Alma Mater Studiorum · Università di Bologna
Campus di Cesena

dipartimento di informatica – scienza e ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

Tecniche e Strumenti per il Monitoraggio di sistemi a Microservizi – Un Caso di Studio

Elaborato in
SISTEMI EMBEDDED E INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI
Correlatore
Prof. ANGELO CROATTI

Presentata da
AUGUSTO BARBADORO

Anno Accademico 2019-2020

Indice

Introduzione

"The purpose of computing is insight, not numbers" afferma Richard Hamming. Con l'aumento e l'espansione dei business informatici, il contesto in cui operano prevalentemente i reparti IT è cambiato nel tempo, passando dal classico EDP (Electronic Data Processing) per poi arrivare ad un più complesso SIS (Strategic Information Systems). Questo passaggio ha avuto come principale conseguenza quella di capire quali siano i dati utili a verificare che il nostro sistema stia funzionando correttamente.

Generalmente i sistemi atti al monitoraggio sono accomunati dall'obiettivo di migliorare l'esperienza di utilizzo all'utente finale. Con l'aumento della diffusione delle *infrastructure as a Service (IaaS)*, riuscire ad estrapolare dati di monitoraggio utili è diventata un'operazione sempre più complessa, specialmente nel caso in cui non si è interessanti allo stato dell'hardware su cui andrà ad operare il nostro servizio in quanto spesso non è gestito direttamente dall'ente che sviluppa il servizio stesso. L'allontanamento dell'interesse dallo stato delle macchine, in quanto non più sotto un controllo diretto, ha portato alla necessità di sviluppare metodologie di monitoraggio in grado di fornire dati che siano abbastanza elastici per scalare tutto il dominio ricoperto da una specifica applicazione. Inoltre questi nuovi modi di operare devono essere in grado di inserirsi in un contesto in cui l'utilizzo dei microservizi è particolarmente diffuso.

Il lavoro descritto comprenderà un'analisi concettuale del monitoring per poi sfociare in un approfondimento pratico attraverso tecnologie specifiche. Si analizzerà perché le strutture di monitoraggio adottano un'architettura a microservizi; come questo approccio permette di migliorare il deployment consentendo di sviluppare in modo parallelo l'architettura di monitoraggio con l'applicazione stessa. Si approfondiranno tutte le caratteristiche necessarie a rendere un sistema di controllo ottimale ed efficiente. Questo, al fine di evitare le procedure che conducono alla generazioni di anti-pattern.

Lo studio effettuato verrà infine applicato a un progetto già in fase di sperimentazione presso l'ospedale Bufalini di Cesena a supporto dei medici del Trauma Team: il sistema Trauma Tracker. Il sistema ha deciso di adottare un approccio a microservizi così da facilitare lo sviluppo e l'aggiornamento delle sue funzionalità. In questo modo vengono facilitate tutte le operazione di aumento delle funzionalità tramite il deployment di servizi aggiuntivi.

La scelta di aggiungere all'ecosistema Trauma Tracker un sistema di monitoraggio

permette di poter controllare in modo più preciso malfunzionamenti ed eventualmente fornire strategie risolutive in modo tempestivo.

L'obiettivo che si pone il lavoro di questa tesi è quindi quello di attribuire a Trauma Tracker dei modi per tenere sotto controllo il suo funzionamento sfruttando i container Docker come strumento per il deployment dell'intero stack software necessario al funzionamento dell'architettura di monitoraggio. Questo servizio verrà dispiegato ricorrendo alle tecnologie di Prometheus e Grafana, utilizzati rispettivamente per la raccolta dei dati e per la loro visualizzazione. Per permettere questa operazione sarà sviluppato un apposito exporter in grado di comunicare con gli endpoint forniti da Trauma Tracker in modo da elaborare informazioni di monitoraggio.

Le tecnologie impiegate per la realizzazione dell'infrastruttura saranno analizzate in dettaglio in modo tale da capirne il funzionamento e vedere quali delle caratteristiche che definiscono un ottimo sistema di monitoraggio riescono a soddisfare.

Si cercherà quindi di implementare una struttura di controllo in grado di fornire grafici dello stato del sistema capaci di adattarsi in modo parallelo all'evoluzione del servizio e in grado di fornire dati che riescano a spaziare in più livelli del dominio di appartenenza, consentendo allo stesso tempo di rendere l'infrastruttura il meno invasiva possibile. La struttura sarà suddivisa in container così da permettere il deployment all'interno di un ambiente dockerizzato.

Capitolo 1

Moderne strategie di gestione dei sistemi informatici

Costruire del software robusto e di alta qualità non è una cosa semplice. Nel tempo lo stack delle applicazioni che permettono un efficiente sviluppo e distribuzione del software sono aumentate in modo considerevole. Questo ha generato una serie di livelli di astrazione che, se considerati in modo superficiale, possono portare alla creazione di software eccessivamente complessi.

Nel corso del tempo, quindi, si è evidenziata l'importanza di sviluppare software in modo strutturato per avere sempre il controllo di come questo reagisce a determinati cambiamenti in tempo reale. Questo è un aspetto essenziale se si vuole implementare un software considerando uno sviluppo continuativo e a lungo termine. Le caratteristiche appena descritte si sono concretizzate in varie pratiche che definiscono delle linee guida riguardanti metodologie di sviluppo e deployment. Grazie a queste regole è possibile infatti riuscire a sostenere lo sviluppo di applicazioni complesse senza risentire eccessivamente delle difficoltà di aggiornamento che si presentano naturalmente con il graduale aumento della complessità di un software. Queste pratiche hanno preso piede in quanto riescono ad inserirsi perfettamente all'interno del mercato del software aziendale moderno. Di conseguenza queste nuove metodologie hanno portato allo sviluppo di tutte quelle tecnologie in grado di ottimizzare i processi descritti al loro interno. In questo capitolo si parlerà di queste nuove correnti di pensiero, di quali sono le tecnologie che si sposano perfettamente con le pratiche sopra descritte e dei nuovi ritmi di sviluppo introdotti. Andremo ad analizzare argomenti come *DevOps* e *Docker*, a fornire una loro introduzione per capire cosa rappresentano e perché hanno formato un formidabile connubio. Utilizzeremo questi concetti come base per descrivere perché il monitoraggio è diventato una caratteristica che può essere messa alla base di ogni moderno percorso di sviluppo.

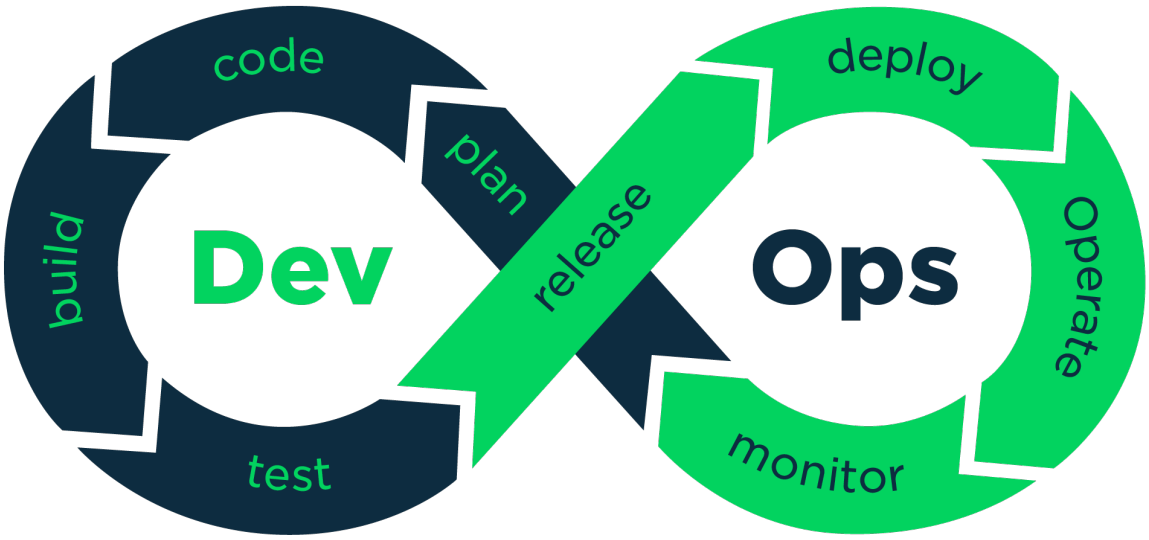

Figura 1.1: CI / CD

1.1 La rivoluzione DevOps

Quando si parla di DevOps si fa riferimento ad un tipo di approccio allo sviluppo del software che punta a migliorare la collaborazione e la coordinazione tra sviluppatori, *Developers*, e addetti alle operazioni, *Operations*, ovvero i responsabili di installazione e manutenzione del software [1]. Si tratta di un cambiamento organizzativo in cui, invece di creare gruppi separati in base alle mansioni sopra indicate, i team diventano interfunzionali. Questa cooperazione viene utilizzata per poter soddisfare i principi di Continuous Integration/Continuous Delivery (CI / CD). Il team di sviluppo dell'applicazione andrà periodicamente a pubblicare il proprio codice o ad aggiornare quello già scritto in modo da seguire il naturale percorso di sviluppo software. Il codice verrà pubblicato in un repository condiviso all'interno dell'azienda chiamato CI server (Continuous Integration). All'interno di questo saranno presenti dei moduli software che verranno successivamente integrati dai gestori del sistema in modo da permettere il funzionamento del servizio nella sua interezza. Questo approccio aiuta a fornire prodotti in modo più rapido e continuo, riducendo i problemi dovuti alle difficoltà di comunicazione tra i membri del team così da diminuire la presenza di errori e da accelerare la risoluzione dei problemi[5]. Le metodologie di sviluppo e deployment del software stanno prendendo sempre più piede e di conseguenza le aziende che utilizzano questa metodologia cercano di seguire in maniera precisa il workflow suggerito da questa pratica.

Queste linee guida permettono di riuscire a raggiungere ottimi livelli di efficienza

per quanto riguarda il rilascio e mantenimento del software e tali caratteristiche sono essenziali in riferimento al modello di business di una software house. Quindi si sono ricercate tutte quelle migliorie che permettono di ottimizzare i principi introdotti da DevOps. L'idea cardine che permette di ottimizzare il processo dietro il CI / CD è la possibilità di separare in moduli di diversa granularità così da poter parallelizzare il lavoro e riuscire a fornire continuità nello sviluppo.

La tecnologia che permette di ottemperare al meglio a questa richiesta è quella definita dall'utilizzo dei microservizi ovvero moduli software interoperabili i quali, comunicando tra loro, definiscono l'architettura necessaria al funzionamento di un servizio. Questa tecnologia non serve per l'utilizzo di DevOps ma l'idea alla base si sposa in maniera ottima con la linee guida definite da questa corrente di pensiero. In questo scenario DevOps fornisce il framework per lo sviluppo, la distribuzione e la gestione dell'ecosistema mentre l'utilizzo di un'architettura basata sui microservizi permette l'ottimizzazione di tutte queste operazioni rendendo l'utilizzo di metodologie DevOps su questa tipologia di architettura un connubio vincente.

1.1.1 L'importanza del monitoraggio in DevOps

Uno dei cardini principali dell'ideologia DevOps è proprio quello del monitoraggio pervasivo [1]. Questa pratica prevede che sia il software, sia l'infrastruttura su cui questo funziona, vengano costantemente monitorati. Questo aspetto è diventato del tutto necessario. La scomposizione in sottomoduli di un software permette certo di avere delle facilitazioni per quanto riguarda lo sviluppo, ma allo stesso tempo rende molto complesso riuscire a rintracciare problemi all'interno del servizio in quanto capire come le singole parti interagiscono una volta che sono collegate agli altri moduli, rimane un'operazione particolarmente complessa. Per questo motivo il monitoraggio assume grande importanza in DevOps in quanto permette di osservare l'applicazione nelle sue sottoparti, così da consentire di controllare come l'applicativo reagisce all'aggiunta o all'aggiornamento di nuovi moduli software atti ad ampliarne le funzionalità. Questi strumenti di monitoraggio consentono alle organizzazioni di identificare e risolvere i problemi dell'infrastruttura IT prima che questi influenzino i processi aziendali critici. Inoltre monitorano continuamente lo stato di salute del sistema e avvisano gli amministratori quando si rilevano problemi in modo che questi possano intraprendere azioni correttive [5]. Per tali ragioni i sistemi di monitoraggio sono funzionali al controllo di un applicativo basato sui microservizi. Nei prossimi capitoli andremo ad approfondire questi aspetti.

1.2 L'arrivo dei microservizi

Tradizionalmente le aziende producono soluzioni informatiche basandosi su un'unica applicazione che racchiude al suo interno tutte le funzionalità necessarie, il cosiddetto

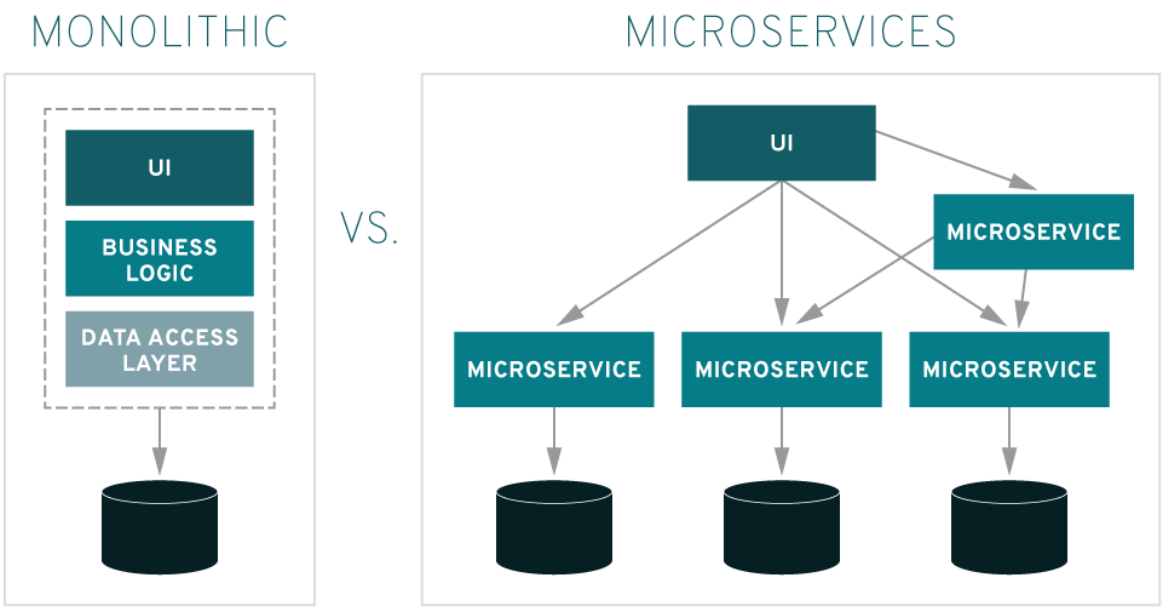

Figura 1.2: Software monolitici vs microservizi

applicativo monolitico. Ciascuna delle sue sottoparti corrisponde ad un modulo che implementa tutte le funzionalità per il dominio a cui appartiene. Inoltre, l'intera applicazione accede ai dati passando per un solo database ed espone delle API a cui i client possono effettuare richieste. Questo è stato il tipico approccio alla programmazione per diversi anni in quanto, almeno inizialmente, costituiva un'architettura facile da sviluppare, testare e dispiegare. Poi ci si è resi conto che, a mano a mano che le dimensioni del software crescevano, quei processi che sembravano semplici diventano esponenzialmente più onerosi. Questo perché avere più team che lavorano allo stesso codice, integrare modifiche portate avanti per mesi, verificare che tale integrazione fosse priva di errori e infine dispiegare sui server tali cambiamenti, risultava sempre più complicato. Veniva quindi a mancare il concetto di agilità e di indipendenza dei team. Scegliere un approccio orientato ai microservizi risponde alla volontà di svincolarsi dalla staticità generata nel tempo dalle applicazioni monolitiche (Fig. 1.2).

In pratica si tratta di suddividere i moduli che costituivano il nucleo delle funzionalità, processi a sé stanti, che prendono il nome di microservizi. Ogni servizio avrà la propria base di dati a cui fare riferimento, separata da quelle degli altri servizi così da rendere il front end del sistema il più indipendente possibile dalla struttura interna. Oltretutto ogni singolo servizio è autonomo rispetto agli altri così lo sviluppo dei singoli moduli può procedere in parallelo rimanendo indipendente dalla catena di produzione ed evitando anche di causare ritardi all'implementazione di altri moduli dipendenti. La struttura

a microservizi permette di disporre un processo di deployment snello e veloce così da poter aggiungere o modificare funzionalità di un sistema software in modo efficace ed efficiente. Queste caratteristiche rappresentano elementi chiave che si sposano in maniera ottimale con le regole imposte da DevOps. In un'architettura a microservizi, quando una componente non funziona non è automatico che tutto il sistema software smetta di funzionare. In molti casi è possibile isolare il problema ed intervenire mentre il resto del sistema continua a funzionare, cosa non possibile in un'architettura monolitica. Questa caratteristica viene definita *Resilienza*. Sotto questo aspetto lo sviluppo del monitoraggio è di fondamentale importanza in quanto permette di individuare velocemente il problema in modo da poterlo risolvere prima che questo causi dei danni maggiori all'intera struttura di funzionamento.

1.2.1 Monitoraggio di microservizi

In questi ambienti modulari risulta anche più pratico andare ad utilizzare nuove tecnologie. In un ambiente monolitico applicare grossi cambiamenti architetturali risulta un'operazione complessa. In un ambiente basato su servizi il tutto può essere suddiviso in step che si concentrano sulla riscrittura dei singoli moduli. Questa possibilità ha portato allo sviluppo di tantissime tecnologie eterogenee che allo stesso tempo riescono ad interoperare in modo ottimale. Nello specifico andremo successivamente ad approfondire il caso di *Prometheus*.

La tecnologia di monitoraggio, presa in esame all'interno del presente lavoro, risulta un esempio significativo di come sfruttare al meglio l'architettura dei microservizi per introdurre un sistema di monitoraggio che sia modulare, indipendente e poco invasivo. La gestione dei microservizi ha inoltre acquisito l'arrivo di nuove tecnologie che permettono un'efficiente orchestrazione dei microservizi stessi. Nel capitolo successivo andremo ad analizzare il prodotto che si è diffuso maggiormente per quanto riguarda la gestione di questo tipo di architetture.

1.3 Gestione dei microservizi con Docker

Una delle tecnologie più diffuse per la gestione delle architetture basate sui microservizi è quella offerta da Docker, prodotto dell'omonima azienda *Docker Inc.* Questa, anche grazie al contributo della sua vasta community, è riuscita ad inserirsi sia nelle piccole che nelle grandi imprese rendendo immediato l'uso delle sue principali funzioni e allo stesso tempo veloce l'apprendimento di operazioni più complesse (Fig. 1.3). Docker permette di gestire container ovvero istanze di un'immagine che sono l'elemento alla base della tipologia di virtualizzazione che gestisce. Su questi container è possibile eseguire dei comandi e gestire delle reti virtuali per permetterne la loro interazione e poter creare quello che viene definito un ambiente containerizzato.

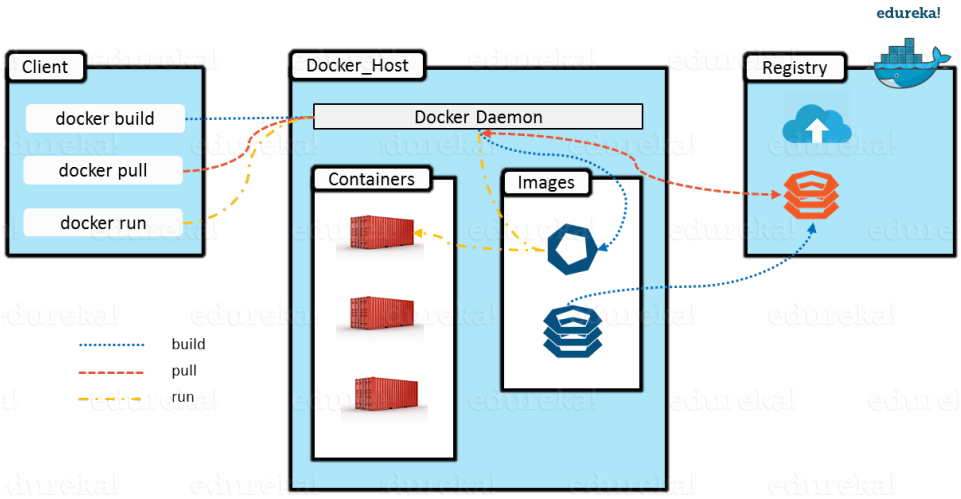

Figura 1.3: Funzionamento di Docker

Architettura

Docker riesce a separare parti del programma in moduli indipendenti così da poter eseguire modifiche real time sui suoi container. Questa sarà la caratteristica fondamentale che useremo per compiere il progetto descritto in questa tesi. I componenti che costituiscono Docker e che consentono di eseguire le azioni sopra elencate sono tre [6].

- **Docker engine**: fornisce le funzionalità principali per la gestione dei container. Si interfaccia con il sistema operativo sottostante ed espone semplici API che permettono di gestire il life cycle dei container.
- **Docker tool**: set di istruzione che vengono impartite da riga di comando che comunicano con le API esposte dal Docker Engine. È usato principalmente per eseguire container, creare nuove immagini, configurare storage e reti.
- **Docker registry**: il luogo in cui vengono archiviate le immagini dei container. Ogni immagine è identificata da un ID univoco; gli utenti utilizzano le immagini presenti nel Docker Registry. Se queste non dovessero essere presenti, Docker consulta il suo repository online nel quale la community rilascia le proprie immagini che sono liberamente utilizzabili dai vari utenti.

Docker object

I suddetti componenti lavorano su quelli che vengono definiti Docker object. Su questi oggetti vengono eseguiti i comandi che Docker espone nelle proprie API. Dato che l'oggetto di questa tesi riguarda l'utilizzo di un'architettura containerizzata, di seguito sarà fornita,

con riferimento alla documentazione ufficiale di Docker[6], una breve descrizione degli oggetti usati più frequentemente.

- **Docker Image**: è un modello di sola lettura che contiene tutte le istruzioni per la creazione di un Docker container. Spesso un'immagine è basata su un'altra immagine alla quale, però, possono essere applicate delle modifiche o personalizzazioni. Ad esempio, è possibile creare un'immagine utilizzando come base Ubuntu, ma installare al suo interno web server Apache, inserendo anche un'applicazione già configurata e pronta all'uso. È possibile creare immagini o usare quelle create da altri utenti e pubblicate nel Docker Registry. Per costruire un'immagine si utilizza un Dockerfile che contiene tutti i passaggi necessari per creare l'immagine ed eseguirla.

- **Docker Container**: è un'istanza eseguibile di un'immagine. Il Docker Engine permette di creare, avviare, arrestare, spostare o eliminare un container utilizzando una delle sue API esposte. Per impostazione predefinita, un container è logicamente isolato dagli altri e dalla sua macchina host ed è definito dalla sua immagine e dalle opzioni di configurazione che gli vengono fornite quando lo si avvia. Quando questo viene eliminato, tutte le modifiche applicate al suo stato originale, che non sono state archiviate in un volume persistente, scompaiono.

- **Docker Volumes**: rappresenta la possibilità di mantenere dei dati prodotti da un container. Docker espone la possibilità di agganciare delle directory del file system dell'host a delle directory del container; Così facendo viene garantita la persistenza dei dati tra vari cicli di vita di un container. In tal modo Docker rende permanenti le modifiche al container e permette di memorizzare su disco i dati generati così da poter essere eventualmente riutilizzati. Per memorizzare le modifiche alle opzioni di configurazione apportate all'immagine, è possibile fare un commit di questa in modo che possa essere richiamata in seguito con le modifiche applicate.

- **Docker Networks**: i Docker network sono formati da dei driver che permettono l'interconnetività dei container. Ogni tipo di driver descrive come i pacchetti vengono instradati al suo interno. Esistono tipi di driver per gestire reti interne all'host definite *bridge network*; nel caso si vogliano far comunicare container su host separati, bisogna utilizzare i driver di rete definiti *Overlay Network*. Connettendo un container ad una rete verrà creata un'interfaccia specifica di quel network per quel container. Ogni container può collegarsi a più reti: così facendo su di questo verranno generate più interfacce. Tutte le caratteristiche di un container sono osservabili con il comando *inspect*.

Capitolo 2

Monitoraggio di Sistemi Informatici

Il monitoring viene definito come un modo per indicare la vigilanza continua di un *soggetto* mediante strumenti che ne misurano le grandezze caratteristiche. Con lo sviluppo e conseguente ingrandimento dei sistemi, il *soggetto* è arrivato ad assumere significati sempre più ampi. Oggi il *soggetto* potrebbe essere identificato nell'intero processo che esso subisce o addirittura nell'intera struttura operativa che ne permette la sua elaborazione. Questo sviluppo del concetto di monitoraggio fu inizialmente applicato soltanto ad un contesto prettamente industriale.

Le metodologie di produzione del software si sono ampliate a tal punto che questi concetti ormai possono adattarsi pienamente anche al mondo dell'informatica. In questo dominio il significato di monitoraggio viene rivisitato per poterne dare una descrizione più appropriata.

L'atto di monitorare un sistema può essere identificato nel processo di registrazione e osservazione dei cambiamenti di stato e dei flussi di dati che produce. Questi possono essere acquisiti registrando richieste e risposte che avvengono tra componenti interne e/o esterne al sistema[1]. Le componenti software che attuano tale processo sono chiamate Infrastruttura di monitoraggio.

Il monitoraggio ha una lunga storia nello sviluppo tecnologico. I primi sistemi di monitoraggio erano dispositivi hardware come gli oscilloscopi o più in generale strumenti di misurazione che potevano essere utilizzati per controllare il corretto funzionamento di componenti elettronici. Tali strumenti esistono ancora in questo ecosistema; tuttavia, ignoreremo questa tipologia di dispositivi e ci concentreremo esclusivamente sul monitoraggio del software.

L'enorme sviluppo di sistemi cloud e l'aumento della complessità dei sistemi informatici ha costretto ad abbandonare le vecchie metodologie per monitorare lo stato di salute di un sistema, stimolando lo sviluppo di nuove infrastrutture in grado di gestire in modo ottimale il monitoring di questa nuova categoria di applicazioni.

Oggi, infatti, è richiesto di monitorare non solo il lato hardware di un sistema poiché, con la diffusione della virtualizzazione, il corretto funzionamento di un servizio si appoggia

su un complesso stack di applicazioni intermedie che in qualsiasi momento potrebbero smettere di funzionare. Le applicazioni che utilizzano questi ambienti di deployment sono sempre più separate dall'hardware stesso e quindi necessitano anche di nuove forme di comunicazione del loro stato e comportamento che non sono più strettamente collegate alla macchina su cui esse funzionano. Questo ha portato a cambiare l'infrastruttura stessa dei software di monitoraggio e anche a rivalutare l'importanza che il monitorare assume all'interno del processo di sviluppo di un'applicazione.

Parlare di monitoring di un carico di lavoro, significa controllare il flusso dati dell'intera struttura in quanto tutte le attività di un ecosistema software contribuiscono al carico di lavoro generato da un servizio e questo include sia il flusso dati generato dall'applicazione sia quello generato dal sistema di controllo.

Ogni Sistema di monitoraggio produce delle informazioni. Queste sono generate dall'impiego di codice nelle entità che si vogliono sottoporre ad un controllo, inserendo in questi oggetti sonde software, hardware o sensori che rilevano eventi e che generano rapporti di stato. Queste informazioni vengono successivamente utilizzate per generare dati di monitoraggio[9].

Per comprendere che cos'è il monitoraggio e come questo viene effettuato, è necessario andare ad analizzare la struttura logica che ne è alla base. In ogni sistema di monitoring si possono trovare cinque parti principali: il tipo di dato, le metodologie per la raccolta dei dati, la loro archiviazione, la presentazione e infine la generazione di alert. [9] [4] [3]. L'insieme di questi componenti di monitoraggio consentono la costruzione di un insieme di informazioni che rappresentano lo storico dei vari stati del sistema nei vari istanti di tempo.

2.1 Modello dei dati

Alla base di ogni sistema di monitoring sono presenti i dati che permettono il trasporto delle informazioni. Le informazioni che un sistema è in grado di produrre dipendono fortemente dalla tipologia di dati che si sceglie di adoperare nel proprio sistema e nel tempo queste hanno assunto varie forme. Osserveremo come la tipologia di informazione che un dato riesce a comunicare cambi in base alla propria classe di appartenenza. Nel presente lavoro ci concentreremo solamente sulle tipologie di dato che riscontrano un utilizzo particolarmente efficace e che riesce ad integrarsi al meglio con le tecnologie di cui si parlerà più avanti in una sezione dedicata.

Health checks

Una delle prime modalità di monitoraggio, definita come sistema *legacy*, è quella che viene identificata con il nome di *check based monitoring*. Questa tecnica agentless utilizza script e protocolli specializzati per verificare attivamente lo stato del sistema. Il protocollo

che rappresenta al meglio questo tipo di dato è SNMP (Simple Network Management Protocol). Questo protocollo è presente in 3 versioni che differiscono principalmente per quanto riguarda la configurazione della sicurezza.

SNMP comunica codici chiamati *OID* con i dispositivi che lo permettono. Ogni codice comunica un valore della macchina che si sta monitorando. I dati vengono successivamente collezionati e mostrati in dashboard di controllo. Gli *health check* producono le informazioni tramite l'esecuzione di script che controllano la stato del sistema. Nel caso di *SNMP* si raccolgono gli *OID* e poi si interpretano i valori ad essi collegati[4]. Questo tipo di dato, nonostante venga ancora ampiamente utilizzato, presenta qualche problema. L'assenza di un contesto temporale globale impedisce di confrontare in modo ottimale i dati ricevuti dalle varie macchine rendendo molto complesso riuscire a capire se determinati problemi sono tra loro collegati. Oltretutto gestire i dati in modo così separato rende praticamente impossibile controllare lo stato di sistemi su larga scala. Essendo un controllo molto legato all'esecuzione di script su macchine specifiche, questo va a generare una conformazione particolarmente statica in cui il sistema non sa cosa monitorare finché questo non viene chiaramente esplicitato dall'utente che vuole verificare il corretto funzionamento degli organi di rete. Le informazioni che questo protocollo riesce a collezionare sono poco scalabili a livello di contenuto poiché riescono a esprimere prettamente caratteristiche legate all'hardware senza poter spaziare tra vari livelli del dominio. Oltretutto programmare un controllo sulla totalità del sistema diventa particolarmente complesso, specialmente se si stanno utilizzando sistemi distribuiti.

Logs

Un'altra tecnica che che viene ancora molto utilizzata è la gestione dei log/eventi testuali generati dall'applicazione. Sono ancora largamente diffusi poiché forniscono molti dettagli su quello che la macchina sta facendo e sono particolarmente facili da implementare poiché basta ricorrere a delle stampe. Ovviamente il dettaglio che i log possono fornire impatta pesantemente dal punto di vista delle performance poiché questo sistema scala in modo direttamente proporzionale al numero di operazioni richieste. In momenti di grande attività, la scrittura dei log potrebbe appesantire eccessivamente il server.

Ogni Applicazione stampa delle stringhe che descrivono il comportamento in modo molto dettagliato di tutto quello che sta succedendo all'interno del programma. Nonostante queste stringhe siano molto utile, l'assenza di uno standard nella scrittura dei log, li rende facilmente leggibili dagli utenti ma difficilmente adoperabili in un contesto di controllo automatizzato.

Metrics

Tra tutti i tipi di dato diffusi, quello più utilizzato è il *metrics based systems*. Il concetto delle metriche è stato introdotto per ottemperare alle richieste dei sistemi più moderni. Le metriche sono dei valori numerici rappresentativi dello stato del sistema che stiamo monitorando. Spesso queste vengono accompagnate da dei time-stamp per poterle inserire in una linea temporale; così facendo l'insieme di metriche riesce ad esprimere il cambiamento di stato di un sistema nello scorrere del tempo. Rimangono comunque molto semplici da produrre e hanno un utilizzo funzionale soprattutto nel caso di sistemi di grosse dimensioni in quanto sono facilmente raggruppabili.

Tuttavia scegliere di utilizzare le metriche piuttosto che i log genera la perdita di qualche informazione in quanto la produzioni di queste non è strettamente legata alle operazioni compiute dal servizio ma allo stato del sistema in precisi istanti di tempo; ne deriva che tutti gli eventi che avvengono tra due istanti temporali sono persi. La memorizzazione di due stati della macchina in due istanti di tempo differenti è comunque un'informazione sufficiente nel caso si voglia anticipare un potenziale problema, a patto che l'intervallo di tempo in cui si va ad osservare lo stato del servizio sia sufficientemente breve da permettere al sistema di reagire.

Le metriche, inoltre, sono dei valori in grado di descrivere dei concetti che possono adattarsi a tutti i livelli del dominio in cui l'applicazione opera. Non debbono essere generate solamente all'interno dell'applicazione ma possono essere prodotte da elementi esterni pur rimanendo consistenti all'applicazione stessa. L'utilizzo di questo tipo di dato viene incontro alle necessità dei sistemi a microservizi in quanto tramite il deployment di parti esterne di codice è possibile rendere monitorabile un'applicazione senza dover necessariamente modificare l'applicazione stessa. Questo permette l'installazione di un architettura di monitoraggio a posteriori in grado di non alterare il codice dell'applicazione principale. Tale aspetto verrà poi approfondito quando analizzeremo le entità chiamate *exporter*.

2.2 Raccolta dei dati

Questa caratteristica delle infrastrutture di monitoraggio definisce come i dati vanno collezionati, portandoli dagli host che devono essere monitorati al server centrale per l'elaborazione dei dati. La discussione su come deve essere effettuata questa operazione resta una questione aperta relativamente al monitoring di un'infrastruttura software. L'argomento polarizza le opinioni in due partiti principali: quelli che preferiscono un approccio *push* e quelli che preferiscono un approccio *pull*. L'intero argomento può essere semplificato ad una singola domanda; quale entità inizia il processo di trasferimento della metrica, l'ente gestore dei dati o l'host monitorato[4] ?

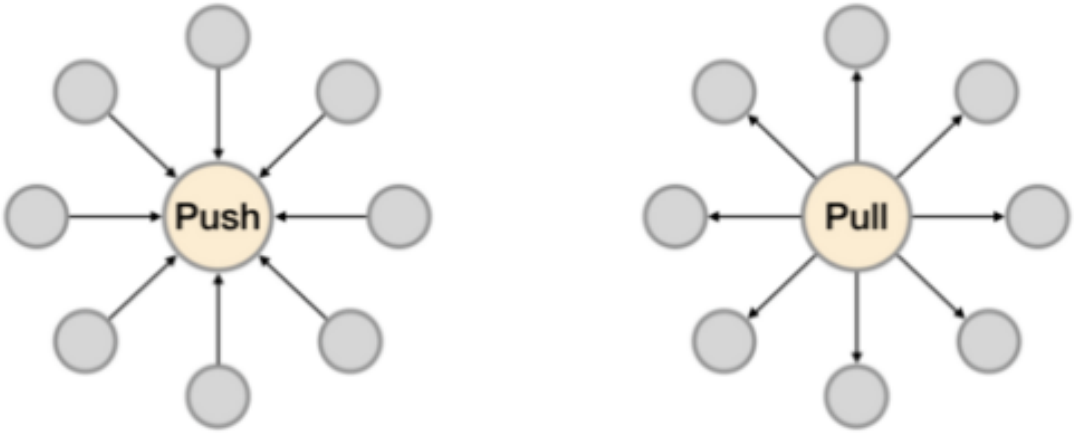

Figura 2.1: Approcci pull e push

L'approccio pull è quando il servizio di monitoring invia un file di richiesta a un host per ricevere le metriche come risposta. Ognuna di queste richieste avviene ogni istante di tempo t; maggiore sarà la grandezza dell'intervallo di tempo Δt, maggiore sarà la granularità dell'informazione risultante.

L'invio di richieste su un intervallo specifico è noto al netto dei ritardi introdotti dalla latenza della rete. Ciò è utile nel caso in cui si verificano eventi in specifici istanti e questi vogliano essere registrati.

Il *Time driven* monitoring si basa sull'acquisizione di informazioni periodiche sullo stato del servizio in modo da fornire una visualizzazione istantanea del comportamento di un oggetto o di un gruppo di oggetti. Esiste una relazione diretta tra la frequenza di campionamento e la quantità di informazioni generate [9]. Questa tecnica è spesso associata ad un approccio di monitoraggio agentless, utilizzato per evitare di apportare modifiche al software dell'host da monitorare (Fig. 2.1).

Si contrappone a questo metodo l'approccio push. In questo caso è il software che deve essere monitorato ad inviare le informazioni sul suo stato. Questa operazione può essere fatta in modo sincrono inviando le informazione ad intervalli di tempo regolari oppure associando l'invio delle metriche a specifici eventi in modo da rendere questa operazione asincrona. Le metodologie di push degli eventi sono molto precise e preferibili nei casi in cui sia richiesta la ricezione di eventi particolarmente importanti o task troppo brevi che non riescono a superare l'intervallo Δt di invio/ricezione delle metriche. L'approccio push è spesso associato all'utilizzo di agenti. Un approccio basato su agenti è quando viene eseguito un modulo software esterno su ogni host monitorato. Ciò influisce sulla scalabilità in quanto il deployment di un servizio deve essere accompagnato dalla scrittura di un agente che si occupi della gestione del monitoraggio. Questo potrebbe riflettersi

anche sulla sicurezza, la manutenzione, la distribuzione del sistema e dei tempi impiegati per l'aggiornamento dei microservizi.

Un approccio senza agente (agentless) è quando un'API o un protocollo è utilizzato da un dispositivo remoto, in questo caso il server di monitoraggio, per analizzare le metriche del software che si vuole monitorare. Questo favorisce l'aggiunta di servizi nella rete di monitoraggio a patto che le applicazioni stesse espongano delle interfacce/endpoint atte a disporre le informazioni di controllo di stato.

2.3 Archiviazione

Scegliere il corretto metodo di archiviazione dei dati è un'operazione molto importante. Questa operazione è in grado di influenzare le prestazioni dell'architettura di monitoring sia in termini di tempo di compimento delle operazione di storage sia per quanto riguarda l'aspetto della scalabilità.

La memorizzazione dei dati di monitoraggio richiede delle specifiche proprietà in un database. Non sono richieste operazioni di scrittura in quanto inserire dati manualmente andrebbe ad alterare lo storico degli eventi di un sistema rendendo impossibile applicare delle analisi valide. L'inserimento è necessario solo tramite operazioni di *append* in quanto lo stato del sistema è in continuo divenire e risulta inutile inserire uno stato passato; oltretutto si presuppone che quell'istante sia già inserito all'interno del database.

Non è richiesta una eccessiva capacità di archiviazione in quanto i dati più significativi sono quelli più recenti. Più un evento risulta distante dal presente, più questo perde significato in termini di causa-effetto e la sua efficacia nel segnalare eventuali errori o l'imminente cessazione del servizio. Tuttavia lo storico più vecchio di un sistema può comunque essere sottoposto a delle operazione di *capacity planning*. Questo tipo di analisi, però, richiedere una granularità dei dati più ampia, che non richiede la precisione del database di storage principale.

La lettura dei dati all'interno del database interessa dati che sono raggruppati in modo sequenziale e non sparso. Vedere lo stato del nostro sistema negli ultimi cinque minuti implica la lettura di tutti gli stati in modo ordinato, partendo da quello più vecchio, fino ad arrivare al più recente. Il miglior sistema di storage per quanto riguarda il monitoring di un sistema è stato trovato nei *Time series database*. Questa tipologia di database permette l'ottimizzazione delle operazioni descritte sopra. Ogni specifica architettura di monitoraggio avrà uno specifico database che fornirà un metodo di interfacciamento dei dati e un linguaggio ad-hoc per la loro lettura.

2.4 Presentazione

La presentazione è il metodo di visualizzazione dei dati raccolti in un'interfaccia. Ciò consente al gestore di rete di essere in grado di mostrare lo storico dei dati dei dispositivi monitorati. Questa rappresentazione è necessaria per identificare tendenze o anomalie in un ampio insieme di dispositivi. Ad esempio, questo può essere utilizzato per identificare visivamente a che ora un server web è sovraccarico di lavoro o ha più tempo di traffico/utilizzo. La visualizzazione dei dati è una tecnica analitica e interpretativa molto efficace; uno straordinario strumento di apprendimento solamente se questa viene eseguita nel modo corretto. Gli esseri umani tendono a quella che viene definita "apofenia" ovvero la percezione di modelli significativi all'interno di dati casuali. Questo spesso porta a credere, in modo errato, che dietro due eventi legati da una semplice correlazione sia presente una causa più grande. Per evitare eventi di questo tipo è necessario riuscire a creare dei metodi di visualizzazione che non inducono l'osservatore a creare dei collegamenti falsi tra due eventi che non sono correlati. Risulta quindi di primaria importanza fornire strategie per permettere la costruzione di grafici che non generino informazioni fittizie[14]:

- Mostrare chiaramente i dati.
- Indurre lo spettatore a pensare alla sostanza e non a quello che vede.
- Evitare di distorcere i dati.
- Rendere coerenti set di dati di grandi dimensioni.
- Consentire il cambiamento della granularità dell'informazione senza influire sulla comprensione

Questa naturale tendenza a preferire dati strutturati mette in primo piano l'importanza di proporre dei metodi adeguati che siano in grado di comunicare all'utente in modo preciso, diretto e immediato il contenuto che un insieme di valori vuole esprimere.

Una Piena comprensione dei dati permette la visualizzazione di anomalie in modo molto più immediato. Utilizzando la vista per la comprensione di grafici, siamo in grado di seguire la velocità del nostro pensiero piuttosto che dover attendere il risultato di una query scritta manualmente. La visualizzazione dei dati è come una forma di compressione delle informazioni, un modo per far entrare un'enorme quantità di informazione in un piccolo spazio[10].

Con L'aumento di sistemi distribuiti e tecnologie cloud, il numero di informazioni che un sistema produce inizia a raggiungere dimensioni notevoli. Le moderne tecniche di visualizzazione dati sono diventate quasi obbligatorie se si vuole riuscire a mostrare, possibilmente contemporaneamente, tutte le informazione che un sistema complesso produce. Le dashboard quindi sono una parte importante di qualsiasi sistema di monitoraggio poiché rappresentano il modo in cui lo stato del sistema viene presentato all'utente[7]. Si

vuole quindi riuscire a trattare lo sviluppo delle dashboard nello stesso modo in cui si opera l'implementazione del servizio che esse mostrano.

Un robusto sistema di monitoraggio dovrebbe consentire di visualizzare in modo conciso i dati delle metriche nei grafici ed essere in grado di strutturarli in tabelle o diagrammi. Le dashboard saranno le interfacce principali per la visualizzazione dei dati di monitoraggio, quindi è importante scegliere i formati che visualizzano più chiaramente i dati che ci interessano. Alcune opzioni includono *istogrammi*, *heat map* e *grafici in scala logaritmica*. Le piattaforme di creazione di dashboard spesso utilizzano metodi *drag and drop* ricorrendo all'approccio *what you see is what you get*. Ogni strumento mette a disposizione varie tipologie di grafico e una serie di opzioni per popolarlo offrendo anche diverse opzioni di formattazione della pagine come la disposizione dei grafici al suo interno, la frequenza di aggiornamento o il colore delle informazioni visualizzate.

Queste metodologie tuttavia, si scontrano con i processi di automazione del deployment proposti dalle correnti di pensiero come il sopracitato DevOps. Questo approccio genera una serie di problemi che vanno ad influire sul workflow di produzione di un servizio rendendo impossibile lo sviluppo parallelo della dashboard rispetto all'applicazione. Questa metodologia impone che una dashboard venga costruita utilizzando le funzioni interne del servizio solamente dopo che questo operi correttamente. Inoltre, nel caso si voglia riprodurre uno stesso grafico più volte e in più dashboard differenti, bisogna ripetere più volte la stessa operazione lasciando un largo spazio alla possibilità di errore umano.

Le moderne metodologie di sviluppo del software hanno influito pesantemente sulla creazione delle dashboard, introducendo il concetto delle dashboard-as-a-code. Trattando le dashboard come parte del codice principale è possibile sviluppare le due cose in contemporanea facendo seguire ad entrambi i moduli gli stessi step: sviluppo, aggiornamento e installazione del prodotto. Oltretutto è possibile adoperare tutti gli strumenti già esistenti per il controllo di versione del codice e profondamente consolidati nell'industria come ad esempio Git e Github. Esiste quindi la possibilità di usare gli stessi software per l'aggiornamento del codice e per l'aggiornamento della dashboard in contemporanea, così da far procedere di pari passo i dati raccolti e la loro elaborazione, con la relativa metodologia di visualizzazione corretta.

2.5 Alerting

L'alerting è una caratteristica tanto importante quanto delicata. Trovare il giusto compromesso tra quantità degli errori segnalati e l'effettiva importanza di questi, rimane un compito particolarmente difficile da compiere. Risolvere gli errori segnalati dal sistema è un'operazione estremamente dispendiosa in termini di risorse in quanto richiede misure straordinarie per la risoluzione di eventi imprevedibili. Ciò sottolinea l'importanza di scegliere con cura quale stato del sistema merita di scaturire un alert. In linea teorica

ogni sistema di alerting dovrebbe riuscire a rispettare al meglio queste regole ed evitare inutili sprechi di risorse[2]:

- Ogni tipologia di notifica deve essere utilizzata come strumento per la risoluzione di un problema. L'eccessiva produzione di notifiche, porta il personale a silenziarne la maggior parte e questa pratica può diventare nociva se non viene gestita con correttezza.
- Ogni notifica di un problema deve richiedere l'intervento del personale. Se la risposta ad un errore può essere automatizzata dallo stesso sistema di controllo, quella tipologia di errore dovrebbe risolversi in modo automatico, senza generare una notifica di alert e senza dover richiedere l'intervento del personale.
- Ogni volta che si riceve una notifica si deve essere in grado di reagire con prontezza e questo porta ad utilizzare una quantità di risorse che sono superiori a quelle normalmente disponibili. La prontezza della risposta tenderà a diminuire man mano che le notifiche di errore si accumulano.
- Ogni notifica dovrebbe riguardare una tipologia di errore che non si è mai verificata in precedenza. L'alerting di malfunzionamenti del sistema infatti dovrebbe fare da guida per lo sviluppo di migliorie al sistema stesso, fornire informazioni per il debugging dell'applicazione così che lo stesso errore non venga lanciato più di una volta.
- Segnalare un comportamento anomalo nel momento in cui una metrica smette di assumere i suoi valori standard e non quando una parte del sistema è già andata fuori uso (Fig.2.3)

Ovviamente questi obiettivi rimangono spesso ideali. Generalmente infatti un sistema di alerting reale deve disporre delle funzionalità che vanno in contrasto con i principi sopra elencati ma che sono vitali per un comodo utilizzo. Ad esempio Una caratteristica molto importante sta nella possibilità di classificare gli alert: più livelli di severità sono disponibili, più si potranno instrumentare risposte proporzionali. Risulta utile anche la possibilità di impostare diversi livelli di pericolosità per diversi alert oppure controllare in modo libero la generazione un alert senza dover limitarsi all'impostazione di soglie fisse. Altra opzione molto comoda è la possibilità di soppressione di un alert così da evitare segnalazioni inutili che tendono a distrarre i tecnici a cura del sistema, a patto che questa funzionalità rispetti delle regole. Per esempio: Quando tutti i nodi riscontrano più avvisi di uno stesso errore, è possibile generare un solo alert che identifichi il gruppo di errori che si sono verificati; oppure quando il servizio riscontra un rallentamento delle prestazione, non è necessario che avvisi anche gli errori che dipendono direttamente da questo. È di fondamentale importanza essere in grado di garantire che gli avvisi

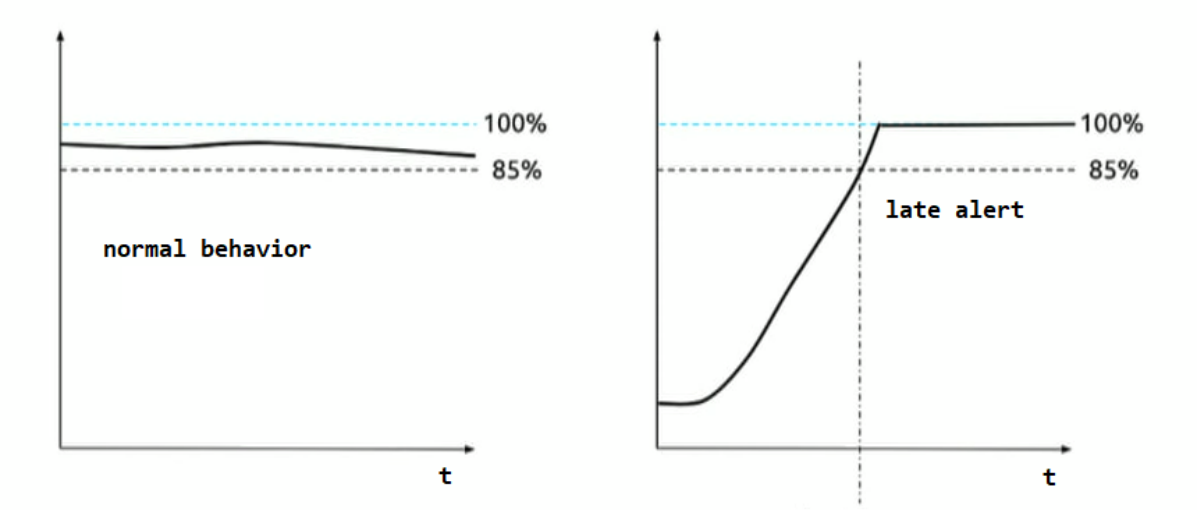

Figura 2.2: Gestione incorretta del lancio degli alert

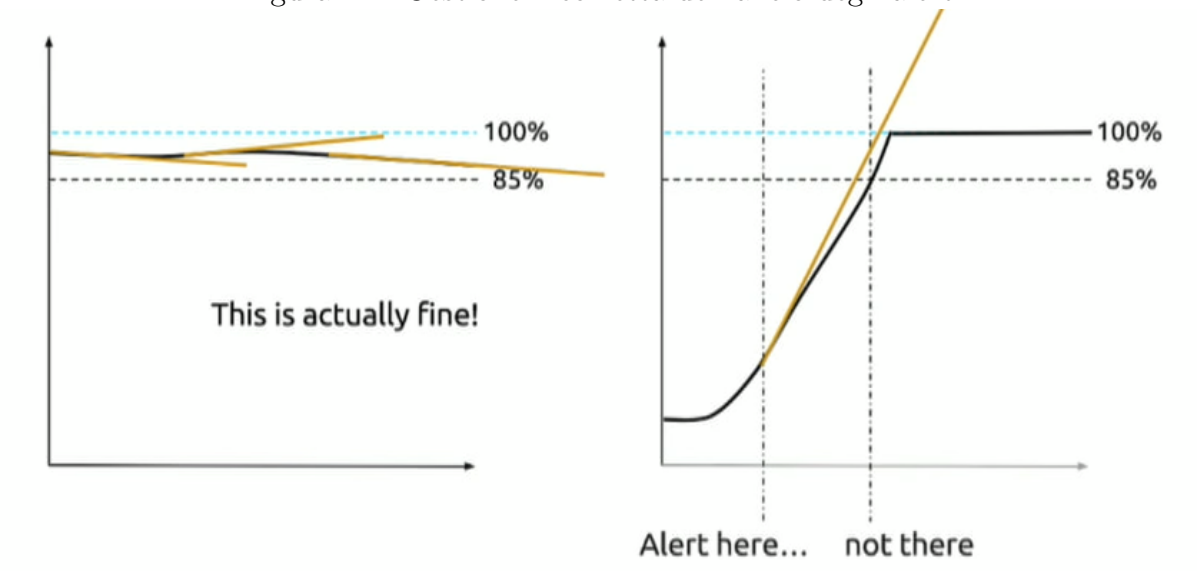

Figura 2.3: Gestione corretta del lancio degli alert

non vengano più soppressi al termine dell'evento, altrimenti si verrebbe a mancare la generazioni di nuovi alert che, se non gestiti con prontezza, potrebbero portare ad un failure dell'intero sistema.

Capitolo 3

Approcci e Tecniche per il Monitoraggio

Il monitoraggio dovrebbe essere uno strumento fondamentale per la gestione dell'infrastruttura su cui si basa un servizio, sia mentre questo viene sviluppato, sia dopo che questo è stato distribuito[1]. Il sistema che si adotta per il controllo del proprio sistema dovrebbe essere creato e distribuito di pari passo con l'applicazione stessa, in quanto strumento di fondamentale importanza per comprendere lo stato del sistema, dovrebbe inoltre diagnosticare prontamente problemi, pianificare investimenti su infrastrutture o fornire informazioni sulle prestazioni (Fig. 3.1)[2].

Un servizio di monitoraggio fornisce varie unità funzionali che ne definiscono il suo comportamento. Queste possono essere combinate in diversi modi per soddisfare i requisiti di monitoraggio che ci poniamo[9]. Ma quali sono gli obiettivi che il monitoring deve soddisfare per essere ritenuto utile; che informazioni deve darci la nostra architettura di controllo?

3.1 The Four Golden Signals

Latenza, traffico, errori e saturazione. Questi quattro valori riescono ad esprime le principali informazioni che sono utili per capire se un sistema risulta in salute e allo stesso tempo poter riuscire a rilevare potenziali guasti imminenti[2] specificando una serie metriche da monitorare. Le classi di metriche presenti sono più incentrate sull'applicazione piuttosto che sul sistema ospitante.

Latenza

La latenza indica il tempo che intercorre tra l'inizio di un'attività e il suo completamento. Può essere misurata a vari livelli, sia a livello di infrastruttura che di

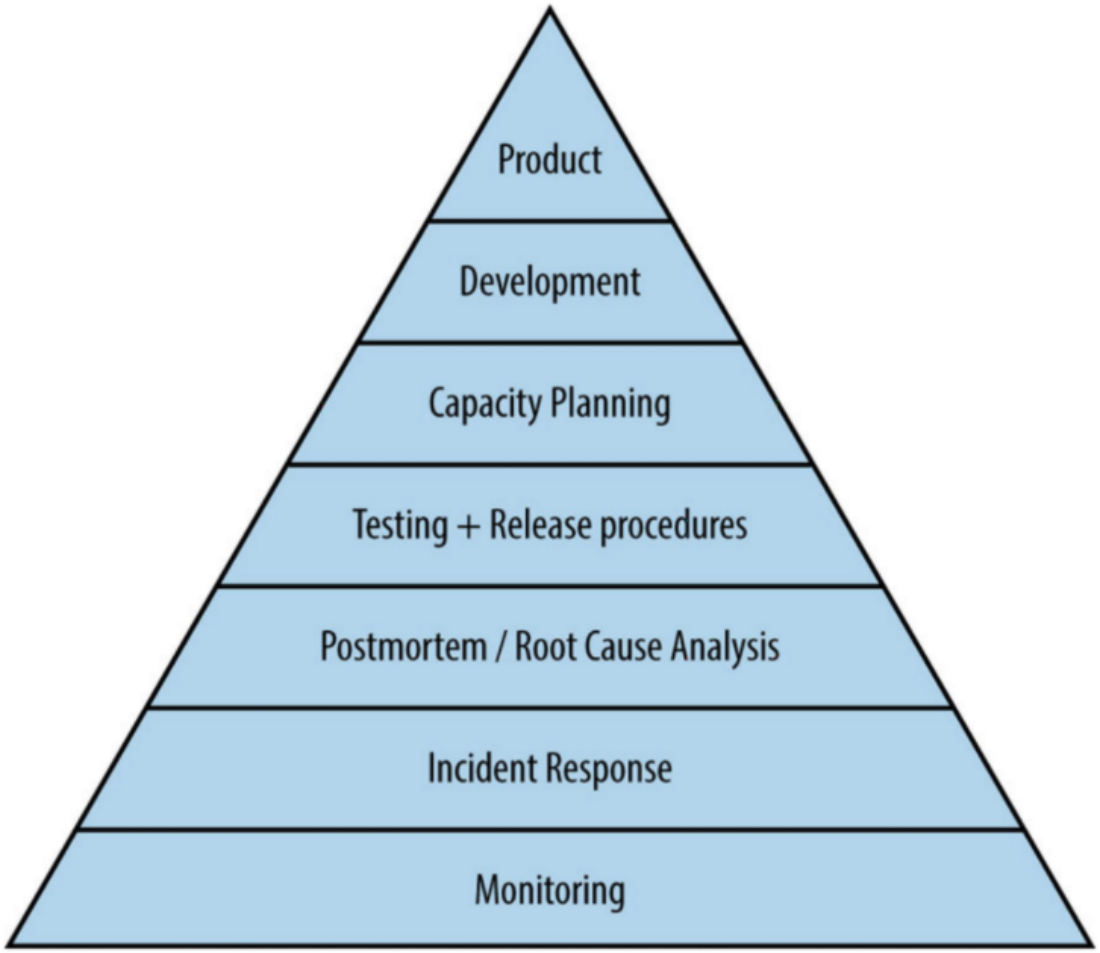

Figura 3.1: Importanza del monitoring secondo le metodologie Google SRE

applicazione.

Se per l'architettura del nostro servizio è sufficiente la misurazione all'interno di un singolo computer, questa può essere eseguita confrontando time stamp precedenti l'inizio dell'attività con quelli effettuati al compimento di questa. Nel caso si sviluppi il sistema in modo distribuito, questo aspetto richiede un'attenzione particolare a causa delle difficoltà di sincronizzazione di due macchine che non si trovano sulla stessa rete.

Il punto di vista cambia quando vogliamo monitorare la latenza da un punto di vista applicativo. Considerando strettamente il dominio applicativo possiamo calcolare la latenza riferendoci al periodo che va da una richiesta effettuata dall'utente alla soddisfazione di questa da parte del sistema. Abbassando il dominio ad un livello più vicino alla comunicazione, è possibile considerare la latenza come il periodo che intercorre tra l'inserimento di un messaggio su una rete e la ricezione di quel messaggio.

È possibile sottolineare la differenza presente tra la latenza delle richieste riuscite e la latenza delle richieste non riuscite. Anche gli errori possono darci informazioni, d'altra parte, un errore lento crea più problemi di un errore veloce. Studiare la tipologia di errore può aiutarci anche a capire quale sia la causa di questo. Più avanti approfondiremo la differenza presente tra un *sintomo* e la sua *causa*.

Traffico

Il traffico è la misura della quantità di richieste di cui il server si prende carico. Utilizzando un sistema basato sulle metriche è possibile misurare questa informazione pur non essendo il traffico un concetto legato strettamente all'hardware ma piuttosto al livello applicativo. Prendendo in esempio un servizio Web, questo valore è solitamente misurato basandosi sul numero di richieste HTTP al secondo, o il numero di operazioni che stanno eseguendo all'interno di un database suddivise in base alla tipologia di queste richieste, tenendo come punto di riferimento il dominio di appartenenza dell'applicazione.

È possibile considerare il numero di operazioni in una specifica unità di tempo. Questo tipo di dato prende il nome di *throughput*. Solitamente analizzare la latenza del traffico è una misura più utile a livello infrastrutturale. Il traffico fornisce una misura a livello di sistema che coinvolge tutti gli utenti, mentre la latenza è focalizzata su un singolo utente. La relazione che intercorre tra throughput dipenderà dal numero di utenti e dal tipo di operazione che quegli utenti stanno compiendo. In questo caso specifico la riduzione del throughput non è di per sé un problema in quanto può essere causata da una riduzione del numero di utenti. I problemi sono indicati attraverso l'analisi del throughput in corrispondenza degli utenti attivi.

Errori

Questo parametro indica la frequenza con cui si presentano gli errori. Sotto questo parametro rientrano sia le richieste che non riescono ad essere portate a termine sia quelle

che restituiscono un contenuto sbagliato o quelle che non riescono a superare determinate soglie temporali imposte a tempo di design del sistema. È giusto notificare anche che qualsiasi elemento dell'infrastruttura fisica può guastarsi generando di conseguenza degli errori; questa tipologia di problema può derivare sia dalla rottura di componenti fisici sia dalla saturazione del traffico sulle linee di comunicazione. Il rilevamento dei guasti dell'infrastruttura fisica è un problema che spesso è delegato al provider che fornisce l'host fisico e di conseguenza in questa tesi questo argomento non verrà approfondito.

Anche il software, allo stesso modo, può smettere di funzionare, sia totalmente che parzialmente. Ironicamente la categoria di errori più facili da rilevare è quella dei guasti totali. Quando dei dati non transitano dove dovrebbero, viene segnalato immediatamente un problema. Gli errori più difficili da rilevare derivano da guasti parziali che non provocano uno stop del sistema nella sua interezza ma solamente in una delle sue sottoparti e spesso sono la causa del degrado delle prestazioni. I guasti totali possono essere rilevati facendo delle verifiche sulle API che il servizio espone, in questo modo si andrà a verificare che tutto il servizio operi in buona salute. Gli errori parziali richiedono un'osservazione interna che vada a verificare log e metriche prodotte dalle singole componenti che permettono il funzionamento dell'intero servizio così da verificare la presenza di malfunzionamenti.

Saturazione

Questo valore indica quanto è in uso il servizio che stiamo controllando; una misura del sistema che enfatizza le risorse più limitate. Molti sistemi subiscono un degrado in termini di prestazioni prima di raggiungere il 100% di utilizzo, quindi è essenziale riuscire a controllare quanto carico di lavoro sta venendo gestito dal nostro sistema. Il rilevamento del degrado delle prestazioni è, probabilmente, l'uso più comune dei dati di monitoraggio. Questo viene osservato confrontando le prestazioni attuali con i dati storici prodotti dal programma o con le stime fatte in fase di design. Idealmente un ottimo sistema di monitoraggio rileva il degrado delle prestazioni prima che gli utenti subiscano un impatto notevole. Gli aumenti della latenza sono spesso l'indicatore principale della saturazione.

3.2 Sintomi e cause

I Four Golden Signals selezionano gruppi di metriche di alto livello. Ogni gruppo corrisponde ad uno dei corrispettivi segnali. Se una di queste metriche genererà un avviso sarà possibile diagnosticare un problema in modo preciso. Le infrastrutture di monitoraggio consentono a un sistema di dirci quando è guasto, oppure di comunicarci cosa sta per rompersi. Quando il problema non può risolversi in modo automatizzato, vogliamo che un essere umano indaghi sull'alert riportato dal sistema, determini se questo rappresenti un problema reale, risolva il problema e successivamente sia anche in grado

Symptom	Cause
I'm serving HTTP 500s or 404s	Database servers are refusing connections
My responses are slow	CPUs are overloaded by a bogosort, or an Ethernet cable is crimped under a rack, visible as partial packet loss
Users in Antarctica aren't receiving animated cat GIFs	Your Content Distribution Network hates scientists and felines, and thus blacklisted some client IPs
Private content is world-readable	A new software push caused ACLs to be forgotten and allowed all requests

"What" versus "why" is one of the most important distinctions in writing good monitoring with maximum signal and minimum noise.

Figura 3.2: Esempi di sintomi messi a confronto con le possibili cause

di determinare la causa principale in modo tale che questa possa essere corretta. Per fare questo il sistema di monitoraggio dovrebbe rispondere a due domande: *cosa non funziona e perché?* Il *che cosa è rotto* indica il sintomo mentre il *perché* indica una causa. Distinguere il *cosa* dal *perché* è la caratteristica principale che permette a un sistema di monitoraggio di essere molto espressivo evitando allo stesso tempo di essere inutilmente verboso[2] (Fig. 3.2).

3.3 Monitoraggio black-box e white-box

Un altro acceso dibattito sul tema del monitoraggio vede come protagonista la domanda "come chiedo al programma i dati da associare alle metriche?". L'evoluzione delle metodologie di sviluppo introdotte dalle moderne correnti di pensiero come la sopracitata DevOps hanno cambiato anche i modi di controllare il comportamento di un programma. La cooperazione tra sviluppatori e tecnici promossa da DevOps si riflette anche nella costruzione dell'applicazione stessa promuovendo l'utilizzo di prodotti white-box dove il programma fornisce metodi per instrumentare del codice al proprio interno. L'interfaccia di un oggetto può essere divisa in due parti: un'interfaccia operativa che supporta il normale adempimento delle operazioni di elaborazione delle informazioni e

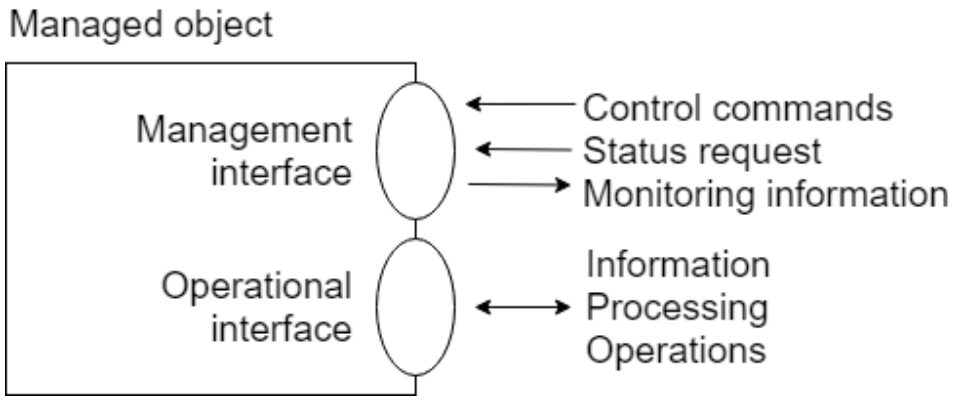

Figura 3.3: Interfaccia di monitoring

un'interfaccia di controllo che supporta il monitoraggio in modo tale da poter conoscere lo stato interno dell'applicazione [9](Fig. 3.3).

Questo impone che gli sviluppatori debbano iniziare a considerare che il loro prodotto sarà monitorato e quindi fornire modi pratici per riuscire a tener traccia dello stato del proprio programma. Solitamente la divisione sviluppo di un'azienda rilasciava un prodotto che gli operatori di sistema utilizzavano senza sapere come funzionasse al suo interno. Questo classico approccio viene definito black-box. A questo software ci si interfacciava tramite un terminale e si impartivano comandi per verificare che il suo comportamento fosse quello atteso. La possibilità di parlare con il programma tramite endpoint di rete o l'uso di query da interpretare, implica una cooperazione con l'applicazione in grado di mostrare quello che sta succedendo al suo interno; questo approccio viene definito white-box. Ognuna delle due metodologie ha i proprio vantaggi e svantaggi. È possibile combinare l'uso del monitoraggio white-box con usi modesti ma critici del monitoraggio black-box, in modo da riuscire a sfruttare i punti di forza di entrambe le metodologie. Il modo più semplice per pensare al monitoraggio black-box rispetto al white-box è che il primo è tendenzialmente orientato al rilevamento dei sintomi e rappresenta problemi che sono attualmente in corso e non previsti: *il sistema non sta funzionando*. Il monitoraggio white-box consente quindi di rilevare problemi imminenti. Questa proprietà dipende direttamente dalla capacità di ispezionare le parti interne del sistema instrumentando del codice apposito. Quando si raccolgono i dati di telemetria per il debug, il monitoraggio white-box è essenziale. Per notificare alert a priorità elevata, il monitoraggio black-box ha il vantaggio di rilevare immediatamente la presenza di un problema. D'altra parte, per problemi non ancora presenti ma imminenti, risulta inefficace.

3.4 Come utilizzare i dati di monitoraggio

Le informazioni che si ottengono dall'analisi dei dati di monitoraggio possono supportare molteplici operazioni che trovano un'utilità fin dalle prime fasi di sviluppo dell'applicazione. Le informazioni acquisite permettono un'analisi del servizio da un punto di vista più tecnico favorendo debug, test, valutazione delle prestazioni e documentazione dinamica del sistema. È possibile studiare anche elementi di stampo più manageriale come il capacity planning e il responso che l'utenza ha nei confronti del servizio.

Debug

Il debug del sistema viene eseguito usando un approccio di tipo Bottom-up [7] in modo che i singoli moduli vengano sottoposti alle procedure di debug separatamente, e solo successivamente integrati per formare il servizio. Questo metodo permette di favorire uno sviluppo *loosely-coupled* [9] [2] in cui i singoli servizi sono il più possibile indipendenti. Successivamente ci si concentra sulla comunicazione tra i processi (IPC) che ricopre una parte molto complessa del debugging del servizio. Questa viene fatta (all'interno di un metric based system) combinando le informazioni ottenute dai log dei singoli servizi con le rispettive metriche[3]. Un requisito importante per il debug è la ripetibilità e la riproduzione dello stato del sistema. Il ruolo del monitoraggio consente di controllare le cause di eventi non deterministici così da avere la possibilità di riprodurli. La capacità di ripetere fedelmente l'esecuzione di un sistema consente di isolare e identificare gli errori che si manifestano sui singoli percorsi di esecuzione studiando i microservizi chiamati in causa[7].

Pianificazione degli investimenti

Il capacity planning riguarda l'utilizzo dei dati di monitoraggio per prendere delle decisioni sulle modalità di messa in produzione e raggiungibilità del nostro servizio come ad esempio capire se si preferisce delegare la gestione a server esterni oppure gestire tutto all'interno di una rete personale.

Questo tipo di operazione si divide in due categorie principali: il capacity planning a lungo termine e quello a breve termine. Pianificare scelte a lungo termine mira a soddisfare le esigenze hardware e/o virtuali riferendosi ai cambiamenti del carico di lavoro nel tempo. Nel caso si parli di un data center fisico questo implica comprare dell'hardware. In un data center virtualizzato, si tratta di decidere il numero e le caratteristiche delle risorse virtuali da allocare. In entrambi i casi il processo di pianificazione delle risorse computazionali dipende dai valori del carico di lavoro raccolto dai dati di monitoraggio e da una proiezione del carico di lavoro futuro in base a considerazioni estrapolate dai dati raccolti.

Pianificare a breve termine nel contesto di un ambiente virtualizzato, significa creare una nuova macchina virtuale (VM) in modo da poter suddividere il carico di lavoro in più nodi operatori oppure, quando non è richiesta un grande potenza di calcolo, eliminarne una esistente. Questo tipo di operazione è delegata al *load balancer*, un componente che le compagnie di gestione di cloud computing offrono per permettere al servizio che viene hostato di non sovraccaricarsi. Se si decide di gestire un programma all'interno di una rete privata questo aspetto assume un'importanza minore.

Interazione dell'utente

La soddisfazione dell'utente è un elemento di fondamentale importanza per la valutazione del servizio che si fornisce. Riuscire a migliorare l'utilizzo del servizio al nostro utente finale è la caratteristica comune a tutte le infrastrutture di monitoraggio[7].

Oltre all'utilità e alla qualità dell'applicazione stessa, la soddisfazione dell'utente dipende da altri fattori che possono essere controllati. Come discusso in precedenza la latenza di una richiesta dell'utente è uno di questi. Gli utenti si aspettano tempi di risposta che siano il più possibile brevi. A seconda dell'applicazione, variazioni apparentemente banali nella risposta possono avere un grande impatto.

Un altro aspetto per riuscire a misurare il tipo di interazione dell'utente sta nella scelta accurate del set di metriche. Passando per un esempio banale, se si gestisce un sito Web di gallerie fotografiche, si è interessati a metriche come velocità di caricamento delle foto, dimensioni delle foto, tempi di elaborazione delle foto ecc. Altre organizzazioni avranno metriche diverse, ma dovrebbero essere tutte indicatori importanti per capire come gli utenti stanno utilizzando il nostro servizio. Queste tipologie di informazioni vengono raccolte tramite due principali metodologie.

La prima viene chiamata *Real User Monitoring* (RUM) [1]. RUM registra essenzialmente tutte le interazioni dell'utente con un'applicazione. I dati RUM vengono utilizzati per valutare il livello di servizio reale che un utente sperimenta e se le modifiche lato server vengono propagate correttamente agli utenti. Il secondo viene definito *monitoring sintetico*. Questo viene identificato nell'esecuzione di stress test sul servizio che si vuole proporre. I comportamenti previsti degli utenti vengono inseriti tramite script utilizzando un sistema di emulazione o un client. Tuttavia, l'obiettivo spesso non è quello di verificare il comportamento del sistema quando subisce una grossa mole di dati da elaborare, ma quello di monitorare l'esperienza dell'utente in modo riproducibile, così da rendere questo tipo di test automatizzato.

3.5 Errori comuni e anti-pattern

In qualsiasi metodologia di sviluppo, è una buona norma identificare ciò che si desidera creare ancora prima di crearlo. Solitamente però si tende a concentrarsi solamente

sull'applicazione lasciando il monitoring e altre funzioni operative, ad esempio la sicurezza, come componenti aggiuntive dell'applicazione piuttosto che inserirle tra le funzionalità di base. Il monitoring, come la sicurezza, è una caratteristica fondamentale dello sviluppo software. Ovviamente questa pratica inizialmente impiega un maggior utilizzo di risorse specialmente se l'architettura di controllo è stata configurata in maniera superficiale. Queste pratiche causano il verificarsi di problemi comuni che vengono definiti anti-pattern.

Di seguito analizzeremo quando questi errori si verificano e cosa comportano per poi successivamente descrivere come i dati raccolti dovrebbero essere utilizzati e che tipo di relazioni possiamo ricavare da quelle informazioni.

Monitorare un dominio sbagliato

Un comune errore che si tende a commettere è monitorare lo stato dei servizi su un host ma non l'effettivo funzionamento. Ad esempio, è possibile monitorare se un'applicazione Web è in esecuzione controllando un codice di risposta HTTP 200. Questo mostrerà che l'applicazione sta rispondendo alle connessioni, ma non se sta restituendo i dati corretti in risposta a quella specifica richiesta. Un approccio migliore consiste nel monitorare innanzitutto la correttezza di un servizio, verificando che una risposta positiva indica allo stesso tempo anche una risposta corretta per quanto riguarda i dati ricevuti. Questa tipologia di operazione consente di ottenere entrambe le informazioni in quanto una dipende dall'altra.

Usare soglie di alerting statiche

I controlli e gli avvisi sono spesso innescati da una logica booleana rigida. Il monitoraggio tradizionale si basa in gran parte sul rilevamento delle anomalie. Si conosce il profilo del sistema durante il suo normale funzionamento, si impostano le soglie sui valori rilevati e si controllano per trovare comportamenti questi ultimi anomali. Nel caso un valori superi una soglia viene segnalato un errore. Questo potrebbe non essere sufficiente poiché potrebbe risultare impossibile definire dei valori di soglia che identificano il comportamento standard del sistema. Al fine di poter monitorare correttamente bisogna fare riferimento a finestre di dati più ampie, non a punti statici nel tempo, utilizzare tecniche più intelligenti per calcolare valori e soglie in base allo specifico dominio di appartenenza dei dati.

Monitorare l'applicazione passando per l'hardware

Un altro errore comune è quello si controllare l'uso di cpu, memoria e disco sui vari host e ignorare tutte le informazioni chiave che indicano che l'applicazione sta funzionando correttamente. Un buon approccio è quello di progettare un piano di monitoraggio top-down basandosi su un livello di dominio più alto, identificare i piani dell'applicazione

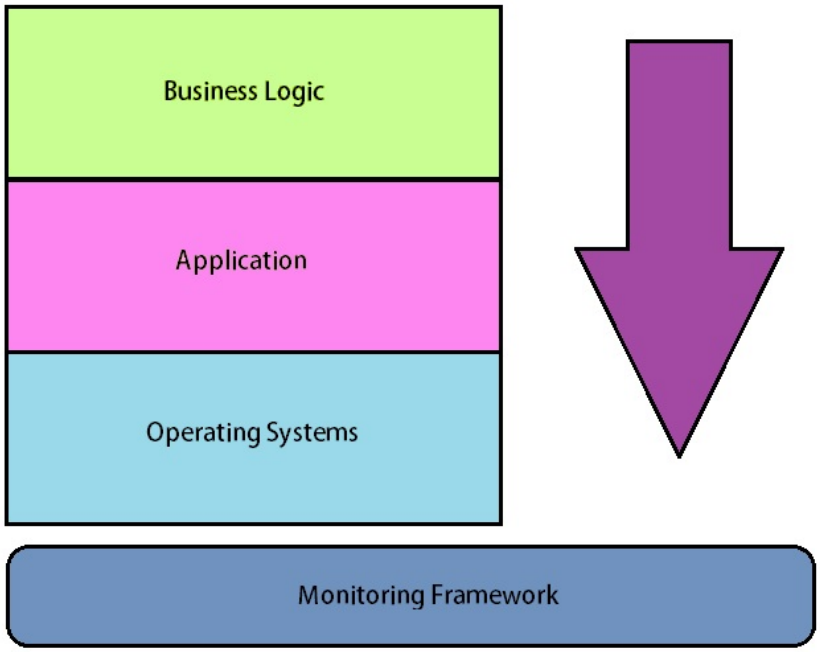

Figura 3.4: Logica di monitoring applicativa

che forniscono queste informazioni e utilizzarle come base del proprio sistema di controllo (Fig. 3.4). Iniziare con la logica del servizio, passare per quella applicativa e solamente alla fine andare a controllare lo stato dell'infrastruttura hardware non implica ignorare lo stato dell'hardware su cui l'applicazione sta funzionando.

Utilizzare configurazioni statiche

Un altro motivo per cui spesso non ci si concentra abbastanza nel creare una corretta infrastruttura di monitoraggio sta nella difficoltà di riuscire a far procedere di pari passo lo sviluppo delle funzionalità principali dell'applicazione con l'aggiornamento della configurazione del sistema di controllo. Trattare la configurazione del sistema *as a code* permette di tenerne traccia all'interno di un DVCS [3]. Questo tipo di approccio permette di fornire tutte operazioni disponibili per il controllo di versione sia per il software, sia per le configurazioni del sistema permettendo così di usufruire di ovvi vantaggi quali: cronologia delle modifiche, collegamenti da modifiche precedenti e tracciamento delle attività.

Se si utilizza un approccio particolarmente statico, questo porterà ad un eccessivo uso di risorse per la configurazione dei controlli, aspetto che molti reputano erroneamente secondario, manifestando così una tendenza a tralasciare questa operazione. Il risultato

sarà evitare di curare la parte di monitoring del sistema rendendo questa operazione sempre più complessa nel futuro a causa dell'aumento della complessità del servizio.

La configurazione del proprio sistema di monitoraggio dovrebbe essere automatizzata ove possibile. L'aggiunta di host e servizi dovrebbe avvenire tramite rilevamento automatizzato con metodologie *service discovery*, in modo che le nuove applicazioni possano essere monitorate automaticamente anziché richiedere che qualcuno le aggiunga manualmente. Utilizzando una configurazione manuale si andrà a definire un modello troppo statico che richiederà un utilizzo sempre maggiore di risorse. Questo potrebbe portare a trascurare il sistema di controllo poiché considerato troppo dispendioso, riscontrando poi effetti negativi sull'interazione dell'utente e sulla difficoltà di risoluzione dei problemi.

3.6 Tecnologie di monitoring

Fino a questo momento abbiamo visto le caratteristiche che descrivono un ottimo servizio di monitoraggio, analizzato i pro e i contro di ogni scelta architetturale, approfondito cosa ogni sistema di monitoring deve controllare e che tipo di informazioni possiamo acquisire in base alle metriche che studiamo. Ovviamente non esiste una soluzione perfetta, per cui ogni tecnologia di monitoraggio si focalizza su determinati aspetti in modo da raggiungere un particolare compromesso, concentrarsi su un determinato tipo di dato o sviluppare una specifica funzionalità tra quelle discusse sopra.

Nel mercato sono presenti svariate tecnologie di monitoraggio divise per funzionalità e ambiti. Non tutte riescono ad adattarsi alle richieste per lo sviluppo di questo progetto. Alcune di queste non dispongono di una capacità espressiva sufficiente per poter soddisfare gli obiettivi richiesti in fase di design del progetto. Alcuni sistemi di controllo possono essere direttamente forniti anche dal servizio stesso, ad esempio il sistema di database di MongoDB mette a disposizione una propria interfaccia grafica accessibile da browser che permette di visualizzare le operazioni interne del database. Ovviamente questi controlli così specifici non permettono di dare una visione generale dell'applicazione e abbiamo scelto di non utilizzarli. Sono disponibili anche protocolli che forniscono opzioni di monitoring nel caso non si vogliano utilizzare servizi integrati in quanto poco trasparenti e improntati più ad un utilizzo aziendale invece che accademico.

Un esempio di questi è il protocollo SNMP sopra discusso, che nonostante venga ancora largamente usato [4], utilizza un modello di dati eccessivamente statico per le finalità di questo progetto di tesi. Si è deciso quindi di utilizzare le metriche in quanto abbastanza elastiche da muoversi in tutti i livelli del dominio e in grado di adattarsi molto facilmente al sistema sottostante. Nel caso il proprio servizio venga rilasciato in una piattaforma cloud come *Amazon Aws* o *Google cloud* è possibile utilizzare le funzionalità di controllo integrate in quelle piattaforme. Al loro interno sono già presenti tutti i cinque aspetti necessari al funzionamento del monitoring. La tecnologia cloud non è però utilizzata all'interno di questa tesi in quanto il progetto verrà adoperato all'interno di una

rete privata e di conseguenza è necessario riuscire a fornire tutti i componenti necessari per il pieno funzionamento dell'architettura di controllo.

Una delle funzionalità da inserire all'interno della propria rete di monitoraggio è quella scegliere come gestire lo storage delle metriche. Questo può influenzare prestazioni di monitoraggio e capacità di scalabilità. Il metodo di archiviazione più efficiente per questo tipo di dato e anche il più utilizzato è quello basato su un *Time Series Database* in grado di permettere la memorizzazione di grandi quantità di metriche [4]. I principali sistemi di database che forniscono questa tipologia di archiviazione possono essere identificati in Apache Cassandra, HBase, InfluxDB, Elasticsearch e Prometheus. Ogni database fornisce le proprie specifiche metodologie per instrumentare il proprio codice ed eseguire le letture dei dati.

Per questo lavoro la scelta è ricaduta su Prometheus. La possibilità di formare un cluster di dati è ottenuta tramite *federation approach* [13]. Questa tecnica permette a Prometheus di raccogliere i dati da altri Prometheus database nella stessa maniera con la quale si raccolgono informazioni dalle applicazioni da monitorare. Questa funzionalità è integrata all'interno di Prometheus, a differenza di altri sistemi di storage come *influx DB* che permettono la possibilità di fare clustering solamente nelle versioni a pagamento del servizio. Un'altra caratteristica di Prometheus, infatti, è quella di essere completamente open source pur rimanendo un sistema in grado di fornire vaste funzionalità, scalabilità e la possibilità di adattarsi alle esigenze di qualsiasi sistema grazie ad una vasta community e la possibilità di essere configurato in ogni suo aspetto[8].

Esistono altri sistemi di monitoraggio open source, ma non tutti si adattano alle esigenze del progetto. Una tra queste è Nagios, ma l'utilizzo di SNMP rende poco scalabile l'applicazione all'interno dello stack applicativo; allo stesso modo Open NMS è stato scartato in quanto rimane uno strumento più adatto al monitoraggio stretto di organi di rete. Altri sistemi come Hyperic Hq riescono a fornire anche dati più vicini al livello applicativo ma rimangono vincolati all'utilizzo di sistemi basati su Windows. Altra fondamentale caratteristica di Prometheus, infatti, è la capacità di essere indipendente dal sistema operativo sottostante; questa opportunità si sposa perfettamente con l'ambiente dockerizzato sul quale verrà rilasciato il sistema di monitoraggio.

Poiché si andrà a monitorare un'applicazione già sviluppata, si vuole rendere meno invasivo possibile il sistema di monitoraggio, conseguentemente si vuole optare per le metodologie agent-less così da limitare lo sviluppo di parti ausiliarie e favorire la scalabilità sia del servizio che del sistema di monitoraggio in modo parallelo. La visualizzazione dei dati è stata affidata a Grafana grazie al supporto nativo con Prometheus e la possibilità di utilizzare lo stesso linguaggio di query per poter popolare i grafici che mette a disposizione. Di seguito parleremo in modo più approfondito di Prometheus e Grafana in modo da spiegare il loro funzionamento e capire meglio perché sono stati scelti per instaurare l'infrastruttura di monitoraggio di Trauma Tracker.

3.6.1 Prometheus

Fino a questo punto abbiamo analizzato le necessità del monitoraggio di sistemi moderni studiando i diversi tipi di approcci disponibili. Per approfondire l'argomento di seguito verrà analizzato in modo specifico Prometheus per poter sottolineare l'importanza di alcuni concetti accennati in precedenza e di come questi vengono ritrovati in pieno all'interno del funzionamento di Prometheus stesso fin dalle prime fasi di design.

Questo servizio è stato creato da ex dipendenti Google passati a lavorare per l'azienda Soundcloud. Prima della diffusione delle tecnologie di containerizzazione, Soundcloud stava già ristrutturando l'architettura software, aprendosi ai sistemi basati sui microservizi. Le vecchie metodologie di controllo risultavano particolarmente difficili da integrare con la nuova struttura basata sui container. Prometheus nasce per poter superare queste difficoltà riuscendo così a diffondersi come prodotto pioniere per quanto riguarda questo nuovo modo di introdurre il monitoring in un'architettura a microservizi. Scritto in Go, Prometheus è un software open source e stand-alone mantenuto in modo indipendente. Si è unito all'iniziativa *Cloud Native Computing Foundation* ed è secondo a contributi soltanto a Kubernetes.

Architettura

Il componente principale che permette il vero e proprio funzionamento di Prometheus è il Prometheus server. A sua volta questo si divide in 3 moduli, ognuno con una specifica funzione (Fig. 3.5) [13]:

- **Retrieval data**: questa modulo si occupa di raccogliere le metriche prodotte dalle varie applicazione che devono essere monitorate. Queste vengono collezionate tramite delle richieste HTTP utilizzando un approccio di tipo pull. Questa scelta caratterizza fortemente l'intero metodo di funzionamento di Prometheus e verrà discussa in maniera più approfondita all'interno della sezione successiva.

- **Time Series Data Base**: questo modulo si occupa della memorizzazione vera e propria di tutti i dati raccolti dalle applicazioni. Su questi dati vengono poi applicate le query che consentono di visualizzare i dati stessi e anche i vari controlli per l'attivazione di alert se determinati valori si comportano in modo inaspettato. Le funzioni di questo database sono ottimizzate per la gestioni delle metriche di monitoraggio.

- **HTTP server**: questo elemento permette a Prometheus di potersi interfacciare con elementi esterni che sono in grado di espanderne o migliorarne le funzionalità. Ad esempio, nonostante Prometheus disponga già di una rudimentale interfaccia grafica, è possibile delegare il compito della visualizzazione di dati a software di terze parti che si interfacceranno al server principale tramite questo endpoint.

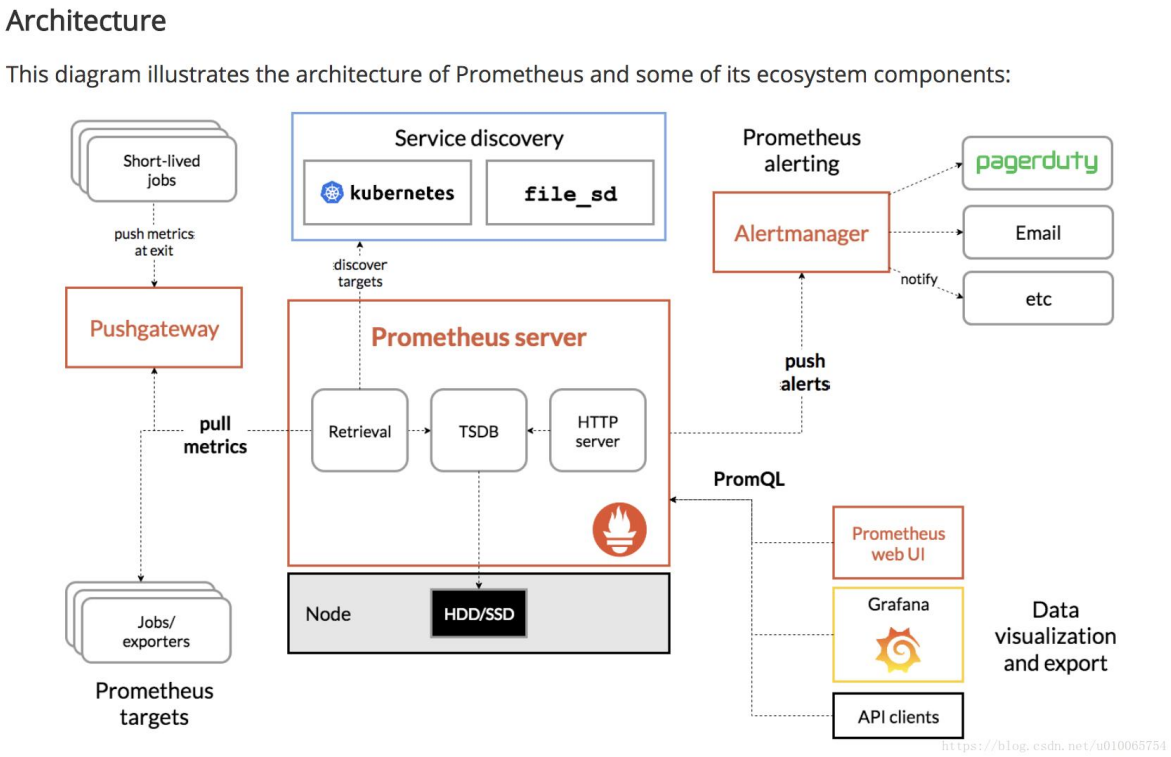

Figura 3.5: Architettura di Prometheus

Oltre alle funzionalità offerte dal Prometheus server, esistono dei moduli esterni che possono essere configurati in modo da permettere a Prometheus di adattarsi alle esigenze specifiche di ogni sistema.

- **Push gateway**: in casi molto specifici è possibile cambiare la metodologia di raccolta dei dati. Questo modulo mette a disposizione anche un metodo push per la raccolta dei dati di tutti quei task che sono troppo corti e non riescono a superare l'intervallo di tempo presente tra due scrape successivi.
- **Service discovery**: questa funzionalità permette a Prometheus di poter essere configurato in maniera più elastica, lasciando l'aggiunta di nuove applicazioni da monitorare al servizio di gestione dei container in cui Prometheus è inserito. Questo metodo permette di abbandonare la staticità data dalla configurazione basata su file YAML.
- **Alert manager**: come approfondiremo più avanti, all'interno di Prometheus verranno generati solamente degli alert nel caso in cui alcuni valori vadano fuori scala. Il compito di comunicare ad un amministratore del sistema che qualcosa non sta funzionando andrà delegato a questo componente aggiuntivo.

Funzionamento

Dopo aver mostrato come è strutturato il prometheus server, andremo ad analizzare come funziona internamente, e perché sono state prese determinate decisioni a livello di design e come queste caratterizzano fortemente il suo funzionamento.

Pull/Push scraping

Le metriche vengono acquisite utilizzando un approccio pull e non push. Ovvero Prometheus chiede periodicamente alle applicazioni o agli eventuali exporter lo status dei servizi immagazzinando le informazioni nel proprio database. Utilizzando questo approccio non è necessario scrivere un software per ogni applicazione. Questo permette anche di capire in anticipo quando un'applicazione smette di funzionare senza dover aspettare che venga rilevato il timeout di controllo di un servizio: se questa non è più raggiungibile dalle richieste di pull allora sarà presente un problema. Il principale difetto di questo sistema si presenta quando bisogna monitorare task più brevi dell'intervallo di pull. Questi rischiano quindi di essere persi. Per questo Prometheus mette a disposizione anche delle metodologie di push ma rimane comunque una metodologia consigliata solamente se è strettamente necessaria in quanto va a cozzare con il design stesso di Prometheus.

Utilizzando il sistema di pull, risulta anche più facile la condivisione dei dati tra vari server Prometheus. In un'ottica distribuita non è necessario richiedere di farsi inviare dei dati ma basta aggiungere ai target il server che si vuole interrogare alla lista degli

scrape. Questa operazione si basa solo sulla corretta configurazione del proprio sistema di monitoraggio senza dover andare ad intaccare sistemi esterni e quindi risulta molto gestibile all'interno di una rete proprietaria. Così facendo è possibile ottenere repliche dei dati, in modo diverso dal solito. Si usa quello che viene definito *federation approach*. I database non comunicano tra di loro per verificare la sincronizzazione delle informazioni, bensì fanno il pull degli stessi dati. Dallo stesso insieme sarà possibile rilevare gli stessi alert e inviarli all'alert manager. Nel caso si verifichino due alert uguali l'alert manager invierà comunque una sola segnalazione riconoscendo l'uguaglianza dei due alert in base alle label.

L'approccio pull ovviamente obbliga la conoscenza di ogni singolo microservizio presente nell'ecosistema di monitoraggio rendendo complessa la configurazione di Prometheus. D'altro canto questo consente di non dover andare a modificare i singoli servizi, permettendo così l'inserimento dell'architettura del monitoraggio in modo non invasivo. Oltretutto i moderni servizi di containerizzazione offrono funzionalità di service discovery che permettono di conoscere dinamicamente la "posizione" dei container all'interno della rete di microservizi.

Alert manager

Dopo che le metriche sono state raccolte, Prometheus analizza i dati contenuti all'interno del suo database e li confronta con i parametri impostati dall'utente che definiscono quando generare un alert. Gli alert quindi nascono all'interno di Prometheus, ma la comunicazione agli utenti esterni è lasciata all'alert manager. Quando viene generato un alert questo viene comunicato all'alert manager. Questo componente, se opportunamente configurato, permette di passare gli alert generati all'interno di Prometheus ad un sistema di notifica esterno che è in grado di raggiungere l'utente. È possibile scegliere vari software per la comunicazione di questi messaggi a patto che questi espongano le corrette API.

La possibilità di decidere in modo molto preciso quando viene generato un alert è uno dei punti di forza di Prometheus. Spesso è necessario riuscire a segnalare un problema quando un valore inizia a comportarsi in modo anomalo e non quando una parte del sistema è già andata fuori uso. Prometheus non si limita a impostare delle soglie di valori ma consente impostare la segnalazione di un andamento pericoloso.

Time Series data base

Prometheus colleziona le metriche inserendole nel proprio *Time series database*. Generalmente questo tipo di dato viene identificato come un singolo nome lungo, separato da punti per specificare con più chiarezza cosa rappresenti il singolo valore. Il problema che si presenta con questa forma di rappresentazione è quello di mostrare grosse difficoltà nell'esplicitare bene cosa la metrica voglia comunicare e inoltre vengono introdotti livelli di gerarchia che non sono basati su alcuna struttura logica; oltretutto questo sistema

rende molto complicato fornire delle procedure sensate per quanto riguarda l'aggiunta di nuove metriche.

Prometheus si sposta su un *label base data model*. In questo sistema ogni metrica è identificata da un nome che rappresenta cosa il valore numerico va a definire e una serie di label, in forma di chiave valore. Ogni insieme univoco di coppie chiave-valore offre una serie temporale che nel suo insieme va a costituire una metrica. A livello di struttura le metriche sono mantenute come una lista identificata da un ID che è definito dal nome della metrica e dalle sua label. La lista contiene delle tuple definite da un time-stamp e il valore registrato dal sistema di monitoraggio in quell'istante di tempo. Prometheus mantiene tutti i dati in un singolo database evitando di fare repliche per la persistenza dei dati. Questa scelta infatti è stata fatta a livello di design poiché riuscire a creare ridondanza e persistenza richiede l'utilizzo di molte risorse di rete e solitamente riuscire a soddisfare queste due proprietà è la prima causa di saturazione della rete. Oltretutto le metriche che si memorizzano sono un tipo di dato che perde la sua importanza man mano che il tempo avanza, quindi memorizzare in modo persistente dati particolarmente vecchi risulta essere uno spreco di risorse piuttosto significativo. Si è quindi preferito avere due sistemi Prometheus separati che fanno il pull degli stessi dati piuttosto che riuscire a mantenerli sincronizzati su dati che man mano che diventano più "vecchi" perdono la loro utilità per il rilevamento di malfunzionamenti.

Job/target and exporter

Sotto questo aspetto Prometheus si apre alle tecnologie white-box portando l'applicazione a farsi monitorare. Ma quando un'applicazione diventa un target ? Quando viene esposto un endpoint per la raccolta delle metriche. Prometheus colleziona le metriche da tutte quelle applicazione all'interno della sua pool di scrape che espongono un endpoint */metrics*. In questo punto è possibile vedere come i concetti di cooperazione tra sviluppatori e gestori di sistema introdotti da DevOps si presentano all'interno dei prodotti.

Prometheus infatti rilascia delle librerie, disponibili per i principali linguaggi di programmazione, che permettono di far esporre all'applicazione metriche che possono venire interpretate da Prometheus. Questo ha permesso l'introduzione del concetto di *exporter*. Un exporter è una parte esterna all'applicazione ma logicamente correlata ad essa che funge da modulo dedicato all'esposizione dell'endpoint */metric*. In questo modo non si va a modificare il codice del servizio mantenendo l'inserimento dell'infrastruttura di monitoring non invasiva. Ogni produttore rilascerà l'exporter per la propria applicazione così da aprirla al monitoraggio. Inoltre tutte le funzionalità di scrittura di un linguaggio SQL-like diventerebbero inutili in questo contesto.

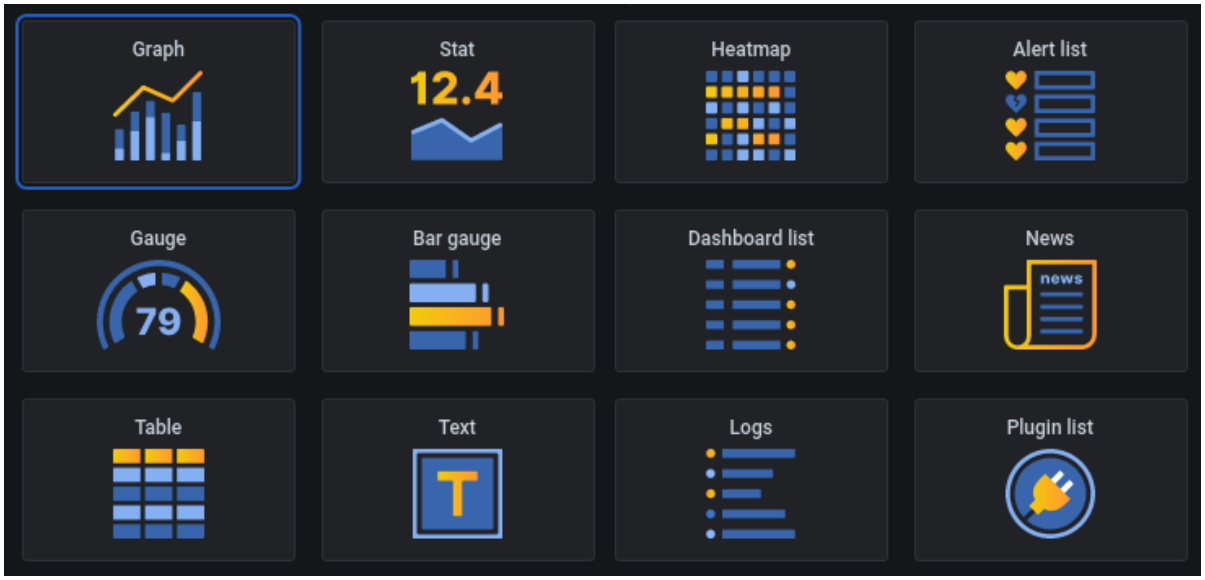

Figura 3.6: Tipologie di Grafico

PromQL query language

Per estrarre delle informazioni dal database di Prometheus è disponibile un linguaggio di query costruito appositamente per ottimizzare questa tipologia di operazioni sul database di Prometheus. PromQL fornisce una sintassi non-SQL by-design. L'utilizzo di query SQL-like risulta particolarmente inefficiente. Le query stesse risultano alquanto intricate nella loro scrittura. PromQl è stato sviluppato solo per eseguire delle letture sul database in modo ottimizzato.

3.6.2 Grafana

Grafana è un tool di visualizzazione grafica che permette di rappresentare comodamente raccolte di dati. Per poter creare una dashboard è necessario avere a disposizione il servizio di Grafana operativo. Nella sua interfaccia sono messi a disposizione dei comodi tool per quanto riguarda la gestione e la creazione delle dashboards stesse. È possibile scegliere il tipo di grafico sul quale si vogliono mostrare i dati(Fig 3.6).

Successivamente si stabilisce una fonte di dati sulla quale verranno effettuate delle letture tramite una query in linguaggio PromQL in modo tale da poter ottenere dei dati con i quali sarà possibile popolare il grafico. La manipolazione matematica dei dati è permessa dalla possibilità di applicare funzioni ai risultati delle query come la media o la somma dei valori ottenuti. È poi possibile sistemare a proprio piacimento i pannelli tramite una comoda interfaccia *drag-n-drop*. Una volta che lo sviluppo della dashboard è stato ultimato è possibile estrarre il file json che descrive la dashboard completa per poi

poter utilizzare tutte le pratiche sul tracciamento del codice tramite l'utilizzo dei DVCS. Inizialmente in Grafana era possibile modificare dashboard o aggiungere sorgenti dei dati solamente se il servizio era già in esecuzione. Dalla v5.0 si è inserito un nuovo sistema di *provisioning attivo* che utilizza i file di configurazione per Grafana prima che questa venga eseguita come servizio, in modo da favorire le più moderne procedure DevOps. Questo approccio permette di ottenere, all'avvio del servizio, dashboards già configurate. Nel file di configurazione vanno specificati dei dati in grado di descrivere la dashboard e le sue caratteristiche. Grafana mette a disponibile una versione containerizzata del suo servizio, quindi si sposa perfettamente con l'ambiente a microservizi su cui questa tesi è basata.

Capitolo 4

Caso di studio

In questa sezione andremo ad analizzare i vari passaggi che sono stati fatti per lo sviluppo del sistema di monitoring del servizio Trauma Tracker. Per prima cosa andremo a descrivere in modo più approfondito il servizio spiegandone il suo funzionamento e le parti principali che lo compongono. Andremo poi ad analizzare i passaggi che sono stati impiegati per dockerizzare la parte del servizio che ci è stata assegnata e come l'architettura di monitoring è stata inserita all'interno del suo funzionamento. L'intento è quello di implementare un servizio di monitoring che sia poco invasivo per l'applicazione e che rispetti i principi che definiscono un ottimo sistema di monitoring definiti nei capitoli precedenti.

4.1 Analisi del servizio Trauma Tracker

Trauma Tracker è un servizio sviluppato per supportare l'accurata e completa documentazione dei processi di rianimazione traumatologica, in particolare ideato per acquisire dati sul percorso assistenziale dei pazienti[11]. TraumaTracker è stato sviluppato come supporto al Trauma Team e, in particolare, al Trauma Leader, il quale ha il compito di annotare ogni evento rilevante che si è verificato durante la gestione del trauma, questa operazione viene salvata in un report. Tradizionalmente il report viene compilato postumo da un medico responsabile su un supporto cartaceo esponendo così la scrittura del report ad un alto rischio di errore umano. Il sistema supporta soccorritori e medici tramite dispositivi smart, dal luogo dell'incidente, passando per il trasporto verso il Pronto Soccorso e durante la gestione del trauma all'interno dell'ospedale.

L'insieme delle tecnologie che sono utilizzate da Trauma Tracker possono essere viste come un unico software integrato per il supporto medico alla gestione dei traumi. È composto principalmente da due differenti applicazioni Android: le prime utilizzate dal capo soccorritore sul luogo dell'incidente e il secondo gestito dal Trauma Leader che gestisce le cure del trauma quando il paziente entra in ospedale. La comunicazione delle

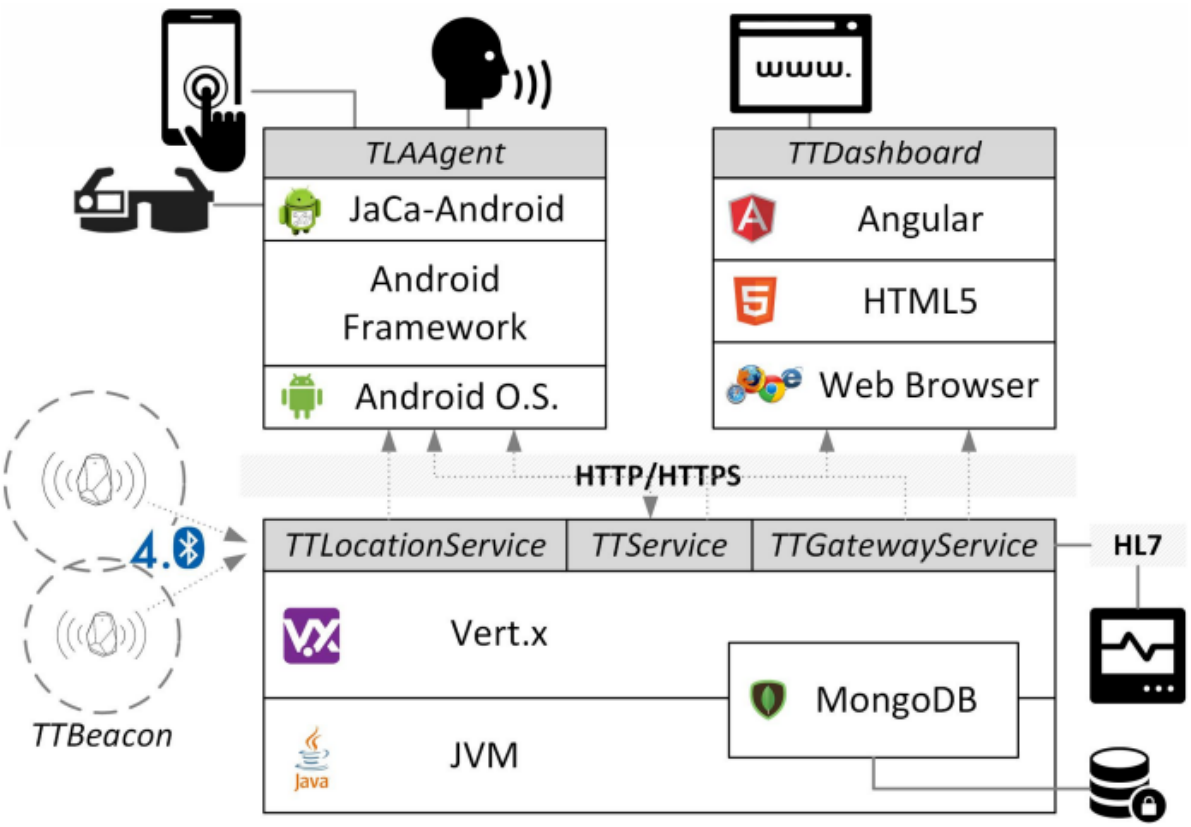

Figura 4.1: Architettura del sistema Trauma Tracker

applicazioni è gestita tramite diversi software ad-hoc basati su vari protocolli: HTTP e Servizi RESTful distribuiti nell'infrastruttura dell'ospedale, tracciamento Bluetooth-LE in stanze dedicate ai traumi che sfruttano un Beacon BLE, mentre nella fase preospedaliera la localizzazione viene gestita utilizzando il segnale GPS.

Architettura generale

Trauma Tracker è suddiviso in 3 parti principali, ognuna di queste utilizza una tecnologia specifica per realizzare al meglio il proprio compito. (Fig. 4.1) [12]

- **Trauma Leader Assistant Agent**: questa parte, basata su tecnologia Android, utilizza strumenti come tablet o smartglass per assistere il Trauma Leader durante la trattazione del paziente e per tutto il suo percorso di riabilitazione.

- **Trauma Tracker Dashboard**: una web app basata su tecnologia Angular che permette di interfacciarsi con il back-end dell'applicazione in modo tale da poter visualizzare statistiche, analizzare i dati e stampare rapporti traumatologici.
- **Trauma Tracker back-end**: costituito da un insieme di microservizi hostati all'interno della rete dell'ospedale che permettono la raccolta e la gestione dei reports.

Il componente che è stato preso in analisi in questa tesi è il Trauma Tracker back-end. Il servizio viene sviluppato in ambiente Java utilizzando la tecnologia Vert.x per implementare le funzionalità web. L'utilizzo di questa tecnologia permette al back-end di operare in modo asincrono. Le varie sottoparti del Trauma Tracker back-end rappresentano ciascuna un microservizio a sé. Queste parti espongono delle API RESTful che utilizzano json come formato per lo scambio di dati. Per la memorizzazione di questi, viene utilizzato un database non relazionale basato su tecnologia MongoDB.

- **Trauma Tracker Report Service**: questo servizio permette di operare direttamente sui report, consente di registrarne di nuovi e di gestire quelli già presenti all'interno del database. Espone degli endpoint HTTP tramite i quali è possibili abilitare le operazioni di monitoraggio.
- **Trauma Tracker Location Service**: consente di tenere traccia del luogo nel quale sta avvenendo la trattazione di uno specifico trauma.
- **Trauma Tracker Gateway Service**: permette la comunicazione con le altre due parti principali che compongono il servizio Trauma Tracker.
- **Trauma Tracker Database**: offre la possibilità di memorizzare i report in modo efficiente. Al suo interno vengono anche memorizzati gli utenti che possono accedere al servizio con i relativi compiti e credenziali.

4.2 Design dell'infrastruttura di monitoraggio

Dopo aver presentato il funzionamento Trauma Tracker vedremo come gli sarà affiancata l'architettura di monitoraggio. L'infrastruttura è costituita da quattro componenti principali: Prometheus, un alert manager, Grafana e gli exporter. Ognuno di questi rappresenta logicamente un microservizio e pertanto verrà lanciato all'interno di un singolo container. Questi quattro componenti funzioneranno all'interno della stessa rete in cui opera Trauma Tracker service, così facendo il servizio sarà raggiungibile dal nostro sistema di monitoraggio. All'interno di questa rete verrà rilasciato un exporter per ogni servizio che si desidera monitorare. Nel nostro caso verrà instrumentato un exporter per il database Mongo e uno per il servizio Trauma Tracker service. Utilizzando questa pratica

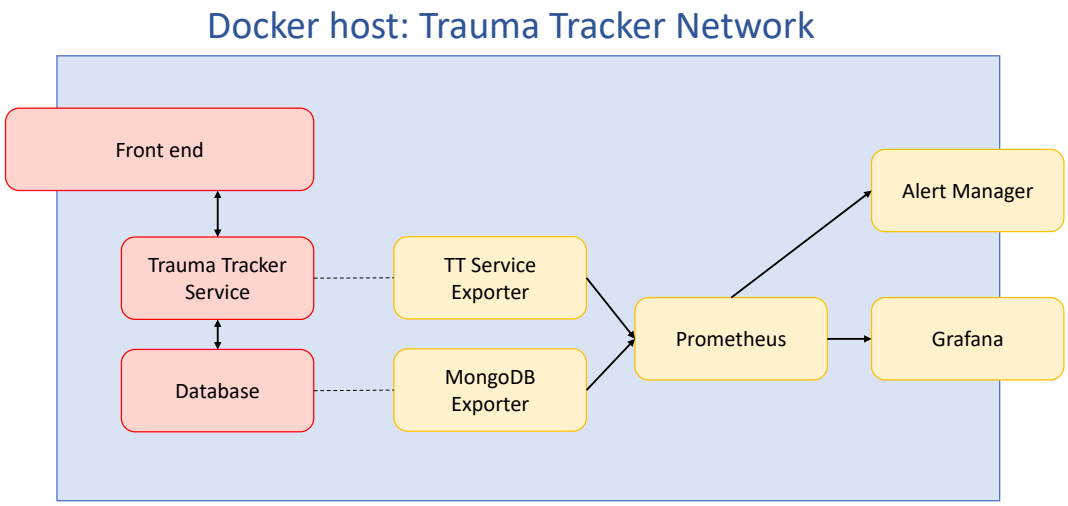

Figura 4.2: Design del sistema di monitoraggio

possiamo trasformare in target i suddetti servizi così da poter aprire la comunicazione delle informazioni di monitoraggio. Per ogni servizio, infatti, va istanziato un exporter specifico nonostante si stiano monitorando due servizi simili, questo poiché si vuole concentrare la logica di controllo su Prometheus e non sugli exporter. Oltretutto in questo modo si ha più controllo sui servizi monitorati in quanto si avrà un feedback diretto dall'exporter collegato ad uno specifico servizio nel caso questo smetta di funzionare.

Prometheus andrà a eseguire degli scrape negli endpoint */metrics* esposti dai singoli exporter passando per il protocollo HTTP. Le informazione raccolte all'interno del suo database verranno utilizzate come fonte di dati per popolare i grafici delle dashboard costruiti in Grafana. Quest'ultimo componente infatti sarà esposto fuori della rete del servizio in modo tale da permetterne il raggiungimento da client esterni. Parallelamente a Grafana sarà istanziato anche un alert manager che si occuperà di leggere i dati all'interno del database di Prometheus e di notificare gli eventuali alert. Anche questo componente sarà esposto; in questo modo gli errori potranno essere notificati e corretti dal personale responsabile. La figura mostra come risulterà l'architettura di Trauma Tracker dopo che sarà installato il sistema di monitoraggio (Fig. 4.2).

```
{
   "scrape route":"/metrics",
   "port":7077,
   "clients":[
      {
         "address":"tt-service",
         "endpoint":"/gt2/traumatracker/api/users",
         "target port":8080,
         "probe intervall millis":5000
      }
   ]
}
```

Figura 4.3: Configruazione del Trauma Tracker exporter

4.3 Design dell'exporter

Un exporter deve funzionare come server per fornire le metriche agli scrape di Prometheus ma allo stesso tempo dovrà chiedere all'applicazione da monitorare dati sul suo funzionamento interno. Al suo interno l'exporter deve comportarsi sia come client nei confronti dell'applicativo da monitorare ma anche come Server nei confronti di Prometheus che richiederà le metriche. Sono disponibili molti modi per l'implementazione di un tale sistema; nel nostro caso la scelta è ricaduta sull'utilizzo del linguaggio *java* e della libreria *Vert.x* per poter utilizzare funzionalità di rete sia nella forma di client che nella forma di server in modo asincrono. Inoltre queste sono le stesse tecnologie instrumentate all'interno di Trauma Tracker stesso, quindi è stato ritenuto opportuno usare il medesimo approccio anche per una questione di coerenza interna in quanto logicamente l'exporter è collegato più all'applicazione che al servizio di Prometheus.

L'exporter può essere definito in 3 classi principali: una in grado di inizializzare il server e il client riuscendo a fornire le giuste configurazioni fornite tramite file, una che permetta di gestire le richieste di Prometheus (server), e una che vada a contattare il servizio che si vuole monitorare (client).

Init class

Quando l'exporter viene lanciato, questo andrà a definire un server e un client in modo tale da poter permettere la formazione e la raccolta dei dati di monitoraggio. Ognuno di questi componenti verrà caricato utilizzando le informazioni presenti all'interno del file di configurazione in formato json (Fig. 4.13). Si è deciso di utilizzare questa metodologia in modo tale da rimanere in linea con quelle offerte dagli altri servizi utilizzati.

```
private void serverSetup() {
   final Router router = Router.router(this.vertx);
   router.route(this.route).handler(new MetricsHandler());
   vertx.createHttpServer().requestHandler(router::accept).listen(this.port);
}
```

Figura 4.4: tt-exporter: Server class

All'interno del file è possibile configurare su quale porta il server dovrà ascoltare le pull request provenienti da Prometheus e in quale endpoint saranno esposte le metriche. Sotto la voce *clients* vengono definiti i servizi che l'exporter dovrà monitorare definendo l'indirizzo e il corrispettivo endpoint da raggiungere. Oltre alle informazioni per la configurazione di rete è possibile definire anche l'intervallo di tempo che intercorre tra ogni chiamata del client. La voce *clients* definisce un array, questo comporta la possibilità di andare a definire più target da monitorare. Ogni client si occuperà di chiamare ciclicamente uno specifico endpoint andando a definire automaticamente delle metriche separate per ogni corrispettivo client. In questo modo è possibile esplorare più endpoint di un servizio così da monitorare il suo funzionamento in modo più completo e, se presenti, poter aggiungere endpoint in grado di fornire risorse contenenti dati specifici di monitoraggio.

Server class

All'interno della classe Server viene creata un'interfaccia di rete utilizzando i valori passati dalla classe di inizializzazione come endpoint e come porta. Questo servizio viene instaurato utilizzando il *Vert.x core* che, grazie ad un apposito handler (sempre fornito dalle librerie), permetterà l'esposizione delle metriche in un formato comprensibile da Prometheus (Fig. 4.4). Vert.x gestisce in modo asincrono tutte le richieste di pull dei dati evitando problemi sull'ordine di deployment dei container.

Client class

La classe client gestisce la formazione delle metriche che definiscono lo stato del servizio Trauma Tracker. Questa contatterà periodicamente il servizio tramite delle chiamate http a specifici endpoint definiti nel file di configurazione. Per gestire gli intervalli in cui si raccolgono le metriche viene dispiegato un *thread*. In questo modo l'exporter continuerà a chiedere informazioni di stato al servizio assegnato in modo periodico (Fig. 4.5). Nel caso si vogliano andare a monitorare più endpoint verrà automaticamente istanziato un thread per ognuno di questi.

```
@Override
public void run() {
   try {
      while (true) {
         client.get(this.port, this.address, this.endpoint).send(response -> {
            //...
         });
         Thread.sleep(this.timeRequestMillis);
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Figura 4.5: tt-exporter: Client class - run()

All'interno di ogni Thread saranno definite le metriche che permettono di soddisfare i *four golden signal* approfonditi nei capitoli precedenti della tesi. Nello specifico si andrà a valutare la latenza del servizio e gli errori di comunicazione. Le informazioni riguardanti il traffico e la saturazione saranno poi formate andando a visualizzare i dati forniti dall'exporter dedicato al monitoraggio del database.

Ogni metrica sarà definita da un nome e da una descrizione che specifichi in modo più chiaro cosa significhi il dato definito al suo interno. Inoltre sarà possibile aggiungere delle label in forma chiave valore, così da avere la possibilità di suddividere le metriche in sottogruppi. Il nome di ogni singola metrica inizia definendo l'applicazione che rappresenta, in questo caso *exporter*, e il tipo di dato che sarà definito al suo interno, ad esempio *http request*. Poiché non possono esistere due metriche con lo stesso nome, in ognuna di queste è inserito anche un numero che identifica il corrispettivo thread che le genera così da non creare conflitti nel caso si vogliano contattare più endpoint. Per distinguere a quale endpoint si riferisce ciascuna metrica, vengono inserite delle label che memorizzano il servizio contattato e l'endpoint di riferimento, per poi essere successivamente inserite all'interno delle singole metriche.

Le librerie che forniscono l'implementazione delle metriche per il linguaggio java utilizzano il pattern builder per definire queste caratteristiche. Ognuna delle operazioni di formattazione sarà implementata tramite l'utilizzo di un metodo apposito. L'uso del metodo build andrà ad eseguire la formattazione delle informazioni inserite. Dopo che le singole metriche sono state costruite e formattate correttamente queste saranno esposte agli scarpe di Prometheus tramite il metodo *register()* (Fig. 4.6). Analizzando la struttura delle librerie possiamo osservare che tutte le classi che definiscono i quattro tipi di metriche, hanno come radice la classe *Collector*. Nella classe Collector è presente il

```
private void buildMetrics() {
   Client.totalRequest = Counter.build()
                        .name("exporter_" + this.threadCounter +
                            "_http_total_request")
                        .help("the number of total requests")
                        .labelNames(this.keyLabel)
                        .register();

   Client.failureRequest = Counter.build()
                          .name("exporter_" + this.threadCounter +
                              "_http_failure_request")
                          .help("the number of requests that returned a failed
                              outcome")
                          .labelNames(this.keyLabel)
                          .register();

   Client.serviceLatency = Gauge.build()
                        .name("exporter_" + this.threadCounter +
                            "_network_service_latency")
                        .help("how long does the service take to satisfy the
                            request")
                        .labelNames(this.keyLabel)
                        .register();
}
```

Figura 4.6: tt-exporter: Client class - buildMetrics()

metodo register() tramite il quale è possibile esporre le metriche agli scrape. All'interno della libreria viene definita una variabile statica che identifica una lista di tutte le metriche di cui il sistema è a conoscenza. Questa viene definita come *defaultRegistry* all'interno della classe *CollectorRegistry*. Il metodo register() permette di inserire la metrica all'interno di questa lista. Quando lo scrape di Prometheus chiederà i dati, verranno passate tutte quelle metriche presenti nel default registry.

4.4 Design della dashboard

La dashboard è stata costruita utilizzando i principi discussi nei capitoli precedenti che consentono di costruire grafici in modo chiaro e non confuso. I grafici servono a rappresentare le metriche raccolte dall'exporter del servizio Trauma Tracker e quelli del servizio di database Mongodb. Nella dashboard vengono visualizzati tutti i dati necessarie

alla comunicazione dei *four golden signal*. Le informazioni vengono suddivise in pannelli che rappresentano una specifica metrica; spesso a queste informazioni possono essere applicate funzioni matematiche per rendere più chiara la visualizzazione dei dati (Fig. 4.7).

La latenza del servizio è ottenuta andando a calcolare la differenza di tempo presente tra l'invio di una richiesta http e la ricezione del messaggio di risposta. La percentuale di raggiungibilità del servizio è calcolata andando a dividere il numero di richieste fallite con il numero di richieste effettuata; in questo modo è possibile controllare gli errori e visualizzare con che percentuale il servizio è risultato attivo per l'utente finale. Le restanti caratteristiche da monitorare sono raccolte utilizzando le metriche fornite dall'exporter di Mongodb. Possiamo osservare che vengono evidenziate tutte le statistiche sulle operazioni che avvengono all'interno del database in modo da poter osservare le operazioni CRUD utilizzate per la gestione dei report. È possibile osservare anche la saturazione del servizio controllando la memoria impiegata e il traffico di network I/O.

L'utilizzo e la visualizzazione di questi dati permette l'instradamento di una forma di controllo sufficientemente completa che sia in grado di verificare il corretto funzionamento del servizio nel tempo offrendo inoltre la possibilità di valutare l'utilizzo di un servizio di hosting esterno. Per lo sviluppo di questa dashboard è possibile estrarre da essa il codice sorgente ed applicare tutte le metodologie di controllo di versione in modo da seguire di pari passo lo sviluppo dell'infrastruttura di monitoraggio.

4.5 Integrazione con le tecnologie di monitoraggio

Promehteus

Dopo aver analizzato in che modo sono stati definiti i singoli componenti andremo a mostrare e spiegare le configurazioni che sono state inserite per far in modo che il sistema funzioni. Primo tra tutti andremo ad analizzare la configurazione di Prometheus (Fig. 4.8)

Prometheus può essere configurato tramite l'inserimento di flag da riga di comando. Questi impostano parametri di sistema immutabili come posizioni di archiviazione, quantità di dati da conservare su disco ecc. Il file di configurazione definisce tutto ciò che riguarda la raccolta dei dati tramite gli scrape e il trattamento dei dati stessi. Poiché stiamo operando in un ambiente containerizzato, andremo a fornire tutte la configurazioni tramite file esterno. Al lancio del container verrà caricata la configurazione di default presente all'interno della cartella */etc/prometheus* del container, tramite le apposite funzionalità offerte da Docker.

Se si vogliono apportare delle modifiche alla configurazione di Prometheus, è possibile aggiornare il file di configurazione in fase di esecuzione, senza dover necessariamente fermare il servizio. La nuova configurazione viene caricata inviando un SIGHUP al

Figura 4.7: Trauma Tracker monitoring dashboard

```
global:
  scrape_interval: 2s

rule_files:
  - "alertmanager.yml"

scrape_configs:
  - job_name: 'service_discovery'
    scrape_interval: 10s
    file_sd_configs:
    - refresh_interval: 10s
      files:
      - /etc/prometheus/service_discovery.yml
```

Figura 4.8: Configruazione di Prometheus

```
groups:
- name: availability
  rules:
  - alert: Excessive_latency_in_the_service_response
    expr: exporter_0_network_service_latency/1000000 > 20
    for: 10s
    labels:
      severity: page

  - alert: The_service_was_more_down_than_up_and_running(percentage)
    expr: exporter_0_http_failure_request_total /
        exporter_0_http_total_request_total * 100 > 20
    for: 10s
    labels:
      severity: page
```

Figura 4.9: Configruazione dell'alert manager

container Prometheus o inviando una richiesta HTTP POST all'endpoint */reload* (quando il flag –web.enable-lifecycle è abilitato). Questo ricaricherà anche tutti i file di regole configurati; se la nuova configurazione non è ben formattata, le modifiche non verranno applicate. Questa operazione forzerà l'aggiornamento anche di tutti i *rule files*.

Il file di configurazione si presenta in formato YAML in tre sezioni principali, ognuna dedicata alla configurazione di una specifica parte di Prometheus. La prima sta ad indicare le impostazioni globali del Prometheus server e indica ogni quanto tempo andranno effettuati gli scrape agli exporter per prelevare le metriche e aggiornare lo stato del database.

Sotto la sezione di alerting è possibile andare a caricare le regole che configurano e gestiscono il lancio degli alert andando ad a inserire soglie e funzioni matematiche che se vengono superate e rispettate queste produrranno il lancio di un alert. Prometheus supporta il caricamento di due tipologie di regole, quelle che regolano come i dati vanno raccolti e quelle che configurano il rilevamento degli alert. Queste vanno definite nel file di configurazione di Prometheus sotto la voce *rule files*. In questo caso è presente solo il set di regole che andrà a regolare la generazione degli alert (Fig. 4.9).

Ogni definizione di alert fa parte di un gruppo di notifiche. Questi alert sono poi identificati da un nome e da una valutazione matematica che, se superata, permette di lanciare una notifica di errore.

Nella configurazione seguente viene esplicitata la possibilità di lanciare due alert, entrambi di gravità *page*: La prima viene generata se la latenza del servizio superi una certa soglia, la seconda viene generata se la percentuale di indisponibilità del servizio

```
- targets: ['tt-exporter:7077']
  labels:
    app:    'monitoring'
    monitor:  'tt-service'
    region: 'internal'

- targets: ['db-exporter:9216']
  labels:
    app:    'monitoring'
    monitor:  'database'
    region: 'internal'
```

Figura 4.10: Configruazione del service discovery

superi il limite del 20%. Ognuno di questi valori deve violare la regola imposta per più di 10s altrimenti non verrà lanciato nessun alert, in questo modo il personale non verrà allertato nel caso si verifichino picchi sporadici di perdita di prestazioni dovuti a fattori che non sono sintomo di alcun problema. Il primo valore permette di notificare la presenza di un problema ai curatori del sistema, l'aumento della latenza è spesso causata da un'eccessiva saturazione del servizio o un aumento del traffico all'interno della rete. Il lancio di questo alert permette infatti di poter controllare lo stato del servizio tramite la dashboard disponibile su Grafana e poter identificare con più chiarezza il problema. Il secondo viene utilizzato più per un monitoraggio a lungo termine poiché permette di capire con che percentuale il servizio non è stato disponibile ed eventualmente poter cercare sistemi di hosting alternativi in gradi di soddisfare la necessità di disponibilità continua del servizio.

Infine è possibile definire come Prometheus acquisisce informazioni per la procedura di service discovery (Fig. 4.10). Sono disponibili svariati metodi, da quelli che utilizzano API messe a disposizione dall'ambiente di deployment, a quelli che necessitano di servizi di terze parti in grado di tener traccia dei servizi. Queste metodologie vengono utilizzate nel caso si impieghi un monitoring di sistemi distribuiti che vada ad impiegare la configurazione della comunicazione dei container su reti separate. In questi casi è necessario utilizzare sistemi di DNS in grado di permettere la raggiungibilità dei servizi da parte di Prometheus. Nel nostro caso l'utilizzo di questi strumenti non è stato necessario, si è adottato un servizio di service discovery basato su file.

Il nome del file viene esplicitato all'interno della configurazione di Prometheus e all'interno contiene tutte le informazione che permettono a Prometheus di contattare tramite scrape i vari exporter. Andremo quindi a ricavare i dati dall'exporter di Mongodb e dall'exporter creato ad hoc per il servizio Trauma Tracker. Così facendo i dati saranno raccolti da entrambi le fonti per poi essere combinati all'interno di Grafana, così da fornire

una visualizzazione ottimale delle informazioni generate. Oltre a definire l'indirizzo per raggiungere ogni singolo exporter è possibile applicare delle label in modo da definire dei sottogruppi logici; in questo modo è possibile tenere ordinata la generazione di metriche nel caso sia necessario orchestrare un maggior numero di exporter.

Grafana

Dalla versione 5.0 di Grafana sono state introdotte delle funzionalità di provisioning così da poter lanciare il container di Grafana con già preconfigurate le eventuali dashbords. Bisogna quindi fornire al container delle configurazioni che indichino come queste sono state costruite e come ottenere i dati per popolarle. In modo simile a Prometheus è possibile eseguire questa operazione sia tramite flag impartiti a linea di comando, sia tramite file di configurazione. Poiché stiamo operando in un ambiente dockerizzato risulta molto più pratico utilizzare un file di configurazione. All'avvio del container, Grafana andrà a prendere queste informazione all'interno della cartella situata in */etc/grafana/-provisioning/*. All'interno di questa cartella si dovranno mettere a disposizione una sotto cartella che definisca le opzioni della dashboard da caricare e un'altra sotto cartella che definisca la fonte di dati da cui attingere per popolare i grafici. All'interno di ogni sotto cartella sarà disponibile un apposito file di configurazione per ogni compito (Fig. 4.12).

Nel file in cui andiamo a configurare la fonte di dati dobbiamo specificare tutte le informazioni di rete necessarie a Grafana per poter raggiungere Prometheus. Grazie al supporto nativo di Grafana questa operazione è particolarmente semplice: è necessario configurare l'indirizzo su cui si trova il server di Prometheus e impostarlo come fonte di dati primaria. Tutte queste impostazioni sono accessibili e modificabili anche dopo che il servizio è stato lanciato utilizzando l'interfaccia grafica fornita dal container di Grafana.

Una volta che abbiamo configurato la fonte dei dati è necessario fornire la dashboard che andrà a visualizzare i dati forniti da Prometheus sullo stato del nostro servizio. Per fare questo andremo a posizionare, nella stessa cartella in cui è collocato il file di configurazione per la dashboard, il codice json che definisce la dashbord stessa. Il caricamento del file contenente la dashboard sarà esplicitato all'interno del file di configurazione sotto la voce *path*. Oltre che fornire il codice da cui è possibile ricostruire i grafici è possibile configurare altri aspetti come la frequenza di aggiornamento dei grafici o la possibilità di apportare delle modifiche.

4.6 Configurazione del deployment

Ogni componente necessario all'orchestrazione dell'intera struttura di monitoraggio è racchiuso all'interno di un container. Ognuno di questi si basa sull'immagine disponibile all'interno del docker hub che viene costruita attraverso l'operazione di building applicata sul Dockerfile da cui deriva il container. Poiché i container come Prometheus, Grafana

```
apiVersion: 1

datasources:
- name: Prometheus
  type: prometheus
  access: proxy
  url: http://prometheus:9090
  isDefault: true
  editable: false
```

Figura 4.11: Configurazione della fonte dei dati

```
apiVersion: 1

providers:
- name: 'Prometheus'
  orgId: 1
  folder: ''
  type: file
  disableDeletion: false
  editable: true
  options:
    path: /etc/grafana/provisioning/dashboards
```

Figura 4.12: Configurazione del caricamento delle dashboards

```
FROM gradle:latest AS builder
COPY --chown=gradle:gradle . /home/gradle/app
WORKDIR /home/gradle/app
RUN gradle shadowJar --no-daemon

FROM adoptopenjdk
EXPOSE 7077
WORKDIR /app
COPY --from=builder /home/gradle/app/build/libs/ .
ENTRYPOINT java -jar *.jar
```

Figura 4.13: Dockerfile del Trauma Tracker exporter

e l'alert manager vengono configurati attraverso un file di configurazione, questi non necessitano una personalizzazione del Dockerfile. Diversamente da questi componenti, il Trauma Tracker exporter è stato sviluppato in modo specifico per dialogare con il Trauma Tracker service. Risulta pertanto necessario andare a creare una versione containerizzata di questa applicazione in modo tale che possa essere inserita all'interno della rete di monitoring. Per fare questo è necessario generare un file di build in modo tale che Docker possa andare a preparare l'ambiente virtualizzato per permettere il corretto funzionamento dell'applicazione. Di seguito andremo ad illustrare tutti i passaggi necessari per la preparazione di tale ambiente; questi sono mostrati in (Fig. 4.13)

Per il deployment di questa applicazione si è deciso di utilizzare un Dockerfile che adotta una multistage build. Così facendo è possibile andare a compilare e risolvere le dipendenze gradle all'interno del Dockerfile senza essere obbligati a generare un Jar prima dell'esecuzione del container. Per prima cosa si andrà a costruire il primo stage della build utilizzando come immagine di base quella di Gradle. All'interno di questa verranno copiati i file necessari per risolvere le dipendenze e generare un jar eseguibile. Dopo aver dato i permessi di esecuzione a gradle si andrà a generare il jar assicurandosi che al completamento dell'operazione il gradle damon si arresti. Una volta completata questa fase si andrà a costruire il secondo stage della build.

Poiché l'applicazione è stata sviluppata in Java, la nostra immagine avrà come base l'immagine della java virtual machine fornita da Oracle e reperibile all'interno di Docker Hub. Una volta pronto il jar eseguibile reso disponibile dallo stage precedente, questo andrà copiato all'interno della cartella definita tramite l'utilizzo del comando WORKDIR; questo comando permetterà di spostare il contesto di esecuzione all'interno della cartella "/app". Tramite il comando ENTRYPOINT sono definiti i comandi che saranno eseguiti con ordine prioritario al momento del lancio del container. Successivamente si andrà a lanciare il jar contenente l'applicativo attraverso l'esecuzione del comando *java -jar ttExporter-fat.jar*.

Ora che abbiamo preparato l'exporter per l'utilizzo in un ambiente containerizzato e abbiamo configurato i componenti per il monitoring in modo corretto possiamo andare ad analizzare come avviene il deployment dell'intera infrastruttura di monitoraggio, in modo da integrare l'exporter che le componenti per l'alerting, la raccolta dei dati e la loro visualizzazione.

Per questa operazione utilizzeremo Docker-compose in modo da poter eseguire il deployment automatizzato attraverso un unico file di configurazione. *Compose* è uno strumento per la definizione e l'esecuzione di applicazioni Docker multi-container. La configurazione di questo tool è effettuata attraverso un file YAML in cui vengono configurati i servizi che dovranno essere containerizzati. Dopo aver preparato il file *docker-compose.yml* è possibile eseguire tutti i container attraverso un solo comando *docker-compose up -d* in cui il parametro "-d" indica la volontà di eseguire tutti i container in modalità *deamon*.

La figura (Fig. 4.14) mostra il file di configurazione di docker compose per l'infrastruttura di monitoraggio da affiancare al servizio Trauma Tracker.

Al suo interno sono gestiti i vari servizi che vanno a costituire l'architettura di monitoring. Questo docker compose dovrà essere eseguito solo dopo l'avvio del servizio di Trauma Tracker in quanto l'exporter dovrà connettersi al Trauma Tracker Service per permettere la comunicazione atta a prelevare i dati di stato.

Il primo servizio che verrà caricato da docker è appunto il Trauma Tracker exporter. L'immagine viene dapprima creata localmente e successivamente il servizio viene lanciato. Se l'immagine non è già presente all'interno del docker registry, questa verrà generata da zero a partire dal dockerfile passato come percorso sotto il parametro *build* definito all'interno del docker-compose. Al momento del caricamento del container, Docker esporrà la porta "7077" in modo tale da poter esporre l'exporter agli scrape di Prometheus.

Poiché il funzionamento di questo container non dipende dal caricamento di altri servizi, verrà inizializzato per primo. L'ordine di caricamento dei container è basato sulla costruzione di un albero di dipendenze che si forma andando ad aggiungere la voce *depends on* all'interno del docker-compose. In questo modo Docker andrà ad inizializzare i container partendo dalle radici dell'albero. In questo caso si parte dall'exporter che va ad affiancarsi al servizio da monitorare in modo da poter subito disporre di una fonte di dati. Successivamente verrà inizializzato Prometheus che andrà a leggere i dati esposti dall'exporter e solamente alla fine verranno dispiegati i container di Grafana e dell'alert manager atti a visualizzare i dati raccolti.

Ora che sappiamo con che ordine i container vengono caricati possiamo procedere andando a spiegare come il container di Prometheus viene configurato. In questo caso il container si basa sull'immagine già buildata di Prometheus presente in Docker hub. Al momento dell'avvio, se questa non è presente all'interno del Docker registry locale, verrà scaricata direttamente dal Docker hub. Dopo che l'immagine sarà pronta andremo ad attaccare un volume condiviso tra il container di Prometheus e la macchina host; così facendo renderemo disponibile all'interno del container i file di configurazione per controllare il comportamento di Prometheus. Questi vengono caricati nella cartella

```
version: "3"

services:
  tt-exporter:
    container_name: tt-exporter
    build: ./tt-exporter
    expose:
      - "7077"

  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus:/etc/prometheus
    ports:          #debug
      - "9090:9090" #debug
    depends_on:
      - tt-exporter

  grafana:
    image: grafana/grafana
    container_name: grafana
    depends_on:
      - prometheus
    volumes:
      - ./grafana:/etc/grafana/provisioning
    ports:
      - "3000:3000"
    depends_on:
      - prometheus

  exporter:
    image: bitnami/mongodb-exporter
    container_name: db-exporter
    environment:
      - MONGODB_URI=mongodb://tt-mongodb:27017
    expose:
      - "9216"


networks:
  default:
    external:
      name: t4c-tt-service_default
```


Figura 4.14: Dockerfile del Trauma Tracker exporter

/etc/prometheus in quanto quella risulta essere la directory di default che il sistema va a leggere per caricare i dati di configurazione. Il comando depends-on andrà a spiegare come deve avvenire la costruzione dell'albero delle dipendenze spiegato sopra.

Il container che identifica Grafana viene configurato in maniera molto simile a Prometheus. Verrà utilizzata l'immagine ufficiale disponibile in Docker hub e verrà inizializzato solamente dopo che il container di Prometheus risulti operativo. Al suo interno verranno mappate le directory del file system della macchina host contenenti i file di configurazione, con le cartelle del file system del container. Quando il container sarà disponibile per l'esecuzione i file saranno letti automaticamente dai rispettivi servizi.

Unica differenza presente può essere trovata nel mapping delle porte. Questo container espone un'interfaccia grafica web, per cui necessitano di poter essere raggiunti da reti esterne. Con l'operazione di mapping delle porte si andranno ad esporre quelle del container, collegandole con quelle del Docker host. Così facendo, quando avverrà una chiamata su una specifica porta mappata all'indirizzo di Docker, questa verrà automaticamente reindirizzata al container con la quale è avvenuta la mappatura. Questa operazione viene effettuata tramite il comando *ports*, andando a specificare a sinistra dei due punti la porta del container e a destra dei due punti la porta del Docker host. In questo caso il container di Grafana mapperà la porta 3000 con la porta 3000 dell'host.

L'ultima voce del Docker compose va a configurare la gestione delle reti. L'insieme dei servizi che vengono caricati e lanciati, infatti, andranno ad operare all'interno di una rete che Docker configurerà in modo automatico utilizzando i driver disponibili. In questo caso, poiché vogliamo che questa infrastruttura si agganci al servizio Trauma Tracker, andremo a fare il deployment dei container all'interno della rete in cui sono definiti tutti i servizi che compongono il Trauma Tracker Service. Quando un container verrà caricato e inizializzato verrà inserito all'interno della rete specificata. In questo caso andremo a definire una rete esterna e già esistente, quella in cui sta funzionando l'infrastruttura di Trauma Tracker. L'assegnazione degli indirizzi ip verrà fatta automaticamente da Docker. All'interno dei file di configurazione non sarà necessario conoscere ogni singolo ip assegnato ai container in quanto andremo ad utilizzare il servizio di DNS integrato all'interno delle reti bridge di Docker.

Conclusioni

A seguito del lavoro svolto emerge che le caratteristiche necessarie per raggiungere le qualità che definiscono un ottimo sistema di monitoraggio sono molteplici e difficili da raggiungere. È necessario possedere una vasta gamma di competenze informatiche che spazino dalla programmazione e configurazione di reti a un'efficiente gestione del software. Caratteristiche ugualmente necessarie per permettere un ottimale sviluppo dell'infrastruttura in grado di evitare rallentamenti nel suo sviluppo e deployment. Le tecnologie discusse nella tesi risultano di fondamentale importanza alla luce di questi aspetti.

Il deployment automatizzato di moduli software che permettono il funzionamento dell'infrastruttura è reso possibile dalla cooperazione di sistemi come Docker-compose e Prometheus. L'utilizzo di queste tecnologie rende il sistema efficace e poco invasivo, caratteristiche ritenute di primaria importanza durante lo sviluppo e la configurazione delle parti del sistema poiché consentono all'architettura di monitoraggio di espandersi parallelamente a Trauma Tracker qualunque sia la direzione che prenderà lo sviluppo del servizio.

Il lavoro effettuato per inserire un ecosistema di monitoraggio a Trauma Tracker, nonostante sia stato portato a termine, rappresenta solamente un punto di partenza per riuscire a soddisfare appieno tutte le caratteristiche di un sistema di monitoraggio adatto ad applicazioni a livello enterprise. Tuttavia la soluzione raggiunta può essere considerata comunque efficace sia da un punto di vista teorico sia da un punto di vista implementativo. I dati raccolti, infatti, permettono di valutare l'andamento del servizio sia sul breve che lungo periodo e offrono la capacità di restringere il campo di ricerca della fonte di un problema.

Il sistema sviluppato su moduli software permette allo stesso tempo di essere ampliato e migliorato andando ad aggiungere altri moduli. Alcuni possibili sviluppi futuri si possono individuare nell'aggiunta di un alert manager che si aggancia ad una piattaforma personalizzata per la comunicazione tempestiva dei malfunzionamenti da comunicare ai curatori del sistema. Nel caso si voglia estendere la rete dei servizi su diversi host fisici è possibile aggiungere un sistema di service discovery basato su DNS così da permettere la raggiungibilità degli exporter dislocati all'interno della rete. Inoltre aggiungere un maggior numero di exporter dedicati offre la possibilità di attingere a una fonte di dati

che rappresenta lo stato del sistema nella sua interezza.

Pertanto, il raffinamento dei valori di monitoraggio ottenuti e l'aggiunta di altri moduli software atti all'ampliamento delle funzionalità costituiscono una prerogativa per futuri aggiornamenti.

Ringraziamenti

Desidero innanzitutto ringraziare i relatori di questa tesi, Prof. Alessandro Ricci e Prof. Angelo Croatti, per la disponibilità e passione con cui svolgono il loro lavoro che sono stato state e saranno per me fonte di ispirazione.

Un grazie di cuore alla mia famiglia, per la grande quantità di supporto avuto sotto ogni punto di vista e senza la quale non avrei mai potuto realizzare questo traguardo.

Ringrazio anche i miei gruppi di amici, colleghi universitari per essere stati vicino a me in questi mesi e per essere passati sopra alle mie assenze. Prometto che mi farò perdonare

Bibliografia

[1] Leonard J. Bass, Ingo M. Weber, and Liming Zhu. *DevOps - A Software Architect's Perspective*. SEI series in software engineering. Addison-Wesley, 2015.

[2] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[3] Betsy Beyer, Niall Richard Murphy, David K Rensin, Kent Kawahara, and Stephen Thorne. *The site reliability workbook: practical ways to implement SRE*. " O'Reilly Media, Inc.", 2018.

[4] M. Brattstrom and P. Morreale. Scalable agentless cloud network monitoring. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 171–176, 2017.

[5] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[6] Docker Inc. *Docker documentation*. Docker Inc., 2013-2020.

[7] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring distributed systems. *ACM Trans. Comput. Syst.*, 5(2):121–150, March 1987.

[8] Anshul Kaushik. Use of open source technologies for enterprise server monitoring using snmp. *International Journal on Computer Science and Engineering*, 2, 10 2010.

[9] M. Mansouri-Samani and M. Sloman. Monitoring distributed systems. *IEEE Network*, 7(6):20–30, 1993.

[10] David McCandless. The beauty of data visualization - david mccandless, 2012.

[11] Sara Montagna, Angelo Croatti, Alessandro Ricci, Vanni Agnoletti, and Vittorio Albarello. Pervasive tracking for time-dependent acute patient flow: A case study in trauma management. In *32nd IEEE International Symposium on Computer-Based*

Medical Systems, CBMS 2019, Cordoba, Spain, June 5-7, 2019, pages 237–240. IEEE, 2019.

[12] Sara Montagna, Angelo Croatti, Alessandro Ricci, Vanni Agnoletti, Vittorio Albarello, and Emiliano Gamberini. Real-time tracking and documentation in trauma management. *Health Informatics J.*, 26(1), 2020.

[13] Prometheus. *Prometheus documentation*. Prometheus, 2014-2021.

[14] Edward Rolf Tufte. *The visual display of quantitative information*. Graphics Press, 1992.