ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ
ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δρομολόγηση βάσει κόστους πόρων και χρονικής απόκρισης σε σύστημα Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Τσιακατάρας

Επιβλέπων : Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.

Αθήνα, 22/03/2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ
ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Δρομολόγηση βάσει κόστους πόρων και χρονικής απόκρισης σε σύστημα Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γεώργιος Τσιακατάρας

Επιβλέπων : Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22/03/2024.

.....	.....	.....
Τσανάκας Παναγιώτης	Σούντρης Δημήτριος	Ξύδης Σωτήριος, Επίκουρος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής

Αθήνα, 22/03/2024

.....
Γεώργιος Τσιακατάρας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στην σημερινή εποχή, η πλατφόρμα Kubernetes είναι ευρέως διαδεδομένη για τη διαχείριση εφαρμογών κυρίως σε cloud συστήματα. Με την ζήτηση αυτή παρουσιάζεται και παράλληλη ανάπτυξή σε εργαλεία που επεκτείνουν τις βασικές λειτουργίες της. Ένα από τα σημαντικότερα προβλήματα που σχετίζονται με ην πλατφόρμα είναι η βέλτιστη δρομολόγηση των εφαρμογών προς εκτέλεση. Γύρω από το πρόβλημα αυτό, αναπτύσσονται συνεχώς καινούργιες τεχνικές για την βελτιστοποίηση του ως προς τις ανάγκες του χρήστη, της εφαρμογής και της υποδομής. Έτσι, σκοπός της εργασίας αυτής είναι η ανάλυση της διαδικασίας scheduling μέσω ορισμένων αλγορίθμων με διαφορετικές ιδιότητες, αξιοποιώντας παράλληλα διαθέσιμα εργαλεία του Kubernetes οικοσυστήματος. Αρχικά θα αναλυθούν ορισμένα στοιχεία του Kubernetes και ένας αριθμός επεκτάσεων του που θα είναι ζωτικής σημασίας για την ανάπτυξη των αλγόριθμων scheduling. Επίσης αναλύεται λεπτομερώς η διαδικασία του scheduling και των στοιχείων που το συντάσσουν καθώς επίσης αναφέρονται και τεχνικές που μπορούν να χρησιμοποιηθούν κατά την εκτέλεση του για παραγωγή καλύτερων αποτελεσμάτων. Οι αλγόριθμοι scheduling που θα αναλυθούν θεωρητικά αλλά και θα συγκριθούν μέσω πειράματος είναι τρεις. Ο πρώτος αποτελεί τον αλγόριθμο NetMARKS που εστιάζει στην βελτίωση του χρόνου απόκρισης χρησιμοποιώντας τις δυνατότητες του εργαλείου Istio αλλά δεν λαμβάνει υπόψιν του άλλους πόρους του συστήματος δημιουργώντας έτσι πιθανά προβλήματα σε κατανάλωση ενέργειας αλλά και σε απόδοση. Ο δεύτερος αλγόριθμος είναι ο Bin Balancer που εστιάζει στην βελτίωση της ισορροπίας κόστους, μια μετρική που θα αναλυθεί περισσότερο εντός του έργου, κάνοντας χρήση του εργαλείου OpenCost αδιαφορώντας ωστόσο για τον χρόνο απόκρισης. Τέλος περιγράφεται ο Combined αλγόριθμος που αποτελεί συνδυασμό του NetMARKS και του BinBalancer. Οι αλγόριθμοι μεταξύ του παρουσιάζουν αρκετές διαφορές οι οποίες θα περιγράφουν θεωρητικά και θα επιβεβαιωθούν πειραματικά. Τα αποτελέσματα τους θα συγκριθούν και με τον default scheduler του Kubernetes, ο οποίος θα αποτελέσει τη σταθερά σύγκρισης για την καταγραφή της επίδοσής τους. Τα συμπεράσματα που θα προκύψουν θα δώσουν μια καθαρή εικόνα για την επίδραση του scheduler στο σύστημα και πως μπορεί να προσαρμοστεί ανάλογα με τις εκάστοτε ανάγκες.

Λέξεις Κλειδιά:

Συστήματα Kubernetes, αλγόριθμος δρομολόγησης, μετρικές σύγκρισης, χρόνος απόκρισης, κόστος πόρων

Abstract

In today's world, Kubernetes systems are widely used for managing applications in cloud systems. With this demand comes parallel development with tools that extend their functionality. However, the problem of scheduling in Kubernetes is of particular importance and new techniques are constantly being developed to optimize it with respect to the needs of the user, the application and the infrastructure. Thus, the purpose of this thesis is to analyze the scheduling process through some algorithms with different properties, while utilizing several tools of the Kubernetes ecosystem. First, some elements of Kubernetes and some of its extensions vital for the development of scheduling algorithms will be analyzed. It also analyses in detail the scheduling process and the elements that compose it, as well as the techniques that can be used in its execution to produce better results. The scheduling algorithms that will be analyzed theoretically and compared through experimentation are three. the first algorithm analyzes the NetMARKS algorithm that focuses on improving the response time of the application using Istio metrics but disregards other system recourses, leading to possible issues regarding energy consumption and system performance. The second algorithm is Bin Balancer that focuses on improving cost balance, a metric that will be further analyzed within the thesis, using metrics provided by OpenCost but does not take into account the response time of the application. The final algorithm is the combined algorithm that will be a combination of NetMARKS and Bin Balancer. Those algorithms present some differences that will be analyzed theoretically and then confirmed with experiments. Their results will also be compared with the default Kubernetes scheduler so that there is a better insight into its influence on the system. The resulting conclusions will give a clear picture of the scheduler's impact on the system and how it can be adjusted as needed.

Keywords:

Kubernetes, scheduling algorithms, performance metrics, response time, resource cost

Ευχαριστίες

Θα ήθελα να ευχαριστήσω αρχικά τον επιβλέποντα καθηγητή μου, κ. Παναγιώτη Τσανάκα για την ανάθεση της διπλωματικής αυτής. Επίσης θα ήθελα να ευχαριστήσω τον συμμετέχοντα στην επίβλεψη, κ. Σταμάτη Κατσαούνη για την βοήθεια που μου πρόσφερε κατά την διάρκεια της εργασίας.

Στην συνέχεια θα ήθελα να ευχαριστήσω την οικογένεια μου που ήτανε πάντα στο πλάι μου κατά την διάρκεια των σπουδών μου.

Αφιερώνω αυτή την διπλωματική εργασία στην γιαγιά μου, την δεύτερη μητέρα μου, που απεβίωσε πρόσφατα.

Πίνακας Περιεχομένων

Λίστα Εικόνων

Λίστα Πινάκων

Εισαγωγή

Το Kubernetes αποτελεί ένα ισχυρό εργαλείο για τη διαχείριση εφαρμογών με βάση τα containers ή αλλιώς απομονωμένων περιοχών χρήστη, στις οποίες εκτελείται λογισμικό. Εξαιτίας της αυτοματοποίησης που παρέχει, ιδιαίτερα στην κλιμάκωση εφαρμογών, παρουσιάζει εξαιρετική δημοφιλία σε περιβάλλοντα υπολογιστικού νέφους (cloud) και διαθέτει μία ισχυρή και πολυπληθή κοινότητα [1]. Το γεγονός αυτό, σε συνδυασμό με τη διάθεσή του στο κοινό μέσω άδειας ανοιχτού λογισμικού, έχει βοηθήσει αρκετά στη διαρκή ανάπτυξή του αλλά και επέκταση με νέες δυνατότητες. Η ανάγκη για όλο και μεγαλύτερη κλιμάκωση αλλά και αυτοματοποίηση, προσφέρει ένα μεγάλο πεδίο έρευνας αλλά και ανοιχτών ζητημάτων γύρω από το Kubernetes. Ένα από τα σημαντικότερα είναι η δρομολόγηση των εφαρμογών προς εκτέλεση στους καταλληλότερους κόμβους-εργάτες βάσει κάποιου κριτηρίου βελτιστοποίησης. Το εν λόγω πρόβλημα, παρότι είναι πολυπαραγοντικό και επιλύεται σε μη πολυωνυμικό χρόνο, μπορεί να περιοριστεί σε συγκεκριμένα κριτήρια, ανάλογα τη φύση της εφαρμογής και να μελετηθεί αρκετά ικανοποιητικά. Παραδείγματος χάρη, για εφαρμογές που απαιτούν γρήγορη απόκριση σε διαδικτυακά αιτήματα, η δρομολόγηση των κόμβων μπορεί να γίνει λαμβάνοντας ως κύριο κριτήριο τον χρόνο απόκρισής τους και η επίλυση του να προσφέρει τον καλύτερο δυνατό χρόνο απόκρισης για την εφαρμογή.

Ο στόχος της διπλωματικής εργασίας αυτής, είναι να παρουσιάσει τρεις αλγόριθμους με διαφορετικές ιδιότητες, να αναλυθούν θεωρητικά και μέσω πειραμάτων σε συγκεκριμένη εφαρμογή με κρίσιμο throughput και να επιλεχθεί ο καλύτερος αλγόριθμος. Ο πρώτος αλγόριθμος που παρουσιάζεται προσφέρει καλύτερο throughput ωστόσο το κόστος του δημιουργεί προβληματισμούς σε μια εποχή που είναι επιθυμητή μια καλύτερη ενεργειακά λύση [2]. Ο δεύτερος αλγόριθμος επιλύει το πρόβλημα του ενεργειακού κόστους παρόλα αυτά όμως έχει κακό αντίκτυπο στο throughput. Ο τρίτος αλγόριθμος αποτελεί συνδυασμό των δύο πρώτων αλγόριθμων με σκοπό την άντληση των θετικών και από τους δύο πρώτους αλγόριθμους και την εξισορρόπηση των αποτελεσμάτων τους, προσφέροντας με τον τρόπο αυτό μια μέση λύση. Για καλύτερη κατανόηση μάλιστα των τελικών αποτελεσμάτων, οι αλγόριθμοι συγκρίνονται στο τέλος ως προς τα αποτελέσματα με τον default scheduler του Kubernetes. Μάλιστα κατά το έργο θα αναλυθούν και αρκετές λειτουργίες και επεκτάσεις του Kubernetes που θα συμβάλουν στην εκτέλεση των αλγορίθμων scheduling.

Η διπλωματική εργασία είναι οργανωμένη σε επτά (7) κεφάλαια στα οποία γίνεται τόσο θεωρητική ανάλυση των schedulers και αναφορά σε λειτουργίες του Kubernetes, όσο και παρουσιάζεται το πειραματικό κομμάτι με τα αποτελέσματα του έργου.

Στο κεφάλαιο δύο (2) γίνεται εισαγωγή στην αρχιτεκτονική Microservices και στο σύστημα του Kubernetes. Μάλιστα όσο αναφορά το Kubernetes αναλύονται ορισμένα συστατικά του χρήσιμα για αυτό το έργο και που θα συμβάλουν στην καλύτερη κατανόηση των αλγορίθμων και της πειραματικής διαδικασίας.

Στο κεφάλαιο τρία (3) γίνεται ανάλυση των λειτουργειών και των επεκτάσεων του Kubernetes που χρησιμοποιήθηκαν για δημιουργία των αλγορίθμων scheduler και των δοκιμών.

Στο κεφάλαιο τέσσερα (4) γίνεται λεπτομερής περιγραφή της διαδικασίας scheduling και των βημάτων που απαιτούνται για την ολοκλήρωσή της. Επίσης

περιγράφονται τεχνικές και επιλογές μετρικών που μπορούν να χρησιμοποιηθούν για την ανάπτυξη τέτοιων αλγορίθμων.

Στο κεφάλαιο πέντε (5) περιγράφονται οι αλγόριθμοι scheduling που θα αναπτυχθούν και αναλύονται ορισμένες ιδιότητές τους. Επίσης στο τέλος συγκρίνονται σε θεωρητικό επίπεδο οι ιδιότητές τους.

Στο κεφάλαιο έξι (6) γίνεται ανάλυση του συστήματος και της εφαρμογής που θα χρησιμοποιηθούν για τις δοκιμές των schedulers που αναλύθηκαν στο προηγούμενο κεφάλαιο. Επίσης γίνεται και περιγραφή της εγκατάστασης των απαραίτητων επεκτάσεων του Kubernetes για την αναπαραγωγή των δοκιμών.

Στο κεφάλαιο επτά (7) γίνεται ανάλυση της διαδικασίας δοκιμών που ακολούθησαν όλοι οι αλγόριθμοι. Επίσης παρουσιάζονται τα αποτελέσματα των δοκιμών αυτών και γίνεται σύγκριση μεταξύ των διάφορων αλγορίθμων. Τέλος, εξάγονται και παρουσιάζονται συμπεράσματα σχετικά με τα αποτελέσματα αυτά.

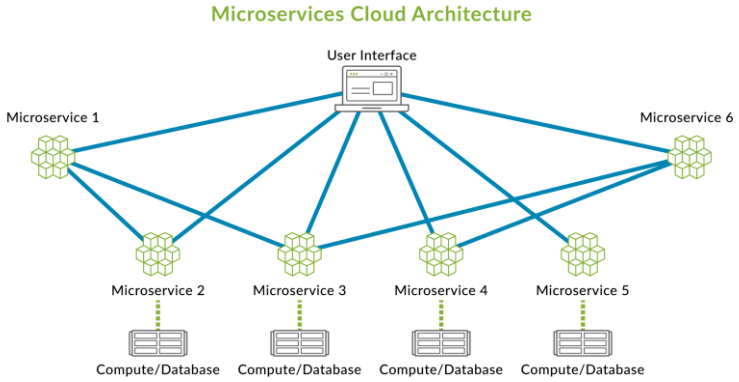
Κεφάλαιο 2: Microservices και Kubernetes

2.1 Αρχιτεκτονική Microservices

Ως microservices (ή αλλιώς microservice architecture) μπορεί να οριστεί η αρχιτεκτονική μιας εφαρμογής η οποία αποτελείται από services (μικρές εφαρμογές) με τα εξής χαρακτηριστικά [3] [4]:

- Το κάθε service αποτελεί μια μικρή εκτελέσιμη μονάδα και είναι ανεξάρτητο από τα υπόλοιπα services, με την έννοια ότι για να παράξει αποτελέσματα δεν απαιτείται η χρήση κάποιου άλλου service της εφαρμογής (Independently deployable).
- Η διαθεσιμότητα ενός service δεν επηρεάζει σημαντικά τη διαθεσιμότητα των υπολοίπων (loosely coupled).
- Το κάθε service μπορεί να δημιουργηθεί χρησιμοποιώντας διαφορετική γλώσσα προγραμματισμού, βάση δεδομένων, περιβάλλον software και hardware. Η επιλογή εξαρτάται από πολλούς παράγοντες και δεν επηρεάζει τα υπόλοιπα services.
- Η επικοινωνία μεταξύ των services επιτυγχάνεται συνήθως μέσω δικτύου.

Η αρχιτεκτονική αυτή δίνει τη δυνατότητα σε έναν οργανισμό να προσφέρει μεγάλες και πολύπλοκες εφαρμογές με ταχύ ρυθμό, καθώς και με μεγάλη αξιοπιστία. Η αρχιτεκτονική αυτή υλοποιείται συνήθως από cloud-native applications (δηλαδή εφαρμογές οι οποίες σχεδιάζονται και εκτελούνται σε κατανεμημένα συστήματα cloud και αξιοποιούν τις λειτουργίες που προσφέρουν τα συστήματα αυτά [5]) και applications που χρησιμοποιούν VMs ή χαμηλά σε κατανάλωση πόρων containers.



Εικόνα 1. Παράδειγμα Microservice αρχιτεκτονικής

2.1.1 Εικονικές μηχανές (VM)

Μια εικονική μηχανή αποτελεί μια υπολογιστική οντότητα η οποία χρησιμοποιεί στοιχεία του λογισμικού αντί ενός φυσικού υπολογιστή, για την εκτέλεση προγραμμάτων. Η εικονική μηχανή εκτελείται με δικό της λειτουργικό σύστημα ανεξάρτητα από τον υπολογιστή που εκτελείται (host) και επομένως συμπεριφέρεται ως ανεξάρτητος υπολογιστής. Μάλιστα, περισσότερα από ένα εικονικά μηχανήματα

μπορούν να εκτελούνται σε έναν host και να είναι ανεξάρτητα μεταξύ τους. Η τεχνολογία των εικονικών μηχανημάτων χρησιμοποιείται σε πολλά περιβάλλοντα τόσο τοπικά όσο και cloud, και η ιδιότητα των εικονικών μηχανών για ανεξαρτησία μεταξύ τους, τα κρίνουν κατάλληλα για την χρήση τους σε αρχιτεκτονικές microservices. [6] [7]

2.2 Kubernetes

2.2.1 Εισαγωγή στο Kubernetes και Containers

Το Kubernetes [1] είναι μια επεκτάσιμη πλατφόρμα ανοιχτού κώδικα για τη διαχείριση φόρτου εργασίας και services με containers, που διευκολύνει τον αυτοματισμό για τον σχεδιασμό και την υποστήριξη εφαρμογών. Το οικοσύστημα που διαθέτει είναι μεγάλο και αναπτύσσεται συνεχώς στα σύγχρονα δεδομένα. Οι υπηρεσίες, η υποστήριξη και τα εργαλεία Kubernetes είναι ευρέως διαθέσιμα.
Τα container που διαχειρίζεται το Kubernetes είναι παρόμοια με τα VM, αλλά έχουν χαλαρές ιδιότητες απομόνωσης (relaxed isolation properties) για κοινή χρήση του λειτουργικού συστήματος (OS) μεταξύ των εφαρμογών και συνεπώς μπορούν να θεωρηθούν ως ελαφριές εφαρμογές. Παρόμοια με ένα VM, ένα container έχει το δικό του σύστημα αρχείων, μερίδιο CPU, μνήμη, χώρο διεργασιών και πολλά άλλα. Καθώς είναι αποσυνδεδεμένα από την υποκείμενη υποδομή, είναι φορητά σε διανομές λειτουργικού συστήματος και στο cloud. Επίσης χρησιμοποιούν τα container images που αποτελούν ένα ανεξάρτητο εκτελέσιμο πακέτο κώδικα που εμπεριέχει ότι χρειάζεται για την εκτέλεση μια εφαρμογής, όπως κώδικα, βιβλιοθήκες , εργαλεία, ρυθμίσεις κλπ. Τα container images κατά την εκτέλεση τους μετατρέπονται σε containers, αποκλείοντας το λογισμικό που εκτελούν από το υπόλοιπο περιβάλλον. Τα container images επίσης είναι ελαφριά πακέτα και διευκολύνεται έτσι η χρήση τους καθώς και διανομή τους σε άλλους χρήστες [8]. Η διασημότητα των containers οφείλεται στις εξής τους δυνατότητες [1] [9]:

- **Ευέλικτη δημιουργία και ανάπτυξη**: Τα container images δημιουργούνται με μεγάλη ευκολία σε σχέση με τη χρήση εικόνας VM.
- **Συνεχής ανάπτυξη και ενσωμάτωση νέων λειτουργειών** Τα containers μπορούν να ‘αναβαθμίζονται’ συνεχώς και αξιόπιστα ενώ επίσης έχουν αποδοτική και άμεση επιστροφή σε προηγούμενη τους έκδοση (rollback).
- **Ανεξαρτησία εφαρμογής από υποδομή**: Τα container images μπορούν να δημιουργηθούν κατά τον χρόνο έκδοσης (release time) αντί στον χρόνο ανάπτυξης (deployment time).
- **Περισσότερες μετρικές**: Τα container μπορούν να εμφανίσουν περισσότερες μετρικές σχετικά με την κατάσταση εφαρμογής και όχι μόνο πληροφορίες επιπέδου λογισμικού.
- **Σταθερότητα περιβάλλοντος**: Τα containers θα εκτελεστούν το ίδιο σε κάθε σύστημα.
- **Αρχιτεκτονική microservices**: Τα containers ‘ταιριάζουν’ σε περιγραφή με τα services ενός microservices συστήματος και μπορούν να αποτελέσουν θεμελιώδες κομμάτι της ανάπτυξής του.
- **Απομόνωση και αποδοτική χρήση πόρων**.

Τα κύρια χαρακτηριστικά του Kubernetes που το ξεχωρίζουν από υπόλοιπες υποδομές που το ανταγωνίζονται (όπως το Docket Swarm, Apache Mesos, Fleet κλπ. [9]) αποτελούν:

- Εύρεση services στο δίκτυο και εξισορρόπησή τους βάσει το βάρος, το οποίο ορίζεται από πλήθος μετρικών που επιλέγει ο σχεδιαστής έτσι ώστε να επιτύχει επιθυμητές ιδιότητες της εφαρμογής, όπως για παράδειγμα την γρήγορη απόκριση της εφαρμογής σε διαδικτυακά αιτήματα.
- Ενορχήστρωση του αποθηκευτικού χώρου από διαφορετικά συστήματα και παρόχους αποθηκευτικού χώρου (storage providers) που ορίζονται εντός του Kubernetes. Επίσης μια εφαρμογή Kubernetes μπορεί να χρησιμοποιεί πολλούς Storage providers για τα διάφορα services που περιέχει.
- Αυτοματοποιημένες αλλαγές κατάστασης στις επιθυμητές για τα container του Kubernetes (rollbacks και rollouts)
- Αυτόματη κατανομή των container στα διάφορα nodes του Kubernetes βασισμένο στις ανάγκες τους σε RAM και CPU
- Αυτόματη επαναφορά από σφάλματα containers μέσω επανεκκίνησης ή αντικατάστασης τους κλπ. (self-healing)
- Εύκολή διαχείριση ρυθμίσεων για τα containers
- Εκτέλεση πολλαπλών φορτίων εργασίας ταυτόχρονα
- Οριζόντια κλιμάκωση του συστήματος
- Σχεδιασμένο για επεκτασιμότητα χωρίς την ανάγκη αλλαγής του υπάρχοντος κώδικα

2.2.2 Τα συστατικά του Kubernetes

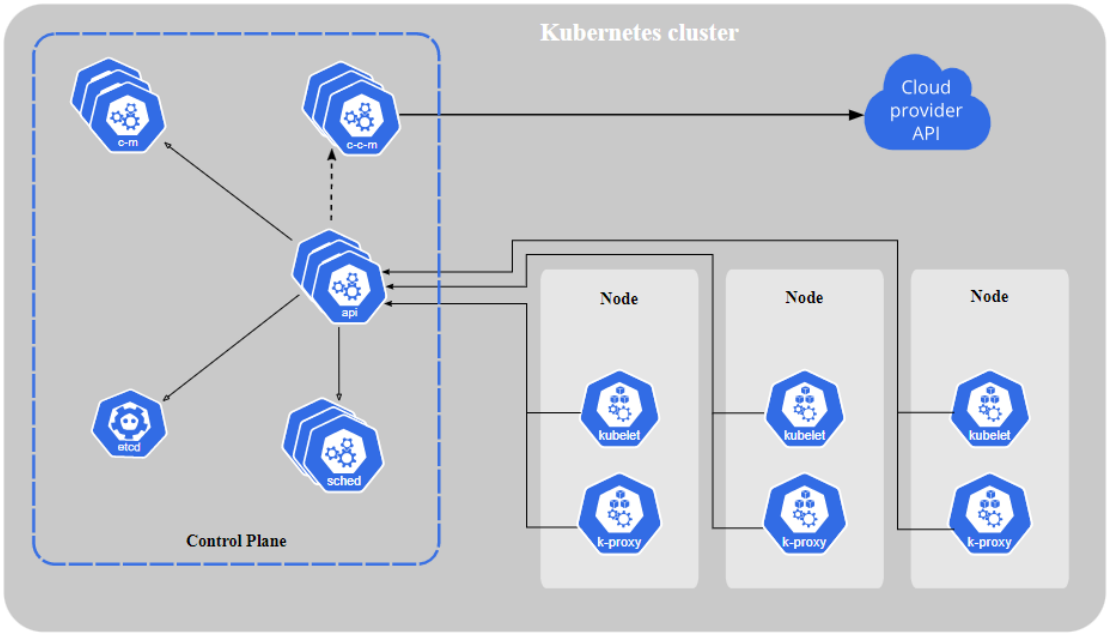
Αρχικά το σύνολο των οντοτήτων του Kubernetes αποτελεί μία συστοιχία: το cluster. Το κάθε cluster αποτελείται από έναν αριθμό μηχανημάτων-εργατών (worker machines) τα οποία καλούνται κόμβοι (nodes) και εκτελούν τα containers. Κάθε cluster περιέχει τουλάχιστον ένα node. Τα nodes φιλοξενούν τα pods που είναι τα συστατικά της εφαρμογής και περιέχουν μέσα τους έναν αριθμό containers (περισσότερες λεπτομέρειες έπειτα). Η διαχείριση των pods και των nodes γίνεται από το control plane που συνήθως εκτελείται πάνω σε πολλά nodes του cluster με σκοπό την αντοχή στα σφάλματα και την υψηλή διαθεσιμότητα. Τα συστατικά του control plane είναι υπεύθυνα για την λήψη αποφάσεων και εντοπισμό γεγονότων σε όλο το cluster. Τα συστατικά αυτά συνοπτικά είναι τα εξής [10]:

- Kube-apiserver: κάνει ορατό το εσωτερικό API του Kubernetes control plane.
- etcd: συνεχώς διαθέσιμος αποθηκευτικός χώρος για τα δεδομένα του Kubernetes και των καταστάσεων (state) των συστατικών του.
- Kube-scheduler: παρατηρεί τα καινούργια pods που δεν έχουν ανατεθεί σε κάποιο node ακόμα και επιλέγει σε ποιο node θα εκτελεστούν. Ο kube-scheduler θα αναλυθεί λεπτομερώς και σε επόμενο κεφάλαιο.
- Kube-controller-manager: Εκτελεί διεργασίες σχετικά με την παρατήρηση της κατάστασης στο Kubernetes και λαμβάνει αποφάσεις έτσι ώστε να οδηγηθεί το cluster στην επιθυμητή κατάσταση.

- Cloud-controller-manager: Παρόμοιο με το kube-controller-manager άλλα ελέγχει καταστάσεις που είναι συγκεκριμένες στον πάροχο cloud που πιθανόν φιλοξενείται το cluster.

Επιπλέον, το κάθε node περιέχει συστατικά τα οποία συμβάλλουν στην ομαλή εκτέλεση και διαχείριση των Pods. Τα συστατικά αυτά είναι συνοπτικά:

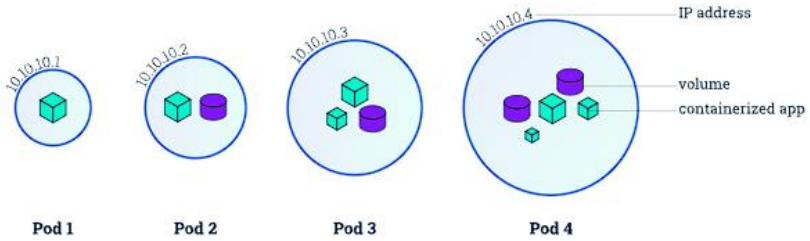
- Kubelet: Εκτελείται σε κάθε node και ελέγχει εάν τα containers ενός pod εκτελούνται
- Kube-proxy: Υλοποιεί την λογική των services του Kubernetes και εφαρμόζει τους κανόνες δικτυακής επικοινωνίας στο node.
- Container Runtime: Υπεύθυνο για τη διαχείριση του κύκλου ζωής των container



Εικόνα 2. Παράδειγμα δομής ενός Kubernetes Cluster [10]

2.2.3 Pods

Τα pods αποτελούν την μικρότερη υπολογιστική μονάδα η οποία μπορεί να δημιουργηθεί και να διαχειριστεί από το Kubernetes. Ουσιαστικά αποτελεί σύνολο ενός η περισσότερων containers με κοινό αποθηκευτικό χώρο και δικτυακούς πόρους ενώ επίσης περιέχει συγκεκριμένες προδιαγραφές για την εκτέλεση των containers. Συνήθως στα Pods που εμπεριέχουν πάνω από ένα container, τα container αυτά είναι στενά συνδεδεμένα στις λειτουργίες τους και πρέπει να συνεργαστούν μεταξύ τους και να μοιραστούν πόρους.



Εικόνα 3. Παράδειγμα Pods και των συστατικών τους

2.2.4 Deployments

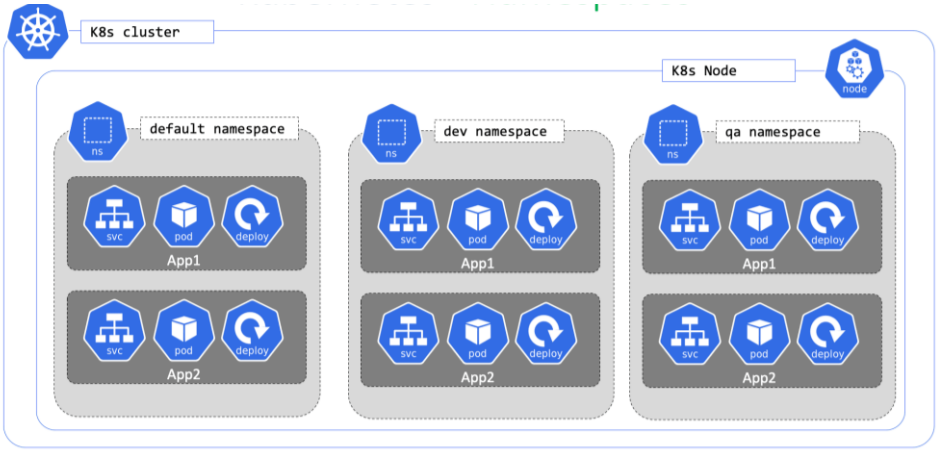
Συνήθως η δημιουργία των Pods δεν γίνεται αυτόματα αλλά μέσω άλλων πόρων των Kubernetes. Ένας τέτοιος πόρος είναι το deployment το οποίο παρέχει ενημερώσεις στα pods για την κατάστασή τους. Συγκεκριμένα, στα deployment δίνεται μια επιθυμητή κατάσταση που θέλει να επιτύχει το pod και ο ελεγκτής του deployment το αλλάζει στην κατάσταση αυτή με ελεγχόμενο ρυθμό.

2.2.5 Service

Το service στο Kubernetes αποτελεί πόρο του Kubernetes ο οποίος εκθέτει μια δικτυακή εφαρμογή η οποία αποτελείται από ένα η περισσότερα Pods. Με τον τρόπο αυτό δεν απαιτούνται αλλαγές στον κώδικα. Επιπλέον με τη χρήση του Service API μπορούν τα Pods να εκτεθούν στο δίκτυο και θέτει ορισμένα endpoints έτσι ώστε τα pods αυτά να είναι προσβάσιμα. Έτσι, παρόλο που τα Pods αλλάζουν κατάσταση συνεχώς, το service τα κάνει προσβάσιμα πάντα από το δίκτυο εντός του cluster με σταθερή IP διεύθυνση [11].

2.2.6 Namespaces

Τα namespaces στο Kubernetes προσφέρουν ένα μηχανισμό απομόνωσης ομάδων πόρων από άλλους πόρους σε ένα cluster Kubernetes. Τα ονόματα των πόρων πρέπει να είναι μοναδικά εντός του namespace αλλά όχι σε σχέση με τα άλλα namespaces. Επίσης, στα namespaces μπορούν να προστεθούν πόροι οι οποίοι είναι συμβατοί με τα namespaces όπως τα deployments και τα services, και όχι πόροι που είναι κοινοί για όλο το cluster όπως τα Nodes. Επιπλέον, τα namespaces μπορούν να χρησιμοποιηθούν για να μοιράσουν τους πόρους στους διάφορους χρήστες τους cluster η να ελέγξουν την πρόσβαση των χρηστών σε ορισμένους πόρους. Τέλος, στα namespaces μπορούν να προστεθούν ετικέτες (labels) με δικές τους τιμές που τα ξεχωρίζουν από τα υπόλοιπα namespaces.



Εικόνα 4. Παράδειγμα πολλών namespaces σε ένα Node

2.2.7 Manifests

Ένα αρχείο Kubernetes Manifest είναι ένα αρχείο YAML ή JSON στο οποίο ορίζονται προδιαγραφές όπως ιδιότητες, μετα-δεδομένα και επιθυμητή κατάσταση για ορισμένα αντικείμενα του Kubernetes, όπως τα Deployments, τα Services, τα Replica Sets κλπ. Μέσω αυτών μπορούμε να ορίσουμε ιδιότητες όπως ποιο image θα χρησιμοποιείται για τα Pods, των αριθμό των replicas και τα όρια των resources που μπορούν να χρησιμοποιηθούν. Στα μετα-δεδομένα μπορούμε να ορίσουμε labels στο κάθε αντικείμενο καθώς και σχόλια. Τα αρχεία αυτά και η επεξεργασία τους κατέχουν σημαντικό ρόλο στο έργο καθώς γίνεται εφικτό μέσω αυτών να δημιουργηθεί κατάλληλο περιβάλλον για τις δοκιμές [12].

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-client-app-deploy
spec:
  replicas: 3
  selector:
    matchLabels:
      app: todo-client-app
  template:
    metadata:
      labels:
        app: todo-client-app
    spec:
      containers:
      - image: todo-client-kubernetes-app:latest
        name: container1
        imagePullPolicy: Always
```

Εικόνα 5. Παράδειγμα ενός Manifest αρχείου

2.2.8 Limits and Requests

Κατά τον ορισμό ενός pod σε ένα cluster, πρέπει να οριστούν επίσης και οι πόροι που θα χρειαστούν τα containers που θα χρησιμοποιήσει, συνήθως μέσω των YAML αρχείων που τα ορίζουν. Οι πόροι αυτοί συνήθως είναι η RAM και η CPU που

θα καταναλώσει το container. Οι ορισμοί των πόρων αυτών γίνονται στο πεδίο request και limit, το κάθε ένα πεδίο εκτελεί διαφορετική λειτουργία με τις τιμές που του δίνονται. Όταν ορίζονται τιμές στο πεδίο requests, οι τιμές αυτές χρησιμοποιούνται από τον kube-scheduler για να αποφασίσει σε ποιο node θα τοποθετήσει το pod. Επίσης το σύστημα kubelet δεσμεύει πόρους τουλάχιστον ίσους με τις τιμές που ορίζονται στο request για την αντίστοιχη τιμή που έχει ορισθεί, έτσι ώστε να χρησιμοποιηθεί από το container του pod. Αξίζει να σημειωθεί ότι παρά τις τιμές του request, το container μπορεί να χρησιμοποιήσει παραπάνω πόρους από αυτούς που του έχουν αποδοθεί. Όταν ορίζονται τιμές στο πεδίο limit για ένα container, τότε το kubelet επιβάλλει ότι οι τιμές αυτές δεν επιτρέπεται να ξεπεραστούν από το container κατά την εκτέλεσή του. Το σύστημα, ανάλογα με τις ρυθμίσεις που του έχουν δοθεί, είτε όταν ξεπεραστούν οι τιμές θα τερματίσει το pod, είτε δεν θα επιτρέψει ποτέ στο pod να ξεπεράσει το όριο αυτό. Ένα τμήμα YAML στο οποίο έχουν οριστεί τιμές request και limit είναι το εξής:

```
apiVersion: v1
kind: Pod
metadata:
 name: frontend
spec:
 containers:
 - name: app
  image: images.my-company.example/app:v4
  resources:
   requests:
    memory: "64Mi"
    cpu: "250m"
   limits:
    memory: "128Mi"
    cpu: "500m"
 - name: log-aggregator
  image: images.my-company.example/log-aggregator:v6
  resources:
   requests:
    memory: "64Mi"
    cpu: "250m"
   limits:
    memory: "128Mi"
    cpu: "500m"
```

Όσο αναφορά τις μονάδες μέτρησης που χρησιμοποιούνται για να περιγράψουν τη μνήμη και τη CPU που χρησιμοποιείται, το Kubernetes εισάγει δικές του μονάδες. Συγκεκριμένα για τη CPU το Kubernetes χρησιμοποιεί ως μονάδα τη χρήση ενός φυσικού ή εικονικού επεξεργαστή εξ ολοκλήρου. Η τιμή ωστόσο μπορεί να πάρει δεκαδικές τιμές που δηλώνουν το ποσοστό του επεξεργαστή. Έτσι στο παραπάνω παράδειγμα το 250m αναπαριστά το 0.25 ποσοστό ενός επεξεργαστή. Για την μνήμη η μονάδα μέτρησης είναι το ένα Byte. Εκτός όμως από του κλασσικούς συμβολισμούς για τα πολλαπλάσιά του μπορούν να χρησιμοποιηθούν και αντίστοιχοι για τις δυνάμεις του δύο. Έτσι τα επιθέματα E, P, T, G, M, k αντιστοιχούν στα αντίστοιχα σε δυνάμεις του δύο: Ei, Pi, Ti, Gi, Mi, Ki

2.2.9 Taints and Tolerations

Τα taints αποτελούν ιδιότητες των nodes που τους δίνει τη δυνατότητα να μην επιτρέπουν να γίνει δρομολόγηση σε αυτά ένα σύνολο από pods. Τα tolerations αποτελούν ιδιότητες των pods και επιτρέπουν να γίνει η δρομολόγησή τους σε ένα node

με taint, εάν το taint και το toleration ταιριάζουν. Ο συνδυασμός των δύο ιδιοτήτων αυτών επιτρέπει τη δρομολόγηση των pods σε nodes που επιθυμεί ο προγραμματιστής. Επίσης σε ένα node μπορεί να εφαρμοστεί παραπάνω από ένα taint και αντίστοιχα στα pods παραπάνω από ένα toleration. Το παρακάτω παράδειγμα περιγράφει πως εφαρμόζονται τα taints και tolerations και τις ιδιότητες τους.

Τα taints εφαρμόζονται σε ένα node μέσω της εξής εντολής kubectl (η οποία εντολή δίνει πρόσβαση σε ένα command line interface το οποίο επιτρέπει πρόσβαση στο Kubernetes API, όπως αναφέρεται και στην σχετική ενότητα 2.4) [13]:

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

Η συγκεκριμένη εντολή θέτει ένα taint στο node1 με κλειδί key1 και τιμή value1. Επίσης ορίζεται η δράση του taint που είναι NoSchedule και μπορεί επίσης να λάβει τις τιμές NoExecute και PreferNoSchedule.Συγκεκριμένα, με την τιμή NoSchedule, καινούργια Pods που δεν έχουν toleration δεν θα δρομολογηθούν στο node που έχει τεθεί το taint. Ωστόσο pods τα οποία ήδη εκτελούνται στο node δεν θα πάψουν να εκτελούνται σε αυτό. Αντιθέτως, με την τιμή NoExecution τα pods που εκτελούνται ήδη στο node και δεν έχουν toleration, με την εφαρμογή του taint, αποκλείονται από το Node και άμεσα, εκτός εάν τους έχει τεθεί τιμή στο πεδίο tolerationSeconds στο οποίο βάσει την τιμή, έπειτα από το χρονικό διάστημα αυτό αποκλείονται. Τέλος, η τιμή PreferNoSchedule, αποτελεί μια χαλαρή εφαρμογή της τιμής NoSchedule καθώς για τα pods που δεν έχουν toleration για το taint του node , θα προσπαθήσει το σύστημα να μην τα δρομολογήσει στο node αυτό αλλά δεν υπάρχει εγγύηση ότι θα συμβεί. Ο ορισμός ενός toleration σε κάποιο pod γίνεται στο PodSpec του και λαμβάνει την εξής μορφή:

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
```

Το συγκεκριμένο toleration έχει ίδιο value με το key που ορίστηκε στο taint του node και συνεπώς ικανοποιεί τις συνθήκες που ορίστηκαν σαυτό. Επομένως, το pod αυτό θα μπορεί να δρομολογηθεί στο node1. Εκτός από τον operator Equal υπάρχει και ο Exists που απλά εξετάζει την ύπαρξη του κλειδιού ανεξαρτήτως τιμής.

2.2.10 Persistent Volume (PV) και Persistent Volume Claim (PVC)

Αρχικά, η διαχείριση του αποθηκευτικού χώρου αποτελεί ένα ξεχωριστό πρόβλημα για την διαχείριση υπολογιστικών εφαρμογών. Για την επίλυση αυτού του προβλήματος, το Kubernetes παρέχει ένα API που κρύβει λεπτομέρειες σχετικά με την παροχή χώρου και κατανάλωσής του από τους χρήστες, κάνοντας το πιο εύκολο στην κατανόηση και στην εφαρμογή του. Για τον λόγο αυτό, το Kubernetes προσφέρει δύο πολύ σημαντικά συστατικά του, το persistent volume και persistent volume claim.

Ένα persistent volume (PV) αποτελεί ένα κομμάτι αποθηκευτικού χώρου στο cluster το οποίο παρέχεται στατικά από ένα διαχειριστή του συστήματος η δυναμικά μέσω ενός άλλους συστατικού του Kubernetes, το storage class. Ο χρόνος ζωής των PVs είναι ανεξάρτητός από την ζωή των pods που τον χρησιμοποιούν και οι

λεπτομέρειες υλοποίησης του αποθηκευτικού του χώρου τις διαχειρίζεται το API του Kubernetes.

Ένα persistent volume claim (PVC) αποτελεί ουσιαστικά ένα αίτημα του χρήστη για αποθηκευτικό χώρο. Λειτουργεί παρόμοια με ένα pod αλλά καταναλώνει πόρους PV. Έτσι ένα PVC μπορεί να κάνει αίτημα στο cluster για συγκεκριμένο αποθηκευτικό χώρο και λειτουργίες πρόσβασής (όπως ReadWriteOnce, ReadOnlyMany, ReadWriteMany κλπ) σε ένα PV. Έτσι επιτρέπουν στους χρήστες την κατανάλωση πόρων αποθηκευτικού χώρου χωρίς να δηλώνουν περαιτέρω λεπτομέρειες. Μάλιστα ένα PVC δένεται (binds) με ένα PV όταν αυτό ικανοποιεί τις απαιτήσεις του και έπειτα ένα pod θα μπορεί να χρησιμοποιήσει ένα PVC ως αποθηκευτικό χώρο. Μάλιστα όταν γίνει το binding, ο χρήστης μπορεί να χρησιμοποιεί το PV για όσο χρόνο χρειάζεται.

2.2.11 Sidecar Containers

Ως sidecar container ορίζονται δευτερεύοντα containers που εκτελούνται παράλληλα με το κύριο container ενός Pod. Με τον τρόπο αυτόν επεκτείνουν η ενισχύουν τις λειτουργίες του κύριου container, χωρίς να επηρεάζουν τον κώδικα εκτέλεσης του. Οι λειτουργίες αυτές αποτελούν συνήθως την παρακολούθηση πόρων, την καταγραφή γεγονότων στην εφαρμογή ή την ασφάλεια της εφαρμογής του pod. [14]

2.3 Kubernetes API

Όπως αναφέρθηκε και στο κεφάλαιο 2.2.2 ο πυρήνας του Kubernetes Control Plane βρίσκεται στον API server. Ο API server κάνει διαθέσιμο στους χρήστες ένα HTTP API στο οποίο μπορούν να γίνουν ερωτήματα σχετικά με αντικείμενα του Kubernetes όπως nodes και namespaces και να γίνει και επεξεργασία τους. Οι περισσότερες λειτουργίες μπορούν να εκτελεστούν είτε μέσω του kubectl είτε μέσω REST κλήσεων που πραγματοποιούν άμεση πρόσβαση στο API. Για τη χρήση του REST API υπάρχει πληθώρα βιβλιοθηκών με υποστήριξη για πολλές γλώσσες προγραμματισμού. Μάλιστα το Kubernetes API είναι σχεδιασμένο έτσι ώστε να παρέχει μακροχρόνια προσβασιμότητα σε όλους τους χρήστες και προσφέρει άπλετο χρόνο στις εφαρμογές να προσαρμοστούν σε τυχόν νέες αλλαγές. Η πρόσβαση στο API μπορεί να επιτευχθεί μέσω των endpoints /openapi/v2 και /openapi/v3 [15].

Η παρούσα διπλωματική εργασία κάνει χρήση της βιβλιοθήκης Kubernetes σε Python (μπορεί να εγκατασταθεί μέσω pip, ένα εργαλείο για την εγκατάσταση πακέτων σε περιβάλλον python [16] η οποία προσφέρει κλήσεις στο API του Kubernetes, οργανωμένες σε ειδικά object και συναρτήσεις. Έτσι διευκολύνεται η χρήση του API από τον σχεδιαστή και γίνεται η διαδικασία του debugging πιο προσιτή. Επίσης οι συναρτήσεις και τα object που παρέχει μπορούν να εκτελέσουν πολλές λειτουργίες και συνεπώς προσφέρουν μεγάλη ελευθερία στη διαχείριση του cluster.

Ένα παράδειγμα κώδικα που κάνει χρήση της βιβλιοθήκης είναι το παρακάτω. Σε αυτό καταγράφονται όλα τα pod του cluster με το αντίστοιχο namespace και την IP τους.

```
from kubernetes import client, config

config.load_kube_config()

v1 = client.CoreV1Api()
```

```
print("Listing pods with their IPs:")
ret = v1.list_pod_for_all_namespaces(watch=False)
for i in ret.items:
  print("%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace,
  i.metadata.name))
```

Παρατηρείται ότι η πρόσβαση στον API server δεν γίνεται μέσω άμεσων κλήσεις αλλά μέσω συναρτήσεων. Εν προκειμένω, μέσω της συνάρτησης list_pod_for_all_namespaces(). Επίσης παρατηρείται ότι το αποτέλεσμα της κλήσης στο API αποθηκεύεται σε ένα σειριοποιημένο αντικείμενο και όχι σε μορφή JSON. Έτσι γίνεται πιο εύκολη η χρήση των αποτελεσμάτων που παράγονται και μπορεί να γίνει πιο αποτελεσματικός έλεγχος για τυχόν προβλήματα στη δημιουργία του κώδικα.

2.4 Kubectl

Το kubectl αποτελεί ένα command line interface (CLI) το οποίο χρησιμοποιείται για την επικοινωνία με το control plane ενός Kubernetes cluster, μέσω του Kubernetes API [17]. Το kubectl παράγει χρήσιμες πληροφορίες σχετικά με τα αντικείμενα του cluster, την υγεία τόσο του cluster όσο και των αντικειμένων του, καθώς επίσης επιτρέπει αλλαγές στις ιδιότητες των αντικειμένων που περιέχει.

```
ishan301@G3-3500:~/Entangle/yaml$ kubectl describe deployment nginx
Name:                   nginx
Namespace:              default
CreationTimestamp:      Tue, 15 Nov 2022 23:56:33 +0530
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavai
lable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
```

Εικόνα 6. Παράδειγμα αποτελέσματος εντολής kubectl

Το kubectl μπορεί να παραμετροποιηθεί μέσω του αρχείου kubeconfig. Το kubeconfig περιέχει μεταξύ άλλων στοιχεία πρόσβασης στο cluster, το κύριο namespace και πολλά άλλα. Επίσης το kubectl υποστηρίζει και την επεκτασιμότητα μέσω Plugins που μπορούν να αναπτυχθούν από τον εκάστοτε χρήστη.

Κεφάλαιο 3: Υποδομές και επεκτάσεις του Kubernetes

3.1 Service Mesh και Istio

3.1.1 Service Mesh

Το service mesh αποτελεί ένα ειδικό επίπεδο υποδομής το οποίο μπορεί να προστεθεί σε εφαρμογές με αρχιτεκτονική microservices. Το service mesh προσφέρει στις εφαρμογές αυτές επιπλέον δυνατότητες παρατηρησιμότητας και παραγωγής μετρικών, διαχείριση της κυκλοφορίας επικοινωνίας μεταξύ των microservices και ασφάλεια χωρίς την ανάγκη προσθήκης κώδικα. Με τον όρο service mesh περιγράφεται τόσο ο τύπος του λογισμικού που χρησιμοποιείται για την υλοποίηση του καθώς και το domain ασφάλειας και δικτύου που δημιουργείται κατά την εφαρμογή του [18].

Με τη χρήση του service mesh, απλοποιείται η διαχείριση πολύπλοκων εφαρμογών με αρχιτεκτονική Microservices, όπως τα Kubernetes-based συστήματα, καθώς επίσης μπορούν να υποστηριχθούν πολύπλοκες διαδικαστικές απαιτήσεις (operational requirements).

3.1.2 Istio

Το Istio αποτελεί ένα service mesh ανοιχτού κώδικα το οποίο εφαρμόζεται με διαφανή τρόπο σε υπάρχουσες κατανεμημένες εφαρμογές. Τα πληθώρα χαρακτηριστικά του Istio παρέχουν ένα κατανεμημένο και αποτελεσματικό τρόπο για την σύνδεση, παρακολούθηση και προστασία από κακόβουλες επιθέσεις των services της εφαρμογής [19]. Το Istio παρέχει κατανομή φορτίου, πιστοποίηση ανάμεσα στα services και παρακολούθηση με ελάχιστες ή μηδενικές αλλαγές στον κώδικα των services. Επιπλέον, προσφέρει τα εξής χαρακτηριστικά:

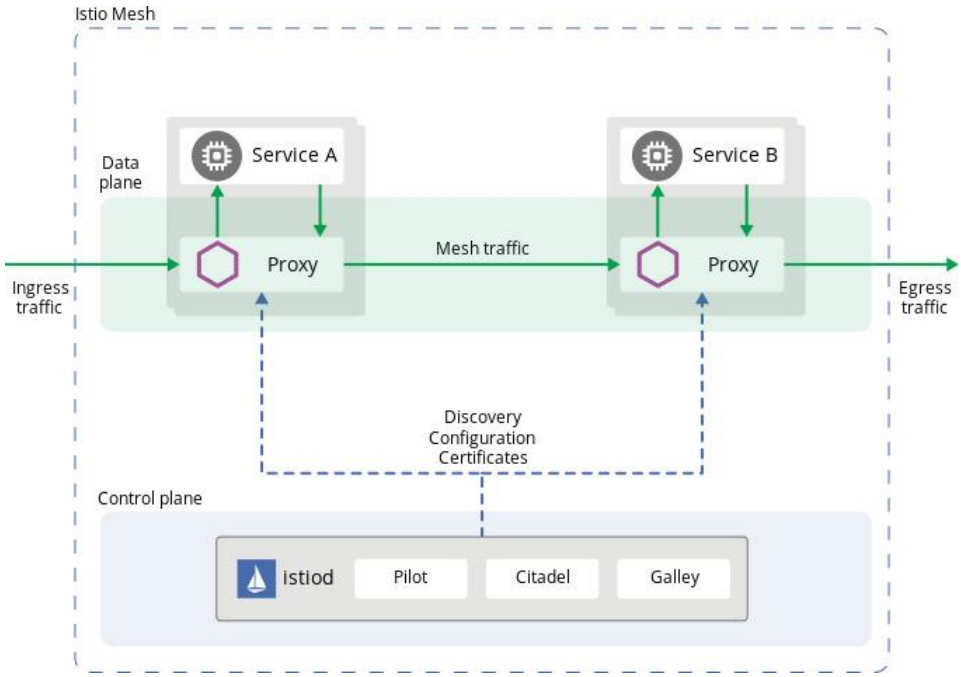
- Ασφαλή service-to-service επικοινωνία σε ένα cluster με κρυπτογράφηση mutual TLS (mTLS)
- Αυτόματη κατανομή φορτίου για επικοινωνία μέσω των πρωτοκόλλων HTTP, gRCP, WebSocket και TCP
- Υψηλός έλεγχος της δικτυακής κίνησης μέσω κανόνων δρομολόγησης, επαναλήψεων και άλλων
- Configuration API για την υποστήριξη ελεγκτών πρόσβασης στους πόρους της εφαρμογής (access controls) και μεθόδων ελέγχου της κίνησης μέσω περιορισμού των αιτήσεων που μπορούν να γίνονται για ένα χρονικό διάστημα (rate limits)
- Αυτόματες μετρικές, καταγραφές και ιχνηλασιμότητα για όλη την κίνηση εντός του cluster

Το Istio είναι σχεδιασμένο ως προς την επεκτασιμότητα και μπορεί να επεκτείνει με ευκολία και άμεσα μια εφαρμογή Kubernetes.

3.1.3 Η Λειτουργία του Istio

Το Istio αποτελείται από 2 βασικά συστατικά στοιχεία [19]:

- Data Plane: Αποτελείται από ένα σύνολο διαμεσολαβητών (Envoy proxies) που γίνονται deploy με πλευρικό τρόπο έτσι ώστε να προσθέτουν η να επεκτείνουν ήδη υπάρχουσες λειτουργικότητες των services (sidecar proxy). Οι διαμεσολαβητές ελέγχουν όλη την επικοινωνία στο δίκτυο μεταξύ των microservices και επιπλέον συλλέγουν και αναφέρουν μετρικές για όλη την κίνηση στο service mesh.
- Control Plane: Το control plane διαχειρίζεται και διαμορφώνει όλους τους μεσάζοντες (proxies) για τη δρομολόγηση της κυκλοφορίας στην εφαρμογή.



Εικόνα 7. Διαγραμματικά οι λειτουργίες του Data και Control Plane

3.1.4 Istiod

Το istiod αποτελεί την κύρια εφαρμογή του Istio στο control plane και παρέχει λειτουργίες όπως την εύρεση services και τη διαχείριση τους καθώς επίσης και τη διαχείριση πιστοποιητικών για τη δικτυακή τους επικοινωνία. Συγκεκριμένα παρέχει τα εξής [20]:

- Μετατρέπει κανόνες δρομολόγησης κίνησης σε ρυθμίσεις συγκεκριμένες για τα Envoy proxies και τις εφαρμόζει σε αυτά κατά την εκτέλεση της εφαρμογής.
- Εφαρμόζει μηχανισμούς εύρεσης services τους οποίους κάθε proxy που διαθέτει το Envoy API μπορεί να χρησιμοποιήσει και υποστηρίζει πολλά περιβάλλοντα.
- Με το Traffic Management API μπορεί να ελέγξει σημαντικά την κίνηση στο Service Mesh.

- Παρέχει σημαντική ασφάλεια για την κρυπτογράφηση της κίνησης εντός του cluster και υλοποιεί πολιτικές προστασίας που βασίζονται στην ταυτότητα του service οι οποίες είναι πιο αξιόπιστες από άλλα πρωτόκολλα επιπέδου 3 και 4.
- Το istiod λειτουργεί ως αρχή πιστοποίησης (CA) και μπορεί να παράγει πιστοποιητικά για την ασφαλή επικοινωνία εντός του cluster.

3.1.5 Istioctl

Το istioctl αποτελεί ένα CLI εργαλείο το οποίο προσφέρει δυνατότητες debug και διάγνωσης προβλημάτων σε ένα Istio mesh. Το istioctl μπορεί να εγκαταστήσει το Istio Mesh με πολλές δυνατότητες προσαρμογής και έτοιμα profile με ορισμένες ρυθμίσεις. Επιπλέον επιτρέπει την ανάκτηση πληροφοριών σχετικά με τις ρυθμίσεις του διακομιστή μεσολάβησης (proxy) για τα διάφορα συστατικά του Kubernetes [21]. Επίσης μέσω της εντολής istioctl analyze επιτρέπει την ανάλυση του cluster για την εύρεση τυχών προβλημάτων με τις ρυθμίσεις του Istio Mesh και παρέχει συμβουλές σχετικά με την αντιμετώπισή τους. Μάλιστα μπορεί να εκτελεστεί και σε αρχεία ρυθμίσεων του Istio τοπικά, εντοπίζοντας έτσι τυχόν προβλήματα πριν παρουσιαστούν στο cluster.

Η εγκατάσταση μέσω istioctl μπορεί να δημιουργήσει ένα Istio Mesh με βάσει κάποιο από τα υπάρχοντα προφίλ που διαθέτει προεγκατεστημένα ή μέσω τοπικών αρχείων ρυθμίσεων. Η εγκατάσταση αυτή παράγει custom recourses (CR) τα οποία αποτελούν επεκτάσεις ήδη υπάρχων αντικειμένων του Kubernetes. Για την διαχείριση τους χρειάζεται η ύπαρξη ενός operator ο οποίος εκτελείται σε κάποιο pod. Έτσι η εγκατάσταση παράγει ένα pod που ονομάζεται Istio Operator για την διαχείριση των custom recourses του Istio και την περιγραφή της επιθυμητής κατάστασής τους

Μετά την εγκατάσταση δημιουργούνται τα κατάλληλα συστατικά του Kubernetes στο namespace istio-system και συνήθως αυτά αποτελούν το deployment και service istiod και istio-ingressgateway. Η σύνθεση του namespace διαφέρει από εγκατάσταση σε εγκατάσταση ανάλογα τις ρυθμίσεις που δίνονται.

3.2 Prometheus

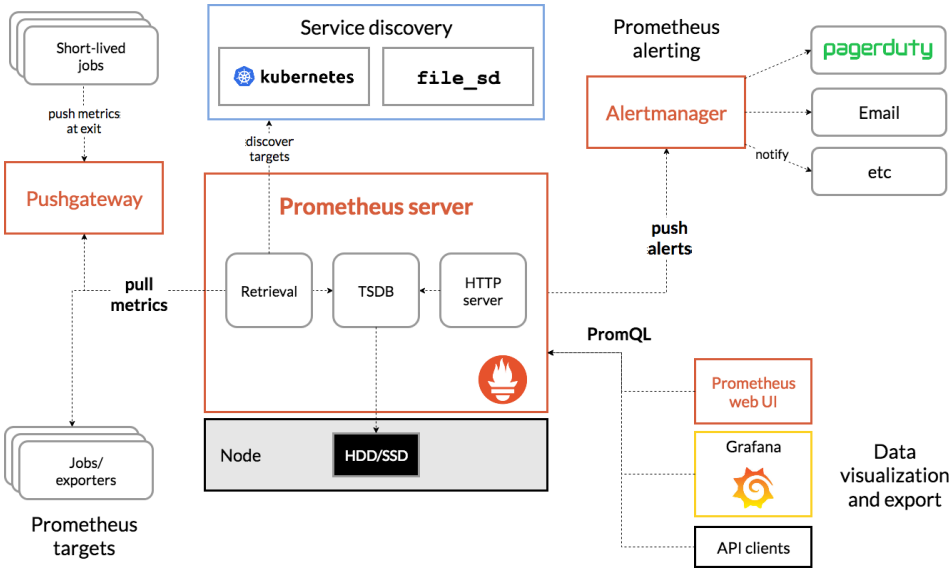
3.2.1 Εισαγωγή στο Prometheus

Το Prometheus αποτελεί μια εργαλειοθήκη ανοιχτού κώδικα για την παρακολούθηση πόρων συστήματος καθώς και την ειδοποίηση συστημάτων η οποία αρχικά ήταν κατασκευασμένη από την SoundCloud. Πλέον αποτελεί ένα αυτόνομο project και συντηρείται ανεξάρτητα από οποιαδήποτε εταιρεία. [22]

Το Prometheus συλλέγει και αποθηκεύει μετρικές ως δεδομένα χρονικού στιγμιότυπου (time series data). Συγκεκριμένα, κάθε πληροφορία μετρικής αποθηκεύεται μαζί με τη χρονική στιγμή στην οποία παρατηρήθηκε και εμπεριέχει προαιρετικά ζεύγη κλειδιών – τιμής τα οποία ονομάζονται labels. Έτσι το Prometheus αποτελεί ένα πολυδιάστατο μοντέλο δεδομένων με time series δεδομένα. Για την άντληση των δεδομένων αυτών υλοποιεί επίσης μια ευέλικτη query γλώσσα, την PromQL. Τέλος αξίζει να σημειωθεί ότι το Prometheus είναι ανεξάρτητο από τον διαμοιρασμένο χώρο αποθήκευσης της εφαρμογής και περιέχει πολλαπλούς τρόπους αναπαράστασης των δεδομένων.

Οι μετρικές που ορίζει το Prometheus αποτελούν αριθμητικές μετρήσεις σχετικές με το σύστημα ενώ ο χρήστης καθορίζει ποιες θα αξιοποιήσει. Οι μετρικές έχουν ζωτικό ρόλο στην κατανόηση της εφαρμογής και στον τρόπο που λειτουργεί, συμβάλλοντας στην καλύτερη ανάπτυξή της.

Η αρχιτεκτονική του Prometheus αποτελείται από πολλαπλά τμήματα. Αρχικά υπάρχει ο κύριος server του Prometheus ο οποίος είναι υπεύθυνος για την άντληση και την αποθήκευση των time-series δεδομένων. Έπειτα υπάρχουν client βιβλιοθήκες οι οποίες ενορχηστρώνουν τον κώδικα της εφαρμογής. Οι βιβλιοθήκες αυτές στέλνουν την τρέχουσα κατάσταση όλων των μετρικών που ελέγχουν στον Prometheus server [23]. Υπάρχει επίσης το push gateway, το οποίο αποτελεί ουσιαστικά μια cache ,δηλαδή έναν μικρό άλλα με γρήγορη απόκριση αποθηκευτικό χώρο με σκοπό την γρήγορη παροχή δεδομένων, για μετρικές. Έτσι μπορεί να λαμβάνει και μετρικές από μικρές σε ζωή εφαρμογές η services. [24]. Επιπλέον διαθέτει ειδικούς εξαγωγείς μετρικών (metrics exporter) για την υποστήριξη διάφορων services (π.χ. HAProxy, StatsD, κλπ.). Σημαντικό τμήμα του Prometheus αποτελεί και ο Alertmanager ο οποίος χειρίζεται τις προειδοποιήσεις που στέλνονται από τα client applications και τα δρομολογεί κατάλληλα σε email η άλλους μηχανισμούς. Τέλος περιέχει ποικιλία εργαλείων υποστήριξης για τις διάφορες λειτουργίες του.



Εικόνα 8. Η αρχιτεκτονική του Prometheus με τα διάφορα τμήματα του

Ένα σημαντικό κομμάτι για την διπλωματική εργασία αυτή, αποτελεί ο Node exporter του Prometheus. Συγκεκριμένα ο node exporter εξάγει μετρικές σχετικές με το hardware και τον πυρήνα του κάθε node που χρησιμοποιούμε στο Kubernetes cluster μας.

Επίσης σημαντικό για την ορθή λειτουργία των εφαρμογών του Kubernetes που θα αξιοποιήσουν το Prometheus είναι ο ορισμός των scrape configs. Συγκεκριμένα τα scrape configs είναι ένας τομέας στις ρυθμίσεις του Prometheus που δηλώνει ένα σύνολο παραμέτρων και endpoint στόχων. Από αυτούς τους στόχους το Prometheus θα αντλήσει δεδομένα και μετρικές και ο τρόπος με τον οποίο θα τα αντλήσει ορίζεται από τις παραμέτρους που δίνονται. Με τον τρόπο αυτό εφαρμογές του Kubernetes μπορούν να προωθήσουν στο Prometheus τα κατάλληλα δεδομένα έτσι ώστε να γίνει η εξαγωγή των μετρικών που ενδιαφέρουν τον διαχειριστή τους. [25]

```
scrape_configs:
- job_name: 'istiod'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istiod;http-monitoring
- job_name: kubecost
  honor_labels: true
  scrape_interval: 1m
  scrape_timeout: 10s
  metrics_path: /metrics
  scheme: http
  dns_sd_configs:
  - names:
    - kubecost-cost-analyzer.kubecost
    type: 'A'
    port: 9003
```

Εικόνα 9. Παράδειγμα scrape configs σε ρυθμίσεις του Prometheus

3.2.2 Prometheus API

Το Prometheus διαθέτει ένα HTTP API προσβάσιμο από το endpoint /api/v1. Οι απαντήσεις που επιστρέφει το API είναι της μορφής JSON, με τις επιτυχημένες να επιστρέφουν κωδικό κατάστασης 2xx. Σε περίπτωση άκυρου αιτήματος επιστρέφεται JSON error object με τον αντίστοιχο κωδικό κατάστασης και περιγραφή του σφάλματος στο πεδίο error. Τα δεδομένα που συλλέχθηκαν με επιτυχία βρίσκονται στο πεδίο data σε μορφή που ορίζεται από το πεδίο resultType ενώ το αποτέλεσμα στην αντίστοιχη μορφή εντοπίζεται στο πεδίο result.

Για το έργο αυτό, τα αιτήματα στο Prometheus API έγιναν μέσω της βιβλιοθήκης Python Prometheus-api-client. Η βιβλιοθήκη αυτή αποτελεί έναν wrapper για το Prometheus HTTP API και επίσης προσφέρει εργαλεία για την επεξεργασία των μετρικών που παράγει. Έτσι μπορεί να γίνει με ευκολία η σύνδεση στον server του Prometheus και να ληφθούν οι ζητούμενες μετρικές. Στις μετρικές αυτές μπορούν να εφαρμοστούν και αθροιστικές πράξεις οι οποίες ορίζονται από την βιβλιοθήκη. Ένα παράδειγμα σύνδεσης και εξαγωγής όλων των μετρικών που παράγει το Prometheus είναι το εξής:

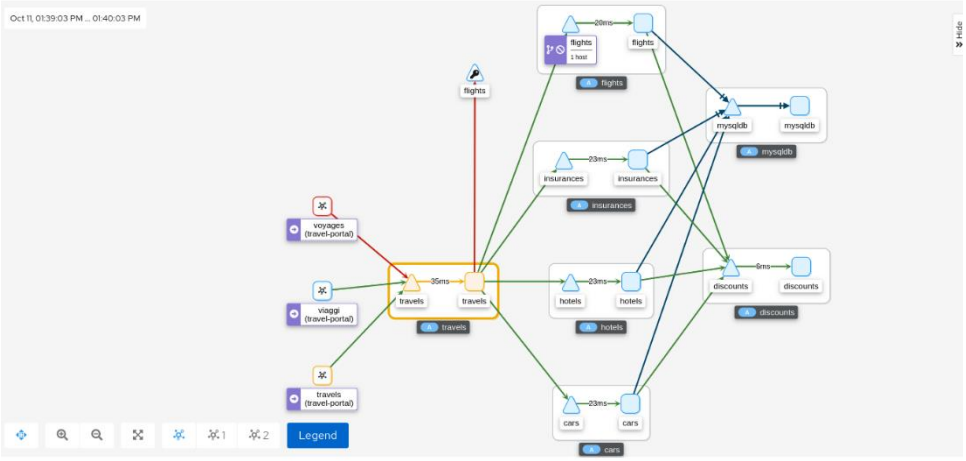
```python
from prometheus_api_client import PrometheusConnect
prom = PrometheusConnect(url ="<prometheus-host>", disable_ssl=True)

# Get the list of all the metrics that the Prometheus host scrapes
prom.all_metrics()
```

3.3 Kiali

Το Kiali αποτελεί μια κονσόλα του Istio που εστιάζει στην επίβλεψη της εφαρμογής. Συγκεκριμένα με χρήση της κονσόλας αυτής μπορεί να παρατηρηθεί η δομή και η υγεία του service mesh μέσω της παρακολούθησης της κίνησής του. Έτσι μπορεί να γίνει αντιληπτή η τοπολογία της εφαρμογής και να αναφερθούν τυχόν σφάλματα. Επίσης το Kiali παρέχει λεπτομερείς μετρικές για το service mesh και υποστηρίζει έμφυτα και άλλες εφαρμογές του Istio (Grafana, Jaeger, etc.) [26]

Το Kiali παρέχει επίσης λειτουργίες σχετικές με τη δρομολόγηση αιτημάτων δικτύου (request routing), τους ρυθμούς των αιτημάτων αυτών, την καθυστέρηση (latency) των services και πολλά άλλα. Επίσης προσφέρει χρήσιμες πληροφορίες για τα στοιχεία του service mesh για τα διαφορετικά επίπεδα της εφαρμογής.

Η πιο χρήσιμη λειτουργία του Kiali στη διαδικασία πειραμάτων της εργασίας αυτής, είναι το γράφημα τοπολογίας (Kiali Graph). Συγκεκριμένα, το γράφημα αυτό οπτικοποιεί την κίνηση του service σε πραγματικό χρόνο βασισμένο στο configuration του Istio, προσφέροντας άμεση παρατήρηση τυχόν συμβάντων. Οι κόμβοι του γραφήματός αποτελούν τα microservices στα οποία υπάρχει η κίνηση και περιέχουν πληθώρα πληροφοριών σχετικά με τις ρυθμίσεις του, ενώ οι ακμές του γραφήματος είναι η κίνηση που παρατηρείται ανάμεσα στα διάφορα microservices με το χρώμα τους να δηλώνει την υγεία της κίνησης.



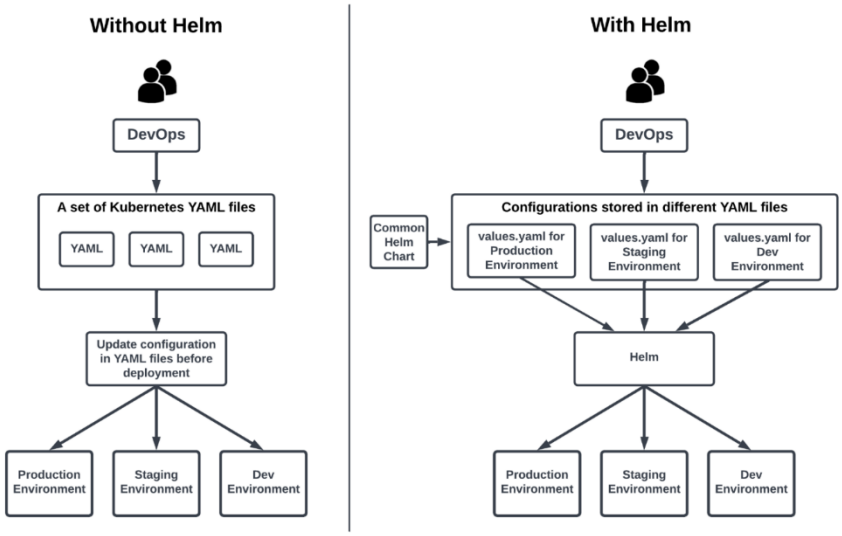
Εικόνα 10. Παράδειγμα Kiali Graph

3.4 Helm

Το helm αποτελεί ένα εργαλείο διαχείρισης πακέτων για το Kubernetes το οποίο αυτοματοποιεί την δημιουργία, πακετάρισμα και εγκατάσταση εφαρμογών σε αυτό [27]. Η εφαρμογή αποτελεί ένα πακέτο το οποίο περιέχει και όλες τις απαραίτητες ρυθμίσεις για την εγκατάσταση του. Έτσι γίνεται εύκολη η δημιουργία και η διαχείριση πολλών εφαρμογών του Kubernetes ταυτόχρονα καθώς κάθε εφαρμογή αποτελεί διαθέτει δικιά της μονάδα ρύθμισης για όλα τα συστατικά της. Αντιθέτως χωρίς το Helm θα πρέπει να γίνει διαχείριση για κάθε συστατικό της εφαρμογής ξεχωριστά έτσι ώστε να ρυθμιστεί στα κατάλληλα μέτρα

Για να επιτύχει τον σκοπό αυτό, το Helm χρησιμοποιεί το Helm Chart. Το Helm Chart αποτελεί ένα πακέτο το οποίο περιέχει όλους τους απαραίτητους πόρους για την εγκατάσταση του application στο cluster. Έτσι στο πακέτο αυτό περιέχονται και τα κατάλληλα yaml αρχεία που περιέχουν τις ρυθμίσεις για τα διάφορα συστατικά της εφαρμογής Kubernetes, όπως deployments, services, config maps κλπ. Το πακέτο Helm επίσης περιέχει πρότυπα που μπορούν να χρησιμοποιηθούν για την ρύθμιση μέσω παραμέτρων της εφαρμογής. Έτσι μπορεί η εφαρμογή να ρυθμιστεί ανάλογα το περιβάλλον και τις προτιμήσεις. Τέλος το Helm chart μπορεί να υποστηρίζει πολλές εκδόσεις του οι οποίες είναι ανεξάρτητες και έτσι κάθε έκδοση μπορεί να συντηρηθεί ξεχωριστά.

Η αρχιτεκτονική του Helm έχει δύο βασικά συστατικά. Το πρώτο αποτελεί τον Helm client, το οποίο αποτελεί το CLI (Command Line Interface) στο οποίο έχουν πρόσβαση οι χρήστες και μπορούν να διαχειριστούν τα τοπικά Helm Charts, όπως να τα εγκαταστήσουν, να αλλάξουν έκδοση κλπ. Το δεύτερο συστατικό αποτελεί το Helm library το οποίο περιέχει την υλοποίηση του κώδικα για της εντολές του Helm Client και εκτελεί τις ανάλογες ενέργειες. Επίσης η παραμετροποίηση ορισμένων μεταβλητών της εφαρμογής γίνεται μέσω του αρχείου values.yaml και με αυτόν τον τρόπο η αλλαγή τους διευκολύνεται σημαντικά σε αντίθεση με το να χρειάζεται η μεταβολή πολλών συστατικών του Kubernetes για την ίδια μεταβλητή.



Εικόνα 11. Η διαφορά εγκατάστασης εφαρμογής χωρίς και με το Helm

Το helm υποστηρίζει επίσης τα Helm repositories. Σε αυτά μπορούν να 'ανέβουν' Helm Charts και μπορούν να βρεθούν και οι χρήστες μέσω του διαδικτύου να έχουν πρόσβαση σε αυτά για την γρήγορη εγκατάστασή τους.

Τα helm charts που διατίθενται στον χρήστη είτε από τοπικό αρχείο είτε από helm repository μπορούν να εγκατασταθούν μέσω της εντολής helm install. Η εντολή αυτή περιέχει και παραμέτρους για την προσαρμογή των παραμέτρων του Chart καθώς και την επιλογή για προσθήκη values.yaml του χρήστη. Επίσης μπορεί να δημιουργηθεί μέσω της εντολής καινούργιο namespace στο Kubernetes cluster και συνεπώς η εντολή θα εκτελείται στο συγκεκριμένο namespace. Ένα παράδειγμα χρήσης της εντολής είναι η εξής:

```
$ helm install -f myvalues.yaml myredis ./redis
```

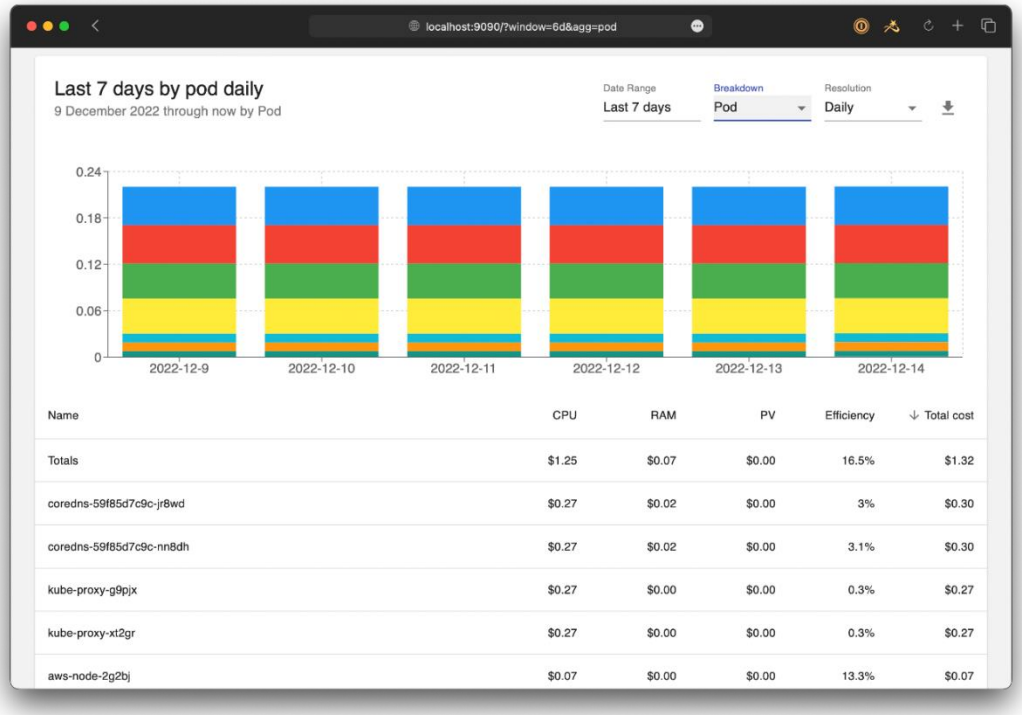
Επιπλέον το helm υποστηρίζει και uninstall ενός chart και μπορεί με ευκολία να αφαιρεθεί κάποιο που πλέον δεν χρειάζεται να εκτελείται στο cluster.

3.5 OpenCost

Το OpenCost αποτελεί ένα έργο ανοιχτού κώδικα για τη μέτρηση και την κατανομή του κόστους σε δομές cloud και των υπολογισμό κόστους container. Είναι κατασκευασμένο έτσι ώστε να παρέχει παρακολούθηση κόστους σε πραγματικό χρόνο για τη χρέωση των διάφορων συστατικών του Kubernetes [28]. Επίσης παρέχει

δυναμική on-demand τιμολόγηση των Kubernetes στοιχείων με δυνατότητα διασύνδεσης με AWS, Azure και GCP billing APIs. Επιπλέον, μπορεί να εξάγει τις μετρικές χρέωσης του στο Prometheus συνδυάζοντας αποτελεσματικά και τις δύο πλατφόρμες, μέσω του endpoint /metrics στο prometheus.

Το OpenCost διαθέτει και UI, το οποίο παρέχει μια οπτικοποίηση της κατανομής του κόστους στο Kubernetes και του σχετικού κόστους στο cloud. Το UI παρέχει πολλές επιλογές για την ομαδοποίηση του κόστους βάσει των επιλεγμένων κριτήριων (π.χ. βάσει namespace, node κλπ.) και στο επιθυμητό time range. Έτσι παράγονται γραφήματα τα οποία μπορούν να αναλυθούν με ευκολία.



Εικόνα 12. Το UI του OpenCost

Το OpenCost προσφέρει επίσης ένα ευέλικτο και πλούσιο API για τη συλλογή real time μετρικών. Επίσης αναφέρει το cloud κόστος του Kubernetes βασισμένο στην τιμολόγηση που του παρέχει ο χρήστης είτε βασισμένο στο κόστος που ανέκτησε από αναφορές των παρόχων cloud. Το κόστος αυτό ουσιαστικά αποτελεί μια μέτρηση των διάφορων πόρων που διαθέτει το cluster (CPU, RAM, PV) επί έναν πολλαπλασιαστή που ορίζει ο κάθε πάροχος. Έτσι η μετρική αυτή μπορεί να περιγράψει την κατανάλωση πόρων εντός του cluster και να χρησιμοποιηθεί σε αναλύσεις του Kubernetes συστήματος. Το κύριο query του OpenCost για τα κόστη και αυτό που θα χρησιμοποιηθεί για αυτό το έργο αποτελεί το query στο endpoint /allocation. Οι παράμετροι που λαμβάνει το query αυτό είναι οι εξής:

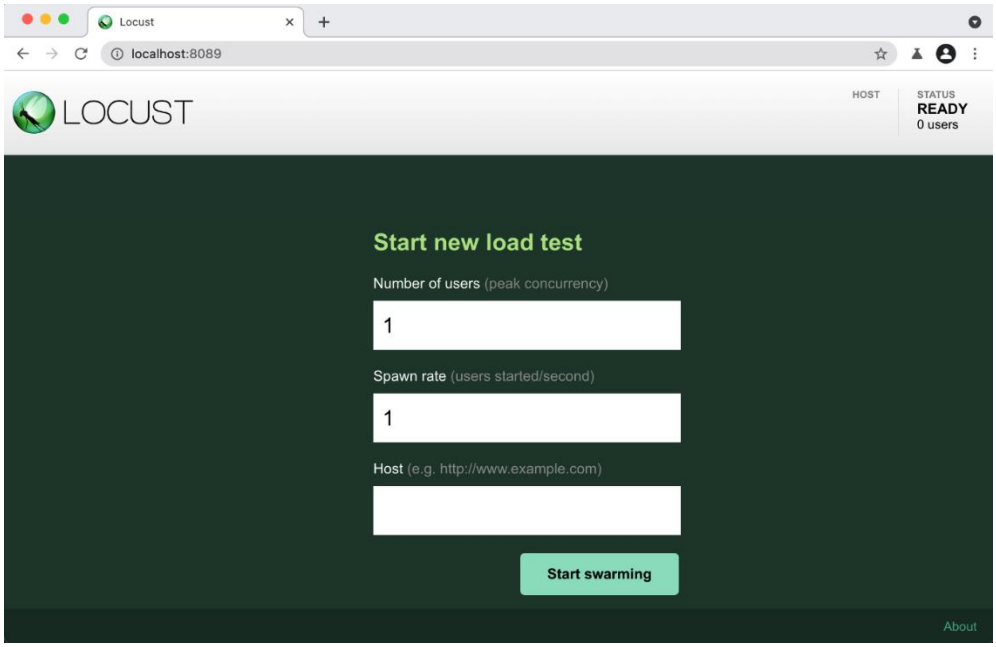
- Window: Αποτελεί το υποχρεωτικό πεδίο του query και δηλώνει την χρονική διάρκεια για την οποία θέλει ο χρήστης να λάβει δεδομένα από το query.
- Aggregate: Δηλώνει ως προς ποιο πεδίο επιθυμεί ο χρήστης να αθροίσει τα αποτελέσματα. Το πεδίο αυτό μπορεί να είναι το namespace, pods, nodes και άλλα.

- Step: Δηλώνει τη χρονική διάρκεια ενός συνόλου συλλογής δεδομένων, δηλαδή το παράθυρο στο οποίο γίνονται οι μετρήσεις για την παραγωγή ενός αποτελέσματος. Εάν δεν ορισθεί τότε λαμβάνει την τιμή της παραμέτρου window και συνεπώς από το query θα παραχθεί μόνο ένα αποτέλεσμα.
- Resolution: Δηλώνει τη χρονική διάρκεια που θα χρησιμοποιηθεί από το Prometheus queries για το πόσο συχνά γίνεται δειγματοληψία. Έτσι μικρές τιμές παρέχουν υψηλή ακρίβεια στη μέτρηση αλλά το query εκτελείται πιο αργά. Αντιθέτως μεγάλες τιμές παρέχουν χαμηλή ακρίβεια μέτρησης αλλά γρήγορη εκτέλεση query.

Το αποτέλεσμα του query παρέχει λεπτομερείς μετρικές για τα ζητούμενα πεδία που γίνονται aggregate καθώς επίσης προσφέρει και πλούσια πληροφορία για τα χαρακτηριστικά τους.

3.6 Locust

Το Locust αποτελεί ένα εργαλείο ελέγχου απόδοσης και φορτίου ανοιχτού κώδικα για HTTP και άλλα πρωτόκολλα. Οι ρουτίνες του ορίζονται μέσω Python από τον προγραμματιστή μέσω κώδικα και μπορούν να εκτελεστούν είτε από γραμμή εντολών είτε μέσω ενός φιλικού προς τον χρήστη UI που προσφέρει. Τα αποτελέσματα του παράγονται σε πραγματικό χρόνο και μπορεί να γίνει εξαγωγή τους για μεταγενέστερη ανάλυση. Είναι συμβατό με άλλες βιβλιοθήκες Python κάνοντας το εξαιρετικά επεκτάσιμο και θέτει λίγους περιορισμούς ως προς την ανάπτυξη των δοκιμών [29].



Εικόνα 13. Παράδειγμα του UI του Locust

Εικόνα 14.Real time αποτελέσματα μέσω του UI

Το Locust επίσης επιτρέπει την εκτέλεση παράλληλων δοκιμών σε πολλά διαφορετικά μηχανήματα. Επιπλέον είναι event-based και επομένως επιτρέπει σε μια διεργασία να χειρίζεται πολλούς χρήστες ταυτόχρονα και έτσι επιτρέπει υψηλότερη παραλληλία σε σχέση με άλλα εργαλεία δοκιμών. Σημαντικό χαρακτηριστικό του locust επίσης είναι ότι λειτουργεί κυρίως σε ιστοσελίδες και Services και επομένως επιτρέπει τη δοκιμή σε σχεδόν όλα τα συστήματα και πρωτόκολλα.

Το Locust προσφέρει μάλιστα και ένα Helm Chart (στα οποία η χρήση τους γίνεται κατανοητή στο κεφάλαιο)το οποίο μπορεί να εκτελεί δοκιμές σε Kubernetes συστήματα ορίζοντας κάποιες παραμέτρους κατά την εγκατάσταση, όπως το αρχείο Python που θεωρούμε ως locustfile, τον διακομιστή στον οποίο θα γίνει η δοκιμή και άλλα.

Κεφάλαιο 4: Διαδικασία scheduling στο Kubernetes

4.1 Kubernetes Scheduling

Ο scheduler του Kubernetes παρατηρεί pods που δημιουργήθηκαν και δεν έχουν ακόμα ανατεθεί σε κάποιο node. Ο scheduler είναι υπεύθυνος να αναθέσει το κάθε pod στο καλύτερο node το οποίο θα επιλέξει βασισμένος σε ορισμένες αρχές που έχουν τεθεί εκ των προτέρων στο cluster και συγκεκριμένα προϋπάρχουν στον kube-scheduler. Ο kube-scheduler αποτελεί τον προεπιλεγμένο scheduler για το Kubernetes ωστόσο μπορεί να αντικατασταθεί με έναν προσαρμοσμένο scheduler ανάλογα με τις απαιτήσεις τις εφαρμογής και τις επιλογές του σχεδιαστή, κάτι το οποίο θα συζητηθεί εκτενώς στα επόμενα κεφάλαια.

Αρχικά στο cluster, τα nodes τα οποία ικανοποιούν τις απαιτήσεις του scheduler θεωρούνται ικανά nodes (feasible) ενώ τα υπόλοιπα απορρίπτονται. Εάν δεν υπάρχουν ικανά nodes, τότε το pod δεν ανατίθεται σε κάποιο αλλά αναμένει έως ότου κάποιο γίνει ικανό να το υποστηρίξει. Όταν υπάρχει πληθώρα ικανών nodes, τότε ο scheduler υλοποιεί μια σειρά συναρτήσεων οι οποίες βαθμολογούν το κάθε node βάσει ορισμένων ιδιοτήτων ή μετρικών όπως η απαιτήσεις σε πόρους, το hardware και software, περιορισμούς, διάφορες προδιαγραφές κλπ. Έτσι το node με το μεγαλύτερο score κρίνεται το πλέον κατάλληλο και ο scheduler ειδοποιεί τον API server για την απόφασή του μέσω μιας διαδικασίας που ονομάζεται binding.

Όσο αναφορά τον kube-scheduler, η διαδικασία επιλογής του καλύτερου node περιλαμβάνει δύο στάδια, το Filtering και το Scoring. Στο στάδιο Filtering εντοπίζονται τα nodes τα οποία είναι ικανά να εκτελέσουν το Pod έπειτα από μια διαδικασία ελέγχων. Στο στάδιο Scoring γίνεται η επιλογή. Τα βήματα αυτά αποτελούν τον ‘πυρήνα’ του kube-scheduler καθώς στη διαδικασία του scheduling εμπλέκονται και άλλα βήματα μέσω του scheduling framework.

Το scheduling framework αποτελεί μια επεκτάσιμη αρχιτεκτονική για τον Kubernetes scheduler η οποία αποτελείται από ένα σύνολο πρόσθετων API (plugins) τα οποία μεταγλωττίζονται στον scheduler [30]. Ορισμένα από αυτά τα plugins επηρεάζουν τη διαδικασία scheduling ενώ αλλά προσφέρουν μόνο πληροφορία σχετικά με το scheduling. Επίσης κάποιο plugin μπορεί να καλείται πάνω από μια φόρα εντός της ροής του scheduling framework.

Για το scheduling ενός pod παρουσιάζονται δύο φάσεις κατά τη ροή του scheduling framework: ο κύκλος δρομολόγησης (scheduling) και ο κύκλος δέσμευσης (binding cycle). Κατά τον κύκλο δρομολόγησης γίνεται η επιλογή σε ποιο Node θα ανατεθεί το Pod ενώ στον κύκλο δέσμευσης εφαρμόζεται η επιλογή αυτή στο cluster. Και οι δύο κύκλοι μαζί αποτελούν το πλαίσιο scheduling. Επίσης αξίζει να σημειωθεί ότι οι κύκλοι δρομολόγησης εκτελούνται σειριακά ενώ οι κύκλοι δέσμευσης μπορούν να εκτελεστούν και παράλληλα.

Στην ροή του Scheduling framework εκτελούνται τα εξής plugins για την επίτευξη του scheduling ενός pod:

- PreEnqueue: Τα plugins αυτά καλούνται πριν προστεθούν τα Pods στην ενεργή ουρά για scheduling και εκτελείται έλεγχος εάν τα Pods είναι έτοιμα για scheduling. Εάν κριθούν ακατάλληλα για scheduling τοποθετούνται σε μια εσωτερική λίστα.
- EnqueExtension: Αποτελεί διεπαφή η οποία επιτρέπει στα plugins που συνδέονται να γίνει επανάληψη της διαδικασίας scheduling για τα pods που

βρίσκονται στην εσωτερική λίστα. Plugins που υλοποιούν το PreEnqueue, PreFilter, Filter και Reserve θα πρέπει να υλοποιούν και αυτή τη διεπαφή.

- QueueingHint: Αποτελεί συνάρτηση που αποφασίζει εάν μπορεί ένα Pod να ξαναμπεί στην ενεργή ουρά scheduling και εκτελείται όταν συμβαίνουν συγκεκριμένα γεγονότα εντός cluster, που θεωρεί ότι κάνουν το Pod ικανό για δρομολόγηση.

Τα plugins που βρίσκονται εντός του κύκλου δρομολόγησης είναι τα εξής:

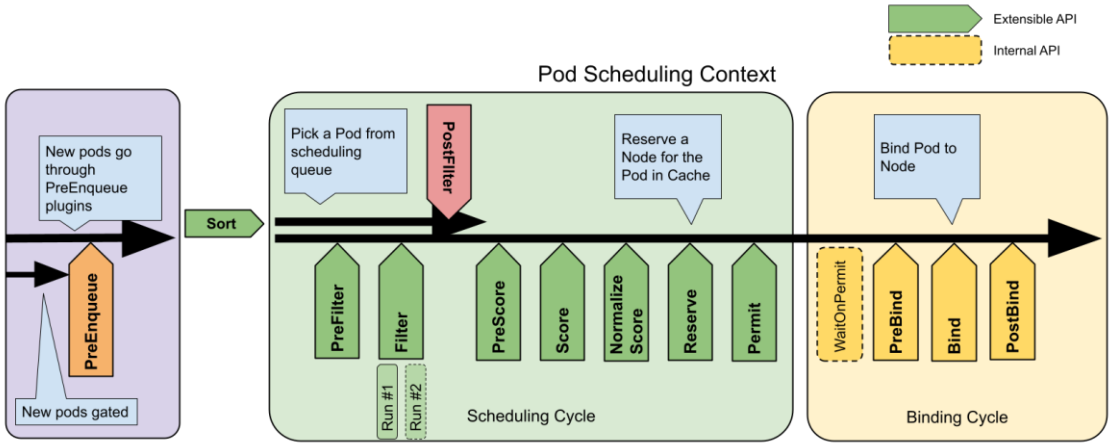
- Prefilter: Τα plugins αυτά καλούνται για την προεπεξεργασία των Pods έτσι ώστε να ελέγξουν εάν τηρούνται ορισμένες συνθήκες τόσο στο pod όσο και στο cluster. Σε περίπτωση λάθους ο κύκλος δρομολόγησης ματαιώνεται.
- Filter: Καλούνται έτσι ώστε να απορριφθούν τα nodes τα οποία δεν μπορούν να εκτελέσουν το pod. Συγκεκριμένα για κάθε node θα επικαλεστούν όλα τα filter plugins με την ορισμένη τους σειρά και σε περίπτωση που ένα από αυτά κρίνει το Node ακατάλληλο, τα υπόλοιπα δεν θα εκτελεστούν. Τα plugins μπορούν να ελέγχουν τα nodes παράλληλα.
- PostFilter: Καλούνται έπειτα από τα Filter plugins μόνο όταν όλα τα nodes κρίνονται ακατάλληλά για το pod. Εκτελεί παρόμοια λειτουργία με τα Filter plugins προσπαθώντας να κάνει το Pod ικανό για scheduling.
- PreScore: Εκτελούν μια πρώιμη βαθμολόγηση δημιουργώντας μια κοινή κατάσταση που θα χρησιμοποιηθεί από τα Score plugins. Σε περίπτωση σφάλματος ο κύκλος δρομολόγησης ακυρώνεται.
- Score: Εκτελείται η βαθμολόγηση για κάθε node που κρίθηκε κατάλληλο από τη φάση φιλτραρίσματος. Κάθε plugin καλείται για κάθε κατάλληλο node και προσθέτει στην συνολική βαθμολόγηση του node. Η βαθμολόγηση λαμβάνει ακέραιές τιμές από ένα ορισμένο σύνολο τιμών με ελάχιστη και μέγιστη τιμή.
- NormalizeScore: Καλούνται να κανονικοποίησουν τα αποτελέσματα του κάθε Score plugin. Άρα για κάθε Score plugin εκτελείται και ένα NormalizeScore plugin εντός του κύκλου. Σε περίπτωση σφάλματος ο κύκλος δρομολόγησης ματαιώνεται.
- Permit: Εκτελούνται στο τέλος του κύκλου Δρομολόγησης με σκοπό να αποτρέψουν ή να καθυστερήσουν τη δέσμευση ενός υποψήφιου node για scheduling. Εάν όλα τα plugins εγκρίνουν το pod, τότε μπορεί να ξεκινήσει ο κύκλος δέσμευσης ενώ εάν έστω ένα plugin απορρίψει το pod, τότε το pod επιστρέφει στην ουρά. Εάν έστω ένα plugin επιστρέψει “wait” τότε το pod αυτό αποθηκεύεται σε μια εσωτερική λίστα και ο κύκλος δέσμευσης του ξεκινά άλλα μπλοκάρεται μέχρι να εγκριθεί. Εάν προκύψει timeout κατά τη διαδικασία αυτή τότε το pod απορρίπτεται και ακολουθείται η αντίστοιχη διαδικασία.

Τα plugins που βρίσκονται εντός του κύκλου δέσμευσης είναι τα εξής:

- PreBind: Τα plugins αυτά εκτελούν τις απαραίτητες προεργασίες έτσι ώστε να εκτελεστεί η δέσμευση στο επόμενο στάδιο, όπως η παροχή χώρου δικτύου. Σε περίπτωση σφάλματος το pod επιστρέφει στην ουρά scheduling.
- Bind: Εκτελούν τη δέσμευση του Pod στο node και καλούνται εφόσον όλα τα PreBind plugins έχουν ολοκληρωθεί. Εκτελούνται με προκαθορισμένη σειρά

και κάθε plugin μπορεί να επιλέξει εάν θα εκτελεστεί για το συγκεκριμένο pod. Εάν ένα Plugin επιλέξει να δεσμεύσει το pod, τότε τα υπόλοιπα δεν εκτελούνται.

- PostBind: Εκτελούνται έπειτα από επιτυχία δέσμευσης του pod και αποτελούν κυρίως διεπαφή για πληροφορίες σχετικά με το scheduling καθώς επίσης μπορούν να εκτελέσουν και εκκαθάριση σχετικών πόρων που χρησιμοποιήθηκαν κατά την διαδικασία του scheduling.



Εικόνα 15.Η ροή του scheduling framework και τα plugins που χρησιμοποιούνται [30]

Έτσι ολοκληρώνεται η διαδικασία του Scheduling. Τα plugins που αναφέρθηκαν μπορούν να επεκταθούν και από τον σχεδιαστή ακολουθώντας το κατάλληλο πρότυπο και να εμπλουτίσουν τα ήδη υπάρχοντα. Επίσης τα plugins αυτά μπορούν να ενεργοποιηθούν και να απενεργοποιηθούν κατά βούληση. Το Kubernetes εκ των προτέρων έχει τα περισσότερα ενεργοποιημένα. Επίσης είναι εφικτό να οριστεί ένα σύνολο plugins το οποίο αποτελεί ένα προφίλ scheduling και να εναλλάσσεται με άλλα προφίλ ανάλογα τις ανάγκες.

Η υλοποίηση των plugins γίνεται στην γλώσσα προγραμματισμού Go και μάλιστα μπορούν να υλοποιούνται λειτουργίες από ένα ή περισσότερα plugins τόσο του κύκλου δρομολόγησης όσο και του κύκλου δέσμευσης. Επιπλέον μέσω labels στο yaml των pods μπορούν να προσαρμοστούν τιμές των plugins του scheduler η ακόμα και να απενεργοποιηθούν. Επομένως, φανερώνεται η μεγάλη επεκτασιμότητα που διαθέτουν οι schedulers στο Kubernetes

4.2 Τεχνικές scheduling

Στο κεφάλαιο αυτό θα συζητηθούν τεχνικές, μετρικές που χρησιμοποιούνται αλλά και περιορισμοί κατά τη διαδικασία του scheduling σε ευρύτερη έννοια έτσι ώστε να γίνει κατανοητή η ανάπτυξη των μετέπειτα αλγόριθμων.

Αρχικά θα αναλυθεί ο μαθηματικός τρόπος με τον οποίο ο scheduler καταλήγει στην απόφαση για τη βαθμολόγηση. Συγκεκριμένα, ιδιαίτερης σημασίας χρίζει ο μαθηματικός τύπος που χρησιμοποιήθηκε για τον υπολογισμό της βαθμολογίας κάθε node. Για την επίλυση προβλημάτων Scheduling χρησιμοποιούνται scheduling αλγόριθμοι οι οποίοι ως στόχο έχουν τη βέλτιστη τοποθέτηση μιας εργασίας στο κατάλληλο Node. Οι αλγόριθμοι αυτοί είναι NP-hard προβλήματα και συνεπώς δεν

υπάρχει αποδοτικός τρόπος εύρεσης της βέλτιστης λύσης για μεγάλα προβλήματα. Για την επίλυση του προβλήματος scheduling χρησιμοποιούνται οι εξής τύποι αλγόριθμων [31]:

- Ευριστικοί αλγόριθμοι: Οι αλγόριθμοι scheduling αυτού του τύπου χρησιμοποιούνται ευρέως από πολλούς scheduler και παρέχουν χαμηλή πολυπλοκότητα και ικανοποιητικά αποτελέσματα, όχι όμως τα βέλτιστα. Παράδειγμα τέτοιου αλγόριθμου είναι ο First-fit [32], ένας bin packing αλγόριθμος που τοποθετεί το pod στο πρώτο node που έχει τους διαθέσιμους πόρους να το υποστηρίξει.
- Integer Linear Programming (ILP) αλγόριθμοι: Οι αλγόριθμοι αυτοί παρουσιάζουν αυξημένη πολυπλοκότητα, κάνοντας τους πιο ικανούς σε μικρά προβλήματα. Οι αλγόριθμοι αυτοί θεωρούν το πρόβλημα ως μια σειρά εξισώσεων και έπειτα με τεχνικές βελτιστοποίησης προσπαθούν να βρουν την βέλτιστη λύση. Παραδείγματα τέτοιων αλγορίθμων είναι: cutting plane methods, branch and bound, branch and cut κλπ. [33]
- Μεταευριστικοί αλγόριθμοι: Οι αλγόριθμοι αυτοί είναι αλγόριθμοι βελτιστοποίησης που προσπαθούν να εντοπίσουν το πιο ικανό και προσαρμόσιμο στο πρόβλημα node βάσει των δεδομένων που τους δίνονται. Οι αλγόριθμοι αυτοί χωρίζονται σε δύο κατηγορίες, τους γενετικούς αλγόριθμους και τους αλγόριθμους νοημοσύνης σμήνους. Παραδείγματα τέτοιων αλγόριθμων είναι οι άπληστοι αλγόριθμοι, οι αλγόριθμοι τοπικής αναζήτησης (Local Search) και ο Simulated Annealing [34]

- Αλγόριθμοι μηχανικής μάθησης: Στους αλγόριθμους αυτούς γίνεται εκπαίδευση σε μεγάλα σύνολα δεδομένων και βάσει των γνώσεων αυτών καλούνται να κάνουν τη βέλτιστη επιλογή στη διαδικασία του scheduling. Στον τύπο αλγορίθμων αυτών μπορεί να ενταχθούν και οι αλγόριθμοι βαθιάς μάθησης. Στους αλγόριθμους αυτούς μπορούν να ενταχθούν τα KNN, support vector machines, δέντρα απόφασης και MLP [35]

Οι αλγόριθμοι Scheduling όμως μπορούν να καταταχθούν και βάσει των μετρικών που χρησιμοποιούν έτσι ώστε να κάνουν την επιλογή του node. Συγκεκριμένα οι αλγόριθμοι μπορεί να έχουν πρόσβαση σε μετρικές σχετικές με το cluster και την υποδομή της εφαρμογής. Οι κυριότερες είναι οι εξής:

- Αξιοποίηση πόρων: Οι πόροι αυτοί περιγράφουν την κατάσταση υπολογιστικών δυνατότητων (CPU), την κατανάλωση μνήμης (RAM και PV) καθώς και τους δικτυακούς πόρους που διαθέτουν τα nodes. Είναι οι πιο διάσημοι μεταξύ των αλγόριθμων scheduling καθώς έχουν την μεγαλύτερη επίδραση στην ομαλή λειτουργία της εφαρμογής και του cluster.
- Συχνότητα αποτυχίας: Η συχνότητα αποτυχίας του node λαμβάνεται υπόψιν σε ορισμένους αλγόριθμους έτσι ώστε να μπορούν να κατανέμουν τα συστατικά της εφαρμογής με τέτοιον τρόπο έτσι ώστε να αυξηθεί το uptime της
- Ενέργεια: Σημαντική μετρική επίσης είναι και η ενέργεια που καταναλώνει το Node και πολλοί αλγόριθμοι τη λαμβάνουν υπόψη έτσι ώστε να μειώσουν την ενεργειακή κατανάλωση κατά τη χρήση της εφαρμογής, κάνοντας έτσι

οικονομικότερη τη συντήρηση του cluster και αποφεύγοντας ορισμένα προβλήματα που προκύπτουν.

- Μετρικές σχετικές με το cluster: Ιδιαίτερης σημασίας χρίζουν και μετρικές που περιγράφουν το cluster όπως η τοποθεσία, η σταθερότητα, η αξιοπιστία και η διαθεσιμότητα, πληροφορίες τις οποίες αξιοποιούν ορισμένοι αλγόριθμοι. Οι αλγόριθμοι αυτοί με τις συγκεκριμένες μετρικές προσπαθούν να προσαρμόσουν την εφαρμογή στις ανάγκες του cluster.
- Ανάγκες εφαρμογής και συστατικών της: Περιγράφει τις μετρικές που δηλώνουν τις ανάγκες της εφαρμογής τόσο σε χρόνο απόκρισης, υπολογιστικούς πόρους και χρόνο ολοκλήρωσης. Συγκεκριμένα οι ανάγκες σε πόρους για τα συστατικά της εφαρμογής είναι πολύ διάσημη μετρική για πολλούς αλγόριθμους.

Από τις μετρικές που αναφέρθηκαν, στους περισσότερους αλγόριθμους εξετάζεται η αξιοποίηση των υπολογιστικών πόρων λόγω της σημαντικής επίδρασης που έχουν στην απόδοση της εφαρμογής. Επίσης εκτός από αυτές τις μετρικές μπορούν να παραχθούν και άλλες από εξωτερικά εργαλεία ή από εργαλεία εντός του cluster, κάνοντας ικανή την ανάπτυξη περαιτέρω ειδικευόμενων αλγορίθμων scheduling. Οι μετρικές αυτές μάλιστα μπορούν να συλλεχθούν σε πραγματικό χρόνο και έτσι οι αλγόριθμοι που τις χρησιμοποιούν έχουν πραγματική εικόνα τόσο για το cluster όσο και για την εφαρμογή.

Ωστόσο οι αλγόριθμοι scheduling μπορούν να διαχωριστούν και βάσει στο αν κάνουν προβλέψεις για το μέλλον κατά την εφαρμογή τους η όχι. Συγκεκριμένα υπάρχουν αλγόριθμοι που βασίζονται στα δεδομένα που τους δίνονται κατά την εκτέλεση της εφαρμογής και η δρομολόγηση γίνεται βάσει των δεδομένων αυτών. Μπορούν όμως να παραχθούν και αλγόριθμοι οι οποίοι επεκτείνουν τα δεδομένα αυτά μέσω προβλέψεων και συνεπώς η δρομολόγηση λαμβάνει και την πιθανή μελλοντική κατάσταση του cluster υπόψη της. Οι αλγόριθμοι αυτοί περιέχουν τεχνικές πρόβλεψης σχετικά με τη συμπεριφορά της εφαρμογής και τις απαιτήσεις των συστατικών της σε πόρους.

Κεφάλαιο 5: Περιγραφή αλγόριθμων scheduling

5.1 NetMARKS scheduler

Ο NetMARKS αποτελεί έναν αλγόριθμο scheduling για Kubernetes βασισμένος σε μετρικές δικτύου που συλλέχθηκαν από το Istio Mesh. Ο αλγόριθμος δημιουργήθηκε και μελετήθηκε από την Samsung R&D Institute Poland σε συνεργασία με το Warsaw University of Technology με σκοπό τη βελτίωση των 5G δικτύων σε απόδοση και μείωση της καθυστέρησης στον χρόνο απόκρισης. Τα αποτελέσματα που παρήχθησαν υποστηρίζουν ως και 36% μείωση στον χρόνο απόκρισης ενός application και τη μείωση του εύρους κατά 50% εντός των Nodes [36]. Ο NetMARKS βασίζεται στις δυνατότητες που προσφέρει το Istio Mesh και το Prometheus για τη συλλογή μετρικών. Επίσης είναι σχεδιασμένος με τέτοιον τρόπο έτσι ώστε να μπορεί να επεκτείνει τον Kube-scheduler μέσω Kube Scheduler Extension μηχανισμών.

Αρχικά για την εκτέλεση του αλγόριθμου, θεωρείται δεδομένο ότι το Istio Mesh είναι ενεργοποιημένο σε όλα τα namespaces του cluster και το Prometheus είναι εγκατεστημένο και συνδεδεμένο με το Istio control plane. Επίσης αξίζει να σημειωθεί ότι ως application θεωρείται οντότητα του Istio Mesh και όχι του Kubernetes, καθώς αποτελεί ένα σύνολο Pods με ίδιο app label. Για να λάβει την απόφαση scheduling, ο NetMARKS χρειάζεται δύο μετρικές από το Prometheus: την istio_request_bytes_sum και istio_response_bytes_sum. Οι μετρικές αυτές περιγράφουν το σύνολο των bytes που μεταφέρονται από ένα application σε request και response πακέτα αντίστοιχα σε ένα άλλο application. Επίσης οι μετρικές αυτές αυξάνονται με τον χρόνο καθώς προσθέτουν στο υπάρχων σύνολο τους τις νέες μετρήσεις που εκτελούν ανά τακτά χρονικά διαστήματα.

Οι παραπάνω μετρικές χρησιμοποιούνται για τον υπολογισμό της μέσης ροής δεδομένων μεταξύ ενός application A και ενός application B σε ένα χρονικό διάστημα $[t_1,t_2]$ με μονάδα μέτρησης τα bytes ανά δευτερόλεπτο μέσω του τύπου (1). Στον τύπο η μεταβλητή $req^{A,B}{}_t$ συμβολίζει τον αριθμό των bytes σε requests από το application A στο application B έως τη χρονική στιγμή t, ενώ η μεταβλητή $resp^{A,B}{}_t$ συμβολίζει τον αριθμό των bytes σε responses από το application A στο application B έως τη χρονική στιγμή t.

$$F^{A,B}{}_{t_1,t_2} = \frac{\left(req^{A,B}{}_{t_2} - req^{A,B}{}_{t_1}\right) + \left(resp^{B,A}{}_{t_2} - resp^{B,A}{}_{t_2}\right)}{t_2 - t_1} \quad (1)$$

Οι μετρικές συλλέγονται περιοδικά και επομένως η μέση ροή μπορεί να υπολογιστεί από τον τύπο (2) από την αρχή των μετρήσεων.

$$F^{A,B}{}_{t_0,t_i} = \frac{F^{A,B}{}_{t_0,t_{i-1}}(t_{i-1} - t_0) + F^{A,B}{}_{t_{i-1},t_i}(t_i - t_{i-1})}{t_i - t_0} \quad (2)$$

Επομένως εάν θεωρηθεί ότι η τωρινή ροή για χρονική στιγμή t_i ως $F(t_i)$, τότε από τους τύπος (1) και (2) προκύπτει ο εξής τελικός τύπος:

$$F(t_i) = F(t_{i-1}) \cdot \frac{t_{i-1} - t_0}{t_i - t_0} + \frac{\left(req^{A,B}{}_{t_i} - req^{A,B}{}_{t_{i-1}}\right) + \left(resp^{B,A}{}_{t_i} - resp^{B,A}{}_{t_{i-1}}\right)}{t_i - t_{i-1}} \quad (3)$$

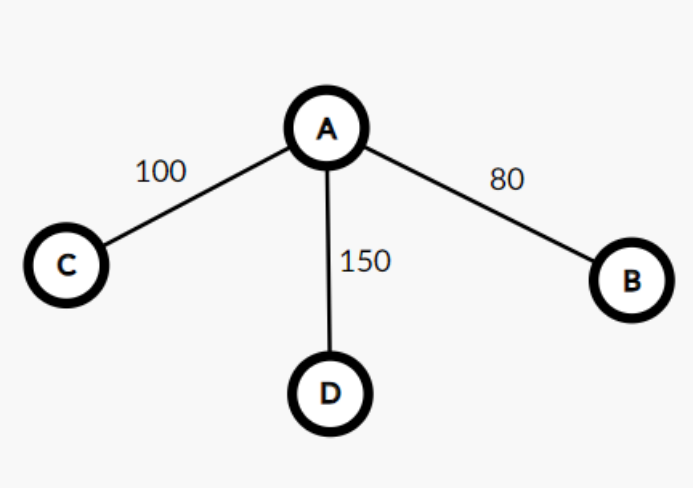
Η παραπάνω υπολογισμένη μετρική έχει πρόσβαση σε κάθε ικανό node που δεν έχει αποκλεισθεί από τον scheduler. Η μετρική αυτή αποτελεί το βασικό στοιχείο απόφασης του αλγόριθμου NetMARKS. Όσο αναφορά τον αλγόριθμο NetMARKS, δέχεται ως είσοδο δύο παραμέτρους, το Pod που θα γίνει scheduled και ένα node. Απαραίτητο για τον αλγόριθμο είναι επίσης να γνωρίζει την κατάσταση των υπόλοιπων pods και επομένως λαμβάνει την πληροφορία αυτή μέσω του Kubernetes API. Ο αλγόριθμος που ακολουθεί το NetMARKS για τη βαθμολόγηση των nodes είναι ο εξής:

Είσοδος: node, pod προς δρομολόγηση
Έξοδος: node

F(node, pod) :
1. Score = 0
2. State = Όλα τα pods που εκτελούνται στο Node
3. Stats = Τα pods που έχουν κίνηση από και προς το pod της εισόδου
4. Για κάθε pod x στο State και για κάθε pod y στο Stats κάνε:
    a. Εάν το x με το y ταυτίζονται τότε
        i. Score += κίνηση-flow μεταξύ Pod και x
5. Επίστρεψε το Score

Ο παραπάνω αλγόριθμος εκτελείται για κάθε node και εν τέλει επιλέγεται το node με το υψηλότερο score.

Ένα παράδειγμα εκτέλεσης του αλγορίθμου είναι αρχικά να έχουμε την εξής κατανομή των A, B, C και D pods που το κάθε ένα αποτελεί ένα ξεχωριστό application με ροές μεταξύ των applications που φαίνονται στο παρακάτω γράφημα:



Εικόνα 16.Κατανομή των applications με τις ροές μεταξύ τους

Θεωρούμε ότι τα pods C, D, B εκτελούνται ήδη σε 3 ξεχωριστούς nodes, όσους διαθέτει και το cluster μας και θέλουμε να δρομολογήσουμε το pod A. Τότε έχουμε τις εξής καταστάσεις βάσει του αλγόριθμου:

- **Node1:** State = {C}, Score = 100
- **Node2:** State = {D}, Score = 150
- **Node3:** State = {B}, Score = 80

Επομένως το υψηλότερο score το έχει ο κόμβος node2 και συνεπώς το pod A θα δρομολογηθεί εκεί.

Το αποτέλεσμα του αλγόριθμου μπορεί να θεωρηθεί ως το άθροισμα της κίνησης του Pod με άλλα pods εντός του node που έχει οριστεί στο Input. Επομένως για το NetMARKS ο καλύτερος node για το pod είναι συνοπτικά αυτός με την περισσότερη επικοινωνία εντός του node με αλλά pods.

Παρατηρείται ότι ο NetMARKS βασίζεται σε μια ήδη υπάρχουσα κατανομή των pods σε nodes και επομένως η αρχική κατανομή των pods συμβάλλει στο αποτέλεσμα του αλγόριθμου. Για να αντιμετωπίσει το πρόβλημα με την αρχική κατανομή, η εφαρμογή της εκτελείται με κάποια κατανομή και κατά την εκτέλεση της συλλέγονται οι μετρικές. Έπειτα εκτελούνται 1000 διαφορετικές εγκαταστάσεις της εφαρμογής με τυχαία αρχική κατανομή και έπειτα scheduling βάσει του NetMARKS. Από αυτές τις εγκαταστάσεις επιλέγεται η καλύτερη. Οι εγκαταστάσεις μπορούν να μειωθούν εάν τα Nodes έχουν το ίδιο hardware.

Αξίζει να σημειωθεί επίσης ότι η επιλογή που κάνει ο NetMARKS είναι η καλύτερη βασισμένη στην τρέχουσα κατάσταση του cluster, των pods και των μετρικών δικτύων. Η μεταβολή των στοιχείων αυτών μπορεί να επηρεάσει σημαντικά τα αποτελέσματα επιλογής. Επίσης σημαντικό ρόλο για το τελικό αποτέλεσμα έχει να κάνει και η σειρά με την οποία γίνονται schedule όλα τα Pods, καθώς με διαφορετική σειρά scheduling μπορούν να παραχθούν διαφορετικά αποτελέσματα. Μάλιστα η σειρά που είναι βέλτιστη να δρομολογηθούν τα pods δεν μπορεί να υπολογιστεί από τον αλγόριθμο.

5.2 Bin Balancer scheduler

Η default διαδικασία scheduling στο Kubernetes λαμβάνει υπόψη της κατά τη διάρκεια του scoring πολλές μετρικές για τη βαθμολόγηση των Nodes έτσι ώστε να επιλέξει το καλύτερο node, συνδυάζοντας τις βαθμολογίες από όλες τις μετρικές. Κατά τη διαδικασία αυτή, εκτελείται και bin packing βάσει διάφορων μετρικών για την καλύτερη επιλογή node εντός του scheduling plugin NodeResourcesFit [37]. Ο αλγόριθμος scheduling που περιγράφεται στο κεφάλαιο αυτό εκτελεί στο scoring μόνο το bin packing με μετρική αναφορά το κόστος που παράγεται από το OpenCost.

Το bin packing αποτελεί ένα πρόβλημα βελτιστοποίησης το οποίο λαμβάνει πολλές μορφές και διαθέτει πολλούς τρόπους επίλυσης. Η μορφή που επιθυμεί να επιλύσει ο default scheduler είναι να μοιράσει αντικείμενα διαφορετικού βάρους (στη συγκεκριμένη περίπτωση pods) σε έναν συγκεκριμένο αριθμό από καλάθια-bins (στην περίπτωση αυτή τα nodes) με συγκεκριμένη χωρητικότητα (περιορισμένα resources) και υπάρχοντα αρχικά βάρη (άλλα pods που εκτελούνται στα nodes). Τα βάρη θα πρέπει να μοιραστούν ισάξια στα καλάθια έτσι ώστε η κατανομή των βαρών να είναι όσο το δυνατόν πιο ομοιόμορφη. Το πρόβλημα αυτό είναι NP δύσκολο και οι αλγόριθμοι που υπάρχουν για την επίλυση του είναι πολλοί παρά την πολυπλοκότητά

του. Μάλιστα, ένας μεγάλος αριθμός από τους αλγόριθμους αυτούς απλώς επιθυμούν να βρουν μια ικανοποιητική λύση με μικρή πολυπλοκότητα δίχως όμως να εντοπίζουν τη βέλτιστη λύση [38].

Ο scheduler που αναπτύσσεται στην εργασία αυτή κατανέμει ομοιόμορφα τα pods στα nodes βάσει της μετρικής του συνολικού κόστους των pods που παράγει το OpenCost. Η μετρική κόστους αυτή, όπως αναφέρθηκε και στο κεφάλαιο OpenCost, περιγράφει την κατανάλωση πόρων εντός του cluster (CPU, memory) κατά τη διάρκεια του χρόνου επί ενός συντελεστή για να ορίσει το κόστος τους, και επομένως μπορεί να χρησιμοποιηθεί ως βάρος για την κατανομή των pods. Ο scheduler αυτός μπορεί να είναι επιθυμητός από ένα σύστημα καθώς στο cluster ιδανικά τα κόστη θα πρέπει να είναι ίδια σχεδόν σε κάθε node. Έτσι, κάποιο Node δεν θα είναι υπερφορτωμένο και η εργασία θα μοιράζεται ισόποσα στο σύστημα, προσφέροντας καλύτερη απόδοση και σταθερότητα συνολικά. Επίσης σε περίπτωση που ένα node «πέσει», ένα μικρό ποσοστό των Pods θα σταματήσει να λειτουργεί σε αντίθεση με το να πέσει ένα Node που εκτελεί πολλά και «βαριά» pods. Τέλος, γίνεται και καλύτερη αξιοποίηση των πόρων από το σύστημα [39] .

Ο Bin Balancer scheduler για να εκτελεστεί, αρχικά θα πρέπει να έχει συλλέξει τα βάρη-κόστη από τα pods που θα δρομολογήσει. Αυτό γίνεται μέσω μιας αρχικής εκτέλεσης της εφαρμογής με οποιαδήποτε κατανομή pods. Μια χρονική στιγμή το OpenCost συλλέγει το συνολικό κόστος του κάθε pod και τα αποθηκεύει σε ένα αρχείο json. Επίσης συλλέγει και τα κόστη τυχόν άλλων εκτελουμένων pods στα nodes έτσι ώστε να αρχικοποιήσει τα καλάθια με αρχικό βάρος και να παράξει στο τέλος σωστά την ομοιόμορφη κατανομή. Το αρχείο περιέχει και πληροφορίες για τα επιμέρους κόστη του pod όπως φαίνεται και παρακάτω:


```
"cart": {
    "cpu": 0.08535,
    "ram": 0.01219,
    "totalCost": 0.09754
  },
  "catalogue": {
    "cpu": 0.08535,
    "ram": 0.01254,
    "totalCost": 0.09789
  },
  "dispatch": {
    "cpu": 0.08535,
    "ram": 0.0114,
    "totalCost": 0.09674
  }
```


Έπειτα βάσει των μετρήσεων αυτών εκτελείται ο αλγόριθμος bin packing. Ο αλγόριθμος δέχεται ως είσοδο τα Pods και τα κόστη τους. Αξίζει να σημειωθεί ότι η χωρητικότητα των nodes θα θεωρηθεί αρκετά υψηλή και επομένως δεν θα ληφθεί υπόψη κατά τους υπολογισμούς του αλγορίθμου**.** Ο αλγόριθμος είναι ο εξής:

Είσοδος: αρχική κατανομή βάρους ως dictionary για κάθε node
αρχική κατανομή pods ως dictionary για κάθε node
index βάρους και pods ως integer,
βάρη ως λίστα
pods ως λίστα
Έξοδος: τελική κατανομή pods

F(κατανομής βάρους, κατανομή pods, index, βάρη, pods):

1. Εάν το index βάρους ισούται με τον αριθμό των βαρών, επίστρεψε την κατανομή βάρους και pods που δόθηκε ως είσοδος
2. Για κάθε node κάνε, κρατώντας σταθερή την κατανομή βάρους και pods στην είσοδο:
    a. Στην κατανομή βαρών πρόσθεσε το βάρος που δείχνει το index στο τρέχον Node.
    b. Στην κατανομή pods πρόσθεσε το pod που δείχνει το Index στο τρέχον Node.
    c. Υπολόγισε την καλύτερη κατανομή βαρών και κατανομή Pods που παράγεται καλώντας την F (νέα κατανομή βαρών, νέα κατανομή pods, index + 1, βάρη, pods)
    d. Υπολόγισε για αυτήν την κατανομή την μετρική ισορροπίας κόστους που αναφέρεται στο κεφάλαιο 7.1.3
3. Από τα αποτελέσματα κάθε node επίλεξε την κατανομή με τη μικρότερη μετρική ισορροπίας κόστους
4. Επίστρεψε την κατανομή βάρους και κατανομή pods που επιλέχθηκε

Αρχικά παρατηρείται ότι ο αλγόριθμος είναι αναδρομικός και σε κάθε εκτέλεση του επιχειρεί να τοποθετήσει το βάρος που ελέγχεται στο καλύτερο δυνατό node. Έτσι ελέγχονται όλες οι πιθανές κατανομές των pods που θα δρομολογηθούν.

Με τον τρόπο αυτόν παράγεται η κατανομή των pods στο cluster. Σημειώνεται ότι ο αλγόριθμος λαμβάνει υπόψη του και τα κόστη που προϋπάρχουν στα nodes από άλλα pods και τα θέτει ως αρχικά βάρη. Σε αντίθετη περίπτωση η ομοιόμορφη κατανομή θα γινόταν μόνο στα pods που θα ήταν προς δρομολόγηση και με την προσθήκη τους η ιδιότητα της κατανομής πιθανώς να χανόταν.

Μάλιστα ο αλγόριθμος αυτός είναι εκθετικής πολυπλοκότητας άλλα βρίσκει κάθε φορά την κατανομή με τη μεγαλύτερη ομοιομορφία. Επίσης η πολυπλοκότητα αυτή δεν είναι απαγορευτική για τον Scheduler καθώς για την δρομολόγηση του συγκεκριμένου συνόλου pods ο αλγόριθμος θα εκτελεστεί μια φορά και μόνο. Ωστόσο δεν είναι δυναμικός, καθώς σε περίπτωση προσθήκης ενός νέου pod θα πρέπει να υπολογιστεί το συνολικό του κόστους και να εκτελεσθεί ξανά ο υπολογισμός της κατανομής.

Με τον συγκεκριμένο τρόπο δρομολόγησης γίνεται εστίαση μόνο στο κομμάτι της ομοιόμορφης κατανομής των πόρων στα nodes με βέλτιστο τρόπο. Παρόλο που και ο default scheduler εκτελεί bin balancing, δεν προκύπτει πάντα η βέλτιστη κατανομή καθώς η απόφαση μπορεί να επηρεαστεί και από πολλούς άλλους παράγοντες. Ο Bin Balancer scheduler αγνοεί τις μετρικές αυτές και θυσιάζει τυχόν άλλα θετικά με σκοπό τη βελτιστοποίηση της κατανομής.

5.3 Combined scheduler

Οι schedulers NetMARKS και Bin Balancer εστιάζουν σε μία μόνο βελτίωση του συστήματος αδιαφορώντας για τους υπόλοιπους παράγοντες που μπορεί να επηρεάσουν το cluster αρνητικά. Συγκεκριμένα ο NetMARKS επιθυμεί να μειώσει το συνολικό χρόνο απόκρισης της εφαρμογής, ωστόσο παρατηρώντας τον αλγόριθμο, γίνεται αντιληπτό ότι προσπαθεί να συγκεντρώσει όσο το δυνατόν περισσότερα pods σε ένα node. Με τον τρόπο αυτό μπορεί να υπάρξει αχρείαστη υπερφόρτωση του node και να παρουσιαστούν προβλήματα. Αντιστοίχως, ο αλγόριθμος Bin Balancer επικεντρώνεται μόνο στην εξισορρόπηση του κόστους στα nodes αδιαφορώντας για το χρόνο απόκρισης. Έτσι μπορεί τα Pods που ανταλλάσσουν πολλά δεδομένα κατά της επικοινωνία τους να τοποθετηθούν σε διαφορετικά nodes, κάτι που μπορεί να οδηγήσει στην αύξηση του χρόνου απόκρισης της εφαρμογής.

Ιδανικά το cluster θα πρέπει να πετύχει καλό χρόνο απόκρισης όπως ο NetMARKS αλλά να είναι και ισορροπημένα βαρύ όπως ο Bin Balancer scheduler. Ένας τρόπος να επιτευχθεί αυτό θα ήταν να βρεθεί η «μέση λύση». Δηλαδή να χρησιμοποιηθούν τεχνικές και από τους δύο αλγόριθμους και αξιοποιώντας διάφορες πληροφορίες να βρεθεί μια κατανομή που εξισορροπεί σε έναν βαθμό τα κόστη και μειώνει επίσης σε έναν βαθμό το response time.

Με το σκεπτικό αυτό προκύπτει ο Combined scheduler που συνδυάζει θετικά και από τον NetMARKS και τον Bin Balancer. Η κατανομή που παράγει ο scheduler θα προσφέρει μια λύση ανάμεσα στα θετικά του NetMARKS και του Bin Balancer και θα μπορεί να προσαρμοστεί ανάλογα με τις ανάγκες μέσω ενός ορίου που θα εξηγηθεί περαιτέρω στο υπόλοιπο του κεφαλαίου αυτού. Απαραίτητη προϋπόθεση είναι ο υπολογισμός και αποθήκευση των μετρικών που απαιτούν ο NetMARKS και Bin Balancer έτσι ώστε να εκτελεστεί ο αντίστοιχος αλγόριθμος.

Αρχικά, όπως αναφέρθηκε και σε προηγούμενο κεφάλαιο, ο NetMARKS για να εκτελεστεί απαιτεί μια αρχική κατανομή των pods σε nodes. Η αρχική κατανομή που δινόταν αποτελούσε μια τυχαία κατανομή και εφόσον είχε εκτελεστεί ο NetMARKS επιλεγόταν η καλύτερη τελική κατανομή βάσει του χρόνου απόκρισης. Στον συγκεκριμένο scheduler, ως αρχική θα δοθεί η κατανομή που παράγει ο Bin Balancer scheduler που θα δημιουργηθεί με εκτέλεση του βασισμένος στις μετρικές που συλλέχθηκαν. Έτσι η αρχική κατανομή αποτελεί τη βέλτιστη ως προς την ομοιομορφία και έπειτα θα μεταβάλλεται από τον NetMARKS.

Έπειτα, για κάθε pod που θέλει να δρομολογηθεί θα πρέπει να αποφασιστεί εάν πρέπει να εκτελεσθεί ο αλγόριθμος NetMARKS για αυτό ή όχι. Συγκεκριμένα πρέπει να αποφασιστεί εάν το pod αυτό είναι καλό να χαλάσει πιθανώς την ισορροπία των nodes σε κόστος έτσι ώστε να βελτιώσει τον χρόνο απόκρισης. Όπως είναι φανερό, για να χρειαστεί το pod να μετακινηθεί θα πρέπει να έχει μεγάλη ροή επικοινωνίας με άλλα pods.

Επομένως τίθεται ένα όριο ροής το οποίο δηλώνει ποια pods θα πρέπει να δρομολογηθούν βάσει NetMARKS. Το όριο αυτό ορίζεται ως το άθροισμα της ροής (flows που έχουν υπολογιστεί ως μετρικές από τον NetMARKS) του app-pod με άλλα apps-pods. Έτσι, pods που έχουν ροή πάνω από το όριο θα δρομολογηθούν βάσει του NetMARKS ενώ τα υπόλοιπα όχι.

Το όριο αυτό όμως δεν είναι εύκολο να οριστεί χειροκίνητα καθώς ο σχεδιαστής δεν γίνεται να μελετήσει όλες τις ροές και να καταλήξει στην καλύτερη. Συνεπώς ακολουθείται η εξής διαδικασία που θα υπολογίσει το όριο:

1. Αρχικά τα apps κατατάσσονται με φθίνουσα σειρά βάσει της συνολικής ροής τους σε μια λίστα. Με τον τρόπο αυτό, ξεχωρίζουμε τα πιο απαιτητικά apps σε διαδικτυακούς πόρους από τα υπόλοιπα.
2. Ο σχεδιαστής επιλέγει ένα ποσοστό των συνολικών apps που επιθυμεί να βελτιώσει βάσει του NetMARKS. Τα apps που θα επιλεχθούν θα είναι με τη σειρά από την αρχή της λίστας καθώς αυτά κρίνονται τα σημαντικότερα για την εφαρμογή.
3. Έπειτα, το όριο ορίζεται ως η συνολική ροή του app με τη μικρότερη ροή που επιλέχθηκε, μειωμένο κατά ελάχιστο έτσι ώστε να μπορεί να δρομολογηθεί και το app αυτό.
4. Τέλος συνεχίζεται η διαδικασία της δρομολόγησης με το όριο που τέθηκε από τη διαδικασία αυτή.

Από την παραπάνω διαδικασία θα ήταν δυνατόν να εκτελεστεί άμεσα η δρομολόγηση στα apps που επιλέχθηκαν δίχως την ανάγκη να τεθεί το όριο. Ωστόσο επιλέχθηκε το όριο να τίθεται έτσι ώστε να είναι δυνατόν να οριστεί και χειροκίνητα, γεγονός αδύνατον στην προηγούμενη περίπτωση.

Ο combined αλγόριθμος αναμένεται να παράγει αποτελέσματα τα οποία βρίσκονται ενδιάμεσα εκείνων του NetMARKS και του Bin Balancer. Αξίζει να σημειωθεί ότι ο αλγόριθμός αυτός διαθέτει και δύο ακραίες περιπτώσεις:

- Για το όριο $\rightarrow \infty$ (ή μεγαλύτερο του μέγιστου συνολικού flow): Κανένα app δεν θα δρομολογηθεί και επομένως η τελική κατανομή θα είναι ίδια της αρχικής. Επομένως ο combined αλγόριθμος ταυτίζεται με τον Bin Balancer
- Για το όριο $= 0$: Όλα τα apps ξεπερνούν το όριο και επομένως θα δρομολογηθούν βάσει του NetMARKS. Συνεπώς, ο combined αλγόριθμος ταυτίζεται με τον NetMARKS

Ωστόσο για να βρεθεί η τοπολογία με τις ιδιότητες που συμφέρουν τον σχεδιαστή απαιτούνται αρκετές δοκιμές με διαφορετικά ποσοστά. Επίσης, εξαιτίας του Bin Balancer ο scheduler δεν είναι δυναμικός και με προσθήκη pod απαιτείται ο υπολογισμός των μετρικών και η επανεκτέλεση του αλγόριθμου.

Τέλος αξίζει να σημειωθεί ότι ως όριο θα μπορούσε και να οριστεί τιμή που συγκρίνεται με το κόστος των apps. Με τη λογική αυτή θα δρομολογούνται βάσει του NetMARKS τα apps που δεν ξεπερνούν τα όρια, ενώ τα υπόλοιπα θα παραμένουν στο Node που τους ορίστηκε από την αρχική κατανομή Bin Balancer. Τα apps που ξεπερνούν το όριο θεωρούνται ότι είναι ζωτικής σημασίας για την ισορροπία λόγω του υψηλού τους βάρους και δεν αξίζει να μετακινηθούν. Η προσέγγιση αυτή δεν μελετήθηκε ωστόσο αξίζει περαιτέρω έρευνας.

5.4 Θεωρητική σύγκριση ιδιοτήτων αλγορίθμων

Για την καλύτερη κατανόηση των schedulers που αναλύθηκαν στις προηγούμενες ενότητες, θα δοθεί ένας πίνακας με τις κύριες διάφορες τους και ομοιότητες και έπειτα θα αναλυθούν κάποιες από αυτές τι διάφορες περαιτέρω. Ο πίνακας είναι ο εξής:

Scheduler	Χρονική πολυπλοκότητα	Μετρική βαθμολόγησης	Απαιτεί αρχική κατανομή
default	υψηλή για μεγάλο αριθμό nodes	πόροι cluster, σχέση με άλλα Pods κλπ.	όχι
NetMARKS	πολυωνυμική	flow μεταξύ των apps	ναι
Bin Balancer	εκθετική	κόστος σε πόρους CPU, RAM και PV	όχι
combined	εκθετική	flow και κόστος	όχι

Πίνακας 1. Θεωρητική σύγκριση αλγορίθμων(1)

Scheduler	Δυναμικός	Βέλτιστη λύση (συνολικά)	Εργαλεία	Αρχικές μετρικές
default	ναι	όχι	Kubernetes API	όχι
NetMARKS	ναι	όχι	Istio, Prometheus	ναι
Bin Balancer	όχι	ναι	OpenCost, Prometheus	ναι
combined	όχι	όχι	Istio, OpenCost, Prometheus	ναι

Πίνακας 2. Θεωρητική σύγκριση αλγορίθμων(2)

Αρχικά παρατηρείται ότι ο combined συνδυάζει τις ιδιότητες του NetMARKS και του Bin Balancer scheduler, αναμενόμενο αποτέλεσμα καθώς αποτελείται και από τον δύο αλγόριθμους. Επίσης παρατηρείται ότι όλοι οι schedulers διαφέρουν αρκετά ακόμα και σε αυτές τις λίγες ιδιότητες τους που παρουσιάζονται τώρα.

Όσο αναφορά τη χρονική πολυπλοκότητα, οι αλγόριθμοι του NetMARKS και ο default είναι οι πιο γρήγοροι καθώς οι υπολογισμοί του γίνονται βάσει συγκρίσεων του pod στο cluster και ιδιοτήτων του με το pod που βρίσκεται προς δρομολόγηση εκείνη την στιγμή. Αντιθέτως, ο αλγόριθμος Bin Balancer εκτελείται με εκθετική πολυπλοκότητα λόγω του γεγονότος ότι για να εντοπίσει την κατανομή με την καλύτερη ισορροπία κόστους ελέγχει σχεδόν κάθε πιθανή κατανομή. Ο combined καθώς χρησιμοποιεί τον αλγόριθμο του Bin Balancer για να παράγει την αρχική κατανομή, έχει επίσης εκθετική πολυπλοκότητα.

Όσο αναφορά τις μετρικές με τις οποίες γίνεται η βαθμολόγηση, ο default scheduler λαμβάνει υπόψη μια πληθώρα μετρικών σχετικά με το pod και την κατάσταση του cluster και βάσει αυτών βαθμολογεί τα nodes. Ο NetMARKS ωστόσο η μόνη μετρική που τον ενδιαφέρει είναι η ροή επικοινωνίας σε byte μεταξύ των apps έτσι ώστε να βαθμολογήσει τα nodes, αδιαφορώντας για τις υπόλοιπες μετρικές. Ομοίως ο Bin Balancer, εξετάζει τα OpenCost κόστη των pods για να κάνει την επιλογή δίχως να λαμβάνει υπόψη του λοιπές μετρικές. Τέλος ο combined scheduler, ως συνδυασμός του NetMARKS και του Bin Balancer, εξετάζει μόνο τις αντίστοιχες

μετρικές που εξετάζουν οι δύο αλγόριθμοι αυτοί. Επομένως η βελτίωση που αναμένεται να έχουν οι Schedulers είναι στις σχετικές μετρικές που μελετούν.

Σχετικά με την αρχική κατανομή, μόνο ο scheduler NetMARKS χρειάζεται μια έτσι ώστε να μπορέσει να παράγει το αποτέλεσμα, καθώς βασίζεται στην θέση των apps στα Nodes για να επιλέξει το καλύτερο. Σχετικά με τον combined scheduler θα μπορούσε να ειπωθεί ότι χρειάζεται αρχική κατανομή για το κομμάτι του NetMARKS που εκτελεί, ωστόσο ο Bin Balancer παράγει αυτή την κατανομή εξ αρχής και δεν την χρειάζεται ως είσοδο.

Ως δυναμικότητα του scheduler, θεωρείται η δυνατότητα του να μπορεί να δρομολογήσει νέα pods δίχως να χρειαστεί ξανά δρομολόγηση και των προηγουμένων. Για τον default και NetMARKS scheduler αυτό είναι εφικτό καθώς βασισμένη στην τωρινή τους κατανομή των pods μπορούν να λάβουν την απόφαση για τον νέο pod και να το δρομολογήσουν, παράγοντας μια νέα κατανομή. Αντιθέτως ο Bin Balancer και συνεπώς ο combined scheduler, με προσθήκη νέου pod θα πρέπει να εκτελεστούν από την αρχή έτσι ώστε να υπολογιστεί πάλι η κατανομή pods με τη βέλτιστη ισορροπία στα nodes.

Επίσης, τη βέλτιστη λύση συνολικά την εντοπίζει μόνο ο Bin Balancer. Αντιθέτως ο default scheduler βρίσκει την καλύτερη λύση βάσει των δεδομένων τωρινών συνθηκών και ο NetMARKS την καλύτερη βάσει της αρχικής κατανομής που του δίνεται. Ο combined scheduler προσπαθεί να βρει την καλύτερη λύση που θα ικανοποίει τις δεδομένες συνθήκες και το ποσοστό εισόδου.

Τέλος, για τον default scheduler οι μετρικές παράγονται μέσω του Kubernetes API και δεν χρησιμοποιεί αρχικά υπολογισμένες μετρικές για την εκτέλεσή του. Αντιθέτως για τον NetMARKS οι μετρικές που χρησιμοποιεί προέρχονται από μετρήσεις σε μια τυχαία κατανομή με χρήση του Istio και του Prometheus. Ομοίως ο Bin Balancer χρησιμοποίει το OpenCost και το Prometheus για να εξάγει μετρικές για τα κόστη των pods από μια τυχαία αρχική κατανομή ώστε έπειτα να μπορεί να εκτελέσει τον αλγόριθμό του.

Κεφάλαιο 6: Σχεδιασμός συστήματος

6.1 Δομή Cluster

Το cluster που χρησιμοποιήθηκε για τις δοκιμές των διάφορων schedulers στο έργο αυτό είναι ένα μικρό cluster το οποίο αποτελείται από τέσσερις εικονικές μηχανές - VMs. Τα τέσσερα μηχανήματα αυτά έχουν τις ίδιες προδιαγραφές και συγκεκριμένα:

- Υπολογιστική ισχύς: 2 V-CPUs
- Μνήμη: 4 GB RAM
- Αποθηκευτικός χώρος: 20 GB storage
- Λειτουργικό σύστημα: Linux Ubuntu 22.04

Οι εικονικές μηχανές που χρησιμοποιούνται βρίσκονται εντός ενός ιδιωτικού OpenStack cloud και συνεπώς η ακριβής τοπολογία τους δεν είναι φανερή.

Το cluster έχει υλοποιηθεί χρησιμοποιώντας το microk8s. Συγκεκριμένα το microk8s αποτελεί μια ελαφριά σε πόρους έκδοση του Kubernetes, που παρέχει τις κύριες δυνατότητες του Kubernetes και μπορεί να χρησιμοποιηθεί ακόμα και στην παραγωγή [40] [41]. Έτσι το σύστημα του Kubernetes δεν καταναλώνει πολύτιμους πόρους από τα ήδη περιορισμένα σε δυνατότητες μηχανήματα και επιτρέπει την επίτευξη του σκοπού του έργου αυτού.

Ως control-plane του cluster έχει επιλεχθεί μόνο ένα node το οποίο περιέχει και τα απαραίτητα συστατικά για το Istio, το Kiali, το OpenCost και το Prometheus που θα χρειαστούν για την ορθή λειτουργία των αλγόριθμων και της ανάλυσής τους. Επίσης το node αυτό, λόγω του ήδη υψηλού φόρτου εργασίας του, θα αποκλειστεί από τις εκτελέσεις των αλγόριθμων δρομολόγησης των pod της εφαρμογής. Επίσης σε όλα τα nodes υπάρχουν τα κατάλληλα στοιχεία για τον Prometheus node exporter και την άντληση των απαραίτητων μετρικών.

6.2 Η εφαρμογή του Robot Shop

Για τους σκοπούς του έργου αυτού, η εφαρμογή που χρησιμοποιήθηκε είναι το Stan’s Robot Shop. Συγκεκριμένα το Robot Shop αποτελεί μια δειγματική εφαρμογή με αρχιτεκτονική microservices που μπορεί να χρησιμοποιηθεί για τη μελέτη μετρικών σε αυτή και τον τρόπο ενορχήστρωσής των microservices της [42] . Η εφαρμογή έχει χτιστεί χρησιμοποιώντας τις εξής τεχνολογίες:

- NodeJS [43]
- Java [44]
- Python [45]
- Golang [46]
- PHP [47]
- MongoDB [48]
- Redis [49]
- MySQL [50]
- RabbitMQ [51]
- Nginx [52]
- AngularJS [53]

Η εφαρμογή έχει επίσης χτιστεί γύρω από την υποστήριξη του Instana, μια εφαρμογή για την παρακολούθηση μετρικών, η οποία ωστόσο στο έργο αυτό θα αντικατασταθεί με τη χρήση του Prometheus, Istio και OpenCost. Η εγκατάσταση του Robot Shop σε ένα Kubernetes cluster γίνεται μέσω του Helm chart (η λειτουργία τους που περιγράφθηκε στο κεφάλαιο 3.4) που δίνεται από το GitHub repository της εφαρμογής, ενώ επίσης παρέχεται υποστήριξη και για Istio αν και απαιτούνται ορισμένες αλλαγές που θα συζητηθούν σε επόμενο κεφάλαιο. Η εφαρμογή είναι επίσης ανοιχτού κώδικα και όλες οι λειτουργίες του είναι διαθέσιμες για χρήση και επεξεργασία στο δημόσιο reposiroty GitHub

Το Robot Shop δημιουργεί μία ιστοσελίδα στην οποία ο χρήστης μπορεί να κάνει εγγραφή/σύνδεση με δικά του στοιχεία, να επιλέξει προϊόντα από τις διάφορες κατηγορίες, να τα βαθμολογήσει και να τα παραγγείλει στην τοποθεσία που τον ενδιαφέρει. Οι δυνατότητες αυτές είναι εφικτές με τη συνεργασία όλων των services (με την έννοια της αρχιτεκτονικής microservices και όχι του Kubernetes) της εφαρμογής και το κάθε service εκτελεί τη δική του ξεχωριστή λειτουργία. Τα services αυτά είναι τα shipping, web, payment, cart, catalogue, ratings, user, mysql, rabbitmq, mongodb και redis. Το κάθε service που αναφέρθηκε έχει δημιουργηθεί με διαφορετική τεχνολογία και έχει διαφορετικές απαιτήσεις σε πόρους τόσο δικτύου όσο και υπολογιστικών δυνατοτήτων (RAM και CPU).

Εικόνα 17.Η αρχική σελίδα του Stan's Robot Shop

Όσον αναφορά την παραγωγή κίνησης στην εφαρμογή, διατίθεται μια ξεχωριστή εφαρμογή γεννήτρια κίνησης. Η εφαρμογή αυτή είναι χτισμένη σε Python με τη βιβλιοθήκη Locust και μπορεί να χρησιμοποιηθεί μέσω ενός Docker image (περιγράφεται στο κεφάλαιο 2.1.3) με ορισμένα environment arguments που μπορούν να οριστούν από τον χρήστη. Για Kubernetes μάλιστα υπάρχει yaml αρχείο το οποίο δημιουργεί ένα deployment στο cluster και παράγει κίνηση στην κύρια εφαρμογή. Η εφαρμογή αυτή ουσιαστικά κάνει requests στο web microservice που είναι εκτεθειμένο εκτός cluster και τα requests που εκτελεί απαιτούν την χρήση όλων των microservices για να παράξουν αποτελέσματα.

Αρχικά η εφαρμογή υποστηρίζει το service mesh του Istio, ωστόσο δεν αναγνωρίζει τα microservices ως apps του Istio. Για την αναγνώριση τους, χρειάζεται τα services και τα deployments που χρησιμοποιεί το Kubernetes για τα microservices αυτά να περιέχουν τα labels app, που η τιμή του είναι η αντίστοιχη τιμή που δόθηκε ως name στα metadata, και version με τιμή που επιλέχθηκε να είναι v1. Όσον αναφορά τα deployments τα labels αυτά πρέπει να προστεθούν και στα πεδία matchLabels και

template metadata labels Επίσης στα services χρειάζεται να τεθεί ως selector app το αντίστοιχο όνομα που υπάρχει στα metadata. Οι αλλαγές αυτές γίνονται στα template yaml του Helm Chart και παρουσιάζονται δειγματικά παρακάτω για το cart deployment και service template.

Για το cart deployment οι αλλαγές είναι:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: cart
 labels:
  service: cart
  app: cart
  version: v1
spec:
 replicas: 1
 selector:
  matchLabels:
   service: cart
   app: cart
   version: v1
 template:
  metadata:
   labels:
    service: cart
    app: cart
    version: v1
  spec:
…
```

Για το cart service είναι:

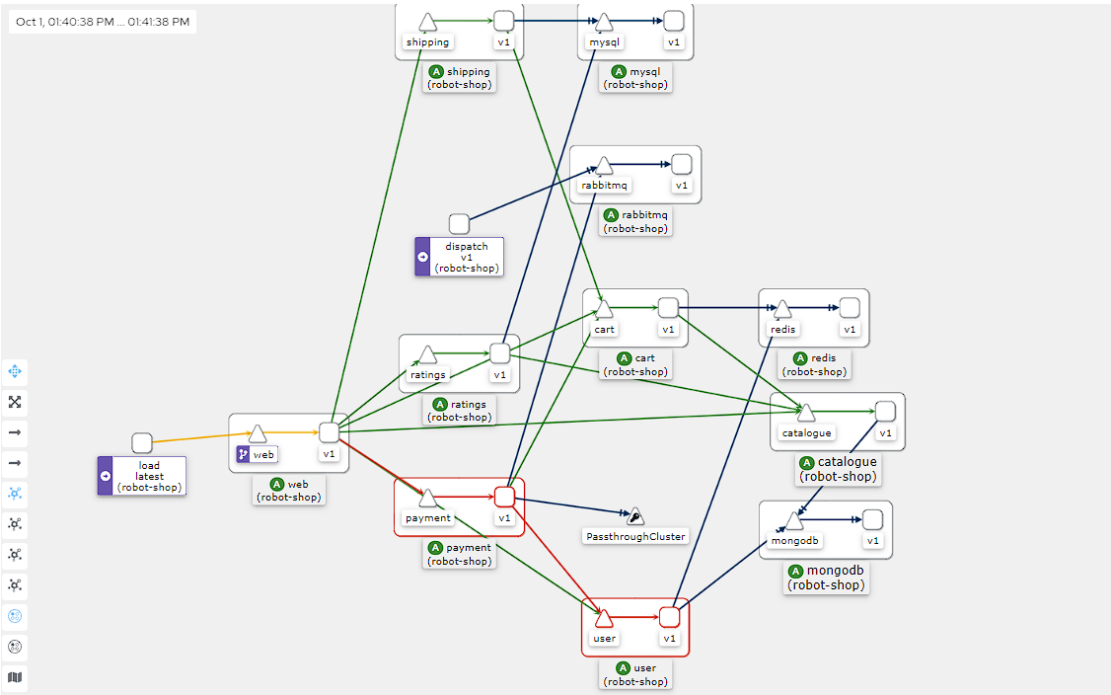
```
apiVersion: v1
kind: Service
metadata:
 name: cart
 labels:
  service: cart
  app: cart
spec:
 ports:
 - name: http
  port: 8080
  targetPort: 8080
 selector:
  service: cart
  app: cart
```

Επίσης αντίστοιχες αλλαγές με εκείνες στα deployment templates έγιναν και στο redis stateful set.

Επίσης για να αξιοποιηθούν πλήρως όλες οι λειτουργίες του Istio, θα πρέπει να ενεργοποιηθεί το sidecar injection, δηλαδή η αυτόματη διαδικασία προσθήκης sidecar containers (βλέπε κεφάλαιο 2.2.10) σε pods, στο namespace του cluster που θα εγκατασταθεί το Istio. Συγκεκριμένα αυτό μπορεί να επιτευχθεί θέτοντας ως label στο namespace το istio-injection με τιμή enabled [54]. Αυτό μπορεί να γίνει μέσω της εξής εντολής kubectl για π.χ. το namespace default:

```
kubectl label namespace default istio-injection=enabled –overwrite.
```

Έτσι πλέον μπορεί να γίνει sidecar injection και εντοπίζονται όλα τα apps του Istio. Έπειτα από τις αλλαγές αυτές και δημιουργώντας κίνηση στο cluster μέσω του load deployment, γίνεται φανερή η δομή της εφαρμογής και της επικοινωνίας των συστατικών της στο cluster μέσω του Kiali. Το γράφημα το οποίο παράγεται είναι το εξής:



Εικόνα 18.Η δομή της εφαρμογής και η επικοινωνία των συστατικών από το γράφημα Kiali

Στο παραπάνω γράφημα παρουσιάζονται τα Apps του Istio και η κίνηση που παρατηρείται ανάμεσα τους όταν υπάρχει δραστηριότητα. Η κίνηση αυτή φανερώνει τον τρόπο που συνδέονται τα microservices καθώς για τα αιτήματα που γίνονται από το load στο app web χρειάζονται δεδομένα και από τα υπόλοιπα microservices. Τα microservices αυτά επίσης για να παράγουν το αποτέλεσμα τους εξαρτώνται από προηγούμενα microservices που πρέπει να τους έχουν στείλει τα απαραίτητα δεδομένα. Έτσι παρατηρείται μια εξάρτηση των microservices έτσι ώστε να μπορέσουν να προσφέρουν το επιθυμητό αποτέλεσμα στο τέλος.

6.3 Εγκατάσταση επεκτάσεων Kubernetes

Με σκοπό την παραγωγή των σωστών μετρικών και την ομαλή εκτέλεση των διαδικασιών δοκιμών στο cluster, θα πρέπει να εγκατασταθούν στο cluster οι κατάλληλες εφαρμογές-εργαλεία. Οι κύριες εφαρμογές που χρειάζονται είναι το Prometheus, το OpenCost και το Istio. Ωστόσο η εγκατάσταση θα πρέπει να γίνει με τρόπο τέτοιον έτσι ώστε να μπορούν το OpenCost και το Istio να αντλήσουν μετρικές από το Prometheus.

Αρχικά λοιπόν στο cluster θα γίνει η εγκατάσταση του Prometheus. Η εγκατάσταση μπορεί να επιτευχθεί μέσω του helm install και συγκεκριμένα μέσω της εξής εντολής:

```
helm install my-prometheus --repo https://prometheus-community.github.io/helm-charts prometheus \
  --namespace prometheus --create-namespace \
  --set prometheus-pushgateway.enabled=false \
  --set alertmanager.enabled=false \
  -f https://raw.githubusercontent.com/opencost/opencost/develop/kubernetes/prometheus/extraScrapeConfigs.yaml
```

Εικόνα 19.Εντολή helm install Prometheus με προσαρμοσμένες ρυθμίσεις

Μέσω της εντολής αυτής, γίνεται εγκατάσταση του επίσημου helm chart του Prometheus στο νέο namespace του cluster Prometheus. Επίσης δεν ενεργοποιείται το pushgateway καθώς δεν παρατηρούνται στην εφαρμογή εργασίες μικρής διάρκειας. Επίσης απενεργοποιείται ο Alertmanager επειδή δεν είναι επιθυμητή η διαχείριση των warnings που παράγονται. Πρέπει να αναφερθεί ότι τα Prometheus-node-exporters δημιουργούνται κατά την εγκατάσταση και δεν χρειάζονται περαιτέρω βήματα υλοποίησης για αυτά. Τέλος προστίθενται στο τελικό yaml της εγκατάστασης ένα κομμάτι yaml κώδικα το οποίο δηλώνει τα ScrapeConfigs που θα χρησιμοποιήσει το OpenCost για να λάβει τις μετρικές. Οι ιδιότητες του Prometheus που αναφέρονται στην πάραγραφο αυτή περιγράφονται στο κεφάλαιο 3.2.1

Έπειτα από την εγκατάσταση του Prometheus ακολουθεί η εγκατάσταση του Istio και προσαρμογή του cluster στο Istio Mesh. Αρχικά για να γίνει η εγκατάσταση των συστατικών του Istio θα χρησιμοποιηθεί το εργαλείο istioctl [55]. Έτσι με την παρακάτω εντολή θα γίνει λήψη της τελευταίας έκδοσης του Istio από την επίσημη ιστοσελίδα:

```
curl -L https://istio.io/downloadIstio | sh -
```

Επομένως δημιουργείται ένας φάκελος που περιέχει τα απαραίτητα για την εγκατάσταση. Πηγαίνοντας, μέσα στον φάκελο είναι εφικτό να προστεθεί το αρχείο istioctl στο path μέσω της εντολής:

```
export PATH=$PWD/bin:$PATH
```

Πλέον μπορεί να γίνει χρήση του istioctl και η εγκατάσταση του Istio γίνεται με την εξής εντολή:

```
istioctl install
```

Η εγκατάσταση έγινε έτσι ώστε να επιλεχθεί το default προφίλ του Istio καθώς δεν κρίνεται αναγκαία η επιλογή κάποιου άλλου προφίλ. Με την εγκατάσταση αυτή επίσης το Istio εντοπίζει το Prometheus και προσθέτει τα απαραίτητα ScrapeConfigs στο yaml του με τα οποία θα μπορεί να εξάγει μετρικές. Επίσης θα ενεργοποιηθεί το istio-injection για το namespace της εφαρμογής, στην περίπτωση του έργου το namespace robot-shop μέσω της εντολής:

```
kubectl label namespace robot-shop istio-injection=enabled
```

Επόμενο και τελευταίο βήμα αποτελεί η εγκατάσταση του OpenCost η οποία μπορεί να γίνει μέσω kubectl με το yaml αρχείο που δίνεται και περιγράφει τα συστατικά του [56]. Πριν την εγκατάσταση του OpenCost όμως θα χρειαστεί να μεταβληθεί το αρχείο αυτό έτσι ώστε να μπορεί να συνδεθεί στον υπάρχον Prometheus server. Συγκεκριμένα στις μεταβλητές περιβάλλοντος του pod OpenCost δίνεται η μεταβλητή PROMETHEUS_SERVER_ENDPOINT η οποία δηλώνει το service της Prometheus εφαρμογής που πρέπει να κοιτάει το OpenCost, όπως φαίνεται παρακάτω:

```
env:
 - name: PROMETHEUS_SERVER_ENDPOINT
   value: "http://my-prometheus-server.prometheus.svc" # The endpoint should
have the form http://<service-name>.<namespace-name>.svc
 - name: CLOUD_PROVIDER_API_KEY
   value: "AIzaSyD29bGxmHAVEOBYtgd8sYM2gM2ekfxQX4U" # The GCP Pricing API
requires a key. This is supplied just for evaluation.
 - name: CLUSTER_ID
   value: "cluster-one" # Default cluster ID to use if cluster_id is not set
in Prometheus metrics.
```

Η τιμή που θα λάβει εξαρτάται από το πως έχουν οριστεί κατά την εγκατάσταση το όνομα του service και του namespace που βρίσκεται το Prometheus. Στην περίπτωση που εξετάζεται όμως δεν χρειάστηκε κάποια αλλαγή. Επομένως η εγκατάσταση των εφαρμογών ολοκληρώνεται και είναι πλέον δυνατόν να ξεκινήσουν οι δοκιμές των Schedulers.

6.4 Custom διαδικασία Scheduling

Όπως περιγράφηκε σε προηγούμενο κεφάλαιο, εάν ο σχεδιαστής επιθυμεί να τροποποιήσει τον scheduler και τις λειτουργίες του, μπορεί να δημιουργήσει API plugins που μπορούν να επέμβουν στη ροή του Scheduling Framework σε αρκετά σημεία τόσο στον κύκλο δρομολόγησης όσο και στον κύκλο δέσμευσης. Μάλιστα προσαρμόζοντας τα κατάλληλα Plugins μέσω προσθήκης ή ακόμα και αφαίρεσης ιδιοτήτων, να δημιουργήσει τον επιθυμητό scheduler.

Για τη διευκόλυνση του έργου ωστόσο επιλέχθηκε να μην τροποποιηθούν τα plugins αλλά η δρομολόγηση να γίνει με έμμεσο τρόπο αξιοποιώντας τις δυνατότητες του Kubernetes API σε συνδυασμό με την Python [57]. Ουσιαστικά με τον τρόπο αυτό, παραλείπεται ο κύκλος δρομολόγησης και αντικαθίσταται από Python scripts που στο τέλος παράγουν την κατανομή των Pods στα nodes. Έπειτα τα nodes αυτά επιστρέφουν στον κύκλο bind.

Μια σημαντική παρατήρηση είναι ότι και τα τρία Nodes που θα εξεταστούν από το έργο όσον αναφορά την δρομολόγηση (το control-plane node δεν λαμβάνει μέρος για σκοπό των δοκιμών) θεωρούνται ικανά (feasible) σε κάθε περίπτωση. Αυτή η υπόθεση γίνεται καθώς κάθε Node του cluster μπορεί να καλύψει τις ανάγκες κάθε Pod της εφαρμογής που εξετάζεται. Επίσης θεωρούνται ικανά για scheduling και τα Pods στο cluster μέσω επιβεβαίωσης από προηγούμενες δοκιμές. Έτσι παραλείπονται τα plugins Prefilter, filter, PostFilter.

Όσον αναφορά τα plugins του PreScore, Score και NormalizeScore οι λειτουργίες τους εμπεριέχονται εντός του Python script που υλοποιεί τον scheduler του έργου που εξετάζεται. Συγκεκριμένα, το Python script δέχεται ως είσοδο δεδομένα που περιγράφουν μια ιδιότητα του pod, π.χ. στον NetMARKS την μετρική ροής με κάποιο άλλο app-pod. Έπειτα εκτελείται ο αλγόριθμος που για κάθε pod, θα βαθμολογήσει τα nodes και θα επιλέξει αυτό με το καλύτερο score. Η κανονικοποιήση μπορεί να συμβεί εντός του αλγορίθμου αλλά λόγω του σχεδιασμού των schedulers δεν γίνεται. Στο τέλος του script έχει παραχθεί ως έξοδος ένα αρχείο json που δείχνει για κάθε pod της εφαρμογής, σε ποιο node επιλέχθηκε να γίνει schedule. Η κατανομή αυτή αποτελεί το αποτέλεσμα του κύκλου δέσμευσης για όλα τα Pods και επομένως έτσι ολοκληρώνεται.

Για την ολοκληρωμένη διαδικασία του scheduling επίσης θα χρειαστεί μια μεταβολή στα template αρχεία yaml που περιγράφουν τα pods. Συγκεκριμένα θα πρέπει

να αποτραπεί η χρήση του default scheduler και η χρήση ενός άλλου, δηλώνοντας το όνομα του scheduler που επιθυμείται να χρησιμοποιηθεί [58]. Ωστόσο θα χρησιμοποιηθεί ένα όνομα από scheduler που δεν υπάρχει έτσι ώστε να μην μπορεί το Kubernetes να τον εντοπίσει και να βρίσκονται τα Pods διαρκώς σε κατάσταση pending. Για παράδειγμα στο τμήμα του cart-deployment.yaml όπως φαίνεται παρακάτω, δηλώνεται στο πεδίο schedulerName ως όνομα το NetMARKS, ωστόσο ο scheduler αυτός δεν έχει δηλωθεί εντός του Kubernetes:

```
…
template:
  metadata:
   labels:
    service: cart
    app: cart
    version: v1
  spec:
   schedulerName: NetMARKS
   {{ if .Values.psp.enabled }}
   serviceAccountName: robot-shop
   {{ end }}
   containers:
   - name: cart
    image: {{ .Values.image.repo }}/rs-cart:{{ .Values.image.version }}
…
```

Για να αλλάξουν κατάσταση τα Pods από pending και να δρομολογηθούν, θα χρησιμοποιηθεί ένα Python script το οποίο θα εντοπίζει τα Pods σε pending κατάσταση και θα τα δεσμεύει στο node που τους αντιστοιχεί από την κατανομή που δίνεται ως είσοδο στο script. Ο τρόπος με τον οποίο γίνεται ο εντοπισμός των pending pods είναι μέσω του παρακάτω for και if δήλωσης:

```
  w = watch.Watch()

  for event in w.stream(v1.list_namespaced_pod, "robot-shop"):
    if event['object'].status.phase == "Pending" and
event['object'].spec.scheduler_name == scheduler_name:
```

Αναλυτικά, ορίζεται ένα αντικείμενο watch το οποίο μέσω της μεθόδου του stream, επιβλέπει συνεχώς τα αποτελέσματα μιας συνάρτησης και παράγει ένα generator βάσει αυτού. Στη περίπτωση αυτή η συνάρτηση είναι η v1.list_namespaced_pod η οποία επιστρέφει τα pods και πληροφορίες σχετικά με αυτά και ως είσοδο δέχεται το namespace όπου θα κοιτάξει τα pods. Τα pods που παράγονται αποτελούν αντικείμενα που περιέχουν τις περισσότερες ιδιότητες των Pods για την χρονική στιγμή που εντοπίστηκαν. Έπειτα, με την if συνθήκη από τα Pods που εντοπίστηκαν ελέγχονται μόνο εκείνα που βρίσκονται σε κατάσταση pending και ως scheduler_name έχουν το επιθυμητό, π.χ. NetMARKS. Το scheduler_name που ελέγχεται αντιστοιχεί σε αυτό που ορίστηκε στα templated yaml. Αξίζει να σημειωθεί ότι το for-loop θα εκτελείται επ’ αόριστον και επομένως θα πρέπει να σταματήσει είτε χειροκίνητα, είτε έπειτα από ορισμένο χρονικό διάστημα όπως και επιλέχθηκε.

Τα pods αυτά που εντοπίστηκαν λοιπόν, είναι αυτά που θα πρέπει να γίνει η δέσμευση στο αντίστοιχο node που ορίζεται από την κατανομή που δίνεται ως είσοδος. Εφόσον βρεθεί το node που αντιστοιχεί στο pod, θα εκτελεστεί η παρακάτω μέθοδος που υλοποιεί την διαδικασία του binding:

```
def binder(name, node, namespace="robot-shop"):

  target=client.V1ObjectReference()
  target.kind="Node"
  target.apiVersion="v1"
  target.name= node

  meta=client.V1ObjectMeta()
  meta.name=name

  body=client.V1Binding(target=target)
  body.target=target
  body.metadata=meta

  return v1.create_namespaced_pod_binding(name, namespace, body)
```

Αναλυτικά, ως είσοδο στη μέθοδο δίνεται το όνομα του pod, το όνομα του node και το namespace. Έπειτα ορίζεται το αντικείμενο target που ουσιαστικά περιγράφει το node και με αυτό το αντικείμενο δημιουργείται μια αναφορά στο πραγματικό αντικείμενο του Kubernetes επιτρέποντας τη μεταβολή του. Έπειτα ορίζεται αντικείμενο σχετικά με τα μεταδομένα του Pod, έτσι ώστε να μπορεί να εντοπιστεί το Pod σε επόμενο βήμα. Ύστερα, ορίζεται το body που αποτελεί το αντικείμενο που θα εκτελέσει τη διαδικασία δέσμευσης και ως ορίσματα θα δεχτεί το target και το meta. Τέλος, η διαδικασία της δέσμευσης θα εκτελεστεί μέσω της μεθόδου v1.create_namespaced_pod_binding, η οποία αποστέλλει αίτημα δέσμευσης στο Kubernetes API. Έτσι, τελικά θα δεσμευτούν όλα τα Pods βάσει της κατανομής που δίνεται ως είσοδο και ολοκληρώνεται και η διαδικασία της δέσμευσης.

Η συνολική διαδικασία του scheduling με τον συγκεκριμένο τρόπο παρόλο που απέχει από τον τρόπο που εκτελείται με το scheduling framework , αποτελεί μια απλή εναλλακτική για την αλγοριθμική μελέτη του έργου αυτού. Δεν μπορεί ωστόσο να συγκριθεί σε ταχύτητα και επεκτασιμότητα με την ανάπτυξη plugins στο Scheduling framework. Επομένως, με σκοπό την μελλοντική χρήση των αλγορίθμων σε πραγματικό περιβάλλον κρίνεται αναγκαία η ανάπτυξη των κατάλληλων plugins για την υποστήριξη τους και την καλύτερη απόδοσή τους.

Κεφάλαιο 7: Πειραματικά αποτελέσματα και ανάλυση

7.1 Μετρικές Μελέτης

7.1.1 Εισαγωγή στις μετρικές μελέτης

Στο συγκεκριμένο έργο τα αποτελέσματα των schedulers που δοκιμάζονται θα συγκριθούν βάσει 2 μετρικών. Οι μετρικές αυτές έχουν προκύψει από μετρήσεις πραγματικού χρόνου στο cluster και έχουν συντεθεί με τρόπο τέτοιον έτσι ώστε να προσφέρουν ένα απόλυτο και κατανοητό αποτέλεσμα για τη σύγκριση των schedulers.

7.1.2 Average Response time (ms)

Η μετρική αυτή περιγράφει τον μέσο χρόνο απόκρισης ενός application του Istio σε όλα τα αιτήματα που λαμβάνει εκφρασμένο σε milliseconds (ms). Για την ανάλυση χρησιμοποιούνται τιμές που παράγει το Istio και τις εξάγει το Prometheus. Έτσι προκύπτει το παρακάτω query για τον υπολογισμό του μέσου χρόνου απόκρισης:

```
sum(rate(istio_request_duration_milliseconds_sum{reporter='destination',
destination_service='…'}[2m])) /
sum(rate(istio_request_duration_milliseconds_count{reporter='destination',
destination_service='…'}[2m]))
```

Η ανάλυση του query είναι η εξής: Αρχικά, γίνεται εξαγωγή των δεδομένων του Istio μέσω των:

- istio_request_duration_milliseconds_sum: Περιγράφει το άθροισμα των bytes από τα requests στο πέρασμα του χρόνου
- istio_request_duration_milliseconds_count: Περιγράφει πόσα requests έχουν συμβεί συνολικά έως μια χρονική στιγμή

Ως ορίσματα δέχονται το reporter, που δηλώνει από ποιο component γίνεται η αναφορά των μετρικών και στη συγκεκριμένη περίπτωση από έναν διαμεσολαβητή server του Istio, και το service που αποτελεί το service του application που αναλύεται. Το αποτέλεσμα των συναρτήσεων αυτών είναι ένα σύνολο από χρονικές παραστάσεις οι οποίες ικανοποιούν τα ορίσματα που τους δόθηκαν. Ουσιαστικά αποτελούν όλες οι χρονικές παραστάσεις που περιγράφουν την επικοινωνία από ένα source service στο destination service που ορίστηκε.

Έπειτα για τις μετρικές αυτές εφαρμόζεται ο μετασχηματισμός rate που υπολογίζει την ανά δευτερόλεπτο αύξηση της μετρικής για το χρονικό διάστημα που του δίνεται και στη συγκεκριμένη περίπτωση είναι τα 2 λεπτά. Με τον μετασχηματισμό αυτό υπολογίζεται σε κάθε χρονικό διάστημα πόσα bytes λήφθηκαν από το προηγούμενο και πόσα νέα responses υπήρξαν αντίστοιχα για τις δύο μετρικές. Τέλος εφαρμόζεται η συνάρτηση sum που αθροίζει τα αποτελέσματα των μετρικών παραστάσεων σε μια χρονική παράσταση και διαιρούνται οι χρονικές παραστάσεις αυτές.

Για τον υπολογισμό του μέσου response time, εκτελείται το query για ένα συγκεκριμένο χρονικό διάστημα, το οποίο επιλέχθηκε να είναι 20 λεπτά όσο και η κάθε δοκιμή που περιγράφεται στο κεφάλαιο των δοκιμών. Επειδή ορισμένες φορές η

μέτρηση αποτυγχάνει, αφαιρούνται οι μη αριθμητικές τιμές π.χ. NaN και έπειτα υπολογίζεται ο μέσος όρος καταλήγοντας στη μέση χρονική απόκριση του application.

Η μετρική αυτή επιλέχθηκε καθώς περιγράφει το αποτέλεσμα που επιθυμεί να βελτιώσει ο NetMARKS με την δρομολόγηση του και μπορεί να συγκριθεί με τον αλγόριθμο Bin Balancer με ευκολία. Επίσης αποτελεί μια μετρική η οποία είναι εύκολα αντιληπτή η σημασία της και η επίδραση της στην απόδοση ενός συστήματος. Μάλιστα και ο υπολογισμός της είναι εύκολος και μπορεί να υπολογιστεί γρήγορα.

7.1.3 Cost Balance

Η μετρική αυτή αποτελεί την ισορροπία του κόστους ανάμεσα στα nodesΑρχικά ως κόστος ορίζεται το αθροιστικό κόστος του ποσοστού της CPU που χρησιμοποιείται καιτου ποσοστού της μνήμης που χρησιμοποιείται όπως υπολογίζεται από το OpenCost. Αξίζει να σημειωθεί ότι για τον σκοπό του έργου οι χρεώσεις που χρησιμοποιήθηκαν αποτελούν τις default χρεώσεις που ορίζει το OpenCost και βασίζονται στον πάροχο GCP us-central1. Η επιλογή αυτή έγινε καθώς τα πειράματα εκτελέστηκαν σε VMs ενός private cloud και συνεπώς δεν υπήρχε ένδειξη του κόστους χρήσης των μηχανημάτων και των πόρων. Η ισορροπία του κόστους ορίζεται με τον εξής τρόπο:

- Για κάθε δοκιμή υπολογίζεται το συνολικό κόστος κάθε node στο cluster.
- Από τα κόστη αυτά εντοπίζεται το μέγιστο και το ελάχιστο και αφαιρούνται. Έτσι προκύπτει η ισορροπία για το cluster αυτό. Ο τύπος αυτός χρησιμοποιείται εφόσον στο cluster που γίνονται οι δοκιμές του έργου εξετάζουμε 3 nodes. Για παραπάνω nodes ένας καλύτερος τύπος αποτελεί ο εξής:

$$balance = \frac{\max \ - \min}{avg}$$

  Ο τύπος αυτός θα μπορούσε να χρησιμοποιηθεί και για τα 3 nodes. Ωστόσο το μέσο κόστος των nodes για την εφαρμογή παραμένει το ίδιο. Αυτό οφείλεται στο γεγονός ότι οι δοκιμές επαναλαμβάνονται για τον ίδιο χρόνο και με την ίδια διαδικασία. Έτσι μπορεί να αμεληθεί από τον τύπο.
- Τέλος για όλες τις δοκιμές υπολογίζεται ο μέσος όρος από τα παραπάνω αποτελέσματα και προκύπτει η μέση ισορροπία κόστους.

Η μετρική αυτή επιλέχθηκε καθώς περιγράφει ακριβώς την επιθυμητή βελτίωση που θέλει να προσφέρει ο Bin Balancer αλγόριθμος δηλαδή να ισορροπήσει τα pods βάσει τα κόστη, και συνεπώς του πόρους, στα nodes. Όσο μικρότερες τιμές λαμβάνει η μετρική αυτή, τόσο μεγαλύτερη ισορροπία κόστους παρατηρείται στο cluster. Επίσης ο υπολογισμός της είναι πολύ γρήγορος και εύκολος στην υλοποίηση.

7.2 Διαδικασία δοκιμών

7.2.1 Default Scheduler

Αρχικά εκτελέστηκαν δοκιμές για την παραγωγή αποτελεσμάτων από τον default scheduler του Kubernetes. Επειδή το load deployment έχει υλοποιηθεί με τέτοιον τρόπο έτσι ώστε να κάνει restart κάθε περίπου 20 λεπτά και για να αποφευχθούν τυχόν προβλήματα όπως επανεκκινήσεις των pods, δεν θα γίνει μια ενιαία δοκιμή για μεγάλο

χρονικό διαστήματα. Αντιθέτως θα γίνουν πολλές μικρές δοκιμές στις ίδιες συνθήκες και θα υπολογιστούν τα αποτελέσματα για κάθε μία. Τα τελικά αποτελέσματα θα αποτελούν τον μέσο όρο των συνολικών αποτελεσμάτων. Η διαδικασία που ακολουθήθηκε για την παραγωγή αποτελεσμάτων είναι η εξής:

1. Εκτελούνται με τη βοήθεια ενός bash script 24 tests τα οποία το καθένα εκτελείται για 20 λεπτά. Συγκεκριμένα στην αρχή του κάθε τεστ εκτελείται helm install της εφαρμογής και ο kube-scheduler επιλέγει που θα βάλει το κάθε pod. Έχει τεθεί taint στο node του control plane έτσι ώστε τα pods να κατανεμηθούν στα υπόλοιπα 3 nodes.

2. Γίνεται deploy το load deployment που παράγει την κίνηση μέσα στο cluster. Στο load έχει τεθεί toleration και θα δρομολογηθεί στο control plane έτσι ώστε να μην επηρεάσει τα αποτελέσματα των αλγορίθμων. Η επιλογή αυτή έγινε καθώς σε πραγματικό σύστημα το pod αυτό δεν θα υπήρχε.

3. Έπειτα από τα 20 λεπτά που εκτελείται το load, με ένα Python script συλλέγονται οι μετρικές που θα οδηγήσουν στα αποτελέσματα της εφαρμογής, δηλαδή το response time σε ms του κάθε application καθώς και το κόστος του κάθε node, μέσω Prometheus api και OpenCost api και αποθηκεύονται σε ένα CSV.

4. Στο τέλος αφαιρούνται το Load και το helm installation έτσι ώστε να ξεκινήσει το επόμενο τεστ.

Η διαδικασία της δοκιμής φαίνεται και στο Bash script που χρησιμοποιήθηκε για την υλοποίησή της, το οποίο παρατίθεται παρακάτω:

```
cd ../
echo "Starting bash script for default scheduler testing"
echo "Time of start:"
date

# taint the control plane
kubectl taint nodes microk8s-tsiakag-control-plane-4gw82
key1=value1:NoSchedule

for i in {1..24}
do
  echo "Iteration $i started..."

  # helm install
  cd ./my-robot-shop/K8s/helm
  helm install robot-shop -n robot-shop .
  cd ../../..

  #wait to init and run load
  sleep 1m

  # apply load
  kubectl apply -f ./my-robot-shop/K8s/load-deployment.yaml

  sleep 20m

  # get results
  cd ./bench_tests
  python3 write_results.py original_results.csv
  cd ../
```

```
  # delete application
  kubectl delete deploy load
  helm uninstall robot-shop

  #wait for load to terminate
  sleep 20s
  echo "Iteration $1 completed!"
  echo "-------------------------------------------------"
done

# untaint the control plane
kubectl taint nodes microk8s-tsiakag-control-plane-4gw82
key1=value1:NoSchedule-

echo "Default Test Completed!"
echo "Ended at:"
date
```

Παρόμοια λογική με τη διαδικασία που περιγράφεται ακολουθούν και οι υπόλοιπες δοκιμές οπότε δεν θα δοθούν τα Bash script τους αλλά θα περιγράφουν οι διαφορές τους με το συγκεκριμένο.

7.2.2 NetMARKS Scheduler

Η διαδικασία για το NetMARKS διαφέρει λίγο εφόσον η κατανομή που θα παράγει ο NetMARKS εξαρτάται από την αρχική. Έτσι όπως αναφέρεται και στο paper θα εκτελεστούν δοκιμές για να βρεθεί η καλύτερη κατανομή και έπειτα θα ληφθούν τα αποτελέσματα από αυτή.

Για να εκτελεστεί ο αλγόριθμός όμως απαραίτητη προϋπόθεση είναι ο υπολογισμός των flows μεταξύ των apps. Για να γίνει αυτό εκτελείται ένα installation του Robot Shop με load για αρκετή ώρα και με τη βοήθεια του Prometheus api αποθηκεύονται τα flows σε ένα json που θα χρησιμοποιείται σε κάθε εκτέλεση του NetMARKS για αναφορά στις μετρικές. Το installation που χρησιμοποιήθηκε δεν περιείχε κάποια συγκεκριμένη κατανομή αλλά αυτή που κατέληξε ο kube scheduler το οποίο εκτελέστηκε μόνο του με load για περίπου 2-3 ώρες. Η δρομολόγηση έγινε με τον kube scheduler καθώς δεν είναι ζητούμενη η κατανομή των apps για να λάβουμε τα flows, άλλα το ποιόν της επικοινωνίας και τα bytes που ανταλλάσσουν τα ζεύγη των apps.Έπειτα θα εκτελεστεί η λογική των 24 τεστ που εκτελέσθηκε και για τον default scheduler με την εξής διαφορά στα βήματα:

1. Εφόσον έχει γίνει το helm install της εφαρμογής από το custom helm Chart που ορίζεται διαφορετικός αλγόριθμος δρομολόγησης από τον default, και όταν όλα τα Pods βρεθούν σε pending state έπειτα από ορισμένο χρονικό διάστημα, εκτελείται το Python script που θα υπολογίσει μια κατανομή pods βάσει NetMARKS. Η κατανομή αυτή αποθηκεύεται με την εξής μορφή σε json αρχείο:

```
{
  "dispatch": "microk8s-tsiakag-md-0-j2cfc",
  "mongodb": "microk8s-tsiakag-md-0-j2cfc",
  "redis": "microk8s-tsiakag-md-0-j2cfc",
  "user": "microk8s-tsiakag-md-0-66rwq",
  "catalogue": "microk8s-tsiakag-md-0-66rwq",
  "cart": "microk8s-tsiakag-md-0-66rwq",
```

```
  "payment": "microk8s-tsiakag-md-0-66rwq",
  "mysql": "microk8s-tsiakag-md-0-66rwq",
  "shipping": "microk8s-tsiakag-md-0-66rwq",
  "ratings": "microk8s-tsiakag-md-0-66rwq",
  "rabbitmq": "microk8s-tsiakag-md-0-66rwq",
  "web": "microk8s-tsiakag-md-0-66rwq"
}
```


2. Στο Python script αυτό παράγεται μια τυχαία αρχική κατανομή των apps στα nodes του cluster.
3. Εκτελείται ο αλγόριθμος NetMARKS για όλα τα apps της κατανομής αυτής και έτσι παράγεται μια νέα κατανομή την οποία και αποθηκεύουμε ως αρχείο json για το επόμενο βήμα.
4. Βάσει της νέας αυτής κατανομής κάνουμε schedule τα apps με το Python script που έχει σχεδιαστεί για custom scheduling και η διαδικασία συνεχίζει κανονικά.

Όταν τελειώσουν τα 24 τεστ αυτά ελέγχονται τα αποτελέσματα στο csv και επιλέγεται η κατανομή με τα καλύτερα αποτελέσματα. Τα καλύτερα αποτελέσματα επιλέχθηκαν βάσει του μέσου average χρόνου (η ισορροπία κόστους δεν είναι σημαντική για τον συγκεκριμένο αλγόριθμο οπότε μπορεί να αγνοηθεί για την επιλογή). Για την κατανομή αυτή εκτελούμε πάλι 24 δοκιμές με την παραπάνω λογική αλλά σε κάθε τεστ το schedule γίνεται βάσει της κατανομής που παράχθηκε.

Πρέπει να σημειωθεί ότι οι συνολικές πιθανές κατανομές είναι περισσότερες από τις 24 που παράχθηκαν με την παραπάνω διαδικασία και πιθανώς να διαφέρουν από εκτέλεση σε εκτέλεση. Ο αριθμός των δοκιμών αυτός επιλέχθηκε έτσι ώστε να δοθεί ένα δειγματικό αποτέλεσμα και να μην καταπονηθεί το ευαίσθητο σύστημα περαιτέρω.

7.2.3 Bin Balancer Scheduler

Η διαδικασία δοκιμών για τον Bin Balancer scheduler ακολουθεί παρόμοια βήματα με εκείνη του NetMARKS, ωστόσο, καθώς ο Bin Balancer βρίσκει άμεσα την καλύτερη κατανομή ως προς την ισορροπία κόστους δεν χρειάζεται να δοκιμαστούν πολλές κατανομές και να επιλεχθεί η καλύτερη όπως στον NetMARKS.

Αρχικά για την εκτέλεση του αλγόριθμου απαιτείται ο υπολογισμός του κόστους για τα pods που είναι προς δρομολόγηση, καθώς και το κόστος των υπόλοιπων pods που εκτελούνται στα nodes των δοκιμών έτσι ώστε να οριστούν τα αρχικά βάρη του αλγόριθμου. Για τον λόγο αυτό εκτελείται όπως και στον NetMARKS, ένα installation του Robot Shop με load για μεγάλο χρονικό διάστημα. Έπειτα, με τη χρήση του OpenCost api και μέσω ενός Python script, αποθηκεύονται τα κόστη σε ένα json αρχείο. Το installation που χρησιμοποιήθηκε δεν περιείχε κάποια συγκεκριμένη κατανομή αλλά αυτή που κατέληξε ο kube scheduler το οποίο εκτελέστηκε μόνο του με load για περίπου 2-3 ώρες. Τα κόστη που παράχθηκαν κατά την περίοδο αυτή, αποτελούν σημείο αναφοράς για τον αλγόριθμο. Επίσης η κατανομή της δοκιμής αυτής μπορεί να είναι τυχαία καθώς η κατανάλωση του κάθε pod δεν επηρεάζεται από την κατανομή.

Έπειτα από τον υπολογισμό του κόστους για κάθε pod εκτελείται, μέσω Python script με είσοδο το json με τα κόστη που υπολογίστηκαν, ο αλγόριθμος Bin Balancer. Η κατανομή των pods που παράγει αποθηκεύεται με την εξής μορφή σε ένα json αρχείο:

```
{
  "mysql": "microk8s-tsiakag-md-0-66rwq",
  "payment": "microk8s-tsiakag-md-0-66rwq",
  "ratings": "microk8s-tsiakag-md-0-66rwq",
  "catalogue": "microk8s-tsiakag-md-0-66rwq",
  "cart": "microk8s-tsiakag-md-0-66rwq",
  "shipping": "microk8s-tsiakag-md-0-9zd48",
  "user": "microk8s-tsiakag-md-0-9zd48",
  "rabbitmq": "microk8s-tsiakag-md-0-9zd48",
  "mongodb": "microk8s-tsiakag-md-0-j2cfc",
  "redis": "microk8s-tsiakag-md-0-j2cfc",
  "web": "microk8s-tsiakag-md-0-j2cfc",
  "dispatch": "microk8s-tsiakag-md-0-j2cfc"
}
```


Η κατανομή που παράγεται θα χρησιμοποιηθεί σε κάθε δοκιμή για τον υπολογισμό των αποτελεσμάτων. Με τον τρόπο αυτό οι 24 δοκιμές ακολουθούν την εξής διαδικασία:

1. Γίνεται το helm install της εφαρμογής από το custom helm Chart που ορίζεται διαφορετικός αλγόριθμος δρομολόγησης από τον default, και ακολουθεί μια χρονική καθυστέρηση έτσι ώστε όλα τα pods να βρεθούν σε κατάσταση pending.
2. Βάσει της κατανομής που υπολογίστηκε προηγουμένως γίνονται schedule τα pods με το Python script που έχει σχεδιαστεί για custom scheduling.
3. Γίνεται deploy το load deployment και παράγει κίνηση στην εφαρμογή.
4. Έπειτα από 20 λεπτά, εκτελείται το Python script που συλλέγει τις μετρικές σύγκρισης και τις αποθηκεύει σε ένα csv.
5. Τέλος, για την προετοιμασία της επόμενης δοκιμής γίνεται απεγκατάσταση της εφαρμογής και διαγραφή του deployment του load και εφόσον έχουν διαγραφεί όλα, μπορεί να ξεκινήσει η επομένη δοκιμή.

Έτσι παράγονται τα αποτελέσματα για τον scheduler και με τον μέσο όρο των αποτελεσμάτων μπορούν να υπολογιστούν οι τελικές μετρικές σύγκρισης.

7.2.4 Combined scheduler

Για τον combined scheduler θα ακολουθηθεί μια διαδικασία που συνδυάζει εκείνες του NetMARKS και του Bin Balancer scheduler.

Αρχικά, αναγκαίες κρίνονται οι μετρικές του NetMARKS και του Bin Balancer. Αυτές μπορούν να υπολογιστούν με τους τρόπους που αναφέρθηκε για κάθε scheduler. Ωστόσο, εάν έχουν εκτελεστεί προηγουμένως οι δοκιμές για τους schedulers, μπορούν να χρησιμοποιηθούν τα ήδη υπάρχοντα αρχεία json με τις μετρικές.

Επόμενο βήμα είναι ο υπολογισμός της κατανομής που παράγει ο Bin Balancer για να δοθεί ως αρχική κατανομή στον NetMARKS. Μέσω του Python script με τον αλγόριθμο Bin Balancer μπορεί να δημιουργηθεί η κατανομή αυτή, αλλά εάν οι δοκιμές για τον scheduler έχουν γίνει προηγουμένως και η εφαρμογή δεν έχει αλλάξει, μπορεί να χρησιμοποιηθεί η κατανομή εκείνης της δοκιμής.

Τώρα πρέπει να υπολογιστεί το όριο που θα προσφέρει τα επιθυμητά αποτελέσματα. Η επιλογή αυτή, θα γίνει βάσει της κατανομής που θα παραχθεί και θα έχει αποτελέσματα εξισορροπημένα και για τους δύο αλγόριθμους. Έτσι εκτελούνται 10 δοκιμές με διαφορετικά ποσοστά με τη διαδικασία που ακολούθησε ο NetMARKS για την εύρεση της καλύτερης κατανομής. Η μόνη διαφορά είναι ότι εκτελείται το

Python script με τον combined αλγόριθμό και ως είσοδο το ποσοστό που ορίζεται από την κάθε δοκιμή.

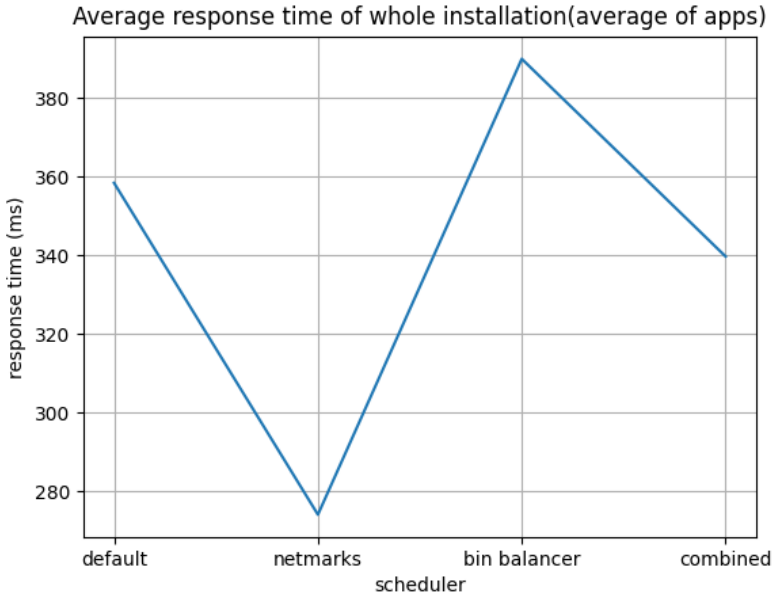
Έτσι προκύπτουν σε ένα csv 10 αποτελέσματα από 10 (πιθανώς) διαφορετικές κατανομές. Επιλέγεται η καλύτερη βάσει των μετρικών σύγκρισης και πλέον θα εκτελεστούν για το ποσοστό αυτό (και συνεπώς το όριο) 24 δοκιμές όπως και στους προηγούμενους schedulers.

Αξίζει να σημειωθεί ότι σε όλες τις δοκιμές εκτός του default scheduler, δεν έγινε taint το control plane καθώς οι schedulers σχεδιάστηκαν με τρόπο τέτοιον έτσι ώστε να τον αγνοούν. Ωστόσο αυτό έγινε μόνο για τις τωρινές δοκιμές και με μικρή αλλαγή να τον συμπεριλάβουν στους υπολογισμούς τους.

7.3 Ανάλυση Αποτελεσμάτων

Συγκεντρωτικά, τα αποτελέσματα των δοκιμών θα παρουσιαστούν σε γραφικές που τα συγκρίνουν με τους υπόλοιπους schedulers έτσι ώστε να γίνουν κατανοητά τα αποτελέσματά τους. Οι γραφικές αυτές θα περιγράφουν τον χρόνο απόκρισης κάθε application και της συνολικής εφαρμογής, καθώς και την ισορροπία κόστους στα nodes. Για τη δημιουργία των γραφικών χρησιμοποιήθηκε η βιβλιοθήκη Matplotlib της Python [59] με είσοδο τα αποτελέσματα των δοκιμών.

Όσον αφορά το μέσο response time του application τα αποτελέσματα είναι τα εξής:



Εικόνα 20.Γράφημα μέσης χρονικής απόκρισης της εφαρμογής για τους διάφορους schedulers

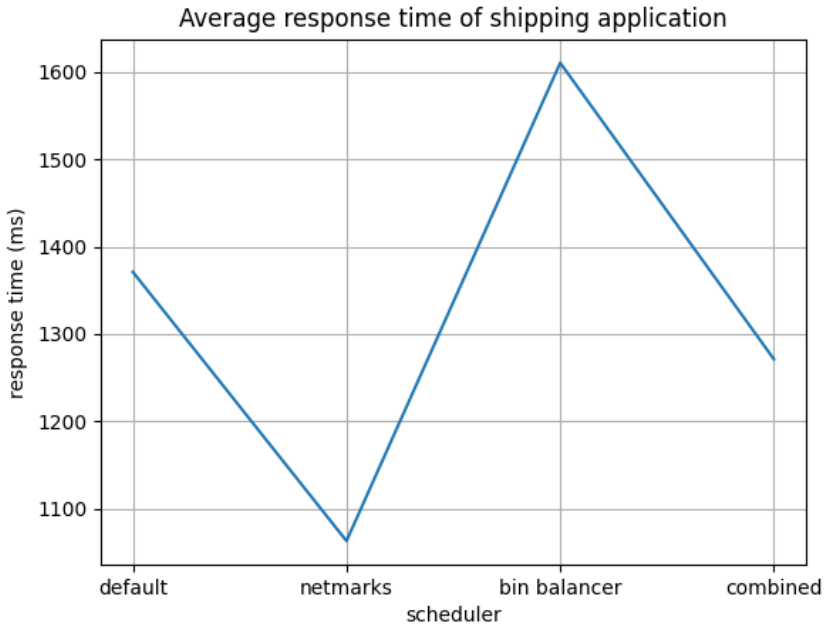
Παρατηρείται αρχικά ότι το μικρότερο response time παρουσιάζεται με τον αλγόριθμο NetMARKS και μάλιστα με μεγάλη διαφορά σε σχέση με τον default scheduler του Kubernetes. Μάλιστα παρατηρείται μείωση του response time κατά περίπου 31%, μια σημαντική βελτίωση για την εφαρμογή. Αντιθέτως, για τον scheduler Bin Balancer είναι φανερό ότι παρουσιάζεται το μεγαλύτερο μέσο response time σε σχέση με τους υπόλοιπους schedulers. Αυτό επιβεβαιώνει το γεγονός ότι ο αλγόριθμος δεν λαμβάνει υπόψη του το response time και τη βελτίωσή του, αλλά μόνο την

ισορροπία του κόστους. Τέλος παρατηρείται ότι ο combined scheduler πετυχαίνει μικρή βελτίωση έναντι του default scheduler (5% μείωση περίπου). Για τη σύγκριση του μέσου response time μεταξύ του κάθε scheduler δίνεται ο παρακάτω πίνακας που δείχνει για κάθε scheduler της στήλης πόσο τις εκατό βελτίωση πέτυχε σε σχέση με τον scheduler της γραμμής

	default	NetMARKS	Bin Balancer	combined
default	-	30.76%	-8.80%	5.50%
NetMARKS	30.76%	-	-42.27%	-23.94%
Bin Balancer	8.80%	42.27%	-	14.78%
combined	-5.50%	23.94%	-14.78%	-

Πίνακας 3. Σύγκριση βελτίωσης χρόνου απόκρισης μεταξύ των schedulers

Αξίζει να αναλυθούν και τα αποτελέσματα για κάθε application της εφαρμογής. Αρχικά για το response time to shipping application προκύπτουν τα εξής αποτελέσματα:

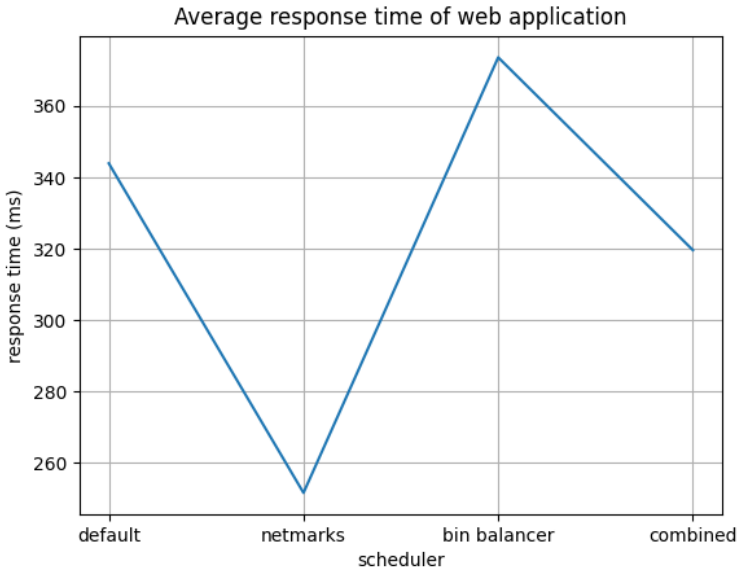


Εικόνα 21.Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το shipping application

Παρατηρείται ότι τα αποτελέσματα είναι παρόμοια με της συνολικής εφαρμογής. Μάλιστα ο NetMARKS σε σχέση με τον default scheduler του Kubernetes

παρουσιάζει βελτίωση κατά 30% ενώ ο Bin Balancer έχει αυξημένο response time κατά 17%. Ο combined scheduler ωστόσο παρουσίασε μικρή βελτίωση κατά 8% περίπου. Επίσης παρατηρείται ότι το μέσο response time σε κάθε περίπτωση είναι πολύ υψηλό, πάνω από 1 δευτερόλεπτο.

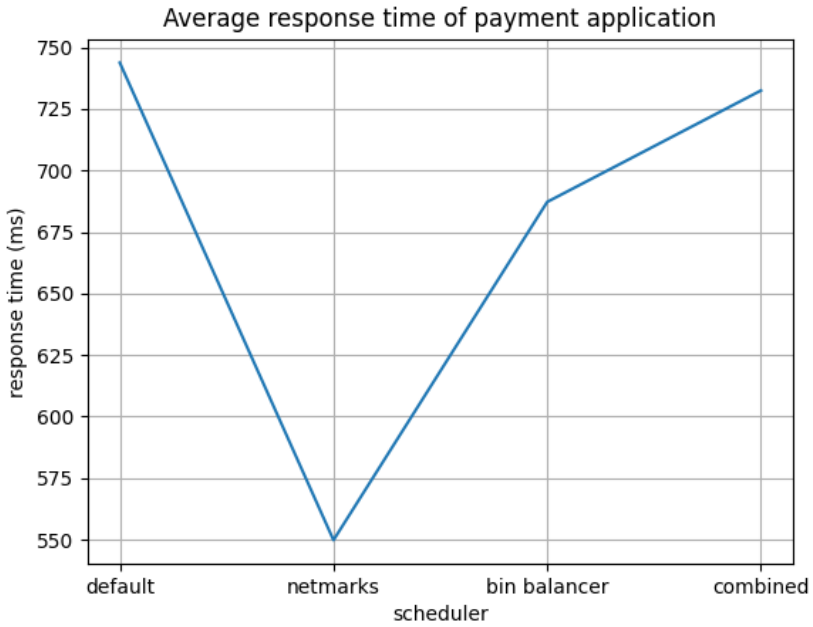
Όσο αναφορά το response time του web application προκύπτουν τα εξής αποτελέσματα:



Εικόνα 22. Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το web application

Παρατηρείται ότι τα αποτελέσματα του web είναι παρόμοια με εκείνα του Shipping. Συγκεκριμένα, σε σχέση με τον default scheduler ο NetMARKS παρουσιάζει 36% βελτίωση, ο Bin Balancer αύξηση response time κατά 9% και ο combined μείωση response time κατά περίπου 7.5%.

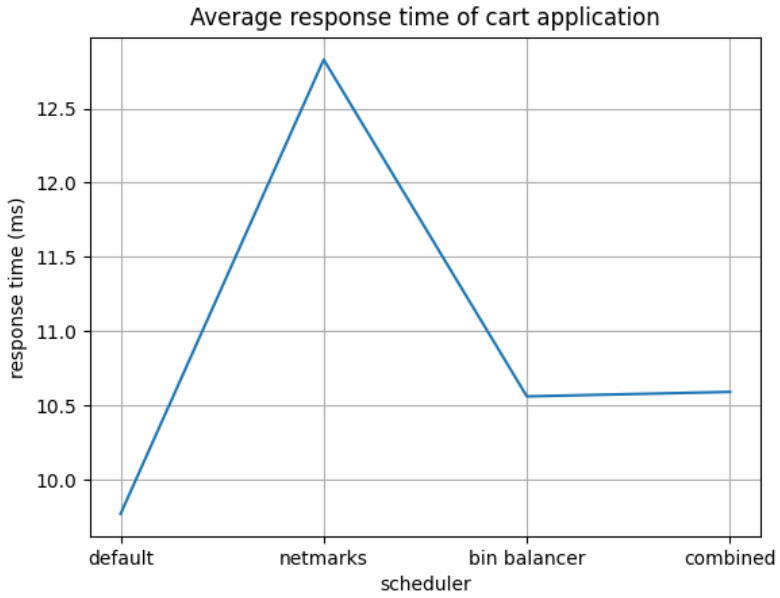
Σχετικά με το payment application, τα αποτελέσματα που προκύπτουν είναι τα εξής:



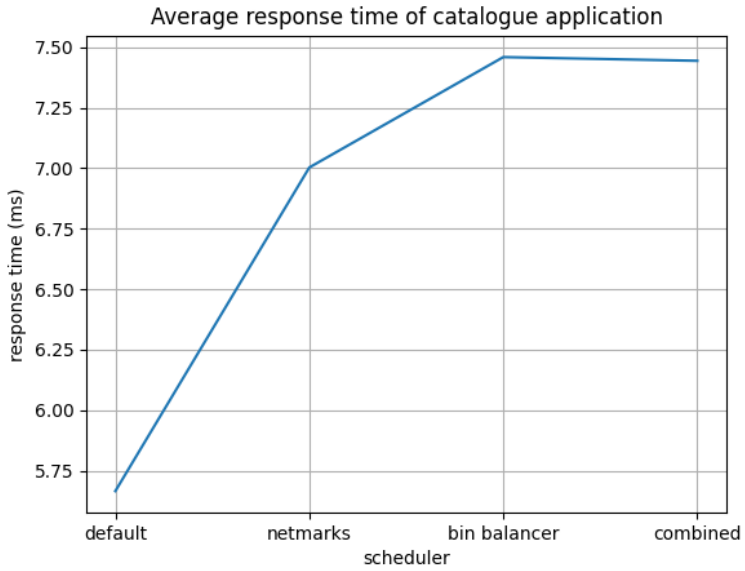
Εικόνα 23. Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το payment application

Στα συγκεκριμένα αποτελέσματα παρουσιάζεται βελτίωση του NetMARKS scheduler κατά 35% σε σχέση με τον default scheduler καθώς επίσης βελτίωση παρουσιάζουν και ο Bin Balancer κατά 8% και ο combined κατά 1% το οποίο μπορεί να θεωρηθεί και αμελητέο.

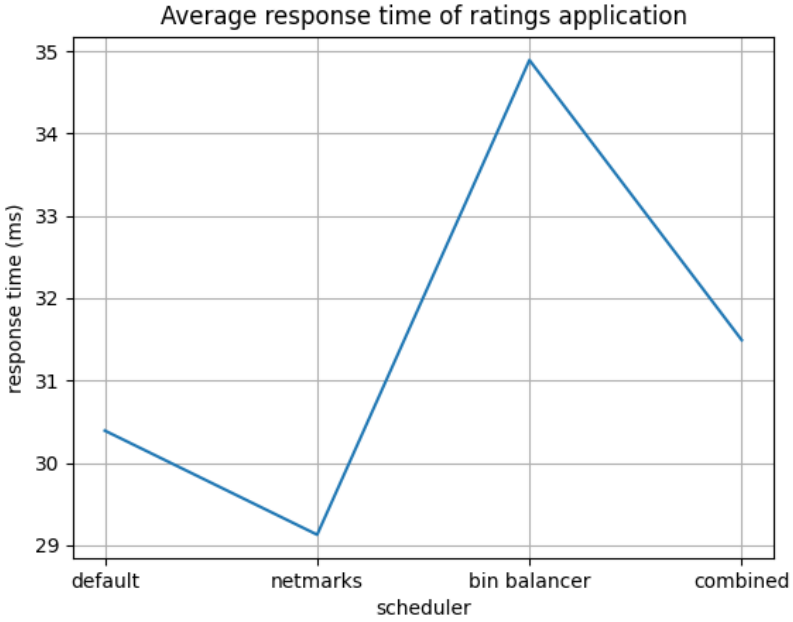
Για τα αποτελέσματα των application cart, catalogue, ratings και user δεν θα δοθούν συμπεράσματα, καθώς οι μέσοι χρόνοι απόκρισης τους είναι εξαιρετικά μικροί για να θεωρηθούν σημαντικοί για την ανάλυση. Ωστόσο θα δοθούν τα αποτελέσματα τους με την σειρά παρακάτω.



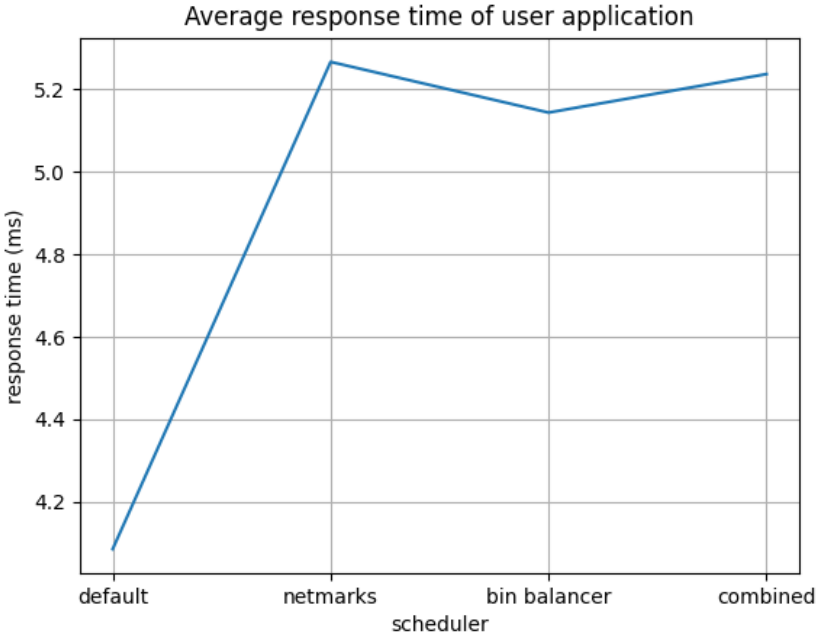
Εικόνα 24. Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το cart application



Εικόνα 25. Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το catalogue application



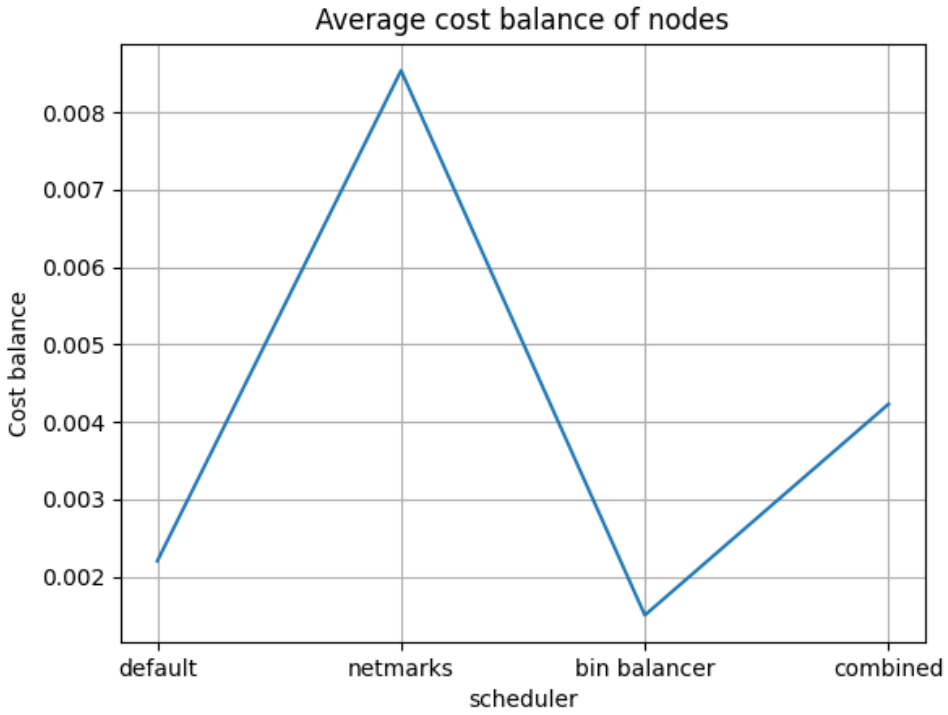
Εικόνα 26.Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το ratings application



Εικόνα 27.Γράφημα μέσης χρονικής απόκρισης εφαρμογής για το user application

Σχετικά με τα αποτελέσματα της συνολικής εφαρμογής τα αποτελέσματα μεταξύ των διάφορων schedulers για τη βελτίωση ή την επιδείνωσή τους, φαίνονται στον παρακάτω πίνακα.

Όσον αφορά τα αποτελέσματα ισορροπίας του κόστους προέκυψαν τα εξής αποτελέσματα:



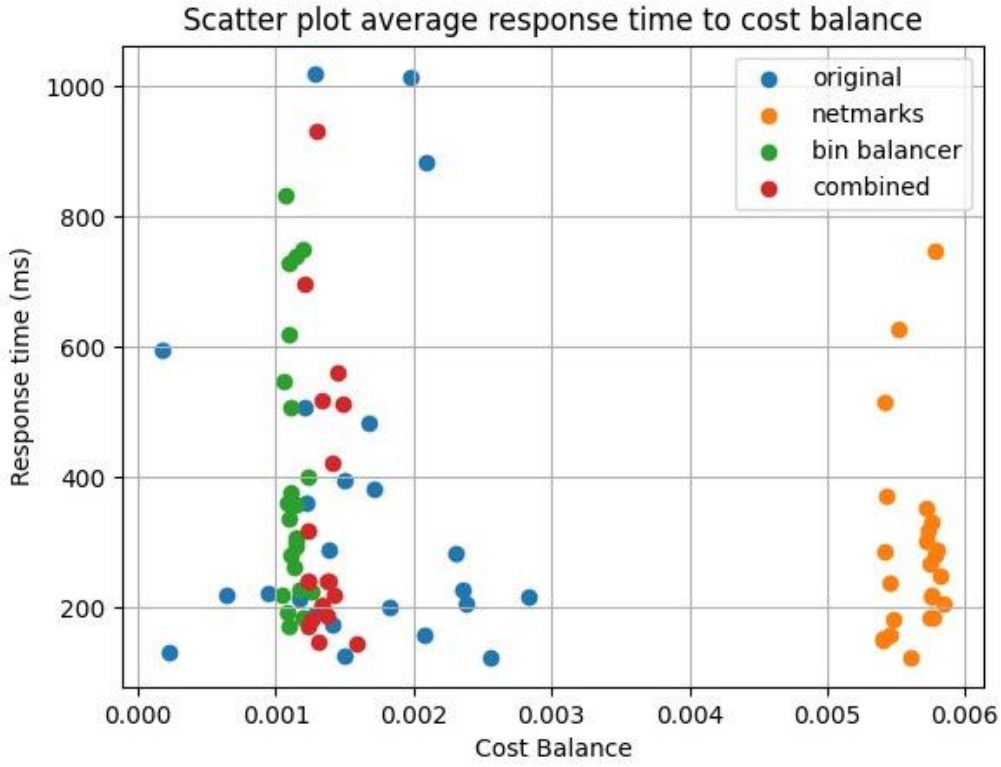
Εικόνα 28.Γράφημα ισορροπίας κόστους της εφαρμογής για τους διάφορους schedulers

Αρχικά, παρατηρείται ότι ο αλγόριθμος NetMARKS παρουσίασε με διαφορά τη μεγαλύτερη τιμή ισορροπίας κόστους, και άρα ανισορροπία, από τους υπόλοιπους αλγόριθμους και μάλιστα μεγαλύτερη κατά 400%. Αυτό οφείλεται στο γεγονός ότι ο αλγόριθμος με σκοπό τη βελτίωση του response time, "συγκεντρώνει" πολλά pods στο ίδιο node οπότε δεν υπάρχει καλή ισορροπία κόστους. Αντιθέτως, ο αλγόριθμος Bin Balancer, επιτυγχάνει όχι μόνο την καλύτερη ισορροπία σε σχέση με τους υπολοίπους schedulers, αλλά και τη βέλτιστη δυνατή για το application. Η βελτίωση που παρέχει είναι κατά 68%. Τέλος παρατηρούμε ότι ο combined scheduler έχει ισορροπία κόστους και συνεπώς ανισορροπία υψηλότερη από εκείνη του default scheduler (100% αύξηση) αλλά μικρότερη σε σχέση με κείνη του NetMARKS. Δίνεται επίσης ο πίνακας βελτίωσης της ισορροπίας κόστους, όπως και για το μέσο response time, που δείχνει για κάθε scheduler της στήλης πόσο τις εκατό βελτίωση πέτυχε σε σχέση με τον scheduler της γραμμής.

	default	NetMARKS	Bin Balancer	combined
default	-	-288%	47%	-92%
NetMARKS	288%	-	469%	102%
Bin Balancer	-47%	-469%	-	-182%
combined	92%	-102%	182%	-

Πίνακας 4. Πίνακας σύγκρισης βελτίωσης στην ισορροπία κόστους μεταξύ των διάφορων schedulers

Τα αποτελέσματα που παρουσιάστηκαν μπορούν να παρατηρηθούν και στο παρακάτω scatter plot, έτσι ώστε να υπάρχει μια εικόνα της κάθε δοκιμής και των αποτελεσμάτων της. Στο γράφημα, οι δοκιμές τοποθετούνται στον χώρο βάσει των αποτελεσμάτων τους στη μέση χρονική απόκριση της εφαρμογής κατά την εκτέλεση τους και την ισορροπία κόστους της εφαρμογής για τη συγκεκριμένη εφαρμογή.



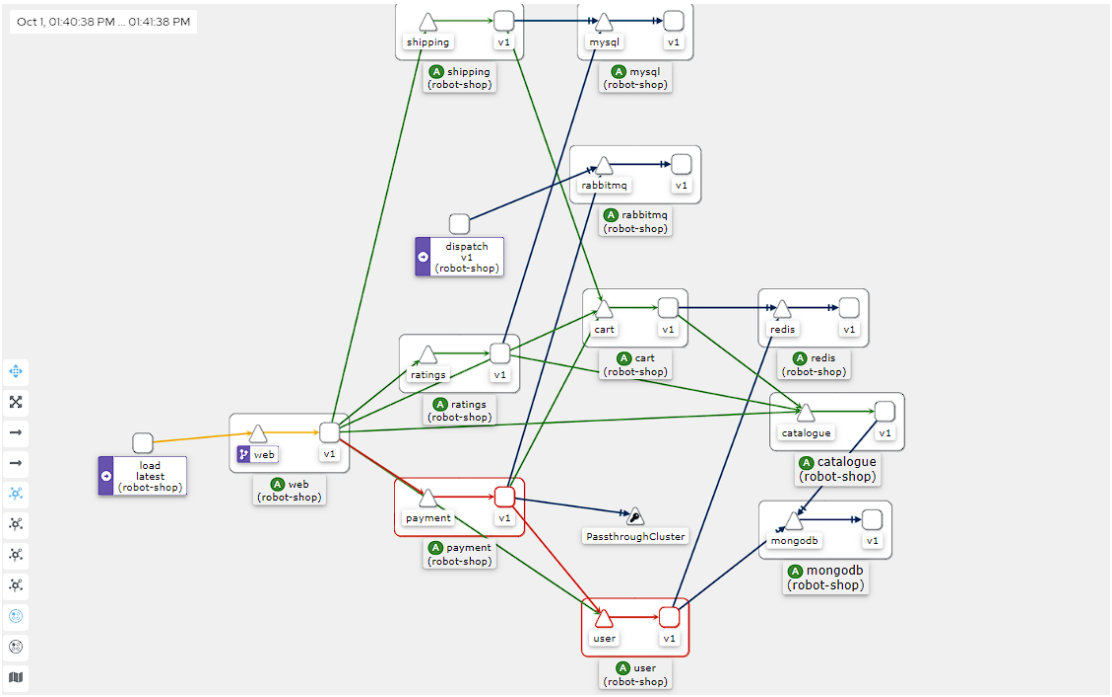
Εικόνα 29.Scatter plot των δοκιμών για όλους τους schedulers βάσει μέσου response time – ισορροπία κόστους.

Παρατηρείται ότι οι αλγόριθμοι που υλοποιήθηκαν από το έργο παράγουν αποτελέσματα σε μικρότερο εύρος από εκείνο του default scheduler, καθώς δεν λαμβάνει υπόψη του μόνο αυτές τις μετρικές για το scheduling των pods. Επίσης για όλους τους schedulers παρατηρούνται δοκιμές που τα αποτελέσματα τους ξεφεύγουν αρκετά από εκείνα των υπόλοιπων που εκτελούνται για τον ίδιο scheduler. Αυτό οφείλεται σε τυχόν προβλήματα των pods κατά την εκτέλεση της δοκιμής. Και πάλι παρατηρείται με ευκολία ότι ο αλγόριθμος NetMARKS λαμβάνει την μεγαλύτερη ανισορροπία κόστους σε σχέση με τους υπόλοιπους ενώ για τον Bin Balancer τα περισσότερα στοιχεία του είναι συγκεντρωμένα σε χαμηλές τιμές ισορροπίας κόστους. Επίσης παρατηρείται ότι οι τιμές του NetMARKS είναι συγκεντρωμένες σε χαμηλότερες τιμές για το μέσο response time σε αντίθεση με τους υπόλοιπους αλγόριθμους. Τέλος, παρατηρείται ότι τα σημεία του combined αλγόριθμου είναι ανάμεσα σε εκείνα του Bin Balancer και του NetMARKS τόσο σε response time όσο και ισορροπία κόστους.

Κεφάλαιο 8: Συμπεράσματα

8.1 Συμπεράσματα δοκιμών

Αρχικά προκύπτει το συμπέρασμα ότι ο NetMARKS επιτυγχάνει σημαντική βελτίωση του response time σε σχέση με τον default scheduler και όλους τους αλγόριθμους και επομένως μπορεί να χρησιμοποιηθεί έναντι αυτού για εφαρμογές που απαιτούν γρήγορη απόκριση. Ωστόσο, προκαλεί μεγάλη ανισορροπία στα κόστη των nodes και ορισμένα καταναλώνουν περισσότερες πόρους σε σχέση με άλλα nodes. Το αποτέλεσμα αυτό είναι το αναμενόμενο καθώς με ανάλυση του αλγόριθμου που χρησιμοποιεί ο NetMARKS γίνεται φανερό ότι προσπαθεί να συγκεντρώσει όσα περισσότερα pods μπορεί σε ένα node με σκοπό να μειώσει το response time. Η ανισορροπία αυτή μπορεί να προκαλέσει υπερφόρτωση του node σε περίπτωση που τα Pods απαιτούν πολλούς πόρους και το node που τα κατατάσσει το NetMARKS δεν μπορεί να τα υποστηρίξει. Έτσι γίνεται φανερό και ότι ο NetMARKS ενδιαφέρεται μόνο για την μετρική του response time αγνοώντας λοιπούς παράγοντες. Επίσης αξίζει να σημειωθεί ότι ο NetMARKS προκάλεσε σημαντική βελτίωση σε applications με πολύ υψηλό flow σε άλλα application (όπως το Shipping application) και σε application που συνδέονται σε πολλά άλλα application και περιμένουν δεδομένα από σειρά άλλων application (όπως το web και payment application). Η σύνδεση αυτή φανερώνεται και στο γράφημα kiali που δόθηκε και στο κεφάλαιο 6.2



Εικόνα 30.Το γράφημα Kiali για την εφαρμογή robot-shop

Από την άλλη, ο scheduler Bin Balancer επιτυγχάνει να ισορροπήσει βέλτιστα στα nodes του cluster τα διάφορα pods ανάλογα με το OpenCost κόστος τους και επομένως βάσει των πόρων CPU και RAM που καταναλώνουν. Έτσι ο αλγόριθμος

αυτός προτιμάται σε περιπτώσεις που επιθυμείται η εξισορρόπηση του φορτίου στα nodes και η αποφυγή υπερφόρτωσης κάποιου από αυτά, διατηρώντας παράλληλα πιο “υγιές” το cluster. Ωστόσο παρατηρείται ότι το response time είναι το χειρότερο από του υπόλοιπους αλγόριθμους γεγονός που επιβεβαιώνεται από το ότι δεν εξετάζει καθόλου τη μετρική του response κατά την εκτέλεσή του. Έτσι τα αποτελέσματα του σε response time δεν είναι ελεγχόμενα και μάλιστα ορισμένες φόρες, εάν τύχει η κατανομή να είναι τέτοια, επιτυγχάνει ακόμα και καλό response time. Όμως δεν μπορεί να θεωρηθεί δεδομένο και στις περισσότερες περιπτώσεις προκαλεί πολύ υψηλό response time.

Ο combined scheduler, συνδυάζει αυτό που επιτυγχάνει ο NetMARKS και ο Bin Balancer scheduler. Έτσι, με tradeoff την ισορροπία κόστους η το response time βελτιώνεται η μετρική που ενδιαφέρει τον σχεδιαστή κάθε φορά. Παρόλη τη μεγάλη διαφορά στο tradeoff που παρατηρήθηκε, δηλαδή 100% αύξηση στην ανισορροπία του κόστους για 5% καλύτερο response time, σε κάποια συστήματα μπορεί να αποτελεί προσιτή βελτίωση καθώς δεν υπερφορτώνεται εύκολα το σύστημα. Επομένως ο combined μπορεί να θεωρηθεί ως η μέση λύση για τον αλγόριθμο NetMARKS και Bin Balancer, σε περίπτωση που δεν είναι επιθυμητή η κατάληξη σε έναν από αυτούς απαραίτητα και να εξισορροπηθούν τα αρνητικά τους αλλά και τα θετικά τους.

Μάλιστα το tradeoff αυτό μπορεί να οριστεί ανάλογα με το limit που τίθεται. Όπως αναφέρθηκε και προηγουμένως, το όριο αυτό για τις ακραίες τιμές του (0 και +inf) καταλήγει στους δύο άλλους αλγόριθμους, τον NetMARKS και τον Bin Balancer αντίστοιχα. Συνεπώς ο combined αλγόριθμος θα μπορούσε να χρησιμοποιηθεί από ένα σύστημα που αξιοποιεί αυτές τις μετρικές. Έτσι αλλαγή limit στο κατάλληλο κάθε φορά δίνει την ευελιξία στην εύκολη προσαρμογή του αλγόριθμου στις συνθήκες.

Σε κάθε περίπτωση, οι αλγόριθμοι που αναλύθηκαν έχουν διαφορετική επίδραση ανάλογα και της εφαρμογής που εφαρμόζονται. Έτσι εφαρμογές με πολύπλοκή σύνδεση των application, υψηλή μεταφορά δεδομένων κατά την επικοινωνία και εξάρτηση των Application από σειρά επικοινωνίας άλλων Application, θα επωφελούναι σημαντικά από τον αλγόριθμο NetMARKS. Αντιθέτως, εφαρμογές με απλή και ελαφριά επικοινωνία μεταξύ των applications δεν θα παρουσιάσουν σημαντικές αλλαγές στο response time με τη χρήση του NetMARKS scheduler. Για τον Bin Balancer, εφαρμογές που περιέχουν pods με υψηλή ανάγκη σε πόρους, η ισορροπία του κόστους κρίνεται πιο σημαντική για την επίδοση της εφαρμογής και την υγεία του cluster σε σχέση με μια εφαρμογή με χαμηλές απαιτήσεις σε πόρους για κάθε pod. Έτσι και ο combined αλγόριθμος μπορεί να πετύχει την καλύτερη μέση λύση σε εφαρμογές με σημαντική επικοινωνία μεταξύ των pods και pods που έχουν υψηλές απαιτήσεις. Στην περίπτωση αυτή η επίπτωσή του θα είναι πιο φανερή για τη βελτίωση της εφαρμογής στον τομέα που θα επιλέξει ο σχεδιαστής, δηλαδή είτε στο καλύτερο response time είτε στην καλύτερη ισορροπία κόστους.

8.2 Μελλοντικές κατευθύνσεις

Οι αλγόριθμοι που αναπτύχθηκαν κατά την έκταση της διπλωματικής και επιβεβαιώθηκαν μέσω πειραματικών δοκιμών, μπορούν να επεκταθούν μελλοντικά και να προσφέρουν καλύτερα τις λειτουργίες τους και τα θετικά τους σε ένα σύστημα. Ορισμένες προτάσεις για την επέκταση τους και την καλύτερη απόδοση τους είναι:

- Μια εφικτή και αποδοτική ιδέα θα μπορούσε να είναι η χρήση μηχανικής μάθησης στους αλγόριθμους για υπολογισμό των παραμέτρων. Συγκεκριμένα

ένα μεγάλο πρόβλημα του NetMARKS αποτελεί η εύρεση της αρχικής κατανομής. Έτσι με την ανάπτυξη ενός μοντέλου που θα επιλέγει την καλύτερη αρχική κατανομή για τον NetMARKS, δεν χρειάζονται πλέον πολλαπλές δοκιμές του Scheduler με διαφορετικές αρχικές κατανομές και μπορεί άμεσα να παραχθεί κατανομή με ικανοποιητικά αποτελέσματα σε χρόνο απόκρισης. Επίσης μοντέλο μηχανικής μάθησης θα μπορούσε να χρησιμοποιηθεί στον combined scheduler για τον υπολογισμό του ορίου βάσει των αναγκών του σχεδιαστή, της εφαρμογής και της υποδομής. Με τον τρόπο αυτό δεν χρειάζονται πολλαπλές δοκιμές για την εύρεση του καταλληλότερου ορίου

- Επίσης οι αλγόριθμοι που παρουσιάστηκαν μπορούν να επεκταθούν. Ο Bin Balancer scheduler θα μπορούσε να επεκταθεί με τεχνικές έτσι ώστε να αποκτήσει δυναμικό χαρακτήρα και να μειωθεί ο χρόνος εκτέλεσης του. Επίσης ο combined θα μπορούσε να επεκταθεί και με άλλες παραμέτρους ως όριο έτσι ώστε οι επιλογές του να βασίζονται σε περισσότερες μετρικές του συστήματος και του pod. Επίσης θα μπορούσε να επεκταθεί και με άλλους αλγόριθμους για την επιλογή δρομολόγησης του pod.
- Μια ακόμη βελτίωση των αλγόριθμων θα μπορούσε να γίνει μέσω χρήσης τεχνικών μελλοντικών προβλέψεων για την κατάσταση και της εφαρμογής και του συστήματος, έτσι ώστε οι schedulers να μπορούν να αντιμετωπίσουν από νωρίς τυχόν προβλήματα στην εκτέλεση της εφαρμογής και στην υπερφόρτωση του συστήματος.
- Σε περίπτωση επίσης που θα χρειαστεί η μελλοντική χρήση των αλγορίθμων σε πραγματικά συστήματα, κρίνεται αναγκαία η ανάπτυξη των κατάλληλων plugins του Kubernetes Scheduling Frameworks καθώς στο τωρινό έργο χρησιμοποιήθηκε μια παραλλαγή της διαδικασίας δρομολόγησης. Με τον τρόπο αυτόν οι αλγόριθμοι θα μπορούν να χρησιμοποιηθούν αποδοτικά από το σύστημα και να επεκταθούν περαιτέρω με μεγαλύτερη ευκολία.

Σε κάθε περίπτωση οι schedulers μπορούν να επεκταθούν αλλά και να επεκτείνουν και άλλα συστήματα με τις λειτουργίες τους. Τα αποτελέσματα που προσφέρουν είναι ικανά να παρέχουν λύσεις στα προβλήματα που περιγράφουν και επομένως ένας scheduler που θα ήθελε να λάβει υπόψη του την μετρική χρόνου απόκρισης και ισορροπίας κόστους-πόρων μπορεί να ενσωματώσει τις λειτουργίες τους.
Οι schedulers όπως επιβεβαιώνεται από την διπλωματική εργασία αυτή χρίζουν ιδιαίτερης προσοχής και τα αποτελέσματα τους όσο αναφορά την επιρροή στο σύστημα είναι πολύ ενδιαφέροντα. Η εξέλιξη τους μπορεί να προσφέρει επαναστατικά αποτελέσματα τόσο στα Cloud συστήματα όσο και στην σύγχρονη τεχνολογία.

Βιβλιογραφία

1. Kubernetes Overview: https://kubernetes.io/docs/concepts/overview/ .
2. Horowitz, Mark. *1.1 Computing's energy problem (and what we can do about it):* 9 February 2014. DOI: 10.1109/ISSCC.2014.6757323.
3. Microservices: https://microservices.io/ .
4. Pahl, Claus. *Microservices: A Systematic Mapping Study:* 2016\. https://www.scitepress.org/PublishedPapers/2016/57855/57855.pdf.
5. Cloud Native: https://aws.amazon.com/what-is/cloud-native/.
6. VM: https://www.vmware.com/content/vmware/vmware-published-sites/us/topics/glossary/content/virtual-machine.html.html.
7. Siddiqui, Tamanna. *Comprehensive Analysis of Container Technology:* 21 November 2019. DOI: 10.1109/ISCON47742.2019.9036238.
8. *Containers:* https://www.docker.com/resources/what-container/.
9. Mercl, Lubos. *The Comparison of Container Orchestrators:* 2020 January 18. DOI: 10.1007/978-981-13-1165-9_62.
10. Kubernetes Components: https://kubernetes.io/docs/concepts/overview/components/ .
11. Kubernetes Services: https://kubernetes.io/docs/concepts/services-networking/service/ .
12. Kubernetes Manifests: https://monokle.io/learn/kubernetes-manifest-files-explained .
13. Kubernetes Taints and Tolerations: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.
14. Sidecar Containers: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/.
15. Kubernetes API: https://kubernetes.io/docs/concepts/overview/kubernetes-api/ .
16. Python pip: https://pypi.org/project/pip/.
17. Kubectl: https://kubernetes.io/docs/reference/kubectl/ .
18. Li, Wubin. *Service Mesh: Challenges, State of the Art, and Future Research Opportunities:* 4 April 2019. DOI: 10.1109/SOSE.2019.00026.
19. Istio: https://istio.io/latest/about/service-mesh/.
20. Istio architecture: https://istio.io/latest/docs/ops/deployment/architecture/ .
21. Istioctl: https://istio.io/latest/docs/ops/diagnostic-tools/istioctl/ .
22. Prometheus: https://prometheus.io/docs/introduction/overview/.
23. Prometheus Libraries: https://prometheus.io/docs/instrumenting/clientlibs/ .
24. Prometheus Push Gateways: https://github.com/prometheus/pushgateway.
25. Prometheus Configuration: https://prometheus.io/docs/prometheus/latest/configuration/configuration/ .

26. Kiali: https://istio.io/latest/docs/ops/integrations/kiali/.

27. Helm: https://circleci.com/blog/what-is-helm/.

28. Opencost: https://www.opencost.io/docs.

29. Locust: https://docs.locust.io/en/stable/ .

30. Kubernetes Scheduling Framework: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ .

31. Carrion, Carmen. Kubernetes Scheduling: Taxonomy, ongoing issues and challenges: https://doi.org/10.1145/3539606.

32. György Dósa, Jiří Sgall. *First Fit bin packing: A tight analysis:* 2021. DOI: 10.4230/LIPIcs.STACS.2013.538.

33. Chaudhur, Samit. Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem: 4 December 1994. DOI: 10.1109/92.335014.

34. https://doi.org/10.1016/j.eij.2015.07.001. A review of metaheuristic scheduling techniques in cloud computing: 25 July 2015. https://doi.org/10.1016/j.eij.2015.07.001.

35. Hongzi Mao, Malte Schwarzkopf. Learning Scheduling Algorithms for Data Processing Clusters: 23 August 2019. https://doi.org/10.1145/3341302.3342080.

36. Wojciechowsk, Łukasz. NetMARKS: Network Metrics-AwaRe Kubernetes: DOI: 10.1109/INFOCOM42981.2021.9488670.

37. Kubernetes Bin Packing Default Scheduler: https://kubernetes.io/docs/concepts/scheduling-eviction/resource-bin-packing/.

38. López-Camacho, Eunice. *Understanding the structure of bin packing problems through principal component analysis:* 21 April 2013. DOI: 10.1016/j.ijpe.2013.04.041.

39. Kubernetes Bin Packing Advantages: https://www.infoq.com/articles/kubernetes-bin-packing/.

40. MickroK8s: https://microk8s.io/docs.

41. Sebastian Bohm, Guido Wirtz. *Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes:* 25 February 2021. https://ceur-ws.org/Vol-2839/paper11.pdf.

42. Robot-Shop: https://github.com/instana/robot-shop .

43. NodeJs: https://nodejs.org/en.

44. Java: https://www.java.com/en/.

45. Python: https://www.python.org/.

46. Golang: https://go.dev/.

47. PHP: https://www.php.net/.

48. Mongo DB: https://www.mongodb.com/.

49. Redis: https://redis.io/.

50. MySQL: https://www.mysql.com/.

51. RabbitMq: https://www.rabbitmq.com/.

52. Nginx: https://www.nginx.com/.

53. AngularJS: https://angularjs.org/.

54. Istio Sidecar Injection: https://istio.io/latest/docs/setup/additional-setup/sidecar-injection/ .

55. Istio installation: https://istio.io/latest/docs/setup/getting-started/#download.

56. Opencost Installation: https://www.opencost.io/docs/installation/install .

57. Kubernetes Client for python: https://github.com/kubernetes-client/python/blob/master/.

58. Multiple Kubernetes Schedulers: https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers.

59. Hunter, John D. Matplotlib: A 2D Graphics Environment: June 2007. DOI: 10.1109/MCSE.2007.55.