Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

PLUGIN KUBERNETES PER OTTIMIZZARE LA PREEMPTION

Relatore:
Chiar.mo Prof.
Saverio Giallorenzo

Correlatore:
Chiar.mo Prof.
Jacopo Mauro

Presentata da:
Alessandro Neri

II Sessione
Anno Accademico 2023/2024

Sommario

La tecnologia del cloud computing ha rivoluzionato il modo in cui le applicazioni vengono sviluppate e distribuite. Hanno preso piede nuove architetture come quella a microservizi e quella serverless. Per affrontare queste nuove sfide i programmatori e gli amministratori di sistema hanno adottato nuovi strumenti. Uno di essi è Kubernetes, un sistema open-source per l'automatizzazione del deployment, della scalabilità e della gestione delle applicazioni containerizzate. Il componente di Kubernetes su cui si sofferma maggiormente questa trattazione è lo scheduler, il quale si occupa di assegnare le risorse ai servizi. Kubernetes è progettato per essere estensibile e modulare, consentendo agli utenti di personalizzare il sistema per soddisfare le proprie esigenze. Per quanto riguarda lo scheduler, ciò si declina nello Scheduling Framework, una architettura a Plugin che implementa numerose interfacce per l'integrazione di funzionalità. Il presente lavoro di tesi si propone di analizzare nel dettaglio lo Scheduling Framework con la realizzazione di un Plugin per implementare la Cross-Node Preemption tramite l'utilizzo di un solver ottimale.

Indice

Introduzione **1**

1 I microservizi e Kubernetes **3**

1.1 Il Cloud 3

1.2 Microservizi e serverless 7

1.3 Kubernetes 8

1.3.1 Basi di Kubernetes 8

2 Lo scheduler di Kubernetes **12**

2.1 Il Default Scheduler 12

2.1.1 Scheduling Context 13

2.2 Configurazione dello scheduler 14

2.2.1 Extension Point 15

2.2.2 Plugin 22

2.2.3 Configurazione di Plugin ed Extension Point 22

2.3 Scheduling Framework 23

3 Creare un Plugin **25**

3.1 Struttura del repository 25

3.2 Scrittura del codice 27

3.2.1 Interfacce 27

3.2.2 Registrazione 29

3.2.3 Parametri 31

3.2.4 Logging 34

3.3 Configurazione 35

3.4 Compilazione, installazione e testing 38
3.4.1 Makefile 41

4 Ottimizzazione del Default Scheduler 43
4.1 Ottimizzazione dello scheduler 43
4.1.1 Problema del Default Scheduler di Kubernetes 43
4.1.2 Proposta 44
4.2 Cross-Node Preemption 45
4.3 Implementazione del Plugin per l'ottimizzazione dello scheduler . . . 46
4.3.1 Lo script e il solver 46
4.3.2 Plugin 48
4.3.3 Limitazioni del Plugin 54

5 Sintesi e prospettive 56
5.1 Conclusioni 56
5.2 Lavori futuri 57

Introduzione

La grande diffusione del cloud computing, avvenuta attorno al 2010, ha portato a nuovi paradigmi di sviluppo software. Sono state proposte architetture basate sui servizi, che permettono di sviluppare applicazioni modulari e scalabili. L'architettura monolitica, precedente a questa innovazione, prevedeva che un'applicazione fosse sviluppata in un unica base di codice, altamente integrato. L'architettura a microservizi, invece, prevede che l'applicazione sia sviluppata come un insieme di componenti disaccoppiati, ma in comunicazione tra loro. Mentre il primo approccio risulta più semplice da sviluppare e per il deployment, ma poco scalabile e poco resiliente; il secondo porta vantaggi in termini di flessibilità e manutenibilità.

Di contro l'architettura a microservizi ha i sui svantaggi nella complessità dello sviluppo e del deployment. Per aiutare gli sviluppatori sono nati nuovi software. Uno di questi è Kubernetes, un orchestratore di container, la versione Open Source di Borg, un prodotto di Google [16]. Esso prende in carico la gestione delle applicazioni containerizzate, chiamate Pod, e si occupa di distribuirle su un cluster di macchine, detti Nodi. Una visione più dettagliata di Kubernetes sarà fornita in seguito.

Il componente sul quale più ci soffermeremo nel corso della trattazione sarà lo scheduler di Kubernetes. Il concetto classico di scheduler, che troviamo nei sistemi operativi, è quello di un software dedicato ad assegnare risorse ai processi per eseguire le loro attività. Nel caso di Kubernetes, lo scheduler si occupa di assegnare i Pod ai Nodi del cluster, in base a una serie di regole. Proprio come nel sistema operativo lo scheduler di Kubernetes è un componente critico del sistema.

Gli sviluppatori di Kubernetes hanno ideato un'architettura a plugin per agevolare lo sviluppo dei contributori. In questo modo sono mantenute ben separate, per questioni di manutenibilità, il core del sistema e le funzionalità aggiuntive. Inoltre è reso possibile agli utenti scegliere solo il set di plugin di cui necessita tramite la configurazione di un adeguato profilo.

Lo Scheduling Framework, così è chiamata questa architettura, sarà trattata in maggior dettaglio in seguito. In questa tesi ci occuperemo infatti del processo di sviluppo di un plugin per lo Scheduling Framework addentrandoci nelle specifiche di quest'ultimo e analizzandone le potenzialità e criticità. Per fare ciò, questa tesi costruisce su una proposta teorica presentata nella tesi di Simone Canova [3].

Nella sua tesi è proposto un plugin di Cross-Node Preemption con un algoritmo basato un solver per ottenere la soluzione ottimale. Nell'algoritmo si tiene conto delle risorse utilizzate da ogni Pod, la loro priorità e si cerca la soluzione che minimizzi le espulsioni e gli spostamenti dei Pod. Il plugin realizzato in questa tesi è una proof of concept, che mira a dimostrare la fattibilità dell'idea presentata.

1 I microservizi e Kubernetes

Non si può parlare di microservizi senza parlare di Cloud computing. Nel 2009 Netflix si trovò ad affrontare il problema della crescita della domanda dei propri servizi; per riuscire a sostenere il carico di richieste, decise di adottare l'architettura a microservizi. Netflix fu una delle prime aziende di grandi dimensioni ad adottare questa architettura con successo, e da allora molte altre aziende hanno seguito il suo esempio [2]. Una su tutte è Amazon che, ha adottato l'architettura a microservizi, dopo aver creato un servizio di cloud computing, chiamato Amazon Web Services, che è diventato uno dei più grandi al mondo [17].

I concetti di microservizi e di cloud sono fortemente accoppiati tra loro. Per spiegare questo legame daremo in questo capitolo una breve introduzione dell'evoluzione del cloud.

1.1 Il Cloud

Negli anni '70, quando i computer avevano grandi dimensioni ed erano molto costosi quindi solo enti come aziende e università avevano questa possibilità, c'era già qualcuno che sognava i personal computer. Si tratta di Ted Nelson che parla di idee rivoluzionarie su questo e altri argomenti nel suo libro "Computer Lib/Dream Machines" [54]. Una decina di anni dopo nascono i primi personal computer che pervaderanno il mercato. Questa tendenza alla decentralizzazione, negli ultimi anni si sta in un certo senso invertendo: sempre più persone, pur avendo un PC, stanno usando servizi che sono basati su computer remoti.

Questo passaggio deriva dall'incontro di tre fattori. Da una parte gli utenti necessitano di una crescente quantità di risorse da utilizzare al bisogno. Dall'altra la dimensione, sempre più compatta, e la portabilità dei dispositivi rappresentano un limite fisico alla quantità di risorse che possono essere integrate. Infine, la diffusione di connessioni internet veloci e affidabili ha reso maggiormente possibile l'accesso a servizi remoti. A conciliare alla perfezione queste necessità e possibilità è nato il cloud.

Su questa base il cloud è diventato il metodo più diffuso di fornire risorse on-demand. In particolare i servizi che offre sono spazio di archiviazione e potenza di calcolo. Esistono poi varie forme di cloud computing che si differenziano per il livello di controllo. Con questo modello si offrono ulteriori vantaggi agli utenti che possono scegliere quale parte dell'infrastruttura dover gestire e quale invece delegare al fornitore del servizio. I principali modelli di cloud computing sono:

- **IaaS** (Infrastructure as a Service): il fornitore mette a disposizione dell'utente l'infrastruttura hardware, come server, storage e rete.
- **PaaS** (Platform as a Service): il fornitore mette a disposizione dell'utente un ambiente di sviluppo, come un sistema operativo, un database e un web server.
- **SaaS** (Software as a Service): il fornitore mette a disposizione dell'utente un'applicazione già pronta all'uso.

Inoltre il fornitore garantisce una qualità del servizio difficilmente replicabile in self hosting.

A questo punto non resta che sviluppare applicazioni che si adattino all'infrastruttura cloud. Per fare ciò, sono nati negli anni linguaggi di programmazione e tecnologie pensate proprio per la programmazione service-oriented. Un esempio è Jolie, un linguaggio di programmazione pensato proprio per questo scopo, che permette di scrivere applicazioni distribuite in modo semplice e veloce [19]. Un punto cruciale per i servizi basati su tecnologie e linguaggi che non li integrano è lo scambio dei dati che avviene tramite le APIs (Application Programming Interface). Sono nati svariati protocolli e tecnologie per lo scambio di dati tra servizi, come SOAP, REST, GraphQL e gRPC. Anche in questo versante sono stati creati tool per aiutare gli sviluppatori. Fern ne è un esempio, un tool che permette di generare interfacce ti-

pate a partire da una definizione di API, ad esempio sulla base di una specificazione OpenAPI [10].

Un altro passaggio ostico per gli sviluppatori è diventato quello di rendere le proprie applicazioni fruibili secondo i principi del cloud. Distribuire applicazioni service-oriented è diventato maggiormente complicato in quanto è necessario prevedere la compatibilità con vari sistemi e macchine. L'idea è quella di incapsulare l'applicazione all'interno dell'ambiente nel quale viene sviluppata e testata. Per questo motivo sono tornati utili concetti già utilizzati in altri campi come le macchine virtuali e altri nuovi come i container.

Macchine Virtuali

Il primo approccio a una procedura di deploy più generale, che è anche il più obsoleto, è quello di utilizzare una macchina virtuale, o virtual machine, in inglese, da cui l'abbreviazione VM, è la virtualizzazione o emulazione di un computer [56]. In questo modo, si cristallizzano l'architettura e il sistema operativo sul quale viene eseguita l'applicazione e non ce ne si deve preoccupare in fase di deployment.

In un ambiente di questo tipo, lo sviluppatore non deve preoccuparsi di compatibilità e dipendenze relative ad altre macchine. Di contro, si hanno alcuni aspetti negativi legati alle prestazioni: l'emulazione o virtualizzazione dell'intera macchina è un operazione dispendiosa di risorse.

Il componente software che gestisce il funzionamento di una VM è l'hypervisor, che traduce in tempo reale le istruzioni della macchina guest per farle eseguire alla macchina host. Per sua natura, la virtualizzazione è possibile a più livelli del sistema operativo, inoltre anche l'Hypervisor si può collocare in diversi punti del sistema [15, 57]. Per questi motivi, è difficile generalizzare il giudizio sulle prestazioni di questa tecnologia. In generale, le VM sono la scelta ideale per applicazioni legacy, necessità di specifici insiemi di dipendenze e test di software in ambiente protetto. La virtualizzazione riduce, però, le prestazioni del sistema guest, anche se col tempo l'efficienza è migliorata.

Questo concetto di virtualizzazione è nato prima delle esigenze derivanti dal passaggio all'infrastruttura cloud. Le macchine virtuali si adattano bene in un contesto di cloud computing, ma non sono la soluzione ideale per tutti i tipi di applicazioni.

Container

Esiste poi una tecnologia di più recente concezione per il deploy delle applicazioni, che si è evoluta in parallelo al cloud computing. Questa tecnologia è rappresentata dai container, una virtualizzazione a livello di sistema operativo o di applicazione che permette di eseguire in spazi utente isolati le applicazioni [6].

La particolarità dei container è che non virtualizzano l'intero sistema operativo; questo li rende più leggeri e veloci rispetto alle macchine virtuali. La separazione tra i container porta con se il vantaggio di separare gli ambienti di esecuzione e quindi di azzerare problemi legati alla compatibilità delle dipendenze, ma anche di affidabilità e sicurezza.

Negli ultimi anni i container sono particolarmente diffusi e utilizzati in ambito cloud. Molte aziende del settore, come Amazon, Microsoft, Google e IBM, hanno adottato questa tecnologia per i propri servizi cloud [6].

È stata creata una struttura di governance per la standardizzazione di diversi aspetti legati ai container, la Open Container Initiative (OCI) [37]. Legate a questo working group e non esistono diverse alternative sul mercato per la creazione e gestione di container. La più famosa e diffusa è Docker [9]; l'alternativa Open Source è Podman [43]; esiste poi un implementazione ottimizzata per il kernel Linux, che ne sfrutta al massimo le feature, chiamata LXC [28].

I vantaggio nell'utilizzo dei container sono principalmente derivati dalla loro portabilità e leggerezza. Queste caratteristiche li rendono maggiormente ottimizzati in fase di esecuzione rispetto a una VM. Con la loro portabilità poi permette di distribuirli con maggiore facilità e di eseguirli in qualsiasi ambiente che supporti i container.

Un problema dei container può essere la sicurezza: se l'isolamento con il sistema host

non è ben configurato, un container può accedere a risorse del sistema principale, oltre che trasferire le proprie vulnerabilità. Un ulteriore svantaggio è che i container sono stati concepiti per un esecuzioni stateless, ovvero senza memoria persistente. È possibile ovviare a questo problema, ma ciò richiede una configurazione aggiuntiva.

Dal punto di vista dello sviluppo di applicazioni cloud poi, si sono sviluppate nuove architetture che sfruttano le caratteristiche delle tecnologie viste finora.

1.2 Microservizi e serverless

Quello a microservizi è un modello architetturale che organizza un applicazione come un insieme di servizi indipendenti e di piccole dimensioni, che comunicano tra loro tramite protocolli leggeri [32]. Adottare l'architettura a microservizi permette di dividere un'applicazione in parti più piccole e gestibili, che possono essere sviluppate, testate e distribuite in modo singolo. Il processo di separazione in microservizi risulta similare a ciò che avviene in un processo di sviluppo agile, in cui si divide un progetto in task più piccoli e gestibili [31].

Un'applicazione basata su microservizi, per la sua natura può essere eseguita su più macchine, in modo distribuito, utile anche a garantirne la scalabilità. Il cloud è l'ambiente ideale per eseguire applicazioni in questo modo. Per permettere una agevole gestione di un'applicazione basata su microservizi, è necessario un paradigma di distribuzione che permetta di gestire i servizi in modo semplice e veloce. La risposta a questo può essere serverless. Il serverless computing è un modello di cloud computing in cui il fornitore del servizio cloud alloca le risorse della macchina su richiesta, curandosi di gestire i server per i suoi clienti [53].

La richiesta da parte dei microservizi di un ambiente di piccole dimensioni, indipendente ed in grado di comunicare con altri ambienti dello stesso tipo è facilmente assimilabile alla containerizzazione delle applicazioni. Lo stesso si può dire della necessità di allocare risorse su richiesta del serverless. La combinazione di cloud computing, container e paradigmi a microservizi e serverless ha spopolato negli anni recenti ed è diventato un workflow consolidato.

Oltre che un'innovazione per gli sviluppatori, questo metodo di lavoro è stato rivoluzionario anche per gli amministratori di sistema. Questi ultimi hanno dovuto imparare a distribuire e gestire al meglio i container e le risorse ad essi associate. Per svolgere al meglio questo compito sono nati nuovi strumenti anche in questo ambito.

1.3 Kubernetes

Come detto, la gestione dei container è diventata la fase più ostica del lavoro dell'amministratore di sistema. I tool che sono stati pensati per aiutare in questo tipo di problematica sono detti orchestratori. L'orchestrazione è la configurazione automatizzata, il coordinamento, l'implementazione, lo sviluppo e la gestione di sistemi informatici e software [39]. Il software più conosciuto e diffuso per tale scopo è Kubernetes.

1.3.1 Basi di Kubernetes

Per la successiva trattazione è necessario introdurre alcuni concetti basilari di Kubernetes.

Nodi e Pod

Gli oggetti principali con cui lavora Kubernetes sono Nodi e Pod. I primi possono essere sia macchine reali che virtuali e formano un cluster [36]. I Pod invece sono l'unità di computazione più piccola di Kubernetes; rappresentano uno o più container, con memoria e risorse di rete condivise, e una specifica di come eseguire i container [44]. I Pod quindi sono utilizzati per distribuire le applicazioni strutturate a microservizi o serverless.

Per gestire i Pod, gli amministratori di sistema devono configurare Kubernetes attraverso dei file `.yaml`. Tramite questi file si possono descrivere ed ottenere i comportamenti desiderati sul cluster.

Esistono anche componenti più complessi che sfruttano i Pod per creare elementi più articolati. I Deployment gestiscono un insieme di Pod per eseguire il carico di lavoro di un'applicazione [8]. Esistono poi i ReplicaSet che mantengono un insieme di copie di Pod in esecuzione in ogni momento. Vengono spesso utilizzati per garantire la disponibilità di uno specifico numero di Pod identici [46].

I nodi assumono una denominazione particolare a seconda di che compito svolgono. Esiste almeno un Nodo Control Plane e possono esistere molteplici Nodi Worker. Il Control Plane è il Nodo che si occupa di gestire lo stato globale del cluster; un Nodo Control Plane può anche svolgere attività di Worker se avanzano sufficienti risorse. Un Nodo Worker si occupa di mantenere i Pod in esecuzione e di provvedere all'ambiente runtime di Kubernetes [22].

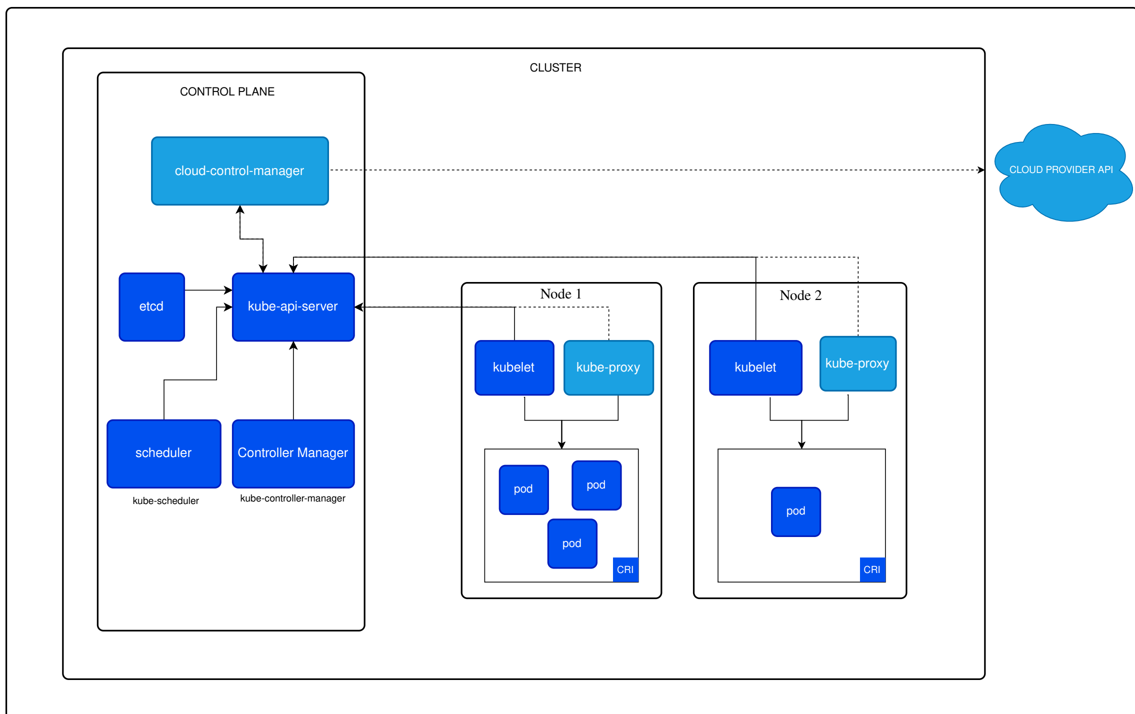

Figura 1.1: Esempio di architettura di un cluster Kubernetes

Per eseguire le proprie funzioni Kubernetes utilizza dei Pod, proprio come quelli che gestisce. Esistono diversi componenti del codice di Kubernetes divisi su più Pod. Per distinguere i Pod di Kubernetes da altri Pod si utilizza un Namespace particolare: `kube-system`.

Namespace

Come già accennato esistono diversi Namespace in Kubernetes. I Namespace forniscono un meccanismo per isolare gruppi di risorse all'interno dello stesso cluster [34]. Inoltre i Namespace forniscono uno scoping per i nomi di alcuni oggetti, che devono essere unici all'interno del Namespace, ma non tra Namespace diversi. Questa feature è sfruttata per garantire l'accesso al cluster a più utenti o gruppi di utenti e mantenere la divisione dei loro ambienti.

Scheduler

Il Pod di sistema di cui ci occuperemo maggiormente nei prossimi capitoli è lo Scheduler. La assegnamento di un Pod ad un determinato Nodo è decisa da Kubernetes attraverso lo Scheduler [25]. Lo Scheduler rimane in attesa della creazione di un nuovo Pod non assegnato ad alcun Nodo; per questi Pod è responsabile di trovare il miglior Nodo su cui farli eseguire [4].

Per come è stato pensato lo Scheduler è possibile sia sostituirlo con uno creato ad hoc, sia estenderne o modificarne le funzionalità tramite un sistema di Plugin. È possibile infatti, tramite i cosiddetti Extension Point, modificare il comportamento dello Scheduler per adattarlo alle proprie esigenze. Più informazioni su ciò saranno fornite nel prossimo capitolo.

Priority

Kubernetes permette di assegnare una priorità, detta Priority, ad ogni Pod che si vuole schedulare. Essa rappresenta l'importanza di un Pod in relazione agli altri. Se un Pod non può essere schedulato, lo Scheduler proverà a fare preemption (espulsione) sui Pod a minore priorità per permettere l'assegnamento del Pod in attesa [41].

La priorità è uno dei meccanismi che permettono agli amministratori di sistema di gestire il comportamento del cluster. Per fare ciò è necessario creare una cosiddetta PriorityClass. Più avanti sarà illustrato come sfruttare questa feature.

Affinity

Un altro metodo per controllare il comportamento del cluster è quello di utilizzare le Affinity.

Per far si che un Pod esegua, o preferisca eseguire, su un particolare Nodo o un insieme di essi si utilizza la Node Affinity. Questo tipo di vincoli si esprimono attraverso un sistema di label. Ogni Nodo ha un insieme di label, alcune sono assegnate da Kubernetes, ma se ne possono anche creare delle personalizzate. Assegnare delle label ad un Pod come Affinity, permette di creare delle regole di associazione che lo Scheduler deve rispettare tra il Pod e il Nodo a cui esso verrà assegnato. Esistono poi configurazioni per rendere le Affinity preferibili e non strettamente vincolanti e altre per prioritizzarle. Tramite i cosiddetti Operator, degli operatori logico-insiemistici, è poi possibile implementare dinamiche più complesse tra le quali il meccanismo delle Anti-Affinity.

Esiste poi la possibilità di creare Inter-Pod Affinity o Anti-Affinity; ovvero la possibilità di vincolare a che Nodo assegnare un Pod sulla base delle label dei Pod già presenti su quel Nodo [1]. Anche per questo tipo di Affinity valgono meccanismi simili a quelli che valevano per la Node Affinity.

In generale la Node Affinity è sfruttata per associare o meno dei Pod a determinati Nodi in base alle loro caratteristiche. Ad esempio un nodo può avere una label che ne descrive il tipo di disco installato, supponiamo `SSD`; e un Pod che ha bisogno di un disco particolarmente veloce avrà quella label come Node Affinity. Viceversa se un Pod farà molti cicli di scrittura sul disco sarà opportuno associargli la label `SSD` come Node Anti-Affinity. Invece l'Inter-Pod Affinity è sfruttata per associare o meno dei Pod ad altri Pod in base alle loro caratteristiche. Ad esempio due Pod possono essere segnati come affini se comunicano molto tra loro e quindi è opportuno collocarli sullo stesso Nodo per ridurre la latenza. Viceversa se due Pod occupano la stessa risorse per un tempo prolungato è opportuno collocarli su Nodi diversi per evitare conflitti.

2 Lo scheduler di Kubernetes

Come già accennato nel capitolo precedente lo scheduler è un componente critico di Kubernetes ed è anche quello sul quale si sofferma maggiormente questa trattazione. Per ciò in questo capitolo indagheremo il funzionamento del Default Scheduler e vedremo come è possibile ampliarlo.

Infatti il Default Scheduler è stato creato in modo da poter essere sostituito da una versione custom per poter eseguire una logica di scheduling differente. Per fare questo non è necessario scriver un intero scheduler da zero, col rischio di introdurre errori, ma è possibile sviluppare plugin da aggiungere al Default Scheduler. In seguito sarà trattata in dettaglio questa architettura a plugin chiamata Scheduling Framework.

2.1 Il Default Scheduler

In Kubernetes lo scheduling ha il compito di assicurarsi che i Pod siano associati ai Nodi di modo che `kubelet`[1] possa eseguirli [25]. Il Pod `kube-scheduler` è il componente di Kubernetes che si occupa di questo compito e viene eseguito sul già citato Control Plane.

Come già accennato, il Default Scheduler rimane in attesa della creazione di nuovi Pod; quando ciò avviene lo scheduler li processa tramite lo Scheduling Context.

[1] `kubelet` è uno dei componenti di Kubernetes; è presente in ogni Nodo e si assicura che i Pod vengano eseguiti correttamente nell'ambiente locale [4].

2.1.1 Scheduling Context

Lo Scheduling Context è diviso in due parti, la prima è detta Scheduling Cycle e la seconda Binding Cycle [49].

Scheduling Cycle

Durante la fase di Scheduling Cycle, lo scheduler trova la soluzione ottimale per inserire il Pod in uno dei Nodi del cluster. Per prendere questa decisione, lo scheduler applica un euristica che si basa sulle risorse richieste dal Pod e sulle risorse disponibili nei Nodi. Per influenzare le scelte dello scheduler poi è possibile utilizzare le tecniche già viste nel capitolo precedente, come Priority e Affinity. Inoltre, Kubernetes applica ottimizzazioni interne al sistema legate alla località dei dati, a vincoli hardware, software e di policy.

I Nodi che vengono ritenuti idonei per l'esecuzione di un Pod sono detti feasible. Se non ve ne sono, il Pod rimarrà in attesa che lo scheduler riesca ad allocarlo nel cluster; questi Pod sono detti unscheduled. Nell'attesa di condizioni più favorevoli, gli unscheduled Pod vengono mantenuti in una coda di attesa. Se invece vengono trovati più nodi per l'esecuzione del Pod, lo scheduler assegnerà un punteggio ai nodi feasible e sceglierà quello con il punteggio più alto per l'esecuzione del Pod.

Nel processo appena descritto, si individuano due fasi principali: la fase di filtering e la fase di scoring. Dalla prima fase esce la lista dei Nodi feasible per l'esecuzione del Pod; se la lista è vuota, il Pod rimane unscheduled. Alla seconda fase, si assegna un punteggio a ciascun nodo e si sceglie il nodo con il punteggio maggiore; se ci sono più Nodi col punteggio più alto, se ne sceglie uno casualmente.

La procedura di Scheduling Cycle appena introdotta viene eseguito in maniera seriale. Quando la decisione è stata presa, lo scheduler passa al processo di binding.

Binding Cycle

Durante la fase di Binding Cycle, lo scheduler applica la decisione presa durante la fase di Scheduling Cycle. Per fare ciò, lo scheduler comunica con l'API Server[2] di Kubernetes per assegnare il Pod al Nodo scelto.

La fase di Binding Cycle può essere eseguita in parallelo.

2.2 Configurazione dello scheduler

Esistono due possibilità per configurare lo scheduler: Scheduling Policies e Scheduling Configuration. Ci soffermeremo in seguito sulla seconda per diverse ragioni; oltre ad offrire maggiore espressività, in particolare per il tipo di modifiche che il progetto di questa tesi si è prefissato di apportare allo scheduler, è la feature più largamente supportata. Infatti, come riportato nella documentazione ufficiale di Kubernetes, le Scheduling Policies sono supportante solo fino alla versione v1.23 [51].

Occupiamoci quindi della Scheduling Configuration. È possibile personalizzare il comportamento di `kube-scheduler` tramite la scrittura di un file di configurazione e passandolo come argomento del comando. Uno Scheduling Profile permette di configurare le diverse fasi di scheduling in `kube-scheduler`. Ogni fase è esposta in un Extension Point. I Plugin forniscono i comportamenti di scheduling, implementando uno o più di questi Extension Points [47].

Di seguito, è riportato l'esempio 1 con una configurazione minimale che può essere applicata allo scheduler col comando `kube-scheduler --config=<path-to-config-file>`. Come si può vedere, in questo file possono essere specificati i profili di scheduling con i rispettivi plugin. Successivamente vedremo nel dettaglio alcuni dettagli di questa configurazione.

[2] API Server è uno dei componenti di Kubernetes che costituisce il Control Plane, in particolare ne è il front end ed espone le API di Kubernetes [4].

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: no-scoring-scheduler
    plugins:
      preScore:
        disabled:
        - name: '*'
      score:
        disabled:
        - name: '*'
```

Listing 1: Esempio di configurazione dello scheduler di Kubernetes.

È possibile specificare più Scheduling Profiles nel file di configurazione; ognuno di essi dovrà avere uno `schedulerName` univoco (come in esempio 1). Ogni Pod, nelle sue specifiche, può indicare quale profilo utilizzare per il proprio scheduling. Se non è presente un profilo nelle informazioni del Pod, verrà utilizzato il profilo `default-scheduler`. Lo Scheduling Profile `default-scheduler` dovrebbe quindi sempre essere presente nel file di configurazione. Se non viene configurato nessun profilo, esso viene generato automaticamente. Infine, i profili creati dall'utente hanno come modello quello di `default-scheduler`; quindi, anche se non esplicitamente specificati, sono attivi tutti i Plugin che sarebbero attivi in quel profilo. Parleremo in seguito dei Default Plugin. Come interpretare gli altri campi presenti nel esempio 1 sarà più chiaro in seguito.

Plugin ed Extension Point sono concetti chiave dello Scheduling Framework di Kubernetes e sono anche le nozioni che abbiamo sfruttato in questa tesi per estendere le funzionalità del Default Scheduler. Per questo motivo, ci addentreremo nel dettaglio di questi concetti.

2.2.1 Extension Point

Il processo già descritto dello Scheduling Context è diviso in fasi, ognuna della quali è rappresentata da un Extension Point. Gli Extension Point non sono altro che interfacce che i Plugin possono implementare per estendere il comportamento dello

scheduler. Ogni Plugin può estendere anche più Extension Point per compiere azioni più complesse.

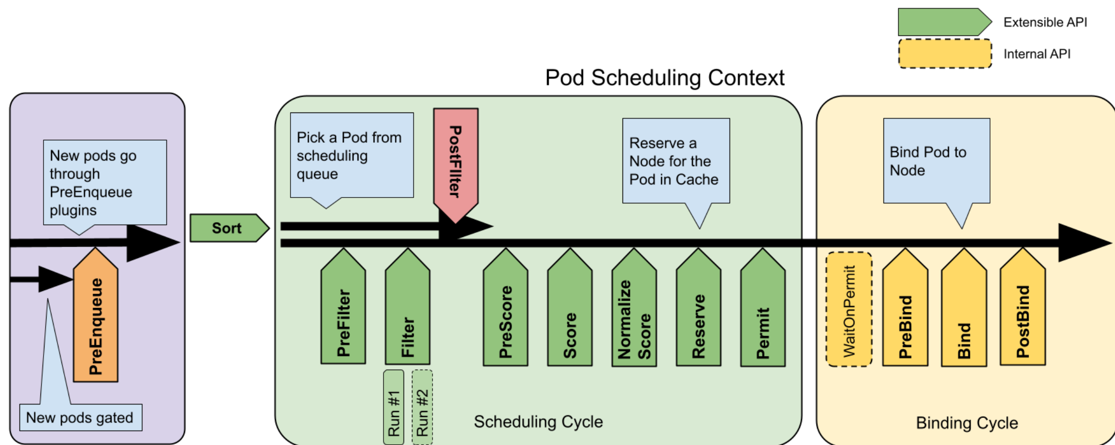

Figura 2.1: Extension Points dello Scheduling Framework di Kubernetes [49].

Come è possibile notare dalla figura 2.1, oltre alle due macro-fasi di Scheduling Cycle e Binding Cycle, ci sono altri Extension Point che possono essere implementati dai Plugin. In particolare, sono presenti delle interfacce per gestire l'ingresso e l'ordinamento dei Pod nella coda di attesa.

Lo scheduler di Kubernetes implementa un sistema di code per gestire i Pod in attesa di essere schedulati. Esso permette allo scheduler di scegliere il Pod più adatto per il prossimo Scheduling Context. Ovvero quello con più possibilità di essere assegnato ad un Nodo del cluster. Un Pod, potrebbe dover rispettare o richiedere diverse condizioni al momento dello scheduling. Questo meccanismo è chiamato a posporre lo scheduling finché non è probabile che il cluster sia in grado di soddisfare i requisiti del Pod [52].

Questo sistema è basato su tre code:

- **Active Queue**: contiene i Pod che sono pronti per lo scheduling.
- **Unschedulable Queue**: contiene i Pod che sono in attesa di certe condizioni prima di poter passare allo scheduling.
- **Backoff Queue**: contiene i Pod che hanno fallito la fase di scheduling, ma che si prevede possano essere schedulati in futuro.

Inoltre, il sistema di code è provvisto di alcune routine, che eseguono in background, responsabili di spostare i Pod alla Active Queue quando sono pronti per lo scheduling. Alcuni di essi sono eseguiti periodicamente, altri sono attivati da eventi specifici.

Trattiamo ora nel dettaglio gli Extension Point presenti nello Scheduling Framework di Kubernetes [49].

PreEnqueue

Questo Extension Point è chiamato prima che un Pod venga inserito nel sistema di code. Se tutti i PreEnqueue Plugin ritornano `Success`, al Pod è consentito di entrare nella Active Queue. Altrimenti il Pod viene inserito nella Unschedulable Queue.

EnqueueExtension

Questa interfaccia viene utilizzata dai Plugin che implementano gli Extension Point PreEnqueue, PreFilter, Filter, Reserve e Permit. Essa permette di controllare se un Pod respinto dal Plugin che la implementa, possa nuovamente riprovare lo scheduling o meno. Questa decisione è presa sulla base di cambiamenti del cluster.

QueueingHint

Si tratta di una callback per decidere se reinserire un Pod nella Active Queue o nella Backoff Queue. Essa è eseguita ogni qualvolta un certo tipo di eventi o cambiamenti si verifica nel cluster. Se QueueingHint scopre che l'evento che si è verificato potrebbe rendere il Pod schedulable, allora il Pod viene spostato nella coda opportuna di modo che lo scheduler possa tentare nuovamente lo scheduling.

Questo Extension Point era di livello beta nella versione di Kubernetes v1.28, quanto si è scoperto che potrebbe causare problemi di prestazioni. Nella più recente versione di Kubernetes v1.31, è disattivato di default; se necessario si deve abilitarlo esplicitamente.

QueueSort

Questa interfaccia permette di ordinare i Pod nella coda di attesa. Si tratta di una funzione `Less(Pod1, Pod2)` che confronta i due Pod fornitigli. È possibile abilitare uno solo di questi Plugin alla volta.

PreFilter

Questo Extension Point è il primo all'interno dello Scheduling Context; in particolare è il primo dello Scheduling Cycle.

Le interfacce di PreFilter sono utilizzate per pre-processare informazioni sul Pod e sul cluster. Se un PreFilter Plugin ritorna un errore, lo Scheduling Cycle viene abortito.

Questo è uno degli Extension Point che è stato utilizzato per implementare il Plugin di questa tesi. In particolare, vedremo che l'utilizzo che ne viene fatto è quello di includere nella prossima fase di scheduling certi Nodi.

Filter

In questa interfaccia sono filtrati i Nodi che non sono idonei per l'esecuzione del Pod. Per ogni Scheduling Cycle, lo scheduler chiama tutti i Filter Plugin nell'ordine specificato nel file di configurazione. Se un Plugin segna un Nodo come unfeasible per il Pod relativo a quel ciclo di scheduling, il Nodo non verrà valutato dai Plugin successivi.

Le valutazioni fatte dai Filter Plugin sono relative alla coppia Nodo-Pod. In uno Scheduling Cycle, viene preso in considerazione un singolo Pod, ma in questa fase sono presi in considerazione tutti i Nodi. Quindi, fissato un Pod, si itera su tutti i Nodi. Per ragioni di efficienza, le valutazioni di Filter sulle accoppiate Nodo-Pod possono essere lanciate in parallelo.

PostFilter

Questa interfaccia è chiamata dopo quella di Filter, ma solo se non sono stati trovati Nodi feasible per il Pod in esame. Anche i Plugin di questo Extension Point sono eseguiti nell'ordine configurato; però, se uno di essi segna il Pod come `Schedulable`, gli altri non verranno eseguiti.

Una tipica implementazione di PostFilter è "preemption", che prova a rendere il Pod schedulable espellendo altri Pod. Tratteremo più in dettaglio nei prossimi capitoli il discorso della preemption. Infatti anche questo è un Extension Point utilizzato nell'implementazione proposta da questa tesi.

PreScore

In questa fase, si genera uno stato condiviso utilizzabile dai Plugin della fase successiva. Se un PreScore Plugin ritorna un errore, lo Scheduling Cycle viene abortito.

Score

In questo Extension Point vengono ordinati i Nodi che hanno passano la fase di Filter. Lo scheduler esegue tutti gli Scoring Plugin sulla lista dei Nodi. Dopo aver eseguito la normalizzazione dei risultati, lo scheduler combinerà i dati ottenuti, tenendo conto del peso attribuito al Plugin nelle configurazioni.

NormalizeScore

Questa interfaccia permette di modificare i punteggi assegnati dagli Score Plugin prima che lo scheduler li utilizzi per prendere una decisione. I NormalizeScore Plugin vengono eseguiti sugli output della fase di Score del Plugin stesso. Questa è un altra interfaccia critica, che in caso di errori può causare l'aborto dello Scheduling Cycle.

L'idea di questo Extension Point è quella di, dopo aver dato uno punteggio ad ogni Nodo, normalizzare i valori ottenuti rispetto al range di valori ammissibili. Può anche essere impropriamente utilizzata, come suggerito nella documentazione, come

fase di “PreReserve”, ovvero preparazione alla fase successiva.

Reserve

Questo Extension Point è costituito in realtà da due metodi: `Reserve` e `Unreserve`. Queste funzioni sono utilizzate per aggiornare lo stato dei Plugin che ne mantengono uno, anche detti Stateful Plugin. In particolare, questi ultimi possono utilizzare queste interfacce per essere notificati dallo scheduler che determinate risorse del cluster sono state riservate o liberate.

Il metodo `Reserve` viene chiamato prima che avvenga l’effettivo binding di un Pod a un Nodo per evitare race condition. Ci si aspetta dai Plugin di questa fase una risposta di successo o fallimento. Se un Plugin ritorna un errore in questa interfaccia, i successivi Plugin non vengono eseguiti e lo stato di questa fase sarà di fallimento. Se tutti i Reserve Plugin ritornano con successo, questa fase è considerata completata correttamente ed il resto dello Scheduling Context può procedere.

L’interfaccia Unreserve è pensata per eseguire il ripristino delle informazioni degli Stateful Plugin a una condizione precedente in caso di fallimento del binding di un Pod a un Nodo. Infatti questo Extension Point viene eseguito se Reserve, o un altra fase successiva, fallisce. Quando ciò avviene i metodi `Unreserve` sono tutti eseguiti nell’ordine inverso a quello nel quale sono configurati i Reserve Plugin.

Permit

Al termine dello Scheduling Cycle, vi è la fase Permit che decide se permettere o posticipare il binding del Pod al Nodo designato. Sono previste le seguenti azioni:

- **approve**: il Pod viene direttamente passato al Binding Cycle.
- **deny**: il Pod viene rimandato alla coda di attesa; si innesca così anche la fase di Unreserve.
- **wait**: il Pod viene salvato in una coda interna, il suo Binding Cycle inizia, ma viene subito stoppato fino a quando il Pod non viene approvato. Se passa troppo tempo senza che il Pod venga approvato, esso viene considerato deny.

PreBind

La prima fase del Binding Cycle è PreBind. In questo Extension Point i Plugin possono eseguire sul Nodo azioni necessarie prima che su di esso venga eseguito il Pod. Problemi con questo Plugin causano l'aborto del Binding Cycle.

Bind

Questa fase non può cominciare prima che tutti i Plugin della precedente non abbiano completato con successo. I Bind Plugin sono chiamati nell'ordine in cui sono configurati e ognuno di essi può decidere se gestire o meno il binding del Pod. Se un Plugin si prende in carico il Pod, i successivi non verranno eseguiti.

PostBind

Questo Extension Point è informativo, viene eseguito quando il binding è terminato correttamente e chiude il Binding Cycle e l'intero Scheduling Context. Può essere utilizzato per rilasciare risorse o per aggiornare lo stato del cluster.

CycleState

Il `CycleState` è un oggetto che viene passato a tutti[3] gli Extension Point; esso fornisce le API per accedere alle informazioni dello Scheduler Context corrente. Inoltre, è presente anche un meccanismo per gestire la comunicazione tra Extension Point di uno stesso Plugin che persiste, però, solo per la durata di un singolo Scheduling Context.

FrameworkHandle

L'oggetto `FrameworkHandle` è invece fornito per procurare le API utili all'intero ciclo di vita del Plugin. Tra le altre cose, questo oggetto permette di accedere alla cache dello Scheduler che mantiene informazioni generali sullo stato del cluster.

[3]Viene passato a tutti i Plugin meno quelli esterni allo Scheduling Context, ovvero EnqueueExtension, QueueingHint e QueueSort.

2.2.2 Plugin

Ogni plugin implementa uno o più degli Extension Point visti in precedenza con l'obiettivo di implementare una logica di scheduling ben definita. Esistono molti plugin già implementati in Kubernetes [47], in quanto lo Scheduling Framework prevede che anche il comportamento predefinito dello scheduler sia implementato con una serie di Plugin. Lo scheduler profile `default-scheduler` ha alcuni di questi Plugin abilitati e altri disabilitati.

È possibile scrivere nuovi Plugin per estendere le funzionalità dello scheduler; ne sono già disponibili dieci creati dalla community di Kubernetes [26]. L'obiettivo della tesi è proprio quello di crearne uno; vedremo infatti nel prossimo capitolo come è possibile svilupparlo, testarlo ed utilizzarlo.

2.2.3 Configurazione di Plugin ed Extension Point

Come già mostrato nel frammento di codice 1, è possibile configurare i Plugin e gli Extension Point dello Scheduler Profile tramite un file di configurazione. Per la precisione, questa procedura prevede che si indichino quali Plugin abilitare e quali disabilitare per un determinato Extension Point. Per fare ciò si utilizzano i nomi degli Extension Point, che in fase di sviluppo si scrivono in pascal case, in camel case.

È possibile utilizzare `*`, un asterisco, per indicare che tutti i Plugin per un certo Extension Point devono essere abilitati o disabilitati. Al contrario, è possibile anche abilitare o disabilitare tutti gli Extension Point per un certo Plugin tramite la keyword `multiPoint` da utilizzare al posto del nome dell'Extension Point. Questa feature è utile per ridurre la quantità di configurazione necessaria e, in fase di sviluppo, per evitare di dover modificare il file di configurazione ogni volta che si aggiunge una nuova interfaccia al Plugin.

È inoltre possibile specificare un peso per ogni Plugin. Questo è utilizzato per ordinare i Plugin all'interno di un Extension Point. Come già detto, l'ordine dei plugin può essere importante per ottenere il comportamento desiderato dallo scheduler. Se

non viene specificato, viene assegnato il peso 1.

Infine, esiste anche la possibilità di passare degli argomenti ai Plugin. Essi vengono inseriti sempre nel file di configurazione nella sezione `pluginConfig` relativa ad uno specifico profilo. Vedremo nel prossimo capitolo come è possibile implementare questa feature.

2.3 Scheduling Framework

Lo Scheduling Framework è un insieme di API per Plugin aggiunte allo scheduler di Kubernetes già esistente. I Plugin sono compilati direttamente nello scheduler e queste API permettono a molte feature dello scheduler di essere implementate come Plugin, mentre si mantiene il nucleo dello scheduler semplice e mantenibile [50]. Le interfacce trattate nelle sezioni precedenti sono il frutto di questo design.

Ci sono diverse ragioni per cui i designer di Kubernetes hanno deciso di implementare lo scheduler in questo modo. Innanzitutto, l'aumento delle feature dello scheduler ha comportato un aumento della complessità del codice e delle logiche. Ciò rendeva difficile la manutenzione, la scoperta di bug e la successiva correzione. Inoltre, in precedenza, le API per personalizzare lo scheduler erano limitate:

- pochi Extension Point disponibili: esistevano solo "Filter", "Prioritize", "Preempt" e "Bind" che ricalcano e aggregano alcune delle interfacce viste in precedenza.
- i Plugin non erano integrati nel codice, per essere richiamati venivano eseguiti webhook, ovvero richieste HTTP, che rallentavano il processo di scheduling.
- anche l'aborto dello scheduling di un Pod era difficile da gestire perché era necessario comunicarlo ai Plugin tramite chiamate HTTP.
- essendo i Plugin esterni, non potevano accedere alle stesse informazioni dello scheduler. Quindi dovevano essere inserite in cache dalle chiamate precedenti, rendendo lo sviluppo più complesso.

Queste criticità ostacolavano la creazione di uno scheduler che fosse contemporaneamente versatile e ad alte prestazioni.

I designer hanno quindi cercato di ottenere un meccanismo di estensione veloce abbastanza da permettesse la trasformazione delle funzionalità esistenti dello scheduler in Plugin. È questo il motivo per cui esistono dei Plugin predefiniti per le funzionalità di base del Default Scheduler. Essi sono Plugin alla stregua di quelli sviluppati dagli utenti per personalizzare lo scheduler.

Pro

I vantaggi di questo design sono:

- rendere lo scheduler più flessibile e personalizzabile;
- rendere il nucleo dello scheduler più semplice spostando alcune feature nei Plugin;
- aggiungere Extension Point;
- rendere l'aborto di uno Scheduling Cycle più semplice;
- aggiungere un meccanismo di gestione degli errori condiviso coi Plugin.

Contro

Esistono anche alcuni svantaggi:

- limitazioni legate al nuovo design;
- fornire API con argomenti e valori di ritorno completi per ogni possibile scenario.

3 Creare un Plugin

Vedremo ora come sviluppare un Plugin per lo scheduler di Kubernetes. Si noti che uno dei contributi di questa tesi è la trattazione esaustiva dello sviluppo di un Plugin per lo scheduler di Kubernetes in quanto i passaggi per fare ciò non sono riportati in modo esaustivo nella documentazione ufficiale di Kubernetes [48]. In particolare non sono documentate le API di Kubernetes utili per implementare le logiche di scheduling. Inoltre, come è possibile osservare dalla Git history del file, o più comodamente dalla modalità di visualizzazione "Blame" di GitHub; la documentazione è alquanto datata e non ha subito modifiche recenti, perciò, non sempre corrisponde alla attuale implementazione del codice. Questo è il motivo per cui si farà riferimento ad alcuni articoli e guide online che trattano l'argomento [45, 40] oltre che al codice dello scheduler stesso e dei Plugin già implementati dalla community [26].

3.1 Struttura del repository

Per creare un Plugin custom, anche detto out-of-tree, è necessario clonare con `git` il repository GitHub scheduler-plugins [26]. Essa contiene l'ambiente di sviluppo per creare Plugin custom, ma anche il codice dello scheduler stesso in quanto, come visto in precedenza, i Plugin vengono compilati direttamente all'interno dello scheduler. Vediamo ora una breve descrizione della struttura della cartella; saranno riportati solo i file e le cartelle più rilevanti per lo sviluppo di un Plugin custom.

```
scheduler-plugins/
├── apis/
├── build/
├── cmd/
├── doc/
├── hack/
├── manifests/
├── pkg/
├── vendor/
└── Makefile
```

Listing 2: Struttura del repository scheduler-plugins.

`apis/` contiene la definizione dei tipi di dato e il codice per gestirle.

`build/` contiene i `Dockerfile` per i container di scheduler e controller.

`cmd/` contiene il codice che invoca scheduler e controller.

`doc/` contiene la documentazione in file `.md` su come creare un Plugin ed installarlo.

`hack/` contiene principalmente script `.sh` che eseguono procedure di testing, compilazione, installazione, aggiornamento e verifica del codice.

`manifests/` contiene i file di configurazione dei Plugin, come quello visto nell'esempio 1.

`pkg/` contiene il codice di tutti i Plugin.

`vendor/` contiene il codice dello scheduler e di tutti i suoi componenti.

`Makefile` è il file utile alla compilazione ed installazione dello scheduler contenente i custom Plugin. Principalmente si limita a richiamare script presenti nella cartella `hack/`.

3.2 Scrittura del codice

Come spiegato nella proposta dello Scheduler Framework, i Plugin sono compilati assieme allo scheduler stesso. Ciò vuol dire che per sviluppare un Plugin è necessario utilizzare lo stesso linguaggio di programmazione utilizzato per lo scheduler: `Go` [55].

`Go` è un linguaggio di programmazione sviluppato da Google a partire dal 2007 [13]. Questo linguaggio è sintatticamente simile al `C`, ma propone caratteristiche che lo avvicinano anche ad altri linguaggi più moderni come la memory safety, lo structural typing e primitive per la gestione della concorrenza. `Go` è stato progettato per aumentare la produttività in ambienti multicore, di networking e in grandi codebase.

Il codice di un Plugin, come visto nella struttura del progetto 2, è contenuto nella cartella `pkg/`. Al suo interno, ogni Plugin crea la sua directory. Il Plugin può essere sviluppato su più file, ma è necessario che gli Extension Point siano definiti in un unico oggetto.

Vedremo nelle prossime sezioni l'implementazione di un Plugin di esempio che chiameremo `MyPlugin`.

3.2.1 Interfacce

Gli Extension Point dello scheduler sono definiti come interfacce `Go`. Queste ultime sono un tipo di dato che definisce un insieme di metodi. Per implementare un'interfaccia è necessario implementare tutti i metodi definiti da essa.

Esiste un'interfaccia principale chiamata `Plugin`, essa definisce il metodo `Name` che restituisce il nome del Plugin. Le interfacce che definiscono gli Extension Point sono invece più articolate: ognuna richiede l'implementazione di uno o più metodi. Le interfacce dei singoli Extension Point richiedono, tramite la feature interface embedding, di implementare l'interfaccia `Plugin`, come si può vedere nel codice 3 alle righe 9 e 17.

Per brevità, nel codice 3 sono contenute poche interfacce e solo esse sono commen-

tate. Nel codice sorgente si possono trovare tutte quelle che corrispondono agli Extension Point e sono commentati anche i singoli metodi. Questi commenti rendono più agevole lo sviluppo.

```
1  // Plugin is the parent type for all the scheduling framework plugins.
2  type Plugin interface {
3      Name() string
4  }
5  ...
6  // PreFilterPlugin is an interface that must be implemented by "PreFilter" plugins.
7  // These plugins are called at the beginning of the scheduling cycle.
8  type PreFilterPlugin interface {
9      Plugin
10     PreFilter(ctx context.Context, state *CycleState, p *v1.Pod) (*PreFilterResult,
       ↪  *Status)
11     PreFilterExtensions() PreFilterExtensions
12 }
13 ...
14 // PostFilterPlugin is an interface for "PostFilter" plugins. These plugins are
   ↪  called
15 // after a pod cannot be scheduled.
16 type PostFilterPlugin interface {
17     Plugin
18     PostFilter(ctx context.Context, state *CycleState, pod *v1.Pod,
       ↪  filteredNodeStatusMap NodeToStatusMap) (*PostFilterResult, *Status)
19 }
```

Listing 3: Alcune interfacce degli Extension Point dello scheduler dal file `vendor/k8s.io/kubernetes/pkg/scheduler/framework/interface.go`.

Un Plugin ha la struttura mostrata nel codice 4. In essa, si implementa l'interfaccia `Plugin` tramite il metodo `Name` che restituisce il nome del Plugin. Il metodo è un "getter" che viene utilizzato per una distinzione tra i Plugin interna allo scheduler. Coi metodi `PreFilter` e `PreFilterExtensions`, invece, si implementa l'interfaccia `PreFilterPlugin`. All'interno di queste funzioni possono essere implementate le logiche del Plugin.

La `struct` che rappresenta il Plugin può contenere un numero arbitrario di campi utili a mantenere uno stato. Inoltre, nel codice 4 si può vedere un escamotage spesso utilizzato, quello di assegnare un oggetto di tipo `MyPlugin` ad una variabile di tipo `Plugin` ed a una di tipo `PreFilterPlugin` per verificare tramite il controllore dei

tipi che non ci siano errori nell'implementazione di queste due interfacce, invece di far emergere questi errori in fase di compilazione.

```go
package myplugin

import (
    "context"
    v1 "k8s.io/api/core/v1"
    "k8s.io/kubernetes/pkg/scheduler/framework"
)

const Name = "MyPlugin"

type MyPlugin struct {
    arg int64
}

var _ framework.Plugin = &MyPlugin{}
var _ framework.PreFilterPlugin = &MyPlugin{}

func (_ *MyPlugin) Name() string {
    return Name
}

func (_ *MyPlugin) PreFilter(_ context.Context, _ *framework.CycleState, _ *v1.Pod)
↪ (*framework.PreFilterResult, *framework.Status) {
    return nil, nil
}

func (_ *MyPlugin) PreFilterExtensions() framework.PreFilterExtensions {
    return nil
}
```

Listing 4: Implementazione di un Plugin.

3.2.2 Registrazione

Per essere compilato nello scheduler, il Plugin deve essere registrato tra i quelli disponibili. Per fare ciò, è necessario che nel file `cmd/scheduler/main.go` si crei un oggetto del tipo `MyPlugin` associato al suo nome. Si deve quindi creare una funzione del tipo `PluginFactory`, come definito nel codice 5. È consuetudine chiamare questa funzione `New`. Si può apprezzare in questa implementazione l'applicazione del factory

design pattern, come suggerisce il nome dato al tipo.

```
1 // PluginFactory is a function that builds a plugin.
2 type PluginFactory = func(ctx context.Context, configuration runtime.Object, f
  ↪  framework.Handle) (framework.Plugin, error)
```

Listing 5: Definizione del tipo `PluginFactory` dal file `vendor/k8s.io/kubernetes/pkg/scheduler/framework/runtime/registry.go`.

La funzione `New` per il Plugin di esempio `MyPlugin` risulta come mostrato nel codice 6. Se il Plugin ha uno stato, in questa funzione è possibile inizializzarlo anche con riferimenti ad oggetti dello Scheduling Context e Framework Handle che vengono passati come parametri. Sarà analizzato in seguito l'utilizzo del parametro di tipo `Object`. Il seguente codice necessita di importare il pacchetto `k8s.io/apimachinery/pkg/runtime` per funzionare, oltre quelli già importati nel codice 4.

```
1 func New(_ context.Context, _ runtime.Object, _ framework.Handle) (framework.Plugin,
  ↪  error) {
2     return &MyPlugin{}, nil
3 }
```

Listing 6: Definizione della funzione `New` per `MyPlugin`.

La registrazione è quindi effettuata dal codice 7. Quello riportato, per brevità, mostra solo la registrazione di `MyPlugin`, ma nel file sorgente sono presenti anche quelle di tutti gli altri Plugin. In genere, non è necessario eliminare da questo file le registrazioni dei Plugin non utilizzati, in quanto essi sono attivati o meno dai file di configurazione. Vedremo nel dettaglio come attivare un Plugin nelle prossime sezioni.

```go
func main() {
	command := app.NewSchedulerCommand(
		// Si noti che `myplugin` fa riferimento al package nel quale sono
		// contenute le funzioni, che deve essere opportunamente importato,
		// e non al plugin che sarebbe scritto `MyPlugin`
		app.WithPlugin(myplugin.Name, myplugin.New),
	)
	code := cli.Run(command)
	os.Exit(code)
}
```

Listing 7: Registrazione nello scheduler di `MyPlugin` nel file `cmd/scheduler/main.go`.

3.2.3 Parametri

Un Plugin può anche accettare parametri. Essi saranno passati al Plugin tramite il file di configurazione, come quello dell'esempio 1.

La procedura per sviluppare questa funzionalità non è ben spiegata nella documentazione di Kubernetes; è però presente in un articolo online [45]. Alcuni passaggi sono considerati deprecati, nonostante siano tra i pochi citati nella documentazione ufficiale [48]. In assenza di un metodo alternativo altrettanto documentato, sono comunque utilizzabili, con le dovute accortezze.

La procedura consiste nel creare alcune strutture dati, impostare un valore di default per i parametri ed eseguire degli script. Le strutture dati serviranno al parsing degli argomenti che verranno passati in formato `yaml` ed alla loro gestione. Esse verranno rese disponibili al Plugin tramite un parametro di tipo `Object` che viene passato alla funzione `New` e in questo modo sarà possibile accedere ai parametri. Gli script generano delle funzioni di basso livello per la gestione di queste strutture dati, come ad esempio la deep copy. Infine, il valore di default, che è opzionale, viene assegnato quando nella configurazione non viene specificato il parametro.

Nei codici 8 e 9 è mostrato come creare le strutture dati per la gestione dei parametri del Plugin. Il primo crea la struttura dati che verrà utilizzata dalla funzione `New`, mentre il secondo riguarda il parsing dei parametri. Si possono notare in entrambi i casi dei commenti che precedono la struttura dati con una sintassi specifica; essa

viene utilizzata dagli script per generare il codice a basso livello per queste `struct`. Il nome di queste strutture segue obbligatoriamente il formato `<NomePlugin>Args` e vi si aggiungono altri campi necessari attraverso la struct embedding del tipo `TypeMeta`. Infine, si nota che nel codice 9, tramite la feature struct tag di `Go`, si aggiungono informazioni utili al parsing, come la stringa che rappresenta la chiave per quello specifico parametro nel file `yaml`.

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

type MyPluginArgs struct {
    metav1.TypeMeta
    Arg int64
}
```

Listing 8: Struttura dati per la gestione dei parametri del Plugin da aggiungere nel file `apis/config/types.go`.

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
// +k8s:defaulter-gen=true

type MyPluginArgs struct {
    metav1.TypeMeta `json:",inline"`
    Arg *int64 `json:"arg,omitempty"`
}
```

Listing 9: Struttura dati per il parsing dei parametri del Plugin da aggiungere nel file `apis/config/v1/types.go`.

In caso sia necessario assegnare un valore di default all'argomento, si deve dichiarare un'apposita funzione nel file `apis/config/v1/defaults.go`, come mostrato nel codice 10. Per rendere riconoscibile la funzione, è necessario che essa abbia il nome nel formato `SetDefaults_<NomePlugin>Args`. È consuetudine dichiarare la variabile che contiene il valore di default, in questo caso `arg`, globalmente, ad inizio file e non all'interno della funzione stessa, ma non è strettamente necessario. Se i passaggi precedenti sono stati eseguiti correttamente, nel file sarà già importato il tipo di dato `MyPluginArgs`.

```go
func SetDefaults_MyPluginArgs(obj *MyPluginArgs) {
    var arg int64 = 42
    if obj.Arg == nil {
        obj.Arg = &arg
    }
}
```

Listing 10: Assegnazione ai parametri dei valori di default nel file `apis/config/v1/defaults.go`.

A questo punto, si può eseguire lo script `hack/update-codegen.sh` per generare il codice necessario alla gestione delle nuove strutture dati a basso livello. Dopo l'esecuzione, nell'output, vengono riportati gli avvertimenti 11 che considerano deprecato l'utilizzo di alcuni comandi che questo script invoca.

```
WARNING: generate-internal-groups.sh is deprecated.
WARNING: Please use k8s.io/code-generator/kube_codegen.sh instead.
```

Listing 11: Warning del comando `hack/update-codegen.sh`.

Nel precedente output viene suggerito l'utilizzo di un comando differente: `k8s.io/code-generator/kube_codegen.sh`. Il percorso fornito è incompleto, ma, conoscendo la struttura del repository, si può facilmente risalire al file `vendor/k8s.io/code-generator/kube_codegen.sh`. Nella stessa cartella, esiste un `README.md` col riferimento ad un articolo che dovrebbe spiegare step-by-step l'utilizzo della libreria. Purtroppo, però il collegamento è ad un articolo datato che suggerisce ancora di utilizzare il comando `hack/update-codegen.sh` [23].

Fortunatamente, nonostante i warning mostrati e l'errore che conclude l'output dello script, `hack/update-codegen.sh` è ancora utilizzabile, ma con un accorgimento. Infatti è necessario recuperare un file che è stato cancellato dal comando: `apis/scheduling/v1alpha1/zz_generated.deepcopy.go`. Per farlo può essere scaricato nuovamente dal repository GitHub oppure ripristinato tramite la Git history.

Per recuperare il valore del parametro e salvarlo nello stato del Plugin è necessario modificare la funzione `New` come nel codice 12. Questo codice necessita anche

di importare i package `fmt` e `sigs.k8s.io/scheduler-plugins/apis/config`, oltre quelle già importati.

```
func New(_ context.Context, obj runtime.Object, _ framework.Handle)
↪  (framework.Plugin, error) {
    var args *config.MyPluginArgs
    args, ok := obj.(*config.MyPluginArgs)
    if !ok {
        return nil, fmt.Errorf("Errore nel recupero degli argomenti")
    }
    return &MyPlugin{arg: args.Arg}, nil
}
```

Listing 12: Modifica alla funzione `New` per ottenere i parametri.

3.2.4 Logging

Il modo più semplice per riuscire ad apprezzare le modifiche apportate dal Plugin è aggiungere una riga ai log dell'applicazione. Questo è anche un metodo utilizzato per il debugging. Nella documentazione di Kubernetes esiste una sezione dedicata proprio al logging [24].

Per questo compito viene utilizzata la libreria `klog` [20]. Si tratta di un fork permanente, mantenuto dai creatori di Kubernetes, di `glog`, la libreria di logging creata dagli sviluppatori di `Go` [12]. Questa libreria è stata sviluppata per sopperire ad alcune mancanze di `glog` che non sarebbero state colmate in quanto il progetto non è più attivamente sviluppato.

Vediamo, nel codice 13, come modificare la funzione `PreFilter` del codice 4 per aggiungere un messaggio di log. È anche necessario importare il package `k8s.io/klog/v2` oltre a quelli già importati.

```
func (mp *MyPlugin) PreFilter(_ context.Context, _ *framework.CycleState, _ *v1.Pod)
↪ (*framework.PreFilterResult, *framework.Status) {
    klog.V(4).Info("L'Extension Point PreFilter del Plugin MyPlugin è stato chiamato
    ↪ col valore del parametro: ", mp.arg)
    return nil, nil
}
```

Listing 13: Esempio di utilizzo di `klog` per il logging.

È importante notare il valore numerico passato come parametro alla funzione `V` di `klog`; esso è il livello di logging al quale verrà stampata la stringa. Quindi è necessario impostare un livello di log adeguato per riuscire a osservare il funzionamento del Plugin. Per fare questo, si utilizza la flag `--v` da aggiungere al comando `kube-scheduler` [21]. Il comando `kube-scheduler` in realtà non viene invocato direttamente, ma, con queste informazioni, nella prossima sezione, potrà essere configurato adeguatamente lo scheduler.

3.3 Configurazione

Per configurare il Plugin è necessario creare due file `.yaml`: il primo, con uno Scheduling Profile, il secondo con la configurazione dello scheduler che indica di utilizzare quel profilo. Questi file sono solitamente contenuti nella cartella `manifests/` del repository.

Il primo file, che è riportato nel codice 14, contiene la definizione di un profilo per lo scheduler che semplicemente attivi il Plugin desiderato. Si nota, infatti, la specificazione di un profilo chiamato `default-scheduler`, come richiesto dalla documentazione già trattata, che abilita per tutti i suoi Extension Point `MyPlugin`. In questo modo, ogni Pod utilizza lo Scheduling Profile con il Plugin appena creato. Inoltre si ha, da riga 11 in poi, il passaggio del parametro a `MyPlugin`. È questo il modo in cui gli utenti possono inserire i parametri desiderati. Se sono richiesti parametri, ma non è stata creata una funzione per il loro valore di default, è strettamente necessario che venga inserito un valore nell'apposito campo di questa configurazione. In seguito, faremo riferimento a questo file con il nome `myplugin-config.yaml`.

```yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/etc/kubernetes/scheduler.conf"
profiles:
- schedulerName: default-scheduler
  plugins:
    multiPoint:
      enabled:
      - name: MyPlugin
  pluginConfig:
  - name: MyPlugin
    args:
      arg: 13
```

Listing 14: Scheduler Profile che abilita MyPlugin.

Il secondo file è la specifica di un Pod di sistema, nello specifico, quello che esegue lo scheduler. Esso esiste già nel cluster e sarebbe sufficiente modificarlo, ma per comodità, se ne può creare uno e sostituire quello pre-esistente.

Come si può vedere nel codice 15, ci sono parecchie configurazioni, ma solo alcune di esse sono di interesse per il funzionamento di MyPlugin. In particolare si notano:

- A riga 14, l'aggiunta al comando `kube-scheduler` della flag `--v` per il livello di verbosità dei log, configurato in accordo con il valore scelto in fase di programmazione. In questo caso specifico, si è scelto il valore 5.
- A riga 20, si ha la flag col riferimento a quello che sarà il percorso del file contenente la specifica del profilo creato in precedenza. Il path è relativo all'ambiente nel quale eseguirà lo scheduler e non a quello del repository. Nella prossima sezione si forniranno più informazioni su questo ambiente.
- Nel campo `image`, alla riga 20, si trova il nome dell'immagine con cui creare il container da utilizzare. Nella riga immediatamente successiva si trova il campo `imagePullPolicy` che potrebbe essere lasciato al valore originale, `Always`, o portato a `IfNotPresent`.
- Si aggiungono poi a `volumeMounts` e `volumes`, rispettivamente a riga 47 e 60, le configurazioni per rendere disponibile lo Scheduling Profile precedentemente creato, tramite il percorso al file `myplugin-config.yaml`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: kube-scheduler
    tier: control-plane
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --v=5    # deve essere non minore del parametro attuale della funzione di log
    - --authentication-kubeconfig=/etc/kubernetes/scheduler.conf
    - --authorization-kubeconfig=/etc/kubernetes/scheduler.conf
    - --bind-address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=false
    - --config=/etc/kubernetes/myplugin-config.yaml
    image: localhost:5000/scheduler-plugins/kube-scheduler:latest
    imagePullPolicy: IfNotPresent
    livenessProbe:
      failureThreshold: 8
      httpGet:
        host: 127.0.0.1
        path: /healthz
        port: 10259
        scheme: HTTPS
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 15
    name: kube-scheduler
    resources:
      requests:
        cpu: 100m
    startupProbe:
      failureThreshold: 24
      httpGet:
        host: 127.0.0.1
        path: /healthz
        port: 10259
        scheme: HTTPS
      initialDelaySeconds: 10
      periodSeconds: 10
      timeoutSeconds: 15
    volumeMounts:
    - mountPath: /etc/kubernetes/scheduler.conf
      name: kubeconfig
```

```yaml
50        readOnly: true
51      - mountPath: /etc/kubernetes/myplugin-config.yaml
52        name: myplugin-config
53        readOnly: true
54    hostNetwork: true
55    priority: 2000001000
56    priorityClassName: system-node-critical
57    securityContext:
58      seccompProfile:
59        type: RuntimeDefault
60    volumes:
61    - hostPath:
62        path: /etc/kubernetes/scheduler.conf
63        type: FileOrCreate
64      name: kubeconfig
65    - hostPath:
66        path: /etc/kubernetes/myplugin-config.yaml
67        type: File
68      name: myplugin-config
69  status: {}
```

Listing 15: Configurazione del Pod di `kube-scheduler` con l'aggiunta del profilo che abilita `MyPlugin`.

3.4 Compilazione, installazione e testing

Per vedere il Plugin creato all'opera, è necessario poter eseguire un'istanza di Kubernetes su un cluster. In un ambiente di sviluppo nel quale non si possa avere un intero cluster a scopo di testing, è possibile utilizzare un tool per simularlo. Si tratta di Minikube, uno strumento che avvia un cluster Kubernetes di macchine virtuali o container in locale [33]. Per gestire il cluster è possibile anche installare `kubectl`, un tool apposito [5]; esso è anche disponibile come istruzione di Minikube. Nel seguito, si fa ricorso al comando `kubectl` fornito da Minikube.

Il codice del repository è compilato ed eseguito all'interno di un container, che viene lanciato come Pod sul cluster. Di conseguenza, per compilarlo ed ottenere l'immagine del container, è necessaria l'installazione di Docker [9].

Infine, è necessario Make, uno strumento che controlla la generazione di file eseguibili a partire dai file sorgenti di un programma [30]. Esso è pensato per compilare i sorgenti del repository, lo testimonia il `Makefile` presente. Può anche essere utilizzato per facilitare alcune parti della compilazioni in fase di sviluppo.

Per cominciare, la compilazione è necessario eseguire il comando `make local-image` nella cartella principale del repository. Con esso, si compila il codice del progetto nelle immagini di scheduler e controller. Per più informazioni su come avviene la compilazione si consultino i `Dockerfile` presenti nella cartella `build/` del repository. Per controllare la corretta generazione delle immagini è possibile utilizzare il comando: `docker images | grep 'localhost:5000/scheduler-plugins/controller\| localhost:5000/scheduler-plugins/kube-scheduler'`. I pattern cercati dal programma `grep` sono i nomi delle due immagini generate; potrebbero variare, in particolar modo nella prima parte. Si noti che si tratta della stessa stringa con cui ci si riferiva all'immagine dello scheduler nella configurazione 15 vista in precedenza.

Lo script che esegue queste azioni compila sia il controller che lo scheduler. Questo non è necessario, infatti successivamente sarà sufficiente utilizzare l'immagine dello scheduler.

Il prossimo passaggio consiste nel esportare l'immagine in formato `.tar`; per farlo è opportuno creare una cartella apposita, ad esempio `tar/`. Docker permette di esportare delle immagini in formato `.tar` tramite il comando: `docker save -o tar/kube-scheduler.tar localhost:5000/scheduler-plugins/kube-scheduler`.

Ora è possibile caricare l'immagine su Minikube; non prima però di averlo avviato col comando `minikube start`. `minikube image load tar/kube-scheduler.tar` è l'istruzione che deve essere eseguita. Per controllare la buona riuscita di questa operazione si utilizza l'istruzione `minikube image ls`; per un output più puntuale è possibile utilizzare `grep` come fatto in precedenza.

Nonostante sia stata caricata l'immagine del container, le modifiche apportate con essa non avranno ancora preso effetto sullo scheduler. Infatti Kubernetes inizia ad utilizzare questa immagine solo quando è nominata nelle configurazioni. È quindi arrivato il momento di applicare i file di configurazione preparati in precedenza. Per

farlo, c'è il comando `minikube cp`, che in generale permette di copiare file tra Nodi del cluster e l'esterno. L'obiettivo è quello di copiare i due file `.yaml` nelle corrette posizioni del file system di modo che essi ci permettano di raggiungere l'effetto desiderato.

Si esegue prima il comando `minikube cp manifests/myplugin/myplugin-config.yaml minikube:/etc/kubernetes/myplugin-config.yaml` per copiare il file di configurazione del profilo di scheduling. Poi `minikube cp manifests/myplugin/kube-scheduler.yaml minikube:/etc/kubernetes/manifests/kube-scheduler.yaml` per copiare il file di configurazione del Pod di `kube-scheduler`.

Per apprezzare il funzionamento dei comandi precedenti, si utilizza un'istruzione per ottenere il nome del container che sta eseguendo attualmente come Pod dello scheduler. Esso è `minikube kubectl -- get pods -l component=kube-scheduler -n kube-system -o=jsonpath="{.items[0].spec.containers[0].image}{'\n'}"`. Perché abbiano effetto i comandi precedenti, potrebbe volerci qualche istante quindi se il nome non è quello dell'immagine caricata riprovare il comando dopo pochi secondi. Infine, per controllare lo stato del Pod che esegue lo scheduler, si utilizza il comando `minikube kubectl -- get pod -n kube-system`.

Non resta che creare un Pod e testare il funzionamento del Plugin. Infatti, l'Extension Point implementato si avvia quando un Pod viene creato e passa per lo Scheduling Cycle, in particolare per la fase di PreFilter. Per apprezzarne il funzionamento, è necessario ispezionare i log. Esiste un comando che permette di leggere i log dei Pod; per ottenere quelli dello scheduler il comando è: `minikube kubectl -- logs -n kube-system kube-scheduler-minikube`. Tra essi ce n'è uno come quello mostrato nell'output 16.

```
08:01:24.901004       1 myplugin.go:28] L'Extension Point PreFilter del
Plugin MyPlugin è stato chiamato col valore del parametro: 13
```

Listing 16: Log di `kube-scheduler` con MyPlugin attivo.

3.4.1 Makefile

Nel caso in cui diventi indispensabile ripetere spesso queste ultime operazioni, è possibile automatizzare il processo con uno o più target nel `Makefile`, come mostrato nel codice 17. Ciò risulta particolarmente utile in fase di sviluppo, quando si effettuano molte modifiche e si vuole testare il funzionamento del Plugin. L'attivazione e disattivazione di Minikube non è gestita dal `Makefile` e deve essere eseguita manualmente. Il target `clean-myplugin` permette di rimuovere l'immagine dal cluster e dal sistema locale, ma deve essere eseguito in seguito ad un restart di Minikube, altrimenti l'immagine non potrà essere rimossa in quanto ancora in uso.

```
TAR_IMAGE_DIR=tar
CONTROLLER_IMAGE_NAME=localhost:5000/scheduler-plugins/controller
SCHEDULER_IMAGE_NAME=localhost:5000/scheduler-plugins/kube-scheduler
CONTROLLER_IMAGE_TAR=controller.tar
SCHEDULER_IMAGE_TAR=kube-scheduler.tar
KUBE_SCHED_FILE=kube-scheduler.yaml
SCHED_CONFIG_FILE=myplugin-config.yaml
MANIFESTS_DIR=manifests/myplugin
MINIKUBE_PROFILE=minikube
SCHED_CONFIG_LOCATION=/etc/kubernetes
KUBE_SCHED_LOCATION=$(SCHED_CONFIG_LOCATION)/manifests

.PHONY: myplugin
myplugin: build-local-image load-local-image copy-config-files

FILTER_IMAGES=grep '$(CONTROLLER_IMAGE_NAME)\|$(SCHEDULER_IMAGE_NAME)' --color=auto

.PHONY: build-local-image
build-local-image: local-image
	docker images | $(FILTER_IMAGES)
	mkdir -p $(TAR_IMAGE_DIR)
	docker save -o $(TAR_IMAGE_DIR)/$(SCHEDULER_IMAGE_TAR)
	↪ $(SCHEDULER_IMAGE_NAME)

.PHONY: load-local-image
load-local-image:
	minikube image load $(TAR_IMAGE_DIR)/$(SCHEDULER_IMAGE_TAR)
	minikube image ls | $(FILTER_IMAGES)

.PHONY: copy-config-files
copy-config-files:
	minikube cp $(MANIFESTS_DIR)/$(SCHED_CONFIG_FILE)
	↪ $(MINIKUBE_PROFILE):$(SCHED_CONFIG_LOCATION)/$(SCHED_CONFIG_FILE)
```

```
32          minikube cp $(MANIFESTS_DIR)/$(KUBE_SCHED_FILE)
            ↪  $(MINIKUBE_PROFILE):$(KUBE_SCHED_LOCATION)/$(KUBE_SCHED_FILE)
33          minikube kubectl -- get pods -l component=kube-scheduler -n kube-system
            ↪  -o=jsonpath="{.items[0].spec.containers[0].image}{'\n'}"
34          minikube kubectl -- get pod -n kube-system | grep kube-scheduler --color=auto
35
36  .PHONY: clean-myplugin
37  clean-myplugin:
38          minikube image rm $(SCHEDULER_IMAGE_NAME)
39          docker image rm $(SCHEDULER_IMAGE_NAME)
40          rm $(TAR_IMAGE_DIR)/*
```

Listing 17: Target per la compilazione, installazione e testing del Plugin da aggiungere a `Makefile`.

4 Ottimizzazione del Default Scheduler

Dopo la panoramica sul funzionamento di Kubernetes, di alcuni suoi componenti, nello specifico dello scheduler, e una visione di come creare un Plugin per esso; la trattazione si sposta ora su come operare in questo contesto e con questi strumenti. L'obiettivo di questo capitolo è quello di giungere all'ottimizzazione del Default Scheduler come proposto dalla tesi di S. Canova [3]. Il risultato ottenuto è una proof of concept di quell'idea. Ma anche della possibilità di avere nello scheduler di Kubernetes un algoritmo di preemption tra Nodi che, come vedremo, non è scontato.

4.1 Ottimizzazione dello scheduler

Prima di entrare nel merito della soluzione che questa tesi tenta di implementare, è opportuno capire da che problema nasce questa necessità.

4.1.1 Problema del Default Scheduler di Kubernetes

Come è stato possibile osservare dalle scelte progettuali dei designer, lo scheduler di Kubernetes è pensato per favorire la rapidità della sua esecuzione. Questo a volte va a discapito della precisione nella scelta di scheduling. I risultati subottimali del Default Scheduler sono rapidi da raggiungere e quindi ideali per uno scenario firm o soft real time, dove la qualità del servizio (quality of service, QoS) degrada tanto

più la risposta del sistema tarda ad arrivare, ma potrebbero non essere adatti ad altri scenari.

Basti pensare, ad esempio, al sopraggiungere di un carico di lavoro in grado di saturare tutti i Nodi a disposizione che renderebbe infattibile l'assegnamento di un eventuale nuovo Pod. Andando a gestire al meglio le risorse a disposizione, potrebbe essere possibile garantire lo scheduling di tale carico di lavoro anche in un cluster con risorse limitate, ottenendo un risparmio per il fornitore del servizio. Infatti quest'ultimo potrebbe evitare di dover aumentare la quantità di risorse del cluster per far fronte al carico di lavoro.

Nelle prossime sezioni scopriremo perché il Default Scheduler o qualche Plugin non implementano già una soluzione per risolvere questo problema.

4.1.2 Proposta

La possibilità di personalizzare lo scheduler, potrebbe permettere di raggiungere una soluzione anche per i casi in cui il riempimento dei Nodi è desiderabile. La proposta è quella di portare un solver che tratti questo problema nel Default Scheduler con un Plugin. Infatti, a differenza di altri lavori in questa direzione, come Boreas [27] e Sage [29], che cercano di applicare la loro logica ad ogni Scheduling Cycle, rischiando di rallentarlo, in questo lavoro si applica un algoritmo per raggiungere l'ottimo solo quando non è possibile assegnare un Pod ad alcun Nodo con le normali logiche di scheduling.

Con l'adeguato utilizzo degli Extension Point dello scheduler, è possibile implementare un Plugin che entri in gioco solo quando necessario. Si tratterebbe proprio di un PostFilter Plugin. Per questa fase dello Scheduling Cycle è già stato pensato un algoritmo di preemption implementato in un default Plugin. Esso però attua solo il meccanismo di espulsione dei Pod da un singolo Nodo per fare spazio ad un altro Pod, presumibilmente a più alta priorità. Starà poi ai prossimi Scheduling Context tentare di riassegnare quei Pod ai Nodi; con il rischio che le scelte prese dallo scheduler facciano incappare nuovamente il cluster in un problema di saturazione. Come vedremo infatti lo scheduler di Kubernetes non prevede la possibilità

di interrompere un Pod su un Nodo con la precisa intenzione di farlo ripartire su un altro Nodo.

Entrando nel dettaglio, l'idea è quella di implementare un algoritmo di Cross-Node Preemption sulla base delle decisioni prese da un solver, il quale massimizza il numero di Pod che possono usufruire delle risorse. Il tutto minimizzando il numero di servizi interrotti per ricollocamento o espulsione e tenendo conto dei vincoli di scheduling. Passando, infatti, le informazioni necessarie ad un solver, si può ottenere una soluzione ottimale per il riempimento dei Nodi del cluster. La soluzione sarebbe ottimale sia in termini di risorse occupate, che di spostamenti di Pod tra i Nodi. Questo tipo di algoritmo stop-the-world che coinvolge un solver, è inevitabilmente più lento di uno greedy. Ma limitando il suo utilizzo ai soli casi critici, nei quali il cluster sembra essere saturo, si può ottenere un buon compromesso tra rapidità e precisione. Inoltre la sua implementazione come Plugin permetterebbe agli utenti di adottare questa soluzione solo se necessario nel loro caso d'uso.

4.2 Cross-Node Preemption

Come già accennato in precedenza, il Default Scheduler di Kubernetes non implementa logiche di preemption tra Nodi. E questa non è tanto una scelta progettuale dello scheduler di default, ma più una scelta di design di Kubernetes. Infatti, essa non è una strada percorribile neanche per gli sviluppatori di Plugin. La ragione è che non si tratta di una feature disponibile di Kubernetes, come riportato nella documentazione ufficiale [42]. La documentazione stessa spiega che i designer potrebbero "considerare di aggiungere la Cross-Node Preemption in futuro se ci sarà abbastanza richiesta e se troveranno un algoritmo con performance ragionevoli".

Tra i Plugin di esempio creati dagli utenti nel repository `scheduler-plugins`, ne è presente uno che tenta inutilmente di sfruttare questa feature [7], a testimonianza della presenza di richiesta.

La proposta di questa tesi può essere anche un punto di partenza per ideare un algoritmo sufficientemente efficiente per la Cross-Node Preemption. Magari riaprendo

la possibilità in futuro di implementare questa feature nello scheduler di Kubernetes da parte degli sviluppatori.

Intanto, per bypassare la mancanza di questa feature, è stato necessario trovare un escamotage. Si tratta di eseguire l'espulsione di uno o più Pod, per fare spazio a quello unscheduled; come già avviene con la preemption intra-Nodo. Per poi ricollocarli, in più cicli di scheduling, su Nodi scelti appositamente. Nelle prossime sezioni sarà analizzato come tutto ciò sia possibile.

4.3 Implementazione del Plugin per l'ottimizzazione dello scheduler

L'implementazione è legata alla già citata tesi di Simone Canova che ha iniziato questo lavoro.

4.3.1 Lo script e il solver

Lo sviluppo è iniziato dal codice scritto da Simone Canova nella sua tesi nella quale ha studiato gli aspetti teorici di questa ottimizzazione. Si tratta di uno script `python` utilizzato per fare test sull'idea. Esso, presi in input dei parametri in formato `csv`, li formalizza per darli in input ad un solver, OrTools o Z3, e generare una soluzione che stampa in formato `json`.

Dai risultati dei test, si è potuto concludere che il solver OrTools è più performante per questo problema, oltre che in grado di fornire maggiore flessibilità. Per questo motivo è stato scelto per lo sviluppo del Plugin. OrTools è una software suite free e Open Source per l'ottimizzazione sviluppata da Google [38]. Sfortunatamente non è presente il supporto per `Go`, il linguaggio in cui è scritto lo scheduler di Kubernetes. Per questo motivo si utilizzerà lo script `python` per interfacciarsi con OrTools.

L'utilizzo di uno script esterno per il funzionamento dello scheduler non è una pratica comune e consigliata in quanto potrebbe rallentare l'esecuzione dello scheduler.

Nonostante ciò, come già detto, si tratta di una proof of concept e non di una soluzione production-ready, quindi si può accettare questo compromesso.

L'utilizzo dello script `python` richiede un po' di configurazione all'interno dell'ambiente di esecuzione dello scheduler. Esso, come visto, è un Pod, quindi un container, creato a partire da un'immagine Docker. Si può trovare il `Dockerfile` che crea l'immagine dello scheduler nella cartella `build/scheduler/` del repository 2. Nel codice 18 è possibile osservare le modifiche effettuate.

L'immagine utilizzata è quella di Alpine, una distribuzione Linux minimale orientata alla sicurezza e alla semplicità [18]. L'installazione dell'ambiente necessario per l'esecuzione dello script in un contesto così minimale, è risultata un operazione complicata. D'altronde, il minimalismo di una distribuzione come Alpine può essere affine alla necessità di un componente come il classico scheduler di Kubernetes che necessita solo di eseguire un codice `Go` compilato. Dal primo momento in cui si esegue codice interpretato, come può essere lo script `python`, le necessità sono inevitabilmente differenti. Per questa ragione si è scelto di utilizzare un'immagine più adatta ai nuovi scopi. Si è optato infatti, per un immagine Ubuntu, una distribuzione Linux general purpose [11].

Per predisporre l'ambiente, in questo primo stage di compilazione dell'immagine, è sufficiente installare `python` e gli strumenti per la creazione di un virtual environment, come è possibile osservare nelle righe 14 e 15 del codice 18.

Inizialmente nel `Dockerfile` copia il Build Context all'interno dell'immagine. Esso non è altro che il repository `scheduler-plugins`. Per inserire quindi nell'immagine dello scheduler lo script, questo deve essere stato spostato all'interno della struttura del repository 2. In particolare, nel seguito si considera il codice `python` all'interno della cartella `script/`. Successivamente, si copia dallo stage iniziale lo script in quello successivo, riga 20. Esattamente come fatto a riga 18 con il comando `kube-scheduler`, frutto della compilazione del progetto.

Infine, nelle righe 22 e 23, si crea il virtual environment e vi si installano le dipendenze dello script tramite il relativo file `requirements.txt`. Per minimizzare la dimensione dell'immagine e la mole di dipendenze da installare, si utilizza una versione ridotta

dello script; per questa ragione le dipendenze si riconducono al solo OrTools.

Come accennato nel capitolo precedente, questo `Dockerfile` verrà utilizzato della entry `local-image` del `Makefile`.

```
ARG ARCH
ARG GO_BASE_IMAGE=golang
ARG UBUNTU_BASE_IMAGE=ubuntu
FROM $GO_BASE_IMAGE:1.21

WORKDIR /go/src/sigs.k8s.io/scheduler-plugins
COPY . .
ARG ARCH
ARG RELEASE_VERSION
RUN RELEASE_VERSION=${RELEASE_VERSION} make build-scheduler.$ARCH

FROM $UBUNTU_BASE_IMAGE:22.04

RUN apt-get update && apt-get install -y python3 python3-pip python3-venv
RUN ln -s /usr/bin/python3 /usr/bin/python

ENV SCRIPT_PATH=/opt/script
COPY --from=0 /go/src/sigs.k8s.io/scheduler-plugins/bin/kube-scheduler
↪ /bin/kube-scheduler
RUN mkdir -p $SCRIPT_PATH
COPY --from=0 /go/src/sigs.k8s.io/scheduler-plugins/script/* $SCRIPT_PATH/

RUN python3 -m venv $SCRIPT_PATH/venv
RUN $SCRIPT_PATH/venv/bin/pip install -r $SCRIPT_PATH/requirements.txt    # ortools

WORKDIR /bin
CMD ["kube-scheduler"]
```

Listing 18: `Dockerfile` per la compilazione dello scheduler.

4.3.2 Plugin

Una volta reso disponibile lo script nell'ambiente di esecuzione dello scheduler, non resta che richiamarlo con un Plugin. Come già osservato, l'Extension Point più adatto per eseguire la logica di Cross-Node Preemption è PostFilter. Così quando lo scheduler non riesce a far fronte ad un carico di lavoro per saturazione dei Nodi, il Plugin si attiva, passando le informazioni allo script `python` e ottenendo una

soluzione ottimale, se possibile. Nello Scheduling Cycle nel quale il Plugin viene attivato e si ottiene una soluzione, si espellono i Pod da spostare e si inserisce il Pod unscheduled. Negli Scheduling Cycle successivi il Plugin rimane attivo ed esegue il ricollocamento dei Pod espulsi. Infine il Plugin si disattiva quando la soluzione è stata interamente applicata al cluster. È in questo modo che si ottiene l'effetto della Cross-Node Preemption in più passaggi.

È possibile consultare il codice del Plugin costruito sulla base di questa idea in un repository GitHub [35], che è una fork di `scheduler-plugins`.

Osserviamo ora nel dettaglio gli Extension Point implementati dal Plugin, gli altri punti di interesse del codice e quali compiti svolgono nell'esecuzione della sua logica.

New

Seguendo il metodo illustrato nel capitolo precedente, si è fornito il Plugin di un argomento che verrà utilizzato dal solver: il `timeout`. Nel metodo `New` si recupera questo parametro e lo si inserisce nello stato. Lo stato del Plugin, in questa funzione, viene anche popolato dell'oggetto `Handle`, che durante l'esecuzione sarà utile ad ottenere informazioni sul cluster o applicarvi le decisioni prese dallo script.

PostFilter

Si tratta della fase più importante nell'esecuzione del Plugin. Innanzitutto, questo Extension Point esegue la sua logica solo se il Plugin non è già attivo. Essa consiste nell'ottenere le informazioni sul cluster attraverso l'oggetto `Handle` salvato nello stato. Il tipo di informazioni che estrapola sono quelle necessarie allo script per funzionare. Tra esse ci sono le risorse a disposizione dei Nodi, quelle utilizzate da ogni Pod, la loro priorità, le Affinity e anche le attuali associazioni tra Nodo e Pod. Si salvano poi queste informazioni su un file in formato `csv`, proprio come richiesto dallo script. Si può osservare un esempio del formato di file `csv` che lo script utilizza nella tabella 4.1. Esso comprende il campo `type` che suggerisce se si tratta di un Nodo, quì chiamato `bin`, o di un Pod. È presente anche un attributo `index` che identifica i Nodi e i Pod; i primi hanno un index che inizia da 1 in quanto se un

Pod è assegnato al Nodo 0 significa che il solver ha deciso di rimuovere quel Pod o che esso non è assegnato ad alcun Nodo. Si può notare inoltre, che alcuni campi sono condivisi, mentre altri sono riservati all'uno o all'altro `type`. Come ad esempio `where`, che rappresenta l'assegnato di un Pod ad un Nodo, che è riservato al `type pod`. I campi `label`, `affinity` e `anti_affinity` sono delle liste di stringhe in quanto possono essere presenti molteplici valori per questi attributi.

type	index	ram	cpu	label	where	priority	affinity	anti_affinity	namespace
bin	1	5	5	['ssd']					
bin	2	6	6	['10Gbps']					
pod	0	5	5		2	10	[]	[]	kube-system
pod	1	2	2		0	50	['ssd']	[]	default
pod	2	2	2		0	50	[]	[]	kube-system
pod	3	2	2		0	50	[]	['10Gbps']	default

Tabella 4.1: Esempio di file `csv` che lo script accetta in input.

Il campo `namespace`, presente nella tabella 4.1, non era inizialmente richiesto dallo script utilizzato nella tesi di Simone Canova. Infatti, erano state fatte supposizioni inesatte sui Pod di sistema, dovute alla scarsa documentazione degli aspetti più profondi dello scheduler. Questi ultimi sono emersi durante lo sviluppo del Plugin e quindi prontamente corretti anche nello script. In particolare i Pod con `namespace kube-system` sono i Pod che permettono il funzionamento di Kubernetes, sono quindi Pod critici che non è possibile spostare. La supposizione dello script era invece che i Pod dei quali non si poteva effettuare l'espulsione, fossero tutti e soli quelli a priorità più alta. Nella realtà lo scheduler permette ai Pod creati dagli utenti, di raggiungere una priorità maggiore di quella dei Pod di sistema. Inoltre, non tutti i pod del `namespace kube-system` hanno la stessa priorità.

Altro particolare degno di nota sullo script, è quello relativo ai valori della colonna `where` della tabella 4.1. Come si può notare ci sono molteplici Pod non allocati, quelli con valore 0. Questo perché il solver è in grado di trovare una soluzione anche in un caso come questo. Però, quella che ci si trova ad affrontare nel Plugin è una situazione dove solo un Pod è unscheduled. Questo viene dalla natura stessa dello scheduler: lo Scheduling Context viene avviato ogni qualvolta si crea un Pod e, anche se ne venissero creati molteplici contemporaneamente, ne gestisce uno per

volta. Questo non costituisce un problema per l'utilizzo dello script, in quanto si tratta di un caso particolare di quelli che il solver sarebbe in grado di gestire. Si farà però utilizzo di questa particolare situazione nel seguito dello sviluppo.

Un'azione complementare alla compilazione del file `csv`, è quella di creare un sistema di riconversione degli `index` di Nodi e Pod in riferimenti utilizzabili nel codice del Plugin. Questa operazione viene svolta da due oggetti di tipo `map` che permettono di riconvertire gli `index` rispettivamente in nomi di Nodi, coi quali si riesce facilmente ad ottenere le altre informazioni, e riferimenti alla struttura dati `Pod`. Questi oggetti sono salvati nello stato del Plugin per essere utilizzati in seguito.

Una volta salvato il file, il Plugin richiama lo script passando alcuni parametri ed argomenti. Tra essi c'è il percorso al file `csv` appena creato, il `timeout` inserito dall'utente ed altri che permettono di eseguire il comando con una quantità minimale di output per una sua migliore gestione.

Proprio l'analisi dell'output è il successivo passaggio. Dopo alcuni controlli per evitare errori o eccezioni, si esegue il parsing dell'output che è in formato `json`. I possibili errori o eccezioni possono derivare da input incorretto, malfunzionamento dell'ambiente `python`, ma anche dal fatto che neppure il solver sia in grado di giungere ad una soluzione per l'attuale situazione del cluster[1]. Dall'output corretto, invece, si estrae solo l'informazione relativa agli spostamenti dei Pod. Essa è sotto forma di una lista nella quale ogni posizione rappresenta l'`index` del Pod e il valore della posizione è l'`index` del Nodo al quale il Pod dovrebbe essere assegnato secondo la soluzione ottimale. Questo, in effetti, è coerente con il range degli `index` dei Pod che parte proprio da 0, come le posizioni degli array. Per quanto riguarda il valore delle sue celle invece, il valore 0 rappresenterebbe la semplice espulsione senza ricollocamento.

È possibile anche che il solver giunga alla conclusione che quella presente sul cluster sia già la miglior soluzione possibile. Per rendersi conto di ciò è sufficiente fare un semplice confronto tra le posizioni iniziali dei Pod fornite in input allo script e quelle

[1] Ciò può accadere, ad esempio, se il `timeout` fornito allo script per eseguire dovesse risultare insufficiente.

restituite in output da esso. Proprio da questo confronto si ottiene una lista delle differenze nella quale, se il Pod con un determinato `index` non è stato spostato, il valore di quella posizione sarà 0, altrimenti assumerà il valore dell'`index` al quale è stato spostato. Questa lista viene salvata nello stato del Plugin ed è utilizzata per applicare la soluzione. Infatti con essa si può tenere traccia dei Pod che sono già nel Nodo previsto, quelli con valore 0 nell'`index` corrispondente, e di quelli da spostare, con relativo `index` del Nodo su cui spostarli. Successivamente si farà riferimento a questa lista semplicemente come "soluzione".

Infine, da questo array si ricava un oggetto di tipo `candidate`, simile a quello utilizzato anche nel Plugin di default per la preemption intra-Nodo. Esso è composto da una lista di riferimenti ad oggetti di tipo `Pod`, alla quale ci si riferisce come "vittime", e una stringa che rappresenta il nome del Nodo al quale il Pod unscheduled dovrebbe essere assegnato. Questa `struct` implementa il metodo `EvictVictims` che si occupa di espellere i Pod da spostare dai Nodi su cui si trovano oppure di arrestare il loro Binding Cycle se essi si trovano in attesa nel Extension Point Permit. Dopo aver eseguito il metodo `EvictVictims`, l'Extension Point PostFilter termina con successo ritornando il nome del Nodo scelto. Sarà lo scheduler a curarsi internamente che l'associazione tra esso e il Pod unscheduled avvenga correttamente.

PreFilter

La fase successiva è quella di PreFilter. Il Plugin in questo Extension Point deve essere eseguito prima di qualsiasi altra logica di scheduling. La sua logica viene eseguita condizionatamente alla presenza della soluzione nello stato del Plugin. Se essa non è presente, viene ritornato lo status `Skip`.

In accordo coi commenti presenti nelle interfacce di questo Extension Point, che si trovano nel file illustrato nel codice 3, se il Plugin ritorna lo status `Skip`, lo scheduler non eseguirà neanche la logica di Filter di questo Plugin per lo Scheduling Cycle in corso. Il che è desiderabile in quanto, se la soluzione non è presente, né l'Extension Point PreFilter, né l'Extension Point Filter necessitano di essere eseguiti.

Sempre dalle informazioni che vengono dai commenti dell'interfaccia `PreFilterPlugin`,

si evince che un esecuzione di successo di questo Extension Point dovrebbe avere come risultato un oggetto di tipo `PreFilterResult`. Esso è un tipo di dato che contiene il campo `NodeNames` di tipo `Set[string]` che serve a indicare allo scheduler quale sottoinsieme di Nodi deve essere considerato nella successiva fase di Filter. Questo meccanismo è pensato per abbassare la complessità computazionale dello scheduling, limitando il numero di Nodi da considerare per l'assegnamento di un Pod.

Lo scheduler quindi, eseguiti tutti i Plugin di PreFilter, unisce i sottoinsiemi di Nodi ottenuti e su essi esegue l'Extension Point Filter. Il compito del Plugin in questa fase, è quello di assicurarsi che il Nodo scelto dal solver per l'assegnamento di un Pod nel precedente Scheduling Cycle, sia inserito nell'insieme di Nodi che verrà valutato dallo scheduler.

Per fare quanto appena descritto, il Plugin cerca di capire innanzitutto se lo Scheduling Cycle in corso coinvolge uno dei Pod precedentemente espulsi. Per farlo consulta la `map` dei Pod salvata nello stato e, nel caso sia presente, ne ottiene l'`index`. Con tale `index` si può consultare la lista della soluzione per controllare se il Pod deve ancora essere spostato su uno specifico Nodo. Se così fosse, dall'arrray della soluzione si ottiene l'`index` del Nodo su cui spostarlo, da cui si ottiene il nome del Nodo, sempre tramite la `map` nello stato del Plugin. A questo punto non resta che inserire il nome del Nodo nell'apposito campo dell'oggetto `PreFilterResult` e ritornarlo.

È possibile che più Pod dello stesso tipo siano presenti nel cluster per effetto di un ReplicaSet. In questo modo si possono presentare più casi: solo uno di essi è da spostare, diversi di essi sono da spostare o tutti questi Pod sono da spostare. Per gestire questa situazione il Plugin in questo Extension Point si occupa di essi sequenzialmente. Scorre le occorrenze di un determinato tipo di Pod sulla `map` e nella soluzione, se la prima occorrenza non era da spostare, si passa alla successiva.

Filter

Infine, si esegue l'Extension Point Filter. Anche in questo caso l'esecuzione è vincolata alla presenza della soluzione nello stato del Plugin. Cosa che dovrebbe essere

garantita anche dal corretto funzionamento del Plugin in PreFilter e del suo valore di ritorno `Skip`.

La logica di questo metodo è quella di garantire che il Pod da spostare sia assegnato al Nodo scelto dallo script. Per farlo, si controlla, similmente a come fatto nell'Extension Point precedente, se si sta trattando l'accoppiata Nodo-Pod designata. Infatti, come visto in precedenza, questa interfaccia analizza Nodo per Nodo la compatibilità con un Pod.

I controlli di Nodi e Pod avvengono consultando le `map` salvate nello stato del Plugin. Da esse, come già visto, si ottengono gli `index` del Nodo e del Pod. Coi quali si può consultare la soluzione per verificare se il Pod deve essere spostato su quel Nodo. In caso affermativo si ritorna lo status `Success`, altrimenti `Unschedulable`.

Se in questo modo si è applicata parte della soluzione, eseguendo uno spostamento, ciò deve essere rispecchiato nella lista della soluzione. Per farlo si aggiorna l'array nella posizione del Pod al valore 0, che indica che il Pod si trova già nel Nodo previsto. È proprio a seguito di questo aggiornamento che il Plugin potrebbe aver esaurito le operazioni da svolgere per applicare la soluzione. Infatti, proprio in seguito a questa modifica, si verifica se la lista della soluzione è composta da tutti 0. In caso affermativo, prima del `return`, si disattiva il Plugin riportandolo allo stato iniziale. La disattivazione prevede la cancellazione dallo stato della soluzione. In questo modo il Plugin è pronto ad essere riattivato in un successivo Scheduling Cycle, se necessario.

4.3.3 Limitazioni del Plugin

Il Plugin, come già detto, è una proof of concept. Questo vuol dire che non è pensato per essere eseguito in ambienti production. Infatti, le limitazioni più evidenti di questo Plugin sono legate alla sua esecuzione su più Scheduling Context. Da questa restrizione non derivano solo problemi di performance, ma anche la fragilità del funzionamento. È infatti molto delicato l'equilibrio su cui questo Plugin opera.

Per evitare di incorrere in problemi infatti, è necessario che durante la fase di attività

del Plugin, ovvero quella nella quale la soluzione viene applicata, non vengano creati nuovi Pod. In caso contrario il comportamento è imprevedibile e si riduce ad una race condition. Se il Pod creato in quel delicato momento venisse assegnato ad un Nodo, occupando le risorse che erano state liberate per uno spostamento, il Plugin non potrebbe terminare la sua esecuzione con successo e si troverebbe in uno stato inconsistente.

Un altro caso nel quale il Plugin potrebbe trovarsi in uno stato inconsistente è quello in cui, durante l'applicazione della soluzione, uno o più Pod, di quelli da spostare, vengano eliminati dal cluster. Il Plugin riuscirebbe ad applicare correttamente la soluzione per quanto riguarda i Pod rimasti, ma attenderebbe invano gli Scheduling Context di quei Pod che sono stati eliminati. Il plugin rimarrebbe quindi attivo e non potrebbe essere eventualmente riattivato in uno Scheduling Cycle successivo se necessario.

L'unico modo di evitare questo tipo di problemi è quello di eseguire il Plugin solo in un ambiente controllato. La soluzione ideale a queste problematiche sarebbe l'implementazione dei meccanismi di Cross-Node Preemption da parte degli sviluppatori di Kubernetes. In questo modo l'intera logica del Plugin potrebbe essere eseguita in maniera atomica, ovvero in un unico Scheduling Cycle, e non sarebbe necessario spezzare l'applicazione della soluzione in più cicli di scheduling. È infatti questa dilatazione nelle operazioni in più fasi dello scheduler che rende il Plugin poco robusto.

Infine, come è risultato chiaro nell'analisi sull'adozione dello Scheduler Framework, i designer di Kubernetes hanno voluto unificare lo sviluppo di Plugin con lo scheduler stesso, vincolando i programmatori all'utilizzo del codice `Go`. Non è sicuramente una pratica consueta quella di utilizzare uno script `python` per l'esecuzione di un Plugin. Tanto meno l'installazione delle sue dipendenze all'interno dell'immagine dello scheduler. Questa scelta però deriva dal fatto che la libreria OrTools non è disponibile per il linguaggio `Go`, nonostante siano entrambi sviluppati da Google. Inoltre, non si è trovato un pacchetto `Go` espressivo come lo è OrTools; col conseguente risultato che il semplice re-writing dello script in `Go` non sarebbe stata una strada percorribile.

5 Sintesi e prospettive

Dopo essere giunti all'implementazione del Plugin non rimane che concludere con un'analisi dei risultati ottenuti e delle possibilità di sviluppi futuri dell'intero lavoro.

5.1 Conclusioni

Alcuni dei primissimi obiettivi di questa tesi purtroppo non sono stati raggiungi a causa degli ostacoli che si sono incontrati lungo il cammino e che sono riportati nella trattazione. Alcuni di essi però, come i problemi della Cross-Node Preemption, sono diventati obiettivi secondari di questa tesi. E questo si può già considerare un obiettivo raggiunto in quanto l'utilizzo di un solver potrebbe essere la base di un'implementazione della logica di questa feature.

Un altro ostacolo all'implementazione è stata la scarsa documentazione che ha reso lo sviluppo lento e faticoso. È da qui che si è deciso di cogliere l'occasione per dare un'ulteriore documento alla community che potesse aiutare altri programmatori ad approcciarsi a un progetto simile a questo. Non si tratta propriamente di un tutorial, ma più di un diario di bordo nel quale si è tenuta traccia, come in un log, delle informazioni trovate, delle azioni eseguite, del loro effetto e, a volte, anche della loro correzione.

Nonostante non sia ancora utilizzabile fuori da ambienti controllati, il Plugin creato riesce a mostrare le potenzialità dell'idea che vuole portare. Ci sono criticità sulle prestazioni e sulla robustezza del Plugin, ma l'aspirazione di poter vedere l'utilizzo

di un solver nell'algoritmo di preemption tra Nodi è stata raggiunta con successo. Il che è proprio l'obiettivo che ci si aspetta sia conseguito da una proof of concept. Per questo motivo il risultato non si può dire completo, ma è sicuramente soddisfacente.

5.2 Lavori futuri

Esistono alcune possibilità di evoluzione di questo lavoro, a cominciare dalle limitazioni descritte nel capitolo precedente. Si potrebbe infatti pensare di rendere il Plugin più robusto cercando di individuare e gestire i casi limite che lo portano in uno stato inconsistente. Questo però, esula dagli obiettivi di questa tesi. Inoltre la cosa più opportuna da fare, come già accennato, sarebbe attendere che alcune feature vengano implementate dal team di Kubernetes.

Il lavoro di S. Canova su cui si basa questa tesi, conclude che il solver OrTools è migliore rispetto a Z3 per questo problema. Questo è il motivo per cui si è scelto di utilizzare il primo tramite lo script `python` in quanto non è disponibile per `Go`. Esiste però un'implementazione di Z3 per `Go` [14]. Si potrebbe pensare di utilizzare questa libreria per richiamare il solver direttamente in `Go` e quindi evitare l'utilizzo dello script `python` e tutte le problematiche ad esso legate.

Ci sono poi proposte di miglioramento sull'implementazione del solver stesso. Si può consultare la tesi di Simone Canova [3] per ulteriori dettagli su questo argomento.

Bibliografia

[1] *Assigning Pods to Nodes*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ (visitato il 05/10/2024).

[2] Atlassian. *Microservices vs. monolithic architecture*. Atlassian. URL: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith (visitato il 25/09/2024).

[3] Simone Canova. "Scheduling optimization in Kubernetes". Tesi di dott. URL: http://amslaurea.unibo.it/32040/.

[4] *Cluster Architecture*. Kubernetes. URL: https://kubernetes.io/docs/concepts/architecture/ (visitato il 07/10/2024).

[5] *Command line tool (kubectl)*. Kubernetes. URL: https://kubernetes.io/docs/reference/kubectl/ (visitato il 14/10/2024).

[6] *Containerization (computing)*. In: *Wikipedia*. Page Version ID: 1241445617. 21 Ago. 2024. URL: https://en.wikipedia.org/w/index.php?title=Containerization_(computing)&oldid=1241445617 (visitato il 03/10/2024).

[7] *crossnodepreemption · kubernetes-sigs/scheduler-plugins*. URL: https://github.com/kubernetes-sigs/scheduler-plugins/tree/master/pkg/crossnodepreemption (visitato il 15/10/2024).

[8] *Deployments*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visitato il 05/10/2024).

[9] *Docker: Accelerated Container Application Development*. 10 Mag. 2022. URL: https://www.docker.com/ (visitato il 03/10/2024).

[10] *Fern | SDKs and Docs for your API*. URL: https://buildwithfern.com/ (visitato il 02/10/2024).

[11] *Get Ubuntu | Download*. Ubuntu. URL: https://ubuntu.com/download (visitato il 16/10/2024).

[12] *glog package - github.com/golang/glog - Go Packages*. URL: https://pkg.go.dev/github.com/golang/glog (visitato il 11/10/2024).

[13] *Go (programming language)*. In: *Wikipedia*. Page Version ID: 1248866625. 1 Ott. 2024. URL: https://en.wikipedia.org/w/index.php?title=Go_(programming_language)&oldid=1248866625 (visitato il 10/10/2024).

[14] Mitchell Hashimoto. *mitchellh/go-z3*. original-date: 2017-03-03T02:53:46Z. 30 Set. 2024. URL: https://github.com/mitchellh/go-z3 (visitato il 18/10/2024).

[15] *Hypervisor*. In: *Wikipedia*. Page Version ID: 1246817202. 21 Set. 2024. URL: https://en.wikipedia.org/w/index.php?title=Hypervisor&oldid=1246817202 (visitato il 02/10/2024).

[16] Ferenc Hámori. *The History of Kubernetes on a Timeline*. RisingStack Engineering. 20 Giu. 2018. URL: https://blog.risingstack.com/the-history-of-kubernetes/ (visitato il 25/09/2024).

[17] Scott M. Fulton III. *What Led Amazon to its Own Microservices Architecture*. The New Stack. 8 Ott. 2015. URL: https://thenewstack.io/led-amazon-microservices-architecture/ (visitato il 25/09/2024).

[18] *index | Alpine Linux*. URL: https://www.alpinelinux.org/ (visitato il 16/10/2024).

[19] *Jolie, the service-oriented programming language*. URL: https://www.jolie-lang.org/ (visitato il 02/10/2024).

[20] *klog package - k8s.io/klog/v2 - Go Packages*. URL: https://pkg.go.dev/k8s.io/klog/v2#section-readme (visitato il 11/10/2024).

[21] *kube-scheduler*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ (visitato il 11/10/2024).

[22] *Kubernetes Components*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/overview/components/ (visitato il 05/10/2024).

[23] *Kubernetes Deep Dive: Code Generation for CustomResources*. URL: https://www.redhat.com/en/blog/kubernetes-deep-dive-code-generation-customresources (visitato il 12/10/2024).

[24] *Kubernetes Logging*. GitHub. URL: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-instrumentation/logging.md (visitato il 11/10/2024).

[25] *Kubernetes Scheduler*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ (visitato il 05/10/2024).

[26] *kubernetes-sigs/scheduler-plugins*. original-date: 2020-01-16T13:34:00Z. 6 Ott. 2024. URL: https://github.com/kubernetes-sigs/scheduler-plugins (visitato il 08/10/2024).

[27] Torgeir Lebesbye et al. "Boreas – A Service Scheduler for Optimal Kubernetes Deployment". In: *Service-Oriented Computing*. A cura di Hakim Hacid et al. Cham: Springer International Publishing, 2021, pp. 221–237. ISBN: 978-3-030-91431-8.

[28] *Linux Containers*. URL: https://linuxcontainers.org/ (visitato il 03/10/2024).

[29] Vlad-Ioan Luca e Mădălina Eraşcu. "SAGE - A Tool for Optimal Deployments in Kubernetes Clusters". In: *2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2023, pp. 10–17. DOI: 10.1109/CloudCom59040.2023.00016.

[30] *Make - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/make/ (visitato il 13/10/2024).

[31] *Microservice architecture is agile software architecture*. InfoWorld. URL: https://www.infoworld.com/article/2256956/microservice-architecture-is-agile-software-architecture.html (visitato il 04/10/2024).

[32] *Microservices*. In: *Wikipedia*. Page Version ID: 1249134393. 3 Ott. 2024. URL: https://en.wikipedia.org/w/index.php?title=Microservices&oldid=1249134393 (visitato il 03/10/2024).

[33] *Minikube*. minikube. URL: https://minikube.sigs.k8s.io/docs/ (visitato il 13/10/2024).

[34] *Namespaces*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visitato il 05/10/2024).

[35] Alessandro Neri. *AlleNeri/scheduler-plugins*. original-date: 2024-05-24T14:04:32Z. 26 Lug. 2024. URL: https://github.com/AlleNeri/scheduler-plugins (visitato il 17/10/2024).

[36] *Nodes*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visitato il 05/10/2024).

[37] *Open Container Initiative - Open Container Initiative*. URL: https://opencontainers.org/ (visitato il 03/10/2024).

[38] *OR-Tools*. Google for Developers. URL: https://developers.google.com/optimization (visitato il 16/10/2024).

[39] *Orchestration (computing)*. In: *Wikipedia*. Page Version ID: 1247618088. 25 Set. 2024. URL: https://en.wikipedia.org/w/index.php?title=Orchestration_(computing)&oldid=1247618088 (visitato il 05/10/2024).

[40] Manjula Piyumal. *Building a Custom Kubernetes Scheduler Plugin: Scheduling Based on Pod-Specific Node Affinity*. Medium. 16 Set. 2024. URL: https://blog.stackademic.com/building-a-custom-kubernetes-scheduler-plugin-scheduling-based-on-pod-specific-node-affinity-7f66b6c607f9 (visitato il 10/10/2024).

[41] *Pod Priority and Preemption*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/ (visitato il 05/10/2024).

[42] *Pod Priority and Preemption*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/ (visitato il 15/10/2024).

[43] *Podman*. URL: https://podman.io/ (visitato il 03/10/2024).

[44] *Pods*. Kubernetes. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visitato il 05/10/2024).

[45] Julio Renner. *K8S- Creating a kube-scheduler plugin*. Medium. 26 Lug. 2021. URL: https://medium.com/@juliorenner123/k8s-creating-a-kube-scheduler-plugin-8a826c486a1 (visitato il 10/10/2024).

[46] *ReplicaSet*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visitato il 05/10/2024).

[47] *Scheduler Configuration*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/reference/scheduling/config/ (visitato il 07/10/2024).

[48] *Scheduler Plugin Documentation*. GitHub. URL: https://github.com/kubernetes-sigs/scheduler-plugins/blob/master/doc/develop.md (visitato il 10/10/2024).

[49] *Scheduling Framework*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (visitato il 25/09/2024).

[50] *Scheduling Framework - Design Proposal*. GitHub. URL: https://github.com/kubernetes/enhancements/blob/master/keps/sig-scheduling/624-scheduling-framework/README.md (visitato il 09/10/2024).

[51] *Scheduling Policies*. Kubernetes. Section: docs. URL: https://kubernetes.io/docs/reference/scheduling/policies/ (visitato il 07/10/2024).

[52] *Scheduling queue in kube-scheduler*. GitHub. URL: https://github.com/kubernetes/community/blob/f03b6d5692bd979f07dd472e7b6836b2dad0fd9b/contributors/devel/sig-scheduling/scheduler_queues.md (visitato il 08/10/2024).

[53] *Serverless computing*. In: *Wikipedia*. Page Version ID: 1249321790. 4 Ott. 2024. URL: https://en.wikipedia.org/w/index.php?title=Serverless_computing&oldid=1249321790 (visitato il 04/10/2024).

[54] *Computer Lib/Dream Machines*. In: *Wikipedia*. Page Version ID: 1193608886. 4 Gen. 2024. URL: https://en.wikipedia.org/w/index.php?title=Computer_Lib/Dream_Machines&oldid=1193608886 (visitato il 01/10/2024).

[55] *The Go Programming Language*. URL: https://go.dev/ (visitato il 10/10/2024).

[56] *Virtual machine*. In: *Wikipedia*. Page Version ID: 1248208291. 28 Set. 2024. URL: https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=1248208291 (visitato il 02/10/2024).

[57] *Virtualization*. In: *Wikipedia*. Page Version ID: 1247355540. 23 Set. 2024. URL: https://en.wikipedia.org/w/index.php?title=Virtualization&oldid=1247355540 (visitato il 03/10/2024).