresce esponenzialmente rispetto alla linearità dell'incremento dell'affidabilità.

1.5 Il prezzo da pagare

Oltre ai costi da sostenere per rendere un servizio disponibile, è necessario definire anche quanto questo debba esserlo.

Un sistema molto affidabile avrà poche feature, di conseguenza potrebbe perdere di competitività e ridurre il numero di utenti. Lo stesso risultato vale

per un sistema con molte feature, questo comporta che gli sviluppatori siano concentrati sullo sviluppo di nuove funzionalità, trascurando il bug fixing e creazione di test, riducendo di fatto l'affidabilità.

Il compromesso tra affidabilità e nuove funzionalità è definito dal responsabile di prodotto, sarà lui a definire il livello di rischio.

Nel corso di un determinato periodo di tempo, se l'affidabilità è maggiore rispetto alla soglia predefinita, si può procedere con lo sviluppo di nuove feature altrimenti è necessario aumentare l'affidabilità con test e risoluzione di bug.

1.6 SLA SLO SLI

Nel contesto dell'affidabilità dei servizi, è fondamentale definire metriche e accordi che regolino le aspettative tra i fornitori di servizi e i clienti. Questo avviene attraverso tre concetti chiave: **Service Level Agreement (SLA), Service Level Objective (SLO) e Service Level Indicator (SLI)**.

1.6.1 Service Level Agreement (SLA)

Il **Service Level Agreement (SLA)** è un contratto formale tra il fornitore di un servizio e il cliente che stabilisce le aspettative relative alla qualità del servizio. Lo SLA definisce [8, 9]:

- **Descrizione del servizio offerto** e i relativi dettagli;
- **Metriche**, i criteri utilizzati per misurare le prestazioni del servizio;
- **Livelli minimi di servizio**;
- **Sanzioni** in caso di mancato rispetto degli standard definiti;

Gli SLA sono particolarmente importanti nei servizi cloud e nei sistemi distribuiti, dove il downtime o una riduzione delle prestazioni possono avere costi significativi per il cliente.

1.6.2 Service Level Objective (SLO)

All'interno dello SLA, viene definito il **Service Level Objective (SLO)**. Sono gli obiettivi tecnici specifici e misurabili che devono essere mantenuti per tutta la durata del contratto. Ad esempio, uno SLO può stabilire che un servizio deve essere raggiungibile al 95% del tempo.

Gli SLO vengono utilizzati internamente dalle aziende per monitorare le prestazioni del servizio e supportare le decisioni strategiche sullo sviluppo del software, garantendo comunque il livello di affidabilità concordato con il cliente.

1.6.3 Service Level Indicator (SLI)

Il **Service Level Indicator (SLI)** è la metrica utilizzata per misurare il raggiungimento degli SLO. Gli SLI rappresentano la qualità del servizio come:

- La percentuale di richieste HTTP servite con successo ;
- La latenza;
- Il numero di richieste eseguite in un determinato periodo di tempo.

Nella tabella sottostante 1.1, viene riportata in formato percentuale la disponibilità e nelle colonne adiacenti il relativo disservizio in un periodo temporale annuale e mensile.

1.7 Come misurare la disponibilità

Una formula che viene impiegata per calcolare la disponibilità di un servizio è la seguente:

$$Availability = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}$$

Availability %	Disservizio annuale	Disservizio mensile
90%	36,5 giorni	3 giorni
99%	3,6 giorni	7,2 ore
99,9%	8,7 ore	43 minuti
99,99%	52,6 minuti	4,3 minuti
99,999%	5,2 minuti	25,9 secondi

Tabella 1.1: Tabella della disponibilità

La disponibilità è data dal tempo in cui l'applicazione è disponibile e lo si divide per il tempo totale che corrisponde alla somma del tempo attivo e del downtime. Questa formula, nonostante sia corretta, non tiene in considerazione casi più complessi, come la presenza di server ridondanti e l'erogazione del servizio su molteplici stati la cui indisponibilità può essere mitigata o solo circoscritta ad una sola zona.

Per avere un risultato più preciso è necessario adottare un'altra formula che non fa riferimento alla disponibilità nel tempo ma alla disponibilità in base alle richieste:

$$Availability = \frac{\text{Successful requests}}{\text{Successful requests} + \text{Bad requests}}$$

Per calcolare la disponibilità sulle richieste, vengono recuperate tutte le chiamate con esito positivo e vengono divise per il numero totale di chiamate, che corrisponde al numero totale di richieste positive sommate al numero di richieste negative.

1.8 Differenza tra SRE e DevOps

Nella definizione di SRE è emerso il termine `operations` per indicare l'eliminazione dello sforzo in favore dell'automazione, del monitoraggio delle metriche, della gestione delle configurazioni e tanto altro. Gli stessi termini vengono usati a loro volta anche in ambito DevOps [10], ma in questo caso i

vari significati si riferiscono all'automazione, al monitoraggio, configurazione e gestione dell'infrastruttura legata allo sviluppatore. Il DevOps engineer è responsabile del processo di rilascio dell'applicazione, dall'ambiente di sviluppo fino alla produzione, e gestisce l'infrastruttura dal punto di vista applicativo, mentre il SRE si occupa di garantire all'utente che il sistema sia affidabile.

I ruoli SRE e DevOps sono molto vicini tra loro nonostante abbiano clienti diversi, rispettivamente gli utenti finali per SRE e gli sviluppatori per il DevOps.

1.9 Gerarchie del SRE

Google, per facilitare l'adozione del modello SRE da parte delle aziende, ha creato una piramide gerarchica 1.1 che descrive i livelli fondamentali per la gestione di un software affidabile, basandosi sui principi del SRE.

1. **Monitoring**: se l'applicazione non viene monitorata non è possibile sapere in che stato essa si trova. Non ha senso sviluppare un'applicazione e non essere a conoscenza dei possibili problemi che questa può avere. In conclusione, non monitorare significa non sapere cosa sta accadendo all'applicazione a runtime e aspettare che sia l'utente a segnalare l'errore. Inoltre, eseguire il debug su un'applicazione senza sapere il motivo per cui ci sono stati degli errori o riavvii anomali, comporta un'enorme difficoltà e una grande perdita di tempo. Per esempio in un'applicazione Java è possibile incorrere a riavvi indesiderati dovuti alla memoria, se non si hanno a disposizione le metriche, l'unica soluzione è tamponare il problema aumentando la memoria RAM, ma è impossibile trovare la causa scatenante, quindi il problema potrebbe ripresentarsi.

2. **Incident Response**: viene fatto riferimento su come gestire l'incidente, quindi innanzitutto capire cosa non stà funzionando attraverso tutti i dati che si hanno a disposizione. Non importa che il problema venga

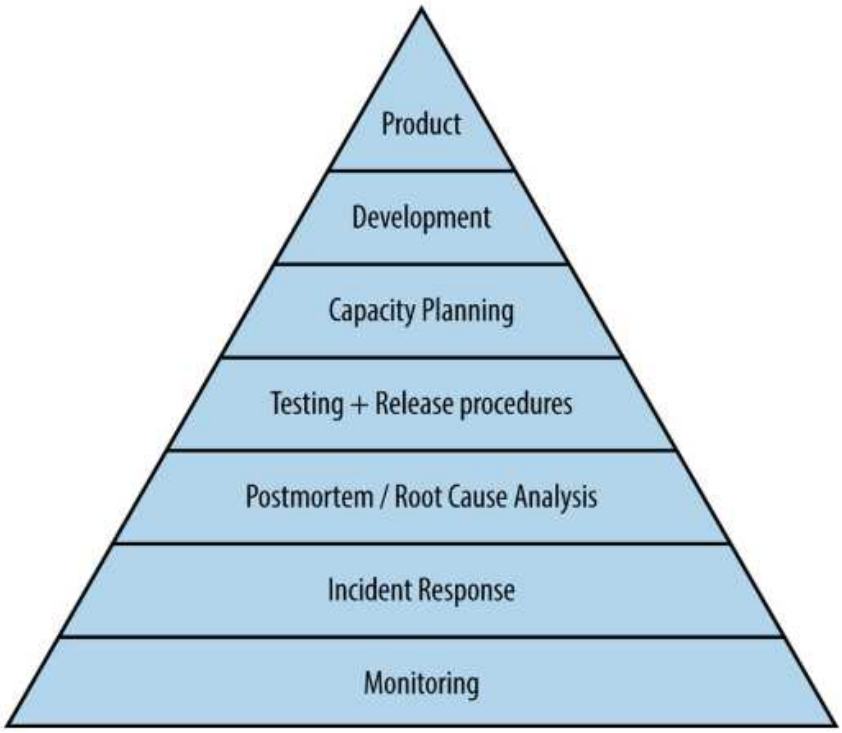

Figura 1.1: Gerarchie del SRE https://sre.google/sre-book/part-III-practices/

risolto definitivamente nel più breve tempo possibile. Se le cause sono ignote è anche sufficiente mitigarlo. La priorità degli incidenti si basa sull'applicativo e dal contratto definito con l'utente.

3. **Postmortem**: scrivere un documento che riassuma cosa è successo e i passi svolti per risolvere il problema. Questo è fondamentale per avere tempi di risoluzione più veloci ed evitare, nel caso in cui lo stesso problema si ripresenti, che il tempo speso per trovare la soluzione sia stato vano.

4. **Testing**: se si è verificato un problema è necessario assicurarsi che non si ripresenti, migliorando i test, aggiungendo unit test e/o integration

test.

5. **Capacity planning**: è un processo di pianificazione per testare carichi di lavoro senza un degrado delle performance [11].

6. **Development**: lo sviluppatore integra programmi che permettono di testare l'affidabilità e la resistenza ai guasti.

7. **Product**: è prevalentemente legato alla scalabilità del prodotto e alla sua robustezza in ambiente di produzione.

1.10 Monitoraggio

Ai fini della tesi verrà analizzato in dettaglio solo il campo Monitoring. Spesso, quando si parla di monitoraggio, si pensa ad un sistema che recuperi tutte le metriche dei servizi in esecuzione come le risorse utilizzate, il numero di richieste ricevute, la latenza, etc. ma non è così. Nelle prossime sezioni verranno definite le principali tipologie di monitoraggio.

1.11 Tipologie di monitoraggio: white-box vs black-box

Sono presenti due tipologie di monitoraggio che sono rispettivamente white-box e black-box. Il monitoraggio white-box, che probabilmente è il più conosciuto, permette di recuperare tutte le metriche di un'applicazione come le risorse impiegate e il traffico. Inoltre, ogni linguaggio di programmazione ha le proprie metriche. In Java, per esempio, è possibile recuperare lo stato dello heap, del garbage collector, e se presente anche il numero di connessioni attive sul database, oltre ad altre informazioni rilevanti per l'analisi delle prestazioni. Se sull'intera architettura è presente `OpenTelemetry (OTel)`[12], è possibile recuperare anche lo stato di tutte le chiamate interne,

la latenza, il tempo impiegato per eseguire una query sul database e verificare dove la richiesta ha impiegato maggior tempo.

Il monitoraggio white-box si concentra sulla singola applicazione, infatti, chi esegue l'analisi sui dati ricevuti avrà una visione dettagliata di un insieme di applicazioni all'interno del sistema. Tra le più usate è presente Prometheus che recupera le metriche esposte da ogni singolo servizio e offre una dashboard basilare su cui è possibile eseguire le query, gestendo così il monitoraggio white-box. Spesso i dati raccolti da `Prometheus` [13]vengono utilizzati da strumenti dedicati alla creazione di dashboard. Tra i più noti è presente `Grafana` [14], che permette di recuperare le informazioni da diverse fonti di dati e analizzare le metriche e statistiche attraverso dashboard personalizzate.

Molti produttori software mettono a disposizione delle dashboard per facilitare il monitoraggio delle applicazioni da parte dei clienti, offrendo così una visione più chiara e nitida delle loro prestazioni all'interno dell'intero sistema.

Il monitoraggio black-box è una tecnica che simula il comportamento degli utenti eseguendo periodicamente delle chiamate che imitano l'attività di un utente e che vanno a testare la disponibilità del sistema, la latenza e lo stato della risposta in un determinato momento. Questo processo è noto anche come `synthetic monitoring`[15], sebbene quest'ultimo abbia un ambito più ampio, in quanto è utilizzato anche per eseguire test end-to-end e simulare flussi di richieste. Nei prossimi capitoli verrà definita una lista di software che permettono di fare black-box monitoring.

L'approccio white-box prevede gli errori, mentre il monitoraggio black-box verifica se in un dato istante sono presenti disservizi. In un sistema di monitoraggio completo vengono utilizzate entrambe le tipologie.

1.12 Monitoraggio: attivo e passivo

In base allo strumento utilizzato per monitorare le applicazioni, emergono due caratteristiche di monitoraggio [16]:

- `monitoraggio attivo`: un sistema che esegue delle chiamate al servizio per simulare il traffico cliente, creando dati che non corrispondono completamente a quelli reali, come potrebbe fare syntethic-monitoring.

- `monitoraggio passivo`: i dati raccolti all'interno del sistema corrispondono a quelli degli utenti reali.

Fare analisi sui dati reali è meglio rispetto a dati spuri, tuttavia con il monitoraggio attivo è possibile sapere con certezza lo stato dell'intero sistema in un dato momento anche se non è presente traffico.

1.13 Le metriche da considerare

Il metodo `Utilization Saturation and Errors (USE)`[17] sviluppato da Brendan Greg permette, monitorando solo alcune metriche, di trovare errori di performance in breve tempo. Per ogni risorsa, vengono recuperate le seguenti metriche:

- `utilization`: il tempo medio delle risorse impegnate (cpu, memoria, dischi, bus ...);

- `saturation`: la capacità di lavoro che il sistema sta gestendo ma non può essere processata immediatamente e di conseguenza è messo in coda;

- `errors`: il numero di eventi in errore.

Tuttavia, questa metodologia è ottima per le performance legate alla componentistica hardware, ma non in un'architettura a microservizi.

Tom Wilkie ha sviluppato il metodo `RED`[18, 19] che per ogni risorsa monitora:

- `rate`: il numero di richieste per secondo;

- `errors`: il numero di richieste fallite;

- `duration`: il tempo impiegato dalle richieste per essere servite.

Invece, Google ha le `four golden signals` che sono come la metodologia RED ma con l'aggiunta della saturazione:

- `traffic`: il numero di richieste per secondo;
- `errors`: il numero di richieste che sono fallite;
- `latenza`: il tempo impiegato delle richieste per essere servite;
- `saturazione`: la capacità di lavoro che il sistema sta gestendo ma non può essere processata immediatamente e di conseguenza è messo in coda.

1.14 Monitoring vs Observability

Spesso il monitoring e observability vengono confusi, ma hanno una differenza nel risultato ottenuto. Il monitoring permette di recuperare le metriche per verificare lo stato del sistema in un dato intervallo temporale. Esso permette di comprendere cosa è successo e quando è avvenuto, ma non fornisce ulteriori informazioni.

L'observability[20], aggregando tutti i dati di un sistema come le metriche, i log, il trace, permette di fare analisi più approfondite, trovando la motivazione per cui è avvenuto un problema.

OpenTelemetry, nonostante utilizzi il monitoraggio white-box per recuperare le informazioni, è un componente fondamentale dell'osservabilità.

Capitolo 2

Kubernetes

Il sistema che si vuole realizzare aiuta ad avere un maggiore controllo sullo stato delle applicazioni di Kubernetes, prima di proseguire è necessario acquisire una conoscenza minima del dominio.

2.1 L'evoluzione dei microservizi

Prima di parlare di Kubernetes, è necessario vedere come e perchè l'infrastruttura sia passata da un'installazione delle applicazioni direttamente sul sistema operativo, alla virtualizzazione con Kubernetes[21].

Il deployment tradizionale delle applicazioni avveniva direttamente sul sistema operativo. Avendo le risorse hardware condivise potevano verificarsi notevoli problemi, come l'utilizzo inaspettato delle risorse da parte di un'applicazione e la condivisione di librerie, dipendenze e dati, impattando negativamente sulle performance, sulla sicurezza delle applicazioni e sui comportamenti indesiderati.

Come soluzione venne proposta l'uso delle `macchine virtuali (VM)`. Su ogni macchina viene creato un hypervisor che virtualizza i sistemi operativi e al loro interno viene installata la singola applicazione isolata dalle altre. Questo comporta maggiore sicurezza perchè i file non sono più accessibili esternamente e le risorse instanziate per una macchina virtuale vengono

utilzzate dal solo sistema operativo e dall'applicazione in esecuzione. Tuttavia, le VM creano astrazioni di sistemi operativi con il conseguente overhead di risorse.

Il container runtime permette di eseguire più applicazioni, astraendo a loro il sistema operativo. In questo modo viene garantito lo stesso livello di sicurezza in quanto isolato.

I container runtime hanno portato innumerevoli vantaggi tra cui la suddivisione delle responsabilità. Sono state frammentate le applicazioni monolite in tante piccoli applicazioni. Questo approccio ha fatto crescere l'adozione dei microservizi, un'architettura altamente scalabile e flessibile, dando maggior responsabilità al singolo sviluppatore. Allo stesso tempo, questa architettura crea un overhead di chiamate tra microservizi. L'aggiornamento, anche di una libreria, all'interno di un intero sistema, richiede molto tempo.

I vantaggi di un'architettura a monolite sono gli svantaggi di una a microservizi e viceversa.

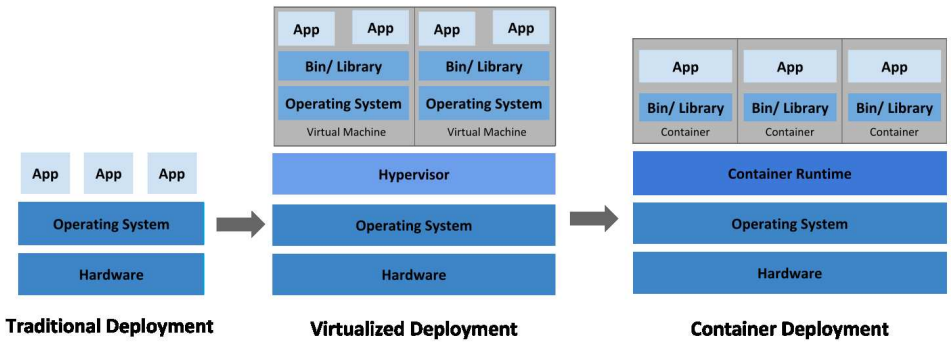

Figura 2.1: Evoluzione del deployment

2.2 Che cos'è Kubernetes

Kubernetes è un orchestratore di container che permette di gestire il ciclo di vita e il rilascio delle applicazioni, il load balancing, l'autoscaling e

l'allocazione di risorse su più nodi (server). A dimostrazione della sua maturità, Kubernetes si è affermato da anni a livello enterprise ed è un progetto `graduated` all'interno della `Cloud Native Computing Foundation (CNCF)`.

2.3 Gli oggetti di Kubernetes

Gli oggetti di Kubernetes sono tutti i componenti che possono essere creati e gestiti all'interno di un cluster Kubernetes. Sebbene esistono numerosi oggetti, di seguito verranno trattati i più rilevanti e quelli citati.

Il `namespace` è un raggruppamento virtuale per isolare le risorse e gli oggetti associati ad un'area tecnica o team all'interno di un cluster. Al suo interno è possibile trovare i seguenti componenti:

- `Pod`
- `ReplicaSet`
- `Deployment`
- `StatefulSet`
- `Secret e ConfigMap`
- `Volume`
- `Service`
- `Ingress`

Il `pod` è l'oggetto più 'piccolo' di Kubernetes e, attraverso i `container`, permette di eseguire l'applicazione. È possibile eseguire più container all'interno di un singolo pod, ma questi devono essere funzionali all'applicazione principale. Per ogni container, è possibile definire variabili d'ambiente, le risorse minime e montare volumi, secret e configMap.

Il `replicaSet` permete di avere più pod per la stessa applicazione, garantendo la disponibilità e la scalabilità del servizio.

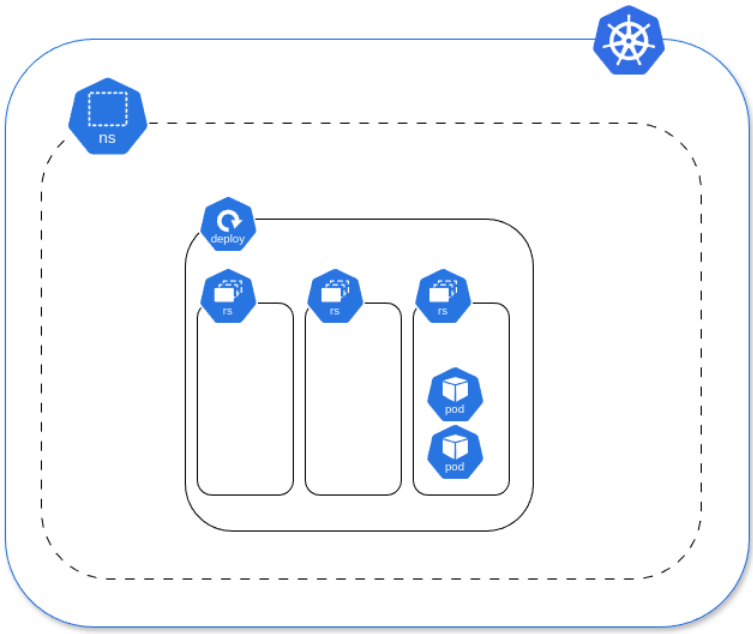

Figura 2.2: Gerarchia deployment

Infine, in "cima alla gerarchia", è presente il `deployment`, che gestisce il rilascio delle applicazioni tramite i replicaSet, permettendo il controllo delle versioni e degli aggiornamenti. Un deployment mantiene attivo solo un replicaSet, tranne quando viene aggiornato il deploy che in quel caso manterrà due replicaSet attivi temporaneamente. Quando il rilscio è terminato, il replicaSet precedente non avrà più alcun pod disponibile. Nella figura 2.2 viene mostrato un deployment con tre replicaSet e solamente uno dei tre è attivo con due pod.

Non sono solo i deploy a generare i pod, ma anche altri come `Job` e `StatefulSet`.

I StatefulSet sono oggetti particolari in Kubernetes, progettati per gestire applicazioni stateful, ossia quelle che mantengono uno stato persistente. Essi gestiscono il numero di repliche dei pod, garantendo che ogni replica sia assegnata a un nome prevedibile, che persiste anche dopo il riavvio dello stesso.

ConfigMap e **Secret** sono oggetti al cui interno vengono salvate le configurazioni di un'applicazione come la porta del server, il livello dei log. Generalmente nelle secret vengono salvate le credenziali del database o di accesso, ma non è buona pratica adottare solo le secret come mezzo di sicurezza. Questi oggetti possono essere legati al container tramite variabili d'ambiente o come file in un path specifico.

I **volumi** sono oggetti su cui è possibile leggere e scrivere i dati garantendo la persistenza in caso di riavvio dei pod. Questi vengono montati come path su un percorso specifico del container.

In un'architettura a microservizi è fondamentale che i pod si scambino informazioni tra loro, ma non è possibile utilizzare il loro indirizzo IP perchè sono dinamici e per la loro natura temporanea risulta estremamente complicato scambiarsi informazioni. Per poter contattare i pod è necessario utilizzare l'oggetto `service` che permette di associare un IP statico ad un gruppo di pod attraverso le label dichiarate. I pod possono comunicare tra loro all'interno dello stesso namespace utilizzando come dominio direttamente il nome del service e la relativa porta. Mentre, per poter comunicare al di fuori, è necessario utilizzare come dominio il service host che è così formato `<nome-service>.<ns-dest>.svc.cluster.local`. Se sono presenti delle restrizioni di rete, è necessario creare delle `NetworkPolicy`. Nella 2.3 è riportato un esempio di chiamate tra namespace. Per i pod dei statefulset è presente un service particolare definito `headless` perchè non necessità di load-balancing e a cui non è assegnato un IP. Questo permette di richiamare un pod specifico del statefulset passando dal service. L'URL è così formata `<nome-del-pod>.<nome-service-ha>.<ns-dest>.svc.cluster.local` .

Esistono 3 tipologie di Service:

1. **ClusterIP**: visibile solamente all'interno del cluster;

2. **NodePort**: il cluster espone una porta ed è possibile raggiungere il service utilizzando l'IP del cluster con la porta generata;

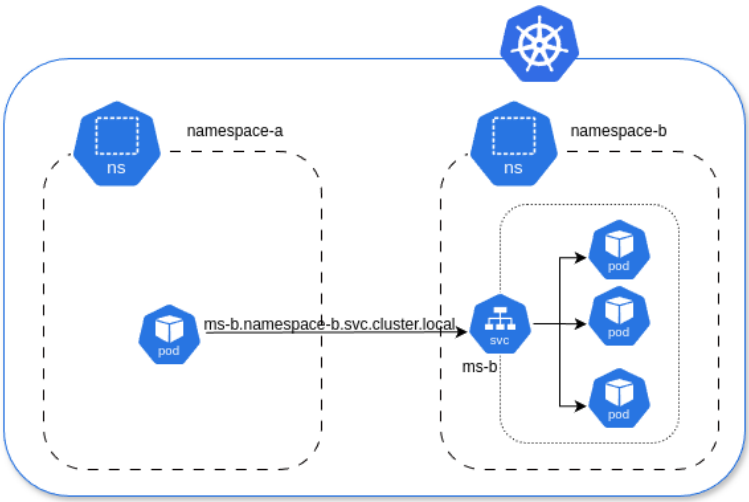

Figura 2.3: Esempio di comunicazione tra namespace

3. **LoadBalancer**: gestisce il bilanciamento delle richieste sui pod ed espone IP pubblici.

I service espongono una singola applicazione. Per esporre un sistema vengono utilizzati gli ingress, che forniscono un punto di ingresso centralizzato per il traffico.

Gli altri oggetti citati sono:

- **ValidatingWebhookConfiguration**: invoca API al cambio di stato di uno o più oggetti di Kubernetes, tuttavia necessita di un certificato valido per invocare le API;
- **podMonitor** e **serviceMonitor**: oggetti messi a disposizione da Prometheus che raccolgono le metriche dai pod o dai service.

Capitolo 3

Sistemi esistenti

In un contesto a microservizi su Kubernetes, se per il monitoraggio white-box l'intera community è riuscita ad individuare alcuni strumenti per recuperare i dati e analizzarsi come Prometheus e openTelemetry, per il monitoraggio black-box non c'è nessuna applicazione leader di mercato, forse dovuto ad una pratica di monitoraggio poco utilizzata oppure perchè molteplici prodotti offrono questa tipologia di servizio direttamente nei propri sistemi.

Sono presenti notevoli soluzioni enterprise per gestire il monitoraggio black-box, i più conosciuti sono:

1. `Black-box exporter`
2. `Cloudprober`
3. `Zabbix`
4. `Nagios`
5. `New Relic`
6. `Datadog, Dynatrace`

Alcuni dei software qui citati sono completamente open source e possono essere utilizzati a livello enterprise, invece altre applicazioni hanno un costo

per l'utilizzo commerciale ma permettono di avere un monitoraggio completo in pochissimo tempo.

Di seguito verranno analizzati tutti i software citati.

3.1 Black-box exporter

Black-box exporter [22] è uno strumento sviluppato da Prometheus che permette di eseguire verifiche periodiche utilizzando lo stesso Prometheus. È possibile effettuare chiamate sia a URL statici ed esterni al cluster, sia per chiamare service e pod interni. Questi vengono recuperati dinamicamente utilizzando le labels. Questo strumento funge solo da proxy, infatti è Prometheus che esegue una richiesta http con tutti i parametri necessari a black-box exporter affinchè possa fare da proxy eseguendo una chiamata con i dati presi in input e rispondere a Prometheus con le metriche ottenute. Prometheus salva il contenuto della risposta e possono essere analizzate direttametne su Prometheus oppure attraverso dashboard Grafana.

Se per chiamare URL statici è necessario modificare la configurazione di Prometheus tramite configMap, per eseguire la probe a URL dinamici è più complicato perchè è necessario creare uno dei due oggetti Kubernetes di Prometheus che sono i serviceMonitor o i podMonitor. Questi consentono a Prometheus di eseguire lo scrape delle metriche a intervalli prestabiliti, su un path specifico esposto dai pod o dai service, a seconda dell'oggetto di Kubernetes scelto. L'acquisizione delle metriche avviene esclusivamente per gli oggetti le cui label corrispondono a quelle definite nella configurazione di Prometheus.

Black-box exporter può interagire con diversi protocolli ed è altamente personalizzabile. Gli errori vengono gestiti da Prometheus alerting oppure da servizi terzi leggendo comunque le metriche di Prometheus.

Tuttavia, gestire un piccolo cluster con Prometheus e OTel è molto facile, ma dal momento che il numero di servizi da monitorare aumenta, anche le risorse di Prometheus aumentano, rendendolo complicato da gestire. Ag-

giungere l'overhead di black-box exporter potrebbe essere molto più dannoso, rischiando riavvi continui di Prometheus e la mancata raccolta di metriche, eliminando sia il monitoraggio black-box sia quello white-box.

Inoltre, nell'oggetto serviceMonitor o podMonitor, se si vogliono monitorare più path, è necessario creare tanti `endpoint` ognuno con un path differente, andando a generare oggetti di notevole dimensione rendendo difficile la manutenzione e la comprensione. Di seguito è riportato un esempio di serviceMonitor, in cui ogni 10 secondi Prometheus esegue una chiamata a black-box exporter con l'host del servizio definito in __param_target con il path /api-used-by-users.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    release: prometheus
  name: nginx-service-monitor
  namespace: mynamespace
spec:
  endpoints:
  - interval: 10s
    params:
    module:
    - http_2xx
    target:
    - pippo
    path: /probe
    port: http
    relabelings:
    - action: replace
      sourceLabels:
      - __meta_kubernetes_pod_label_app
```

```
      targetLabel: app
    - action: replace
      sourceLabels:
      - __address__
      targetLabel: __param_target
    - action: replace
      replacement: ""
      sourceLabels:
      - __address__
      - __param_target
      targetLabel: __address__
    - action: replace
      replacement: >|
       "blackbox-exporter-blackbox-exporter.
           monitoring.svc.cluster.local:9115"
      sourceLabels:
      - __param_target
      targetLabel: __address__
    - action: replace
      replacement: http://${1}/api-used-by-users
      sourceLabels:
      - __param_target
      targetLabel: __param_target
  selector:
    matchLabels:
      app: nginx
```

3.2 Cloudprober

Cloudprober [23] è un microservizio sviluppato da Google che monitora un livello notevole di applicaziozioni dislocate tra Kubernetes, macchine virtuali, host terze parti, utilizzando protocolli differenti. Espone anche delle metriche con le quali sono state create dashboard su molteplici vendor tra cui Grafana. Inoltre, è possibile configuralo per inviare notifiche in caso di errori[24].

Nonostante sia possibile installare Cloudprober tramite Docker o come applicazione sul sistema operativo, verrà focalizzato il solo caso con l'installazione su Kubernetes, in quanto è l'unico modo per gestire in maniera ottimale la probe delle applicazioni interne al cluster.

Cloudprober mette a disposizione due modi per monitorare:

1. tramite URL statici: adatti principalmente al monitoraggio di servizi che non risiedono all'interno del cluster
2. tramite URL dinamici: adatti per monitorare oggetti che risiedono all'interno del cluster. Nell'esempio seguente è possibile vedere come avviene il monitoraggio di tutti gli endpoint il cui nome termina con '-service'.

```
targets {
    k8s {
        endpoints: ".*-service"
    }
}
```

Cloudprober presenta anche dei limiti:

- lo sviluppatore deve creare un API di monitoring per CloudProbe, per esempio "/api/monitoring";
- non è possibile utilizzare path diversi di monitoraggio senza dover contattare il team che gestisce CloudProbe.

La soluzione fornita è molto interessante ma i limiti che presenta non permettono di effettuare un monitoraggio efficiente direttamente sul service.

3.3 Zabbix

Zabbix[25] è una soluzione open source che permette di monitorare l'intera infrastruttura IT ed è dotato di un agent Kubernetes che permette di recuperare tutte le metriche relative al cluster. Zabbix è dotato di UI e tramite ad essa è possibile configurare i syntethic monitoring di URL ma non di service Kubernetes. Zabbix è utilizzato principalmente per fare monitoraggio di infrastruttura più che di applicazioni. Lo svantaggio principale di Zabbix è che deve essere installato su una macchina virtuale e quindi è necessario crearla e mantenerla. Inoltre, risulta essere molto complicato nonostante abbia una UI.

3.4 Nagios

Nagios[26] è uno dei sistemi di monitoraggio black-box più vecchi, e dà la possibilità di scrivere plugin custom. Questo gli ha permesso di avere una community molto attiva, dandogli la possibilità di rimanere aggiornato anche su Kubernetes.

Anche se Nagios supporta molteplici protocolli e gestisce molto bene il monitoraggio black-box richiamando URL, non riesce a contattare direttamente i service di Kubernetes perchè, essendo Nagios installato a livello di sistema operativo e dato che non mette a disposizione nessun agent o plugin nativo per Kubernetes, non gli è permesso fare alcun check dei service.

3.5 New Relic

New Relic [27] è il primo dei Application Performance Monitoring (APM) analizzato. Gli APM permettono di fare observability monitorando e verificando lo stato delle applicazioni. Essi prevengono l'assenza di anomalie all'interno del sistema, eseguono diagnosi rapide e l'invio di allarmi in caso di problemi, il tutto con un basso effort in fase di installazione e configurazione.

Tra i concetti principali dei APM si trovano i syntehic monitoring che permettono di verificare attivamente se una serie endpoint risponde correttamente. New Relic è gratuito per i primi 100 GB di dati al mese e poi il prezzo è in relazione ai giga dello spazio occupato. Tuttavia il piano gratuito non mette a disposizione il syntethic monitoring che può risultare una limitazione del prodotto.

3.6 Datadog e Dynatrace

Datadog e Dynatrace [28, 29] sono i leader di mercato tra gli APM ed entrambi danno la possibilità di eseguire il syntethic monitoring, tuttavia, trattandosi di software enterprise, il prezzo potrebbe non essere sostenibile per una piccola o media azienda.

Capitolo 4

Requisiti del nuovo sistema di monitoraggio

I sistemi di monitoraggio esistenti adottano diversi approcci validi, sebbene ciascuno presenti dei limiti. Un ulteriore problema è dato dal monitoraggio black-box in un'architettura a microservizi, in quanto il più delle volte tutte le applicazioni sono protette da un API gateway o da un servizio come Nginx o Apache httpd24 che permettono di fare da reverse proxy e controllare il traffico in ingresso. Eseguire una chiamata per microesrvizio passando da un proxy potrebbe essere dannoso, poichè aumenterebbe la probabilità di sovvracaricare il sistema inutilmente. Per risolvere questo problema si è pensato di monitorare direttamente l'applicazione e solo poche chiamate eseguirle con l'approccio black-box. Dato che tale metodologia non invoca le API esattamente come farebbe un utente, e allo stesso tempo non raccoglie informazioni relative allo stato di un pod, questa metolodiga prende il nome di monitoraggio grey-box.

Allo stesso tempo, se si hanno a disposizione più cluster, si è voluto risolvere il problema di verificare che fossero tutti raggiungibili e funzionanti.

4.1 Monitoraggio delle applicazioni

Si è dunque pensato ad uno strumento molto più semplice e immediato per gestire il monitoraggio black-box sfruttando direttamente i `Service` e gli `Ingress` di Kubernetes, lasciando dunque allo sviluppatore la libertà di mettere sotto monitoraggio le applicazioni e i relativi path.

Greye, il sistema sviluppato, deve essere in grado di recuperare tutti i Service e gli Ingress definiti nel cluster, verificare che sia attiva l'annotation `ge-enabled` che abilita Greye al monitoraggio dell'applicazione e chiamare periodicamente i path, anch'essi definiti nell'annotation `ge-paths`. Per ogni path, è possibile definire il metodo con il quale deve essere invocato.

Un esempio di service è il seguente :

```
apiVersion: v1
kind: Service
  metadata:
  annotations:
    ge-enabled: "true"
    ge-paths: |-
      /path/to/api1
      GET/path/to/api2
      DELETE/path/to/api3
  labels:
    app: my-pod
  name: my-pod
  namespace: my-namespace
spec:
  ...
```

In caso di mancata disponibilità di un servizio o status code minore di 200 o maggiore di 299, Greye ha la possibilità di inviare notifiche push tramite bot Telegram. Inoltre, la presenza di errori è esposta anche come metrica

dando la possibilità di inviare un allarme tramite servizi come Prometheus e Grafana.

4.2 Monitoraggio dei cluster

Greye permette di monitorare la disponibilità delle applicazioni eseguendo chiamate periodiche, ma è necessario poter monitorare che lo stesso cluster sia attivo e funzionante e che non siano presenti sovraccarichi o errori all'interno della rete. Per risolvere questo problema, viene installata un'istanza di Greye su ogni cluster ed ognuna di esse esegue controlli periodici sulle altre. Ogni Greye, sfruttando il `protocollo di gossiping`, verifica la disponibilità solo di un sottoinsieme casuale di altre istanze, se così non fosse si creerebbe un overhead di chiamate inutili e rallentamenti sulla rete e sui pod stessi.

4.3 Requisiti funzionali

Per garantire il corretto funzionamento e la completa fruibilità del sistema, è necessario definire i requisiti funzionali. Questi, descrivono le funzionalità principali che il sistema deve implementare per soddisfare le esigenze di un utente. Di seguito, vengono riepilogati alcuni requisiti già definiti in precedenza per mostrare una visione completa dei servizi offerti.

1. Aggiungere le applicazioni da monitorare: il sistema deve consentire l'aggiunta di applicazioni sia all'avvio sia a runtime;

2. `Verifica periodica dello stato delle applicazioni` così da garantire il corretto funzionamento dei microservizi;

3. Supporto API web differenti: è possibile effettuare chiamate utilizzando HTTP REST o GraphQL;

4. Integrazione con altri protocolli: è possibile implementare altri protocolli, anche personalizzati, per il monitoraggio delle applicazioni;

5. Inviare alert in caso di fallimenti dell'applicazione: segnalare guasti o rallentamenti prima che gli utenti siano impattati;
6. Supporto a diverse piattaforme di notifica: possibilità di inviare notifiche a più piattaforme;
7. Configurazioni specifiche per alcuni microservizi: il sistema deve consentire configurazioni personalizzate ai microservizi che richiedono un monitoraggio più frequente rispetto ad altri oppure su più path;
8. Mostrare grafici di monitoraggio;
9. Sospensione temporanea del monitoraggio: bloccare l'invio di allarmi se le applicazioni sono in manutenzione;
10. Attivabile solamente per il check delle applicazioni: installare Greye per l'utilizzo esclusivo del monitoraggio delle applicazioni;
11. Rete distribuita con iscrizione dinamica: creare una rete che cresce dinamicamente e che permetta di scambiarsi informazioni;
12. Monitoraggio del cluster in maniera intelligente: il sistema deve monitorare lo stato dei cluster in maniera efficiente;
13. Inviare alert in caso di fallimenti del cluster: segnalare un fallimento prima che tutti gli utenti di un cluster siano impattati;
14. Rimuovere un indirizzo IP dalla rete in caso di dismissione di un cluster;
15. Sospendere il monitoraggio su un cluster: come per esempio durante un aggiornamento;
16. Installazione facile e veloce: utilizzare Helm chart per installare Greye in maniera veloce e affidabile;
17. Cloud-native compliance;
18. Tolleranza ai guasti.

Aggiungere le applicazioni da monitorare

Dato che tutte le informazioni sono già presenti sul cluster, Greye non ha la necessità di persistere i dati su nessun volume o database. Questo comporta che il recupero dei service da monitorare debba avvenire con due tipologie distinte:

1. `all'avvio`: richiamando tutti i service attualmente presenti;
2. `a runtime`: rimanendo in ascolto sui cambi di eventi dei service.

Un service o ingress, viene preso in considerazione solamente se è presente l'annotation `ge-enabled: true`.

Verifica periodica dello stato delle applicazioni

Greye deve poter richiamare periodicamente tutti i path definiti dallo sviluppatore nell'annotations `ge-paths`. Dato che i service non risiedono nello stesso Namespace di Greye, per poterli richiamare, è necessario dover calcolare dinamicamente il *Service Host* di Kubernetes, che permette di identificare un Service al'interno del cluster, concatenando *nome-svc.namespace.svc.cluster.local* .

Supporto API web differenti

Data l'esistenza di diversi API web è necesario poter garantire la gestione di diverse tipologie. Allo stato attuale le API web supportate sono REST.

Integrazione con altri protocolli

Garantire l'integrazione con altri protocolli è fondamentale per avere maggior integrabilità con i sistemi. Nel caso in cui sia un protocollo standard è possibile integrarlo direttamente nel repository attraverso un'interfaccia e attendere il nuovo rilascio, altrimenti è necessario eseguire una fork del repository. Di default viene utilizzato il protocollo HTTP con REST, ma è possibile modificarne il comportamente con l'annotation `ge-protocol`.

Inviare alert in caso di fallimenti dell'applicazione

Per evitare problemi di raggiungibilità degli ambienti di produzione o lentezza è doveroso inviare una segnalazione di errore prima che gli utenti vengano impattati. È dunque necessario verificare che lo statusCode sia compreso tra 200 e 299 per le chiamate REST, se così non fosse, ha inizio la procedura di invio di notifica.

Nel caso in cui le chiamate siano lente, viene verificato che il tempo di risposta sia minore della variabile di Greye `ge-timeout-seconds`. Anche in questo caso, se non viene rispettato il timeout, ha inizio la procedura di invio di notifica.

La procedura di notifica consiste nel verificare che il problema persista per un massimo di fallimenti pari alla variabile dell'annotation `ge-maxFailedRequests`, raggiunta questa soglia, viene inviata una notifica.

Supporto a diverse piattaforme di notifica

È possibile configurare l'invio di notifiche su divese piattaforme come Telegram. Inoltre, viene esposta una metrica che segnala la necessità di inviare una notifica. Questa può essere letta da sistemi di monitoraggio come Prometheus e gestita tramite Prometheus Alerting oppure Grafana. Deve essere possibile aggiungere qualsiasi piattaforma di notifica anche direttamente nel codice.

Configurazioni specifiche per alcuni microservizi

Come già anticipato nei requisiti funzionali precedenti, ogni microservizio può sovrascrivere le configurazioni di default direttamente nelle annotations.

Mostrare grafici di monitoraggio

Poter monitorare lo stato di Greye in tempo reale è molto importante per il monitoraggio dell'applicazione stessa. Per fare questo viene messo a

disposizione una dashboard su Grafana dove vengono mostrate le metriche più importanti.

Sospensione temporanea del monitoraggio

In alcuni casi, per evitare invii inutili di allarmi in caso di aggiornamento, è necessario sospendere il monitoraggio di alcune applicazioni. Inserendo come annotation del service la variabile `ge-stopMonitoringUntil` e come valore la data e l'ora in formato UTC, è possibile posticipare la probe del microservizio fino alla data prestabilita.

Attivabile solamente per il check delle applicazioni

Greye può essere utilizzato anche solo per monitorare le applicazioni di un cluster. Per attivare questa funzionalità è necessario lasciare vuota la lista degli indirizzi IP dei cluster definita nel values file in fase di installazione.

Rete distribuita con iscrizione dinamica

Avere molteplici cluster significa anche doverli monitorare e verificare la presenza di rallentamenti nella rete. Per fare ciò, è necessario inserire nel file `values.yaml` del chart, un sottoinsieme degli indirizzi IP disponibili nella rete. Non è necessario inserire tutti, in quanto sarà Greye a creare una rete distribuita con tutti i cluster. Ciò avviene esponendo Greye attraverso un Service di tipo NodePort.

Monitoraggio del cluster in maniera intelligente

Se ogni pod di Greye dovesse contattare gli altri cluster, si potrebbero creare dei rallentamenti sulla rete, è quindi necessario cambiare approccio pur mantenendo le informazioni decentralizzate. Non c'è bisogno che tutti i pod di Greye vengano aggiornati in tempo reale, tuttavia è importante che le informazioni vengano scambiate velocemente. È quindi necessario ridurre il numero di chiamate il più possibile garantendo allo stesso modo uno scambio

di informazioni veloce. Il protocollo di gossiping in poco tempo replica tutte le informazioni su tutte le istanze di Greye.

Inviare alert in caso di fallimenti del cluster

Un pod di Greye è in errore quando il tempo di risposta è maggiore della variabile di configurazione `timeoutSeconds` oppure quando lo statusCode della risposta è diverso da 200. Il pod che ha riscontrato il problema eseguirà nuovamente la chiamata di verifica nel turno successivo. Nel frattempo, l'informazione dell'errore e il timestamp della richiesta vengono aggiunti nella lista degli IP in allarme e comunicata all'interno della rete. Se più pod rilevano il disservizio, sarà responsabilità del pod che ha effettuato per primo la richiesta gestire l'allarme. Se il numero di probe del pod del cluster supera la variabile `maxFailedRequests` verrà inviata la notifica.

Se il pod dovesse tornare disponibile, allora viene inserito nella lista contenente la history degli IP.

Rimuovere un indirizzo IP dalla rete

La dismissione di un cluster non deve limitare Greye. La cancellazione di un'istanza avviene inserendo all'interno della rete l'informazione che un determinato cluster non sarà più monitorato.

Sospendere il monitoraggio su un cluster

Ci sono dei momenti in cui è necessario spegnere un cluster oppure intervenire sulla rete. È importante dunque poter mettere in pausa la probe di un cluster, onde evitare che vengano inviate notifiche. Invocando un API di Greye contenente la chiave `stopMonitoringUntil` e come valore la data e l'ora in formato UTC, è possibile posticipare la probe di un cluster fino alla data prestabilita.

Installazione facile e veloce

L'installazione avviene tramite Helm Chart. Con Helm è possibile installare tutti gli oggetti per il corretto funzionamnento di Greye.

Cloud-native compliance

Greye deve rispettare gli standard Cloud-native[30], deve essere veloce allo startup e scalabile. In particolar modo non devono essere presenti codebase differenti in base all'ambiente di esecuzione e i log devono essere parlanti e ben precisi.

Tolleranza ai guasti

È necessario che l'applicazione sia affidabile e resiliente ai guasti potendo scalare anche orrizzontalmente. Inoltre, dato che i guasti possono coinvolgere anche i dischi, è necessario fare utilizzo solo della memoria RAM dell'applicazione. Questo permette di decentralizzare i dati e garantisce che, qualora un pod dovesse riavviarsi, sia possibile recuperare lo stato di monitoraggio precedente al riavvio.

Capitolo 5

Greye

5.1 Struttura del progetto

Greye è stato sviluppato utilizzando il linguaggio di programmazione Go e il framework Go Fiber[31] che fornisce ottimi strumenti per lo sviluppo di applicazioni web, mantenendo il codice semplice e le prestazioni ottimali. Sono stati presi in considerazione diversi linguaggi tra cui Python, Java e Javascript con NodeJs[32]. Tuttavia tutti i linguaggi precedenti soffrono di alcune limitazioni che Go non ha. È stato adottato Go come linugaggio di programmazione perchè è l'unico che soddisfa le esigenze di Greye:

- Consumo di risorse ridotto al minimo;
- Ottimale nel multitasking;
- Performance elevate;
- Avvio veloce;
- Linguaggio di alto livello.

Il progetto è strutturato sulla base del template `solrac97gr/go-jwt-auth`[33] che rispetta gli standard impartiti da `golang-standards/project-layout`[34].

Esso ha ragruppato le strutture e le best practices adottate dai grandi progetti già consolidati nel tempo, nonostante non sia mai stata riconosciuta da Go.

Il repository GitHub greye-monitoring/greye[35] si suddivide in cartelle, ciascuna con una funzionalità ben precisa:

- **/config**: contiene tutte le configurazioni per eseguire Greye in locale;
- **/deploy**: tutti i file necessari per creare immagini Docker;
- **/docs**: swagger generato da `Fiber-swagger`;
- **/internal**: contiene la business logic di Greye, al suo interno è presente un'ulteriore suddivisione `application` e `cluster`;
- **/pkg**: rappresenta il codice in comune utilizzato da entrambe le cartelle presenti in `/internal`;
- **/scripts**: tutti gli script utilizzati per fare verifiche o per generare dati.

Come si evince nella sezione precedente, la business logic di Greye è suddivisa in due parti distinte, la prima si occupa del monitoraggio delle applicazioni, mentre la seconda verifica il monitoraggio dei cluster. Questa divisione separa chiaramente gli ambiti di competenza, favorendo così la creazione di un'architettura chiara ma sopratutto semplificando notevolmente la manutenzione e lo sviluppo del codice.

L'approccio adottato nella stesura del codice richiama i seguenti principi SOLID[36, 37]:

1. **Single-responsability Principle**: ogni classe deve avere una responsabilità e un solo compito, garantendo che la classe faccia esattamente quello per cui è stata progettata;
2. **Open-Closed Principle**: Le entità sono estensibili ma non modificabili, garantendo nuove implementazioni senza dover modificare il codice esistente;

3. **Interface Segregation Principle**: le classi devono implementare le interfacce di cui hanno bisogno;

4. **Dependency Inversion Principle**: le classi devono dipendere da astrazioni e non da implementazioni concrete.

5.2 Concetti chiave dell'architettura di Greye

Greye è implementato come uno `StatefulSet` di Kubernetes, semplificando notevolmente i problemi sulla sincronizzazione dei dati.

5.2.1 Tipologia di sincronizzazione

Dato che Greye è uno StatefulSet, allora mantiene in memoria lo stato e ogni Pod è identificabile con un id incrementale. Sfruttando la possibilità di identificare i Pod e quindi di utilizzare il concetto di `controller e worker`, si permette di applicare la tipologia di `sincronizzazione asimmetrica` dove il solo pod `controller` gestisce i service da monitorare per ogni singolo Pod.

Nel caso in cui un Pod `worker` dovesse riavviarsi, sarà questo ad invocare una API del `controller` chiedendo quali service deve monitorare. Questa tipologia di sincronizzazione è chiamata `lazy`.

Dato che il sistema presenta due distinti approcci, la tipologia di `sincronizzazione è ibrida`.

5.2.2 Il protocollo di gossiping

Il protocollo di gossiping[38] consente ad un insieme di sistemi decentralizzati di scambiarsi dati in tempi molto brevi adottando un modello `peer-to-peer`. Per fare questo, ogni nodo sceglie casualmente un numero fisso di altri nodi a cui inviare le proprie informazioni, i receiver aggiornano

le configurazioni con quelle ricevute. Sfruttando il protocollo di gossiping risulta facile creare una rete che cresca dinamicamaente, infatti ogni nodo quando viene creato, ha una sottolista di indirizzi IP che dovrà contattare per entrare a fare parte della rete.

Il tempo impiegato affinchè un'informazione venga propagata a tutti i nodi è pari a[39]:

$$\log_{\text{nChiamate}} \text{nNodi}$$

dove `nChiamate` corrisponde al numero di chiamate che ciascun cluster esegue ad ogni iterazione e `nNodi` è il totale di nodi attivi. Il risultato ottenuto è un'approssimazione del numero di iterazioni delle chiamate tra i nodi necessarie affinchè l'informazione venga completamente propagata tra i nodi.

Greye adotta il protocollo di gossiping per inviare costantemente le informazioni ad una lista casuale di cluster, verificando contemporaneamente la loro disponibilità. Il pod che esegue la chiamata invia le proprie informazioni ottenendo come risposta i dati dell'altra istanza, che aggiornerà a sua volta.

5.3 Vincoli e limiti monitoraggio applicativo

I vincoli presenti su Greye sono:

- la scalabilità orizzontale del pod `controller`, che oltre a monitorare lo stato delle applicazioni, monitora anche gli altri cluster. Sebbene questo limite possa sembrare grave ad un primo impatto, questo non compromette le performance dell'applicazione. Nel capitolo relativo ai test di carico, è possibile verificare che le metriche rimangano basse anche quando sono presenti 2000 applicazioni da monitorare.

- Ogni Pod di Greye è fondamentale per il monitoraggio di un sottoinsieme di applicazioni. Se uno dei Pod dovesse riavviarsi allora quel sottoinsieme non viene monitorato fino a quando non ritorna disponi-

bile. È fondamentale inviare un allarme qualora non dovesse ritornare disponibile in breve tempo.

5.4 Vincoli e limiti monitoraggio dei cluster

Per quanto riguarda il monitoraggio dei cluster sono presenti i seguenti limiti:

- Ogni installazione di Greye deve avere specificato il proprio indirizzo IP e la relativa porta, dove il Pod di tipo `controller` rimane in ascolto. Questo permette di identificare il cluster all'interno della rete generata senza incorrere a problemi legate alla variazione degli indirizzi IP in ambienti con più nodi Kubernetes. Tuttavia, è possibile utilizzare gli Ingress per risolvere questo limite.

- Il concetto principale del protocollo di gossiping si basa sulla probabilità in quanto si scelgono casualmente un numero prefissato di host da chiamare e scambiare le informazioni. Dunque, in caso di allarme, è presente una piccola probabilità che i dati non raggiungano tutti i nodi nei tempi prestabiliti creando così l'invio di notifiche duplicate.

- È possibile creare erroneamente più reti, se nessun Pod di ciascuna rete ha un indirizzo IP appartenente a un Pod esterno alla rete.

- È possibile installare istanze di Greye con configurazioni diverse ma non è possibile mantenerle allineate anche in caso di aggiornamento. È dunque necessario eseguire un aggiornamento per ogni istanza.

- La presenza di innumerevoli cluster da mantenere monitorati potrebbe causare un overhead nella rete, tuttavia avere un numero molto alto di cluster comporta anche ad una rete solida.

Capitolo 6

Monitoraggio delle applicazioni

Di seguito viene analizzato in maniera precisa e puntuale l'architettura adottata e il motivo di determinate decisioni intraprese per quanto concerne il monitoraggio delle applicazioni. Successivamente verranno mostrate le scelte implementative di maggior rilievo.

6.1 Architettura

6.1.1 Aggiunta delle applicazioni

La prima architettura riguarda il monitoraggio delle applicazioni all'interno del cluster, dove ogni Pod di Greye verifica la disponibilità dei servizi su Namespace differenti 6.1.

Ogni Pod ha in gestione solo un sottoinsieme di oggetti distinti e univoci da monitorare e questo viene gestito dal `controller`, che mantiene in memoria tutte le applicazioni sotto monitoraggio.

È fondamentale sincronizzare i dati responsabilizzando ogni Pod con un sottoinsieme distino di servizi da monitorare, evitando di fatto l'esecuzione di probe duplicate e garantendo così la propria indipendenza all'invio di notifiche. Questo fa sì che il controller non sia scalabile per via del ruolo centrale in quanto è l'unico Pod a spartire le applicazioni da monitorare, garantendo comunque una scalabilità illimitata per i `worker`.

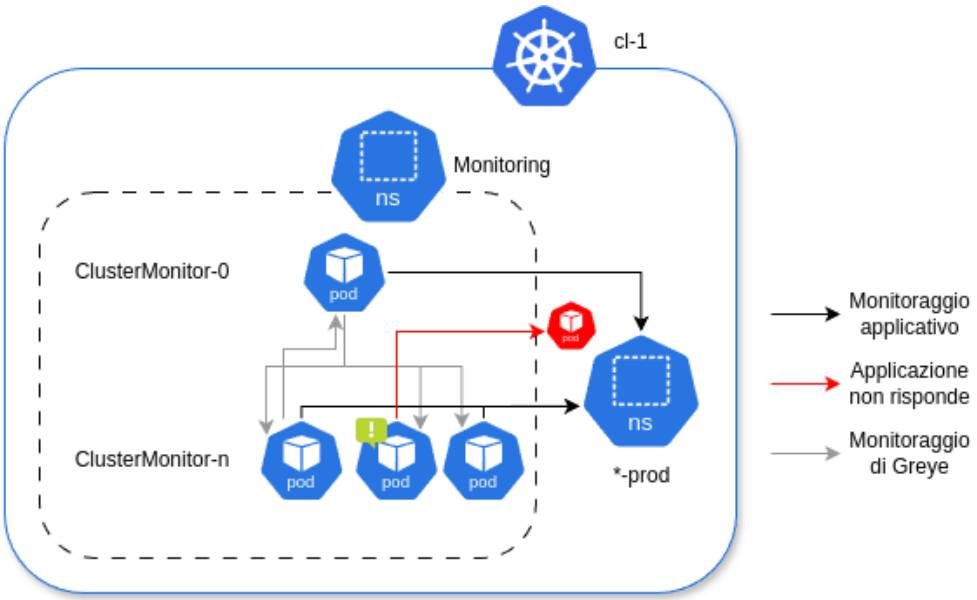

Figura 6.1: Monitoraggio all'interno del cluster

Di default viene adottato una tipologia di `sincronizzazione asimmetrica` perchè è il controller che esegue chiamate HTTP ai worker al cambiamento di stato di un service. In caso di riavvio del Pod worker, questo esegue una richiesta al controller recuperando tutti i service che stava monitorando, creando di fatto una `sincronizzazione ibrida`

L'aggiunta di un nuovo oggetto da monitorare può avvenire in due modi distinti all'avvio o a runtime.

Operazioni sulle applicazioni all'avvio

In base alla tipologia del Pod, se controller o worker, vengono eseguite due azioni distinte all'avvio:

All'avvio il `controller` esegue i seguenti passi in ordine:

1. attende che tutti i `worker` siano disponibili e per ognuno recupera le applicazioni monitorate

2. ottiene tutti i `service` di Kubernetes utilizzando la libreria `client-go`.

3. cancella tutti gli oggetti non più monitorati dai `worker`;

4. per ogni oggetto di Kubernetes, esegue le seguenti operazioni:

   - determina se il Service deve essere monitorato o da cancellare.
   - recupera il pod responsabile del monitoraggio, se presente. In base alla tipologia vengono compiuti passi diversi;

     (a) in caso di cancellazione:

     - se esiste il Pod che gestisce l'oggetto, viene invocata la cancellazione.

     (b) in caso di aggiunta o modifica:

     - se non viene recuperato alcun Pod, significa che l'applicativo non è mai stato sotto monitoraggio e viene scelto il Pod che sarà responsabile del servizio.
     - viene aggiornato l'oggetto contenuto del Pod Greye che dovrà monitorarlo

   - il processo continua con l'oggetto successivo

5. terminata l'analisi di tutti i service, viene eseguita una sola richiesta per ogni Pod, contenente la lista completa con i servizi da monitorare.

Nel caso in cui il pod riavviato abbia come tipologia `worker`, all'avvio recupera tutte le applicazioni da monitorare recuperandole dal pod `controller`, invocando l' API `/api/v1/application/monitor`. Queste applicazioni vengono filtrate utilizzando la variabile `hostaname` e aggiunte al monitoraggio. Qualora il `controller` non fosse attivo allora il Pod si avvierà senza applicazioni monitorate.

Questo comportamento garantisce a tutti i Pod di Greye, la resistenza ai guasti.

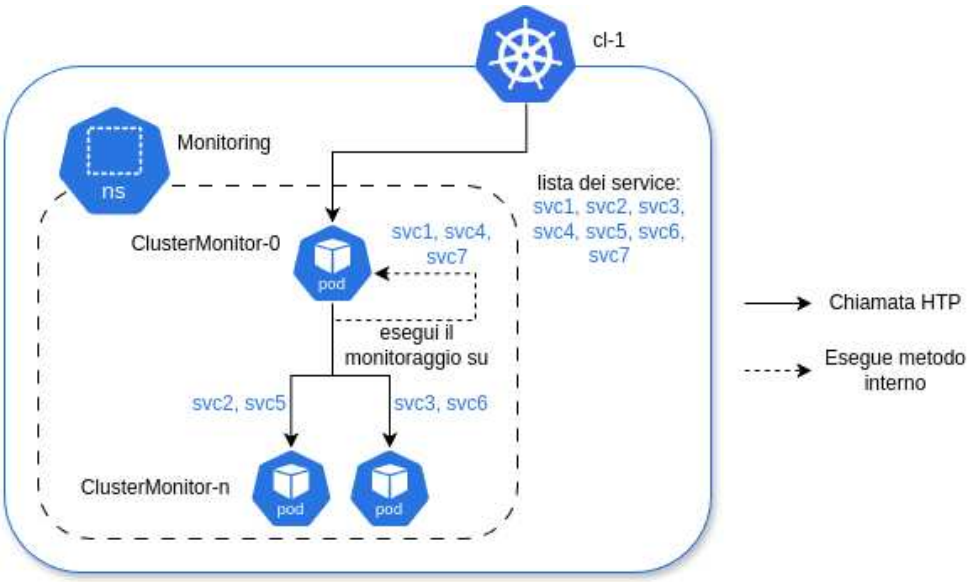

Figura 6.2: Sincronizzazione del monitoraggio applicativo

Operazioni sulle applicazioni a runtime

Le operazioni svolte per la gestione delle applicazioni da monitorare all'avvio è ben strutturato e presenta già dei punti che garantiscono un ottimo approccio anche per la gestione delle applicazioni a runtime.

Tuttavia è necessario apportare delle modifiche affinchè l'algoritmo sia corretto anche per questa casistica. In particolare è necessario rivedere il punto 4 e 5.

Il ciclo, presente nel punto 4, deve essere infinito, rimanendo così in ascolto dei cambiamenti sui service. Di conseguenza bisogna portare all'interno del loop l'invio massivo dei servizi monitorati e la gestione successiva che permette di eseguire la singola richiesta. Per risolvere questo problema viene recuperato il numero di elementi da monitorare. All'interno del loop è presente un contatore che viene incrementato ad ogni iterazione. Quando questi valori sono uguali viene eseguita la richiesta massiva e a seguire vengono eseguite le sole richieste singole.

6.1.2 Lettura delle annotations disponibili

Come già enunciato, Kubernetes interagisce solamente con il `controller`. Seppur ogni singolo Service venga analizzato da Greye, per abilitare il monitoraggio e per personalizzare la richiesta è necessario inserire le annotazioni su di esso. Nella tabella 6.1 vengono elencate tutte le annotations disponibili.

Annotations	Descrizione	default
enabled	abilita il service al monitoraggio con Greye	False
paths	lista dei metodi e dei path su cui eseguire le chiamate.	/
body	oggetto inserito come body per i metodi POST e PUT	""
headers	headers aggiunti ad ogni chiamata	""
intervalSeconds	Intervallo di tempo tra una chiamata e la successiva	60
protocol	Protocollo utilizzato per eseguire le richieste	REST
timeoutSeconds	Timeout per ogni chimata	10
max-fallback	Numero di fallimenti consecutivi prima che venga inviata una notifica	4
stopMonitoringUntil	Sospensione del monitoraggio fino alla data prestabilita	""
forcePodMonitor	Forza il monitoraggio ad un pod specifico di Greye	""
authentication	Metodo utilizzato per autenticarsi	""
authPassword	Password utilizzato in fase di autenticazione	""
authUsername	Username utilizzato in fase di autenticazione	""

Tabella 6.1: Tabella delle annotations. Ognuna deve avere il prefisso `'ge-'`

6.1.3 Modifica di un parametro

La modifica di un parametro nel monitoraggio di un Service, avviene cancellando la schedulazione presente e aggiungendone una nuova. Questo comportamente permette di schedulare applicazioni anche ad intervalli differenti qualora l'intervallo dovesse venire modificato.

6.2 Implementazione

Monitoraggio delle applicazioni e channel in Go

Una delle funzionalità uniche di Go è l'uso delle **goroutine** e dei **canali**[40]: il primo esegue funzioni in maniera concorrente senza dover creare thread, riducendo il consumo di risorse, mentre il secondo sincronizza i dati tra le goroutine. Questo approccio permette di semplificare notevolmente la programmazione concorrente.

La schedulazione del monitoraggio per un Service, avviene attraverso una funzione anonima concorrente e viene creato un canale di tipo `Ticker`, che consente l'invio di messagi a intervalli temporali costanti. È possibile impostare la frequenza recuperando il valore dall'annotazione `ge-intervalSeconds` specificata nel Service e in sua assenza, viene usata la variabile globale presente nel file di configurazione. Ad ogni periodo, viene eseguita una chiamata al Service con i parametri definiti nell'annotation.

Struttura dati

La struttura dati utilizzata per salvare le informazioni è una mappa. La chiave corrisponde al service host e il valore rappresenta tutte le informazioni relative all'applicazione, ad esempio, i dati necessari per eseguire la richiesta, la gestione dei fallimenti e della schedulazione.

Le mappe in Go non sono operazioni atomiche, di conseguenza non sono sicure per l'utilizzo concorrente, generando errori a runtime sia in lettura sia in scrittura. Per risolvere questo problema, Go mette a disposizione

i tipi `sync.Mutex` e `sync.RWMutex` che gestiscono la mutua esclusione[41]. Sync.RWMutex, rispetto a sync.Mutex, permette la gestione della lettura concorrente. Infatti, sono ottime per eseguire tante letture ma poche scritture. Questo è un vantaggio in Greye, nel caso in cui la situazione sia già stabilizzata. Tuttavia, potrebbe creare problemi di starvation nel caso di nuovi inserimenti o applicazioni in allarme. Per risolvere questo problema, si è deciso di utilizzare la struttura `sync.Map` che permette di eseguire letture e scritture senza problemi di concorrenza, in quanto è gestita internamente. Tuttavia, il valore memorizzato nella mappa è un'interfaccia quindi non ha un tipo statico, dunque è necessario eseguire la `type assertion` per ottenere il valore desiderato. La creazione o l'aggiornamento e la lettura del valore avvengono tramite l'invocazione di metodi messi a disposizione dalla `sync.Map` che sono rispettivamente `map.store(k, v)` e `map.Load(k)`.

Recupero dei Service

Kubernetes mette a disposizione degli oggetti chiamati `Validating Admission Policy` che permettono di eseguire richieste HTTP al cambiamento di un oggetto e verificare che disponga dei requisiti necessari per completare l'operazione richiesta. È possibile utilizzare questi oggetti anche per il caso di Greye, quindi per segnalare il cambio di stato di un Service. Questa pratica, per quanto possa risultare corretta, presenta delle limitazioni:

- se il Pod di Greye non dovesse risponde, viene bloccata l'esecuzione del Service rendendo di fatto Greye vincolante alla gestione del cluster stesso;
- installare questo oggetto può risultare molto complicato poichè è necesario eseguire operazioni manuali per la creazione dei certificati.

Kubernetes non mette a disposizione altri oggetti che invocano API al cambiamento di stato ma la libreria Kubernetes client-go[42] permette di eseguire comandi sul cluster. Questa viene utilizzata per recuperare a runtime tutti i Service presenti in quel dato istante e rimanere in ascolto dei

cambiamenti. Il comando sottostante permette di inviare ad un canale gli eventi.

```
app.clientSet.CoreV1().Services("").
Watch(context.TODO(), metav1.ListOptions{})
```

La riga di codice Go precedente ha lo stesso significato del comando CLI:

```
$ kubectl get services -A --watch
```

Questa operazione recupera tutti i Service, il flag `-A` indica che l'operazione debba avvenire su tutti i Namespace e `--watch` e permetta di rimanere in ascolto dei cambiamenti.

Personalizzare ogni richiesta

Le applicazioni sono tutte diverse, il che implica che ciascuna abbia endpoint distinti. Per ogni servizio è possibile: invocare diversi path ognuno con metodi differenti, inserire un body, un header e gestire l'autenticazione. Per l'utente risulta semplice poichè esso può valorizzare per ogni Service le annotazioni appropriate. Il metodo, se differente dalla `GET`, deve essere specificato all'inizio di ogni path, mentre per gli altri valori sono presenti annotation che sono `ge-header` e `ge-body`. Il body viene inserito nella richiesta solamente se la tipologia del metodo è diversa dalla GET e DELETE, mentre gli header, se presenti, vengono sempre inseriti.

Protocolli multipli

Una delle funzionalità più interessanti di Greye è la sua estendibilità. Lo sviluppatore può implementare protocolli di comunicazione senza dover modificare la business logic di monitoraggio. Questo approccio soddisfa pienamente il principio SOLID `Dependency Inversion (DIP)`. Infatti, lo sviluppatore dovrà solo inizializzare il nuovo protocollo ed implementare l'unico metodo, il quale eseguirà la chiamata con il protocollo custom e ritornerà un

oggetto contenente la lateza, il risultato delle richieste e l'esito complessivo. Questo permette anche di creare metodologie di verifica differenti, qualora lo sviluppatore abbia tale necessità.

Per utilizzare un protocollo differente da quello di default, basta utilizzare l'annotatione nel Service `ge-protocol`.

Autenticazione

Prima di eseguire ogni probe, Greye verifica la presenza di una tipologia di autenticazione. Se l'annotazione `ge-authentication` è valorizzata, allora esegue il codice relativo a quella specifica tipologia di autenticazione. Anche in questo caso, viene adottato il principio SOLID `DIP` per creare nuovi client. Lo sviluppatore dovrà solo inizializzare la tipologia di autenticazione ed implementare il relativo metodo, il quale ritornerà il valore da inserire all'interno del header `Authorization`. Nella versione `1.0.0` di Greye è stata implementata l'autenticazione `basicAuth`.

Invio delle notifiche

Il pod che riscontra un errore per un numero prefissato di tentivi, deve inviare una notifica. Anche in questo caso, come per i precedenti, è stato adottato il principio SOLID `DIP`, perciò lo sviluppatore può implementare il proprio client di notifiche personalizzato, qualora non fosse già presente. Greye gestisce anche il caso in cui l'utente non possa implementare alcun canale di notifica, esponendo delle metriche personalizzate di Prometheus, che segnala quali applicazioni monitorate sono in errore. Sarà poi lo sviluppatore a configurare gli allarmi utilizzando `Prometheus Alerting`[43] oppure `Grafana`. Nella figura 6.3 è possibile vedere alcune delle metriche esposte.

Prometheus

La base del SRE è il recupero dei dati di monitoraggio, dunque è necessario sottoporre Greye sia al monitoraggio `white-box` sia `black-box`. Con

```
# HELP application_in_alarm Indicates if an application is in alarm (1) or not (0).
# TYPE application_in_alarm gauge
application_in_alarm{name="printall-1.te.svc.cluster.local"} 0
application_in_alarm{name="printall-2.te.svc.cluster.local"} 0
application_in_alarm{name="printall-3.te.svc.cluster.local"} 0
application_in_alarm{name="printall-4.te.svc.cluster.local"} 0
application_in_alarm{name="printall-5.te.svc.cluster.local"} 0
application_in_alarm{name="printall-6.te.svc.cluster.local"} 0
application_in_alarm{name="printall-7.te.svc.cluster.local"} 0

# HELP application_under_monitoring Indicates if an application is under monitoring (1) or not (0) or not presente.
# TYPE application_under_monitoring gauge
application_under_monitoring{name="printall-1.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-2.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-3.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-4.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-5.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-6.te.svc.cluster.local"} 1
application_under_monitoring{name="printall-7.te.svc.cluster.local"} 1
```

Figura 6.3: Metriche delle applicazioni

la libreria `client_golang`[44] di `Prometheus` vengono esposte sia le metriche che fanno riferimento a Go, sia quelle personalizzate. I dati esposti vengono poi analizzati e garantiscono che il servizio funzioni correttamente.

Monitoraggio dei Pod di Greye

Greye nasce per monitorare la disponibilità dei Service presenti su un cluster Kubernetes. Nel caso in cui un Pod di Greye non dovesse funzionare più correttamente, non è presente alcun servizio che notifichi tale problema. In fase di installazione di Greye, vengono creati tanti service Kubernetes quanti sono i Pod. Per ogni `worker`, viene inserita l'annotation `ge-forcePodMonitor` il cui valore corrisponde al nome del Pod `controller`. Tale annotation permette di forzare il monitoraggio sul Pod specificato. In fase di installazione di Greye, se non vengono specificati cluster da monitorare e il numero di repliche è almeno due, viene aggiunta al Service del `controller` l'annotazione ge-forcePodMonitor con riferimetno il primo `worker`. Questo concetto è presente nella figura 6.1.

Gestione delle eccezioni nella comunicazione tra i Pod

Quando Greye esegue una richiesta di monitoraggio ad un Service all'interno del cluster e questo non risponde, il Pod esegue ulteriori tentativi prima di inviare una notifica. Fintanto che non supera la soglia di invio non accade nulla, di conseguenza non è necessario avere politiche di retry.

Al contrario se il pod `controller` deve contattare un pod di tipo `worker`, è necessario che questo risponda, onde evitare di perdere delle schedulazioni di uno o più Service. È dunque necessario implementare politiche di retry particolari. La funzione che esegue questa chiamata è intrappolata in un ciclo infinito fintanto che non ottiene un esito positivo.

Il client HTTP che mette in contatto i Pod di Greye, ha delle politiche di retry le quali, in caso di fallimento, eseguono al massimo tre chiamate ad intervalli di cinque secondi. Questo è adatto nei casi di avvio dei Pod, ma non risolverebbe il problema di perdita delle schedulazioni precedentemente descritto.

Capitolo 7

Monitoraggio dei cluster

Questo capitolo è strutturato come il precedente ma si concentra sul monitoraggio dei cluster. Descrive l'architettura adottata, le deciscioni intraprese e le scelte implementative più importanti.

7.1 Architettura

7.1.1 Monitoraggio dei cluster

Con il monitoraggio del cluster viene verificato che tutte le istanze Kubernetes siano attive e raggiungibili, garantendo inoltre il corretto funzionamento di Greye. Data la decentralizzazione dei cluster e l'assenza di una base dati condivisa, è necessario che tutti i pod `controller` di Greye comunichino tra loro per scambiarsi le informazioni rapidamente. Questo approccio seppur corretto non tiene in considerazione l'overhead delle chiamate, rischiando così di rallentare l'intero sistema.

La soluzione consiste nell'adottare il `protocollo di gossiping`, che consente di ridurre notevolmente il numero di richieste, mantenendo comunque decentralizzate le informazioni. La figura 7.1 rappresenta un esempio del protocollo appena citato con 4 cluster. Il funzionamento del `protocollo di gossiping` è semplice perchè ad ogni intervallo temporale viene chiamato un numero specifico di nodi scelti casualmente. Vengono inviati a loro

i propri dati e questi aggiornano le proprie variabili. Greye aggiunge una particolarità: ogni nodo, oltre a modificare i dati, restituisce in output la nuova struttura. Il Pod che ha eseguito la richiesta, aggiorna le proprie informazioni con le nuove. Questo approccio differisce rispetto al `protocollo di gossiping` tradizionale, ma ha permesso di verificare la correttezza del codice anche con pochi cluster, riducendo di fatto il budjet economico di creazione dei cluster Kubernetes. Per ogni installazione di Greye, viene creato un Service di tipologia `NodePort` collegato direttamente al Pod `controller`. Esso permette di esporre le API fuori dal cluster.

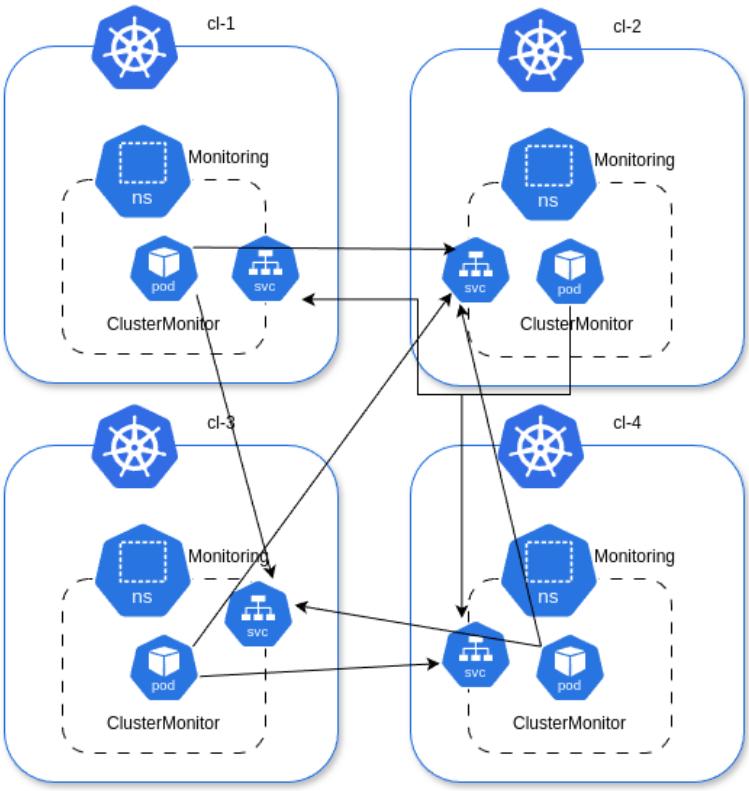

Figura 7.1: Monitoraggio dei cluster

7.2 Implementazione

Struttura dati

Anche la struttura dati utilizzata per il monitoraggio di tipo cluster è una mappa. La chiave corrisponde all'indirizzo di un'istanza Kubernetes, al suo interno sono presenti tutte le informazioni relative a quel cluster. Questo comporta una notevole semplicità nella gestione dei dati, soprattutto quando quest ultimi vengono modificati. Esattamente come nel monitoraggio delle applicazioni, anche in questa casistica potrebbero venire eseguite più chiamate contemporaneamente ad un nodo e questo deve essere in grado di poter aggiornare i dati se necessario. Anche in questo caso non è possibile utilizzare `Sync.RWMutex` poichè il numero di letture e scritture combacia nella casistica in cui i dati cambiano frequentemente. I dati vengono aggiornati principalmente all'avvio oppure quando un cluster è in errore, creando situazioni spiacevoli come race condition o deadlock.

Anche per il monitoraggio dei cluster viene utilizzata la `sync.map` che consente di gestire in modo ottimale tutte le casistiche.

Le informazioni contenute nei valori sono:

- **status**: lo stato che un cluster può assumere. Può essere: `Running`, se l'applicazione sulla rispettiva istanza Kubernetes funziona correttamente, `Deleted`, se è stato cancellato, `Suspended`, per sospendere temporaneamente il monitoraggio su un cluster ed infine `Error`, quando l'istanza non è raggiungibile.

- **timestamp**: il tempo in cui è avvenuto un cambiamento di stato. Con il confronto dei dati tra due cluster è possibile capire i passi da seguire per riconciliare i valori in base allo stato ed il tempo.

- **Error**: questo oggetto viene valorizzato solamente quando un cluster non è raggiungibile. Al suo interno sono presenti le variabili `Count` che corrisponde al numero di richieste eseguite con un fallimento e `FoundBy` che è l'indirizzo del cluster che ha riscontrato il problema.

Entrare nella rete

Sono presenti due casi distinti che permettono ad un cluster di entrare a fare parte della rete e sono:

1. Greye riceve una richiesta da parte di un'altra istanza: in questo caso, il Pod chiamato, salva tutte le informazioni passate nella richiesta e inizia a sua volta il monitoraggio fuori dal cluster.

2. all'avvio: il Pod di Greye ha una lista di IP e per ognuno esegue una richiesta. Se l'indirizzo è raggiungibile allora continuerà ad avvenire lo scambio di dati, altrimenti questo viene inserito in una lista di IP non ancora disponibili.

Scambio di dati e aggiornamenti

Lo scambio di informazioni avviene ad intervalli regolari: anche in questo caso vengono utilizzate le `goroutine` e i `ticker` che, come gia descritto nel capitolo precedente, permettono di eseguire determinate azioni periodiche. Per il monitoraggio dei cluster è presente solo un `ticker` per gestire tutto il monitoraggio. Le azioni compiute sono le seguenti:

- verifica la disponibilità degli host non raggiungibili;
- calcola il numero di cluster da contattare e recupera gli host;
- esegue le chiamate sequenzialmente.

La verifica della raggiungibilità degli host permette di inserire un cluster all'interno della rete anche dopo l'avvio. Questo può accadere nelle prime fasi di installazioni perchè vengono definiti a priori gli indirizzi delle altre istanze di Kubernetes, nonostante il servizio non sia ancora installato. In questa casistica vengono eseguiti ulteriori test anche dopo l'avvio, fino a quando l'applicazione non ottiene esito positivo.

Il numero di chiamate da eseguire è calcolato dinamicamente. Data la presenza del numero massimo di fallimenti che un Pod possa avere prima

di inviare una notifica, il quale corrisponde al risultato dell'approssimazione delle iterazioni della formula precedente, possiamo ottenere la seguente formula:

$$nChiamate = \text{nCluster}^{\frac{1}{\text{maxFallimenti}}}$$

Il risultato ottenuto, se arrotondato per eccesso, è il numero di chiamate che ogni Pod Greye deve eseguire.

Gli host vengono recuperati casualmente filtrando solamente quelli attivi ed escludendo il cluster stesso. Infine, vengono eseguite le richieste e vengono aggiornati i dati.

L'aggiornamento, una volta che il cluster si è stabilizzato, rimane costante. La prima fase consiste nel verificare che i dati recuperati in input siano uguali ai dati presenti sul cluster. Per questa verifica viene utilizzato un metodo custom che confronti l'uguaglianza dell'intera struttura dati e restituisca un valore booleano. Nel caso in cui il risultato sia negativo viene iterata l'intera mappa presente sul cluster e confrontata l'uguaglianza con la funzione `reflect.DeepEqual(map1, map2)` . Se i valori differiscono avviene l'aggiornamento, altrimenti passa al controllo successivo. Infine, gli elementi ricevuti in input ma non presenti tra i dati di Greye, vengono aggiunti.

La procedura di aggiornamento consiste nel rimpiazzare completamente l'oggetto più vecchio in base alla variabile `timestamp`. Tuttavia sono presenti delle eccezioni per avere il comportamento desiderato quando gli stati cambiano. In particolare:

- un cluster è andato in errore e riceve una richiesta da parte di un'altra istanza con lo stesso stato, allora, il Pod con timestamp minore, ha la priorità e non dovrà essere sostituito. È responsabilità di chi risolve il conflitto contattare l'host con timestamp superiore inviando la nuova base dati.

- Un cluster è andato in errore ma riceve una richiesta da parte di un'altra istanza con uno stato che è `Deleted` o `Suspended`. Lo stato viene

aggiornato a priori.

- Lo stato del chiamante è di tipo `Suspended`, mentre lo stato del cluster che riceve i dati è `Running`. In questo scenario, prima di aggiornare i dati, viene verificata l'assenza della variabile che indica la durata della sospensione del monitoraggio. Questa verifica tiene in considerazione eventuali riavvii anomali da parte del cluster sospeso.

Invio delle notifiche

Il pod che riscontra per primo l'irraggiungibilità di un cluster per un numero prefissato di tentivi deve inviare una notifica. Esattamente come nella casistica delle applicazioni, è stato utilizzato il principio SOLID `DIP`, perciò lo sviluppatore può implementare il proprio client di notifiche qualora non dovesse essere presente. Greye gestisce anche il caso in cui l'utente non possa eseguire alcuna implementazione, dando la possibilità di esporre delle metriche custom di Prometheus 7.2, che segnala tutti i cluster in errore. Sarà poi lo sviluppatore a configurare gli allarmi utilizzando `Prometheus Alerting` oppure `Grafana`.

```
# HELP cluster_in_alarm Indicates if an cluster is in alarm (1) or not (0).
# TYPE cluster_in_alarm gauge
cluster_in_alarm{name="localhost:7080"} 0
cluster_in_alarm{name="localhost:7090"} 0
cluster_in_alarm{name="localhost:8000"} 0
cluster_in_alarm{name="localhost:8010"} 0
cluster_in_alarm{name="localhost:8020"} 0
cluster_in_alarm{name="localhost:8030"} 0
cluster_in_alarm{name="localhost:8040"} 0
cluster_in_alarm{name="localhost:8050"} 0
cluster_in_alarm{name="localhost:8060"} 0
cluster_in_alarm{name="localhost:8070"} 0

# HELP cluster_under_monitoring Indicates if an cluster is under monitoring (1) or not (0) or not presente.
# TYPE cluster_under_monitoring gauge
cluster_under_monitoring{name="localhost:7080"} 1
cluster_under_monitoring{name="localhost:7090"} 1
cluster_under_monitoring{name="localhost:8000"} 1
cluster_under_monitoring{name="localhost:8010"} 1
cluster_under_monitoring{name="localhost:8020"} 1
cluster_under_monitoring{name="localhost:8030"} 1
cluster_under_monitoring{name="localhost:8040"} 1
cluster_under_monitoring{name="localhost:8050"} 1
cluster_under_monitoring{name="localhost:8060"} 1
cluster_under_monitoring{name="localhost:8070"} 1
```

Figura 7.2: Metriche del monitoraggio dei cluster

Dismissione di un cluster

Quando un'istanza di Greye deve essere rimossa, è necessario cambiare lo stato e propagare questa informazione sulla rete. Per fare ciò, in fase di cancellazione del pod tramite Helm, viene eseguito il hook `pre-delete`: prima che avvenga la cancellazione effettiva, viene installato un job temporaneo che invoca una API di Delete. Questa fa scattare un processo che setta la variabile `Status` a `Deleted` ed esegue per l'ultima volta il processo di monitoraggio, non atto a verificare la disponibilità degli altri cluster, bensì di informarli che non sarà più disponibile.

Gestione delle eccezioni nella comunicazione tra cluster

Per le chiamate tra i cluster, data la delicatezza e il numero ridotto di istanze Kubernetes, prima di incrementare il contatore degli errori riscontrati, vengono adottate politiche di retry. Queste eseguono tre ulteriori tentativi ad intervalli di cinque secondi.

Capitolo 8

Installazione

8.1 Esecuzione in locale

L'esecuzione sulla propria macchina è fondamentale per sviluppare nuove feature di Greye e poterle testare prima che vengano rilasciate. Per fare ciò è necessario avere Go e kubectl installati e configurati. Per ogni applicazione lanciata localmente, è necessario copiare il file `env.json` presente nella cartella `./config/` e rinominarlo in `env-<nomeProgressivo>`.json. Per semplificare il test, è necessario eseguire una tipologia di monitoraggio per volta. Nel caso di monitoraggio applicativo, è consigliabile avere tre istanze in running, dove il `controller` è esposto sulla porta 8080, mentre i restanti `worker` nelle porte successive, fino ad un massimo di 8089.

È importante sottolineare che tutte le applicazioni esposte sulla porta che termina con `'0'` sono considerati `controller`. Questo permette di simulare il monitoraggio su più cluster anche in locale. Dato l'elevato numero di esecuzioni contemporanee, è consigliabile valorizzare il valore di `numberGreye` pari a 1, così che per ogni istanza ci sia solo un `controller`.

```
$ cat ./config/env-nomeProgressivo.json
{
  "server":{
    "port": 8080,
```

```
    "applicationName": "localhost",
    "numberGreye": 3
  },
  "application": {
    "intervalSeconds": 30,
    "timeoutSeconds": 5,
    "protocol": "http",
    "method": "GET",
    "headers": "{}",
    "body": "{}",
    "port": 80,
    "maxFailedRequests": 3
  },
  "cluster": {
    "intervalSeconds": 10,
    "timeoutSeconds": 5,
    "maxFailedRequests": 4,
    "myIp": "localhost:8080",
    "ip": []
  },
  "notification": {
    "telegram": {
      "destination": "<destination-id>",
      "token": "<your-id-token>"
    }
  },
  "protocol": [
    "http"
  ]
}
```

Oltre ai file di configurazione è necessario settare due variabili d'ambiente

```
$ EXPORT LOCAL=nomeProgressivo
$ HOSTNAME=localhost:8080
```

che permettono rispettivamente di recupere il file JSON associato all'esecuzione e l'hostname che il programma deve avere. Ora è possibile eseguire Greye in locale.

8.2 Installazione su Kubernetes

Kubernetes, per la creazione dei Pod, ha bisogno di immagini containerizzate e per crearle è necessario utilizzare un software come Docker o Podman[45, 46]. All'interno della cartella `./deploy/` è presente il `Dockerfile` una serie di comandi sequenziali che, se eseguiti, permettono di generare un'immagine. Per generala basta eseguire il seguente comando:

```
$ docker build -f deploy/Dockerfile -t greye:latest .
```

L'immagine generata deve essere salvata su un registry e accessibile da Kubernetes. Una è già presente sul dockerHub al seguente indirizzo https://hub.docker.com/r/ftrigari/greye .

L'installazione di Greye deve essere facile quanto l'attivazione del monitoraggio stesso e deve rispecchiare la direzione in cui l'intera community di Kubernetes sta andando. Helm Chart[47] è un software di templating che permette di installare e aggiornare le applicazioni su Kubernetes ed è CNCF Graduated. Attraverso Helm è possibile configurare e installare tutti gli oggetti necessari di Greye e per fare ciò è sufficiente eseguire i seguenti comandi:

```
$ helm repo add greye https://greye-monitoring.github.io/helm-charts/
$ helm install greye greye/greye --namespace greye --create-namespace
```

Se è necessatio sovrascrivere i valori di default, bisogna creare un file vuoto e aggiungere al comando di installazione `-f <nome-file.yaml>`. Tutti gli aggiornamenti devono avvenire tramite Helm, di conseguenza è necessario modificare il file contenente i values e applicare l'aggiornamento:

```
$ helm upgrade greye -n monitoring -f <nome-file>.yaml .
```

Allo stesso modo vale per la cancellazione:

```
$ helm uninstall greye -n monitoring
```

Questi comandi permettono di gestire in maniera semplice e sicura Greye installato su Kubernetes.

È possibile visionare tuttii values del Chart di Greye sia su GitHub sia su ArtifactHub [48, 49]

Capitolo 9

Test

9.1 Testare il codice

I test sono fondamentali per il processo di sviluppo di un'applicazione, perchè garantiscono il corretto funzionamento dell'intero prodotto senza dover testare ogni singola funzionalità costantemente. Inoltre, sono importanti anche per il SRE. Nella piramide delle gerarchie del SRE è presente questa voce. In questo caso, i test prevengono il ripresentarsi dello stesso problema.

Più generale, i test permettono di prevenire e mitigare l'insorgere di problemi a seguito di uno sviluppo.

Tra gli approcci di sviluppo più famosi è presente il `test-driven development (TDD)`[50], che consiste nello scrivere prima i test e in seguito i metodi a cui sottoporrli. Lo sviluppo termina quando tutti i test hanno esito positivo. Questo comporta diversi vantaggi:

- **qualità del codice**: creando funzioni semplici e modulari in quanto ci si concentra sui test;
- **debug facile**: i test che falliscono mostrano una chiara visione del metodo che potrebbe contenere un bug;
- **sviluppi veloci**: non è necessario testare tutti i casi ogni volta che viene apportata una modifica;

- **refactory del codice sicuro**: la riscrittura di un metodo è validata dal successo dei test.

Esistono diverse tipologie di test, per ognuna è possibile suddividerle in queste tre aree gerarchiche: `unit test`, `integration test` ed `end-to-end test`.

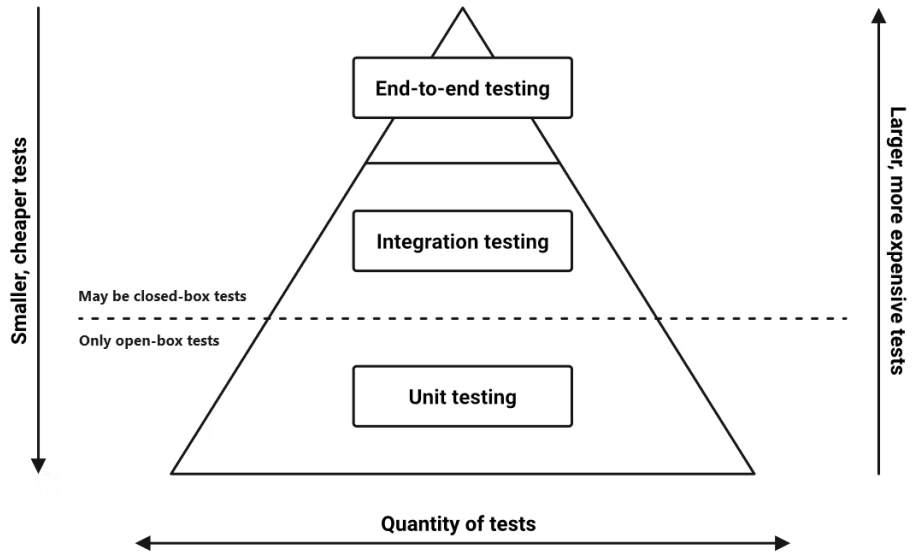

Figura 9.1: Piramide dei test https://circleci.com/blog/testing-pyramid/

Come viene mostrato nella figura 9.1, i test unitari si trovano alla base della piramide. Sono meno costosi in termine di tempo da produrre, ma possono risultare molto complicati se non vengono scritti con approcci corretti. Questi si occupano di testare l'affidabilità di un singolo metodo, andando così a creare oggetti mock nel caso in cui ci siano dipendenze esterne. La maggior parte di test scritti dallo sviluppatore deve essere di questa tipologia.

I test di integrazione sono molto più veloci da strutturare perchè permettono di testare l'integrazione tra un insieme di metodi. Ciò nonostante, i dati con cui vengono eseguite le operazioni provengono da fonti come database o servizi esterni, con la conseguenza che al cambiamento di questi i test

possano fallire. Questo fa sì che tale approccio sia più difficile da mantenere rispetto agli unit test.

I test end-to-end sono atti a testare il flusso che gli utenti svolgono. Questi sono molto complicati sia da scrivere sia da mantenere, perchè anche al più piccolo cambiamento dell'interfaccia potrebbe far fallire il test.

Greye, nella versione 1.0.0 non ha test, tuttavia è stato sviluppato seguendo il principio SOLID `Single-responsability Principle` che comporta ad una segregazione netta per ogni metodo all'interno delle classi, facilitando così lo sviluppo futuro di questi.

9.2 Helm test

Helm ha la possibilità di testare che i componenti installati funzionino correttamente. È possibile eseguire anche test di integrazione invocando direttamente le API dei pod installati.

Nel Chart di Greye sono presenti tre test e per ognuno di essi, al momento dell'esecuzione, viene creato un Pod distinto. Questi verificano la presenza dei permessi per recuperare tutti i Service dal cluster, controllano che i Pod siano attivi e confrontano il numero di applicazioni monitorate da Greye con il numero di Service Kubernetes da monitorare.

Capitolo 10

Monitoraggio di Greye

10.1 Log

Tra i dati esposti da Greye sono presenti i log applicativi generati da Logrus[51]. I log sono in formato `JSON` questo fa sì che possano essere recuperati e indicizzati da software come `ElasticSearch` o `Loki` [52, 53] che permettono di aggregarli. Questi strumenti, salvando i dati su volumi persistenti o object storage come amazon s3, rendono accessibili i log anche dopo il riavvio o il rilascio di nuove versioni, potendo dunque verificare gli avvenimenti precedenti. Poichè i log in formato `JSON`, vengono indicizzati, consentono di eseguire query sulle relative chiavi del messaggio.

Il formato dei log è composto da tre chiavi che sono il tempo in cui viene effettuata la stampa, il messaggio e il livello del log.

Greye ha cinque livelli di log che sono in ordine:

1. `trace`
2. `debug`
3. `info`
4. `warn`
5. `error`

È possibile specificare il livello di log nel file values, utilizzato per l'installazione e gli aggiornamenti, inserendo la chiave `server.logLevel` e come valore il livello desiderato. Nel caso in cui quest ultimo non venga specifciato, allora viene utilizato di default il livello che corrisponde a `info`. Aumentare il numero di log significa generare più messaggi, utili soprattutto in fase di risoluzione dei problemi. Avendo un livello basso, verranno stampati i messaggi più rilevanti. In base al livello scelto, verranno stampati esclusivamente i log di quella fascia e quelli con priorità superiore. Ad esempio, impostando come livello di log `warn` verrano stampati solamente i log di `warn` ed `error`.

Per ogni richiesta HTTP ricevuta, vengono stampati i log di `goFIber`, i quali, pur avendo lo stesso formato `JSON`, presentano chiavi diverse rispetto a quelli di Greye: viene specificato il path, il metodo, l'indirizzo IP del chiamante, la latenza nel generare la risposta e lo status code.

10.2 Metriche

Greye esegue il monitoraggio black-box sui servizi interni e sui cluster esterni, tuttavia è necessario che anche il monitoraggio white-box venga effettuato anche su di esso. Ciò avviene esponendo molteplici metriche che permettono una visione completa sullo stato del sistema.

Oltre alle metriche predefinite esposte da `Prometheus go_client`, ne sono state create altre, che permettono di avere maggiore chiarezza sullo stato effettivo delle applicazioni e dei cluster monitorati.

Esporre i dati su sistemi come `Prometheus` e `Grafana`, offre il vantaggio di inviare allarmi sfruttando i canali di notifica già preconfigurati, evitando di implementarli direttamente nel codice di Greye.

Le metriche esposte per le due tipologie di monitoraggio, nonostante il nome in parte differisca, hanno lo stesso significato. Ognuna ha una label denominata `name` contenente il service host per le applicazioni, mentre per i cluster, è contenuto l'indirizzo a cui l'altra istanza di Greye è in ascolto.

Nome	Valori	Descrizione
application_counter	$0, ..., \infty$	Conta il numero di check eseguiti per un dato sevice
application_in_alarm	$0, 1$	Restituisce 1 se il service è in allarme, 0 altrimenti
application_under_monitoring	$0, 1$	Indica se il service è monitorato (1) oppure no (0)
application_latency	$0.1, ..., \infty$	Latenza dell'ultima verifica per il service specificato

Tabella 10.1: Metriche delle applicazioni

Nelle tabelle 10.1 e 10.2 sono elencate tutte le metriche personalizzate di Greye.

Nome	Valori	Descrizione
cluster_counter	$0, ..., \infty$	Conta il numero di check eseguiti per un dato cluster
cluster_in_alarm	$0, 1$	Restituisce 1 se il cluster è in allarme, 0 altrimenti
cluster_under_monitoring	$0, 1$	Indica se il cluster è monitorato (1) oppure no (0)
cluster_latency	$0.1, ..., \infty$	Latenza dell'ultima verifica per il cluster specificato

Tabella 10.2: Metriche dei cluster

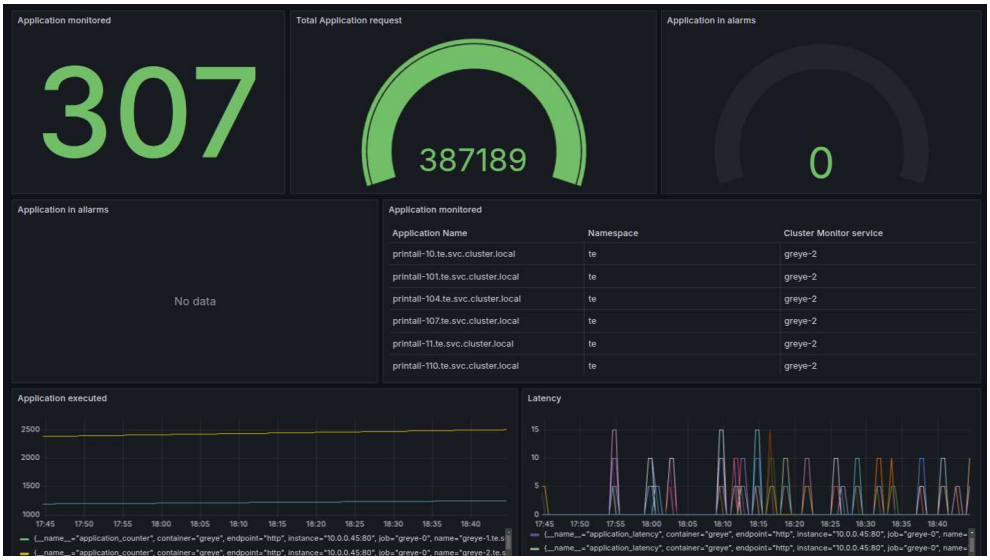

Figura 10.1: Applicazioni eseguite

10.3 Visualizzazione delle metriche attravreso grafici

Grafana è un software open source che permette di creare dei grafici eseguendo query sulle metriche di Prometheus. Il linguaggio utilizzato è chiamato **PromQL**(Prometheus Query Language), è stato creato da Prometheus per selezionare e aggregare dati in formato temporale.

Utilizzando le metriche custom di Greye è stato possibile creare una dashboard Grafana personalizzata, denominata `greye`[54]. Questa dashboard è stata suddivisa in tre parti distinte: il monitoragio delle applicazioni, del cluster e delle statistiche di Go e dei Pod.

Per quanto riguarda il monitoraggio applicativo, vengono esposte le principali informazioni atte a comprendere meglio il numero di chiamate eseguite, gli allarmi inviati, la latenza di ogni richiesta 10.1. Se non sono presenti dati, come nel grafico denominato `Application in allarms`, significa che nel range temporaneo non sono presenti informazioni, quindi in questo caso non sono presenti applicazioni in allarme.

Nel grafico `Application executed`, è presente l'andamento delle richieste eseguite per servizio. La linea più alta indica il numero di chiamate effettuate ai Pod di Greye e in questo caso sono state eseguite in totale circa 2500 richieste, ognuna ad un intervallo di 30 secondi dall'altra. La linea più in basso corrisponde a tutte le altre chiamate sotto monitoraggio eseguite ogni minuto. Si può notare come a distanza di molto tempo, le chiamate ai pod di Greye siano il doppio rispetto alle altre.

Per l'ambiente di sviluppo e di test, sono state eseguite molteplici chiamate ad un singolo deploy esposto da notevoli service. Questo ha permesso di verificare anche il caso in cui una chiamata dovesse andare in errore a causa di una saturazione delle richieste, causate dall'alto numero di invocazioni.

Figura 10.2: Protocollo di gossiping in esecuzione

Per il monitoraggio dei cluster vengono esposti i dati che fanno riferimento all'istanza corrente. Tra i dati analizzati sono presenti i cluster sotto monitoraggio, quelli in errore e quelli che non sono ancora attivi o non sono raggiungibili dall'avvio. Nell'imaggine 10.2 è presente il grafico denominato `Request executed`, il quale mostra il corretto funzionamento del `protocollo di gossiping`. Al suo interno sono esposte tutte le chiamate

effettuate ai cluster e si può notare come l'aumento sia costante ma non omogeneo, in quanto ad ogni turno vengono scelti casualmente i cluster con cui interagire.

Figura 10.3: Metriche di Go

Nella terza e ultima sezione 10.3 sono stati inseriti tutti i grafici che fanno riferimento alle metriche di Go utili a coprendere lo stato dei pod. Le metriche analizzate sono di tre tipologie:

- **goroutine**: essere a conoscenza di questo numero è fondamentale perchè permette di comprendere quanti processi sono in esecuzione contemporaneamente e di disporre di un dato chiaro su cui basare la decisione di scalare orizzontalmente;
- **memoria**: vengono mostrate le metriche riguardo alla memoria heap effettivamente utilizzate e quella già riservata. All'aumentare delle goroutines aumenta anche l'utilizzo di memoria poichè è dove risiede tutta la struttura dati. Nel grafico `Container memory usage` è presente la memoria utilizzata dai container recuperata direttamente dalle metriche di `Kubernetes`;

- **CPU**: rappresenta la CPU utilizzata dai container di Greye. Il valore della metrica `container_cpu_usage_seconds_total` è incrementale. È stata applicata la funzione `rate` che consente di calcolare la velocità di incremento su base temporale.

Capitolo 11

Prestazioni

Un'applicazione è stata sviluppata correttamente anche quando i test di carico vegono superati con successo. Su Greye non sono stati eseguiti dei test di carico utilizzando strumenti come `Grafana k6`, che permettono di eseguire migliaia di chiamate in un range temporale molto piccolo, bensì sfruttando dei casi reali creando migliaia di Service. Sono stati eseguiti i test utilizzando solo tre pod di Greye e ben 2006 service 11.1. Ciascun Service è così formato:

```
ge-enabled: "true"
ge-intervalSeconds: "60"
ge-paths: DELETE/test/delete/1234
```

Con la prima riga è stato indicato di attivare il Service al monitoraggio, con la seconda è stato impostato un intervallo pari a 60 secondi tra una chiamata e la sua successiva. Nell'ultima riga, il path da chiamare con il relativo metodo `DELETE`.

Un sottoinsieme pari a 306 servizi era già monitorato quando sono stati creati i restanti 1700 e in fase di creazione non sono stati riscontrati riavvi dei Pod o altri problemi. A seguito del caricamento e della verifica del corretto funzionamento, sono state eseguite delle cancellazioni manuali dei Pod per verificarne la stabilità anche dopo il riavvio. Anche questa volta non sono stati riscontrati errori.

I Pod associati ai service monitorati erano spenti, onde evitare di sovraccaricare notevolmente il cluster. Infatti, viene mostrato che 2004 applicazioni sono in allarme. Le uniche eccezioni sono i check ai due Pod di Greye. Il Pod greye-0 non è sotto monitoraggio dai restanti Pod di questa istanza, perchè l'installazione è di tipo multi-cluster. Quindi un'altra istanza eseguirà le probe a questo Pod.

Figura 11.1: Service caricati per eseguire i test di carico

Nella figura 11.2, sono presenti le metriche dei tre Pod di Greye quando sono stati creati i service. Si può notare un'impennata della memoria e della CPU, ma anche delle goroutines che in precedenza erano fisse ad un valore di circa 100 goroutine a Pod.

È interessante confrontare l'utilizzo delle risorse della memoria e della CPU, sia nel caso in cui vengono aggiunti Service, sia nel caso in cui il Pod sia stato riavviato e abbia già a disposizione i service 11.3. È possibile notare che nella prima immagine, dato che i service sono stati creati utilizzando il comando da terminale `kubectl create`, ciò fa sì che i Service vengano creati sequenzialmente, con qualche millisecondo di distanza tra l'uno e l'altro. Pertanto, questo ritardo nella creazione, viene mantenuto anche nella fase successiva di verifica. Al contrario, come mostrato nella seconda immagine, il Pod ha tutti i Service a disposizione all'avvio e questi sono stati

schedulati contemporaneamente. Questo comporta una differenza nelle risorse utilizzate: nel secondo caso viene impiegata maggiore memoria ma meno CPU rispetto al caso sequenziale. Ciò è dovuto da come vengono gestite le goroutine e i canali da Go.

Figura 11.2: Metriche di Greye in fase di test di carico

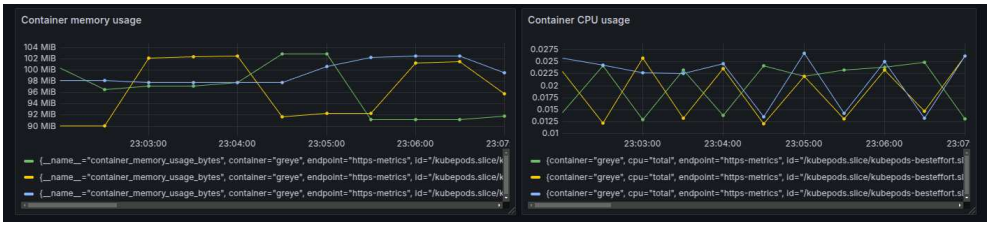

Figura 11.3: Metriche di Greye dopo il riavvio

Capitolo 12

Open source

12.1 I vantaggi dell' open source

Greye è open source protetto da licenza `Apache-2.0`. I vantaggi di un software libero, sopratutto quando è supportato da una community, sono molteplici[55]:

- **Sviluppi veloci**: sarà la community a sviluppare nuove feature che permettono di risolvere il loro problema. L'aggiunta di nuove feature comporta un aumento di visibilità del prodotto.

- **Sicurezza**: il codice open source è accessibile a tutti, di conseguenza, se vengono riscontrati dei problemi di sicurezza, è interesse dell'intera community risolverli il prima possibile.

- **Manutenzione**: viene mantenuto il codice affidabile e aggiornato dalla community nel corso degli anni.

12.2 Creazione di un progetto open source

Rendere un progetto open source non significa semplicemente rilasciare il codice in un repository Git e renderlo accessibile a chiunque. È necessario

specificare una licenza, aggiungere un file README e includere linee guida per contribuire. Ogni elemento ha un file dedicato all'interno di Greye:

- **README.md**: contiene tutte le informazioni basilari del progetto: lo scopo, come installarlo, come aggiornalo e come utilizzarlo.
- **LICENSE**: per Greye è stato adottato la licenza `Apache-2.0`.
- **linee guida per contribuire**: Viene definito precisamente come debba avvenire la contribuzione da parte di un utente. Per Greye è necessario innanzitutto aprire una `issue` specificando il problema o la nuova funzionalità. Lo sviluppatore che intende contribuire, esegue il fork del progetto sul proprio account GitHub e procede con le modifiche. Terminato lo sviluppo, viene creata una `pull request` e dopo la `code review` da parte degli admin o dei manteiner del progetto viene eseguita la merge dei commit. Lo sviluppatore ha in carico anche la creazione dei test unitari, fondamentali per la verifica della correttezza del codice.

Capitolo 13

Sviluppi futuri

Greye continuerà ad evolversi aggiungendo sempre nuove feature rispettando gli standard e i principi impartiti. In particolare verranno aggiunti:

- **Gestione degli Ingress**: nella versione attuale sono stati utilizzati solamente i Service. Per un monitoraggio completo è necessario estendere questa gestione anche agli Ingress di Kubernetes. In seguito, verrà ampliata la copertura anche ad altri oggetti come le `Routes` per `RedHat OpenShift`.

- **Semplificare l'installazione**: negli Helm values, quando si esegue l'installazione multicluster, è neceessario specificare l'indirizzo IP con la porta o l'host dove è esposto Greye. Questo comporta una difficoltà in fase di installazione, in quanto è necessario installare Greye senza specificare alcuna porta e successivamente aggiornarlo specificando la porta nei service e nel campo `cluster.myIp`.

- **Aggiungere altri protocolli, metodologie di autenticazione e client di notifiche**: Aggiungere altri protocolli, metodologie di autenticazione e altri canali di notifica permette di avere un pubblico più ampio. Tra i protocolli è indispensabile aggiungere **GraphQL** e **gRPC**, per l'autenticazione è necessario inserire **OAuth**, mentre per i canali di notifica, **Teams** e **mail**.

Conclusioni

Negli utlimi anni, con la crescente adozione dell'architettura a microservizi, il monitoragio e l'osservabilità hanno acquisito una rilevanza sempre maggiore. Questi strumenti non permettono soltanto di individuare la presenza di un errore in un determinato periodo, ma anche di prevenirlo e trovare velocemente la causa. Il ruolo del Site Reliability Engineering (SRE) svolge un ruolo centrale nel garantire l'affidabilità delle applicazioni e dell'infrastruttura, automatizzando i processi, creando dashboard di monitoraggio e ottimizzando la gestione degli allarmi.

Sono presenti due diverse tipologie di monitoraggio che sono white-box monitoring e black-box monitoring. A livello enterprise sono presenti entrambe le tipologie. Tuttavia, il monitoraggio white-box con le metriche di Prometheus e OpenTelemetry, ha avuto una crescita vertiginosa. Questi strumenti permettono di prevenire un errore e trovare la causa scatenante in tempi rapidi.

Il monitoragio black-box garantisce che un'applicazione funzioni correttamente in un determinato periodo. Sono presenti diversi software che eseguono queste verifiche, ma tutti quelli analizzati hanno dei limiti. È stato necessario creare uno strumento che, oltre al classico monitoraggio black-box: contatti direttamente i Service di Kubernetes, permetta allo sviluppatore di essere autonomo ad aggiungere l'applicazione al sistema di monitoraggio, sia in grado di verificare che anche gli altri cluster siano attivi e funzionanti, che sia Cloud native compliance e le prestazioni siano ottimali.

Greye, il software sviluppato, presenta due tipologie di monitoraggio,

quello applicativo e quello del cluster.

Lo sviluppatore può utilizzare le annotatazioni dei Service Kubernetes per aggiungere la propria applicazione al monitoraggio. Può anche modificare il comportamento di default, aggiungendo le opportune configurazioni: può gestire i path e i metodi di ogni singola chiamata, i body gli header, l'autenticazione, i timeout e gli intervalli temporali tra una verifica e la successiva.

Il codice di Greye è altamente personalizzabile. Qualora non dovesse essere presente un protocollo o un client di notifica, è possibile implementarlo senza modificare la business logic.

Il monitoraggio dei cluster utilizza il protocollo di gossiping, ciò mantiene le informazioni decentralizzate eseguendo le richieste ad un sottoinsieme di cluster casuale.

Sono stati eseguiti dei test di carico su Greye aggiungendo al monitoraggio circa 2000 servizi. Il risultato è stato ottimale senza riscontrare alcun errore.

Le funzinoalità principali di Greye sono già disponibili nel repository GitHub greye-monitoring/greye. È possibile installarlo sul cluster attraverso il Chart ufficiale greye-monitoring/helm-charts .

In futuro verranno integrati nuovi protocolli, sistemi di autenticazione e client di invio notifiche. Questi sono fondamentali per ampliare il numero di utenti.

A prescindere dallo strumento utilizzato, il monitoraggio black-box è essenziale per prevenire incidenti e per la miglioria dell'affidabilità dell'intero sistema.

Bibliografia

[1] blog.sparkfabrik.com
https://blog.sparkfabrik.com/en/
guides/sre-definition-and-advantages

[2] rootly.com
https://rootly.com/blog/
history-of-sre-why-google-invented-the-sre-role

[3] David N. Blank-Edelman, Becoming SRE, O'Reilly Media, Inc., 2024

[4] aws.amazon.com
https://aws.amazon.com/what-is/sre/

[5] devops.com
https://devops.com/
defining-availability-maintainability-and-reliability-in-sre/

[6] Niall Richard Murphy, Betsy Beyer, Chris Jones, Jennifer Petoff,
Site Reliability Engineering, O'Reilly Media, Inc., 2016

[7] www.openrefactory.com
https://www.openrefactory.com/intelligent-code-repair-icr/

[8] www.atlassian.com
https://www.atlassian.com/it/itsm/
service-request-management/slas

[9] www.ninjaone.com
https://www.ninjaone.com/it/blog/sla-slo-e-sli/

[10] spacelift.io
https://spacelift.io/blog/sre-vs-devops

[11] www.gremlin.com
https://www.gremlin.com/site-reliability-engineering

[12] opentelemetry.io
https://opentelemetry.io/

[13] prometheus.io
https://prometheus.io/

[14] grafana.com
https://grafana.com/

[15] www.dynatrace.com
https://www.dynatrace.com/news/blog/what-is-synthetic-monitoring/

[16] blog.redsift.com
https://blog.redsift.com/brand-protection/
active-vs-passive-monitoring-whats-the-difference-why-it-matters/

[17] www.brendangregg.com
https://www.brendangregg.com/usemethod.html

[18] grafana.com
https://grafana.com/blog/2018/08/02/
the-red-method-how-to-instrument-your-services/

[19] pagertree.com
https://pagertree.com/learn/devops/
what-is-observability/use-and-red-method

[20] faun.dev
https://faun.dev/c/stories/eon01/
monitoring-vs-observability-whats-the-difference/

[21] kubernetes.io
https://kubernetes.io/

[22] github.com
https://github.com/prometheus/blackbox_exporter

[23] cloudProber
https://cloudprober.org/

[24] medium.com
https://medium.com/@manugarg/story-of-cloudprober-5ac1dbc0066c

[25] zabbix.com
https://www.zabbix.com/

[26] nagios.org
https://www.nagios.org/

[27] newrelic.com
https://newrelic.com/

[28] datadoghq.com
https://www.datadoghq.com/

[29] dynatrace.com
https://www.dynatrace.com/

[30] 12factor.net
https://12factor.net/

[31] docs.gofiber.io
https://docs.gofiber.io/

[32] geeksforgeeks.org
https://www.geeksforgeeks.org/golang-vs-node-js/

[33] solrac97gr/go-jwt-auth
https://github.com/solrac97gr/go-jwt-auth

[34] golang-standards/project-layout
https://github.com/golang-standards/project-layout

[35] greye-monitoring/greye
https://github.com/greye-monitoring/greye

[36] www.digitalocean.com
https://www.digitalocean.com/community/
conceptual-articles/
s-o-l-i-d-the-first-five-principles-of-object-oriented-design

[37] Robert C. Martin, Clean Architecture, Pearson, 2017

[38] highscalability.com
https://highscalability.com/gossip-protocol-explained/

[39] flopezluis.github.io
https://flopezluis.github.io/gossip-simulator/

[40] go101.org
https://go101.org/article/control-flows-more.html

[41] medium.com
https://medium.com/@okoanton/
the-internals-of-sync-map-and-its
-performance-comparison-with-map-rwmutex-e000e148600c

[42] kubernetes/client-go
https://github.com/kubernetes/client-go

[43] prometheus.io/docs/alerting
https://prometheus.io/docs/alerting/latest/overview/

[44] prometheus/client_golang
https://github.com/prometheus/client_golang

[45] docker.com
https://www.docker.com/

[46] podman.io
https://podman.io/

[47] helm.sh
https://helm.sh/

[48] greye-monitoring/helm-charts
https://github.com/greye-monitoring/helm-charts

[49] artifacthub.io
https://artifacthub.io/packages/helm/greye/greye

[50] conformiq.com
https://www.conformiq.com/resources/
blog-the-six-benefits-of-test-driven-development-05-16-2023

[51] sirupsen/logrus
https://github.com/sirupsen/logrus

[52] www.elastic.co
https://www.elastic.co/elasticsearch

[53] grafana.com/oss/loki
https://grafana.com/oss/loki/

[54] dashboard
https://grafana.com/grafana/dashboards/22926-greye/

[55] enterprisersproject.com
https://enterprisersproject.com/article/2015/1/
top-advantages-open-source-offers-over-proprietary-solutions

Ringraziamenti

Con questo documento si conclude un percorso molto importante per me e voglio ringraziare tutte le persone che mi hanno sostenuto. In particolare il Professore Davide Rossi, relatore di questa tesi di Laurea, per la passione trasmessa della materia insegnata, per la disponibilità e per tutti gli spunti di riflessione che mi permetteranno di continuare questo percorso con grande curiosità.

Ai miei genitori, che mi hanno supportato in qualsiasi momento, rimanendo sempre un punto di riferimento forte e solido. Ogni momento di difficoltà è stato alleggerito dai miei fratelli Roberto e Francesco, un sostegno per me indispensabile e un forza incondizionata che mi ha permesso di continuare anche nelle difficoltà. Non mancano i nonni, grande esempio di dedizione, grazie a loro ho imparato il valore del sacrificio e la soddisfazione nel vedere i propri obiettivi compiuti. Ogni progetto concluso, ha visto come grande sostenitrice Rossella, grazie davvero per tutto quello che hai fatto per me e per il cammino che stiamo vivendo insieme.

Nella mia vita gli amici sono sempre stati una fonte di supporto e spensieratezza: ora avremo più tempo da passare insieme e questo mio traguardo è anche vostro. Nonostante il lavoro abbia inciso sulla durata del percorso universitario, ringrazio i miei colleghi che con la loro professionalità e amicizia hanno reso questo cammino ancora più stimolante.

Ognungo di voi è stato importante per il mio percorso e vi ringrazio davvero tanto per quello che avete fatto per me.