



Universitat
de les Illes Balears

TREBALL DE FI DE GRAU

ANÀLISI EXHAUSTIVA DE PLANIFICACIONS D'EMPLAÇAMENTS DE SERVEIS AMB KUBERNETES

Andreu Garcia Coll

Grau d'Enginyeria Informàtica

Escola Politècnica Superior

Any acadèmic 2021-22

ANÀLISI EXHAUSTIVA DE PLANIFICACIONS D'EMPLAÇAMENTS DE SERVEIS AMB KUBERNETES

Andreu Garcia Coll

Treball de Fi de Grau

Escola Politècnica Superior

Universitat de les Illes Balears

Any acadèmic 2021-22

Paraules clau del treball: Edge Computing, Containers, Kubernetes, scheduler

Tutor: Isaac Lera Castro

Autoritz la Universitat a incloure aquest treball en el repositori
institucional per consultar-lo en accés obert i difondre'l en línia, amb
finalitats exclusivament acadèmiques i d'investigació

Autor/a		Tutor/a	
Sí	No	Sí	No
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Gràcies a mun pares que m'han recolçat sempre i ajudat a arribar aquí. Tampoc vull oblidar a sa meua parella que s'ha preocupat en tot moment de que tengués el temps per poder acabar aquest treball. Gràcies Duvi.

SUMARI

Sumari	iii
Acrònims	v
Resum	vii
1 Introducció	1
1.1 Objectiu del projecte	5
2 Coneixement actual	7
2.1 Virtualització per contenidors	7
2.2 Eina Docker	8
2.3 Kubernetes	9
2.3.1 Entendre Kubernetes	9
2.3.2 Arquitectura del Pla de Control	11
2.3.3 Arquitectura del Pla de Càrrega de Treball	12
2.4 El component: <i>kube-scheduler</i>	13
2.4.1 Marc de Treball de Planificació	14
3 Accions per a la modificació de la selecció i emplaçament de pods/contenidors	19
3.1 Una breu història de la planificació a Kubernetes	20
3.2 Mètodes d'emplaçaments bàsics per a la gestió dels Pods	22
3.2.1 Assignació de Pods a Nodes	23
3.2.2 Un planificador a mida	29
3.3 Configurar múltiples perfils d'assignació	31
3.4 Múltiples planificadors	31
3.4.1 Desplegar un nou planificador	34
3.5 Construint el teu propi planificador	38
3.5.1 Mètriques	38
3.5.2 Llenguatge de Programació	39
3.5.3 Comunitat <i>Kubernetes-SIGs</i>	40
3.5.4 Construcció	41
3.6 Implantació del planificador segons distribució i plataforma	41
3.7 Eines d'avaluació del comportament del planificador	44
4 Construcció d'un connector	49
4.1 Motivació	49

4.2	Creació d'un connector personalitzat	50
4.2.1	Desplegament del connector amb un segon planificador	58
4.2.2	Comprovació del funcionament del connector al segon planifi- cador	61
5	Discussió	65
6	Conclusió	71
6.1	Opinió personal de l'evolució del TFG	71
6.1.1	Opinió Personal	73
A	Apèndixs	75
A.1	Selector de Nodes. Exemple	75
	Bibliografia	77

ACRÒNIMS

IoT Internet of Things	1
VM Virtual Machine	4
LXC LinuX Containers	4
CI/CD Continuous Integration/Continuous Delivery	9
AWS Amazon Web Services	4
K8s Kubernetes	vii
IP Internet Protocol	10
DNS Domain Name Service	10
CNCF Cloud Native Computing Foundation	20
CRI Container Runtime Interface	13
VR Virtual Reality	2
AR Augmented Reality	2
ML Machine Learning	19
DL Deep Learning	20

API Application Programming Interface	vii
KEP Kubernetes Enhancement Proposals	49
SIG Special Interest Group	49
GPU Graphics Processing Unit	65
TPU Tensor Processing Unit	65
NPU Neural Processing Unit	65
SO Sistema Operatiu	43
SPOF Single Point of Failure	67
CRD Custom Resource Definitions	45
TFG Treball Final de Grau	72

RESUM

L'aparició de nous paradigmes de computació com el Perimetral o Boira on els serveis de computació es podem distribuir al llarg de tota la infraestructura TIC, inclosa la xarxa de comunicacions, han donat lloc a una nova necessitat de poder controlar l'emplaçament de aplicacions en aquestes infraestructures. Eines com a *Kubernetes*, un orquestrador de contenidors, ha estat de les primeres en prestar servei a aquest paradigma.

Kubernetes decideix a on desplegar els components d'una aplicació, que estaran integrats en contenidors, en Nodes del clúster. Aquesta decisió se basa en una operació de tres passes. A la primera fase executa una sèrie de comprovacions per trobar els Nodes factibles. A la segona, classifica aquests Nodes assignant-li una puntuació en funció de diverses funcions de prioritat. A la darrera passa, s'informa a la Application Programming Interface (API) de qui ha estat el Node que ha obtingut la màxima puntuació i el responsable de totes aquestes operacions és el planificador.

L'objectiu plantejat en aquest projecte es fer un estudi detallat de les diferents opcions que tenim de particularitzar el planificador juntament amb una implementació d'un planificador totalment personalitzat que consideri criteris no funcionals dels Nodes per desplegar les nostres aplicacions.

Per aconseguir aquest objectiu s'ha proposat un conjunt de tasques que són desenvolupades al llarg de la documentació:

1. Anàlisi bàsica del contenidors, virtualització i eines de contenedorització.
2. Realitzam un estudi detallat de l'arquitectura de Kubernetes (K8s) i en particular del component planificador.
3. Descrivim els modes de funcionament del planificador.
4. Presentam les opcions que té el planificador per assignar les càrregues de treball als Nodes treballadors amb els seus pros i contres, juntament amb la definició d'uns criteris d'anàlisi. En total, es descriuen 8 maneres de procedir a la personalització del planificador.
5. Entre aquestes 8 maneres, es descriu i implementa detalladament un planificador totalment personalitzat.
6. Finalment, també s'expliquen les eines per a la seva monitorització.

INTRODUCCIÓ

Devers l'any 2000 va aparèixer el concepte de Computació al núvol o en anglès *Cloud Computing* [1]. Aquest model arquitectònic va ser immortalitzat per George Gilder al seu article d'octubre 2006 a la revista Wired titulat "The Information Factories" [2]. Les granges de servidors sobre les quals Gilder va escriure eren similars en la seva arquitectura a la Computació en malla (en anglès *Grid Computing*), però mentre que el *grid* és un sistema compost de subsistemes amb certa autonomia d'acció alhora que mantenen una interrelació continuada amb els altres components, aquest nou model de núvol s'estava aplicant als serveis d'Internet [3].

Aquesta tecnologia permetia i permet, contractar recursos computacionals sota demanda. Va suposar un canvi evolutiu significant a la tradicional arquitectura web de client-servidor, donen eines més flexibles per crear noves arquitectures adaptables als nous models de negoci i àgils amb la càrrega de peticions depenent de la implicació del client.

Durant més d'una dècada, la computació al núvol centralitzat s'ha considerat un estàndard com a plataforma de lliurament de Tecnologia de la Informació. Encara que la computació al núvol és omnipresent a les nostres vides, exemples evidents són: Google Drive, Netflix, Spotify, Gmail, etc el model d'emmagatzematge de les dades al núvol s'està tornant massa costós i lent per complir amb els requisits de l'usuari final i especialment amb els dispositius d'Internet de les Coses, en anglès Internet of Things (IoT) que produeixen dades en temps real i estan contextualitzades a una regió espacial molt concreta.

Aquestes noves aplicacions tenen uns requisits que portem al límit la infraestructura del núvol centralitzada o relativament escampada a certes regions i/o instal·lacions concretes. Per entendre la importància d'aquesta nova necessitat d'una nova arquitectura citarem uns exemples que justificaran la situació [4]:

- Vehicles autònoms: La efectivitat de l'execució d'accions que s'executen en temps real sense que hi hagi un desfasament pel temps d'espera de la resposta o ins-

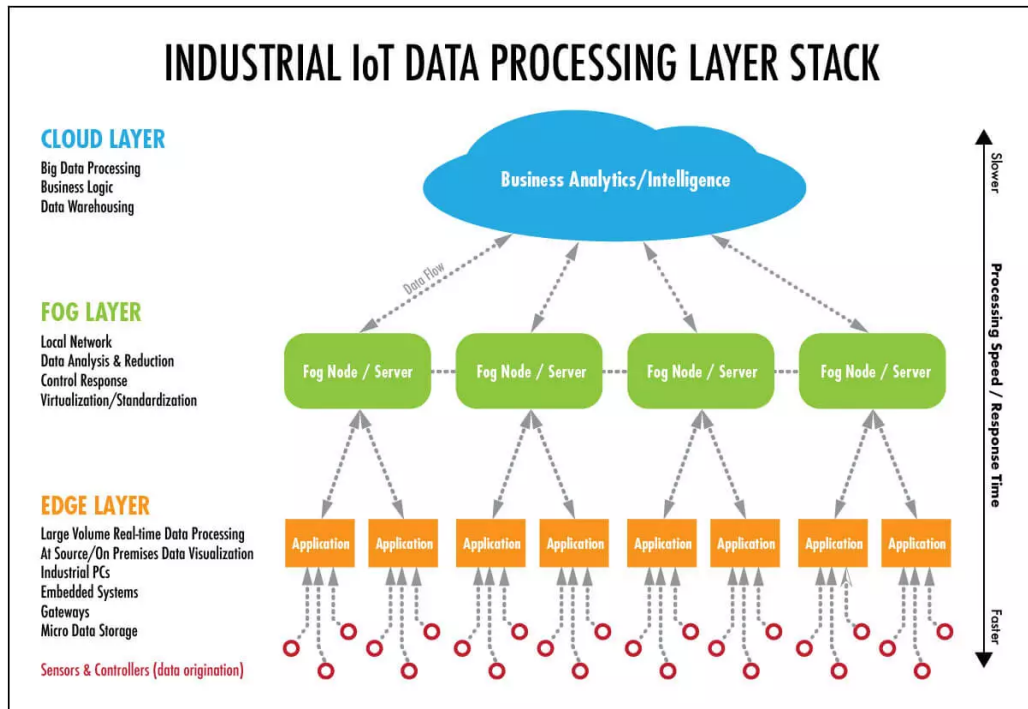


Figura 1.1: Representació de la computació al núvol, boira i al perímetre. Font: WINSYS-TEMS [8]

trucció d'un servidor.

- Sistemes de seguretat: Càlcul d'un sensor de moviment que pot estar integrat al mateix computador intern de la càmera, poder executar l'aplicació de detecció de moviment i enviar les imatges al servidor en el núvol.
- Dispositius de monitoreig mèdic: Amb capacitat d'oferir resposta en temps real per donar informació immediata després d'una monitorització, seguiment, supervisió o mesura, sense esperar respostes o notícies d'un servidor, sinó que compten amb el seu propi hardware.

Per tant, es necessiten noves condicions de disponibilitat i capacitat de serveis del núvol en llocs més específics i distribuïts per recolzar tant els requisits actuals (anàlisi de dades, serveis de xarxa) com les innovacions del demà (Ciutats intel·ligents (en anglès *Smart Cities*, Realitat Virtual/Realitat Augmentada (en anglès *Virtual Reality (VR)/Augmented Reality (AR)*)) [5].

Per millorar aquesta situació, va sorgir una nou paradigma que consisteix en tenir dispositius amb serveis de núvol més a prop on es generen les nostres dades. Aquests models es coneixen com: Computació a la boira [6] (en anglès *Fog Computing*) i Computació Perimetral [7] (en anglès *Edge Computing*). Aquest paradigma emergent està atraient un interès creixent, ja que aconseguim avantatges com a:

-
- Velocitat: Allibera amplada de banda de la xarxa per reduir el nivell de latència. La informació a tractar està a prop de l'usuari, pel que tenim una sèrie de conseqüències:
 - Les dades es distribueixen més ràpidament per a una millor experiència del client.
 - Permet tractar les dades en temps real que provenen dels dispositius en lloc d'enviar-los a través de llargs recorreguts per a que arribin a centres de dades i núvols de computació.
 - Millora l'agilitat de tots els serveis a més d'apropar els recursos informàtics als usuaris finals.
 - Seguretat: Si bé es parla de vulnerabilitats davant riscos de possibles atacs externs, atès que les seves infraestructures de seguretat no són tan desenvolupades com els grans centres de dades, sí és cert que com es redueix la informació que es transmesa per la xarxa, fa que les dades estiguin menys exposades, viatgin menys distància i es mantinguin sempre a prop del seu origen, preservant la informació que no necessita ser tramitada.

Paral·lelament a l'evolució d'aquests nous paradigmes de computació, es va començar a canviar el model monolític del disseny d'aplicacions. Un model tradicional d'una aplicació de programari que s'implementa i administra com una sola entitat, que és autònoma i independent d'altres aplicacions l'anomenem model monolític. Els canvis en una part de l'aplicació requereixen d'una redistribució de tota l'aplicació i, amb el temps, la manca de límits estrictes entre les parts dóna com a resultat l'augment de la complexitat i el conseqüent deteriorament de la qualitat de tot el sistema a causa del creixement. La figura 1.2 representa un exemple clar d'aquesta definició.

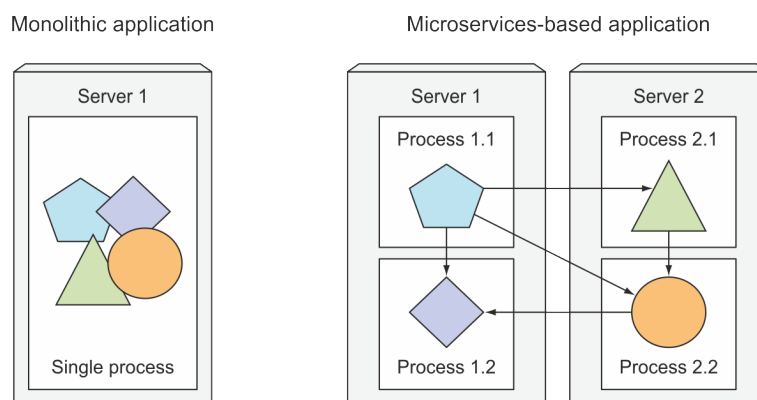


Figura 1.2: Components d'una aplicació monolítica en front dels microserveis. *Font: Manning Publications Co. [9]*

L'execució d'una aplicació monolítica generalment requereix una petita quantitat de servidors que puguin proporcionar recursos suficients per executar l'aplicació.

- *Escalament vertical*: Per fer front a l'augment de les càrregues al sistema, afegint més CPU, memòria i altres components del servidor. Generalment no requereix cap canvi en l'aplicació, es torna costosa amb relativa rapidesa i, a la pràctica, sempre té un límit superior.
- *Escalament horitzontal*: Configurant servidors addicionals per executar diverses còpies d'una aplicació poden requerir grans canvis en el codi de l'aplicació i no sempre és possible (p.e.: bases de dades relacionals).

Aquesta nova situació ha propiciat a començar a dividir aplicacions monolítiques complexes en components més petits que han evolucionat al concepte de *microservei*. Un *microservei* és un procés independent, i com a tal, és possible desenvolupar i implementar cadascun d'ells per separat. Un canvi en un, no requereix canvis o reimplementació de cap altre servei. Un exemple actual d'aquesta evolució del monòlit als microserveis és la va fer *Netflix* al 2009 quan la seva infraestructura no podia seguir el ritme de la demanda dels seus serveis de streaming de vídeo, que creixia a tota velocitat. En aquesta situació, l'empresa va decidir migrar la seva infraestructura de TI dels seus dos centres de dades privats a un núvol públic (Amazon Web Services (AWS)) i reemplaçar l'arquitectura monolítica per una altra de microserveis quan encara el terme de "*microservei*" no existia. Vàren ser més de vuit anys per a completar la migració i aquestes eines que ha fet servir les ha posades a l'abast de tothom com a codi obert ([Netflix Open Source Software](#)). Actualment, l'aplicació de *Netflix* compta amb més d'un miler de microserveis i els seus serveis s'executen en centenars de milers d'instàncies a AWS [10].

Quan una aplicació només es compon de quantitats més petites de components grans, és completament acceptable assignar una màquina virtual (Virtual Machine (VM) [11]) dedicada a cada component i aïllar els seus entorns proporcionant a cadascun d'ells la seva pròpia instància de sistema operatiu. En el cas dels microserveis, els desenvolupadors estan recorrent a les tecnologies de contenidors de Linux (Linux Containers (LXC) [12]). Li permeten executar múltiples serveis a la mateixa màquina host, mentre que no només exposen un entorn diferent per a cadascun, sinó que també els aïllen entre si, de manera similar a les màquines virtuals, però amb molta menys sobrecàrrega.

Com és de suposar, els microserveis també tenen inconvenients. Quan el nostre sistema consta només d'una petita quantitat de components implementables, administrar aquests components és fàcil. És trivial decidir on implementar cada component, perquè no hi ha tantes opcions. Quan augmenta la quantitat d'aquests components, les decisions relacionades amb la implementació esdevenen cada cop més difícils perquè no només augmenta la quantitat de combinacions d'implementació, sinó que la quantitat d'interdependències entre els components augmenta en un factor encara més gran. Per a resoldre aquest problema es va crear *Kubernetes*. *Kubernetes* (també coneguda com K8s o *kube*) és un sistema de programari que permet automatitzar molts del procesos manuals vinculats a la gestió, implementació i escalabilitat d'aplicacions en contenidors.

K8s no té perquè conèixer cap detall intern d'aquestes aplicacions i com aquestes s'executen dins contenidors, no afecten a altres aplicacions que s'executen al mateix servidor, reforçant l'aïllament complet entre aplicacions que tant important resulta

pel proveïdors del núvol públic que solen gestionar organitzacions diferents al mateix maquinari.

K8s executarà les aplicacions en contenidors en algun lloc del clúster, brindarà informació als seus components sobre com trobar-se entre si i els mantindrà en funcionament. Com que a les aplicacions no els importa a quin node s'està executant, K8s les pot reubicar en qualsevol moment i, aconseguir una utilització de recursos millor del que és podria fer amb la programació manual.

1.1 Objectiu del projecte

Kubernetes decideix a on desplegar els components d'una aplicació, que estaran integrats en contenidors, en Nodes del clúster. Aquesta decisió se basa en una operació de tres passes. A la primera fase executa una sèrie de comprovacions per trobar els Nodes factibles. A la segona, classifica aquests Nodes assignant-li una puntuació en funció de diverses funcions de prioritat. A la darrera passa, s'informa a la API de qui ha estat el Node que ha obtingut la màxima puntuació i el responsable de totes aquestes operacions és el planificador.

L'**objectiu** plantejat en aquest projecte es fer un estudi detallat de les diferents opcions que tenim de particularitzar el planificador juntament amb una implementació d'un planificador totalment personalitzat que consideri criteris no funcionals dels Nodes per desplegar les nostres aplicacions.

Per aconseguir aquest objectiu s'ha proposat un conjunt de tasques que són desenvolupades al llarg de la documentació:

1. Anàlisi bàsica del contenidors, virtualització i eines de contenedorització.
2. Realitzam un estudi detallat de l'arquitectura de K8s i en particular del component planificador.
3. Descrivim els modes de funcionament del planificador.
4. Presentam les opcions que té el planificador per assignar les càrregues de treball als Nodes treballadors amb els seus pros i contres.
5. Especificam detalladament les passes necessàries per implementar un planificador totalment personalitzat.
6. Explicam les eines per a la seva monitorització.
7. Comentam els criteris que poden obtenir-se per a estudiar-se.

CONeixement Actual

Avui en dia, termes com a *Computació al núvol*, *Docker*, *Kubernetes* o *kube-scheduler* són paraules que sentim sovint als entorns informàtics i que tenen com a denominador comú els contenidors.

Al llarg d'aquest capítol coneixerem de quina manera es despleguen les nostres aplicacions mitjançant eines que ens permeten empaquetar-les (*Docker*) i després emplaçar-les a ubicacions generalment disperses geogràficament (*Kubernetes*). Les tecnologies descrites i utilitzades en aquest projecte estan condicionades a l'ecosistema de Kubernetes. L'elecció de K8s és un requisit propi del projecte.

2.1 Virtualització per contenidors

Un contenidor és una unitat abstracta i executable de programari on s'empaqueta el codi d'aplicació, juntament amb les seves biblioteques i dependències, de forma comuna perquè es pugui executar a qualsevol lloc.

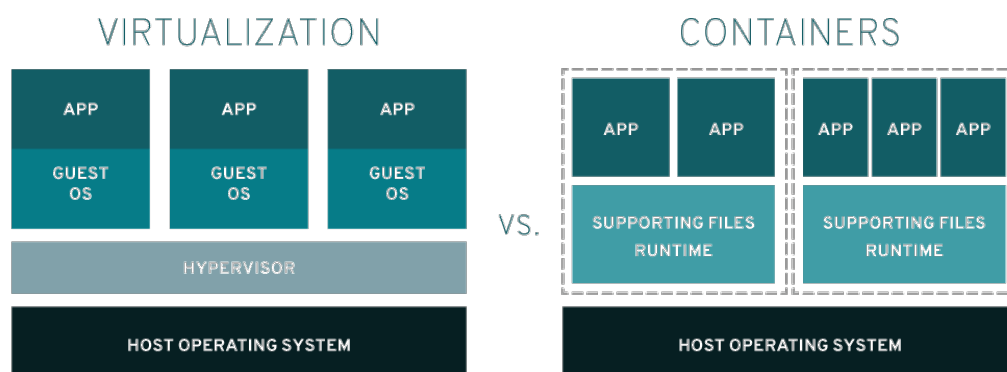


Figura 2.1: Diferències entre Màquines Virtuals i Containers. *Font: RedHat* [13]

A diferència dels contenidors, les VM [11] requereixen executar-se en un entorn d'hipervisor on a cada màquina es *virtualitza el maquinari (hardware) subjacent* a més d'incloure una rèplica completa d'un sistema operatiu per funcionar, juntament amb una aplicació i les seves biblioteques i dependències associades. A la Figura 2.1 podem veure clarament aquestes diferències.

Els contenidors en canvi, comparteixen el mateix nucli del sistema operatiu, pel que aquí es *virtualitza el sistema operatiu* [14]. Això implica que la gran majoria de vegades, un contenidor és molt més lleuger que una VM [15] .

2.2 Eina Docker

Docker és un projecte de codi obert creat amb la finalitat d'automatitzar el desplegament i execució d'aplicacions dins de contenidors, proporcionant així una capa addicional d'abstracció i automatització de virtualització d'aplicacions. Això permet als desenvolupadors realitzar l'empaquetatge de les nostres aplicacions juntament amb les dependències corresponents dins d'una unitats estandarditzades conegudes sota el terme de contenidors de programari.

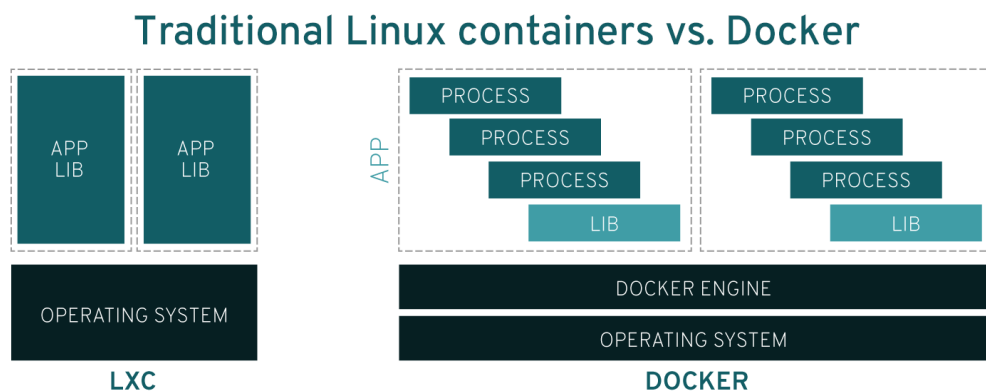


Figura 2.2: Contenidors tradicionals VS Contenidors Docker. *Font: RedHat* [16]

Els contenidors de Docker estan dissenyades a partir dels contenidors de Linux, i ofereixen als usuaris accés sense aplicacions, la possibilitat de fer implementacions en poc temps i el control sobre les versions i la seva distribució.

Els contenidors Docker es van dissenyar a partir de la tecnologia LXC [12]. La tecnologia Docker no només ofereix la capacitat per executar els contenidors, sinó que també en facilita la creació i el disseny, així com l'enviament i el control de versions de les imatges, entre altres funcions. Els contenidors tradicionals de Linux fan servir un sistema *init/Systemd* [17] per gestionar diversos processos, i permet que les aplicacions completes s'executin com una de sola. La tecnologia Docker divideixen les aplicacions en els seus processos individuals i ofereix les eines per fer-ho. Aquest enfocament de separació dels elements permet:

- Modularitat: Capacitat de separar una part de l'aplicació per actualitzar-la o reparar-la, sense deshabilitar l'aplicació completa.

- **Capes i control de versions d'imatges:** Cada fitxer d'imatge Docker està format per diverses capes que conformen una imatge, de tal manera que per agilitzar el disseny dels contenidors nous, reutilitzem les capes. A més, d'un control total de les imatges de contenidors cada vegada que es produeix una modificació.
- **Restauració:** Com totes les imatges compten amb capes, pel que ens permet restaurar a una versió anterior en cas de que nos ens agradi com ha quedat. Pel que des de la perspectiva de les eines dona suport a Continuous Integration/Continuous Delivery (CI/CD)[18].

Docker Inc. també és el nom de l'empresa que promou i impulsa aquesta tecnologia. Encara que hi ha diferents tipus d'implementacions de contenidors, ha esdevingut la solució estàndard de facto per al seu propòsit i ara per ara és el més famós motor per crear contenidors de programari.

2.3 Kubernetes

A mesura que les aplicacions varen anant fent-se més complexes per incloure contenidors distribuïts en diferents servidors, es varen començar a plantejar problemes; per exemple, com coordinar i programar diversos contenidors, com permetre la comunicació entre contenidors o com escalar instàncies de contenidor. Kubernetes es va crear per resoldre aquest tipus de problemes.

K8s és un sistema de codi obert per a l'automatització del desplegament, ajustament d'escala i maneig d'aplicacions en contenidors que va ser originalment dissenyat per Google i donat a la Cloud Native Computing Foundation (part de la Linux Foundation).

A K8s, les unitats de computació desplegables més petites que es poden crear i gestionar s'anomenen Pods. Un Pod és un grup d'un o més contenidors (com a contenidors Docker), amb emmagatzematge/xarxa compartits, i unes especificacions de com executar els contenidors. Els continguts d'un Pod sempre són còpiedats, coprogramesats i executats en un context compartit. Un Pod modela un host lògic específic de l'aplicació: conté un o més contenidors d'aplicacions relativament entrellaçats. Abans de l'arribada dels contenidors, executar-se a la mateixa màquina física o virtual significava ser executat al mateix host lògic [19].

2.3.1 Entendre Kubernetes

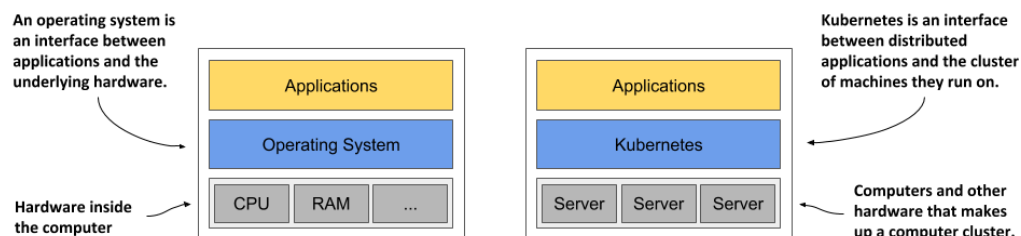


Figura 2.3: Kubernetes és per a un clúster d'ordinadors el que és un sistema operatiu per a un ordinador. Font: *Kubernetes in Action* [20]

Imaginem K8s com un sistema operatiu per al clúster. A la figura 2.3 s'il·lustra les analogies entre un sistema operatiu que s'executa en un ordinador i K8s que s'executa en un clúster d'ordinadors.

De la mateixa manera que un sistema operatiu planifica el processos a la CPU, i actua com a interfície entre l'aplicació i el maquinari de l'ordinador, Kubernetes programa els components d'una aplicació distribuïda en ordinadors individuals del clúster d'ordinadors subjacent i actua com a interfície entre l'aplicació i el clúster.

Entre les funcions principals de K8s podem destacar:

- **Planificació de nodes i contenidors:** Decideix a quin Node s'executarà cada contenidor, en funció dels recursos necessaris i d'altres restriccions. A més, podem barrejar càrregues de treball crítiques per tal de potenciar l'estalvi de recursos.
- **Abstracció de la infraestructura:** K8s gestiona els recursos dels que disposau. D'aquesta manera, els desenvolupadors es poden centrar a escriure codi d'aplicacions i oblidar-se de la infraestructura de computació, xarxes o emmagatzematge subjacent
- **Supervisió de l'estat dels serveis:** Revisa l'entorn en temps d'execució (en anglès *runtime* [21] ¹) i el compara amb l'estat que voleu. Du a terme comprovacions d'estat automàtiques als serveis i reinicia els contenidors que han fallat o s'han aturat, fent que els serveis estiguin disponibles quan estan en execució i llestos.
- **Descobriments de serveis i l'equilibri de càrrega:** K8s pot exposar un contenidor mitjançant el nom Domain Name Service (DNS) o la seva pròpia adreça Internet Protocol (IP). Si el trànsit cap a un contenidor és elevat, K8s pot equilibrar la càrrega i distribuir el trànsit de xarxa perquè el desplegament sigui estable.
- **Seguiment de l'assignació de recursos i els escala en funció de l'ús de la capacitat de procés.**
- **Comprovar l'estat dels recursos individuals i permet que les aplicacions es recuperin automàticament reiniciant o replicant els contenidors.**
- **Herència:** Implementació i gestió de les aplicacions heretades, desenvolupades al núvol, en contenidors i de microserveis.

Com podem veure a la figura 2.4, un clúster de Kubernetes consta d'un conjunt de màquines de treball, anomenades Nodes, que executen aplicacions en contenidors. Cada clúster té almenys un Node de treball. Els nodes de treball allotgen els Pods que com ja hem comentat abans, són els components de la càrrega de treball de l'aplicació. El *Pla de Control* gestiona els nodes de treball i els Pods del clúster. En entorns de producció, el pla de control sol executar-se en diversos ordinadors i un clúster sol executar diversos nodes, proporcionant tolerància a errors i alta disponibilitat.[23]

¹No confondre amb la fase del cicle de vida d'un programa en temps d'execució

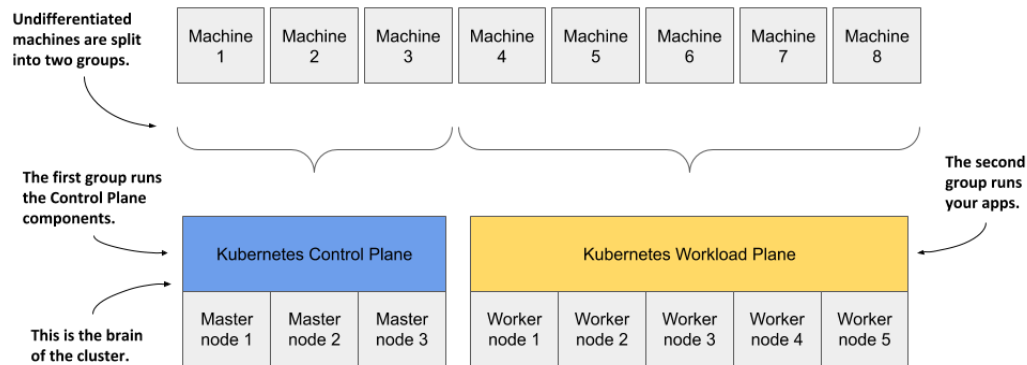


Figura 2.4: Desplegament d'un clúster de Kubernetes. Font: *Kubernetes in Action* [22]

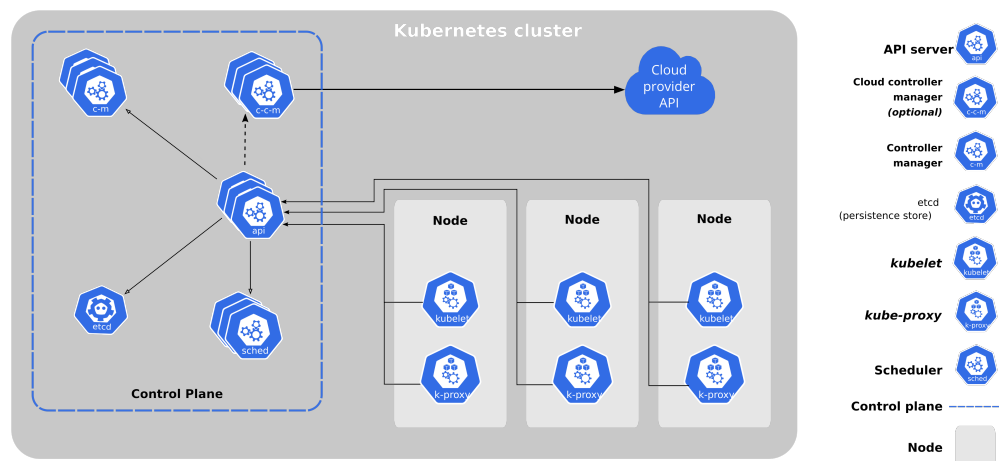


Figura 2.5: Els components d'un clúster. Font: *kubernetes.io* [24]

2.3.2 Arquitectura del Pla de Control

Els components del **Pla de Control** responen als esdeveniments del clúster i prenen decisions damunt ell. Controlen el clúster. Dins del Pla de Control, hi ha diferents entitats amb funcions específiques:

- **kube-apiserver**: És el teixit fonamental de K8s, el *front-end* del pla de control. Totes les operacions i comunicacions entre components i ordres d'usuari externs són trucades d'API REST que gestiona el servidor d'API. En conseqüència, tot el que hi ha a la plataforma K8s es tracta com un objecte API i té una entrada corresponent a l' API.
- **etcd**: És el magatzem de dades més important de K8s. Aquí, s'emmagatzema i replica tot l'estat del clúster: la configuració, especificacions i els estats de les càrregues de treball en execució. etcd s'utilitza com a *back-end* per al descobriment de serveis i emmagatzema l'estat del clúster i la seva configuració. etcd emmagatzema l'estat real del sistema i l'estat desitjat del sistema, després utilitza

la funcionalitat d'observació de etcd per monitoritzar els canvis en qualsevol d'aquestes dues coses. Si divergeixen, K8s fa els canvis necessaris per conciliar l'estat real i l'estat desitjat.

- **kube-scheduler:** És el procés del pla de control que assigna Pods als nodes. El planificador determina quins nodes són ubicacions vàlides per a cada Pod de la cua de planificació segons les restriccions i els recursos disponibles. Aleshores, el planificador classifica cada Node vàlid i lliga el Pod a un Node adequat. Es poden utilitzar diversos programadors diferents dins d'un clúster.

A grans trets, les passes bàsiques de *kube-scheduler* són:

1. Cerca Pods recent creats no assignats a cap Node (Pods no vinculats).
 2. Examina l'estat del clúster (emmagatzemat a la memòria cau).
 3. Tria un Node que tingui espai lliure i compleixi altres condicions.
 4. Uneix aquest Pod a un Node.
- **kube-controller-manager:** És el controlador dels controladors. Aquest component està format per diversos controladors que s'executen com un sol procés. Els controladors són bucles de control que miren contínuament l'estat del clúster i fan o solliciten canvis quan sigui necessari. Cada controlador intenta apropar l'estat del clúster actual a l'estat desitjat. Els controladors parlen contínuament amb el kube-apiserver i el kube-apiserver rep tota la informació dels nodes a través de *kubelet*.
 - **cloud-controller-manager:** El gestor del controlador de núvol ens permet enllaçar el nostre clúster a l'API del nostre proveïdor de núvol i separa els components que interactuen amb aquesta plataforma de núvol dels components que només interaccionen amb el nostre clúster. Executa controladors responsables d'interactuar amb la infraestructura subjacent d'un proveïdor de núvol quan els nodes no estan disponibles, de gestionar els volums d'emmagatzematge quan els proporciona un servei de núvol i de gestionar l'equilibri de càrrega i l'encaminament.

2.3.3 Arquitectura del Pla de Càrrega de Treball

El Pla de Càrrega de Treball, també anomenat *Worker Node* o simplement *Node*, s'encarrega de mantenir els Pods en execució i proporcionar l'entorn d'execució de K8s. Cadascun d'aquests components, són executats a cada *Worker Node* del clúster.

Els components són:

- **kubelet:** El *kubelet*, es l'agent que parla amb el servidor de l'API i gestiona les aplicacions que s'executen al seu node. És responsable de mirar el conjunt de Pods que estan vinculats al seu Node i assegurar-se que aquests Pods s'executen. A continuació, informa de l'estat a mesura que canvien les coses respecte a aquests Pods.
- **kube-proxy:** És un servidor intermediari de xarxa. Manté les regles de xarxa als nodes que permeten la comunicació de xarxa amb Pods des de dins o fora

del clúster. S'assegura que cada Pod tengui una adreça IP única. Fa possible que tots els contenidors d'un Pod comparteixin una única IP. Gestiona la traducció i l'encaminament de IP.

- **container runtime:** És el programari responsable d'executar els contenidors. Inicialment el que utilitzava era el de Docker però més endavant es va crear l'API Container Runtime Interface (CRI) que permetia integrar-se amb molts altres container runtimes de forma transparent.

2.4 El component: *kube-scheduler*

La planificació (en anglès *scheduling*), s'assegura que els Pods s'aparellin amb els Nodes per a que *kubelet* pugui executar-los.

kube-scheduler és el planificador nadiu de K8s i s'executa com un component més del pla de control.

Per a cada Pod de nova creació o altres Pods no planificat, *kube-scheduler* selecciona un Node òptim per a que s'executi. No obstant això, cada contenidor dels Pods té requisits de recursos diferents així com cada Pod. Aleshores, els Nodes s'han de filtrar segons requisits de planificació específics [25, pàgines 623-625].

kube-scheduler selecciona un Node per al Pod en una operació de tres passos esquematitzats a la figura 2.6:

- **Filtratge** (en anglès *Filtering*): En aquest procés *kube-scheduler* executa una sèrie de comprovacions per trobar els *Nodes factibles*, que és el conjunt de Nodes que compleixen una sèrie de restriccions donades. Sovint n'hi haurà més d'un. Si el conjunt és buit, aquest Pod es considerarà no planificable (en anglès *unschedulable*) i encara no es podrà planificar.
- **Puntuació** (en anglès *Scoring*): Una vegada filtrat, el planificador classifica els Nodes restants per trobar la ubicació més adequada del Pod. El Node passa per diverses funcions de prioritat i el planificador li assigna una puntuació. D'aquests *Nodes factibles*, aquest amb la puntuació més alta s'erigirà finalment com a *Node viable*. Cada funció li assigna una puntuació entre 0 i 10 a on 0 és la més baixa i 10 la més alta.
- **Assignació** (en anglès *Binding*): Finalment, el planificador informa al servidor de la API sobre el Node que s'ha seleccionat (el que ha obtingut la puntuació més alta). Si hi ha diversos nodes amb la mateixa puntuació el planificador tria un Node aleatori i aplica efectivament un desempat.

El comportament del *kube-scheduler* es pot personalitzar escrivint un arxiu de configuració i passant-ne la ruta com a argument de línia d'ordres. Un perfil de planificació ens permet configurar les diferents etapes de la planificació. Cada etapa s'exposa en un punt d'extensió. Els connectors (en anglès *plugins*) ens proporcionen els comportaments de la planificació mitjançant la implementació d'un o més d'aquests punts d'extensió. Per entendre un poc més els perfils de planificació hem de conèixer un

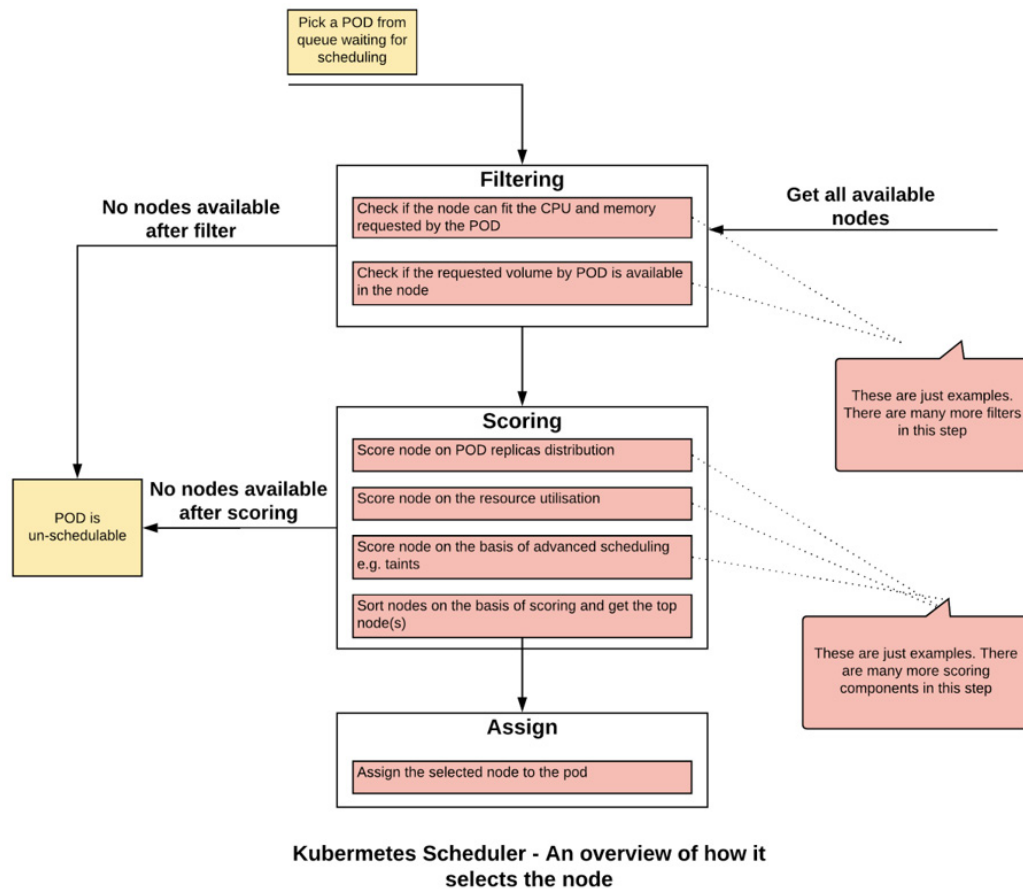


Figura 2.6: Una visió general de com el *kube-scheduler* selecciona un Node adequat.
Font: *The Kubernetes Workshop* [26]

les etapes per les que passa el Marc de Treball de Planificació (en anglès *Scheduling Framework*).

2.4.1 Marc de Treball de Planificació

És una arquitectura connectable per al *kube-scheduler*. Afegeix un nou conjunt de connectors de la API al ja existent planificador. Alguns d'aquests connectors canviaran les decisions del planificador i d'altres només són informatius.

Cada intent de planificar un Pod es divideix en dues fases: el **Cicle de Planificació** (en anglès *Scheduling Cycle*) i el **Cicle de Vinculació** (en anglès *Binding Cycle*).

El cicle de planificació selecciona un Node per al Pod i el cicle de vinculació aplica aquesta decisió al clúster. Un cicle de Planificació i un cicle de Vinculació s'anomena "Context de Planificació" (en anglès *Scheduling Context*).

Els cicles de planificació s'executen en sèrie mentre que els cicles de vinculació es poden executar simultàniament. Si es determina que un Pod no es planificable o si hi ha un error, el cicle de planificació o de vinculació es poden cancel·lar, el Pod tornarà a la cua i es tornarà a provar. A la figura 2.7 es visualitzen tots dos cicles. En color verd

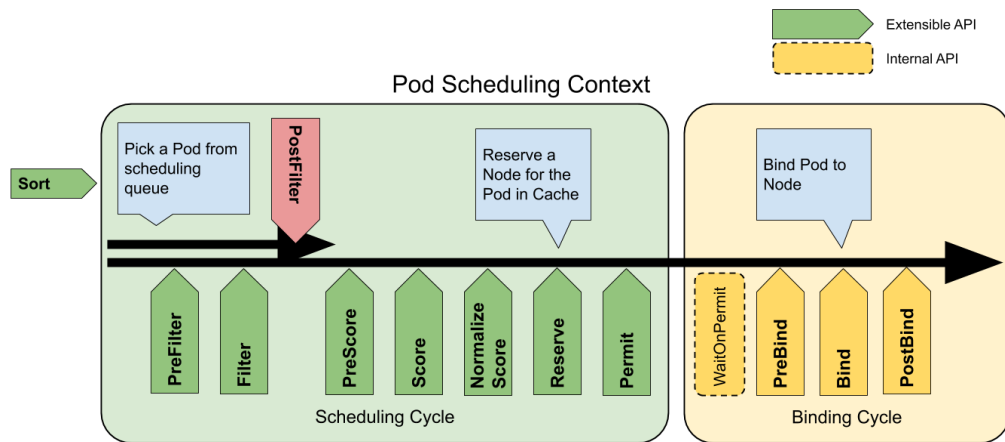


Figura 2.7: Context de Planificació. Punts d'extensió pels que pot passar el Pod. Font: *kubernetes.io* [27]

els passos corresponents al cicle de planificació i en color groc els passos del cicle de vinculació. Aquests passos o fases es denominen punts d'extensió [28] i és aquí on es registren els connectors dels usuaris per a ser connectats per si volen canviar aquest comportament. Els punts d'extensió són els següents²:

- **queueSort** : Ordena els Pods pendents a la cua de planificació. Només pot haver habilitat un únic connector a la vegada.
- **preFilter** : Comprova la informació sobre un Pod o el clúster abans de filtrar-lo.
- **filter** : Filtra els nodes que no poden executar el Pod. Els filtres es criden en l'ordre configurat.
- **postFilter** : Es criden en l'ordre configurat quan no s'ha trobat nodes viables per al Pod. Si qualche connector **postFilter**, marca el Pod com a planificable els connectors restants no es criden.
- **preScore** : Punt d'extensió informatiu.
- **score** : Proporciona la puntuació a cada Node que ha passat el filtratge.
- **reserve** : Punt d'extensió informatiu que notifica als connectors quan s'ha reservat recursos per a un determinat Pod.
- **permit** : Aquests connectors poden prevenir o retardar la vinculació d'un Pod.
- **preBind** : Realitza qualsevol treball necessari abans que un Pod estigui vinculat.

²Per a facilitar la lectura i interpretació s'ha mantingut un subratllat amb el color pertinent del punt d'extensió conservant aquest estil com es fa a la documentació oficial.

- **bind** : Uneix un Pod amb un Node. Aquests connectors es criden en ordre i un cop s'ha fet el vincle, els connectors restants es boten. Es requereix almenys un connector de vinculació.
- **postBind** : Punt d'extensió informatiu que es crida després d'haver vinculat un Pod.

Hi ha habilitats uns connectors per defecte [29] que implementen un o més d'aquests punts d'extensió. Aquests connectors, es poden desactivar i activar els nostres propis.

La relació de connectors que venen habilitats per defecte que inclou el planificador són prop d'una vintena. A continuació, es comenten aquests connectors sense aprofundir gaire en detalls:

- **ImageLocality** : Afavoreix als Nodes que ja tenen les imatges de contenidor que executa el Pod. Punt d'extensió: **score** .
- **TaintToleration** : Implementa *Taints i Toleràncies* (vegeu 7). Punts d'extensió: **filter** , **preScore** , **score** .
- **NodeName** : Comprova si un nom de Node de l'especificació del Pod coincideix amb el Node actual. Punt d'extensió: **filter** .
- **NodePorts** : Comprova si un Node té ports lliures per als ports de Pod sol·licitats. Punts d'extensió: **preFilter** , **filter** .
- **NodeAffinity** : Implementa *selectors de Nodes* (vegeu 2) i *afinitat de Nodes* (vegeu 3). Punts d'extensió: **filter** , **score** .
- **PodTopologySpread** : Implementa *Restriccions de repartiment de la topologia del Pod* (vegeu 6). Punts d'extensió: **preFilter** , **filter** , **preScore** , **score** .
- **NodeUnschedulable** : Filtra els Nodes que tenen l'especificació de **Unschedulable** posada a *true* . Punt d'extensió: **filter** .
- **NodeResourcesFit** : Comprova si el Node té tots els recursos que el Pod li demana. Punts d'extensió: **preFilter** , **filter** , **score** .
- **NodeResourcesBalancedAllocation** : Afavoreix els Nodes que obtindrien un ús de recursos més equilibrat si el Pod s'assigna allà. Punt d'extensió: **score** .
- **VolumeBinding** : Comprova si el Node té o pot enllaçar els volums sol·licitats. Punts d'extensió: **preFilter** , **filter** , **reserve** , **preBind** , **score** .
- **VolumeRestrictions** : Comprova que els volums muntats al Node satisfan les restriccions específiques del proveïdor de volums. Punt d'extensió: **filter** .
- **VolumeZone** : Comprova que els volums sol·licitats satisfan els requisits de zona que puguin. Punt d'extensió: **filter** .

- `NodeVolumeLimits` : Comprova que els límits de volum CSI poden ser satisfets per al Node. Punt d'extensió: `filter` .
- `EBSLimits` : Comprova que els límits de volum AWS EBS poden ser satisfets per al Node. Punt d'extensió: `filter` .
- `GCEPDLimits` : Comprova que els límits de volum GCP-PD poden ser satisfets per al Node. Punt d'extensió: `filter` .
- `AzureDiskLimits` : Comprova que els límits de volum Azure poden ser satisfets per al Node. Punt d'extensió: `filter` .
- `InterPodAffinity` : Implementa *Afinitat i Antiafinitat dels Pods* (vegeu 4). Punts d'extensió: `preFilter` , `filter` , `preScore` , `score` .
- `PrioritySort` : Proporciona l'ordenació per defecte basada en la prioritat. Punt d'extensió: `queueSort` .
- `DefaultBinder` : Proporciona el mecanisme d'enllaç per defecte. Punt d'extensió: `bind` .
- `DefaultPreemption` : Proporciona el mecanisme de preferència per defecte. Punt d'extensió: `postFilter` .
- `SelectorSpread` : Afavoreix la propagació a través dels Nodes per als Pods que pertanyen a *Services*, *ReplicaSets* i *StatefulSets*. Punts d'extensió: `preScore` , `score` .
- `CinderLimits` : Comprova que els límits de volum OpenStack Cinder poden ser satisfets per al Node. Punt d'extensió: `filter` .

ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

Les grans empreses aposten per K8s com a eina d'administració i gestió de les seves aplicacions, essent aquestes tan diverses que ens podem trobar des de l'administració d'una pàgina web fins altres tan dispars com la computació perimetral, dades massives o macrodades (en anglès *Big Data*), aprenentatge automàtic (en anglès Machine Learning (ML)), per citar-ne un parell.

Essent el Pod sa unitat més petita que és pot crear a K8s, i que no es té en compte el contingut del Pod (el contenidor) a l'hora de decidir el planificador a quin Node el col·locarà, ens podem arribar qüestionar si realment gestiona correctament la ubicació d'aquests Pods als Nodes adequats, de quina manera ho fa, i si *kube-scheduler*¹ disposa de les estratègies necessàries per a poder modificar aquest emplaçaments per a qualsevol tipus de càrrega de treball.

Aquesta necessitat de poder aprofundir sobre les estratègies d'emplaçament es fa més apressant als paradigmes dels *núvols híbrids* i els *multi núvols*, que es comença a veure com una tendència i bona senyal pel futur empresarial [30]. També, no és estrany trobar-nos cada dia més empreses que per temes de seguretat i solidesa tinguin més d'un clúster i que cada un d'ells estiguin allotjats a diferents proveïdors al núvol. Distribuir la càrrega de treball a diferents clústers és una solució intel·ligent, els clústers més petits són més fàcils d'administrar, si fallessin, el radi d'acció de les aplicacions i el usuari afectats es limitat però una assignació estàtica d'usuaris o conjunts d'aplicacions a clústers és una manera poc eficient d'utilitzar els recursos disponibles als clústers. Això fa que aquest tipus d'arquitectura fragmentada generi una gestió addicional a l'hora de prendre decisions quan s'executen certs tipus de càrregues de treball com, a quin clúster ha de col·locar-se, i després a quin Node.

¹ Fent referència al planificador nadiu de Kubernetes

A l'àmbit de IoT, l'augment en l'ús de dispositius i sensors al perímetre de la xarxa ha aconseguit que el poder de còmput s'apropi a la font on es genera aquesta informació, permetent reduir la latència, agilitzar el processament i eventualment millorar l'experiència de l'usuari i el rendiment general, en lloc de la sobrecàrrega de tenir que transferir enormes fluxos de dades fins a un punt centralitzat. De vegades, aquestes transmissions poden no ser factibles degut a falta d'una connexió ràpida i/o confiable o simplement sense connexió. Això ha donat lloc a un tipus d'escenari que de vegades es cita com a un avantatge a la computació al núvol, el *núvol híbrid*. La capacitat de passar d'una infraestructura privada (o núvol privat) a on consumim recursos locals, a un núvol públic en moments alta demanda.

Podríem dir que la unió d'aquestes noves aplicacions, les arquitectures i tecnologies que li donem suport han fet que sigui necessari tenir eines per gestionar l'emplaçament d'aquestes tasques a les entitats on s'executin.

3.1 Una breu història de la planificació a Kubernetes

kube-scheduler és un dels components bàsics del pla de control, i el seu comportament predeterminat és assignar Pods als “millors” Nodes alhora que equilibra la utilització dels recursos entre ells. Tot i que les propostes i millores que van apareixent dia a dia a l'entorn de la planificació han aconseguit que milloràs el seu rendiment i s'adaptés a la majoria d'escenaris de planificació de Pods, hi ha d'altres que no pot fer de forma adequada i senzilla. I és que *kube-scheduler* no ho pot fer tot [31].

Al llarg d'aquests anys d'existència de K8s, veient aquestes mancances, s'han anat creant al ecosistema, planificadors que s'ajustessin a les necessitats de cada entorn de feina i s'ha substituït pel nadiu. Així doncs, ens podem trobar planificadors com:

- *Edge Controller* de Kube Edge [32], enfocat a la computació al perímetre.
- *Apache YuniKorn Scheduler* [33] orientat a ML i dades massives.
- *Volcano* de Cloud Native Computing Foundation (CNCF) [34] per altes càrregues de treball amb ús intensiu de còmput com a podrien ser ML, Deep Learning (DL) i bioinformàtica/genòmica.
- *Slurm*, *Safe-Scheduler*, *Kubernetes Scheduler by Dokku*, etc.

A un entorn de producció la substitució del planificador és el mode més eficient d'optimitzar les càrregues de treball sense oblidar dels inconvenients que ens podríem trobar de prendre aquesta decisió:

- Compatibilitat a l'hora d'actualitzar K8s o el planificador personalitzat. Pel que és una càrrega afegida per l'equip d'operacions vetllar per l'estabilitat del sistema.
- Tant si aquest nou planificador està construït per l'equip de desenvolupament com per un tercer ens hem d'assegurar que tenim un suport en cas de malfuncionament.

- En el cas que vulguem desplegar altres tipus d'aplicacions a les actuals no podem assegurar que la planificació sigui la més adient per aquestes noves tasques.
- Si volem fer la substitució quan l'entorn de producció ja es troba en explotació, hem de contemplar la possibilitat d'aturar tot el sistema per realitzar aquesta tasca en cas de no disposar d'un clúster d'alta disponibilitat.

Si encara així el planificador no s'ajusta a les nostres necessitats, o no aconseguim que faci el que nosaltres volem, podem construir el nostre propi planificador. La feina no és senzilla i com sempre està subjecte a inconvenients però les passes a seguir són les següents:

1. Clonar el codi font del propi *kube-scheduler* disponible al repositori de K8s [35].
2. Modificar el codi que ens hem descarregat i adaptar-lo per als nostres requisits.
3. Tornar a compilar el codi retocat i així obtenir el nou planificador com a un arxiu executable.

Podem arribar a un punt intermedi entre la substitució i la persistència del planificador nadiu evitant aquests inconvenients i augmentant l'estabilitat dels nostres clústers. Existeixen a la comunitat de K8s una sèrie de grups de treball i d'interès especial, un d'ells *SIG-Scheduling* [36], que contínuament van proposant millores de planificació, desenvolupant noves eines, o eliminant altres que han quedat obsoletes.

D'entre una de les millores que proposaren inicialment va ser l'ús d'un extensor del planificador (en anglès *Scheduler Extender*) [37]. Aquesta funcionalitat del planificador habilita extensions no intrusives mitjançant la implementació d'un servei (extern) web, es tracta de un *webhook* [38] configurable amb polítiques de planificació especificant *predicats* i *prioritats* que el planificador nadiu executa per a filtrar i puntuar Nodes respectivament. Tot i que la seva implementació és senzilla, requereix un baix cost de manteniment i s'ha fet servir àmpliament, l'extensor fa enviaments de sol·licituds HTTP i aquestes es poden veure afectades pels entorns de xarxa pel que seu rendiment clarament és molt menor que una simple cridada local, pel que des de la versió 1.23² de K8s varen decidir que no era una solució viable i es deixés de fer servir [39].

Actualment, K8s ens brinda altres estratègies per modificar l'emplaçament dels nostres Pods:

1. Definir perfils de configuració diferents pel mateix planificador i modificar el seu comportament depenent de quin perfil triem. Això ho veurem a la secció 3.3.
2. Executar un (o més d'un) planificador que coexistesqui amb el programador pre-determinat al mateix clúster i indicar quan es definesqui el Pod quin planificador volem fer servir. I això a la secció 3.4.

Aquests plantejaments també podem presentar inconvenients, un dels més clars és, que quan els diferents planificadors planifiquen diferents Pods al mateix Node i al

²A la redacció d'aquest document ens trobam a la versió 1.24

mateix temps durant la presa de decisions de planificació per que la vista de recursos que es mostra per a cada planificador és global. Aleshores, com a resultat pot produir-se un conflicte de recursos de Node. D'aquí sorgeixen solucions com a dividir recursos que un planificador pot planificar en diferents grups, etiquetant-los per evitar conflictes, però això, igualment, disminueix la utilització dels recursos [40].

La comunitat Kubernetes va notar poc a poc les dificultats que enfrontaven als desenvolupadors i per resoldre tots aquests problemes i fer *kube-scheduler* més escalable varen crear a la versió v1.15.0-alpha.3 (07/05/19) el *Marc de Treball de Planificació de Kubernetes* [27]. En essència, aquest marc de treball afegeix un conjunt de connectors API al planificador, pel que manté el "nucli" del planificador simple i fàcil de mantenir mentre fa que la major part de la funcionalitat de la planificació estigui disponible en forma de connectors [40]. A la secció 2.4.1 ja vàrem parlar d'ell i és el fonament de tot el que actualment s'està realitzant al voltant del planificador.

A més de totes aquestes opcions que hem vist fins ara, *kube-scheduler* també disposa d'una sèrie d'estratègies per a assignar Pods a Nodes mitjançant bàsicament l'etiquetatge quan cream el manifest del Pod. Aprofundirem més a la secció 3.2. Encara que existeixen unes quantes polítiques més, aquestes, tracten d'afinar el rendiment del clúster o són combinacions dels procediments anteriors:

1. Nom de Node
2. Selector de Nodes
3. Afinitat i Antiafinitat dels Nodes
4. Afinitat i Antiafinitat dels Pods
5. Prioritat i Preferència dels Pods
6. Restriccions de repartiment de la topologia del Pod
7. Prevencions i Toleràncies

Un assumpte a considerar és el tema de les distribucions de K8s. A la secció 3.6 veurem de quina manera és s'inicialitza el pla de control dependent de la distribució que triem, i de quin mode aquest pot determinar els procediments que tracten de modificar la planificació al nostre clúster.

En aquest capítol veurem de quina manera podem emplaçar les nostres càrregues de treball (Pods/Contenidors) i com controlar aquest component clau.

3.2 Mètodes d'emplaçaments bàsics per a la gestió dels Pods

Per a entendre com funciona el *kube-scheduler* és necessari comprendre el seu context d'execució dins de la gestió de Pods que realitza K8s. És a dir, és necessari tenir una visió del flux típic d'un Pod des que es crea fins que es comença a executar a un Node. Aquest flux es compon de 4 passos i el diagrama de flux de la figura 3.1 facilitarà el seu enteniment. Els components que apareixen van ser introduïts a la secció 2.4.

Aquests passos són:

1. L'usuari crea un Pod a través del servidor de l'API i el servidor de l'API ho escriu al *etcd*.

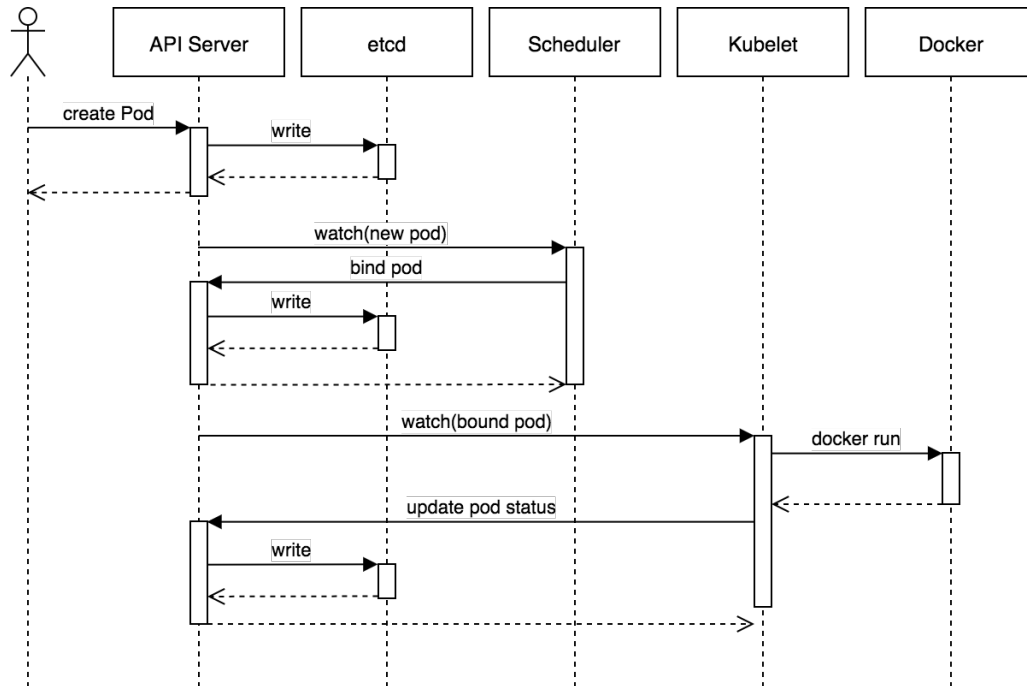


Figura 3.1: Enfocament general d'un Pod al temps. Font: Joe Beda a Heptio [41]

2. El Planificador s'adona que hi ha un Pod no assignat (en anglès *Unbound*) i **decideix** a quin node executar aquest Pod, i torna a escriure aquesta vinculació al servidor de l'API.
3. El *Kubelet* s'adona d'un canvi en el conjunt de Pods que estan vinculats al seu Node. Aquest, a més, executa el contenidor a través de l'entorn en temps d'execució [21]³ del contenidor (en aquest cas, el de *Docker*).
4. El *Kubelet* monitoritza l'estat del Pod a través de l'entorn en temps d'execució del contenidor. A mesura que les coses vagin canviant, el *Kubelet* reflectirà l'estat actual al servidor de l'API.

En aquesta secció en centrarem en aquesta **decisió** (concretament al pas 2) que hem comentat abans i de quin mode manipulam la conducta del planificador. A continuació, veurem les formes de procedir que K8s ens ofereix per a poder gestionar el comportament de *kube-scheduler*. Establirem dos blocs/subseccions principals per destriar-los:

1. Assignació de Pods a Nodes
2. Un Planificador a mida

3.2.1 Assignació de Pods a Nodes

K8s ens proporciona molts de paràmetres i objectes a través dels quals podem gestionar el comportament de *kube-scheduler*. Podem dir que aquests són els mecanismes més

³runtime system o runtime environment

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

fàcils per emplaçar les nostres càrregues de treball. Es tracta de restringir el Pod perquè només s'executi en un conjunt determinat de Nodes. Hi ha diverses maneres de fer-ho i els enfocaments recomanats utilitzen selectors d'etiquetes per facilitar la selecció [42]. A continuació veurem les següents formes de planificació:

1. Nom de Node
2. Selector de Nodes
3. Afinitat i Antiafinitat dels Nodes
4. Afinitat i Antiafinitat dels Pods
5. Prioritat i Preferència dels Pods
6. Restriccions de repartiment de la topologia del Pod
7. Prevencions i Toleràncies

1. **Nom de Node** [43]: És la forma més directa de selecció de Nodes. Per fer-lo servir, afegirem el camp `nodeName` en l'especificació del Pod amb el nom del Node a on volem ubicar el nostre Pod. *kube-scheduler* ignorarà el *kubelet* del Node en qüestió i intentarà emplaçar el Pod en aquest Node. L'ús de `nodeName` anullarà l'ús de `nodeSelector` o qualsevol regla d'afinitat i antiafinitat. També cal mencionar que existeixen una serie de limitacions per fer ús d'aquest tipus d'estrategia:

- Si el Node anomenat no existeix, el Pod no s'executarà, i en alguns casos es pot eliminar automàticament.
- Si el Node anomenat no té els recursos necessaris per ubicar el Pod, aquest fallarà.
- Els noms de Node en entorns de núvol no sempre és previsible o estable.

Un exemple d'especificació de Pod emprant el camp `nodeName` ho tenim al llistat 3.1:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx
8        image: nginx
9    nodeName: nodeWorker-01
```

Llistat del Codi 3.1: Especificació d'un Pod que s'ha d'ubicar al Node nodeWorker-01

2. **Selector de Nodes** [44] : És la forma recomanada més simple. Amb només afegir el camp `nodeSelector` a l'especificació del Pod i especificar les etiquetes de node que volem que tingui el Node de destinació, aconseguim que el Pod sigui assignat al Node que tingui la mateixa etiqueta. A l'apèndix A.1 mostrem un exemple il·lustratiu d'un selector de Nodes emprant comandes de K8s.

3. **Afinitat i Antiafinitat dels Nodes** [25, pàgines 628-629] : Un altre mode de controlar l'ubicació dels Pods a un conjunt específic de Nodes és emprant *regles d'afinitat o anti-afinitat*. Recolzant-nos de les etiquetes dels nodes podem restringir quins Nodes un Pod pot executar.

Les regles d'afinitat/antiafinitat dels Nodes es defineixen amb dos passos:

- a) Assignam una etiqueta a un conjunt de Nodes.
- b) Configurar els Pods perquè s'associïn només als Nodes amb determinades etiquetes.

Hi ha dos tipus de *regles d'afinitat o antiafinitat dels Nodes*:

- **Regles exigides:** Si aquestes regles no es compleixen, el Pod no es pot planificar. Es defineix com a `requiredDuringSchedulingIgnoredDuringExecution` a l'especificació del Node i funciona com al `nodeSelector`, però amb una sintaxi més expressiva.
- **Regles preferides:** El planificador intenta aplicar les regles preferides sempre que sigui possible, però si aquestes regles continuessin sent tan rígides (no poden ser aplicades), el Pod es tornaria no planificable (*Unschedulable*). Es defineix com a `preferredDuringSchedulingIgnoredDuringExecution` a l'especificació del Node. Les regles preferides tenen pesos associats a cada criteri. El planificador crearà una puntuació basada en aquests pesos per programar un Pod al Node correcte. El valor del camp de pes oscil·la entre 1 i 100.

Les regles d'afinitat/antiafinitat es defineixen a l'especificació del Pod. Basant-nos en les etiquetes dels nostres Nodes desitjats/no desitjats, proporcionaríem la primera part dels criteris de selecció a l'especificació Pod. Consisteix en el conjunt d'etiquetes i, opcionalment, els valors (*key=value*).

L'altra part dels criteris és proporcionar la manera com volem comparar les etiquetes. Definim aquests criteris de correspondència amb el `operator` en la definició d'afinitat. Aquest operador pot tenir els següents valors: `In`, `NotIn`, `Exists`, `DoesNotExist`.

Cal esmentar que, si especifiquem `nodeSelector` i `nodeAffinity`, tots dos s'han de complir perquè el Pod es programi en un node.

4. **Afinitat i Antiafinitat dels Pods** [25, pàgines 635-636] : Aquest tipus de regla permet als nostres Pods comprovar quins *altres Pods* s'estan executant a un Node determinat abans de ser assignats en aquest Node. Els *altres Pods* en aquest context no significa una nova còpia del mateix Pod, sinó Pods relacionats amb diferents càrregues de treball. La idea és considerar la necessitat de col·locar dos tipus diferents de contenidors relacionats al mateix lloc (afinitat) o mantenir-los separats (antiafinitat).

Considerem una aplicació que té dos components: una part de frontend (per exemple, una GUI) i un backend (per exemple, una API). Suposem que volem executar-los al mateix host perquè les comunicacions entre els Pods del frontend i del backend serien més ràpides si s'allotgen al mateix Node. Per defecte, en

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

un clúster de diversos Nodes, el planificador assignarà aquests Pods a diferents Nodes. L'afinitat dels Pods proporciona una manera de controlar l'assignació dels Pods entre ells, de manera que puguem assegurar el rendiment òptim de la nostra aplicació. Hi ha dues passes necessàries per definir l'afinitat dels Pods:

- a) Definim com el planificador relacionarà el Pod destí (al nostre exemple anterior, el Pod del frontend) amb els Pods ja en execució (el Pod del backend). Això es fa a través d'etiquetes al Pod, esmentant quines etiquetes dels altres Pods s'han d'usar per relacionar-se amb el nou Pod.

Els selectors d'etiquetes tenen operadors similars, com els descrits a la secció d'afinitat i antiafinitat de Nodes, per fer coincidir les etiquetes dels Pods.

- b) Descriuim a on es volen executar els Pods de destinació. Podem utilitzar les regles d'afinitat de Pods per assignar un Pod al mateix Node que *l'altre Pod* (en el nostre exemple, estem assumint que el Pod backend és *l'altre Pod* que ja està en execució).

En aquesta segona passa definim el conjunt de Nodes on es poden assignar els Pods. Per això, etiquetem el nostre grup de Nodes i definim aquesta etiqueta com a `topologyKey` a l'especificació del Pod. Per exemple, si utilitzem el `hostname` com a valor de `topologyKey`, els Pods es col·locaran al mateix Node. K8s disposa d'un conjunt de etiquetes ben conegudes (en anglès *Well-Known Labels*) que si volem, podem fer servir a les nostres regles. [45] Aquesta regla, la podem fer servir per planificar un Pod a un Node, per a un Node al mateix rack que l'altre Pod, qualsevol Node al mateix centre de dades que l'altre Pod, etc.

Si no l'etiquetam els nostres Nodes amb al nom de rack damunt del qual estan allotjats i definim el nom del rack com a `topologyKey`, els Pods candidats es planificaran a un dels Nodes amb el mateix nom d'etiqueta del nom del rack.

- 5. **Prioritat i Preferència dels Pods** [25, pàgines 642-643] : K8s permet associar una *prioritat* a un Pod. Si hi ha limitacions de recursos i se sol·licita la planificació d'un nou Pod amb alta prioritat, el planificador de K8s pot desallotjar els Pods amb menys prioritat per deixar espai al nou Pod d'alta prioritat (*preferència*).

Per aclarir aquest cas, suposem que sóc l'administrador del clúster d'un banc on es fan càrregues de treball crítiques i no crítiques. En aquest cas, tenim un servei de pagaments així com el lloc web del banc. Puc decidir que el processament dels pagaments és més important que l'execució del lloc web. En configurar la prioritat de Pod, puc evitar que les càrregues de treball de menor prioritat afectin les càrregues de treball crítiques del clúster, especialment en els casos en què el clúster comença a assolir la capacitat de recursos. Aquesta tècnica de desallotjar Pods de menor prioritat per assignar Pods més crítics podria ser més ràpida que afegir Nodes addicionals i ens ajudaria a gestionar millor els pics de trànsit al clúster.

La manera d'associar una prioritat a un Pod és definir un objecte conegut com a `PriorityClass`. Aquest objecte conté la prioritat, que es defineix com un nombre

entre 1 i 1.000 milions. Com més gran sigui el nombre, més gran serà la prioritat. Quan hem definit les nostres classes de prioritat, assignem una prioritat a un Pod associant una `PriorityClass` amb el Pod. Per defecte, si no hi ha cap classe de prioritat associada al Pod, se us assigna la classe de prioritat per defecte si està disponible, o se li assigna el valor de prioritat 0.

6. Restriccions de repartiment de la topologia del Pod [46] : L'afinitat i anti-afinitat dels Pods permeten un cert control de la ubicació del Pod en diferents topologies. Tot i això, aquestes funcions només resolen una part dels casos d'ús de distribució de Pods. Pel que per a poder aconseguir una millor utilització del clúster i una alta disponibilitat de les aplicacions hem de poder distribuir els Pods de manera uniforme entre les topologies. Com hem vist als altres mètodes, aquest tipus de restricció es basa en etiquetes de Node per identificar els dominis de topologia en els que es troba cada Node. És defineixen una sèrie de camps a l'especificació del Pod per indicar-li al planificador com col·locar cada Pod entrants en relació amb els Pods existents al nostre clúster:

- `topologySpreadConstraints` : Aquest és el nom del camp / connector que es definirà a l'especificació del Pod per fer-hi ús. Es pot fer servir nivell de *Pod*, *Deployment*, *DaemonSet*, *StatefulSet*, ...
- `maxSkew` : Grau màxim en què els Pods es poden distribuir de manera desigual
- `whenUnsatisfiable` : Quan `maxSkew` no es pot satisfer quines accions hem de prendre:
 - `DoNotSchedule` (per defecte): Que no assigni el Pod.
 - `ScheduleAnyway` : El planificador donarà més prioritat a les topologies (el Nodes) que ajudarien a reduir el biaix.
- `labelSelector` : Serveix per trobar Pods que concordin. Els Pods que coincideixen amb aquest selector d'etiquetes es compten per determinar la quantitat de Pods al domini de topologia corresponent.
- `topologyKey` : Clau de les etiquetes de Node. Si dos Nodes estan etiquetats amb aquesta clau i tenen valors idèntics per a aquesta etiqueta, el planificador tracta tots dos Nodes com si estiguessin en la mateixa topologia i intentarà col·locar una quantitat equilibrada de Pods a cada domini de topologia.

7. Prevencions ⁴ i Toleràncies [25, pàgines 651-653] : Una *taint* impedeix la planificació d'un Pod a no ser que aquest Pod tingui una tolerància coincident per al Pod. Per sintetitzar, vegem el *taint* com un atribut de Node i la tolerància com un atribut de Pod. El Pod es planificarà al Node només si la tolerància del Pod coincideix amb el *taint* del Node. Les *taints* d'un Node indiquen al planificador

⁴El títol en anglès és **Taints and Tolerations** i representen forces de repulsió i atracció respectivament. La documentació oficial de K8s al *taint* l'anomena *contaminació*. Personalment, m'agrada més *prevenció* que manifesta el posat o gest de rebutjar, desestimar o denegar. Per no donar una accepció incorrecta d'ara endavant l'escriuré en anglès.

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

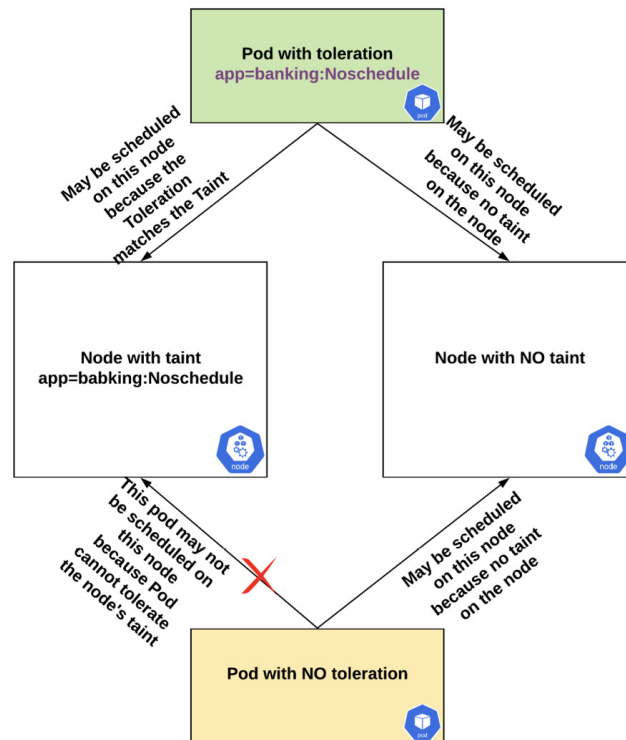


Figura 3.2: Visió general de l'impacte de la planificació amb l'ús de taints i toleràncies.
Font: *The Kubernetes Workshop* [25]

que comprovi quins Pods toleren el *taint* i que executi només aquells Pods que coincideixin en la seva tolerància amb el *taint* del Node.

Una definició de *taint* es compon de clau, valor i efecte. La clau i el valor coincidirán amb la definició de tolerància del Pod a la seva especificació, mentre que l'efecte indica al planificador el que ha de fer una vegada que la *taint* del Node coincideixi amb la tolerància del Pod.

Com veiem a la figura 3.2 que un Pod amb tolerància també pot ser programat en un Node sense *taint*.

Quan definim una *taint*, hem d'especificar el comportament (*effect*) de la *taint* que pot tenir els valors següents:

- **NoSchedule** proporciona la capacitat de rebutjar l'assignació de nous Pods al Node. Els Pods existents que van ser programats abans que es definís la *taint* continuaran executant-se al Node.
- **NoExecute** rebutja els nous Pods que no tinguin una tolerància que coincideixi amb la *taint*. A més, comprova si tots els Pods existents que s'executen al Node coincideixen amb aquesta *taint*, i elimina els que no.
- **PreferNoSchedule** indica al planificador que eviti assignar Pods que no tolerin la *taint* al Node. Es tracta d'una *regla suau*, en què el planificador intentarà trobar el Node correcte, però assignarà els Pods al Node si no

pot trobar cap altre Node que sigui apropiat segons les regles de *taint* i tolerància que hi ha definides.

Hi pot haver moltes raons per les quals vulguem que alguns Pods (aplicacions) no s'executin en determinats Nodes. Un exemple podria ser el requisit de maquinari especialitzat, com ara una GPU per a aplicacions d'aprenentatge automàtic. Un altre podria ser una restricció de llicència per al programari del Pod que estableix que s'hagi d'executar en uns Nodes específics. Utilitzant la combinació de *taints* i toleràncies, pot ajudar el planificador a assignar els Pods al Node correcte.

3.2.2 Un planificador a mida

Podem particularitzar el comportament del planificador permetent-nos configurar les diferents etapes de la planificació del `kube-scheduler`, es a dir, podem habilitar/deshabilitar connectors, canviar l'ordre d'avaluació i afegir arguments per particularitzar el connector. Per fer això, hem de definir un perfil de planificació mitjançant un fitxer de configuració del tipus `KubeSchedulerConfiguration`. Al camp `profiles` l'indicarem de cada una de les etapes que volem emprar, quin(s) connector(s) voldrem fer servir. Cada etapa s'exposa en un punt d'extensió vists a la secció 2.4.1. Els connectors (vegeu 2.4.1) ens proveirà el comportament del planificador mitjançant la implementació d'un o més d'aquests punts d'extensió.

El llistat del codi 3.2 se correspon a una part d'un arxiu de configuració del planificador i en concret a la configuració dels connectors. La resta del manifest és interessant però no rellevant pel que pertoca ara mateix i l'obviarem degut a la dilatada parametrització que pot arribar a tenir.

Al codi en qüestió, començam amb el camp clau `profiles` (línia 6) a on li indicam el nom del planificador a on li seran aplicats aquests canvis (`schedulerName: default-scheduler`).

A partir d'aquí, al camp `plugins` veiem cadascun dels punts d'extensió que volem modificar indicant-li amb la clau `enable` el nom (`- name:`) del connector (o connectors) que voldrem fer servir. En particular, a nostre codi podem veure que el punt d'extensió `filter` té habilitats quatre connectors i que s'executaran seqüencialment i en aquest ordre, de dalt a baix.

Notau que amb el camp `disabled` (línies 12, 17, 20 i 28) ho indicam amb el valor `'*'`, denotant que volem deshabilitar la resta de connectors per defecte en aquest punt d'extensió.

La metodologia per a habilitar els connectors del planificador una vegada fets els canvis depèn de la distribució de K8s que fem servir i podríem dir que és determinant a l'hora de triar-ne un, ja que si les instruccions, funcionament, càrregues de treball, arquitectura pot ser la mateixa, quan es tracta d'aprofundir a un nivell superior de desenvolupament, el tipus de distribució pot facilitar-nos (o dificultar-nos) les tasques més tècniques.

Més endavant, al capítol 3.6, comentarem per els diferents tipus de distribució que hi ha al ecosistema K8s però degut a la rapidesa dels canvis en aquest marc tecnològic, per ventura podria quedar obsolet parlar de quantitats, ordres, percentatges d'ús i altres números que només atindrà a dades del moment d'exposició.

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

```
1  apiVersion: kubescheduler.config.k8s.io/v1beta2
2  kind: KubeSchedulerConfiguration
3  ...
4  clientConnection:
5    kubeconfig: "/etc/srv/kubernetes/kube-scheduler/scheduler.conf"
6  profiles:
7    - schedulerName: default-scheduler
8    plugins:
9      preScore:
10        enabled:
11          - name: TaintToleration
12        disabled:
13          - name: PodTopologySpread
14      score:
15        enabled:
16          - name: VolumeBinding
17        disabled:
18          - name: '*'
19      preFilter:
20        disabled:
21          - name: '*'
22      filter:
23        enabled:
24          - name: VolumeRestrictions
25          - name: NodeName
26          - name: NodePorts
27          - name: TaintToleration
28        disabled:
29          - name: '*'
30  ...
```

Llistat del Codi 3.2: Extracte d'un perfil de configuració per al planificador

```

1  apiVersion: kubescheduler.config.k8s.io/v1beta2
2  kind: KubeSchedulerConfiguration
3  ...
4  profiles:
5    - schedulerName: default-scheduler
6    - schedulerName: no-scoring-scheduler
7    plugins:
8      preScore:
9        disabled:
10         - name: '*'
11      score:
12        disabled:
13         - name: '*'
14    ...

```

Llistat del Codi 3.3: Múltiples perfils de configuració per un mateix planificador

3.3 Configurar múltiples perfils d'assignació

Amb el mateix mètode que a l'apartat anterior definíem una personalització per al planificador per defecte, podem igualment configurar *una única instància* del *kube-scheduler* i executar diversos perfils de planificació amb diferents comportament mitjançant l'exposició de les etapes dels punts d'extensió del marc de treball del planificador.

En aquest cas, haurem de definir per a cada perfil, un nou `schedulerName` i a dins d'ell, l'especificació de cada perfil.

Al llistat del codi 3.3 hi ha dos perfils definits. El primer és el del planificador per defecte (`default-scheduler`, línia 5) al qual no li hem volgut canviar res. El següent, que l'hem anomenat `no-scoring-scheduler`, i que com indica el seu nom li hem deshabilitat tots els connectors de puntuació (`score`) i prepuntuació (`preScore`).

Cal dir que, com el planificador és únic, tots els perfils hauran de fer servir el mateix connector al punt d'extensió `queueSort` i tenir els mateixos paràmetres de configuració, ja que la cua d'ordenació de Pods pendents és única (vegeu la secció 2.4.1).

Els pods que volgum planificar d'acord amb un perfil específic poden incloure el nom del planificador corresponent en el vostre manifest a l'apartat `.spec.schedulerName`. Al llistat del codi 3.4 observam que a la seva especificació del nom del planificador li demanem que el Pod es planifiqui amb el perfil de configuració `no-scoring-scheduler`.

Si un Pod no especifica un nom de planificador, `kube-apiserver` l'establirà al `default-scheduler`. Per tant, hi ha d'haver un perfil amb aquest nom de planificador per planificar aquests Pods. A l'exemple del llistat 3.5 ho hem explicitat.

3.4 Múltiples planificadors

Hem vist que el planificador per defecte ens permet una certa elasticitat per afinar a on volem el nostres Pods, però encara així, les restriccions que podem aplicar als

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: annotation-second-scheduler
5    labels:
6      name: multiprofile-example
7  spec:
8    schedulerName: no-scoring-scheduler
9    containers:
10 -   name: pod-with-second-annotation-container
11     image: k8s.gcr.io/pause:2.0
```

Llistat del Codi 3.4: Especificació de Pod que es planificarà al segon perfil

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: annotation-default-scheduler
5    labels:
6      name: multiprofile-example
7  spec:
8    schedulerName: default-scheduler
9    containers:
10 -   name: pod-with-default-annotation-container
11     image: k8s.gcr.io/pause:2.0
```

Llistat del Codi 3.5: Especificació de Pod que es planificarà al perfil per defecte

nostres Pods perquè s'assignin a uns Nodes determinats solen ser restriccions dures, anotacions o etiquetes ben conegudes (en anglès Well-Known Labels, Annotations and Taints) [45]:

- CPU
- Memòria
- Indisponibilitat de Xarxa
- Regions i/o zones topològiques, entre altres mètriques [47]

I altres més laxes com seria l'etiquetatge estàtic que ja hem comentat a la secció 3.2:

- Selector de Nodes
- Afinitat i antiafinitat
- Taints i toleràncies
- Prioritats

- Distribució de la topologia

Però quan tenim un altre tipus de necessitats i volem assignar els Pods basant-nos en altres mètriques i veim que les eines de les que disposam no són suficients per respondre als nostres requeriments. Podríem tenir la demanda d'assignar Pods amb unes exigències ben concretes:

- Ample de Banda de determinats serveis (HTTP, IMAP, FTP, POP, ...).
- Sol·licituds HTTP per segon: Per a controlar el tràfic de peticions d'un servei REST.
- Latència HTTP: Aconseguir que els Pods estiguin més a prop del consumidor.
- Rendiment GPU: A quins Nodes tenc les GPU amb més de n TFLOPS disponibles a un instant de temps determinat.
- Colls d'ampolla (en anglès *bottlenecks*): el rendiment general és bo, però un Node en particular té problemes.

Aquestes mètriques són variables, i durant l'execució de les aplicacions van canviant. No les poden obtenir directament del planificador i requerirem d'altres eines que podran interactuar amb K8s i suplir aquestes limitacions. Des de la versió 1.6, K8s suporta la coexistència de múltiples planificadors [48].

Si havíem vist que amb el mateix planificador podíem definir diversos perfils amb els quals canviar el comportament de la planificació dels Pods. Aquí bàsicament la idea és la mateixa, tan sols que ara no operarem a nivell de perfils sinó que ho farem directament amb distints planificadors.

Per incloure un altre planificador al nostre clúster de K8s tenim un parell d'opcions:

- Fent ús d'un gestor d'aplicacions per a K8s: Actualment, l'aplicació per excel·lència és *Helm* [49] .

Helm és una eina que per gestionar paquets de K8s, és un projecte graduat de la CNCF i mantengut per la Comunitat *Helm* amb col·laboració de Microsoft, Google i Bitnami. Aquests paquets s'anomenen *Cartes de Navegació* o *Helm Charts*, (en endavant *charts*). Una *chart* és una col·lecció de fitxers que descriuen a un conjunt de recursos de l'API de K8s, aquestes *charts* en permeten crear, actualitzar i publicar una aplicació.

Al lloc web [ArtifactHUB](#) disposam d'un repositori de *charts* de Helm a on trobarem infinitat d'aplicacions.

Al subprojecte *scheduler-plugins* de *Kubernetes SIGs* hi ha disponible una *chart* per a la instal·lació d'un planificador com a segon planificador⁵

Aquesta, podríem dir que pot ser la més simple, ja que tenim una *chart* que ja ho fa i que el podem personalitzar canviant-li els valors.

- S'altre opció és fer-ho manualment sense cap tipus d'automatització (a diferència del cas anterior). A continuació, explicarem amb més detall el procediment a seguir per a desplegar un planificador addicional al que ja tenim.

⁵ [as-a-second-scheduler](#)

3.4.1 Desplegar un nou planificador

Inicialment, assolirem una sèrie de premisses:

1. La primera i no per això la més important, que per facilitat i comprensió del procediment, el nou planificador l'executarem com un Pod estàtic i no com un servei dependent de K8s (vegeu 3).
2. Tenim un clúster de *Kubernetes* en funcionament amb l'eina de línia de comandes *kubectl* configurada per a comunicar amb el clúster.
3. El clúster està conformat per al manco dos Nodes que no funcionen al *Pla de Control*, es a dir, són *Nodes Treballadors*.
4. Tenim també instal·lada l'eina Docker.
5. Disposam d'un planificador (arxiu binari executable) que actuarà com a segon planificador al nostre clúster. Aquest arxiu podria esser cridat des de el mateix clúster, però per il·lustrar el procés ho resoldrem com sol ser més habitual, des d'un Registre de Containers.

Empaquetar my-scheduler

1. Empaquetam el nostre planificador que anomenarem `my-scheduler` com una imatge de contenidor. Per construir-lo ho farem mitjançant un arxiu `Dockerfile`. Ho guardarem amb aquest mateix nom. Al llistat 3.6 adjuntam el codi necessari.

```
1 FROM busybox
2 ADD ./_output/local/bin/linux/amd64/kube-scheduler /usr/local/bin/kube-scheduler
```

Llistat del Codi 3.6: Codi del Dockerfile per a construir la imatge de contenidor

2. Construïm la imatge i ho pujam a un Registre de Containers (i.e. Azure Container Registry, Amazon Elastic Container Registry, Google Container Registry, IBM Cloud Container Registry, Docker Hub,...). En el meu cas ho pujaré a *Docker Hub* al disposar d'un compte gratuït. Una mostra de com realitzar la tasca és l'exemple del codi 3.7.

```
1 docker build -t andreuet/my-scheduler:0.1 .
2 docker login
3 docker push andreuet/my-scheduler:0.1
```

Llistat del Codi 3.7: Comandes Docker a la *shell* del sistema operatiu

Definició d'un manifest pel nostre planificador

A l'hora de desplegar el planificador tenim varies possible solucions. Tot i que triarem la segona opció per simplicitat, convé conèixer la resta:

1. Pod: El desplegament d'un Pod és l'opció més senzilla i és la unitat més petita que es poden crear (vegeu 2.3 tercer paràgraf) però a la vegada que amb menys estabilitat. Els Pods com a entitats d'un sol ús requereixen de controladors per gestionar la seva replicació desplegament i reparació automàtica. Poques vegades farem servir pods per a desplegar les nostres aplicacions, normalment emprarem recursos de càrrega de treball com *Deployment*, *StatefulSet*, *Jobs* o *DaemonSet* [50].
2. Deployment: Aquest objecte té l'avantatge de que ja crida al controlador de rèpliques i és molt més fàcil mantenir la consistència de les dades. A efectes de proves i desenvolupament és una opció ràpida, però per operar en producció no és massa aconsellable per que essent un component crític del pla de control, aquest objecte és tractat com una càrrega de treball més i no té cap privilegi o prioritat que el distinguesqui dels altres Pods.
3. Pod estàtic : K8s per defecte, al directori `/etc/kubernetes/manifests` es troben els arxius de manifests a on es despleguen tots els components del pla de control, i ho fan com a Pods estàtics. Aquests tipus de Pods tenen unes particularitats o propietats comunes que els fan diferents a la resta de desplegaments. Citarem les més destacables, tot i que n'hi força més [51]:
 - Es despleguen al espai de noms `namespace:kube-system`.
 - Se'ls etiqueta automàticament amb:
 - `tier:control-plane`
 - `component:{component-name}`
 - Usen la classe de prioritat *PriorityClass* (Vegeu 5) `system-node-critical`, una de les dues classes més prioritàries que K8s té definides per defecte.
 - No són controlats pel servidor de l'API, pel que els fan útils pel cas d'iniciar components del pla de control.
 - Si aquest Pod fallés, automàticament es reiniciaria.
4. Servei : Aquest tipus de desplegament és inevitable en determinades distribucions de K8s, i el motiu és bàsicament per temes d'aprofitament de memòria. Tot i que hi ha més raons, aquestes les detallarem a la secció 3.6. En aquests cas, el planificador s'executa com un procés controlat pel de `systemd` [17] i no a dins de contenidors.

A continuació, al llistat 3.8 mostrem una part de la configuració.

Degut a la seva extensió, només destacam les parts més importants. En cas de voler consultar l'arxiu complet ho podem fer a través de l'adreça a peu de pàgina ⁶:

Al llistat 3.8 podem apreciar que:

- Hem de crear un compte de servei dedicada per poder adquirir el mateixos privilegis que `system:kube-scheduler` (línia 2).
- S'executarà amb el nom `my-scheduler` i al nom d'espai `kube-system` (línies 4-5).

⁶[my-scheduler.yaml](#)

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

```
1  apiVersion: v1
2  kind: ServiceAccount
3  metadata:
4    name: my-scheduler
5    namespace: kube-system
6  ---
7  ...
8  apiVersion: v1
9  kind: ConfigMap
10 metadata:
11   name: my-scheduler-config
12   namespace: kube-system
13 data:
14   my-scheduler-config.yaml: |
15     apiVersion: kubescheduler.config.k8s.io/v1beta2
16     kind: KubeSchedulerConfiguration
17     profiles:
18       - schedulerName: my-scheduler
19     leaderElection:
20       leaderElect: false
21   ---
22 apiVersion: apps/v1
23 kind: Deployment
24 metadata:
25   labels:
26     component: scheduler
27     tier: control-plane
28   name: my-scheduler
29   namespace: kube-system
30 spec:
31   ...
32   replicas: 1
33   ...
34   spec:
35     serviceAccountName: my-scheduler
36     containers:
37     - command:
38       - /usr/local/bin/kube-scheduler
39       - --config=/etc/kubernetes/my-scheduler/my-scheduler-config.yaml
40       image: hub.docker.com/repository/docker/andreuet/my-scheduler:1.0
41   ...
```

Llistat del Codi 3.8: Extracte de configuració del desplegament: `my-scheduler.yaml`

- El desplegament és farà indicant-li que el component serà `scheduler` i que el seu nivell és el pla de control `tier: control-plane` (línies 26-27).
- El ConfigMap enmagatzema s'arxiu de configuració `my-scheduler-config` i el montará a un volum. (línies 9-11).
- Utilitzam un `kind: KubeSchedulerConfiguration` per personalitzar el comportament del planificador, li passam amb l'opció `-config` s'arxiu de configuració (línies 16 i 39).
- La imatge la descarregarà del repositori creat abans (línia 40).

Executar el nostre planificador al clúster

Una vegada que ja tenim definit com i on hem de desplegar el planificador, és hora de executar-ho:

```
$ kubectl create -f my-scheduler.yaml
```

I comprovam que el Pod s'està executant

```
kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
....				
my-scheduler-lnf4s-4744f	1/1	Running	0	2m
...				

Com podem veure, el Pod del planificador *my-scheduler* s'està executant. És hora de fer-lo funcionar.

De la mateixa manera que férem a la secció 3.3, per a que un Pod (o un altre tipus de càrrega de treball) empri el nostre planificador hem d'incloure `spec.schedulerName: my-scheduler` a l'arxiu de manifest de creació del Pod. Cal mencionar que:

- Si no indicam cap nom de planificador el desplegament s'executarà emprant el planificador predeterminat.
- Ens hem d'assegurar que el valor de `spec.schedulerName` ha de coincidir amb el nom que varem assignar al camp `schedulerName` al `KubeSchedulerProfile` de l'arxiu `my-scheduler.yaml` (vegeu 3.8) a la línia 18.

Amb el següent exemple de codi del llistat 3.9, il·lustrem el mode en que feim servir el planificador:

I amb la següent comanda creariem el pod per a que es planifiqui al planificador assignat:

```
kubectl create -f pod3.yaml
```

A la línia 8 del codi 3.9 especificam a `my-scheduler` com a planificador.

Verificar que el Pod es planifica al planificador desitjat

Per a verificar que realment funciona podem fer el següent (i amb aquest ordre):

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: annotation-second-scheduler
5    labels:
6      name: multischeduler-example
7  spec:
8    schedulerName: my-scheduler
9    containers:
10 -   name: pod-with-second-annotation-container
11     image: k8s.gcr.io/pause:2.0
```

Llistat del Codi 3.9: `pod3.yaml` que es planificarà a `my-scheduler`

1. Cream el Pods que volem planificar al nostre planificador abans d'enviar la configuració d'implementació del planificador `my-scheduler`. Això farà que l'estat d'aquests Pods sigui `Pending`.
2. El Pods romandran amb aquests mateix estat fins que enviem la configuració d'implementació del planificador `my-scheduler`. Una vegada fet això els nostres Pods passaran a l'estat de `Running`.

Alternativament, podem veure ses entrades "Planificades"el registres d'events per comprovar que els Pods s'han planificat al nostre planificador amb la següent comanda:

```
$ kubectl get events
```

3.5 Construint el teu propi planificador

Com justificàrem a la secció anterior, existeix la necessitat de construir el nostre propi planificador. Tot i que hi ha moltes eines que ens faciliten la feina per realitzar aquestes tasques, no ens queda una altra que crear el nostre connector codificant-lo amb un llenguatge de programació adient. De vegades el que volem aconseguir sol ser tan simple que amb unes línies de codi ens bastarà. Però, de vegades per a la construcció, requerirem l'ús d'eines de tercers per poder accedir a informació que d'entrada, no és tan evident d'aconseguir. Ens estam referint a les mètriques de monitorització.

3.5.1 Mètriques

A través de l'API de mètriques de K8s podem obtenir informació dels recursos que estam emprant actualment i valorar la situació del clúster amb una regularitat i prendre, si cas accions d'optimització, salut i eficiència, però aquesta API no emmagatzema els valors de les mètriques [52]. És per això que moltes vegades es solen fer ús d'eines de

tercers per aquests tipus de tasques, que a més de mesurar les mètriques que genera l'aplicació, poden emmagatzemar en bases de dades les sèries temporals de la informació obtinguda, s'integren amb altres sistemes mitjançant l'exportació de dades de tercers, permeten realitzar consultes de les dades recopilades, envien alertes, d'entre altres opcions.

Eines de monitoreig basades en el núvol n'hi ha moltes i triar-ne una pot arribar a trabucar a més d'un, però si hem de fer una aposta segura *Prometheus* és considerada per molts l'eina de monitoreig per excel·lència. Com a segon projecte a unir-se a CNCF i també el segon en *graduar-se* [53] (després de Kubernetes), *Prometheus* no ha deixat de créixer i s'ha establert com un *standard de facto*, fins a tal punt que CNCF ha establert un Programa de Conformitat de Prometheus que garanteix consistència i portabilitat de cada versió. Conseqüentment, el Programa de Conformitat de Kubernetes té dos tipus de certificacions de conformitat: - Compliment de components i - Compatibilitat de Prometheus [54].

Prometheus destaca en una sèrie de punts clau:

1. Recopilació de Mètriques: *Prometheus* recupera les mètriques a través de HTTP. Si no és possible *Prometheus* pot enviar-les a través d'una porta d'enllaç intermitja *PushGateway*.
2. Recurs metric: Els sistemes que volen monitoritzar amb *Prometheus* han d'exposar les mètriques en un recurs que s'anomena `/metrics`. *Prometheus* fa servir aquest recurs per extreure les mètriques a intervals regulars.
3. Llenguatge de Consulta Prometheus: Anomenat comunament *PromQL* (Prometheus Query Language) permet a l'usuari consultar les mètriques i afegir dades de sèries temporals en temps real.
4. Bases de Dades de Sèries Temporals: Per emmagatzemar totes les dades de manera eficient *Prometheus* fa servir TSDB (Time Series DataBase) localment de manera predeterminada encara que hi ha opcions per integrar emmagatzemament remot.
5. Exportadors: Són llibreries que converteixen les mètriques existents de tercers al format de mètriques de *Prometheus*. Existeixen moltíssims exportadors comunitaris i oficials, un exemple és l'exportador de Nodes [55] (*node-exporter*) que exposa totes les mètriques a nivell del sistema de *Linux* en format *Prometheus*.

Com veurem un poc més endavant, aquest darrer punt serà determinant per il·lustrar l'exemple que volem dur a la pràctica.

3.5.2 Llenguatge de Programació

Per una altra banda hem de fer l'elecció del llenguatge de programació. S'elecció per excel·lència (i el que triarem) és *Go* [56]. El motiu és evident, K8s i moltes tecnologies natives al núvol estan escrites en *Go*, en particular, *Prometheus* també està escrit en *Go*, de manera que la integració és total. Això no implica que no puguem fer servir altres llenguatges de programació [57]. Tractant-se d'un sistema de codi lliure, K8s posa a la

nostra disposició una sèrie de biblioteques de client per utilitzar l'API. De les que són mantengudes oficialment per *API Machinery Special Interest Group* estan: Python, Java, Javascript, C#, Go, Haskell, Perl. Ruby i C encara estan en progrés. Hi ha també unes altres biblioteques mantegudes per la comunitat dels que trobarem per llibreries de les anteriorment mencionades i llenguatges com: Clojure, Elixir, Lisp, PHP, Node.js, Rust, Scala, Swift entre d'altres del que K8s no es fa responsable [58].

Golang, comunament conegut com a *Go*, va ser llançat el 2009. Creat pels enginyers de Google Robert Griesemer, Rob Pike i Ken Thompson, aquest llenguatge va ser dissenyat per a resoldre certs problemes combinant els punts forts i eliminant les mancances dels llenguatges de programació existents.

Com a característiques clau podem destacar:

1. És un projecte de *codi obert* pel que el codi font del seu compilador, les llibreries i les eines estan a disposició de qualsevol
2. Tipatge estàtic: La comprovació de tipificació es realitza durant la compilació, i no mentre s'executa.
3. Multiplataforma a través de la Compilació creuada que implementa de manera nativa⁷. Funciona en sistemes de tipus *Unix* com podrien ser *-Linux, FreeBSD, OpenBSD* i *Mac OS X-*, en *Plan 9*⁸ i en *Microsoft Windows*.
4. En termes de velocitat i sintaxi és equiparable al llenguatge de programació *C*.
5. Dissenyat pensant en *escalabilitat* i *concurrència* a través de les *goroutines*[59].
6. Llenguatge compilat que s'acobla directament al codi de la màquina i no necessita un intèrpret. Com a resultat, el seu rendiment és comparable al de llenguatges com a *C++*.

Go és especialment adequat per a la construcció d'infraestructures, com a servidors de treball en xarxa, i eines i sistemes per a desenvolupadors, però és realment un llenguatge de propòsit general i s'utilitza en àmbits tan diversos com els gràfics, les aplicacions mòbils i l'aprenentatge automàtic [60].

3.5.3 Comunitat *Kubernetes-SIGs*

Per al disseny d'un connector personalitzat farem ús del subprojecte *scheduler-plugins* que ens proveeix els Grups d'Interès Especial de Kubernetes (*Kubernetes-SIGs*). Aquesta comunitat formada per empreses i particulars existeix des de el llançament de la versió 1.0 i comparteixen com es centren en l'experiència dels desenvolupadors i del moviment DevOps⁹ en l'execució d'aplicacions a Kubernetes [61].

⁷Capacitat de crear codi executable per a una altra plataforma diferent d'aquella en què s'executa el compilador

⁸Plan 9 from Bell Labs. url: <http://p9f.org/>

⁹Conjunt de pràctiques que agrupen el desenvolupament de programari i les operacions de TI. El seu objectiu és fer més ràpid el cicle de vida del desenvolupament de programari i proporcionar un lliurament continu d'alta qualitat. Font: Wikipedia

Cada un dels grups s'enfoquen en una part del projecte Kubernetes. Qualsevol pot participar i contribuir sempre que es segueixi un Codi de Conducta. Cada SIG pot tenir un conjunt de subprojectes més petits que poden treballar de forma independent. Alguns subprojectes seran part dels lliurables principals de Kubernetes, mentre que altres seran més especulatius i viuran a l'organització de *Kubernetes-SIGs* [62].

Un d'aquests projectes és *Scheduling Special Interest Group* o *SIG Scheduling* [36] que és la part responsable dels components que prenen decisions d'ubicació de Pods. Dissenyen planificadors de K8s i implementen funcions que permeten als usuaris personalitzar la ubicació dels Pods en els Nodes d'un clúster. Actualment tenen 7 subprojectes en marxa i un d'ells és *scheduler-plugins* que emprarem nosaltres per crear el nostre connector personalitzat [36].

3.5.4 Construcció

Vist tot l'exposat i entenent el funcionament del Marc de Treball del Planificador ens podem fer una idea del que realment necessitam és construir un component que implementi els Punts d'Extensió 2.4.1. Aquest component és el *Connector*.

Per fer ús d'aquest connector no basta amb crear-lo i prou, hem de dir-li a quins punts d'extensió es pot aplicar. Un connector es pot aplicar a un o més punts d'extensió, és per això que després haurem de configurar-lo mitjançant els Perfiles de Planificació, un tema que vérem a la subsecció 3.2.2.

Al següent capítol ens endinsarem pròpiament en la construcció d'un model de connector fent ús dels coneixements adquirits fins ara.

3.6 Implantació del planificador segons distribució i plataforma

Des de la seva aparició al 2014 ¹⁰ (*Kubernetes First Commit*), l'elecció de K8s com a plataforma tecnològica més comú per a les operacions aplicades a contenidors de Linux, han sorgit multitud de distribucions per a executar de K8s en un clúster i l'elecció dependrà dels requisits que precisem al definir el nostre entorn.

Actualment, CNCF ha donat la conformitat *Certified Kubernetes* a més d'un centenar de distribucions certificades. Aquesta llista contínuament està canviant i per conèixer quines són ho podem consultar des del *Panorama a CNCF* [63] o des de la llista actualitzada de conformitat de K8s que tenen disponible mitjançant un enllaç a *GitHub* [64].

La distribució de K8s (o plataforma com l'anomenen a CNCF) és determinant si hem d'operar el nostre planificador i cal comentar-lo amb un poc més de detall. Així doncs categoritzant-les segons la seva *plataforma* tendrem [65]:

1. Distribució (*Distribution*): Un proveïdor pren el nucli de Kubernetes, que és el codi font obert sense modificar (encara que alguns el modifiquen), i l'empaqueta per redistribuir-lo. En general, això implica trobar i validar el programari

¹⁰A mitjans de l'any 2014 aparegué la primera entrega de la versió alliberada de Kubernetes i no va ser fins a mitjans de 2015 quan s'alliberà la versió 1.0.1

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

- de Kubernetes i proporcionar un mecanisme per manejar la instal·lació i les actualitzacions del clúster.
2. Allotjat (*Hosted*): Un servei ofert per proveïdors d'infraestructura com AWS, Digital Ocean, Azure i Google, que permet als clients activar un clúster de Kubernetes a comanda. El proveïdor del núvol assumeix la responsabilitat d'administrar part del clúster de Kubernetes, generalment anomenat pla de control. Són similars a les distribucions però administrades pel proveïdor del núvol a la seva infraestructura.
 3. Instal·lador (*Installer*): Ajuden a instal·lar Kubernetes en una màquina. Automatitzen el procés d'instal·lació i configuració de Kubernetes i fins i tot poden ajudar amb les actualitzacions. Els instal·ladors de Kubernetes sovint es combinen amb les distribucions de Kubernetes o les ofertes allotjades de Kubernetes o els utilitzen.

En el cas d'una elecció *allotjada* (també conegut com a *Kubernetes Administrat*), el pla de control és gestionat pel proveïdor, conseqüentment, el planificador també serà gestionat per ell. Adicionalment, el proveïdor pot oferir certes funcionalitats per planificar els Pods però tot dins del seu propi entorn de gestió.

A les altres dues plataformes (conegudes com a *Kubernetes estàndar* o *Vanilla Kubernetes* [66]), el pla de control pot ser gestionat per nosaltres, i en conseqüència també el planificador. Dins aquesta vessant, també ens trobarem que la inicialització (en anglès *bootstrapping*) del planificador pot realitzar-se com a:

1. Pod estàtic: Com vàrem comentar anteriorment a 3, aquest tipus de Pods són ideals per a inicialitzar els components del pla de control ja que no són controlats pel servidor de l'API, són executats inicialment pel *kubelet* que el va observant contínuament i el reiniciarà si per qualche motiu falles.

Aquest tipus de desplegament té com a avantatges:

- Tenir el control del planificador ubicat al nom d'espai `kube-system` a través de `kubectl` i visualitzable com un Pod més.

```
kubectl get pods -o wide -n kube-system
```

- Veure versió dels arxius binaris que s'estan executant tan sols mirant la imatge del contenidor.
- El registre i estat es recopilen i processen com a qualsevol altre registre de contenidor.
- El planificador està al mateix nivell amb altres serveis crítics com DNS, eines de monitoreig i altres complements, que s'executen com a desplegaments (*deployments*) o demonis (*DaemonSet* [67]).
- Des de que va aparèixer K8s, ha estat la forma predeterminada d'iniciar-se el planificador, pel que moltes eines i guions (*scripts*) s'han dissenyat per aquest tipus de funcionament i és molt més fàcil trobar documentació al respecte.

D'aquest tipus de desplegament/instal·lació no citaré cap perquè ens trobam a la majoria de membres de CNCF.

2. Servei init del Sistema Operatiu (SO) (daemon): Aquest gestor de serveis i sistemes en *Linux* (*systemd*) crida al binari executable (en el nostre cas, el planificador) des del fitxer de configuració anomenat normalment *kube-scheduler.service*, deixant la gestió i el control en mans del SO que s'executarà com un procés més.

Per consultar i gestionar aquest procés emprarem eines/comandes com a *journalctl* o *systemctl* respectivament, pròpies del SO. També podem comprovar l'estat dels components del pla de control amb una petició al servidor de l'API [68]:

```
kubectl get -raw='/readyz?verbose' O
kubectl get -raw='/livez?verbose'
```

Com a avantatge podem dir que:

- L'aïllament del planificador com a servei al *systemd* comporta una major seguretat pel fet que viu fora del l'àmbit de K8s i és més complicat comprometre els meus Pods si el atacant no pot accedir als components del pla de control.
- No és el denominador comú, però les plataformes més livianes solen fer servir aquest tipus d'instal·lació transferint-li el pla de control directament al SO i alliberant de càrregues a l'orquestrador i agilitzant la seva execució. Això permet la seva instal·lació a un portàtil o ordinador d'escriptori per fer proves, experimentació o inclús per a producció a un perfil molt baix de maquinari.

Per contra, tenim que:

- Fer aquesta separació dificulta la seva gestió ja que hem d'administrar dos sistemes/capes (el propi SO i K8s).
- És feina nostra recopilar els registres del pla de control amb eines alienes a K8s (docker, journalctl, etc).
- Disposar de la documentació per a realitzar modificacions del pla de control queden a criteri de cada plataforma. Pel que ens podem trobar amb manca d'informació a l'hora dur a terme aquestes tasques.

En aquest tipus de plataforma ens podem trobar plataformes com a: MicroK8s de Canonical¹¹, minikube, kubernetes kind, Desktop Kubernetes, etc.

¹¹ Canonical també té una distribució més robusta amb el desplegament del pla de control amb Pods estàtics (*Charmed Kubernetes*) [69]

3.7 Eines d'avaluació del comportament del planificador

A priori, per conèixer la conducta en detall del es requereix accés privilegiat al pla de control per poder llegir els registres. Tot que aquest tipus de comesa no és habitual al món de K8s, sí que ens pot aportar avantatges a l'estudi experimental com a:

1. Respostes més ràpides a les proves o hipòtesi de recerca. Evitant crear un entorn que hem d'anar modificant contínuament.
2. Avaluació rigorosa de l'efecte que pot tenir un determinat desplegament.
3. De vegades, pel tipus de disseny que volem fa que el simulador sigui l'única manera que tenim per veure realment com es comporta el planificador.

A l'informe anual de 2021 de Kubernetes la comunitat *Kubernetes-SIG Scheduling* [70] i al *KubeCon 2021 North America* [71] varen presentar el subprojecte *Kubernetes scheduler-simulator* [72].

El simulador internament està format per un *kube-apiserver*, un *kube-scheduler* i un *servidor HTTP*. El client del simulador, o la interacció amb ell la podem fer a través de:

1. Interfície d'usuari web
2. *kube-apiserver* via *kubectl*
3. Llibreries de client K8s

La forma més còmoda és amb l'interfície d'usuari web que ens permet:

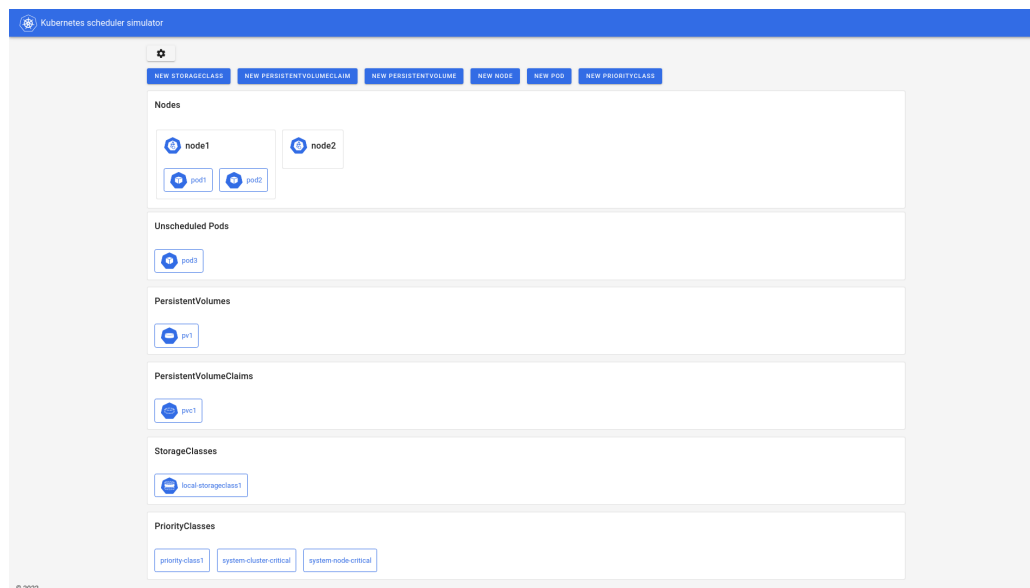


Figura 3.3: Interfície web del simulador. Font: *Kubernetes* [72]

- Crear recursos (Nodes, Pods, PVs, PVCs, etc) amb només clicar un botó o amb un arxiu *yaml* que també podem crear i editar a la mateixa interfície. Vegeu la figura 3.4



Figura 3.4: Creació d'un node amb un arxiu *yaml*. Font: Kubernetes [72]

- A la figura 3.5 podem comprovar com podem visualitzar els resultats per:
 1. Connectors de filtre `filter`
 2. Connectors de puntuació `score`
 3. Puntuació final, després d'haver normalitzat y aplicat el pes del connector.
- També podem configurar el planificador mitjançant `KubeSchedulerConfiguration`. Vegeu figura 3.6.

Com a nou subprojecte, el simulador està subjecte a canvis, però actualment és totalment funcional encara que es cerquen millores com les que actualment estan en marxa [73]:

- Simulació basada en escenaris: Creació d'un nou Custom Resource Definitions (CRD) anomenat `Scenario` que ens permetrà simular el planificador amb diversos escenaris definits.
- Operador de simulació: Creació d'un nou CRD anomenat `Simulation` per a representar un simulador i un controlador personalitzat per administrar-lo. D'aquesta manera els usuaris poden definir simuladors a través dels recursos del simulador i el controlador crearà, editarà i eliminarà els simuladors.
- Simulació del planificador: Creació d'un nou CRD anomenat `SchedulerSimulation`. Aquest recurs crearà el recurs `Simulator` per executar un simulador i executarà un `Scenario` en aquest simulador.

3. ACCIONS PER A LA MODIFICACIÓ DE LA SELECCIÓ I EMPLAÇAMENT DE PODS/CONTENIDORS

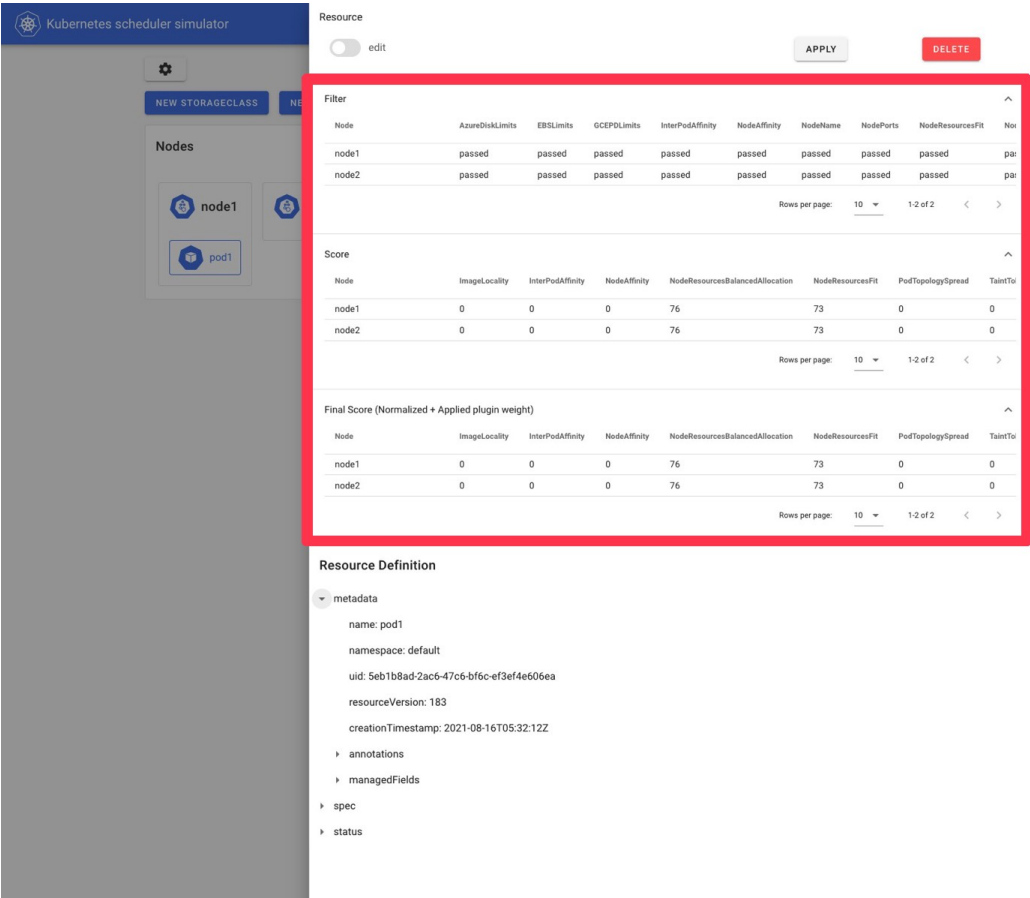
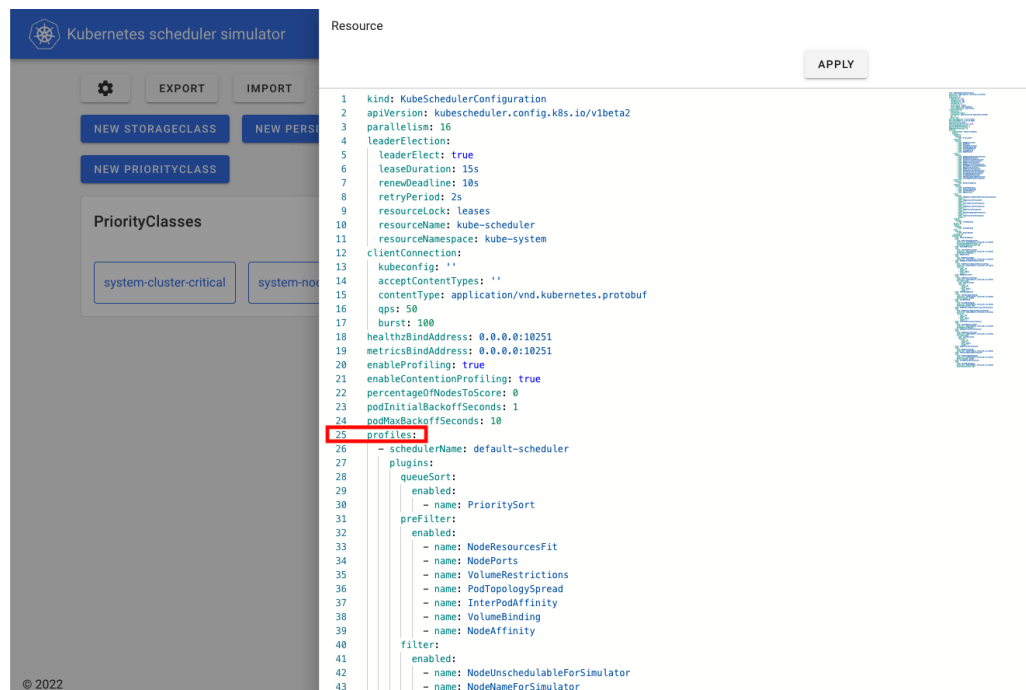


Figura 3.5: Visualització dels resultats obtinguts. Font: Kubernetes [72]

3.7. Eines d'avaluació del comportament del planificador



Resource

```
1 kind: KubeSchedulerConfiguration
2 apiVersion: kubescheduler.config.k8s.io/v1beta2
3 parallelism: 16
4 leaderElection:
5   leaderElect: true
6   leaseDuration: 15s
7   renewDeadline: 10s
8   retryPeriod: 2s
9   resourceLock: leases
10  resourceName: kube-scheduler
11  resourceNamespace: kube-system
12  clientConnection:
13    kubeconfig: ''
14  acceptContentTypes: ''
15  contentType: application/vnd.kubernetes.protobuf
16  qps: 50
17  burst: 100
18  healthzBindAddress: 0.0.0.0:10251
19  metricsBindAddress: 0.0.0.0:10251
20  enableProfiling: true
21  enableContentionProfiling: true
22  percentageOfNodesToScore: 0
23  podInitialBackoffSeconds: 1
24  podMaxBackoffSeconds: 10
25  profiles:
26    - schedulerName: default-scheduler
27      plugins:
28        queueSort:
29          enabled:
30            - name: PrioritySort
31        prefilter:
32          enabled:
33            - name: NodeResourcesFit
34            - name: NodePorts
35            - name: VolumeRestrictions
36            - name: PodTopologySpread
37            - name: InterPodAffinity
38            - name: VolumeBinding
39            - name: NodeAffinity
40        filter:
41          enabled:
42            - name: NodeUnschedulableForSimulator
43            - name: NodeNameForSimulator
```

Figura 3.6: Configuració del planificador amb un arxiu *yaml*. Font: *Kubernetes* [72]

CONSTRUCCIÓ D'UN CONNECTOR

4.1 Motivació

Els fonaments amb els que ens basarem per a la creació d'aquesta personalització del selector mitjançant el connector sorgeixen d'una *Proposta de Millora de Kubernetes*¹ [74], i particularment el que fa referència al Marc de Treball de planificació (*KEP-624 Scheduling Framework*) [75], lògicament, patrocinat per *sig-scheduling*.

El marc de treball de planificació és el conjunt d'APIs dels connectors que s'afegeix al *kube-scheduler*. Els connectors s'inclouen en temps de compilació al planificador i aquestes APIs permeten implementar noves característiques de planificació com a connectors, canviant el comportament del planificador, mantenint el nucli de planificació simple i actualitzable (Vegeu secció 2.4.1).

Posteriorment, el `KubeSchedulerConfiguration` (Vegeu secció 3.2.2) permetrà habilitar, deshabilitar i reordenar els connectors.

Per fer això, a la *API del connector* hem de registrar i configurar els connectors per a que després puguem fer-hi referència a través de la interfície² [76] dels punts d'extensió. A la figura 4.1 es mostra quina forma tenen aquestes interfícies. Al marc de treball de planificació hi ha definit un tipus "patró" *interface* anomenat *Plugin* que fan servir a tots i cadascun dels connectors del planificador. Cada punt d'extensió serà també de tipus interfície a on anirà allotjats tots els connectors que emprarà el nostre planificador [77].

Al nostre connector l'anomenarem *NetworkTraffic* i afavorirà els Nodes en funció de la seva quantitat de tràfic de xarxa. Es tracta prioritzar els Nodes amb menor tràfic

¹Un Kubernetes Enhancement Proposals (KEP) és una forma de proposar, comunicar i coordinar nous esforços per al projecte Kubernetes. Es comparteix una idea amb la comunitat amb el patrocini d'un Special Interest Group (SIG) i si és acceptada se li assignarà un identificador únic per a facilitar la seva localització. Al final de ser graduada, arribarà a ser una millora a noves versions de K8s.

²En anglès *interface*: Tipus abstracte que no exposa la representació o estructura interna del seus valor o el conjunt bàsic d'operacions que té. Només revela alguns dels mètodes. Quan es té un valor d'un tipus d'interfície, no saps res sobre què és, només quins comportaments proporcionen els seus mètodes.

```
// Plugin is the parent type for all the scheduling framework plugins.
type Plugin interface {
    Name() string
}

// QueueSortPlugin is an interface that must be implemented by "QueueSort" plugins.
type QueueSortPlugin interface {
    Plugin
    // Less are used to sort pods in the scheduling queue.
    Less(*QueuedPodInfo, *QueuedPodInfo) bool
}

// PreFilterPlugin is an interface that must be implemented by "PreFilter" plugins.
// These plugins are called at the beginning of the scheduling cycle.
type PreFilterPlugin interface {
    Plugin

    PreFilter(ctx context.Context, state *CycleState, p *v1.Pod) *Status

    PreFilterExtensions() PreFilterExtensions
}
```

Figura 4.1: interface.go

de xarxa i puntuar cada node en funció del tràfic que genera. Els Nodes amb menor tràfic de xarxa seran els candidats (Vegeu Filtratge a 2.4) per a allotjar els Pods. Pel que com podem sospitar aquest connector ho implementarem per al punt d'extensió *score*.

Per aconseguir la informació del tràfic generat necessitarem fer consultes a cada Node en un rang de temps i guardar aquesta informació per després avaluar-la. Aquestes consultes les farem amb *Prometheus* [78].

4.2 Creació d'un connector personalitzat

Hem d'assolir una sèrie de premisses abans de començar a dissenyar:

1. Tenim un clúster de *Kubernetes* en funcionament. En aquest punt hem de destacar que per el correcte funcionament és necessari mantenir un criteri de versions entre el paquet del client K8s (client-go, apimachinery, etc.) i el repositori de *sig-scheduling* des d'on construirem el nostre connector. A la fi d'evitar problemes de compatibilitat el repositori disposa d'una matriu de compatibilitat per a poder triar-ne la versió correcta. [79].
2. Tenim instal·lat el llenguatge de programació *Golang*. Se aconsella una versió igual o superior a v1.17.
3. Tenim també instal·lat al nostre clúster el sistema de monitoreig *Prometheus* i configurat el *Node Exporter*.

Procediment a seguir

1. Clonarem el repositori *scheduler-plugins* a on tenim una sèrie de connectors ja creats i que ens poden servir com a exemples per començar i l'ubicarem al directori `$GOPATH/src/sigs.k8s.io`. Una altra opció podria ser la de clonar el repositori de *kubernetes* i afegir un connector nou o modificar dels que ja existeixen al directori `kubernetes/pkg/scheduler/framework/plugins/`, però trob una obra arriscada actuar directament damunt del repositori clonat *kubernetes* i el seu planificador nadiu en comptes de fer ús del repositori *sig-scheduling* ja que al final hauríem de dedicar esforços addicionals per mantenir "al dia" els canvis que es produeixen constantment al repositori principal de K8s. Havent-hi la possibilitat de clonar el repositori *scheduler-plugins* i independitzar els possibles canvis que puguin anant apareixent de mentres, optant per la primera opció. A la figura 4.2 ens mostra la sortida que obtindríem de la clonació del repositori.

```
andreuet@prestige-15:~/go/src$ cd $GOPATH/src/sigs.k8s.io.
andreuet@prestige-15:~/go/src/sigs.k8s.io$ git clone https://github.com/kubernetes-sigs/scheduler-plugins.git
Clonando en 'scheduler-plugins'...
remote: Enumerating objects: 12630, done.
remote: Counting objects: 100% (4931/4931), done.
remote: Compressing objects: 100% (2654/2654), done.
remote: Total 12630 (delta 2417), reused 3531 (delta 2146), pack-reused 7699
Recibiendo objetos: 100% (12630/12630), 22.63 MiB | 9.00 MiB/s, listo.
Resolviendo deltas: 100% (4689/4689), listo.
andreuet@prestige-15:~/go/src/sigs.k8s.io$
```

Figura 4.2: Resultat després de clonar el repositori de *scheduler-plugins*

2. Amb el repositori ja creat anirem al directori `.../scheduler-plugins/pkg/` i crearem un nou directori amb el nom del nostre connector `networkTraffic`. Aquí tindrè dos arxius:
 - a) `networkTraffic.go` : Comprèn la implementació de la interfície *ScorePlugin* del Marc de Treball del Planificador 2.4.1, concretament, que ha de fer el nostre connector quan es trobi al punt d'extensió *Score*.

```
382 // ScorePlugin is an interface that must be implemented by "Score" plugins to rank
383 // nodes that passed the filtering phase.
384 type ScorePlugin interface {
385     Plugin
386     // Score is called on each filtered node. It must return success and an integer
387     // indicating the rank of the node. All scoring plugins must return success or
388     // the pod will be rejected.
389     Score(ctx context.Context, state *CycleState, p *v1.Pod, nodeName string) (int64, *Status)
390
391     // ScoreExtensions returns a ScoreExtensions interface if it implements one, or nil if does not.
392     ScoreExtensions() ScoreExtensions
393 }
```

Figura 4.3: Interfície *ScorePlugin* de l'arxiu *interface.go*. Font: Kubernetes

- b) `prometheus.go` : que implementarà la lògica per a interactuar amb *Prometheus*.
3. Toca ara escriure pròpiament el codi font. Començarem amb la comunicació amb *Prometheus*, per després fer-ho amb la implementació de la *puntuació*.

a) **prometheus.go**: Per a poder comunicar-nos el primer que ens farà falta és definir una estructura per a que els Nodes puguin interactuar amb ell. Necessitam definir els camps. A la figura 4.4 podem veure com quedaria:

- `networkInterface` : A quina interfície de xarxa enviaran els Nodes la informació obtinguda.
- `timeRange` : Durada del mostreig.
- `address` : Nom del recurs (*endpoint*) del servei *Prometheus* al clúster.
- `api` : Emmagatzema el client de *Prometheus*, el qual està creat basant-nos en la `address` proporcionada.

```
// Handles the interaction of the networkplugin with Prometheus
type PrometheusHandle struct {
    networkInterface string
    timeRange         time.Duration
    address            string
    api                v1.API
}
```

Figura 4.4: Estructura principal per a l'arxiu prometheus.go

A continuació implementariem la consulta fent servir la suma del bytes rebuts a un període de temps per un Node a una interfície de xarxa específica.

```
18 ...
19 func (p *PrometheusHandle) GetNodeBandwidthMeasure(node string) (*model.Sample, error) {
20     query := getNodeBandwidthQuery(node, p.networkInterface, p.timeRange)
21     res, err := p.query(query)
22     ...
23     nodeMeasure := res.(model.Vector)
24     ...
25     return nodeMeasure[0], nil
26 }
27
28 func getNodeBandwidthQuery(node, networkInterface string, timeRange time.Duration) string {
29     return fmt.Sprintf(nodeMeasureQueryTemplate, node, networkInterface, timeRange)
30 }
31 ...
```

Figura 4.5: Extracte del llistat del codi prometheus.go

Al llistat 4.5 a la funció `GetNodeBandwidthMeasure` a la línia 19 li passam un nom de Node i ens retornarà la consulta de la mètrica (número de bytes) obtinguda d'aquest Node.

b) **networktraffic.go**:

El punt d'extensió *Score* comprèn dues fases:

- i. La primera fase s'anomena *puntuació* i s'utilitza per a classificar els Nodes que han passat la fase de filtrat. El planificador cridarà al `Score` de cada connector de puntuació per a cada Node.
- ii. La segona fase s'anomena *normalitzar puntuació* i s'utilitza per a modificar les puntuacions abans que el planificador calculi la classificació final dels Nodes. Hem de saber que aquesta puntuació ha de ser un valor entre 0 i 100 i s'han d'adaptar aquests valors a aquest interval, sinó, el cicle de planificació es cancel·laria.

Per tant, a l'interfície *ScorePlugin* que mostrarem a la Figura 4.3 és a on implementarem la funció del connector *Score* del Marc de Treball del Planificador.

```

48 func (n *NetworkTraffic) Score(ctx context.Context, state *framework.CycleState,
49     p *v1.Pod, nodeName string) (int64, *framework.Status) {
50     nodeBandwidth, err := n.prometheus.GetNodeBandwidthMeasure(nodeName)
51     if err != nil {
52         return 0, framework.NewStatus(framework.Error, fmt.Sprintf("error getting node bandwidth measure: %s", err))
53     }
54
55     klog.Infof("[NetworkTraffic] node '%s' bandwidth: %s", nodeName, nodeBandwidth.Value)
56     return int64(nodeBandwidth.Value), nil
57 }
58
59 func (n *NetworkTraffic) ScoreExtensions() framework.ScoreExtensions {
60     return n
61 }

```

Figura 4.6: Implementació de la funció *Score*

Com hem explicat abans, la funció *Score* es crida per a cada node i tornarà el total de bytes rebuts en un determinat període de temps. A la figura 4.6 a la línia 56 `nodeBandwidth.Value` té aquest valor.

Si el que realment necessitam és un número enter entre 0 i 100 necessitam de normalitzar la puntuació. No tan sols això, el Nodes amb major tràfic de xarxa tendran major puntuació, pel que si hem d'afavorir el que tenen menor tràfic de xarxa hem de capgirar aquesta llista. Amb la implementació de la funció *NormalizeScore* realitzarem aquestes tasques.

```

63 func (n *NetworkTraffic) NormalizeScore(ctx context.Context, state *framework.CycleState,
64     pod *v1.Pod, scores framework.NodeScoreList) *framework.Status {
65     var higherScore int64
66     for _, node := range scores {
67         if higherScore < node.Score {
68             higherScore = node.Score
69         }
70     }
71     for i, node := range scores {
72         scores[i].Score = framework.MaxNodeScore - (node.Score * framework.MaxNodeScore / higherScore)
73     }
74     klog.Infof("[NetworkTraffic] Nodes final score: %v", scores)
75     return nil
76 }

```

Figura 4.7: Implementació de la funció *NormalizeScore*

A la figura 4.7 a la línia 72 es normalitza la llista de tots els Nodes (`Scores[i]`) i a la vegada (amb la substracció) es cap gira. La constant `MaxNodeScore` val 100.

Posteriorment, necessitam declarar una nova estructura `NetworkTrafficArgs` que ens servirà per analitzar la configuració proporcionada a l'arxiu de configuració `KubeSchedulerConfiguration`. Aquests paràmetres de configuració no són uns altres que `networkInterface`, `timeRange` i `address` que vérem a la figura 4.4. Hem d'afegir al nostre codi una nova (*New*) funció que ens crei una instància del connector *NetworkTraffic*.

4. CONSTRUCCIÓ D'UN CONNECTOR

```
32 // New initializes a new plugin and returns it.
33 func New(obj runtime.Object, h framework.Handle) (framework.Plugin, error) {
34     args, ok := obj.(*config.NetworkTrafficArgs)
35     if !ok {
36         return nil, fmt.Errorf("[NetworkTraffic] want args to be of type NetworkTrafficArgs, got %T", obj)
37     }
38
39     klog.Infof("[NetworkTraffic] args received. NetworkInterface: %s; TimeRangeInMinutes: %d, Address: %s",
40         args.NetworkInterface, args.TimeRangeInMinutes, args.Address)
41
42     return &NetworkTraffic{
43         handle:      h,
44         prometheus:  NewPrometheus(args.Address,
45                               args.NetworkInterface,
46                               time.Minute*time.Duration(args.TimeRangeInMinutes)),
47     }, nil
48 }
```

Figura 4.8: Implementació de la funció *New*

4. Una vegada tinguem el codi, hem de declarar s'estructura `NetworkTrafficArgs`. Aquí és on s'adona que hi ha un nou connector i de quin tipus és.

Per tant, hem d'afegir la configuració a l'arxiu `types.go` que es troba a 3 llocs distints:

- `.../scheduler-plugins/apis/config/`
- `.../scheduler-plugins/apis/config/v1beta2/`
- `.../scheduler-plugins/apis/config/v1beta3/`

Aquesta configuració permetrà al planificador analitzar aquesta nova estructura des de `KubeSchedulerConfiguration` i fer servir la nova funció. Hem d'anar alerta en aquest pas per que la configuració de l'estructura ha de seguir un patró en concret `<Plugin Name>Args`, sinó, no ho descodificarà bé i el nostre connector fallarà.

```
...
// NetworkTrafficArgs holds arguments used
// to configure NetworkTraffic plugin.
type NetworkTrafficArgs struct {
    metav1.TypeMeta

    // Address of the Prometheus Server
    Address string
    // NetworkInterface to be monitored, assume that
    // nodes OS is homogeneous
    NetworkInterface string
    // TimeRangeInMinutes used to aggregate the network metrics
    TimeRangeInMinutes int64
}
```

Llistat del Codi 4.1: `types.go`

5. Amb aquestes noves estructures hem d'actualitzar els *arxius generats* [80] com el *DeepCopy* [81] i altres generadors de codi necessaris amb l'execució del guió (en anglès *script*) `.../scheduler-plugins/hack/update-codegen.sh`

```
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins$ ./hack/update-codegen.sh
go: creating new go.mod: module fake/mod
Generating deepcopy funcs
Generating defaulters
Generating conversions
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins$
```

Figura 4.9: Resposta a l'execució del guió `update-codegen.sh`

6. Si hem d'inicialitzar valors per al nostre connector ho haurem de fer als arxius `defaults.go` que es trobaran a `.../config/vbeta2/` i `.../config/vbeta3/`. Al nostre cas posam els valors que ens interessin. Al llistat del codi 4.2 definim el temps a 5 minuts i li posem el nom de la targeta de xarxa a on farà les consultes.

```
...
// SetDefaultNetworkTrafficArgs sets the default parameters
// for the NetworkTraffic plugin
func SetDefaultNetworkTrafficArgs(args *NetworkTrafficArgs) {
    if args.TimeRangeInMinutes == nil {
        defaultTime := int64(5) // 5 minuts de durada
        args.TimeRangeInMinutes = &defaultTime
    }

    if args.NetworkInterface == nil || *args.NetworkInterface == "" {
        netInterface := "ens192" // nom de la meua targeta de xarxa
        args.NetworkInterface = &netInterface
    }
}
```

Llistat del Codi 4.2: `defaults.go`

7. A continuació, haurem d'executar novament el guió del punt anterior i comprovar que en aquests mateixos directoris que ha quedat registrada la funció a dins de l'esquema (*scheme*) [82] als arxius `zz_generated.defaults.go`. Al llistat del codi 4.3 tenim una possible representació de com podria quedar.
8. Ara que tenim l'estructura d'arguments definida, hem de registrar la configuració en el Marc de Treball del Planificador. Per això haurem de declarar el nostre conector a l'arxiu `register.go` a on es registrarà. El projecte *scheduler-plugins* ja en té un quants connectors registrats, pel que ens serà més fàcil veure com es fa. Hem d'afegir `NetworkTrafficArgs` a cada cridada a la funció `addKnownTypes`. L'arxiu `register.go` es troba dins dels 3 directoris:

- `.../scheduler-plugins/apis/config/`

4. CONSTRUCCIÓ D'UN CONNECTOR

```
...
// RegisterDefaults adds defaulters functions to the given scheme.
// Public to allow building arbitrary schemes.
// All generated defaulters are covering - they call all nested defaulters.
func RegisterDefaults(scheme *runtime.Scheme) error {
    scheme.AddTypeDefaultingFunc(&NetworkTrafficArgs{},
        func(obj interface{}) {
            SetObjectDefaultNetworkTrafficArgs(obj.(*NetworkTrafficArgs))
        })
    return nil
}
func SetObjectDefaultNetworkTrafficArgs(in *NetworkTrafficArgs) {
    SetDefaultNetworkTrafficArgs(in)
}
```

Llistat del Codi 4.3: zz-generated.defaults.go

- `.../scheduler-plugins/apis/config/v1beta2/`
- `.../scheduler-plugins/apis/config/v1beta3/`

Al llistat del codi 4.4 podem veure un exemple de com quedaria:

```
...
// addKnownTypes registers known types to the given scheme
func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &CoschedulingArgs{},
        ...
        &NetworkTrafficArgs{}, // Aquí registram el nostre connector
    )
    return nil
}
...
```

Llistat del Codi 4.4: register.go

9. Ja ens queden un parell de passos més. El següent és registrar el connector i això ja ho feim a l'arxiu `scheduler-plugins/cmd/scheduler/main.go`.

Al llistat de codi 4.5 podem apreciar que dins de la funció principal `main()` afegim el nom i el constructor del nostre connector como arguments per a la funció `NewSchedulerCommand()`.

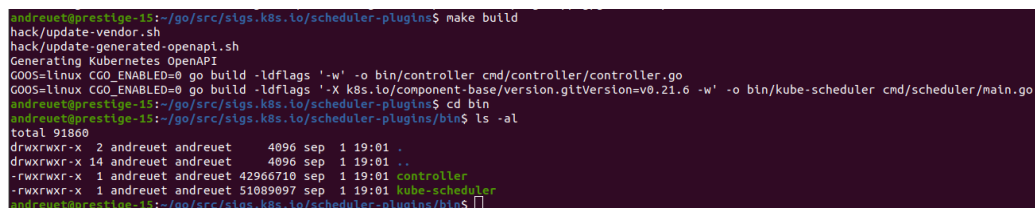
```

...
package main
import (
    ...    // Importam el paquet local networktraffic
           "sigs.k8s.io/scheduler-plugins/pkg/networktraffic"
           // Importam ../apis/config/scheme que inicialitzarà
           // el esquema amb totes les configuracions que
           // posarem als arxius ../scheduler-plugins/apis/config
           _ "sigs.k8s.io/scheduler-plugins/apis/config/scheme"
)
func main() {
    ...
           // Registre dels connectors personalitzats el Marc de Treball
           // Per poder ser configurats als perfils del planificador.
    command := app.NewSchedulerCommand(
        app.WithPlugin(coscheduling.Name, coscheduling.New),
        ...    // Aquí hi hauria la resta de connectors
        app.WithPlugin(networktraffic.Name, networktraffic.New),
    )
    ...
}

```

Llistat del Codi 4.5: main.go ha de tenir un aspecte semblant a això

- Finalment, per a la construcció del nostre arxiu executable de kube-scheduler podem crear una imatge local anant al directori `$GOPATH/src/sigs.k8s.io/scheduler-plugins` i teclejar `$ make build`.



```

andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins$ make build
hack/update-vendor.sh
hack/update-generated-openapi.sh
Generating Kubernetes OpenAPI
GOOS=linux CGO_ENABLED=0 go build -ldflags '-w' -o bin/controller cmd/controller/controller.go
GOOS=linux CGO_ENABLED=0 go build -ldflags '-X k8s.io/component-base/version.gitVersion=v0.21.6 -w' -o bin/kube-scheduler cmd/scheduler/main.go
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins$ cd bin
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins/bin$ ls -al
total 91808
drwxrwxr-x 2 andreuet andreuet 4096 sep  1 19:01 .
drwxrwxr-x 14 andreuet andreuet 4096 sep  1 19:01 ..
-rwxrwxr-x 1 andreuet andreuet 42966710 sep  1 19:01 controller
-rwxrwxr-x 1 andreuet andreuet 51089097 sep  1 19:01 kube-scheduler
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins/bin$

```

Figura 4.10: Implementació de la funció *New*

El que es mostra a la figura 4.10 després de l'execució del *make* farà que construeixi el binari executable del planificador al directori ubicat a `../scheduler-plugins/bin/`.

A partir d'aquí ja només ens queda desplegar el nostre planificador com a una de les possibles opcions que varem estar comentant al capítol 3, es a dir, substituint `kube-scheduler` pel nostre o, que convisqui el nostre planificador amb el nadiu.

En el meu cas, el desenvolupament ho estic realitzant amb *MicroK8s*, i cal comentar que després de demanar ajuda directament al personal de *Canonical* per fer funcionar

aquest connector a la distribució de K8s *MicroK8s* com un únic planificador, em confirmem que actualment no era possible (de manera trivial). No és pot iniciar *MicroK8s* sense arrancar el propi planificador nadiu degut a l'arquitectura particular d'aquesta distribució.

L'altre opció que al meu entendre trob més elegant, és la coexistència d'un altre planificador (vegeu la secció 3.4). En aquest cas, conviuen tot dos planificadors i la selecció de cada un d'ells es fa indicant-li al manifest del desplegament quin és el planificador que es vol fer servir.

Com ja vàrem comentar a la secció 3.4, tenim la possibilitat de instal·lar un planificador mitjançant l'ús del gestor d'aplicacions *Helm*. També comentarem que al mateix subprojecte *scheduler-plugins* hi ha una *chart* per a la instal·lació d'un planificador com a segon planificador amb documentació de com fer-ho i comprovar el seu funcionament.

4.2.1 Desplegament del connector amb un segon planificador

Per això partirem d'una sèrie d'antecedents que tenim establerts:

- Clúster de K8s *MicroK8s* en funcionament. Començarem a la versió v1.23 i a mitjans de maig vàrem canviar a v1.24 a tots els Nodes.
- Gestor de paquets *Helm 3* en funcionament. *MicroK8s* té una eina d'instal·lació de complements a on venen pre-instal·lats una sèrie de complements i un d'ells és *Helm 3* versió v3.8.0. Per fer-ho simplement ho habilitam amb la comanda:

```
$sudo microk8s enable helm3
```

- Eina de creació de contenidors *Docker* en funcionament i amb un compte creat per a pujar les meves imatges al registre de contenidors *Docker Hub*.

Es tracta de modificar el *chart* que té preestablert el subprojecte *scheduler-plugins* i substituir les imatges que venen per defecte i col·locar les nostres, a més de fer les modificacions pertinents a la configuració del nostre connector. Les passes són les següents:

1. Amb els dos arxius que hem construït

- `kube-scheduler`
- `controller`

ubicats al directori `.../scheduler-plugins/bin` crearem dues imatges locals amb l'eina *docker* i després les pujarem al servei de registre de repositoris *Docker Hub*.

- a) Cream un arxiu anomenat *Dockerfile* a on llegirem les instruccions que ens permetran crear la imatge.

Per a la imatge del arxiu `controller` l'arxiu *Dockerfile* serà similar substituint allà a on ens trobem *kube-scheduler* per *controller*.

```
FROM alpine:3.16
COPY ./kube-scheduler /bin/kube-scheduler

WORKDIR /bin

CMD ["kube-scheduler"]
```

Llistat del Codi 4.6: Dockerfile: Creació de la imatge del planificador

b) Cream la imatge del planificador amb la instrucció:

```
$docker build -t andreuet/kube-scheduler:v0.21.7 .
```

d'aquesta manera cream al nostre propi equip la imatge. Igualment que a l'apartat anterior per al controlador substituïm *kube-scheduler* per *controller*. Vegeu la figura 4.11 del resultat obtingut.

```
andreuet@prestige-15:~/pelardo/imagen-scheduler$ docker build -t andreuet/kube-scheduler:v0.21.7 .
Sending build context to Docker daemon 51.09MB
Step 1/4 : FROM alpine:3.16
--> 9c6f07244728
Step 2/4 : COPY ./kube-scheduler /bin/kube-scheduler
--> 313d687479b6
Step 3/4 : WORKDIR /bin
--> Running in e2e993e58084
Removing intermediate container e2e993e58084
--> 2193dc48f3b7
Step 4/4 : CMD ["kube-scheduler"]
--> Running in 0101ce9d3ad4
Removing intermediate container 0101ce9d3ad4
--> 55d2e7985783
Successfully built 55d2e7985783
Successfully tagged andreuet/kube-scheduler:v0.21.7
andreuet@prestige-15:~/pelardo/imagen-scheduler$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
andreuet/kube-scheduler	v0.21.7	55d2e7985783	9 seconds ago	56.6MB
andreuet/controller	v0.21.7	72222425fc66	About a minute ago	48.5MB
alpine	3.16	9c6f07244728	3 weeks ago	5.54MB
simulator-frontend	latest	31fa4bd562f0	2 months ago	330MB
simulator-server	latest	0f7051578694	2 months ago	342MB

Figura 4.11: imatges locals *Docker*

c) Iniciam sessió en el servei de registre de repositoris *Docker Hub* amb

```
$docker login
```

d) Pujam les imatges al meu compte de *Docker Hub*

```
$docker push andreuet/kube-scheduler:v0.21.7
```

Novament, per al controlador, tornam a substituir *kube-scheduler* per *controller*.

2. Partint del directori `$GOPATH/src/sigs.k8s.io/scheduler-plugins` com a origen, editarem l'arxiu `values.yaml` que està ubicat a:

```
../scheduler-plugins/manifests/install/charts/as-a-second-scheduler/
```

4. CONSTRUCCIÓ D'UN CONNECTOR

Al llistat de codi 4.7 substituïm les imatges que venen per defecte per les nostres que tenim al nostre registre a *Docker Hub*.

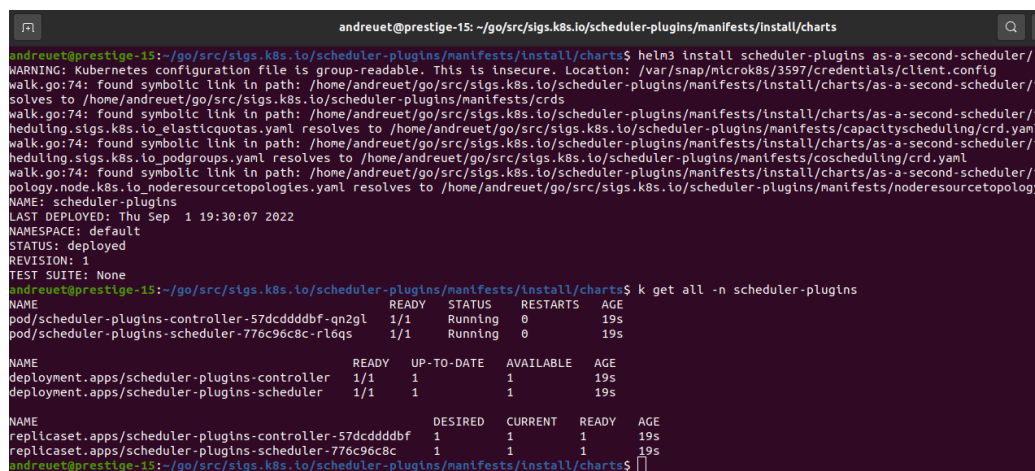
```
1 # Default values for scheduler-plugins-as-a-second-scheduler.
2
3 scheduler:
4   name: scheduler-plugins-scheduler
5   image: andreuet/kube-scheduler:v0.21.7
6   namespace: scheduler-plugins
7   replicaCount: 1
8
9 controller:
10  name: scheduler-plugins-controller
11  image: andreuet/controller:v0.21.7
12  namespace: scheduler-plugins
13  replicaCount: 1
```

Llistat del Codi 4.7: values.yaml: Imatges personalitzades

3. A continuació, anam al directori `../as-a-second-scheduler/templates` i editarem l'arxiu `configmap.yaml` per a personalitzar el comportament del nostre connector. Deixarem en funcionament el connector que ja hi és (*Coscheduling*), afegirem al punt d'extensió `score` el nostre connector *NetworkTraffic* i el parametritzarem a `profiles.pluginConfig`. El resultat ho podem veure al llistat del codi 4.8.
4. Tornant al directori `../manifests/install/charts/` ja podem executar la carta de navegació amb la comanda:

```
$ helm3 install scheduler-plugins as-a-second-scheduler
```

A continuació, comprovam que ja s'està executant el nostre planificador. Com mostra la figura 4.12 ja veiem que tant el *controller* com el *scheduler* es troben amb un STATUS *Running*.



```
andreuet@prestige-15: ~/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts$ helm3 install scheduler-plugins as-a-second-scheduler/
WARNING: Kubernetes configuration file is group-readable. This is insecure. Location: /var/snap/microk8s/3597/credentials/client.config
walk.go:74: found symbolic link in path: /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts/as-a-second-scheduler/t
solves to /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/crds
walk.go:74: found symbolic link in path: /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts/as-a-second-scheduler/t
heduling.sigs.k8s.io_elasticquotas.yaml resolves to /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/capacityscheduling/crd.yaml
walk.go:74: found symbolic link in path: /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts/as-a-second-scheduler/t
heduling.sigs.k8s.io_podgroups.yaml resolves to /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/coscheduling/crd.yaml
walk.go:74: found symbolic link in path: /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts/as-a-second-scheduler/t
pology.node.k8s.io_noderesourcetopologies.yaml resolves to /home/andreuet/go/src/sigs.k8s.io/scheduler-plugins/manifests/noderesourcetopology
NAME: scheduler-plugins
LAST DEPLOYED: Thu Sep 1 19:30:07 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts$ k get all -n scheduler-plugins
NAME                                READY    STATUS    RESTARTS   AGE
pod/scheduler-plugins-controller-57dcd4dbf-qn2gl  1/1      Running   0           19s
pod/scheduler-plugins-scheduler-776c96c8c-rl6qs  1/1      Running   0           19s

NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/scheduler-plugins-controller  1/1      1              1            19s
deployment.apps/scheduler-plugins-scheduler  1/1      1              1            19s

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/scheduler-plugins-controller-57dcd4dbf  1          1          1        19s
replicaset.apps/scheduler-plugins-scheduler-776c96c8c  1          1          1        19s
andreuet@prestige-15:~/go/src/sigs.k8s.io/scheduler-plugins/manifests/install/charts$
```

Figura 4.12: Instal·lació del segon planificador

4.2.2 Comprovació del funcionament del connector al segon planificador

Per a comprovar que el connector està funcionant com esperaven, desplegarem un Pod al nostre clúster. El clúster en qüestió està format pel Node mestre a on resideix el pla de control que també actua com a Node de treball, dues màquines virtuals creades amb l'aplicació *VirtualBox* [83] allotjades a l'equip a on tenim el Node mestre i un ordinador físic més. A continuació, a la taula 4.1 mostrem amb un poc més de detall la nostra infraestructura:

Clúster MicroK8s				
Nom del Node	Adreça IP	Sistema Operatiu	Infraestructura	Rol
prestige-15	192.168.1.133	Ubuntu 20.04	Portàtil	master//worker
vm-nodo-1	192.168.1.141	Ubuntu 20.04	Màquina Virtual	worker
vm-nodo-2	192.168.1.139	Ubuntu 20.04	Màquina Virtual	worker
pc-ubuntu	192.168.1.140	Ubuntu 22.04	Ordinador sobretaula	worker

Taula 4.1: Infraestructura del clúster

Per a la realització de les proves del correcte funcionament del nostre planificador realitzarem les següents passes:

1. Tenim creada la definició de un Pod per a que es desplegui al nostre planificador. Com podem veure a la línia 8 del llistat del codi 4.9 denotam el nom del nostre planificador que a la secció anterior hem desplegat.
2. Desplegam el Pod al nostre clúster amb la comanda:

```
$ kubectl apply -f nginx-second-sched.yaml
```

3. Comprovam que s'ha desplegat i s'està executant el nostre Pod. A la figura 4.13 podem veure que:

```
andreut@prestige-15:~/pelardo$ k get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
pi-njpdv      0/1     Error     0           70d   10.1.70.26    prestige-15   <none>           <none>
pi-8vx7n      0/1     Error     0           70d   10.1.70.17    prestige-15   <none>           <none>
pi-fsjkn      0/1     Error     0           70d   10.1.70.29    prestige-15   <none>           <none>
pi-pgkcs      0/1     Error     0           70d   10.1.70.59    prestige-15   <none>           <none>
pi-7bf5c      0/1     Error     0           70d   10.1.70.37    prestige-15   <none>           <none>
maradb-deployment-858b7d94fb-dfjw8  1/1     Running   33 (3h29m ago)  87d   10.1.70.40    prestige-15   <none>           <none>
nginx         0/1     ContainerCreating  0           16s   <none>        vm-nodo1      <none>           <none>

andreut@prestige-15:~/pelardo$ k get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP            NODE          NOMINATED NODE   READINESS GATES
pi-njpdv      0/1     Error     0           70d   10.1.70.26    prestige-15   <none>           <none>
pi-8vx7n      0/1     Error     0           70d   10.1.70.17    prestige-15   <none>           <none>
pi-fsjkn      0/1     Error     0           70d   10.1.70.29    prestige-15   <none>           <none>
pi-pgkcs      0/1     Error     0           70d   10.1.70.59    prestige-15   <none>           <none>
pi-7bf5c      0/1     Error     0           70d   10.1.70.37    prestige-15   <none>           <none>
maradb-deployment-858b7d94fb-dfjw8  1/1     Running   33 (3h29m ago)  87d   10.1.70.40    prestige-15   <none>           <none>
nginx         1/1     Running   0           21s   10.1.234.81   vm-nodo1      <none>           <none>

andreut@prestige-15:~/pelardo$ k get events -A
NAMESPACE LAST SEEN    TYPE     REASON      OBJECT        MESSAGE
default    <invalid> Normal    Scheduled    pod/nginx     Successfully assigned default/nginx to vm-nodo1
default    30s       Normal    Pulling      pod/nginx     Pulling image "nginx:1.10"
default    16s       Normal    Pulled       pod/nginx     Successfully pulled image "nginx:1.10" in 13.969058205s
default    15s       Normal    Created      pod/nginx     Created container nginx
default    15s       Normal    Started      pod/nginx     Started container nginx

andreut@prestige-15:~/pelardo$
```

Figura 4.13: Desplegament d'un Pod amb el planificador personalitzat

- a) Comprovam els Pods que tenim al nostre clúster. Inicialment, el nostre Pod (*nginx*) encara està a un estat de Creació de Contenidor (*ContainerCreating*).

4. CONSTRUCCIÓ D'UN CONNECTOR

- b) El nostre planificador ha vinculat el Pod *nginx* al Node *vm-nodo1* i passa a l'estat de Execució (*Running*).
 - c) Com a darrera part, comprovam els events generat en aquest període desde que es vincula al Node fins que arranca.
4. Paral·lelament per fer la comprovació de que realment l'ha desplegat allà a on tocava tenim en execució des d'un navegador una instància de *Prometheus* a on podem fer-li consultes com les consultes que feïm des del connector *NetworkTraffic*. Per ser el més fidel possible a comportament del connector *NetworkTraffic*, llançem la mateixa consulta per veure quin és el valor que en retorna:

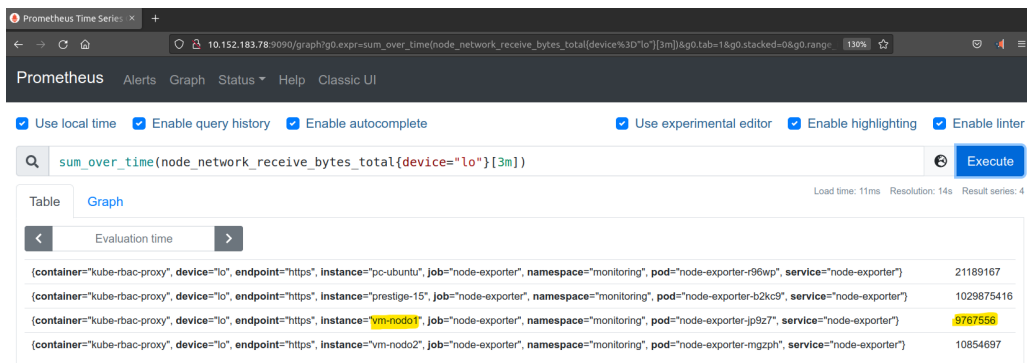


Figura 4.14: *Prometheus*. Consulta tràfic de xarxa a cada Node del clúster

A la figura 4.14 podem veure una sèrie de coses que val la pena comentar:

- L'adreça URL a la barra de direccions es correspon amb el paràmetre *prometheusAddress* de l'arxiu *configmap.yaml* per desplegar el nostre planificador. Línia 44 del llistat del codi 4.8.
- La consulta en qüestió fa la suma en el període de temps indicat (3 minuts), del total de bytes rebuts al nodes de xarxa que te com a *device="lo"*. Línia 45 del llistat del codi 4.8.
- El resultats de la consulta són exactament tots el Nodes del clústers i a la banda dreta podem veure l'acumulat de bytes totals de cada un d'ells.

Com és d'esperar el Node amb menor tràfic de xarxa es *vm-nodo1* pel que serà el millor puntuat i candidat a ser triat per a desplegar el Pod com realment ocorre.

La resta de proves realitzades per a comprovar de quina manera responia el planificador han estat les següents:

- Desplegament d'una aplicació (*nginx* també) amb 3 rèpliques.
- Desconnexió d'un Node amb poc tràfic.
- Desconnexió de dos Nodes de manera intempestiva .
- Executem diversos manifests amb distintes càrregues de treball, amb un interval molt curt d'execució entre cadascun d'ells.

El resultat obtingut en tots els casos ha estat l'esperat, pel que el nostre planificador funciona correctament.

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: scheduler-config
5    namespace: {{ .Values.scheduler.namespace }}
6  data:
7    scheduler-config.yaml: |
8      apiVersion: kubescheduler.config.k8s.io/v1beta1
9      kind: KubeSchedulerConfiguration
10     leaderElection:
11       leaderElect: false
12     profiles:
13     - schedulerName: scheduler-plugins-scheduler
14       plugins:
15         queueSort:
16           enabled:
17             - name: Coscheduling
18           disabled:
19             - name: "*"
20         preFilter:
21           enabled:
22             - name: Coscheduling
23         permit:
24           enabled:
25             - name: Coscheduling
26         score:
27           enabled:
28             - name: NetworkTraffic
29           disabled:
30             - name: "*"
31         reserve:
32           enabled:
33             - name: Coscheduling
34         postBind:
35           enabled:
36             - name: Coscheduling
37         pluginConfig:
38         - name: Coscheduling
39           args:
40             permitWaitingTimeSeconds: 10
41             deniedPGExpirationTimeSeconds: 3
42         - name: NetworkTraffic
43           args:
44             prometheusAddress: "http://10.152.183.78:9090/"
45             networkInterface: "lo"
46             timeRangeInMinutes: 3
```

Llistat del Codi 4.8: configmap.yaml: Personalització del connector *NetworkTraffic*

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5    labels:
6      app: nginx
7  spec:
8    schedulerName: scheduler-plugins-scheduler
9    containers:
10 -   name: nginx
11     image: nginx:1.10
```

Llistat del Codi 4.9: nginx-second-sched.yaml

DISCUSSIÓ

De fa ja prop d'una dècada, *Kubernetes* és una de les majors revolucions a la indústria del desenvolupament de programari, garantit per ser un sistema de codi obert i pel suport de companyies com IBM, Microsoft, RedHat o Google entre d'altres. La seva evolució ha estat imparable i no atura de créixer, fonamentant cada dia més i més projectes demandats tant pel sector empresarial com per l'educatiu. Uns requeriments, que cada vegada més s'allunyen de les típiques càrregues de treball com podrien ser les aplicacions web, clients de correu o televisió a la carta. Els clústers de K8s a dia d'avui són capaços de administrar recursos tant dispars com Unitats de Processament de Gràfics (en anglès Graphics Processing Unit (GPU)), Unitats de Processament de Tensors (en anglès Tensor Processing Unit (TPU)), Unitats de Processament Neuronal (en anglès Neural Processing Unit (NPU)), Computació a la boira i Computació Perimetral per citar-ne un parell.

Tot l'anteriorment comentat, ha donat peu a que a l'existència d'un ampli ventall de projectes amb la finalitat d'atendre a totes aquestes “noves” càrregues de treball, no tant per no poder suportar-les sinó més bé, com, i a on ubicar-les, tasca enterament confiada al planificador. Tot i que aquests projectes es fonamenten en l'ecosistema de K8s solen ser tercers qui desenvolupen aquestes idees aprofitant les necessitats del mercat.

Simultàniament, el projecte *Kubernetes* advertint aquesta necessitat, tampoc ha deixat d'evolucionar i ha vist aquests inconvenients com una oportunitat per a millorar el seu planificador, concentrant esforços en un grup dedicat única i exclusivament al tema de la planificació. Aquest col·lectiu (*SIG Scheduling*), a més de impulsar contínues millores, des de la seva aparició han estat rebent propostes, problemes/errades (en anglès *issues*) dels vertaders consumidors d'aquest component tan específic com és el planificador (*kube-scheduler*).

És tal la rapidesa evolutiva en que es produeixen aquests canvis que resulta prou complicat estar completament al corrent de les contínues transformacions que ha expe-

rimentat *kube-scheduler* des de la seva creació. Si seguim un poc el desenvolupament cronològicament es pot apreciar que, funcionalitats que pareixien quasi impensables actualment s'hagin graduades i d'altres que han quedat en desús directament i s'han eliminat. Des de que començà aquest treball hem passat per 3 versions distintes de K8s (v1.23 a v1.25) i en aquest període han aparegut 10 noves característiques de millora i altres tantes deprecacions o regressions només per a la planificació [84].

Com s'ha descrit al capítol anterior, el planificador ens brinda una sèrie de possibilitats per a ubicar els nostres Pods a uns Nodes determinats. Comentarem a grans trets els casos d'ús, avantatges i desavantatges de cada una d'aquestes estratègies:

- **Nom de Node:** No deixa lloc a confusió, si tenim un clúster petit (pocs Nodes) i les càrregues de treball sabem com la volem repartir perquè coneixem de cada Node quin tipus de càrrega està suportant, aquesta, és una opció evident i molt senzilla. Per contra, si no coneixem l'arquitectura del clúster, les dimensions comencen a ser considerables o si el Node al que li volem assignar les càrregues de treball no disposa dels recursos necessaris (processador, memòria), l'emplaçament d'aquest Pod no es durà a terme i molt probablement s'elimini.
- **Selector de Nodes:** Etiquetam cada Node amb les característiques que els definim: millor, localització, capacitat, funcionalitat o qualsevol altre camp que se'ns ocorri. A l'hora de crear el nostre Pod bastarà especificar quina/es etiqueta/es de Node volem que tenguim el nostre Node destí, la resta, ja s'encarregarà el planificador, al final, aconseguim que el Pod sigui assignat al Node que tenguim la mateixa etiqueta.

Aquest mètode ja ens dóna una certa flexibilitat que no ens permetia l'anterior cas. Les limitacions arriben quan ens plantejam casos de que vull emplaçar el meu Pod a un Node de *tamany: no petit, tamany: mitjà o gran*. Aquests tipus d'expressions lògiques no les pot gestionar el Selector de Nodes.

Els casos d'ús no deixen de ser bastant restrictius. Assignació de Pods a Nodes amb etiquetes específiques, no admeten els operadors lògics. En canvi, són molt senzills d'emprar i requereixen pocs canvis a l'hora de declarar l'especificació del Pod.

- **Afinitat i Antiafinitat dels Nodes:** La funció d'afinitat de Nodes és una millora qualitativa en comparació amb l'enfocament d'emplaçament de Pods amb selector de Nodes i és una bona pràctica fer-ne ús d'aquesta estratègia en comptes de les anteriorment comentades. Ofereix un llenguatge d'afinitat expressiu que utilitza operadors lògics (In, NotIn, Exist, NotExist) que brinden un control detallat sobre la ubicació dels Pods. També admet regles de planificació "suaus" i "dures" que permeten controlar el rigor de les restriccions d'afinitat de Nodes segons els requisits de cada usuari.

Els casos d'ús a on es podria fer servir aquest mètode serien:

- Distribució de Pods en diferents zones de disponibilitat per a millorar la resiliència i disponibilitat de les aplicacions.

-
- Assignació de Nodes per a Pods amb un ús intensiu de memòria. Es podrien tenir alguns Nodes dedicats a Pods amb menys ús intensiu de processador i un o dos Nodes amb més processador i memòria dedicats a Pods amb un ús intensiu de memòria. D'aquesta manera, evitem que els Pods no desitjats consumeixin recursos dedicats a altres Pods.

L'afinitat de Nodes ens permet la implementació de localitat de dades, executant el Pods a Nodes amb programari dedicat. Per contra, requereix de la modificació dels Pods existents per canviar el seu comportament.

- **Afinitat i Antiafinitat dels Pods:** Definir si un Pod determinat ha de planificar-se o no a un Node en particular en funció de les etiquetes dels altres Pods que ja s'estan executant en aquest Node és el següent mètode i els casos d'ús més probables podrien ser:

- Ubicar els Pods d'un servei o treball en particular a la mateixa zona de disponibilitat.
- Ubicar els Pods de dos serveis dependents entre sí a un Node per a reduir la latència de xarxa entre ells. Un exemple molt clar seria una clàssica aplicació web amb una interfície gràfica d'usuari (GUI / Frontend) i una interfície de programació d'aplicacions (API / Backend).
- Com a exemple de cas d'antiafinitat: que vulguem evitar que els Pods de dades d'algunes bases de dades visquin al mateix Node per evitar el punt únic de fallida (en anglès Single Point of Failure (SPOF)). Distribuïm els Pods d'un servei entre Nodes o zones de disponibilitat per a reduir les fallides.
- Un altre cas d'antiafinitat: No deixar planificar els Pods d'un servei en particular als mateixos Nodes que els Pods d'un altre servei que pugui interferir amb l'acompliment dels Pods del primer servei.

L'afinitat de Pods ens permet col·locar els Pods en el servei codependent, el que permet la localitat de les dades. L'antiafinitat de Pods ens permet l'alta disponibilitat (a través de la distribució de Pods), evitant la competència entre serveis pels recursos.

Per contra, requereix de la modificació dels Pods existents per canviar el seu comportament igual que ocorre al cas de l'Afinitat i Antiafinitat dels Nodes.

- **Prioritat i Preferència dels Pods:** Per a garantir la disponibilitat dels Pods crítics es poden crear una jerarquia de nivells de Pods amb prioritats. Quan hi ha una mancança de recursos *kubelet* intentarà eliminar els Pods de baixa prioritat per posar els Pods amb una prioritat més alta.

Aquest mètode pot ser de utilitat quan tenim càrregues de treball específiques que requereixen d'alta disponibilitat, inclús si poden aparèixer mancança de recursos o si tenim aplicacions en producció que no desitjam que s'eliminin com recollidors de mètriques, agents de registre, serveis de pagament, aquesta opció és la més adequada.

- **Restriccions de repartiment de la topologia del Pod:** La necessitat de distribuir els Pods de manera uniforme a tot el clúster per aconseguir una alta disponibilitat i una utilització eficient de tots els recursos és una pràctica comú a tots els administradors de sistemes. Aquesta funció (de relativa nova aparició amb la versió v1.19), ha estat dissenyada per omplir aquest buit. Per contra, la reducció d'un desplegament (en anglès *scaling down*) no garantirà que es mantengui aquesta distribució uniforme i fins i tot pot desequilibrar la resta de Pods.
- **Previsions i Toleràncies:** Les previsions juntament amb les toleràncies permeten un control més detallat damunt les preferències i l'antiafinitat de Pods en Nodes personalitzat amb operadors lògics. També ens permetrà controlar el comportament del Pod als Nodes que experimentin problemes com la falta de disponibilitat de xarxa, baixa capacitat de disc, memòria i/o processador. El cas més típic d'ús d'aquesta estratègia és:
 - Nodes amb un maquinari específic, per exemple GPU, i desitjam desallotjar els Pods que no necessiten aquest maquinari i atreure els Pods que si els hi fa falta. Podem limitar la quantitat de Nodes en el que planificar Pods mitjançant l'ús d'etiquetes i afinitat de Nodes, aplicar previsions (*Taints*) a aquest Nodes i després afegir les toleràncies als Pods.

No es requereix la modificació dels Pods existents, admet el desallotjament automàtic de Pods sense la tolerància requerida i admet diferents efectes per als *Taints*. Per contra, no admet la sintaxi que s'emptra a l'afinitat de Pods i Nodes que resulta tan fàcil d'entendre degut a la seva expressivitat.

- **Un planificador personalitzat:** Si encara així, després de totes aquestes estratègies que ens ofereix *kube-scheduler* no trobam la manera de resoldre les nostres necessitats, sempre podrem fer un planificador a mida. La personalització del planificador, no només implica **uns coneixements avançats en programació i de l'arquitectura interna del planificador de K8s**. A més a més, hem d'adquirir els coneixement necessaris per fer ús d'eines de tercers. Particularment, per a la creació d'aquest component hem fet servir:
 - Sistema de monitoreig basat en mètriques *Prometheus*.
 - Creació dels contenidors que inclouran la imatge del nostre planificador amb *Docker*.
 - Programari de control de versions de codi *GitHub*.
 - Llenguatge de programació *Go*.

Cal mencionar que totes les estratègies presentades per assignar les nostres càrregues de treball als Nodes, llevat del planificador personalitzat són funcionalitats que per defecte estan incloses a *kube-scheduler*.

Queda encara un llarg camí per recórrer, però els fonaments ja estan establerts. A títol personal he trobat a faltar:

-
- **Documentació:** Es parla molt de tota la documentació que hi ha disponible i en certa manera és cert quan parlem de *Kubernetes*, però quan ho feim de la planificació, la resposta no està tan clara:
 - **De la programació de connectors** per al planificador, o de la programació pròpiament en *Go* per a K8s només he trobat un llibre interessant [57] i és de l'any 2019. Ja sabem com va tot això, podem dir sense equivocar-nos que està obsolet.
 - **Substitució del planificador nadiu:** La inicialització del planificador es realitza amb els *daemons* o servei init del SO (*systemd*). Aquest ha estat el nostre cas. La distribució de K8s és *MicroK8s* i el seu planificador és un procés controlat pel SO. Per complicar-ho encara un poc més, aquesta distribució té integrats *kube-scheduler*, *kube-controller-manager*, *kube-proxy*, *kubelet* i els serveis de *kube-apiserver* a un sol procés *daemon* que s'anomena *kubelite* i tots aquests components que ho integren, s'executen com a subprocesos. Per solventar-ho, la primera passa va ser una lectura exhaustiva de la documentació disponible a la xarxa (pàgina oficial i altres pàgines alternatives) com és un tema que no té tanta difusió, la informació al respecte és molt vaga i no obtingué res. La segona passa va ser demanar informació directament a *Canonical MicroK8s* [85], i em vaig trobar que ni als propis enginyers que varen desenvolupar *MicroK8s* els hi resulta fàcil documentar clarament les particularitats d'aquesta distribució.
 - Encara que ja existeixen grups d'interès especial dedicats als multiclústers [86], les pinzellades que se li estan donant a la planificació de Pods en aquests entorns encara són molt verds. Tornem a la joventut de l'ecosistema.

CONCLUSIÓ

L'objectiu del projecte que consistia a fer un estudi detallat de les diferents opcions de particularitzar el planificador d'emplaçaments existent de l'ecosistema de Kubernetes s'ha assolit completament. El treball s'ha conduït fent una introducció a l'ecosistema i als seus principals components juntament amb una detallada descripció de les peces que el planificador de Kubernetes fa per acomplir la tasca d'assignar Pods a Nodes. A continuació, s'ha especificat i argumentat quines són els avantatges i desavantatges de les estratègies disponibles. Les estratègies analitzades són totes les existents: Nom de Node, Selector de Nodes, Afinitat i Antiafinitat dels Nodes, Afinitat i Antiafinitat dels Pods, Prioritat i Preferència dels Pods, Restriccions de repartiment de la topologia del Pod, Previsions i Toleràncies, i finalment, la creació d'un planificador a mida.

6.1 Opinió personal de l'evolució del TFG

En el present projecte he tractat de fer un estudi detallat de les diferents opcions que tenim de particularitzar el planificador de *Kubernetes* (requisit propi del projecte) juntament amb una implementació d'un planificador totalment personalitzat que considerava criteris no funcionals (una interfície externa i quantificable) dels Nodes per desplegar les nostres aplicacions.

Inicialment, ha estat necessari conèixer i entendre l'abast del projecte. Una vegada entès, he realitzat una recerca d'articles d'estudi dels termes clau com: Edge/Fog Computing, Cloud Computing, virtualització, contenidors i Kubernetes. A més, per a l'adquisició d'aquests coneixements he hagut de realitzar dos cursos: IBM CC0201EN - Introduction to Containers, Kubernetes and OpenShift i LinuxFoundationX LFS158x - Introduction to Kubernetes que m'han permès redactar la **Introducció**.

Al començament del projecte, decideixo instal·lar la distribució *Canonical Microk8s* versió v1.23 després de diferents reunions amb el tutor i d'estudiar les distribucions. Com no disposava d'una ampla infraestructura, vaig haver de fer ús del meu maquinari disponible i del programari de virtualització *VirtualBox* per a la construcció, acon-

seguint un clúster de 4 Nodes: 1 ordinador portàtil (master/worker), 1 ordinador de sobretaula (worker) i 2 màquines virtuals (workers) instal·lades al portàtil per disposar de més capacitat per suportar-les. A partir d'aquí començ a realitzar proves en real de tot l'après fins ara.

A continuació, he realitzat un estudi detallat de l'arquitectura de K8s i el seu planificador (*kube-scheduler*). Personalment aquest punt m'ha suposat invertir pràcticament una quarta part del Treball Final de Grau (TFG). Amb aquesta tasca realitzada he pogut redactar el capítol **Coneixement Actual**.

Simultàniament a l'estudi del planificador he hagut d'accedir als repositoris de K8s i de *K8s SIG Scheduling* i he hagut d'aprendre el llenguatge de programació *Go* amb la instal·lació de la seva versió v1.16 per a començar a entendre el contingut d'aquests repositoris i familiaritzar-me en la codificació, necessaris per a la consecució del projecte. Degut a una sèrie de problemes de compatibilitat amb K8s a l'hora de compilar petits projectes que hi havia de demostració, vaig acabant esbrinant que l'actual K8s dóna menys problemes si la versió era v1.17 o superior. Aquí substitueixo la versió a la v1.18. Això em genera més d'una setmana d'estar quasi aturat en l'aprenentatge del llenguatge de programació. Amb totes aquestes tasques realitzades em permeten redactar la part dedicada al planificador nadiu, a les estratègies que té per defecte, el seu marc de treball de planificació i els connectors inclòs al capítol **Accions per a la modificació de la selecció i emplaçament de pods/contenidors**.

Per tot l'exposat fins ara donam resposta a la primera part dels objectius que volia aconseguir que era l'estudi detallat de les diferents opcions que tenim de particularitzar el planificador de *Kubernetes*.

A continuació, cal ressaltar la importància de la lectura de l'article de Julio Renner "K8S - Creating a kube-scheduler plugin", a on es perfilen els punts més importants del capítol **Construcció d'un connector**. També he hagut de fer un estudi de l'eina de monitoreig de mètriques *Prometheus* necessària a la codificació del connector i familiaritzar-me amb l'interfície que té per a realitzar les consultes a la API de K8s i als mòduls externs (*node-exporter*) per a la posterior comprovació del correcte funcionament. La dificultat d'aquest capítol ha residit bàsicament en l'antiguitat de l'article. Si bé l'article es publicà al juliol de 2021, les contínues actualitzacions que sofert K8s en un any han estat suficients per a que hagi ralentit la fase de disseny. Ha estat necessària la cerca d'informació de temes molts específics del disseny i construcció amb llenguatge de programació *Go*. I per una altra banda, la metodologia que es segueix per a la compilació dels repositoris a la comunitat de K8s. Pentura aquí, una mica més de coneixement del llenguatge de programació i d'altre, el poc temps que tingut per endinsar-me en aquest món no han estat suficients per absorbir tota la informació que qualsevol altre amb unes bases molt més sòlides hagués pogut resoldre sense tanta dificultat.

Una vegada superades aquestes traves i aconseguir tenir un arxiu executable del planificador va aparèixer un nou problema al intentar desplegar aquest nou planificador al meu clúster. En concret, les particularitats de la distribució *Canonical MicroK8s* m'ha impossibilitat substituir el planificador nadiu pel meu. No trobant cap altre sortida, vaig haver de demanar ajuda tant a la comunitat *K8s SIG Scheduling* com a *Canonical MicroK8s*. Aquests darrers, em deien que provés una cosa o l'altra i al final m'acabaven remetent a un enllaç de la seva documentació. No em varen aclarir massa més del que

jo ja sabia. Aquest obstacle ho he acabat veient com a una oportunitat per aconseguir mètodes alternatius per a fer funcionar el meu planificador.

K8s SIG Scheduling després de 4 mesos de cap resposta i que el *k8s-Prow-Robot* afegís el flag de “lifecycle/stale” i a punt per tancar-lo, Huang Wei, el Copresident de *K8s SIG Scheduling* em va contestar amb una resposta que ja no tenia sentit en aquest punt del projecte, però vaig aprofitar que estava en contacte amb ell per a poder demanar-li sobre les branques i l'ús de les *cartes de navegació* de l'eina *Helm* que tenen al seu repositori. De l'intercanvi d'informació vaig resoldre part del problema i a la vegada ha donat peu per l'apertura d'una millora en la personalització dels connectors [87].

De tot el projecte, aquest període ha estat el moment en que he arribat a perdre un poc l'esperança d'arribar a aconseguir els objectius que ens havíem marcat. Al final, la coexistència de *kube-scheduler* amb un planificador personalitzat ha estat l'opció més intel·ligent que m'ha permès avaluar i comparar a la vegada ambdues planificadors.

Amb el desplegament totalment funcional i amb petites correccions ben localitzades a la configuració del connector que em van apareixent mentre ho vaig provant, finalment aconseguixo posar en marxa les càrregues de treball amb el planificador personalitzat i emplaçant-les a on s'espera que han d'ubicar-se. Per fer això, genero una sèrie de manifests per a la creació de càrregues de treball i simultàniament amb l'interfície de *Prometheus* vaig comprovant a on han d'anar cada una d'elles.

Per tant, queda patent el correcte funcionament d'un planificador personalitzat i aconseguida de manera satisfactòria la segona part dels objectius del projecte que era la implementació d'un planificador totalment personalitzat que considerava criteris no funcionals dels Nodes per desplegar les nostres aplicacions.

De cara a un futur projecte, m'agradaria arribar a provar aquest mateix projecte a un clúster real i amb Nodes geogràficament dispersos, com podria ser als entorns i seus de la *UIB*. I anant un poc més enllà, poder interconnectar d'aquest clúster experimental amb altres universitats i reproduir un escenari més heterogeni a la fi d'emular arquitectura *MultiClúster* a on la *Computació al Perímetre/Boira* i tenen molt a dir ja que involucra clústers locals que processen dades i envien els resultat a clústers regionals, que a la vegada realitzen operacions i envien els resultat a un clúster central.

6.1.1 Opinió Personal

Han passat més de vint anys per tornar a reprendre els estudis i d'ençà, molt han canviat les coses en aquesta disciplina. Motiu de més per posar-me al dia a tot el que envolta aquest món. Han estat moltíssimes coses que he hagut d'aprendre en tan poc temps, i ho agraesc, ha valgut la pena. Aquest projecte m'ha suposat un desafiament en majúscules com mai m'hagués esperat, i personalment ja que tornava a reprendre els estudis, si havia de fer alguna cosa, m'havia de suposar un repte per fer-ho bé.

Va ser un dia a una classe d'arquitectures avançades amb n'Isaac que va parlar per primera vegada del contenidors i aquí ja em va sonar la campana. Un tema totalment desconegut per jo i que em va cridar l'atenció. El que no m'esperava era tot el que li envoltava per darrera.

Han estat 7 mesos de vertadera bogeria en el que partint de res en qüestió de coneixements previs i he hagut d'adquirir els següents coneixements:

- **Contenidors i Kubernetes:** A més dels dos cursos que vaig fer i la quantitat de proves, exemples exercicis que vaig executar, també vaig haver de patir la degradació del clúster i total mal funcionament degut a les proves de substitució del planificador i vaig haver d'instal·lar-lo dues vegades més, sa darrera ja ho vaig fer amb la versió v1.24.
- **Llenguatge de Programació Go:** Si tenc clar que no tenc un total domini d'aquest llenguatge he adquirit els coneixements suficients per a la realització d'aquest projecte i a partir d'ara projectes futurs.
- **Programari de control de versions de codi *GitHub*:** Aquí, a més de conèixer únicament les comandes per fer funcionar aquesta eina, hem hagut de conèixer la jerarquia que s'estableix als repositoris d'una organització com a K8s com la relació amb comunitats afins com (*K8s SIG Scheduling*), la política de governança, seguiment de millores, reunions comunitàries, registre de canvis, dependències d'altres proveïdors (en anglès *vendor*), peticions per difondre a la comunitat (en anglès *pull request*), i moltes altres que si ets un iniciat del tema et pot arribar a aclaparar.
- **Editor col·laboratiu en LaTeX *Overleaf*:** Si bé és cert que la dificultat per utilitzar aquest editor és gran i la quantitat de temps dedicada és similar, el petit curset que ens va impartir n'Ignasi en poc més d'una hora i la plantilla disponible per a la redacció del treball ha facilitat molt aquesta tasca i no deixa lloc a dubtes que la qualitat obtinguda està a l'altura dels esforços invertits. Tot i coneixent bé el processador de textos Word, no m'atreviria mai a realitzar cap treball acadèmic amb ell. La seva qualitat en aquest àmbit per jo deixa molt que desitjar.
- Totes les **eines que envolten l'ecosistema de K8s** que he emprat per a la realització del TFG com són: *kube-scheduler simulator*, *Prometheus* amb les funcions de *node-exporter*, *Graphana*, *Lens IDE*, *Helm*, *Docker* i el seu servei de repositori *Docker Hub* i un poc fora d'aquest ecosistema però que cal mencionar es l'eina *Virtualbox* per crear màquines virtuals i que mai havia fet servir.

Per concloure només em queda agrair a n'Isaac tot el suport i ànims que m'ha donat durant la realització del treball que d'un principi vaig veure com una obra titànica i m'ha fet veure-la des d'una altre perspectiva alleugerant-me tot aquest pes.



APÈNDIXS

A.1 Selector de Nodes. Exemple

1. Mostrem els Nodes del nostre clúster, junt amb els seus etiquetes (columna amb el nom de *LABELS*):

```
kubectl get nodes --show-labels
```

Amb una sortida pareguda a això:

NAME	STATUS	AGE	LABELS
worker0	Ready	1d	hostname=worker0
worker1	Ready	1d	hostname=worker1
worker2	Ready	1d	hostname=worker2

2. Triem un dels Nodes i li afegim una etiqueta:

```
kubectl label nodes <worker-n> disktype=ssd
```

A on worker-n és un dels Nodes que triarem

3. Comprovem que el Node que hem triat té afegida l'etiqueta.

```
kubectl get nodes --show-labels
```

Amb una sortida pareguda a això:

NAME	STATUS	AGE	LABELS
worker0	Ready	1d	disktype=ssd ,hostname=worker0
worker1	Ready	1d	hostname=worker1
worker2	Ready	1d	hostname=worker2

4. Cream un Pod que sigui assignat al Node que hem triat. Aquest fitxer manifest descriu que té un 'selector de node', `disktype=ssd`. Això vol dir que el Pod es serà assignat al node que te l'etiqueta, `disktype=ssd` com veim al llistat del codi A.1 a la línia 13.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5    labels:
6      env: test
7  spec:
8    containers:
9      - name: nginx
10        image: nginx
11        imagePullPolicy: IfNotPresent
12    nodeSelector:
13      disktype: ssd
```

Llistat del Codi A.1: pod-nginx.yaml

5. Cream el Pod mitjançant el fitxer manifest de la passa anterior:

```
kubectl apply -f ... / pod-nginx.yaml
```

6. Verificam que el Pod s'està executant al node triat:

```
kubectl get pods --output=wide
```

Amb una sortida pareguda a això:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
nginx	1/1	Running	0	13s	10.200.0.4	worker0

Com hem pogut comprovar el nostre Pod s'assigna al Node a on l'etiqueta `disktype=ssd` (*worker0*) com havíem previst.

BIBLIOGRAFIA

- [1] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011. 1
- [2] G. Gilder, “The information factories,” *Wired*, October 2006. [Online]. Available: <https://www.wired.com/2006/10/cloudware/> 1
- [3] sun.com, “¿cómo empezó el cómputo cloud?” 2010. [Online]. Available: <https://web.archive.org/web/20100218121439/http://www.itnews.ec/news/000396.aspx> 1
- [4] Cotel, “Descifrando la computación perimetral,” 2021, aplicacions. [Online]. Available: <https://cotel.com.co/descifrando-la-computacion-perimetral/> 1
- [5] B. E. Whitaker, “Cloud edge computing: Beyond the data center,” OpenStack Foundation - Edge Computing Group, Tech. Rep., 2021. [Online]. Available: <https://www.openstack.org/use-cases/edge-computing/cloud-edge-computing-beyond-the-data-center/pdf> 1
- [6] W. contributors, “Fog computing,” Wikipedia, The Free Encyclopedia, Agost 2022. [Online]. Available: https://en.wikipedia.org/wiki/Fog_computing 1
- [7] —, “Edge computing,” Wikipedia, The Free Encyclopedia, Juliol 2022. [Online]. Available: https://en.wikipedia.org/wiki/Edge_computing 1
- [8] WinSystems, “Cloud, fog and edge computing – what’s the difference?” Desembre 2017, accés a la imatge i l’article. [Online]. Available: <https://www.winsystems.com/cloud-fog-and-edge-computing-whats-the-difference/> 1.1
- [9] M. Lukša, *Kubernetes in Action*, 1st ed. Manning Publications Co., Desembre 2017, ch. 1. [Online]. Available: <https://www.manning.com/books/kubernetes-in-action> 1.2
- [10] GeeksforGeeks-Community, “The story of netflix and microservices,” *Open Knowledge*, Maig 2020. [Online]. Available: <https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/> 1
- [11] W. contributors, “Virtual machine,” Wikipedia, The Free Encyclopedia, Juliol 2022. [Online]. Available: https://en.wikipedia.org/wiki/Virtual_machine 1, 2.1
- [12] C. Ltd., “Linux containers,” Setembre 2021. [Online]. Available: <https://linuxcontainers.org/lxc/introduction/> 1, 2.2

- [13] RedHat, “Diferencias entre los contenedores y las máquinas virtuales,” Gener 2020, accés a la imatge. [Online]. Available: https://www.redhat.com/cms/managed-files/styles/wysiwyg_full_width/s3/virtualization-vs-containers_transparent.png?itok=q-E2I2-L 2.1
- [14] W. contributors, “Os level virtualization — Wikipedia, the free encyclopedia,” Juliol 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=OS-level_virtualization 2.1
- [15] Google, “What are containers?” 2022, containers vs. VMs. [Online]. Available: <https://cloud.google.com/learn/what-are-containers?hl=en#section-4> 2.1
- [16] RedHat, “Traditional linux containers vs docker,” Gener 2018, accés a la imatge. [Online]. Available: https://www.redhat.com/cms/managed-files/traditional-linux-containers-vs-docker_0.png 2.2
- [17] W. contributors, “Init,” Wikipedia, The Free Encyclopedia, Abril 2022. [Online]. Available: <https://en.wikipedia.org/wiki/Init> 2.2, 4
- [18] RedHat, “What is ci/cd?” Red Hat, Tech. Rep., Maig 2022. [Online]. Available: <https://www.redhat.com/en/topics/devops/what-is-ci-cd> 2.2
- [19] Kubernetes, “Pods,” Gener 2022. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/> 2.3
- [20] M. Lukša, “Kubernetes in action, second edition - version: 13,” Juny 2022, accés a la imatge. [Online]. Available: <https://drek453711klr.cloudfront.net/luksa3/v-13/Figures/1.8.png> 2.3
- [21] W. contributors, “Runtime system,” Juny 2022. [Online]. Available: https://en.wikipedia.org/wiki/Runtime_system 2.3.1, 3
- [22] M. Lukša, “Kubernetes in action, second edition - version: 13,” Juny 2022, accés a la imatge. [Online]. Available: <https://drek453711klr.cloudfront.net/luksa3/v-13/Figures/1.9.png> 2.4
- [23] —, *Kubernetes in Action - Version 13*, 2nd ed. Manning Publications Co., Juny 2022, ch. 1.2. [Online]. Available: <https://www.manning.com/books/kubernetes-in-action-second-edition> 2.3.1
- [24] Kubernetes, “The components of a kubernetes cluster,” Abril 2022, accés a la imatge. [Online]. Available: <https://d33wubrfki0l68.cloudfront.net/2475489eaf20163ec0f54ddc1d92aa8d4c87c96b/e7c81/images/docs/components-of-kubernetes.svg> 2.5
- [25] Z. Arnold, S. Dua, W. Huang, F. Masood, M. Qin, and M. Taleb, *The Kubernetes Workshop: Learn how to build and run highly scalable workloads on Kubernetes*. Packt Publishing, 2020, ch. 17 - Advanced Scheduling in Kubernetes. [Online]. Available: <https://www.packtpub.com/product/the-kubernetes-workshop/9781838820756> 2.4, 3, 4, 5, 7, 3.2

-
- [26] —, “The kubernetes workshop: Learn how to build and run highly scalable workloads on kubernetes,” 2020, accés a la imatge. [Online]. Available: <https://www.packtpub.com/product/the-kubernetes-workshop/9781838820756> 2.6
- [27] Kubernetes, “Scheduling framework extensions,” Octubre 2021, accés a la imatge. [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/> 2.7, 3.1
- [28] —, “Extensions point,” Octubre 2021, punts d’extensió. [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/#extension-points> 2.4.1
- [29] —, “Scheduling plugins,” Octubre 2021, connectors de planificació. [Online]. Available: <https://kubernetes.io/docs/reference/scheduling/config/#scheduling-plugins> 2.4.1
- [30] —, “Multicluster special interest group,” 2022. [Online]. Available: <https://github.com/kubernetes/community/tree/master/sig-multicluster> 3
- [31] A. Youssef, “Why the kubernetes scheduler is not enough for your ai workloads,” *CNCF Member Blog Post*, Agost 2020. [Online]. Available: <https://www.cncf.io/blog/2020/08/10/why-the-kubernetes-scheduler-is-not-enough-for-your-ai-workloads/> 3.1
- [32] CNCF, “Kubeedge - edge controller,” 2022, overview. [Online]. Available: https://kubeedge.io/en/docs/architecture/cloud/edge_controller/ 3.1
- [33] A. S. Foundation, “Yunikorn - scheduler,” 2022, overview. [Online]. Available: <https://yunikorn.apache.org/docs/api/scheduler/> 3.1
- [34] CNCF, “volcano,” 2022, a Kubernetes Native Batch System. [Online]. Available: <https://volcano.sh/en/> 3.1
- [35] Kubernetes, “Scheduler source code,” Juliol 2022, scheduler.go. [Online]. Available: <https://github.com/kubernetes/kubernetes/tree/master/pkg/scheduler> 1
- [36] —, “Scheduling special interest group,” Juliol 2022. [Online]. Available: <https://github.com/kubernetes/community/tree/master/sig-scheduling> 3.1, 3.5.3
- [37] —, “Scheduler extender,” Gener 2019, design Proposals Archive. [Online]. Available: https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md 3.1
- [38] W. contributors, “Webhook,” Abril 2022, HTTP callback. [Online]. Available: <https://en.wikipedia.org/wiki/Webhook> 3.1
- [39] Kubernetes, “Scheduling policies,” Decembre 2021, remove kube-scheduler-policy contents. [Online]. Available: <https://kubernetes.io/docs/reference/scheduling/policies/> 3.1

- [40] W. Q. L. Fan) and Z. Kai, “The burgeoning kubernetes scheduling system – part 1: Scheduling framework,” *Alibaba Cloud Community Blog*, Febrer 2021. [Online]. Available: https://www.alibabacloud.com/blog/the-burgeoning-kubernetes-scheduling-system-part-1-scheduling-framework_597318 3.1
- [41] J. Beda, “Core kubernetes: Jazz improv over orchestration,” *Heptio Blog Archive & Medium*, Maig 2017, accés a la imatge i l’article. [Online]. Available: <https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca> 3.1
- [42] Kubernetes, “Assign pods to nodes. example,” Juny 2022, example. [Online]. Available: <https://kubernetes.io/docs/tasks/configure-pod-container/assign-pods-nodes/#add-a-label-to-a-node> 3.2.1
- [43] —, “Assign pods to nodes. nodename,” Juliol 2022. [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodename> 1
- [44] M. Lukša, *Kubernetes in Action - Version 13*, 2nd ed. Manning Publications Co., Juny 2022, ch. 10.3.2 Utilizing label selectors within Kubernetes API objects, p. 323, scheduling Pods to Nodes with Specific Labels. [Online]. Available: <https://www.manning.com/books/kubernetes-in-action-second-edition> 2
- [45] Kubernetes, “Well-known labels, annotations and taints,” Juny 2022, reference to Well-Known Labels. [Online]. Available: <https://kubernetes.io/docs/reference/labels-annotations-taints/> 4b, 3.4
- [46] A. C. G. Wei Huang (IBM), “Introducing podtopologyspread,” Kubernetes, Tech. Rep., Maig 2020. [Online]. Available: <https://kubernetes.io/blog/2020/05/introducing-podtopologyspread/> 6
- [47] Kubernetes, “Well-known node taints,” Juny 2022, taints sempre emprats als Nodes. [Online]. Available: <https://kubernetes.io/docs/reference/labels-annotations-taints/#node-kubernetes-io-not-ready> 3.4
- [48] M. Burrillo, “Kubecon + cloudnativecon europe 2018,” in *Building a Kubernetes Scheduler using Custom Metrics*. The Linux Foundation Events, Maig 2018. 3.4
- [49] CNCF, “Helm,” 2016, accés al lloc Web. [Online]. Available: <https://helm.sh/> 3.4
- [50] Kubernetes, “Pods,” Juny 2022, working with Pods. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/#working-with-pods> 1
- [51] —, “Generate static pod manifests for control plane components,” Juny 2022, configure the Kubernetes Scheduler. [Online]. Available: <https://kubernetes.io/docs/reference/setup-tools/kubeadm/implementation-details/#generate-static-pod-manifests-for-control-plane-components> 3
- [52] —, “La api de métricas,” Juliol 2022, pipeline de métricas de recursos. [Online]. Available: <https://kubernetes.io/es/docs/tasks/debug-application-cluster/resource-metrics-pipeline/#la-api-de-métricas> 3.5.1

- [53] C. N. C. Foundation, “Graduated and incubating projects,” 2022, projects Maturity Levels. [Online]. Available: <https://www.cncf.io/projects/> 3.5.1
- [54] Staff, “Announcing the intent to form the prometheus conformance program,” Cloud Native Computing Foundation, CNCF Blog, Maig 2021, cNCF during *PromCon* at *KubeCon*. [Online]. Available: <https://www.cncf.io/blog/2021/05/03/announcing-the-intent-to-form-the-prometheus-conformance-program/> 3.5.1
- [55] Prometheus, “Monitoring linux host metrics with the node exporter,” 2022, doc Guides. [Online]. Available: <https://prometheus.io/docs/guides/node-exporter/#monitoring-linux-host-metrics-with-the-node-exporter> 5
- [56] Google, “Golang,” 2012, homepage. [Online]. Available: <https://go.dev> 3.5.2
- [57] M. Hausenblas and S. Schimanski, *Programming Kubernetes: Developing Cloud-Native Applications*. O’Reilly Media, 2019, ch. 1 - Introduction. [Online]. Available: <https://programming-kubernetes.info/> 3.5.2, 5
- [58] Kubernetes, “Client libraries,” Juny 2022, kubernetes Clients. [Online]. Available: <https://kubernetes.io/docs/reference/using-api/client-libraries/> 3.5.2
- [59] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st ed. Addison-Wesley Professional, 2015, ch. 8 - Goroutines and Channels, pp. 217–256. [Online]. Available: <https://www.gopl.io/> 5
- [60] —, *The Go Programming Language*, 1st ed. Addison-Wesley Professional, 2015, ch. Preface, pp. xi – xiv. [Online]. Available: <https://www.gopl.io/> 3.5.2
- [61] M. Farina, “Kubernetes special interest groups,” *Kubernetes Blog*, Agost 2016. [Online]. Available: <https://kubernetes.io/blog/2016/08/sig-apps-running-apps-in-kubernetes/> 3.5.3
- [62] Kubernetes, “Kubernetes community,” Setembre 2020, governance. [Online]. Available: <https://github.com/kubernetes/community#governance> 3.5.3
- [63] CNCF, “Kubernetes conformance.” [Online]. Available: <https://landscape.cncf.io/card-mode?category=certified-kubernetes-distribution,certified-kubernetes-hosted,certified-kubernetes-installer> 3.6
- [64] —, “Kubernetes conformance,” enllaç **list**. [Online]. Available: <https://github.com/cncf/k8s-conformance/blob/master/README-WG.md#kubernetes-conformance> 3.6
- [65] —, “Certified kubernetes - platforms.” [Online]. Available: <https://landscape.cncf.io/guide#platform> 3.6
- [66] S. Jha, “Vanilla kubernetes vs managed kubernetes,” *Digital Ocean Blog*, Octubre 2021. [Online]. Available: <https://www.digitalocean.com/blog/vanilla-kubernetes-vs-managed-kubernetes> 3.6
- [67] Kubernetes, “Daemonset,” Abril 2021. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/> 1

- [68] K. Documentation, “Kubernetes api health endpoints,” Novembre 2021. [Online]. Available: <https://kubernetes.io/docs/reference/using-api/health-checks/> 2
- [69] C. Ltd., “Charmed kubernetes.” [Online]. Available: <https://ubuntu.com/kubernetes/charmed-k8s> 11
- [70] CNCF, “2021 annual report: Sig scheduling,” Febrer 2022. [Online]. Available: <https://www.cncf.io/reports/kubernetes-annual-report-2021/> 3.7
- [71] W. Huang, “Sig-scheduling intro and deep dive,” in *Make the most of your scheduler*. KubeCon 2021 - North America, Octubre 2021, p. 25. [Online]. Available: https://static.sched.com/hosted_files/kccncna2021/88/KubeCon_NA_2021-SIG-Scheduling-Deep-Dive.pdf 3.7
- [72] Kubernetes, “Kubernetes scheduler simulator,” Agost 2021. [Online]. Available: <https://github.com/kubernetes-sigs/kube-scheduler-simulator> 3.7, 3.3, 3.4, 3.5, 3.6
- [73] —, “Internal keps for kube-scheduler-simulator,” Maig 2022. [Online]. Available: <https://github.com/kubernetes-sigs/kube-scheduler-simulator/tree/master/keps> 3.7
- [74] —, “Kubernetes enhancement proposals (keps),” Febrer 2022, what the KEPs are. [Online]. Available: <https://github.com/kubernetes/enhancements/tree/master/keps#kubernetes-enhancement-proposals-keps> 4.1
- [75] —, “Kep 624 - scheduling framework),” Abril 2021. [Online]. Available: <https://github.com/kubernetes/enhancements/tree/master/keps/sig-scheduling/624-scheduling-framework#scheduling-framework> 4.1
- [76] A. A. Donovan and B. W. Kernighan, *The Go Programming Language*, 1st ed. Addison-Wesley Professional, 2015, ch. 7 - Interfaces, pp. 171–216. [Online]. Available: <https://www.gopl.io/> 4.1
- [77] Kubernetes, “Declaració dels punts d’extensió del marc de treball del planificador,” Març 2022, interface.go. [Online]. Available: <https://github.com/kubernetes/kubernetes/blob/4ce5a8954017644c5420bae81d72b09b735c21f0/pkg/scheduler/framework/interface.go#L295> 4.1
- [78] J. Renner, “Creating a kube-scheduler plugin,” *Medium*, Juliol 2021. [Online]. Available: <https://medium.com/@julio renner123/k8s-creating-a-kube-scheduler-plugin-8a826c486a1> 4.1
- [79] Kubernetes, “Repositori de *Scheduler-Plugins*. matriu de compatibilitat,” Agost 2022. [Online]. Available: <https://github.com/kubernetes-sigs/scheduler-plugins#compatibility-matrix> 1
- [80] S. Schimanski, “Kubernetes deep dive: Code generation for customresources,” *Red Hat. Hybrid cloud blog*, Octubre 2017. [Online]. Available: <https://cloud.redhat.com/blog/kubernetes-deep-dive-code-generation-customresources> 5

- [81] jorge, “Shallow copy y deep copy en c#,” *Open Knowledge - Arquitectura*, Maig 2020. [Online]. Available: <https://geeks.ms/jorge/2020/05/06/shallow-copy-y-deep-copy-en-c/> 5
- [82] M. Hausenblas and S. Schimanski, *Programming Kubernetes: Developing Cloud-Native Applications*. O’Reilly Media, 2019, ch. 3 - Basics of client-go. API Machinery in Depth. [Online]. Available: <https://programming-kubernetes.info/7>
- [83] Oracle, “Oracle vm virtualbox.” [Online]. Available: <https://virtualbox.org> 4.2.2
- [84] D. Simionato, “Kubernetes 1.25 – what’s new?” *Sysdig Blog*, Agost 2022. [Online]. Available: <https://sysdig.com/blog/kubernetes-1-25-whats-new/> 5
- [85] K. Tsakalozos and A. Kolaitis, “Slack - #microk8s discussion,” Maig 2022. [Online]. Available: https://app.slack.com/client/T09NY5SBT/threads?selected_team_id=T09NY5SBT 5
- [86] K. SIG, “Multicluster special interest group,” Setembre 2019. [Online]. Available: <https://github.com/kubernetes/community/tree/master/sig-multicluster> 5
- [87] Kubernetes, “Creating a kube-scheduler plugin on microk8s #379,” Setembre 2022. [Online]. Available: <https://github.com/kubernetes-sigs/scheduler-plugins/issues/379#issuecomment-1227798395> 6.1