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

Δυναμική και Αποδοτική Χρονοδρομολόγηση σε Συστοιχίες Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντώνιος Γ. Καραντώνης

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.

Αθήνα, Φεβρουάριος 2024

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

Δυναμική και Αποδοτική Χρονοδρομολόγηση σε Συστοιχίες Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αντώνιος Γ. Καραντώνης

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 2η Απριλίου 2024.

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

Αθήνα, Φεβρουάριος 2024

.....
Αντώνιος Γ. Καραντώνης

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

Περίληψη

Στο σύγχρονο ψηφιακό κόσμο οι αρχιτεκτονικές μικροϋπηρεσιών (microservices) και η τεχνολογία των κιβωτίων (containers) έχουν αναδειχθεί ως κυρίαρχες πρακτικές στην ανάπτυξη και διανομή λογισμικού χάρη στην ευελιξία, την επεκτασιμότητα και την αποδοτικότητά τους. Ως φυσική εξέλιξη η ανάγκη για αποτελεσματική διαχείριση των εφαρμογών αυτών οδήγησε στην ανάδυση του Kubernetes ως το βασικό εργαλείο για την ενορχήστρωση (orchestration) των συστοιχιών (clusters) που τις απαρτίζουν. Βασικό κομμάτι στην αρχιτεκτονική του Kubernetes είναι ο χρονοδρομολογητής ο οποίος αναλαμβάνει την αποτελεσματική διανομή και εκτέλεση των containers στον κατάλληλο υπολογιστικό κόμβο (node) εντός του cluster που απαρτίζει την εφαρμογή.

Η παρούσα διπλωματική εργασία αναπτύσσει και αξιολογεί έναν εναλλακτικό δρομολογητή για το Kubernetes, με στόχο τη βελτιστοποίηση της απόδοσης των εφαρμογών μέσω της δυναμικής και αποδοτικής διαχείρισης των πόρων. Αντλώντας μετρικές από το Istio Service Mesh, ο εναλλακτικός χρονοδρομολογητής επιδιώκει να βελτιώσει τον χρόνο απόκρισης των εφαρμογών. Παράλληλα με δυναμικό τρόπο εξασφαλίζει διαρκώς τη βέλτιστη δρομολόγηση ενώ διασφαλίζει την υψηλή διαθεσιμότητα και την ομαλή λειτουργία του συστήματος. Με τη χρήση των δυνατοτήτων που προσφέρει το Istio, ο σχεδιασμός ενσωματώνει μετρικές παρακολούθησης και ανάλυσης της κίνησης δικτύου μεταξύ των υπηρεσιών, προσφέροντας μια πιο ενημερωμένη και προσαρμοσμένη στην πραγματικότητα δρομολόγηση των containers. Αυτό επιτρέπει την βελτίωση της απόδοσης κάθε εφαρμογής, μειώνοντας ταυτόχρονα το συνολικό κόστος λειτουργίας του συστήματος.

Η εργασία εξετάζει επίσης τη λειτουργία και τους περιορισμούς που παρουσιάζουν τόσο ο προκαθορισμένος χρονοδρομολογητής του Kubernetes όσο και άλλες προτεινόμενες λύσεις. Μέσω μίας σειράς πειραματικών δοκιμών αποδεικνύει την αποτελεσματικότητα του εναλλακτικού χρονοδρομολογητή, ενώ παράλληλα προτείνει πιθανές μελλοντικές επεκτάσεις για περεταίρω έρευνα πάνω στο συγκεκριμένο αντικείμενο.

Λέξεις κλειδιά:

Μικρουπηρεσίες, Containers, Kubernetes, Ενορχηστρωτής, Χρονοδρομολογητής, Δυναμική Χρονοδρομολόγηση, Istio, Kiali, Locust, Μείωση Χρόνου Απόκρισης

Abstract

In the modern digital world, microservices architectures and container technology have emerged as dominant practices in software development and distribution, thanks to their flexibility, scalability, and efficiency. As a natural progression, the need for effective management of these applications has led to the emergence of Kubernetes as the primary tool for orchestrating the clusters that compose these applications. A key component in the architecture of Kubernetes is the scheduler, which efficiently assigns pods to the appropriate nodes within the cluster that constitutes the application.

This thesis develops and evaluates an alternative scheduler for Kubernetes, aiming to optimize application performance through dynamic and efficient resource management. Leveraging metrics from the Istio Service Mesh, the alternative scheduler seeks to improve application response time. Concurrently, it dynamically ensures constant optimal scheduling while maintaining high availability and smooth system operation. By utilizing the capabilities offered by Istio, the design incorporates network traffic monitoring and analysis metrics between services, offering a more informed and reality-adapted container scheduling. This allows for the enhancement of the performance of each application while simultaneously reducing the overall operational cost of the system.

The thesis also examines the functionality and limitations of both the default Kubernetes scheduler and other proposed solutions. Through a series of experimental tests, it demonstrates the effectiveness of the proposed scheduler while also suggesting possible future extensions for further research on this particular subject.

Keywords:

Microservices, Containers, Kubernetes, Orchestrator, Scheduler, Dynamic Scheduling, Istio, Kiali, Locust, Decrease in response time

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Παναγιώτη Τσανάκα για την ευκαιρία που μου έδωσε και για την άμεση ανταπόκρισή του καθ' όλη τη διάρκεια της εκπόνησης της διπλωματικής μου εργασίας.

Επίσης, ευχαριστώ τον υποψήφιο διδάκτορα κ. Κατσαούνη Σταμάτη για την ουσιαστική του βοήθεια και τη καθοδήγηση που μου παρείχε σε όλα τα στάδια της διπλωματικής μου εργασίας.

Ακόμα, θα ήθελα να ευχαριστήσω τους γονείς μου και τα αδέρφια μου για την συνεχή αγάπη, υποστήριξη και κατανόηση τους.

Τέλος, θα ήθελα να ευχαριστήσω τη κοπέλα μου και τους φίλους μου για την συμπαράστασή τους σε όλη αυτή την πορεία.

Αθήνα, Φεβρουάριος 2024
Αντώνης Καραντώνης

Στη γιαγιά και τον παππού μου,

Περιεχόμενα

Κατάλογος Εικόνων

Κατάλογος Πινάκων

ΚΕΦΑΛΑΙΟ 1 – Εισαγωγή

Οι σύγχρονες εφαρμογές διαδικτύου χρησιμοποιούν πληθώρα τεχνολογιών που αποσκοπούν στη διευκόλυνση τόσο του σχεδιασμού και της υλοποίησης τους όσο και της εγκατάστασης και της συντήρησής τους. Μία από τις βασικές τεχνολογίες που πλέον αποτελεί κοινή πρακτική για τις περισσότερες μεγάλες εφαρμογές είναι η χρήση κιβωτίων (containers). Για την εύρυθμη λειτουργία των containers χρησιμοποιείται στις περισσότερες περιπτώσεις κάποιος ενορχηστρωτής (orchestrator). Το Kubernetes είναι ο συχνότερος orchestrator συστοιχιών (cluster) από containers που χρησιμοποιείται στην αγορά τη δεδομένη χρονική στιγμή [1]. Η διπλωματική εργασία αυτή ασχολείται με ένα συγκεκριμένο κομμάτι του orchestrator, που αποκαλείται χρονοδρομολογητής (scheduler), και παρουσιάζει τη λειτουργία του, τους περιορισμούς που έχει καθώς και μία λύση ώστε να γίνει πιο αποδοτικός.
Στο κεφάλαιο αυτό περιγράφεται συνοπτικά το θέμα καθώς και το πρόβλημα στο οποίο η παρούσα εργασία επιχειρεί να δώσει λύση. Επιπλέον, παρουσιάζεται ο στόχος της εργασίας καθώς και μία επισκόπηση της λύσης που προτείνεται. Τέλος, παρουσιάζεται η δομή της διπλωματικής εργασίας αυτής.

1.1 Παρουσίαση θέματος και προβλήματος

Η ανάγκη για σύγχρονες εφαρμογές διαδικτύου που να μπορούν να εξυπηρετούν ταυτόχρονα και γρήγορα μεγάλο αριθμό αιτημάτων έχει οδηγήσει τις εταιρείες στο να απομακρυνθούν από τις παραδοσιακές μονολιθικές αρχιτεκτονικές και να στραφούν προς την κατεύθυνση των αρχιτεκτονικών μικροϋπηρεσιών (microservices), οι οποίες εξυπηρετούν καλύτερα τις ανάγκες τους. Για την αποδοτική υλοποίηση τέτοιου είδους αρχιτεκτονικών έχουν αναπτυχθεί τα τελευταία χρόνια εργαλεία και τεχνολογίες που αποτελούν πλέον βασικά στοιχεία της διαδικασίας ανάπτυξης λογισμικού. Το Kubernetes είναι μία από τις τεχνολογίες αυτές που διαχειρίζεται τα containers της εφαρμογής και αποτελείται από πολλά διαφορετικά κομμάτια, καθένα από τα οποία έχει το δικό του ρόλο. Ο scheduler είναι ένα από τα στοιχεία αυτά και η βασική αρμοδιότητα του είναι να επιλέγει και να δρομολογεί σε κάποιον κόμβο (node) του cluster τις αντίστοιχες κάψουλες (pods). Στην ορολογία του Kubernetes, nodes είναι οι υπολογιστικοί κόμβοι που αποτελούν το cluster ενώ pods είναι οι θεμελιώδεις μονάδες που δρομολογούνται στα nodes και αποτελούνται από ένα ή περισσότερα containers. Ο scheduler λοιπόν επιλέγει με βάση τις απαιτήσεις του κάθε pod σε ποιο node θα ήταν το ιδανικότερο να δρομολογηθεί το pod αυτό, έτσι ώστε να υπάρχει όσο το δυνατόν πιο ομοιόμορφη κατανομή των πόρων του cluster.
Το πρόβλημα που αποτέλεσε και αφορμή για την παρούσα διπλωματική εργασία είναι ότι ο scheduler, όταν καλείται να κάνει την επιλογή αυτή, δεν έχει στη διάθεσή του στοιχεία που να περιγράφουν τη δυναμική λειτουργία της εφαρμογής. Έτσι λοιπόν αποφασίζει χρησιμοποιώντας μόνο στατικά δεδομένα που ορίζονται στην περιγραφή των pods και αποτελούν κυρίως άνω και κάτω φράγματα σχετικά με τις ανάγκες που αναμένεται να έχουν. Η πραγματικότητα αυτή έχει ως αποτέλεσμα οι επιλογές που κάνει ο scheduler να είναι υποβέλτιστες ως προς την απόδοση της εφαρμογής. Υπάρχουν λοιπόν περιθώρια για

αποτελεσματικότερη δρομολόγηση που θα μπορούσε να βελτιώσει την απόδοση της εφαρμογής προς μία ή και περισσότερες κατευθύνσεις. Οι βελτιώσεις αυτές θα μπορούσαν να στοχεύουν για παράδειγμα στη μείωση του χρόνου απόκρισης της εφαρμογής [2], ζητούμενο για όλες τις σύγχρονες εφαρμογές. Άλλο παράδειγμα είναι η καλύτερη διαχείριση των πόρων [3] που θα μπορούσε να μειώσει τις ανάγκες για υπολογιστική ισχύ και συνεπώς το κόστος λειτουργίας μιας σύγχρονης εφαρμογής. Αν αναλογιστεί κανείς τα χρηματικά ποσά που πληρώνουν οι εταιρείες για αγορά εξυπηρετητών και υποδομών ή τα ποσά που δαπανώνται σε υπηρεσίες φιλοξενίας (hosting) εφαρμογών στο υπολογιστικό νέφος (cloud) [4], γίνεται αντιληπτό πως τέτοιες βελτιώσεις έχουν ουσιαστική σημασία. Ο άλλος βασικός περιορισμός του scheduler αυτού είναι πως από τη στιγμή που θα δρομολογήσει κάποιο pod, δεν επαναξιολογεί την επιλογή του αυτή καθ΄ όλη τη διάρκεια ζωής του pod. Η δυναμική φύση όμως τον εφαρμογών δημιουργεί την ανάγκη για συνεχή αξιολόγηση των επιλογών και διορθωτικές κινήσεις που θα εξασφαλίζουν διαρκώς τη βέλτιστη τοπολογία της εφαρμογής.

1.2 Στόχος

Στόχος της εργασίας αυτής είναι η παρουσίαση μιας λύσης που αναπτύχθηκε με σκοπό να αντιμετωπίσει τους υπάρχοντες περιορισμούς του Kubernetes scheduler. Η λύση αυτή είναι ένας νέος scheduler ο οποίος μπορεί δυναμικά να αντλεί μετρικές σχετικά με την επικοινωνία μεταξύ των pods του cluster. Η λύση που σχεδιάστηκε και υλοποιήθηκε, χρησιμοποιεί μια υπηρεσία που λειτουργεί σαν επιπλέον στρώμα πάνω από τα pods της εφαρμογής και έχει τη δυνατότητα να συλλέγει μετρικές σχετικά με την επικοινωνία των στοιχείων που την απαρτίζουν. Με τη χρησιμοποίηση των μετρικών αυτών, ο νέος scheduler επιτυγχάνει την πιο αποδοτική δρομολόγηση των pods, που οδηγεί σε σημαντική μείωση του χρόνου απόκρισης της εφαρμογής. Επιπλέον, ο νέος αυτός scheduler παρακολουθεί δυναμικά τη λειτουργία της εφαρμογής και προβαίνει σε διορθωτικές κινήσεις που εξασφαλίζουν διαρκώς την βέλτιστη δρομολόγηση των pods. Αξίζει να σημειωθεί ότι ο scheduler αυτός επιτυγχάνει τη λειτουργία του χωρίς να θυσιάζει τη διαθεσιμότητα των εφαρμογών αφού προβαίνει στις διορθωτικές κινήσεις με τρόπο τέτοιο που η εφαρμογή παραμένει συνεχώς διαθέσιμη. Το στοιχείο της συνεχούς διαθεσιμότητας κρίνεται απαραίτητο για όλες τις σύγχρονες εφαρμογές διαδικτύου και η αξία μιας λύσης που θα θυσίαζε τη διαθεσιμότητα αυτή δεν θα μπορούσε παρά να είναι αρκετά περιορισμένη.

1.3 Δομή

Στα επόμενα κεφάλαια της εργασίας αυτής θα αναπτυχθούν με περισσότερη λεπτομέρεια τόσο οι τεχνολογίες που αναφέρθηκαν παραπάνω όσο και η προτεινόμενη λύση και τα αποτελέσματα που επιτυγχάνει. Πιο συγκεκριμένα:

- Στο κεφάλαιο 2 θα αναπτυχθεί το θεωρητικό υπόβαθρο της εργασίας ξεκινώντας από τις σύγχρονες αρχιτεκτονικές ανάπτυξης λογισμικού και καταλήγοντας στο Kubernetes.

- Στο κεφάλαιο 3 θα παρουσιαστεί αναλυτικά ο scheduler του Kubernetes και οι αδυναμίες του και θα γίνει βιβλιογραφική αναφορά σε άλλες λύσεις που έχουν προταθεί.
- Στο κεφάλαιο 4 θα παρουσιαστεί η αρχιτεκτονική της προτεινόμενης λύσης και ο σχεδιασμός της.
- Στο κεφάλαιο 5 θα γίνει αναφορά σε όλα τα εργαλεία που χρησιμοποιήθηκαν για την ανάπτυξη του scheduler και θα παρουσιαστεί ο ακριβής τρόπος λειτουργίας του.
- Στο κεφάλαιο 6 θα αναπτυχθεί η σχεδίαση των πειραμάτων που πραγματοποιήθηκαν για την αξιολόγηση της αποτελεσματικότητας του scheduler ενώ θα παρουσιαστούν και θα αναλυθούν και τα αποτελέσματα των πειραμάτων αυτών.
- Στο κεφάλαιο 7 θα αναπτυχθούν τα συμπεράσματα και θα γίνει αναφορά στους περιορισμούς της παρούσας εργασίας καθώς και στις πιθανές μελλοντικές επεκτάσεις που θα μπορούσε να έχει το συγκεκριμένο θέμα.

ΚΕΦΑΛΑΙΟ 2 – Θεωρητικό Υπόβαθρο

Στο κεφάλαιο αυτό θα παρουσιαστεί το θεωρητικό υπόβαθρο που είναι απαραίτητο για την κατανόηση των επόμενων κεφαλαίων της εργασίας. Η περιγραφή των απαραίτητων εννοιών θα ξεκινήσει από το υψηλότερο επίπεδο της αρχιτεκτονικής και θα καταλήξει στο χαμηλότερο επίπεδο που είναι ο scheduler, το βασικό δηλαδή θέμα της διπλωματικής αυτής.

2.1 Σύγχρονες Αρχιτεκτονικές

Στην ανταγωνιστική και συνεχώς αναπτυσσόμενη αγορά της πληροφορικής οι σύγχρονες αρχιτεκτονικές μικροϋπηρεσιών (microservices) τείνουν να αντικαταστήσουν τις κλασσικές μονολιθικές αρχιτεκτονικές. Ο λόγος που οδηγεί στη νέα αυτή πραγματικότητα είναι οι πολλές διευκολύνσεις που προσφέρουν οι αρχιτεκτονικές microservices σχετικά με την ταχύτητα ανάπτυξης της εφαρμογής, το κόστος συντήρησης, την ευκολία κλιμάκωσης καθώς και πολλές άλλες [5].

2.1.1 Μονολιθική Αρχιτεκτονική

Μονολιθικές αποκαλούνται οι αρχιτεκτονικές στις οποίες όλα τα διαφορετικά κομμάτια μίας εφαρμογής ενώνονται μεταξύ τους δημιουργώντας μία ενιαία οντότητα. Αυτό σημαίνει πως όλη η λειτουργικότητα μίας εφαρμογής εκτελείται σαν μία υπηρεσία. Τέτοιου είδους αρχιτεκτονικές αποτελούσαν το βασικό τρόπο ανάπτυξης πριν από μερικά χρόνια λόγω των πλεονεκτημάτων που προσφέρουν. Αρχικά, οι μονολιθικές εφαρμογές είναι, στην αρχή του κύκλου ανάπτυξης τους, πιο εύκολο να αναπτυχθούν. Ο λόγος που ισχύει αυτό είναι πως επειδή ακριβώς όλα τα τμήματα βρίσκονται στο ίδιο σημείο, η επικοινωνία μεταξύ τους είναι ευκολότερη και συνεπώς επιταχύνεται η ανάπτυξη των εφαρμογών. Επιπλέον, οι μονολιθικές αρχιτεκτονικές τείνουν να έχουν ταχύτερους χρόνους απόκρισης χάρη στη συγκεντρωτική δομή τους. Πέρα από τη βελτίωση της απόδοσης, οι αρχιτεκτονικές αυτές προσφέρουν και ευκολότερη διαδικασία στησίματος αφού λειτουργούν σαν μια υπηρεσία και συνεπώς δεν απαιτούν σύνθετες διαμορφώσεις. Οι παραπάνω λόγοι σε συνδυασμό με την έλλειψη εργαλείων που θα διευκόλυναν την υλοποίηση διαφορετικών αρχιτεκτονικών καθιστούν σαφές γιατί οι μονολιθικές αρχιτεκτονικές αποτελούσαν για καιρό τη συχνότερη επιλογή στον χώρο της πληροφορικής.
Τα τελευταία χρόνια όμως τα δεδομένα έχουν αλλάξει. Οι ανάγκες για περιορισμό του κόστους, συνεχή διαθεσιμότητα των εφαρμογών και δυνατότητα ανάπτυξής τους σε μικρότερες αυτόνομες ομάδες έχουν καταστήσει εμφανείς τις αδυναμίες των κλασσικών αυτών αρχιτεκτονικών. Συγκεκριμένα, το πρώτο πρόβλημα εντοπίζεται στο ότι όταν αυξηθεί η χρήση σε ένα τμήμα της εφαρμογής, ολόκληρη η υποδομή της εφαρμογής πρέπει να μεγαλώσει, αφού λειτουργεί σαν ενιαία οντότητα. Η πραγματικότητα αυτή συνεπάγεται τη σημαντική αύξηση του κόστους ακόμα και αν ένα μικρό τμήμα εμφανίζει την ανάγκη. Επιπλέον, παρόλο που στην αρχή η ανάπτυξη λογισμικού επιταχύνεται, όταν οι εφαρμογές αρχίζουν να μεγαλώνουν σημαντικά είναι πολύ πιο δύσκολο και χρονοβόρο να γίνουν και να

δοκιμαστούν αλλαγές. Τέλος, σε περιπτώσεις σφαλμάτων σε τμήματα της εφαρμογής είναι πιθανό να τίθεται σε κίνδυνο ολόκληρη η λειτουργία της αφού ακριβώς συμπεριφέρεται σαν μια ενιαία οντότητα. Οι αδυναμίες αυτές έδωσαν χώρο με το πέρασμα του χρόνου στην αρχιτεκτονική των microservices να αναπτυχθεί και να κερδίσει μεγάλο κομμάτι της αγοράς [6]. Στην μετάβαση αυτή καθοριστικό ρόλο είχαν τα εργαλεία που αναπτύχθηκαν εκμεταλλευόμενα τις καινούργιες δυνατότητες που προσέφερε το υλικό (hardware) για εικονικοποίηση (virtualization) και τα λειτουργικά συστήματα για δημιουργία cgroups και kernel namespaces [7].

2.1.2 Αρχιτεκτονική Microservices

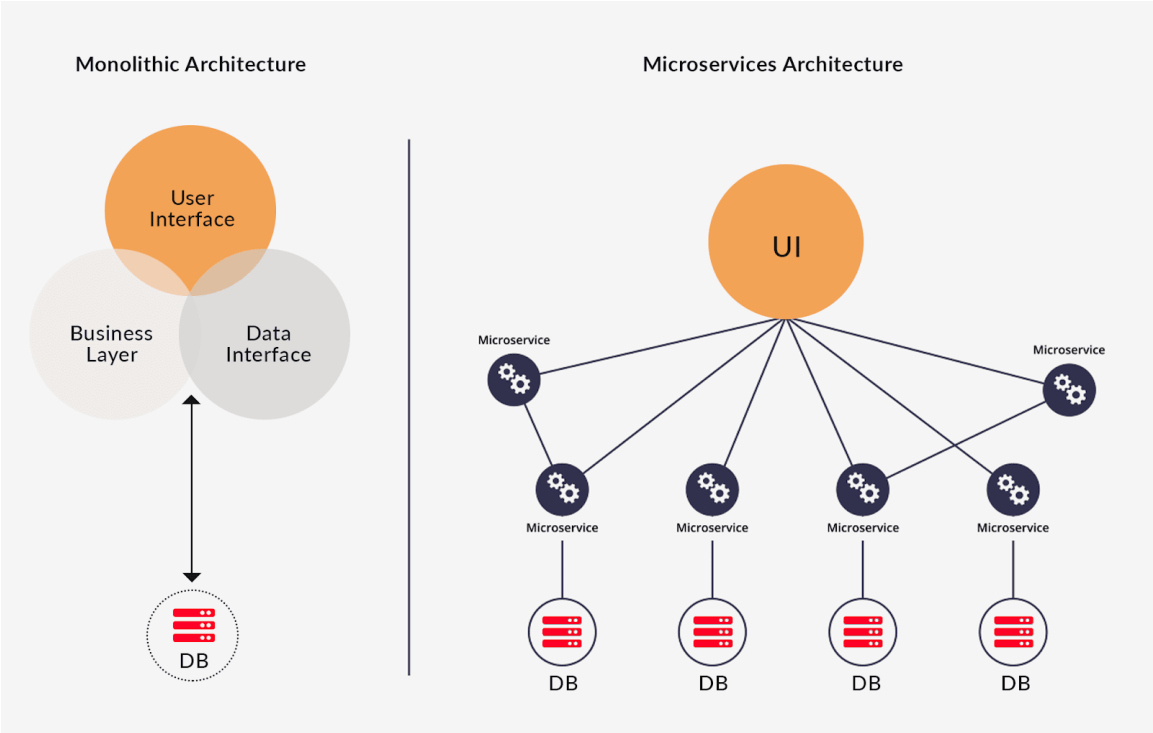
Στις αρχιτεκτονικές microservices η λειτουργία της εφαρμογής χωρίζεται σε επιμέρους μικρότερες αυτοτελείς και αυτόνομες υπηρεσίες. Κάθε μία από αυτές μπορεί να έχει τη δική της βάση δεδομένων και στόχος είναι να εκτελεί μία συγκεκριμένη και περιορισμένη λειτουργία. Η σύνθεση όλων των επιμέρους αυτών υπηρεσιών αποτελεί τελικά την εφαρμογή. Το γεγονός ότι πλέον υπάρχουν πολλές μικρές ανεξάρτητες εφαρμογές προσφέρει μεγαλύτερη ευελιξία τόσο στη διαδικασία υλοποίησης της εφαρμογής όσο και στη συντήρηση, το στήσιμο και τη διαθεσιμότητά της. Αξίζει βέβαια να σημειωθεί ότι πέρα από τα πολλαπλά πλεονεκτήματα που προσφέρει η προσέγγιση αυτή, δημιουργούνται και σημαντικές δυσκολίες. Αρχικά, αυξάνεται ο όγκος της δουλειάς σε ό,τι αφορά στη σχεδίαση και στην υλοποίηση, ειδικά στα πρώτα βήματα, αφού εισάγεται ένα επιπλέον επίπεδο που αφορά την επικοινωνία μεταξύ των διαφόρων υπηρεσιών. Επίσης, η απόδοση τέτοιων εφαρμογών συνήθως μειώνεται αφού οι ενδιάμεσες αυτές επικοινωνίες, όπως είναι λογικό, εισάγουν και ανάλογες καθυστερήσεις. Αυτοί οι γενικοί προβληματισμοί καθώς και άλλοι ειδικότεροι κατά περίπτωση καθιστούν τα microservices ιδανική λύση για συγκεκριμένα προβλήματα και όχι τη μοναδική λύση για οποιαδήποτε περίπτωση ανάπτυξης λογισμικού.

Για να γίνει η αναφορά στα πλεονεκτήματα της αρχιτεκτονικής αυτής που την έχουν καθιερώσει στην αγορά, είναι σημαντικό να εισαχθεί ο όρος του cloud computing.

Το cloud computing είναι η παροχή υπολογιστικών πόρων μέσω του διαδικτύου με βάση την αντίστοιχη χρήση. Έτσι λοιπόν αντί μια επιχείρηση να αγοράζει και να συντηρεί δικές τις υποδομές, επιλέγει έναν πάροχο ο οποίος προσφέρει τους ζητούμενους πόρους και τους κοστολογεί με βάση τις ακριβείς προδιαγραφές τους [8]. Το cloud computing αποτελεί αξιόλογη λύση τόσο για μικρές όσο και για μεσαίες ή και μεγαλύτερες εταιρείες, αφού τις αποδεσμεύει από την ευθύνη της συντήρησης τέτοιων υποδομών. Παράλληλα οι πάροχοι δημιουργούν μεγάλα data centers, με τα οποία εξυπηρετούν ταυτόχρονα σημαντικό αριθμό πελατών και είναι σε θέση να κάνουν πιο αποτελεσματικό διαμοιρασμό των πόρων, χωρίς να σπαταλούν άσκοπα υποδομές, όπως μπορεί να γινόταν στο πλαίσιο μίας μεμονωμένης εταιρίας. Ο τρόπος λειτουργίας αυτός τους δίνει τη δυνατότητα να προσφέρουν ιδιαίτερα ανταγωνιστικές τιμές.

Η ταχύτατη ανάπτυξη του cloud computing έχει βοηθήσει σημαντικά στην καθιέρωση της αρχιτεκτονικής microservices. Το βασικό προνόμιο που προσφέρουν τα microservices είναι η ανεξαρτησία και η απομόνωση των επιμέρους υπηρεσιών. Η ανεξαρτησία αυτή αφορά τόσο

στο γεγονός ότι μπορούν ξεχωριστά να υλοποιηθούν, να δοκιμαστούν και να εξελιχθούν όσο και στη δυνατότητα να στηθούν και να κλιμακωθούν με βάση μόνο τις δικές τους ανάγκες. Με τον τρόπο αυτό οι εταιρείες μπορούν να ζητούν και να πληρώνουν ακριβώς τους υπολογιστικούς πόρους που έχουν ανάγκη μειώνοντας έτσι τα συνολικά κόστη. Πέρα από αυτό το πολύ βασικό πλεονέκτημα, οι εφαρμογές microservices δίνουν τη δυνατότητα σε επιμέρους μικρότερες ομάδες να αναπτύξουν τα τμήματα της εφαρμογής με ελάχιστη ανάγκη για επικοινωνία μεταξύ τους, χαρακτηριστικό που ευνοεί τη σύγχρονη αντίληψη σχετικά με την ανάπτυξη λογισμικού (μικρές ομάδες, σύντομοι κύκλοι ανάπτυξης, ταχεία δημιουργία λειτουργικών κομματιών). Ένας άλλος καθοριστικός παράγοντας που οδήγησε στην υιοθέτηση τέτοιων αρχιτεκτονικών είναι πως η ανεξαρτησία των τμημάτων μειώνει τον κίνδυνο για συνολική δυσλειτουργία της εφαρμογής. Έτσι λοιπόν, ιδιαίτερα σε μεγάλες εφαρμογές, ακόμα και αν ένα τμήμα παρουσιάσει δυσλειτουργίες, δεν θα σταματήσει η λειτουργία όλης της εφαρμογής, πάρα μόνο όσων τμημάτων επηρεάζονται άμεσα ή έμμεσα από αυτό το προβληματικό κομμάτι. Τα χαρακτηριστικά αυτά και οι ανάγκες της αγοράς που έχουν προκαλέσει την τόσο ευρεία χρησιμοποίηση της αρχιτεκτονικής αυτής, έχουν δημιουργήσει και το πλαίσιο για να αναπτυχθεί πληθώρα νέων τεχνολογιών και εργαλείων που συνεισφέρει στην καλύτερη και ευκολότερη υλοποίησή τους. Η βασικότερη σχετικά νέα τεχνολογία από αυτές, σε ότι αφορά την ευκολότερη διαχείριση τέτοιων εφαρμογών, είναι η χρήση των εικονικών μηχανημάτων (virtual machines) και των κιβωτίων (containers).



Εικόνα 1. Monolithic και Microservices Αρχιτεκτονική [9]

2.2 Virtual Machines και Containers

Ένας βασικός προβληματισμός στη διαδικασία ανάπτυξης λογισμικού είναι η απόφαση σχετικά με τα χαρακτηριστικά των μηχανημάτων πάνω στα οποία πρόκειται να στηθεί μία

εφαρμογή. Συγκεκριμένα, το λειτουργικό σύστημα, τα εργαλεία και οι εκδόσεις τους καθώς και πολλές επιμέρους ρυθμίσεις παίζουν καθοριστικό ρόλο στη διαδικασία υλοποίησης της εφαρμογής. Για τον λόγο αυτό υπάρχει η ανάγκη για μια λύση που να μπορεί να προσφέρει τη δυνατότητα στους προγραμματιστές να επιλέγουν όλα τα χαρακτηριστικά που επιθυμούν χωρίς περιορισμούς αλλά και χωρίς να επιβάλλουν τις επιλογές τους σε ολόκληρο το φυσικό μηχάνημα που θα φιλοξενεί την εφαρμογή. Το πρόβλημα αυτό μπορεί να αντιμετωπιστεί επιτυχώς με δύο τεχνολογίες που έχουν καθοριστικό ρόλο στις σύγχρονες μεθόδους ανάπτυξης λογισμικού: τα εικονικά μηχανήματα (virtual machines) και τα κιβώτια (containers).

2.2.1 Virtual Machines

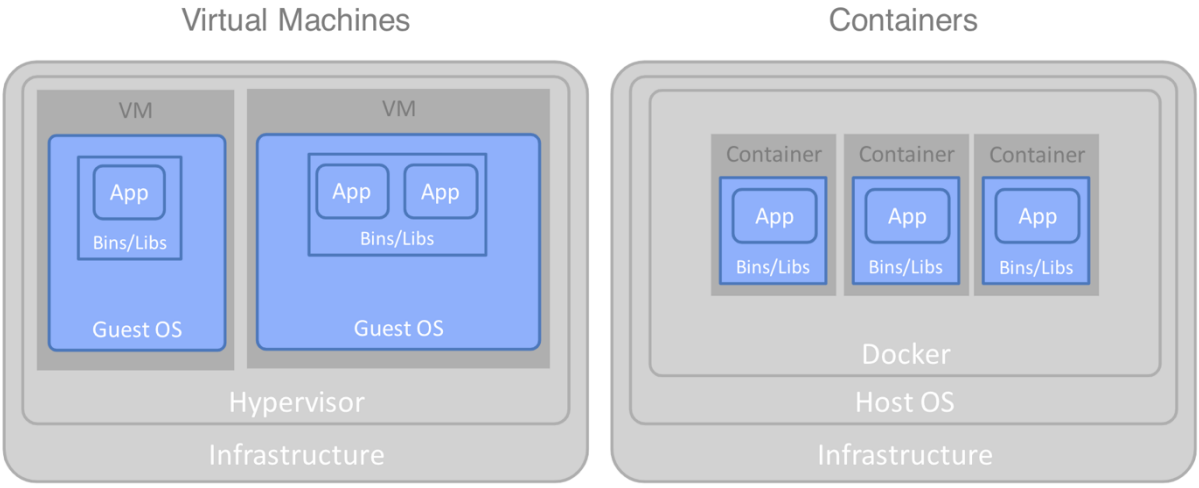
Τα virtual machines είναι υπολογιστικές μονάδες που χρησιμοποιούν λογισμικό αντί για φυσικούς υπολογιστές για την εκτέλεση προγραμμάτων και την εγκατάσταση εφαρμογών [10]. Με τον τρόπο αυτό σε ένα φυσικό μηχάνημα μπορούν να συστεγάζονται ένα ή περισσότερα virtual machines και να λειτουργούν παράλληλα. Κάθε ένα από τα μηχανήματα αυτά έχει το δικό του λειτουργικό σύστημα και πρακτικά λειτουργεί σαν ένας αυτόνομος και ανεξάρτητος φυσικός υπολογιστής. Με τον τρόπο αυτό δίνεται η δυνατότητα σε εξυπηρετητές (servers) να μπορούν να φιλοξενούν διαφορετικές εφαρμογές που τρέχουν σε διαφορετικά virtual machines.
Εκμεταλλευόμενοι την τεχνολογία αυτή οι προγραμματιστές έχουν την δυνατότητα να ορίζουν επακριβώς και να διαχειρίζονται όπως οι ίδιοι κρίνουν όλα τα στοιχεία ενός μηχανήματος που επηρεάζουν το στήσιμο μίας εφαρμογής. Το κρίσιμο στοιχείο μάλιστα είναι ότι μπορούν να το κάνουν αυτό χωρίς να δεσμεύουν ολόκληρο το μηχάνημα και να επιβάλλουν τις επιλογές τους, αφού οι αποφάσεις τους επηρεάζουν μόνο το δικό τους εικονικό μηχάνημα. Έτσι λοιπόν είναι συνήθης πρακτική πάνω στα φυσικά μηχανήματα να στήνονται εικονικά, μέσα σε καθένα από τα οποία μπορούν να τρέχουν οι διαφορετικές εφαρμογές. Αυτός είναι ο τρόπος που διαχειρίζονται τους υπολογιστικούς πόρους τόσο οι εταιρείες στα δικά τους μηχανήματα όσο και οι μεγάλοι cloud service providers που διαχειρίζονται τεράστιο πλήθος φυσικών μηχανημάτων πάνω στα οποία στήνουν virtual machines, που στην συνέχεια παρέχουν στους πελάτες τους. Τα εικονικά μηχανήματα όμως έχουν και ένα βασικό μειονέκτημα. Επειδή ουσιαστικά κάθε εικονικό μηχάνημα προσομοιάζει ένα φυσικό υπολογιστή έχοντας το δικό του λειτουργικό σύστημα και τις δικές του εφαρμογές, η διαδικασία του να στήσει κανείς ένα τέτοιο μηχάνημα εισάγει μια σημαντική επιβάρυνση στους υπολογιστικούς πόρους του αντίστοιχου server. Το πρόβλημα αυτό δεν είναι τόσο εμφανές όταν μιλάμε για μονολιθικές εφαρμογές διότι η διαδικασία ανέγερσης ενός virtual machine για κάθε εφαρμογή προσφέρει ανεξαρτησία και αυτονομία χωρίς ιδιαίτερα σημαντική επιβάρυνση των πόρων και συνεπώς αποτελεί μία ικανοποιητική λύση. Το πρόβλημα γίνεται ιδιαίτερα εμφανές όταν επιλέγονται αρχιτεκτονικές microservices και πρακτικά υλοποιούνται πολλές μικρές επιμέρους εφαρμογές κάθε μία με τις δικές της ανάγκες. Στην περίπτωση αυτή η δέσμευση ενός ολόκληρου εικονικού μηχανήματος για κάθε μία από τις -συνήθως μικρές- επιμέρους εφαρμογές ώστε να προσφέρει στους αντίστοιχους προγραμματιστές τη μέγιστη ελευθερία, δεν είναι ρεαλιστική, αφού η αθροιστική επιβάρυνση καθιστά το εγχείρημα ασύμφορο. Στο πρόβλημα αυτό έρχονται

να δώσουν λύση τα containers τα οποία, λειτουργώντας συμπληρωματικά με τα virtual machines, προσφέρουν τη βέλτιστη ισορροπία μεταξύ κόστους και ελευθερίας στην ανάπτυξη λογισμικού.

2.2.2 Images και Containers

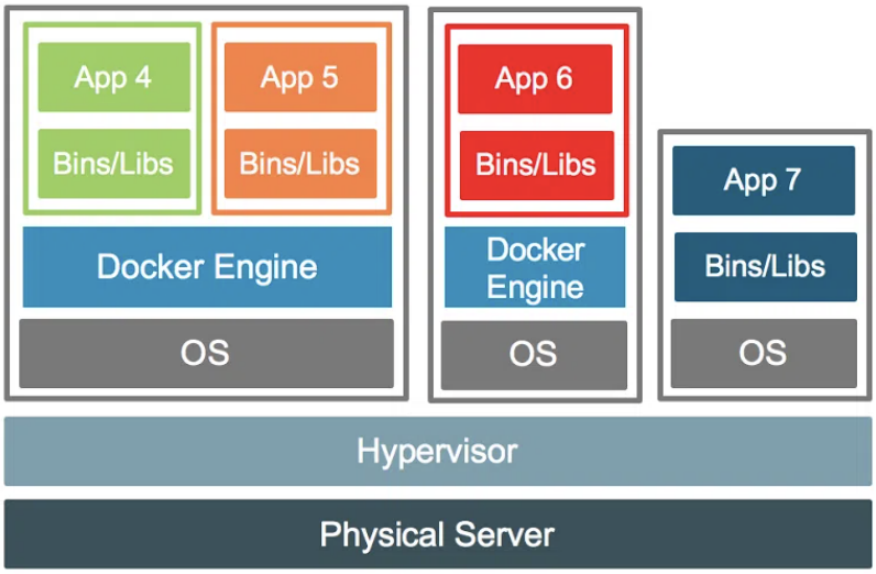
Τα containers είναι τυποποιημένες μονάδες λογισμικού που πακετάρουν τον κώδικα, τις εξαρτήσεις του και όλα τα απαραίτητα στοιχεία ώστε να μπορεί το λογισμικό αυτό να εκτελεστεί σε οποιοδήποτε περιβάλλον [11][12]. Μια εικόνα (image) είναι ένα ελαφρύ, αυτόνομο πακέτο λογισμικού που περιέχει όλη την απαραίτητη πληροφορία για την εκτέλεση μίας εφαρμογής. Η πληροφορία αυτή εμπεριέχει τον κώδικα, το περιβάλλον, τα εργαλεία του συστήματος και τις ρυθμίσεις. Από τα images δημιουργούνται τα containers κατά τη διάρκεια της εκτέλεσης, μέσα στα οποία τελικά θα τρέξει μια εφαρμογή. Τα containers προσφέρουν αυξημένη φορητότητα και συμβατότητα. Επιπλέον, διευκολύνουν την κλιμάκωση και την διαχείρισή τους ενώ παράλληλα προσφέρουν απομόνωση και μεγαλύτερη ασφάλεια. Τέλος, παρουσιάζουν σταθερή συμπεριφορά ανεξαρτήτως περιβάλλοντος, χωρίς παράλληλα να καταναλώνουν σημαντικούς υπολογιστικούς πόρους [13]. Για τη δημιουργία των images όσο και των αντίστοιχων containers απαιτείται ένα ενδιάμεσο λογισμικό που αναλαμβάνει τη συγκεκριμένη διαδικασία. Το σημαντικότερο τέτοιου είδους εργαλείο τη δεδομένη χρονική στιγμή είναι το Docker.

Το Docker είναι μία πλατφόρμα ανοιχτού κώδικα που αναλαμβάνει τόσο τη διαδικασία δημιουργίας των images όσο και τη δημιουργία, εκτέλεση, διαχείριση και ενημέρωση των containers. Ουσιαστικά το Docker λειτουργεί σαν ένα ενδιάμεσο στρώμα μεταξύ του λειτουργικού συστήματος και των containers που εξασφαλίζει ότι ανεξάρτητα της τοποθεσίας στην οποία τρέχουν τα containers, η συμπεριφορά τους θα είναι πάντα η ίδια. Παρακάτω φαίνεται η διαφορά μεταξύ των containers και των virtual machines:



Εικόνα 2. Virtual Machines vs Containers [14]

Όπως αναφέρθηκε και παραπάνω τα containers προσφέρουν παρόμοια επίπεδα αυτονομίας χωρίς όμως να εισάγουν σημαντικές επιβαρύνσεις σε ότι αφορά τις υπολογιστικές ανάγκες του συστήματος. Έτσι, η τελική εικόνα που προκύπτει συνδυάζει και τις δύο τεχνολογίες και έχει την παρακάτω μορφή:



Εικόνα 3. Complete architecture of server [15]

Ο φυσικός server υποστηρίζει πολλά διαφορετικά virtual machines, τα οποία παρέχει σε διαφορετικές ομάδες/εταιρείες/εφαρμογές... και μέσα σε κάθε ένα από αυτά τα εικονικά μηχανήματα, αν υπάρχει ανάγκη για περισσότερες από μία επιμέρους εφαρμογές, τρέχει το Docker και στη συνέχεια σηκώνονται τα αντίστοιχα containers. Έτσι μπορούν να υλοποιηθούν όλα τα services σε μία αρχιτεκτονική microservices με απόλυτη αυτονομία, χωρίς παράλληλα να απαιτούν σημαντικό αριθμό υπολογιστικών πόρων. Η ευρεία εφαρμογή της παραπάνω προσέγγισης, τη δεδομένη χρονική στιγμή, έχει δημιουργήσει μία μεγάλη νέα αγορά. Στο πλαίσιο της αγοράς αυτής δημιουργούνται διαρκώς εργαλεία που διευκολύνουν τις επιχειρήσεις να υλοποιούν τέτοιου είδους αρχιτεκτονικές που κάνουν χρήση των containers. Ένα από τα εργαλεία αυτά που έχει ως ρόλο τη γενική διαχείριση μιας συστοιχίας (cluster) τέτοιων containers που αποτελούν μία εφαρμογή είναι ο ενορχηστρωτής (Orchestrator).

2.3 Container Orchestration και Kubernetes

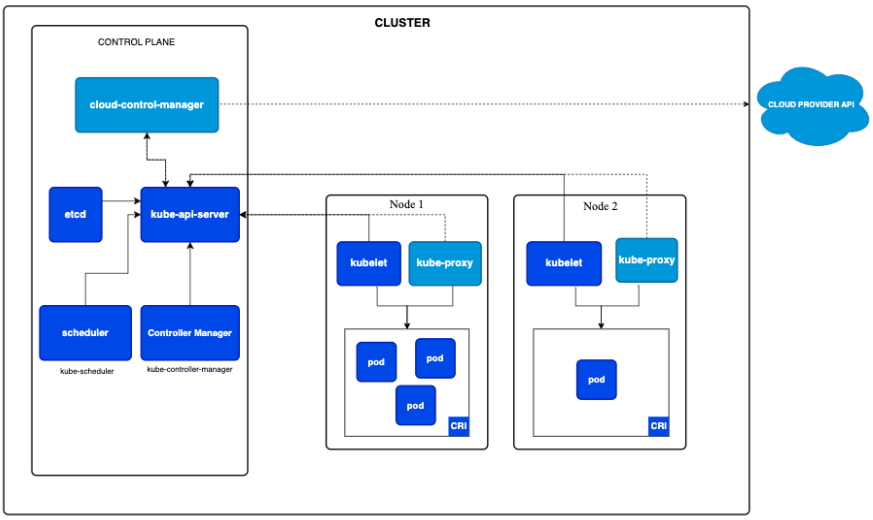
Όπως αναφέρθηκε και στα προηγούμενα κεφάλαια μεγάλος αριθμός εφαρμογών πλέον χτίζεται χρησιμοποιώντας αρχιτεκτονικές Microservices. Πιο συγκεκριμένα, το 85% των μεγάλων εταιρειών με προσωπικό από 5000 άτομα και πάνω ήδη χρησιμοποιεί αρχιτεκτονικές Microservices, ενώ για τις μεσαίες και μικρότερες εταιρείες το νούμερο αυτό είναι 75%-84% [6]. Η ανάπτυξη των επιμέρους services, η δημιουργία των αντίστοιχων containers και η

διαχείριση και η συντήρησή τους αποτελούν σημαντικό κομμάτι της διαδικασίας ανάπτυξης λογισμικού. Ένα από τα βασικότερα εργαλεία που έχει την γενικότερη εποπτεία των containers είναι ο ενορχηστρωτής κιβωτίων (container orchestrator). Ο Orchestrator είναι ένα εργαλείο που αναλαμβάνει την εγκατάσταση της εφαρμογής, την διαχείρισή της και την κλιμάκωση όταν αυτό κριθεί αναγκαίο.

Όταν γίνεται αναφορά σε container orchestration, η de facto επιλογή στην αγορά τη δεδομένη χρονική στιγμή είναι το Kubernetes. Το Kubernetes κατέχει το 77% της συγκεκριμένης αγοράς, ενώ το ποσοστό φτάνει στο 89% αν συμπεριλάβουμε τα Red Hat OpenShift και Rancher, τα οποία είναι βασισμένα στο Kubernetes [1]. Το Kubernetes είναι ένα εργαλείο container orchestration ανοιχτού κώδικα που σχεδιάστηκε αρχικά από την Google [16]. Οργανώνει τα containers τα οποία αποτελούν μία εφαρμογή σε λογικές μονάδες για ευκολότερη ανακάλυψη και διαχείριση. Οι βασικές λειτουργίες που προσφέρει είναι η ανακάλυψη υπηρεσιών στο πλαίσιο της εφαρμογής και η διαχείριση φόρτου, η διαχείριση του αποθηκευτικού χώρου καθώς και η αυτόματη ανάθεση πόρων στα containers με βάση τις ανάγκες τους. Τέλος, προσφέρει τη δυνατότητα διαχείρισης των containers και την αυτό-επούλωση του συστήματος, δηλαδή την αυτόματη επιστροφή στην ζητούμενη κατάσταση σε περίπτωση που δημιουργηθεί κάποιο πρόβλημα στα containers.

2.3.1 Αρχιτεκτονική

Το Kubernetes χρησιμοποιεί μία συγκεκριμένη αρχιτεκτονική για τη διαχείριση της συστοιχίας (cluster) και της εφαρμογής γενικότερα: τοποθετεί containers μέσα σε κάψουλες (pods) για να τρέξουν μέσα σε κόμβους (nodes). Τα nodes είναι οι διαφορετικές υπολογιστικές μονάδες που αποτελούν το cluster της εφαρμογής. Μπορεί να είναι τόσο φυσικοί υπολογιστές όσο και εικονικοί, ανάλογα με τις ανάγκες που υπάρχουν. Σε ένα cluster μπορεί να υπάρχει είτε ένα, είτε όπως γίνεται συνήθως, παραπάνω nodes. Μέσα στα nodes στήνονται τα pods, που είναι οι βασικές μονάδες του Kubernetes. Κάθε pod μπορεί να αποτελείται από ένα ή περισσότερα containers. Επιπλέον ένα σύστημα Kubernetes έχει πολλά διαφορετικά κομμάτια (components) το καθένα από τα οποία αναλαμβάνει ένα συγκεκριμένο τμήμα της διαχείρισης του cluster.



Εικόνα 4. Kubernetes Architecture

2.3.2 Components

Τα components του Kubernetes χωρίζονται σε τρεις κατηγορίες.

Control plane components:
Το control plane αναλαμβάνει τη διαχείριση των nodes και των pods του cluster. Αποτελείται από τα εξής components:

- kube-apiserver: Ρόλος του είναι να εκθέτει προς τα έξω το Kubernetes API το οποίο με τη σειρά του προσφέρει στους προγραμματιστές μια σειρά εντολών για τη διαχείριση του cluster.
- etcd: Συνεκτικός και υψηλά διαθέσιμος αποθηκευτικός χώρος που χρησιμοποιείται για την αποθήκευση όλων των δεδομένων του συστήματος Kubernetes.
- kube-scheduler: Component που αναζητεί νέα pods που μόλις έχουν δημιουργηθεί, και τα αναθέτει σε κάποιο node του συστήματος, λαμβάνοντας υπόψη τα δεδομένα που έχει στη διάθεσή του.
- kube-controller-manager: Αναλαμβάνει την εκτέλεση όλων των διαφορετικών controllers (node, pod, serviceAccount, ...) μαζί ώστε να μειώνεται η γενικότερη πολυπλοκότητα του συστήματος.
- Cloud-controller-manager: Δίνει τη δυνατότητα σύνδεσης με το API του αντίστοιχου cloud provider και διαχωρίζει τα components που αλληλεπιδρούν με το cloud platform από εκείνα που αλληλεπιδρούν μόνο με το cluster.

Node Components
Τα node components τρέχουν σε κάθε node, διαχειρίζονται τα pods που τρέχουν και δημιουργούν το Kubernetes runtime environment.

- kubelet: Διασφαλίζει ότι τρέχουν τα pods σε κάθε node.
- kube-proxy: Είναι ένας δικτυακός (network) διαμεσολαβητής (proxy). Διατηρεί κανόνες δικτύου στα nodes και επιτρέπει την επικοινωνία με τα pods τόσο από μέσα όσο και από έξω από το cluster.
- Container runtime: Αναλαμβάνει τη διαχείριση της εκτέλεσης και του κύκλου ζωής των containers εντός του περιβάλλοντος Kubernetes.

Addons
Πέρα από τα βασικά components που αναφέρθηκαν παραπάνω υπάρχουν και πολλά επιπλέον που χρησιμοποιούν πόρους του cluster και προσφέρουν σημαντικές λειτουργικότητες. Κάποια από τα σημαντικότερα είναι τα παρακάτω:

- DNS
- Web UI (Dashboard)
- Container Resource Monitoring
- Cluster-level Logging
- Network Plugins

Ενώ υπάρχουν και πολλά άλλα που προσφέρουν δυνατότητες που μπορεί να είναι χρήσιμες σε συγκεκριμένα σενάρια ανάπτυξης λογισμικού.

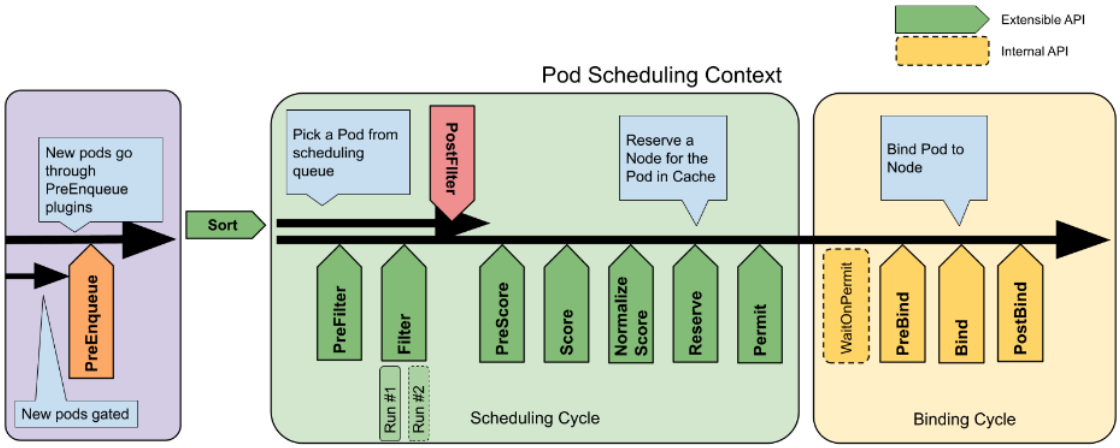
Η παρούσα διπλωματική εργασία αφορά στο component του kube-scheduler που βρίσκεται στο control plane όπως αναφέρθηκε παραπάνω. Για το λόγο αυτό στη συνέχεια θα ασχοληθούμε αναλυτικότερα με το συγκεκριμένο βασικό εργαλείο του περιβάλλοντος Kubernetes.

ΚΕΦΑΛΑΙΟ 3 – Kubernetes Scheduler και Υφιστάμενη Δουλειά

Ο χρονοδρομολογητής (scheduler) είναι ένα από τα βασικά components του περιβάλλοντος Kubernetes που έχει ως ρόλο την αναζήτηση καινούργιων pods που μόλις έχουν δημιουργηθεί και τη δρομολόγησή ρους στο κατάλληλο node του cluster. Για να το πετύχει αυτό με αποδοτικό τρόπο, χρησιμοποιεί τα διαθέσιμα δεδομένα που του παρέχει το περιβάλλον του Kubernetes. Η αξία της αποδοτικής δρομολόγησης είναι ιδιαίτερα υψηλή, αφού μπορεί να συνεισφέρει στη βελτίωση της συνολικής απόδοσης του συστήματος προς οποιαδήποτε κατεύθυνση κρίνεται σημαντική για την εκάστοτε περίπτωση. Έτσι λοιπόν ο scheduler μπορεί να δώσει έμφαση στην ταχύτερη απόκριση του συστήματος, στη μείωση του κόστους και των πόρων που απαιτούνται ή σε οτιδήποτε άλλο κρίνεται σημαντικό στο πλαίσιο ανάπτυξης μιας συγκεκριμένης εφαρμογής. Το Kubernetes έχει τον δικό του δρομολογητή που ονομάζεται kube-scheduler και είναι κομμάτι του control-plane. Ο συγκεκριμένος scheduler θα αναλυθεί πλήρως στη συνέχεια τόσο για τις δυνατότητες που παρέχει όσο και για τα προβλήματα και τους περιορισμούς του. Επειδή όμως η επιρροή των αποφάσεων δρομολόγησης στη συνολική απόδοση του συστήματος είναι ιδιαίτερα σημαντική, ο default scheduler του Kubernetes δίνει τη δυνατότητα επεκτάσεων και την προσθήκη νέων λειτουργιών που τον καθιστούν ακόμα πιο αποτελεσματικό. Επιπλέον, τη διαδικασία της χρονοδρομολόγησης μπορεί να την αναλάβει και ένα εξωτερικός scheduler ο οποίος, εκμεταλλευόμενος το Kubernetes API, μιμείται τον τρόπο λειτουργίας του default και εισάγει νέες δυνατότητες στο σύστημα. Η ραγδαία εξέλιξη και υιοθέτηση του Kubernetes από την αγορά έχει οδηγήσει στην παρουσίαση βελτιωμένων εκδόσεων του scheduler, που επιχειρούν να επιτύχουν αποδοτικότερη δρομολόγηση. Παρακάτω θα παρουσιαστεί ο default scheduler καθώς και ο scheduler που προτείνεται στη δημοσίευση NetMARKS [2], ο οποίος βελτιώνει σημαντικά την απόδοση της εφαρμογής στην οποία επιλέγεται.

3.1 Kube-scheduler

Ο kube-scheduler είναι το προεπιλεγμένο εργαλείο που αναλαμβάνει τη χρονοδρομολόγηση σε ένα περιβάλλον Kubernetes αν δεν έχουν γίνει ειδικές ρυθμίσεις. Όπως αναφέρθηκε παραπάνω, ο ρόλος του είναι να ελέγχει συνεχώς για καινούρια pods και, όταν τα βρίσκει, να επιλέγει τον κατάλληλο node για να τα δρομολογήσει. Ολόκληρη η διαδικασία της δρομολόγησης ονομάζεται Scheduling Framework και χωρίζεται σε τρία βασικά στάδια, όπως φαίνεται παρακάτω.



Εικόνα 5. Kubernetes Scheduling Framework [17]

Το πρώτο στάδιο είναι το PreEnqueue, το οποίο έχει ως ρόλο να βάζει τα νέα pods σε μία ουρά έτσι ώστε να δρομολογηθούν.

Το δεύτερο στάδιο αφορά τη διαδικασία επιλογής του κατάλληλου node. Χωρίζεται σε δύο βασικά τμήματα, καθένα από τα οποία χωρίζεται σε επιμέρους μικρότερα τμήματα. Οι δύο κύριες διεργασίες που γίνονται στο στάδιο αυτό είναι το φιλτράρισμα (filtering) και η βαθμολόγηση (scoring). Το filtering επιλέγει από όλα τα nodes του συστήματος εκείνα στα οποία είναι δυνατό να δρομολογηθεί το νέο pod, ενώ το scoring βαθμολογεί όλα τα πιθανά nodes που έχουν προκύψει από το στάδιο του filtering έτσι ώστε εν τέλει να επιλεχθεί το ιδανικότερο. Το στάδιο αυτό εκτελείται σειριακά, δηλαδή για να ξεκινήσει η διαδικασία επιλογής για ένα pod, πρέπει να ολοκληρωθεί η διαδικασία για το ακριβώς προηγούμενο της ουράς.

Το τρίτο και τελευταίο στάδιο είναι αυτό του binding, δηλαδή της δρομολόγησης ουσιαστικά του pod στο επιλεγμένο node. Σε αντίθεση με το προηγούμενο στάδιο, το binding δεν είναι σειριακό και συνεπώς μπορεί παράλληλα να διαχειριστεί πολλά pods.

Όπως αναφέρθηκε και στην εισαγωγή, ο kube-scheduler δίνει τη δυνατότητα για επεκτάσεις. Για τον λόγο αυτό κάθε ένα από τα βασικά στάδια που αναλύθηκαν παραπάνω χωρίζεται σε επιμέρους μικρότερα. Με τον τρόπο αυτό δίνεται η δυνατότητα στους προγραμματιστές να κάνουν πολύ στοχευμένες τροποποιήσεις και να διαμορφώνουν τη τελική λειτουργία του scheduler με βάση τις δικές τους ιδιαίτερες ανάγκες. Τις αλλαγές αυτές μπορούν να τις επιτύχουν με δύο διαφορετικούς τρόπους. Ο πιο απλός είναι η χρήση των scheduling policies, στα οποία ουσιαστικά παρέχουν κριτήρια για το στάδιο του φιλτραρίσματος και προτεραιότητες για το στάδιο της βαθμολόγησης. Ο δεύτερος και πιο σύνθετος τρόπος που όμως προσφέρει και περισσότερες δυνατότητες, είναι η χρήση των scheduling profiles. Με τον τρόπο αυτό δίνεται η δυνατότητα υλοποίησης plugins με συγκεκριμένο τρόπο λειτουργίας σε οποιοδήποτε από τα μικρότερα στάδια που απεικονίζονται στην εικόνα 5. Σημειώνεται επίσης πως υπάρχει η δυνατότητα ο kube-scheduler να ρυθμιστεί με τρόπο τέτοιο ώστε να μπορεί να χρησιμοποιεί και διαφορετικά scheduling profiles.

Ο kube scheduler είναι μια καλή αρχική λύση για τη δρομολόγηση των pods αλλά πολύ γρήγορα γίνονται εμφανείς οι αδυναμίες του. Το κύριο πρόβλημα έγκειται στο ότι δεν έχει

αρκετά δεδομένα σχετικά με τη λειτουργία της εφαρμογής ώστε να κάνει ιδιαίτερα αποδοτικές επιλογές δρομολόγησης. Οι βασικές πληροφορίες τις οποίες μπορεί να εκμεταλλευτεί αφορούν τα στατικά χαρακτηριστικά των nodes και των pods. Συγκεκριμένα, για τις επιλογές του χρησιμοποιεί τις ανάγκες για υπολογιστικούς πόρους που έχουν δηλωθεί για τα pods και τις αντίστοιχες δυνατότητες των nodes, ενώ παράλληλα λαμβάνει υπόψη και άλλα στατικά δεδομένα που μπορεί να παρέχουν οι προγραμματιστές στις οδηγίες δημιουργίας των pods. Τέτοια δεδομένα μπορεί να είναι το node affinity (προτίμηση ή όχι για συγκεκριμένα nodes), pod affinity (προτίμηση ή όχι για συνύπαρξη pods στο ίδιο node), topology constraints (περιορισμούς που προκύπτουν από τη φυσική ή λογική τοπολογία του cluster) καθώς και άλλα. Όλα αυτά τα δεδομένα, αν και ιδιαίτερα χρήσιμα, έχουν τον ίδιο περιορισμό: είναι στατικά και δεν λαμβάνουν υπόψη τη δυναμική συμπεριφορά της εκάστοτε εφαρμογής. Τα δεδομένα αυτά δεν περιέχουν πληροφορία σχετικά με την πραγματική χρησιμοποίηση του κάθε pod και τις αντίστοιχες ανάγκες του καθώς και με το βαθμό αλληλεπίδρασης και επικοινωνίας μεταξύ των pods. Ο λόγος είναι ότι τέτοιου είδους πληροφορίες προκύπτουν κατά τη διάρκεια ζωής της εφαρμογής, οπότε δεν είναι εύκολο εκ των προτέρων να τις γνωρίζει κανείς. Το πρόβλημα είναι ότι τα δεδομένα αυτά είναι που έχουν τη μεγαλύτερη βαρύτητα για το πόσο αποδοτικές μπορεί να είναι οι επιλογές της δρομολόγησης. Για τον λόγο αυτό είναι συνήθης πρακτική να μην χρησιμοποιείται ο απλός kube-scheduler αλλά κάποια βελτιωμένη έκδοση είτε με τη χρήση των μεθόδων επέκτασης που προσφερει ο kube-scheduler ή με τη δημιουργία εξ' ολοκλήρου καινούριων λύσεων. Μια ενδιαφέρουσα τέτοια προσέγγιση που επιτυγχάνει σημαντική βελτίωση παρουσιάζεται στη δημοσίευση NetMARKS.

3.2 NetMARKS

Το NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh είναι μία πρόταση που έρχεται να δώσει λύση σε κάποια από τα προβλήματα που προκύπτουν από την περιορισμένη πρόσβαση του kube-scheduler σε χρήσιμα για τη δρομολόγηση δεδομένα. Για να το πετύχει αυτό συλλεγει δυναμικές μετρικές δικτύου σχετικά με την εφαρμογή και στη συνέχεια τις χρησιμοποιεί για να βελτιώσει τις αποφάσεις του. Τα δεδομένα συλλέγονται με τη χρήση του Istio Service Mesh [18], ένα εργαλείο που εισάγει ένα καινούριο component πριν από κάθε pod του cluster και μέσα από το οποίο περνάει όλη η δικτυακή κίνηση. Η χρησιμοποίηση του Istio δίνει επιπλέον τη δυνατότητα για καταγραφή των δεδομένων των κινήσεων που γίνονται μέσα στο cluster μέσω των αντίστοιχων components. Τα δεδομένα αυτά προσφέρονται μέσω μετρικών στον προγραμματιστή δίνοντάς του μια καλύτερη εικόνα για τον δυναμικό τρόπο λειτουργίας της εφαρμογής (Το Istio θα αναλυθεί περαιτέρω σε μεταγενέστερο κεφάλαιο). Στο πλαίσιο της δημοσίευσης αυτής, με βάση τα δεδομένα που προσφέρει το Istio, παρουσιάζεται μια εναλλακτική πρόταση scheduler, η οποία είναι χτισμένη πάνω στο kube-scheduler κάνοντας χρήση των scheduling profiles που αναφέρθηκαν παραπάνω. Η πρόταση αυτή χρησιμοποιεί το Prometheus [19], ένα εργαλείο που δίνει, μεταξύ άλλων, τη δυνατότητα για ερωτήματα (queries) πάνω σε δεδομένα όπως αυτά του Istio (Το Prometheus θα αναλυθεί περεταίρω σε μεταγενέστερο κεφάλαιο), και συλλέγει δύο συγκεκριμένες μετρικές: τις istio_request_bytes_sum and istio_response_bytes_sum, που

περιέχουν τον αριθμό των bytes που μεταφέρονται τόσο στα requests όσο και στα responses μεταξύ των pods του cluster. Στη συνέχεια τα δεδομένα αυτά χρησιμοποιούνται για να υπολογιστούν οι μέσοι ρυθμοί μετάδοσης πληροφορίας μεταξύ των pods.

Πιο αναλυτικά, προτείνεται ένας αλγόριθμος ο οποίος υπολογίζει για κάθε νέο pod που επιχειρείται να δρομολογηθεί και με βάση τις μετρικές που έχουν ήδη συλλεχθεί, το βαθμό επικοινωνίας που έχει με κάθε node του cluster, αθροίζοντας δηλαδή τους αντίστοιχους βαθμούς επικοινωνίας με κάθε ένα από τα pods του node αυτού. Στη συνέχεια επιλέγει να τοποθετήσει το pod αυτό στο node το οποίο είχε το υψηλότερο σκορ. Η λογική είναι ότι με την επιλογή αυτή ελαχιστοποιούνται οι επικοινωνίες μεταξύ των διαφορετικών nodes της εφαρμογής και άρα μειώνονται και οι αντίστοιχες χρονικές επιβαρύνσεις που εισάγουν τέτοιου είδους επικοινωνίες. Αντιθέτως, μεγιστοποιούνται οι εσωτερικές επικοινωνίες σε κάθε υπολογιστική μονάδα της συστοιχίας που αποδεδειγμένα είναι πιο γρήγορες και χρονικά αποδοτικές. Έτσι λοιπόν, για να λειτουργήσει η προτεινόμενη λύση, στήνεται αρχικά η εφαρμογή για παράδειγμα με χρήση του default kube-scheduler και τίθεται σε λειτουργία για κάποιο χρονικό διάστημα έτσι ώστε να συλλεχθούν αρκετά δεδομένα σχετικά με τον τρόπο λειτουργία της. Στη συνέχεια καταργείται η εφαρμογή και στήνεται εκ νέου, τοποθετώντας κάθε pod στον ιδανικό κόμβο του συστήματος.

Στο πλαίσιο της δημοσίευσης αυτής, το πρόβλημα της αποδοτικής δρομολόγησης μελετήθηκε για το σενάριο χρήσης των τηλεπικοινωνιακών δικτύων. Για το πειραματικό στάδιο χρησιμοποιήθηκε ένα cluster πέντε πανομοιότυπων υπολογιστών ενώ αναπτύχθηκε και μία εφαρμογή η οποία αναδεικνύει με αποτελεσματικότερο τρόπο τη βελτίωση της προσέγγισης χρησιμοποιώντας μεγάλο αριθμό requests μεταξύ των pods. Παρακάτω φαίνονται τα αποτελέσματα της πειραματικής διαδικασίας που ακολουθήθηκε:

MEAN APPLICATION RESPONSE TIME

Prot.	Size	Sche.	log	board	beer	pig	flour	boat	meat	bread	gold	irono.	coal	iron	coin	tools	sword
gRPC	16	Def.	10.2	15.7	14.9	15.1	9.8	21.3	20.6	20.4	53.1	53.3	51.1	112	111	134	170
		NetM.	9.7	14.7	14.7	14.9	9.5	20.2	20.4	19.9	51.7	51.0	51.4	106	106	130	162
	256	Def.	10.3	15.7	15.2	15.2	9.7	21.5	20.8	21.2	53.3	52.8	50.2	112	112	134	167
		NetM.	9.7	14.4	14.8	14.9	9.6	20.1	20.3	20.3	51.7	51.3	50.4	108	105	130	164
	4k	Def.	10.8	16.6	16.3	16.1	10.3	22.4	21.6	22.0	54.4	55.8	53.5	117	117	139	178
		NetM.	10.1	15.3	15.2	15.6	10.1	20.7	21.2	21.1	53.3	53.6	53.3	112	112	133	168
	64k	Def.	19.4	28.9	28.7	28.5	18.7	38.7	38.4	37.5	94.3	97.7	95.0	202	203	243	309
		NetM.	18.3	25.9	26.6	28.0	18.5	33.6	36.6	30.1	93.1	92.4	91.4	192	181	233	289
	1M	Def.	102	110	131	134	103	172	176	182	441	445	452	933	913	1080	1409
		NetM.	55.7	84.8	95.7	91.6	61.5	115	151	138	351	334	346	711	684	805	1068
HTTP	16	Def.	9.9	14.0	15.1	14.9	10.0	19.6	20.2	20.4	51.7	52.3	52.2	109	110	130	168
		NetM.	9.6	14.1	14.0	13.6	9.5	18.5	19.2	19.4	49.1	49.2	49.9	105	103	125	158
	256	Def.	9.8	13.8	15.1	15.1	10.1	19.8	20.3	20.3	51.7	52.5	52.1	110	109	131	168
		NetM.	9.5	14.6	13.9	13.5	9.7	18.6	19.5	19.3	49.2	49.7	49.9	105	104	125	159
	4k	Def.	8.6	12.5	13.0	13.1	8.7	17.5	18.2	17.7	45.7	45.4	45.8	97.4	96.7	115	147
		NetM.	8.2	12.7	11.9	11.7	8.3	16.4	17.4	16.7	42.7	43.1	43.3	91.8	90.1	109	137
	64k	Def.	14.8	21.9	22.9	23.5	16.2	30.9	32.9	31.7	81.1	79.5	80.8	172	172	204	258
		NetM.	11.8	22.0	18.0	14.5	14.5	22.0	30.4	28.5	75.8	76.8	75.9	160	157	194	241
	1M	Def.	42.8	93.5	96.0	115	79.0	137	143	156	350	340	350	723	722	848	1115
		NetM.	28.2	40.0	47.5	45.6	28.7	54.9	117	108	283	278	283	586	586	680	896

Πίνακας 1. NetMARKS πίνακας αποτελεσμάτων με τους χρόνους απόκρισης [2]

Οι στήλες παρουσιάζουν τον χρόνο απόκρισης για τα διαφορετικά requests στο πλαίσιο της εφαρμογής. Οι συντάκτες της δημοσίευσης κατέληξαν ότι ακόμα και για μία εφαρμογή με λίγα services η προσέγγιση τους μπορούσε να μειώσει το χρόνο απόκρισης μέχρι και κατά 30%, αποτέλεσμα που έχει μεγάλη αξία.

Η προσέγγιση του NetMARKS, παρόλο που παρουσιάζει πολύ καλά αποτελέσματα, έχει και συγκεκριμένους περιορισμούς. Αρχικά, απαιτεί το να τρέξει η εφαρμογή σε πρώτο χρόνο ώστε

να συλλεχθούν οι κατάλληλες μετρικές σχετικά με την ιδανική δρομολόγηση και στη συνέχεια εκ νέου να στηθεί ολόκληρη η εφαρμογή. Επιπλέον, η προσέγγιση είναι άπληστη που σημαίνει ότι επιβαρύνει σε υψηλότερο βαθμό συγκεκριμένους κόμβους από άλλους. Το γεγονός αυτό, ενώ μπορεί να οδηγεί σε καλύτερα αποτελέσματα σε ότι αφορά το χρόνο απόκρισης της εφαρμογής, πιθανώς να αυξάνει το κόστος για την υποστήριξή της, αφού οι κόμβοι με τις αυξημένες ανάγκες πιο εύκολα αποτελούν σημείο συμφόρησης (bottleneck) και έχουν αυξημένες ανάγκες για υπολογιστικούς πόρους. Τέλος, η διαδικασία αυτή δεν είναι δυναμική, αφού από τη στιγμή που τα pods δρομολογηθούν στα αντίστοιχα nodes, δεν ελέγχεται κατά πόσο η τοπολογία που δημιουργούν είναι η βέλτιστη για όλη τη διάρκεια ζωής της εφαρμογής. Η δυναμική όμως αλληλεπίδραση των χρηστών με μία εφαρμογή καθώς και οι αλλαγές στις οποίες μπορεί να προβαίνουν οι προγραμματιστές κατά τη διάρκεια ζωής της ίσως να καθιστούν τις αρχικές επιλογές δρομολόγησης μη βέλτιστες.

Οι εμφανείς περιορισμοί του kube-scheduler σε συνδυασμό με τα καλά αποτελέσματα αλλά και κάποιους περιορισμούς της πρότασης του NetMARKS οδήγησαν, στο πλαίσιο της διπλωματικής εργασίας αυτής, στη δημιουργία μιας λύσης που επιχειρεί να προσφέρει μια πιο ολοκληρωμένη πρόταση αποδοτικής δρομολόγησης.

ΚΕΦΑΛΑΙΟ 4 – Προτεινόμενη Αρχιτεκτονική

Η λύση που σχεδιάστηκε και αναπτύχθηκε στο πλαίσιο της παρούσας διπλωματικής επιχειρεί να προσφέρει μια διαφορετική προσέγγιση σε σχέση με τις υπάρχουσες ως προς τη διαδικασία της δρομολόγησης. Στόχος είναι να βελτιστοποιήσει την απόδοση της εκάστοτε εφαρμογής σε ότι αφορά τον χρόνο απόκρισης, χωρίς ωστόσο να επιβαρύνει δυσανάλογα συγκεκριμένους κόμβους του cluster. Πέρα, όμως, από την αρχική επιλογή της βέλτιστης τοπολογίας, η λύση αυτή αναλαμβάνει να παρακολουθεί διαρκώς το σύστημα και να προχωράει στις κατάλληλες διορθωτικές κινήσεις έτσι ώστε να διατηρεί τη βέλτιστη κατάσταση καθ' όλη τη διάρκεια ζωής του, χωρίς να θυσιάζει τη διαθεσιμότητά του.

4.1 Γενική Ιδέα

Η προτεινόμενη αρχιτεκτονική σχεδιάστηκε και υλοποιήθηκε γύρω από τις εξής τέσσερις βασικές ιδέες:

1. Μείωση του αριθμού των requests μεταξύ pods που βρίσκονται σε διαφορετικά nodes
2. Συμμετρική, σε γενικές γραμμές, επιβάρυνση των κόμβων του συστήματος, έτσι ώστε να αποφεύγεται η δυσανάλογη ανάγκη για υπολογιστικούς πόρους σε κάποιον από αυτούς
3. Δυναμική διατήρηση της βέλτιστης λύσης σε όλη τη διάρκεια ζωής της εφαρμογής
4. Συνεχής διαθεσιμότητα της εφαρμογής

Η ιδέα για τη μείωση του αριθμού των requests μεταξύ διαφορετικών κόμβων κρίθηκε ιδιαίτερα σημαντική, διότι οι επικοινωνίες αυτές εισάγουν τις μεγαλύτερες καθυστερήσεις και συνεπώς επιβαρύνουν περισσότερο τον χρόνο απόκρισης της εφαρμογής. Η επικοινωνία μεταξύ pods που βρίσκονται στο ίδιο μηχάνημα θα είναι πάντα η πιο γρήγορη. Επιπλέον όμως, στα σύγχρονα cluster που είναι στημένα σε cloud providers, πολλές φορές δεν μπορούμε να γνωρίζουμε πόσο δικτυακά κοντά ή μακριά βρίσκονται δύο nodes, και συνεπώς πόση καθυστέρηση μπορεί να εισάγει η μεταξύ τους επικοινωνία. Έτσι λοιπόν, είναι κρίσιμο να μπορέσουμε να μειώσουμε τέτοιου είδους επικοινωνίες. Από την άλλη πλευρά, η προφανής λύση για την επίτευξη του στόχου αυτού θα ήταν η δρομολόγηση όλων των pods στον ίδιο κόμβο. Κάτι τέτοιο όμως δεν θα ήταν ιδανικό. Η ασύμμετρη επιβάρυνσή ενός ή περισσότερων κόμβων δημιουργεί bottlenecks σχετικά με την απόδοση του συστήματος και παράλληλα μεγαλύτερη ανάγκη για υπολογιστικούς πόρους στους συγκεκριμένους κόμβους. Η προσέγγιση αυτή δεν συμφωνεί με τη γενικότερη αρχή των αρχιτεκτονικών Microservices για μικρές και μοιρασμένες υπηρεσίες [20] ενώ αυξάνει τον κίνδυνο για γενικότερα προβλήματα στη χρήση των εφαρμογών αν οι υπερφορτωμένοι κόμβοι αντιμετωπίσουν οποιοδήποτε τεχνικό ζήτημα. Πέρα όμως από την αρχική βέλτιστη δρομολόγηση, όπως αναφέρθηκε και στα αρνητικά της πρότασης του NetMARKS, είναι κρίσιμο να λάβουμε υπόψιν ότι οι εφαρμογές μπορεί να μην παρουσιάζουν παρόμοια συμπεριφορά σε όλη τη διάρκεια ζωή τους. Αναβαθμίσεις και αλλαγές από την πλευρά των προγραμματιστών αλλά και αλλαγές στον τρόπο χρήσης της εφαρμογής από τους χρήστες μπορούν να αλλάξουν τα δεδομένα και να

δημιουργήσουν ανάγκη για τροποποίηση της αρχικής λύσης. Σε ό,τι αφορά την τέταρτη ιδέα, η συνεχής διαθεσιμότητα αποτελεί προτεραιότητα για μεγάλο ποσοστό των εφαρμογών στην αγορά. Έτσι λοιπόν οποιαδήποτε διορθωτική κίνηση επιλεχθεί να γίνει, πρέπει να γίνει με τρόπο τέτοιο ώστε η εφαρμογή να συνεχίσει αδιάλειπτα τη λειτουργία της.

4.2 Επιλογή Μετρικών και Αρχική Κατάσταση

Όπως και στην περίπτωση του NetMARKS, επιλέχθηκαν μετρικές οι οποίες παρέχουν πληροφορία σχετικά με την επικοινωνία μεταξύ των pods του cluster. Με τη χρήση των μετρικών αυτών επιτεύχθηκε η βέλτιστη ομαδοποίηση των κόμβων με στόχο την ελαχιστοποίηση των επικοινωνιών μεταξύ διαφορετικών κόμβων. Συγκεκριμένα οι μετρικές αυτές δίνουν τον αριθμό των requests ανά δευτερόλεπτο. Στη συνέχεια χρειάστηκε να στηθεί η εφαρμογή με μία αρχική τοπολογία και να μπει σε λειτουργία έτσι ώστε να συλλεχθούν οι μετρικές και να ληφθεί η απόφαση. Η κρίσιμη αλλαγή που προσφέρει η δυναμική προσέγγιση της εργασίας αυτής είναι ότι, αφού περάσει ικανοποιητικό διάστημα και μπορεί πλέον να προκύψει η βέλτιστη τοπολογία, η εφαρμογή μπορεί να μεταβεί αυτόματα στην κατάσταση αυτή, χωρίς να χρειάζεται να υπάρξει χρόνος αδράνειας (downtime). Για την επίτευξη του στόχου αυτού εκμεταλλευόμαστε τη δυνατότητα του Kubernetes για αύξηση και μείωση των pods ενός deployment χωρίς downtime [21]. Σε ό,τι αφορά την αύξηση των pods το Kubernetes επιτυγχάνει τον τρόπο λειτουργίας αυτό με το να εξασφαλίζει πως τα νέα pods είναι ετοιμα πριν προωθήσει κίνηση σε αυτά, ενώ στην περίπτωση της μείωσης των pods σταματάει να δρομολογεί κίνηση σε όσα πρόκειται να διαγραφούν αλλά τα διαγράφει αφού έχουν ολοκληρώσει όλα τα ενεργά requests που έχουν στη διάθεσή τους.

4.3 Αλγόριθμος ομαδοποίησης

Με τη χρήση των μετρικών που αναφέρθηκαν παραπάνω δημιουργείται ένας γράφος. Στον γράφο αυτό οι κόμβοι αποτελούν τα pods και οι ακμές τους αντίστοιχους ρυθμούς παραγωγής requests. Ο αλγόριθμος που υλοποιήθηκε έχει στόχο να κάνει έναν διαμερισμό (partitioning) του γράφου σε υποομάδες με στόχο τη μείωση των ακμών μεταξύ των ομάδων. Επιπλέον, δίνει τη δυνατότητα στον προγραμματιστή να επιλέγει πόσο άπληστη θέλει να είναι η λύση. Με βάση το μέγεθος του cluster ο αλγόριθμος μπορεί να δεχτεί ως είσοδο και το μέγιστο αριθμό των pods που μπορούν να δρομολογηθούν σε κάθε node. Έτσι λοιπόν αν κάποιος προτιμάει να δώσει έμφαση στην ομοιόμορφη χρησιμοποίηση των κόμβων, μπορεί εύκολα να το πετύχει, ενώ αντίστοιχα μπορεί κανείς να επιλέξει να πετύχει τη βέλτιστη απόδοση επιβαρύνοντας ασύμμετρα έναν ή παραπάνω κόμβους του cluster. Για παράδειγμα σε ένα cluster με τρεις κόμβους και έντεκα pods η πιο ομοιόμορφη κατανομή θα μπορούσε να επιτευχθεί με την επιλογή των τεσσάρων pods ώς το μέγιστο όριο δρομολόγησης. Με την επιλογή αυτή τελικά στα δύο nodes θα ομαδοποιούνταν τέσσερα pods και στο τελευταίο τρία pods. Αντίθετα, αν το όριο επιλεγόταν να είναι τα έντεκα pods ανά node η ομαδοποίηση, ανάλογα και με τις επικοινωνίες μεταξύ των pods, θα μπορούσε να δώσει σαν αποτέλεσμα την ομαδοποίηση και των έντεκα pods σε έναν μόνο από τους τρεις κόμβους αφήνοντας τους άλλους δύο εντελώς

ανεκμετάλευτους. Ο αλγόριθμος που υλοποιήθηκε εγγυάται πως σε κάθε περίπτωση η λύση που προκύπτει είναι η βέλτιστη με βάση τις επιλογές που έχει κάνει ο εκάστοτε χρήστης.

4.4 Συνεχής Παρακολούθηση και διορθωτικές κινήσεις

Η προτεινόμενη λύση λειτουργεί με παρόμοιο τρόπο όπως ο default scheduler, παρακολουθώντας διαρκώς το σύστημα για καινούρια pods που μπορεί να έχουν δημιουργηθεί. Για κάθε νέο pod επιλέγει με βάση τη βέλτιστη τοπολογία που έχει επιλεγεί ποιος είναι ο κατάλληλος κόμβος για να το τοποθετήσει. Αν δεν υπάρχουν δεδομένα ή πρόκειται για καινούριο service, η επιλογή γίνεται τυχαία με βάση τις υπολογιστικές ανάγκες του και τις αντίστοιχες δυνατότητες του κάθε κόμβου.
Πέρα όμως από τον συνεχή έλεγχο για καινούρια pods, στο παρασκήνιο ο χρονοδρομολογητής αυτός συλλέγει ανά κάποιο προκαθορισμένο χρονικό διάστημα τις μετρικές που συζητήθηκαν παραπάνω. Με βάση τις μετρικές αυτές τρέχει τον αλγόριθμο από τον οποίο προκύπτει η νέα βέλτιστη τοπολογία. Η τοπολογία αυτή αποτελεί τον χάρτη για τη δρομολόγηση όσων νέων pods δημιουργούνται από εκείνο το χρονικό σημείο και μετά.
Επιπλέον αναλαμβάνει να μεταφέρει το ήδη υπάρχον σύστημα στη νέα αυτή βέλτιστη κατάσταση. Για να το πετύχει αυτό, χρησιμοποιείται ένας ακόμα αλγόριθμος που αναπτύχθηκε στο πλαίσιο της εργασίας αυτής. Ο αλγόριθμος αυτός συγκρίνει την προηγούμενη βέλτιστη ομαδοποίηση των pods με τη νέα και βρίσκει τον ελάχιστο αριθμό μετακινήσεων pods μεταξύ των κόμβων που να οδηγεί στην κατάσταση αυτή. Στη συνέχεια, για να αποφευχθεί οποιοδήποτε downtime, δημιουργεί αντίγραφα για όλα τα pods που πρέπει να μεταφερθούν στους νέους κόμβους και σταδιακά μεταφέρει όλη την κίνηση σε αυτά. Όταν πλέον τα παλιά pods έχουν σταματήσει να λειτουργούν, τότε και μόνο τότε τα διαγράφει για να ολοκληρωθεί η διαδικασία. Η προσέγγιση αυτή εξασφαλίζει τη συνεχή διαθεσιμότητα της εφαρμογής και δίνει την επιλογή να χρησιμοποιηθεί ακόμα και σε περιβάλλον παραγωγής (production environment), όπου ακόμα και σύντομα αδρανή διαστήματα δεν γίνονται εύκολα αποδεκτά.

4.5 Επιλογή Εφαρμογής και Περιβάλλον εγκατάστασης

Για να αποτιμηθούν τα αποτελέσματα επιλέχθηκε η εφαρμογή Online Boutique της Google [22]. Είναι ένα πλήρες ηλεκτρονικό κατάστημα το οποίο δίνει δυνατότητα για περιήγηση, αναζήτηση προϊόντων, δημιουργία καλαθιού, πληρωμή και ολοκλήρωση των αγορών. Η εφαρμογή αυτή είναι σχεδιασμένη με σκοπό να αποτελέσει σημείο αναφοράς για τη δημιουργία εφαρμογών που χρησιμοποιούν αρχιτεκτονικές Microservices και Kubernetes orchestration. Έτσι λοιπόν, αποτελεί κατάλληλη επιλογή για να αξιολογήσουμε τα αποτελέσματα της υλοποίησης του δρομολογητή. Παράλληλα, αναμένεται να υπάρχει ανάλογη συμπεριφορά σε άλλες παρόμοιας λογικής και όχι μόνο εφαρμογές. Σε ό,τι αφορά το περιβάλλον στο οποίο στήθηκε αυτή η εφαρμογή, χρησιμοποιήθηκε ένα τοπικό cluster από τέσσερις, όσο το δυνατόν περισσότερο, πανομοιότυπους, nodes.

4.6 Σχεδιασμός Πειραμάτων

Για το πειραματικό στάδιο, που θα παρουσιαστεί αναλυτικά παρακάτω, επιλέχθηκε η σύγκριση μεταξύ της προτεινόμενης λύσης, του kube-scheduler, της πρότασης NetMARKS αλλά και δύο τυχαίων χρονοδρομολογητών. Με τον τρόπο αυτό εξασφαλίστηκε ότι θα έχουμε πλήρη εικόνα για τις βελτιώσεις και τα προβλήματα της κάθε λύσης. Επιπλέον, επιλέχθηκαν σενάρια και στατικά αλλά και δυναμικά με σταδιακές αλλαγές στη συμπεριφορά της εφαρμογής με σκοπό να αναδείξουν τη συμπεριφορά των δρομολογητών σε κάθε ένα τέτοιο σενάριο. Τα πειράματα είχαν ως βασικό δείκτη σύγκρισης το μέσο χρόνο απόκρισης της εφαρμογής, αλλά έδωσαν και μια καλή εικόνα για επιπλέον στοιχεία όπως για παράδειγμα το μέγιστο αριθμό requests που μπορεί σε κάθε περίπτωση να αντέξει το σύστημα. Οι επιλογές αυτές κρίθηκε ότι μπορούν με γενικό και αντικειμενικό τρόπο να οδηγήσουν σε συμπεράσματα σχετικά με την απόδοση του προτεινόμενου χρονοδρομολογητή.

Στη συνέχεια θα παρουσιαστεί αναλυτικά η υλοποίηση της λύσης καθώς και όλα τα εργαλεία που χρησιμοποιήθηκαν. Επιπλέον θα αναλυθούν τα πειράματα που έγιναν και τα συμπεράσματα στα οποία οδήγησαν, ενώ τέλος θα γίνει αναφορά στα προβλήματα που αντιμετωπίστηκαν και τις πιθανές μελλοντικές επεκτάσεις.

ΚΕΦΑΛΑΙΟ 5 – Υλοποίηση Λύσης

Ο χρονοδρομολογητής που υλοποιήθηκε κατασκευάστηκε εξ ’ολοκλήρου στο πλαίσιο αυτής της διπλωματικής χρησιμοποιώντας τη γλώσσα προγραμματισμού Python και δεν χρησιμοποίησε τις δυνατότητες επέκτασης του kube-scheduler. Η επιλογή αυτή έγινε ώστε να προσφέρει τη δυνατότητα για ταχύτερη ανάπτυξη του δρομολογητή και μεγαλύτερο έλεγχο της συμπεριφοράς του χωρίς την ανάγκη για πλήρη εξοικείωση με το σύνθετο μοντέλο του kube-scheduler και με τη γλώσσα προγραμματισμού Go στην οποία είναι γραμμένο. Για την δημιουργία του δρομολογητή χρησιμοποιήθηκαν βιβλιοθήκες της Python που εκμεταλλεύονται το Rest API του Kubernetes για να παρέχουν τον απαιτούμενο έλεγχο πάνω στο cluster και τα στοιχεία του. Για τη δημιουργία των images χρησιμοποιήθηκε το Docker, ενώ το Kubernetes αποτέλεσε την επιλογή για την ενορχήστρωση του cluster. Η εφαρμογή που χρησιμοποιήθηκε είναι η Online Boutique της Google, και πάνω στα pods του cluster στήθηκε επιπλέον το Istio Service Mesh για τη συλλογή των απαραίτητων μετρικών. Τέλος, χρησιμοποιήθηκε το Prometheus και το Kiali, που προσφέρουν τη δυνατότητα για ερωτήματα πάνω στα δεδομένα του Istio.

5.1 Εργαλεία

5.1.1 Python

Η Python [23] είναι μια γλώσσα προγραμματισμού υψηλού επιπέδου και γενικής χρήσης. Ο σχεδιασμός της δίνει ιδιαίτερη έμφαση στην αναγνωσιμότητα του κώδικα, ενώ είναι σχετικά εύκολη στη χρήση της. Επιπλέον, προσφέρει μεγάλη φορητότητα μεταξύ διαφορετικών μηχανημάτων και λειτουργικών συστημάτων. Είναι δυναμική γλώσσα (dynamically typed) και υποστηρίζει τη συλλογή απορριμμάτων (garbage collection). Είναι γλώσσα προστακτικού προγραμματισμού και υποστηρίζει τόσο τον αντικειμενοστραφή προγραμματισμό όσο και τον διαδικαστικό. Δημιουργήθηκε από τον Ολλανδό Guido van Rossum το 1989 και κυκλοφόρησε για πρώτη φορά το 1991. Στη διάρκεια ζωής της Python έχουν υπάρξει τρεις σημαντικές εκδόσεις: η Python, η Python 2.0 και η Python 3.0. Η μεγάλη κοινότητα που χρησιμοποιεί την Python έχει δημιουργήσει σημαντικό αριθμό βιβλιοθηκών που συνεισφέρουν επιπλέον στην ευκολία της γλώσσας και τις πολλές δυνατότητες που δίνει τόσο σε αρχάριους όσο και σε έμπειρους προγραμματιστές. Στο πλαίσιο της διπλωματικής αυτής χρησιμοποιήθηκε η έκδοση 3.10.11.

5.1.2 Docker και Docker Engine

Το Docker [24] είναι μία πλατφόρμα ανοιχτού κώδικα που υλοποιεί Εικονικοποίηση (Virtualization) σε επίπεδο λειτουργικού συστήματος. Το Docker δίνει τη δυνατότητα για ανάπτυξη εφαρμογών σε απομονωμένα τμήματα μέσα στο λειτουργικό σύστημα, που ονομάζονται containers. Δημιουργήθηκε από την εταιρεία Docker Inc. και βγήκε στην αγορά για πρώτη φορά το 2013. Το Docker Engine [25] είναι ένα από τα βασικά τμήματα του Docker

που έχει ως ρόλο να χτίζει (build) images και να δημιουργεί containers (containerize). Στο πλαίσιο της διπλωματικής αυτής χρησιμοποιήθηκε το Docker για τη δημιουργία των images της εφαρμογής που στήθηκε. Η έκδοση που χρησιμοποιήθηκε είναι η 24.0.6.

5.1.3 Kubernetes

Το Kubernetes [16] ή για συντομία K8s είναι ένα σύστημα ενορχήστρωσης containers ανοικτού κώδικα που συνεισφέρει στην αυτοματοποίηση της διαδικασίας ανάπτυξης, κλιμάκωσης και διαχείρισης λογισμικού πακεταρισμένου σε containers . Σχεδιάστηκε αρχικά από την Google ενώ πλέον διατηρείται από μια παγκόσμια κοινότητα ενδιαφερόντων και η εμπορική του σήμανση ανήκει στην Cloud Native Computing Foundation. Το Kubernetes ομαδοποιεί ένα ή περισσότερους υπολογιστές, φυσικούς ή εικονικούς, σε ένα cluster, το οποίο στη συνέχεια μπορεί να τρέξει μια εφαρμογή δομημένη σε containers. Για την εργασία αυτή χρησιμοποιήθηκε η έκδοση 1.27.2.

5.1.4 Istio Service Mesh

Το Istio [18] είναι ένα πλέγμα υπηρεσιών (service mesh) ανοιχτού κώδικα που προσφέρει λύση σε κάποιες από τις δυσκολίες που προκύπτουν κατά τη διαδικασία ανάπτυξη λογισμικού σε κατανεμημένες αρχιτεκτονικές ή αρχιτεκτονικές Microservices. Προσφέρει δυνατότητες για διαχείριση της κυκλοφορίας, παρακολούθηση του δικτύου αλλά και αύξηση της ασφάλειας της εφαρμογής. Λειτουργεί σαν sidecar injection proxy πάνω σε κάθε pod του cluster, δηλαδή για κάθε pod υπάρχει ένα αντίστοιχο component μέσα από το οποίο περνάει όλη η εισερχόμενη και εξερχόμενη κίνηση. Δημιουργήθηκε από τη Google σε συνεργασία με τις IBM και Lyft και παρουσιάστηκε για πρώτη φορά το 2017. Στην εργασία αυτή χρησιμοποιήθηκε το Istio για να παρέχει τα δεδομένα εκείνα που είναι απαραίτητα έτσι ώστε να μπορεί να τρέξει ο προτεινόμενος δρομολογητής. Η έκδοση που χρησιμοποιήθηκε είναι η 1.18.0.

5.1.5 Prometheus

Το Prometheus [19] είναι ένα εργαλείο παρακολούθησης και ειδοποίησης συστημάτων, ανοιχτού κώδικα, που δημιουργήθηκε το 2012 από την εταιρεία SoundCloud. Πλέον, όπως και το Kubernetes, διατηρείται από μια παγκόσμια κοινότητα ενδιαφερόντων και η εμπορική του σήμανση ανήκει στην Cloud Native Computing Foundation. Το Prometheus λειτουργεί συλλέγοντας δεδομένα από διαφορετικές πηγές χρησιμοποιώντας HTTP endpoints, exporters αλλά και απευθείας από εφαρμογές που αποστέλλουν δεδομένα χρησιμοποιώντας το πρωτόκολλο του. Το Prometheus μαζεύει τα δεδομένα και τα αποθηκεύει στη δική του βάση σε μορφή χρονοσειρών, προσθέτοντας και την ακριβή στιγμή που τα σύλλεξε. Στη συνέχεια δίνει πρόσβαση στα δεδομένα αυτά με τη χρήση μία γλώσσας που ονομάζεται PromQL και χρησιμοποιείται για ερωτήματα στη βάση του Prometheus. Στην εργασία αυτή το Prometheus χρησιμοποιήθηκε ώστε να συλλέγει τα δεδομένα από το Istio και να δίνει μετά τη δυνατότητα για ερωτήματα (queries) πάνω σε αυτά. Η έκδοση που χρησιμοποιήθηκε είναι η 2.41.0.

5.1.6 Kiali

Το Kiali [26] είναι ένα εργαλείο ανοιχτού κώδικα για την οπτικοποίηση μίας εφαρμογής που τρέχει με χρήση Kubernetes και Istio. Προσφέρει τη δυνατότητα παρακολούθησης της υγείας όλων των υπηρεσιών που τρέχουν στο cluster. Επιπλέον, παρουσιάζει, και με γραφικό τρόπο, τις σχέσεις μεταξύ των διαφορετικών υπηρεσιών, εφαρμογών και γενικότερα τμημάτων της εφαρμογής. Δίνει έτσι μια καλύτερη εικόνα στους χρήστες για τις αλληλεξαρτήσεις που υπάρχουν και για την γενικότερη δομή και αρχιτεκτονική του συστήματος. Για να πετύχει την λειτουργικότητα αυτή το Kiali χρησιμοποιεί τόσο δεδομένα από το Kubernetes API σχετικά με την υγεία και τα χαρακτηριστικά των nodes, pods και όλων των τμημάτων του συστήματος, όσο και το Prometheus στο οποίο κάνει queries σχετικά με τη δυναμική συμπεριφορά της εφαρμογής. Αξίζει να σημειωθεί πως ο πειραματισμός μέσα από το Kiali σε συνδυασμό με την εργασία NetMARKS έδωσαν τη βασική ιδέα για την ανάπτυξη του χρονοδρομολογητή της εργασίας αυτής. Η έκδοση που χρησιμοποιήθηκε είναι η 1.67.2.

5.1.7 Grafana

Το Grafana [27] είναι ένα εργαλείο ανοιχτού κώδικα που χρησιμοποιείται για την παρακολούθηση και την οπτικοποίηση δεδομένων. Δημιουργήθηκε το 2014 από τον Torkel Ödegaard και αρχικά είχε ως ρόλο την προβολή δεδομένων από το Graphite, ένα σύστημα γραφημάτων και παρακολούθησης. Από τότε έχει εξελιχθεί και επεκταθεί σε μεγάλο βαθμό και πλέον μπορεί να διαχειρίζεται και να προβάλει δεδομένα που συλλέγονται από διάφορες πηγές όπως Prometheus, Elasticsearch, Graphite και πολλά άλλα. Η χρήση πολλών εργαλείων που βοηθούν στην οπτικοποίηση δίνει στους χρήστες του Grafana τη δυνατότητα να εξάγουν χρήσιμα αποτελέσματα για τα δεδομένα τους ανακαλύπτοντας συσχετισμούς, τάσεις, προβλήματα απόδοσης αλλά και πολλά άλλα. Η χρησιμότητα του συγκεκριμένου εργαλείου το έχει καταστήσει ως ένα από τα πιο συχνά χρησιμοποιούμενα εργαλεία παρακολούθησης στην αγορά της πληροφορικής [28]. Η έκδοση που χρησιμοποιήθηκε είναι η 9.0.1.

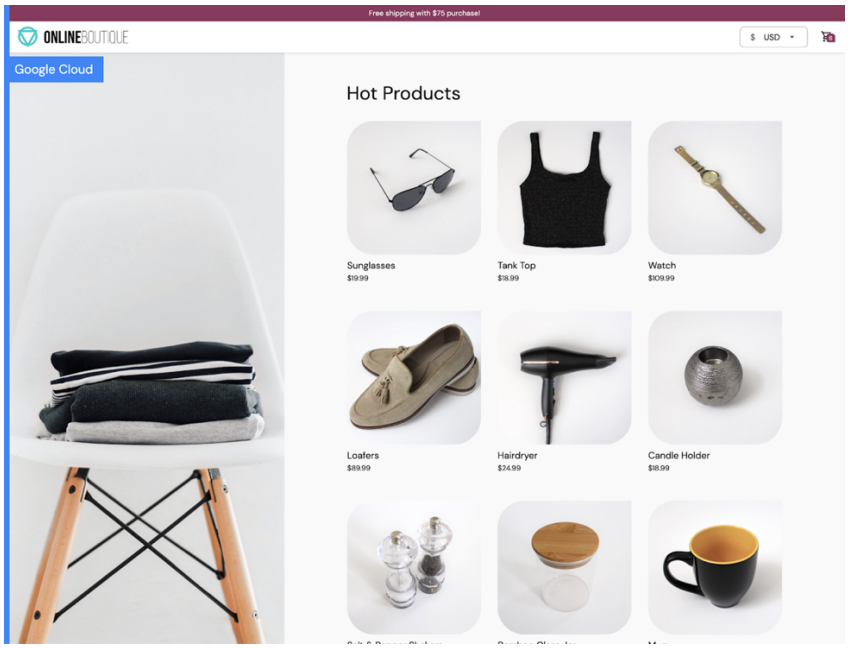
5.2 Εφαρμογή

Η εφαρμογή που επιλέχθηκε για να δοκιμαστεί η προτεινόμενη λύση της διπλωματικής αυτής είναι η Online Boutique της Google [22]. Συγκεκριμένα χρησιμοποιήθηκε η υλοποίηση της σελίδας https://github.com/GoogleCloudPlatform/microservices-demo διότι παρέχει εύκολη εγκατάσταση τόσο της ίδιας της εφαρμογής σε cluster Kubernetes όσο και των απαραίτητων εργαλείων Istio, Prometheus και Kiali. Η εφαρμογή αυτή δημιουργήθηκε από τη Google σαν τμήμα του Google Cloud Platform με σκοπό να παρέχει ένα πρότυπο μιας πλήρους εφαρμογής e-commerce στημένη σε Kubernetes, δίνοντας το περιβάλλον σε προγραμματιστές για σχετικές δοκιμές.

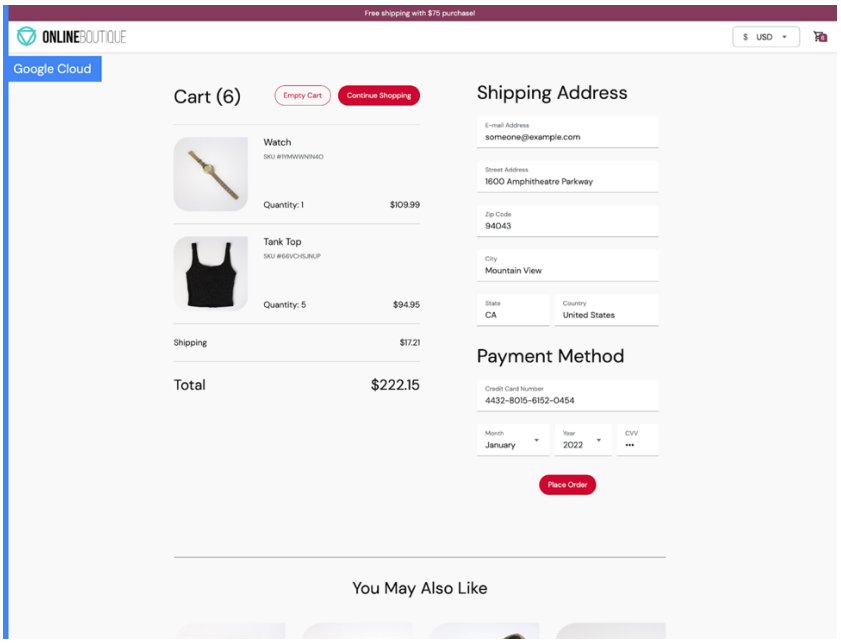
Το Online Boutique περιλαμβάνει όλες τις βασικές υπηρεσίες που παρέχει ένα πραγματικό e-commerce site. Παρέχει δυνατότητα για αναζήτηση και προβολή προϊόντων, δημιουργία καλαθιού, ολοκλήρωση αγορών αλλά και παρακολούθηση των παραγγελιών. Οι λειτουργίες

αυτές επιτυγχάνονται μέσα από έναν αριθμό υπηρεσιών (services) πάνω στις οποίες είναι στημένη. Στην ορολογία του Kubernetes, τα services είναι ομαδοποιημένα pods που εκτελούν μία συγκεκριμένη λειτουργία [29]. Τα pods αυτά γίνονται services και αποκτούν στο πλαίσιο του cluster ένα όνομα και μία σταθερή εσωτερική IP (καθώς και μερικές ακόμα ιδιότητες). Με τον τρόπο αυτό διευκολύνεται, εσωτερικά του cluster, η ανεύρεση των pods που επιτελούν κάποια συγκεκριμένη λειτουργία. Τα services του Online Boutique είναι:

- frontend: Αποτελεί όλη την εμφάνιση της εφαρμογής με την οποία αλληλεπιδρά ο χρήστης. Μέσα από το frontend ο χρήστης επιλέγει συγκεκριμένες δράσεις οι οποίες πυροδοτούν requests προς τα υπόλοιπα services της εφαρμογής. Παρακάτω φαίνονται δύο στιγμιότυπα από την εμφάνιση της εφαρμογής:



Εικόνα 6. Σελίδα προϊόντων της εφαρμογής Online Boutique [22]



Εικόνα 7. Σελίδα ολοκλήρωσης αγοράς της εφαρμογής Online Boutique [22]

- adservice: Η υπηρεσία αυτή αναλαμβάνει την προβολή διαφημίσεων στο πλαίσιο της εφαρμογής, όπως θα γινόταν και σε μία πραγματική σελίδα ενός ηλεκτρονικού καταστήματος.
- checkoutservice: Η υπηρεσία αυτή αναλαμβάνει την ολοκλήρωση της αγοράς του καλαθιού του χρήστη.
- cartservice: Η υπηρεσία που διαχειρίζεται το καλάθι με τα προϊόντα που έχει επιλέξει ο χρήστης. Για τη λειτουργία της χρησιμοποιεί επιπλέον και μία Redis cache [30] η οποία δημιουργεί ξεχωριστά pods στο σύστημα.
- currencyservice: Η υπηρεσία αυτή είναι υπεύθυνη για την παρουσίαση των τιμών των προϊόντων μέσα στο site στο επιλεγμένο από το χρήστη νόμισμα.
- emailservice: Η υπηρεσία αυτή αφορά τις λειτουργίες της εφαρμογής σχετικά με την ηλεκτρονική αλληλογραφία όπως για παράδειγμα την αποστολή επιβεβαιωτικού email αγοράς.
- paymentservice: Η υπηρεσία αυτή διεκπεραιώνει τη διαδικασία της πληρωμής κατά την ολοκλήρωση της αγοράς.
- productcatalogservice: Η υπηρεσία αυτή αναλαμβάνει την εμφάνιση και διαχείριση των προϊόντων του καταστήματος που φαίνονται στο site.
- recommendationservice: Η υπηρεσία αυτή παράγει τις προτάσεις που γίνονται προς τον χρήστη σχετικά με τα προϊόντα του καταστήματος.
- shippingservice: Η υπηρεσία αυτή αναλαμβάνει την αποστολή και παρακολούθηση των προϊόντων.

Όπως φαίνεται και από τα παραπάνω services και θα φανεί και ακόμα περισσότερο στην πορεία, η εφαρμογή αυτή διαθέτει μια συγκεκριμένη εσωτερική τοπολογία. Αυτό σημαίνει ότι τα services που την απαρτίζουν δημιουργούν έναν γράφο επικοινωνίας, με τους κόμβους να είναι οι ίδιες οι υπηρεσίες και τις ακμές να είναι οι αντίστοιχες εσωτερικές επικοινωνίες της εφαρμογής. Ο γράφος αυτός δεν είναι πλήρης (δεν επικοινωνούν όλα τα services μεταξύ τους) και ακριβώς αυτή η πληροφορία είναι που θα φανεί χρήσιμη στη διαδικασία επιλογής της βέλτιστης δρομολόγησης. Η εφαρμογή αυτή αποτελεί καλό περιβάλλον για δοκιμές και πειράματα γιατί έχει ένα σχετικά πυκνό γράφο αλληλεξαρτήσεων, ενώ παράλληλα προσομοιάζει έναν τύπο εφαρμογής πολύ συχνό στο διαδίκτυο. Έτσι λοιπόν τα συμπεράσματα στα οποία θα οδηγήσει ο πειραματισμός σε ένα τέτοιο περιβάλλον έχουν αυξημένες πιθανότητες να οδηγήσουν σε πορίσματα και για μεγάλο πλήθος άλλων εφαρμογών.

5.3 Cluster και στήσιμο εφαρμογής

Για τη διπλωματική αυτή χρησιμοποιήθηκε ένα cluster στημένο σε τοπικό δίκτυο. Το cluster αυτό αποτελείται από τέσσερις πανομοιότυπες εικονικές μηχανές που λειτουργούν ως nodes του συστήματος. Τα nodes αυτά έχουν λειτουργικό σύστημα Linux, αρχιτεκτονικής amd64, και τρέχουν Ubuntu 22.04.2 LTS. Καθένα από αυτά διαθέτει δύο εικονικούς επεξεργαστές (vCPUs), 4GB μνήμη και 20GB αποθηκευτικό χώρο. Επιπλέον, δόθηκε ιδιαίτερη έμφαση στο να βρίσκονται οι κόμβοι αυτοί σε διαφορετικά φυσικά μηχανήματα, έτσι ώστε οι επιβαρύνσεις

της επικοινωνίας μεταξύ διαφορετικών κόμβων να είναι ίδιες σε κάθε περίπτωση και συνεπώς τα αποτελέσματα να είναι όσο το δυνατόν πιο αντικειμενικά.
Μετά τη δημιουργία του cluster εγκαταστάθηκε το Kubernetes ως ο βασικός ενορχηστρωτής για όλα τα components της εφαρμογής. Στη συνέχεια, εγκαταστάθηκε και διαμορφώθηκε το Istio σαν sidecar injection έτσι ώστε αυτόματα να μπαίνει το αντίστοιχο proxy container μπροστά από κάθε pod που στήνεται στο cluster. Έπειτα, χρησιμοποιήθηκαν τα YAML αρχεία της εφαρμογής του Online Boutique σε συνδυασμό με το εργαλείο Kustomize για να δημιουργήσουν όλα τα pods της εφαρμογής.

- YAML (Yaml Ain't Markup Language) [31] είναι μία γλώσσα σειριοποίησης δεδομένων. Είναι ιδιαίτερα διαδεδομένη και πολλές φορές χρησιμοποιείται για τη δημιουργία αρχείων ρυθμίσεων (configuration files). Στη συγκεκριμένη περίπτωση χρησιμοποιείται για τη δημιουργία των configuration files για όλα τα services της εφαρμογής, τα οποία μετά προωθούνται στο Kubernetes και του παρέχουν οδηγίες για τα pods τα οποία πρέπει να δρομολογήσει, αλλά και για γενικότερες πληροφορίες σχετικά με το cluster.
- Kustomize [32] είναι ένα εργαλείο διαμόρφωσης και διαχείρισης αρχείων YAML για το Kubernetes που δίνει τη δυνατότητα για τροποποιήσεις πάνω σε αρχεία χωρίς να τροποποιούνται τα ίδια τα αρχεία.

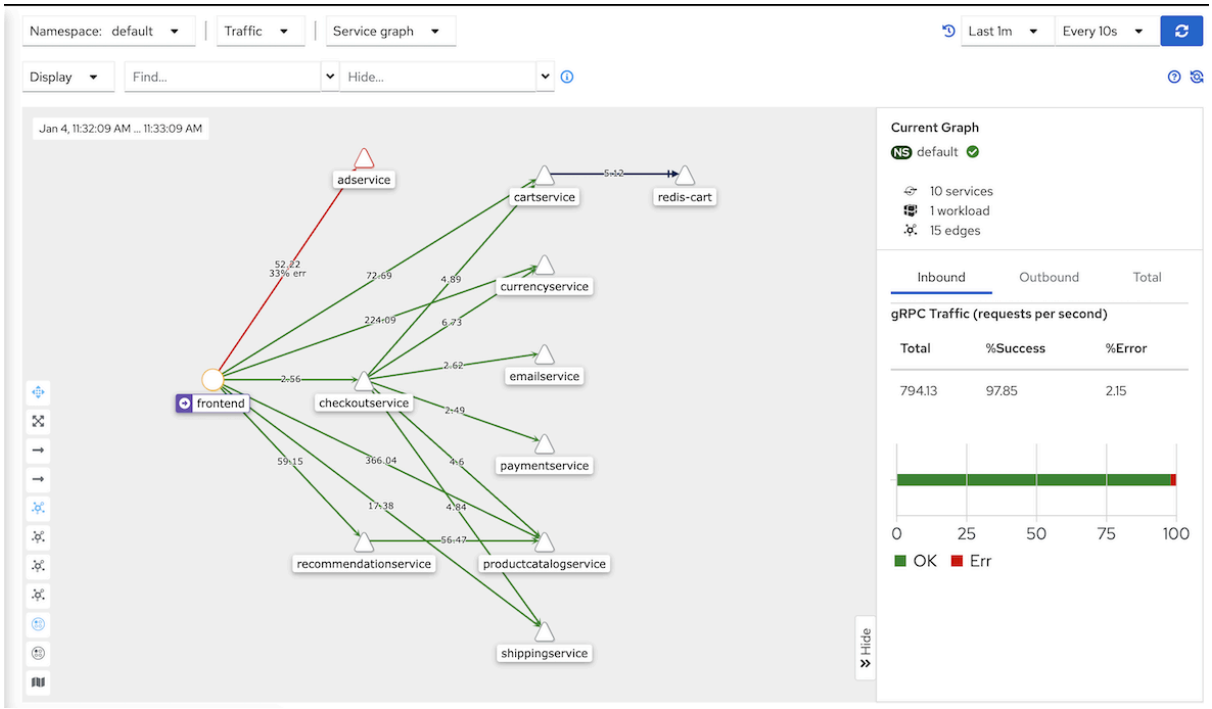
Κατά το αρχικό στήσιμο της εφαρμογής δεν έγιναν τροποποιήσεις στα YAML αρχεία και συνεπώς τη δρομολόγηση των pods ανέλαβε ο kube-scheduler. Παράλληλα με τη χρήση του εργαλείου εντολών istioctl που παρέχει το Istio, εγκαταστάθηκαν το Prometheus, το Kiali και το Grafana. Στη συνέχεια εγκαταστάθηκε το εργαλείο Goldpinger [33].

- Goldpinger είναι ένα εργαλείο ανοιχτού κώδικα για παρακολούθηση του εσωτερικού δικτύου ενός cluster Kubernetes. Παρέχει μετρικές στις οποίες έχει πρόσβαση το Prometheus για δεδομένα σχετικά με το εσωτερικό δίκτυο της εφαρμογής. Στη συγκεκριμένη περίπτωση το Goldpinger χρησιμοποιήθηκε για να διασφαλίσουμε ότι όλοι οι κόμβοι του συστήματος είχαν μεταξύ τους παρόμοιους χρόνους απόκρισης και συνεπώς δεν επηρέαζαν την αντικειμενικότητα των μετρήσεών μας. Για να πετύχει τη λειτουργία του το εργαλείο αυτό δρομολογεί ένα pod σε κάθε node του συστήματος.

Αφού ολοκληρώθηκε το στήσιμο της εφαρμογής και όλων των εργαλείων, δοκιμάστηκε η ορθή λειτουργία τους συστήματος. Για να γίνει αυτό προωθήθηκε η εσωτερική πόρτα του δικτύου του Kubernetes στο προσωπικό μηχάνημα που χρησιμοποιούταν, δίνοντας έτσι πρόσβαση στο frontend της εφαρμογής. Η εντολή που χρησιμοποιήθηκε για την επίτευξη του στόχου αυτού είναι η "kubectl port-forward svc/frontend 8080:80" η οποία προωθεί την πόρτα 80 του service frontend στην τοπική πόρτα 8080 δίνοντας έτσι πρόσβαση στην εφαρμογή μέσω της διεύθυνσης http://localhost:8080/.

5.4 Istio Service Graph

Αφού επαληθεύτηκε η ορθή λειτουργία της εφαρμογής, έγιναν αρκετοί πειραματισμοί εως ότου προκύψει κάποια εναλλακτική μέθοδος δρομολόγησης με καλά αποτελέσματα. Οι πειραματισμοί αυτοί συνέκριναν τα αποτελέσματα των δρομολογήσεων ως προς διαφορετικές μετρικές. Έτσι λοιπόν έγιναν δοκιμές για τη μείωση του κόστους φιλοξενίας της εφαρμογής χρησιμοποιώντας δυναμικές μετρικές για την βέλτιστη κατανομή των πόρων του cluster. Έγιναν επίσης δοκιμές με βάση την πρόταση του NetMARKS και στόχο μία λύση με παρόμοια αποτελέσματα αλλά με πιο ομοιόμορφη χρησιμοποίηση των nodes. Τελικά, η ζητούμενη βελτίωση επιλέχθηκε να αφορά τον χρόνο απόκρισης αφού κρίθηκε ως ένα από τα σημαντικότερα χαρακτηριστικά για μία εφαρμογή λόγω της αντικειμενικότητάς του αλλά και τις επιρροής που έχει στη συνολική εμπειρία του χρήστη της εφαρμογής. Για την επίτευξη του στόχου αυτού επιλέχθηκε η χρησιμοποίηση ενός εκ των γραφημάτων που προσφέρει το εργαλείο Kiali. Το γράφημα αυτό παρουσιάζει έναν γράφο με τον αριθμό ανά μονάδα χρόνου των requests που γίνονται μεταξύ των διαφορετικών services. Ένα στιγμιότυπο του γραφήματος αυτού παρουσιάζεται παρακάτω:



Εικόνα 8. Γράφος της εφαρμογής Kiali με τις επικοινωνίες μεταξύ των services της εφαρμογής

Στο γράφημα αυτό παρουσιάζονται ως κόμβοι όλες οι υπηρεσίες της εφαρμογής. Οι ακμές είναι ο αριθμός των requests ανά μονάδα χρόνου μεταξύ των αντίστοιχων υπηρεσιών. Το γράφημα αυτό δίνει μια πολύ καλή εικόνα για το πώς επικοινωνούν οι υπηρεσίες και συνεπώς τα pods μεταξύ τους. Με αφορμή το συγκεκριμένο διάγραμμα δημιουργήθηκε η ιδέα για υλοποίηση ενός δρομολογητή, ο οποίος θα τρέχει έναν αλγόριθμο βέλτιστης ομαδοποίησης των κόμβων σε τόσες ομάδες όσες και οι κόμβοι του cluster. Η ομαδοποίηση έχει ως στόχο να ελαχιστοποιήσει τα αθροιστικά βάρη των ακμών που ενώνουν κόμβους που ανήκουν σε

διαφορετικές ομάδες. Από το αποτέλεσμα ενός τέτοιου αλγορίθμου στη συνέχεια θα προκύψει ένας χάρτης δρομολόγησης pods σε nodes που θα χρησιμοποιεί ο scheduler που θα υλοποιηθεί. Έτσι λοιπόν με τη δρομολόγηση αυτή είναι πιθανό να μειωθεί και ο χρόνος απόκρισης του συστήματος αφού μειώνονται όσο το δυνατόν περισσότερο οι χρονικά πιο δαπανηρές επικοινωνίες του συστήματος.

5.5 Υλοποίηση Αλγορίθμου Βέλτιστης Ομαδοποίησης

Για την υλοποίηση του συγκεκριμένου αλγορίθμου όπως και όλων των υπόλοιπων τμημάτων της εφαρμογής επιλέχθηκε η γλώσσα προγραμματισμού Python. Ο αλγόριθμος χρειάζεται σαν είσοδο τον αριθμό των μηχανημάτων του συστήματος καθώς και τον μέγιστο αριθμό των pods που μπορούν να δρομολογηθούν σε κάθε node.
Στο πρώτο στάδιο, ο αλγόριθμος κάνει ένα request προς το Kiali από το οποίο μαζεύει τα δεδομένα που χρησιμοποιούνται για τη δημιουργία του γράφου που φαίνεται στην εικόνα 8. Τα δεδομένα αυτά βρίσκονται σε ένα μεγάλο αρχείο JSON το οποίο περιέχει σχεδόν όλες τις πληροφορίες που εμφανίζονται μέσα από την εφαρμογή του Kiali. Στο request αυτό καθορίζεται και το χρονικό διάστημα για το οποίο θα επιστραφούν τα αποτελέσματα. Στο πλαίσιο της εργασίας αυτής χρησιμοποιήθηκε το χρονικό διάστημα των τελευταίων τριών λεπτών, διότι είναι σχετικά μεγάλο για να παρουσιάσει με αντικειμενικό τρόπο τη λειτουργία της εφαρμογής αλλά και αρκετά μικρό ώστε να προλάβει να επηρεαστεί από αλλαγές που προσπαθήσαμε να δημιουργήσουμε για να μπορέσουμε να μελετήσουμε τη συμπεριφορά του χρονοδρομολογητή. Στη συνέχεια, γίνεται η κατάλληλη επεξεργασία των δεδομένων που έρχονται καθώς απορρίπτονται όλα εκείνα που δεν αφορούν το γράφημα. Έπειτα, από τα δεδομένα δημιουργείται ένα αρχείο CSV που περιέχει το πίνακα γειτνίασης του γράφου.
Στο δεύτερο στάδιο, ο αλγόριθμος διαβάζει τον πίνακα γειτνίασης και τον αποθηκεύει σε μια μεταβλητή-πίνακα της Python. Στη συνέχεια δημιουργεί τους διαφορετικούς συνδυασμούς δρομολόγησης για τα nodes που έχει δεχτεί ως είσοδο και τα pods που βρίσκονται στις πληροφορίες του γραφήματος που έχει διαβάσει. Στο σημείο αυτό γίνονται και κατάλληλοι έλεγχοι για το κατά πόσο η δρομολόγηση αυτή είναι εφικτή.
Αξίζει να αναφερθεί ότι το γράφημα που προσφέρει το Kiali αφορά services και όχι pods. Για την εργασία όμως αυτή χρησιμοποιήθηκαν μόνο services που αποτελούνται από ένα pod οπότε στη πραγματικότητα services και pods αναφέρονται στην ίδια οντότητα. Ο λόγος είναι ότι το μέγεθος του τοπικού cluster και οι δοκιμές που μπορούσαν να γίνουν δεν δικαιολογούσαν μεγαλύτερο αριθμό pods οπότε σε συνδυασμό με την αφαίρεση ενός ακόμα σταδίου στην επεξεργασία της πληροφορίας επιλέχθηκε η προσέγγιση αυτή. Για κάθε έναν από τους συνδυασμούς που προέκυψαν χρησιμοποιείται μια ρουτίνα η οποία μετράει το πλήθος των επικοινωνιών με διαφορετικούς κόμβους και επιστρέφει το αντίστοιχο σκορ. Στο τέλος προκύπτει σαν καλύτερη λύση εκείνη με το χαμηλότερο σκορ η οποία επιστρέφεται σαν ένας πίνακας πινάκων όπου όλοι οι εσωτερικοί πίνακες περιέχουν τα pods-services που πρέπει να δρομολογηθούν εκεί. Παρακάτω παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου αυτού:

```
Algorithm 1: Clustering Algorithm
Input: NumberOfNodes, NumberOfPods, KialiInterval
Output: BestTopology
data <- getDataFromKiali(KialiInterval) // http request to kiali
graphData <- filterGraph(data) // keep data regarding graph
pods <- getPodsFromData(graphData)
if NumberOfPods*NumberOfNodes < pods: // not possible to create topology
    return
trafficArray <- getArrayFromData(graphData, pods) // turn data into an nxn array showing traffic between pods
BestTopology <- none
bestScore <- inf
for all allowed possible topologies do
    tempScore <- calculateInterNodeTraffic(topology)
    if tempScore < bestScore then
        bestScore <- tempScore
        BestTopology <- topology
return BestTopology
```

Εικόνα 9. Ψευδοκώδικας για την υλοποίηση του αλγορίθμου βέλτιστης ομαδοποίησης

Αξίζει επίσης να σημειωθεί ότι η είσοδος που δέχεται ο αλγόριθμος μεταβάλλει σημαντικά το αποτέλεσμα που παράγει. Αν ο αριθμός των pods που μπορούν να δρομολογηθούν σε κάθε node είναι μικρός, τότε προκύπτει μια σχετικά πιο ομοιόμορφη χρησιμοποίηση των κόμβων του συστήματος. Και σε αυτή τη περίπτωση βέβαια μπορεί να υπάρχουν σημαντικές διαφορές, αφού οι ανάγκες των pods μπορεί να έχουν αποκλίσεις μεταξύ τους. Από την άλλη πλευρά αν ο αριθμός που θα επιλεγεί είναι μεγαλύτερος, η στρατηγική γίνεται πιο επιθετική. Μια τέτοια επιλογή είναι λογικό να έχει τα καλύτερα τελικά αποτελέσματα, αλλά δημιουργεί και προβλήματα όπως ανάγκη για μεγαλύτερους και ισχυρότερους κόμβους και πιο εύκολα bottlenecks του συστήματος. Η επιλογή αυτή δίνεται στον εκάστοτε χρήστη για να διαμορφώσει το τρόπο λειτουργίας με βάση τις προσωπικές του ανάγκες. Στο πλαίσιο της εργασίας αυτής έγιναν δοκιμές με όλα τα μεγέθη εισόδων και τελικά επιλέχθηκε η πιο συντηρητική προσέγγιση λόγω και των περιορισμένων υπολογιστικών πόρων που είχαμε στη διάθεση μας. Πιο συγκεκριμένα δοκιμάστηκαν τόσο άπληστες όσο και ομοιόμορφες ομαδοποιήσεις. Επειδή οι προδιαγραφές των κόμβων ήταν συγκεκριμένες οι πιο ομοιόμορφες ομαδοποιήσεις οδηγούσαν σε καλύτερη εκμετάλλευση του υλικού και προσέφεραν τη δυνατότητα για πειραματισμούς με μεγαλύτερους αριθμούς χρηστών. Για τον λόγο αυτό η τελική επιλογή ήταν αυτή της συντηρητικής προσέγγισης με την ομοιόμορφη δρομολόγηση των pods στα nodes του συστήματος.

5.6 Υλοποίηση Στατικού Χρονοδρομολογητή

Ο στατικός χρονοδρομολογητής αποτελεί τη βάση του τελικού δρομολογητή που παρουσιάζεται στην εργασία αυτή. Για την υλοποίηση του χρησιμοποιήθηκε η γλώσσα Python. Ο χρονοδρομολογητής επιλέχθηκε να δημιουργηθεί από την αρχή και όχι να γίνει κάποια επέκταση πάνω στην υφιστάμενη λύση που παρέχει το Kubernetes. Έτσι λοιπόν χρειάζεται να υποκαθιστά πλήρως τη λειτουργία του kube-scheduler, παρέχοντας την ίδια λειτουργικότητα με αυτόν. Βασικός ρόλος του χρονοδρομολογητή είναι να ελέγχει συνεχώς το cluster για καινούρια pods. Για να μπορέσει να κάνει τον έλεγχο αυτό αλλά και γενικά να επικοινωνήσει με το cluster χρησιμοποιεί το API που παρέχει το Kubernetes. Για κάθε νέο pod που δημιουργείται επιλέγει με βάση την τοπολογία που έχει δεχτεί ως είσοδο για το ποιο θα ήταν το κατάλληλο node να το δρομολογήσει.

Πιο συγκεκριμένα κατά την εκκίνηση της λειτουργίας του δέχεται σαν είσοδο το αποτέλεσμα του αλγορίθμου που αναλύθηκε παραπάνω. Στη συνέχεια χρησιμοποιώντας το API βρίσκει όλους τους κόμβους του cluster. Σε περίπτωση που ο αριθμός των κόμβων είναι μικρότερος από τον αριθμό των πινάκων που έχει δεχτεί σαν είσοδο, τερματίζει τη λειτουργία του επιστρέφοντας το αντίστοιχο μήνυμα λάθους. Αντιθέτως, εάν ο αριθμός των nodes είναι μεγαλύτερος, τότε λειτουργεί κανονικά και δεν δρομολογεί κανένα pod σε όλους τους υπεράριθμους κόμβους. Έχοντας εξασφαλίσει ότι η δρομολόγηση είναι εφικτή, δένει πάνω σε κάθε έναν από τους διαθέσιμους κόμβους τα pods που του αναλογούν και το κρατάει σαν δεδομένο σε όλη τη διάρκεια λειτουργίας του. Αφού τελειώσει με όλες τι προκαταρκτικές διαδικασίες, εισέρχεται σε μία επαναληπτική δομή, στην οποία ελέγχει συνεχώς το cluster για μη δρομολογημένα pods. Όταν βρει κάποιο μη δρομολογημένο pod, επιχειρεί την δρομολόγησή του στο node που υποδεικνύει η αρχική επεξεργασία. Παρακάτω παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου αυτού:

```
Algorithm 2: Static Scheduler
Input: BestTopology
allNodes <- kubernetesGetNodes()
podsToNodes <- mapBestTopologyPodsToNodes(allNodes, BestTopology) // map all pods to real cluster nodes using BestTopology
while true
    podsPending <- checkForPendingPods()
    for all pod in podsPending do
        schedulePod(pod, podsToNodes) // schedule pod to appropriate node using the podsToNodes
```

Εικόνα 10. Ψευδοκώδικας για τον υλοποίηση του στατικού χρονοδρομολογητή

Για τη λειτουργία του στατικού δρομολογητή χρειάζεται αρχικά να γίνουν οι κατάλληλες ρυθμίσεις στα YAML αρχεία των services. Πρέπει δηλαδή να μπει μία εντολή που να δηλώνει τον υπεύθυνο δρομολογητή για το service αυτό. Επειδή ο δρομολογητής που έχουμε λειτουργεί εξωτερικά αρκεί να δοθεί ένα όνομα δρομολογητή που δεν υπάρχει. Αυτό θα οδηγήσει τα pods να παραμένουν σε κατάσταση αναμονής για δρομολόγηση (pending) μέχρι να τα βρει και να τα δρομολογήσει η προτεινόμενη λύση. Επιπλέον, για να τρέξει αυτόνομα ο στατικός δρομολογητής, πρέπει αρχικά να στηθεί η εφαρμογή με χρήση του kube-scheduler, και στη συνέχεια να μείνει σε λειτουργία για ένα ικανό χρονικό διάστημα. Αυτό απαιτείται για να μπορέσουν να συλλεχθούν αρκετά δεδομένα, να τρέξει ο αλγόριθμος ομαδοποίησης και να παράξει τη βέλτιστη ομαδοποίηση. Στη συνέχεια πρέπει να καταργηθούν όλα τα pods, να γίνουν οι απαιτούμενες αλλαγές και να δημιουργηθούν εκ νέου, αυτή τη φορά χρησιμοποιώντας για τη δρομολόγηση τους το στατικό δρομολογητή. Ο λόγος που υπάρχει ανάγκη για τα επιπλέον αυτά βήματα είναι πως η στατική αυτή λύση χρησιμοποιεί σαν δεδομένο τη βέλτιστη τοπολογία σε όλη τη διάρκεια εκτέλεσής της.

5.7 Υλοποίηση Αλγορίθμου Μετάβασης σε Νέα Κατάσταση

Για να είναι εφικτή η υλοποίηση ενός δυναμικού χρονοδρομολογητή με βάση τον στατικό που παρουσιάστηκε παραπάνω, χρειάστηκε να υλοποιηθεί ένας ακόμα βοηθητικός αλγόριθμος. Σκοπός του αλγορίθμου αυτού είναι να βρίσκει τον ελάχιστο αριθμό κινήσεων που απαιτούνται για τη μετάβαση από μία τοπολογία δρομολόγησης σε μία καινούρια. Πιο συγκεκριμένα, σε αντίθεση με τον στατικό χρονοδρομολογητή, ένας δυναμικός θα συλλέγει συνεχώς δεδομένα και θα υπολογίζει τη βέλτιστη τοπολογία των υπηρεσιών. Η λειτουργία

αυτή δημιουργεί την ανάγκη για μετάβαση από την παλιά στη νέα κατάσταση. Επειδή όμως τα pods είναι πλέον δρομολογημένα και πρέπει να αποφευχθούν όσο το δυνατόν περισσότερες διορθωτικές κινήσεις, κρίθηκε αναγκαίο να δημιουργηθεί ένα αλγόριθμος που θα επιτελεί την λειτουργία αυτή. Θα υπολογίζει δηλαδή τον ελάχιστο αριθμό κινήσεων για τη μετάβαση αυτή. Ο αλγόριθμος που κατασκευάστηκε λειτουργεί ως εξής: Αρχικά δέχεται σαν είσοδο δύο πίνακες πινάκων που αποτελούν την παλιά και την καινούρια τοπολογία. Είναι σημαντικό να τονιστεί πως η σειρά των εσωτερικών πινάκων είναι σημαντική διότι αφορά συγκεκριμένους κόμβους. Πρέπει λοιπόν ο αλγόριθμος να λαμβάνει υπόψιν τη σειρά αυτή όταν αξιολογεί τις πιθανές λύσεις. Στη συνέχεια δημιουργεί όλους τους πιθανούς επιτρεπτούς συνδυασμούς σχετικά με τις κινήσεις που μπορούν να οδηγήσουν από την μία κατάσταση στην άλλη. Βαθμολογεί καθέναν από αυτούς με βάση τον αριθμό των κινήσεων που χρειάζονται και τελικά επιστρέφει τη λύση με τα λιγότερα βήματα. Η λύση αυτή περιέχει όλες τις κινήσεις που απαιτείται να γίνουν για να μεταβεί το σύστημα στην επιθυμητή κατάσταση. Παρακάτω παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου αυτού:

```
1  Algorithm 3: Transition to new best topology
2  Input: NewTopology, OldTopology
3  Output: Moves
4  if size(NewTopology) != size(OldTopology) then
5      return
6  numberOfMoves <- inf
7  for all possible mapping of nodes between OldTopology and NewTopology do // Every node from old will map to a node from
8                                                                           // new topology and then the number of moves
9                                                                           // to get from the old node state to the new state
10                                                                          // will be counted. If we have 5 nodes for example
11                                                                          // in each topology the total mappings would be 5!
12     tempMoves <- 0
13     for all nodes do
14         tempMoves += movesOfNodeFromOldToNewState(mapping)
15     if tempMoves < numberOfMoves then
16         Moves <- getMovesForThisMapping(mapping)
17 return Moves
```

Εικόνα 11. Ψευδοκώδικας για την υλοποίηση του αλγορίθμου μετάβασης σε νέα κατάσταση

5.8 Υλοποίηση Δυναμικού Χρονοδρομολογητή

Ο δυναμικός δρομολογητής είναι η ολοκληρωμένη λύση που παρουσιάζεται στην παρούσα διπλωματική εργασία. Επεκτείνει τον στατικό δρομολογητή που παρουσιάστηκε προσθέτοντας δυναμικό χαρακτήρα στη λειτουργία του. Με τον τρόπο αυτό δημιουργείται ένας δρομολογητής, ο οποίος εξασφαλίζει τη βέλτιστη τοπολογία των services και των pods για όλη τη διάρκεια ζωής της εφαρμογής. Για να υλοποιηθεί ο δρομολογητής αυτός στην πραγματικότητα συνδυάζονται ο αλγόριθμος της ομαδοποίησης, ο στατικός δρομολογητής και ο αλγόριθμος της μετάβασης, ενώ υλοποιούνται και μερικές επιπλέον λειτουργίες. Σε αντίθεση με τον στατικό δρομολογητή, ο δυναμικός μπορεί να ξεκινήσει τη λειτουργία του απευθείας δρομολογώντας αρχικά με Round-Robin τρόπο τα pods. Η προσαρμογή του σε νέα δεδομένα του δίνει τη δυνατότητα να ξεκινάει από μία τυχαία και μη βέλτιστη τοπολογία και σταδιακά να μεταβαίνει στη βέλτιστη.

Πιο συγκεκριμένα ο δυναμικός χρονοδρομολογητής ξεκινάει τη λειτουργία διαβάζοντας μέσω του Kubernetes API όλα τα nodes του συστήματος και τα αποθηκεύει με συγκεκριμένη σειρά. Στη συνέχεια ελέγχει αν υπάρχουν pods σε κατάσταση αναμονής. Σε περίπτωση που υπάρχουν, τα δρομολογεί με Round-Robin τρόπο σε όλα τα nodes του cluster. Επιλέχθηκε η αρχική

δρομολόγηση να γίνει με Round-Robin τρόπο επειδή η υλοποίηση δεν έγινε σαν επέκταση του kube-scheduler και συνεπώς θα ήταν περίπλοκο να αναλάβει αυτός την αρχική δρομολόγηση. Με τον τρόπο αυτό η εφαρμογή βρίσκεται πλέον σε κατάσταση λειτουργίας. Μετά από ένα ικανό διάστημα αναμονής, που στο πλαίσιο της εργασίας αυτής ορίστηκε σε τρία λεπτά, ο δρομολογητής καλεί τον αλγόριθμο της βέλτιστης ομαδοποίησης. Ο αλγόριθμος, αφού διαβάσει τα δεδομένα από το Kiali, παράγει την ιδανική τοπολογία και την επιστρέφει στον δρομολογητή. Ο δρομολογητής αποθηκεύει την τοπολογία αυτή, ώστε να μπορεί να τη χρησιμοποιήσει για τη δρομολόγηση όλων των νέων pods. Πριν όμως μπει στη διαδικασία παρακολούθησης για καινούρια pods, καλείται να φέρει το σύστημα στην νέα βέλτιστη τοπολογία. Για να το πετύχει αυτό, χρησιμοποιεί τον αλγόριθμο της μετάβασης στον οποίο δίνει ως είσοδο την παλιά τοπολογία και τη νέα. Στη συνέχεια, παίρνει τις κινήσεις που επιστρέφει ο αλγόριθμος αυτός και σειριακά αρχίζει να τις εκτελεί. Η εκτέλεση έχει ως εξής: αρχικά αυξάνει τον αριθμό των pods για το συγκεκριμένο service. Με τον τρόπο αυτό δημιουργείται ένα νέο ίδιο pod με αυτό που πρέπει να μετακινηθεί. Το νέο pod δρομολογείται στη σωστή θέση. Όταν ολοκληρωθεί η δρομολόγηση και το pod μπει σε κατάσταση λειτουργίας, μεταφέρεται όλη η λειτουργία του service στο pod αυτό. Έτσι λοιπόν σταδιακά τα παλιό pod σταματάει να δέχεται δικτυακή κίνηση. Όταν και μόνο όταν σταματήσει πλήρως η λειτουργία του, διαγράφεται, μειώνεται κατά ένα ο αριθμός των pods για το service και ολοκληρώνεται η διαδικασία. Με τον τρόπο αυτό διασφαλίζεται η συνεχής διαθεσιμότητα της εφαρμογής. Ακολουθεί η ίδια διαδικασία για όλες τις κινήσεις. Στη συνέχεια αφού το σύστημα έχει έρθει για πρώτη φορά σε κατάσταση βέλτιστης δρομολόγησης ξεκινάει η βασική επαναληπτική δομή. Η δομή αυτή αποτελείται από έναν ελαφρώς παραλλαγμένο στατικό δρομολογητή ο οποίος δρομολογεί με βάση την τελευταία βέλτιστη τοπολογία που έχει στη διάθεσή του. Παράλληλα ανά κάποιο προκαθορισμένο χρονικό διάστημα που δίνεται σαν είσοδος στον δρομολογητή καλείται ο αλγόριθμος της ομαδοποίησης, τροποποιείται η βέλτιστη τοπολογία και ακολουθείται η διαδικασία μετάβασης εφόσον υπάρχουν αλλαγές, προτού η εκτέλεση επιστρέψει στον ελαφρώς παραλλαγμένο στατικό δρομολογητή. Παρακάτω παρουσιάζεται ο ψευδοκώδικας του αλγορίθμου αυτού:

```
Algorithm 4: Dynamic Scheduler
allNodes <- kubernetesGetNodes()
firstTopology <- topologyUsingRoundRobin()
podsToNodes <- mapBestTopologyPodsToNodes(allNodes, firstTopology) // map all pods to real cluster nodes using BestTopology
for all pod in getPodsPending() do
    schedulePod(pod, podsToNodes) // schedule pod to appropriate node using the podsToNodes
wait()
bestTopology <- getBestClustering() // use of Algorithm 1
movesToBestTopology <- transitionToNewBestTopology(firstTopology, bestTopology) // use of Algorithm 2
podsToNodes <- updateMapping(movesToBestTopology)
for all move in movesToBestTopology do
    scaleUpPod(move)
    schedulePod(pod, podsToNodes)
    waitForPodToStart()
    scaleDownPod(move) // deletes oldest pod after all traffic has moved to new pod
while true do
    podsPending <- checkForPendingPods()
    for all pod in podsPending do
        schedulePod(pod, podsToNodes) // schedule pod to appropriate node using the podsToNodes
    if x time has passed then
        bestTopology <- getBestClustering() // use of Algorithm 1
        movesToBestTopology <- transitionToNewBestTopology(oldTopology, BestTopology) // use of Algorithm 2
        podsToNodes <- updateMapping(movesToBestTopology)
        for all move in movesToBestTopology do
            scaleUpPod(move)
            schedulePod(pod, podsToNodes)
            waitForPodToStart()
            scaleDownPod(move) // deletes oldest pod after all traffic is served by new pod
```

Εικόνα 12. Ψευδοκώδικας για τον υλοποίηση του δυναμικού χρονοδρομολογητή

Ο δυναμικός χρονοδρομολογητής επιχειρεί να δώσει λύση σε μερικά από τα προβλήματα που αντιμετωπίζουν τόσο ο default kube scheduler όσο και ο δρομολογητής που προτείνεται στη δημοσίευση του NetMARKS και αναλύθηκε σε προηγούμενο κεφάλαιο. Για τον λόγο αυτό, αφού επιβεβαιώθηκε η ορθή του λειτουργία, σχεδιάστηκε ένα εκτενές σχέδιο πειραμάτων, από το οποίο μπορούν να προκύψουν συμπεράσματα σχετικά με την επίδραση που έχει στη βελτίωση της απόδοσης για μια εφαρμογή με αρχιτεκτονική Microservices.

ΚΕΦΑΛΑΙΟ 6 – Πειραματικό Μέρος

Για την αξιολόγηση της απόδοσης του χρονοδρομολογητή που αναπτύχθηκε, δημιουργήθηκε ένα πειραματικό σχέδιο με στόχο να δοκιμάσει τη συμπεριφορά του κάτω από διαφορετικά σενάρια λειτουργίας. Η βασική μετρική που επιλέχθηκε για την αξιολόγηση αυτή είναι ο μέσος χρόνος απόκρισης της εφαρμογής. Μέσα από τη διαδικασία προέκυψαν συμπεράσματα και για άλλες κρίσιμες παραμέτρους όπως η ισορροπία του φορτίου μεταξύ των κόμβων του συστήματος και ο μέγιστος αριθμός παράλληλων χρηστών που μπορούν να εξυπηρετηθούν. Για την καλύτερη εκτίμηση των αποτελεσμάτων χρησιμοποιήθηκαν σαν μέτρο σύγκρισης τέσσερις ακόμα λύσεις χρονοδρομολογητών με συγκεκριμένα προτερήματα και αδυναμίες.

6.1 Σχεδιασμός Πειραμάτων

Τα πειράματα που πραγματοποιήθηκαν επιχειρούν να αναδείξουν τη συμπεριφορά του χρονοδρομολογητή σε στατικά και δυναμικά σενάρια. Η προτεινόμενη λύση είχε ως στόχο τη δημιουργία ενός χρονοδρομολογητή που θα μπορούσε να προσφέρει τόσο μια ικανή στατική δρομολόγηση όσο και μια δυναμική διαδικασία προσαρμογής στις διαφορετικές ανάγκες της εφαρμογής κατά τη διάρκεια ζωής της. Για να μπορέσει να είναι δυνατή μία τέτοιου είδους αξιολόγηση, τα πειράματα διενεργήθηκαν σε συνολικά πέντε διαφορετικούς δρομολογητές με μοναδικά χαρακτηριστικά. Σημειώνεται, επίσης, πως παρόλο που βασικός στόχος της πειραματικής διαδικασίας είναι η παρακολούθηση της συμπεριφοράς της προτεινόμενης λύσης, τα πειράματα δομήθηκαν με τέτοιο τρόπο ώστε επιπρόσθετα να βελτιώσουν την κατανόησή μας σε γενικότερο βαθμό σχετικά με τη δρομολόγηση σε περιβάλλον Kubernetes.

6.1.1 Μετρική Πειραματικής Διαδικασίας

Η βασική μετρική πάνω στην οποία δομήθηκε το πειραματικό στάδιο της εργασίας αυτής είναι ο μέσος χρόνος απόκρισης του συστήματος. Η επιλογή αυτή έγινε για τους παρακάτω συγκεκριμένους λόγους: Αρχικά, όταν πρόκειται για μια δικτυακή εφαρμογή και πόσο μάλλον για μία εφαρμογή που απευθύνεται σε καταναλωτές, ο χρόνος απόκρισης αποτελεί πολύ κρίσιμο παράγοντα για τη διαμόρφωση της συνολικής εμπειρίας του χρήστη. Επιπλέον, ο μικρότερος χρόνος απόκρισης μεταφράζεται και σε δυνατότητα εξυπηρέτησης μεγαλύτερου αριθμού ερωτημάτων. Συνεπώς, μειώνονται οι ανάγκες για υπολογιστικούς πόρους, και άρα περιορίζονται τα αντίστοιχα κόστη. Τέλος, η μετρική αυτή είναι απλή και αντικειμενική, και συνεπώς διευκολύνει την εξαγωγή συμπερασμάτων σε αντίθεση με πιο σύνθετες μετρικές.

Για να είναι τα πειράματα και οι μετρήσεις όσο το δυνατόν πιο αντικειμενικές και συγκρίσιμες, έγινε σημαντική προσπάθεια ώστε όλα τα υπόλοιπα μεγέθη να παραμείνουν σταθερά (Ceteris paribus). Πιο συγκεκριμένα επιλέχθηκε να χρησιμοποιηθεί σταθερός αριθμός requests σε όλες τις περιπτώσεις. Επιπλέον, τα πειράματα πραγματοποιήθηκαν με τέτοιο τρόπο ώστε, ακόμα και στις περιπτώσεις των άπληστων δρομολογητών, κανένας από τους κόμβους να μην λειτουργεί σε οριακές συνθήκες. Έτσι, λοιπόν, αποφεύχθηκαν πειράματα που στις πιο

άπληστες υλοποιήσεις δημιουργούσαν bottlenecks σε κάποιον από τους κόμβους του cluster, αφού κρίθηκε ότι τέτοιου είδους αποτελέσματα δεν θα μπορούσαν να αναλυθούν ως προς τον χρόνο απόκρισης της εφαρμογής. Παρόλα αυτά ο σχετικός πειραματισμός έδωσε και μερικά κρίσιμα συμπεράσματα για το συγκεκριμένο ζήτημα, τα οποία θα αναλυθούν παρακάτω.

6.1.2 Χρονοδρομολογητές

Οι παρακάτω πέντε δρομολογητές συμμετείχαν στα πειράματα:

Kube-scheduler: Ο πρώτος δρομολογητής που επιλέχθηκε είναι ο default kube-scheduler. Όπως αναφέρθηκε εκτενώς στα προηγούμενα κεφάλαια, ο συγκεκριμένος δρομολογητής έχει περιορισμούς σχετικά με τη πληροφορία που έχει στη διάθεσή του, και συνεπώς οι επιλογές που κάνει βασίζονται σε μη δυναμικά δεδομένα σχετικά με τη λειτουργία της εφαρμογής. Η σύγκριση μαζί του δίνει μια καλή εικόνα για τις βελτιώσεις που μπορεί να πετύχει μία εφαρμογή και μία ομάδα που θα επιλέξει να υιοθετήσει την προτεινόμενη λύση σε σχέση με την προκαθορισμένη που προσφέρει το Kubernetes.

Random scheduler: Στο πλαίσιο της εργασίας αυτής δημιουργήθηκε επιπλέον ένας απλός τυχαίος δρομολογητής. Ο δρομολογητής αυτός αναθέτει με απόλυτα τυχαίο τρόπο στους κόμβους του cluster τα pods που αναμένουν δρομολόγηση. Ο λόγος που συμπεριλήφθηκε στα πειράματα είναι για να λειτουργήσει σαν μέτρο σύγκρισης τόσο σχετικά με την προτεινόμενη λύση όσο και με τον kube-scheduler. Συγκεκριμένα η σύγκριση με τον kube-scheduler έχει αξία αφού θα αναδείξει το κατά πόσο οι επιλογές που κάνει η default λύση με τα περιορισμένα δεδομένα που έχει στη διάθεσή της προσφέρουν πλεονεκτήματα απέναντι σε μία εντελώς τυχαία προσέγγιση σε ότι αφορά το χρόνο απόκρισης της εφαρμογής.

Round-Robin scheduler: Πέρα από τον Random scheduler δημιουργήθηκε και ένας ακόμα δρομολογητής. Ο Round-Robin scheduler λειτουργεί δρομολογώντας τα pods στα αντίστοιχα nodes με κυκλικό τρόπο. Αναλυτικότερα έχει μία λίστα με όλους τους διαθέσιμους κόμβους του συστήματος και, κάθε φορά που καλείται να δρομολογήσει ένα νέο pod, το αναθέτει στον επόμενο κόμβο της λίστας από εκείνον στον οποίο είχε δρομολογήσει την προηγούμενη φορά. Ο δρομολογητής αυτός έχει σε κάποιο βαθμό παρόμοια λειτουργία με τον τυχαίο (Random scheduler), αφού και οι δύο αποφασίζουν χωρίς να χρησιμοποιούν κάποια πληροφορία σχετικά με το cluster. Η βασική διαφορά τους είναι πως ο Round-Robin εξασφαλίζει ομοιόμορφο διαμοιρασμό των pods σε nodes, κάτι που στην απολύτως τυχαία προσέγγιση δεν είναι δεδομένο. Ο λόγος που συμπεριλήφθηκε ο δρομολογητής αυτός στη πειραματική διαδικασία είναι τόσο για να συγκριθεί η συμπεριφορά του με την προτεινόμενη αρχιτεκτονική όσο και με την default λύση του Kubernetes.

NetMARKS scheduler: Ο επόμενος δρομολογητής που χρησιμοποιήθηκε είναι αυτός που παρουσιάζεται στη δημοσίευση του NetMARKS. Όπως αναφέρθηκε αναλυτικά σε προηγούμενα κεφάλαια, ο δρομολογητής αυτός προσφέρει μια διαφορετική λύση σε σχέση με τον kube-scheduler. Χρησιμοποιεί δεδομένα σχετικά με την επικοινωνία των pods του cluster και δρομολογεί προσπαθώντας να μεγιστοποιήσει την επικοινωνία μεταξύ pods που ανήκουν στο ίδιο node. Η προσέγγισή του είναι σχετικά άπληστη και επιλέγει να υπερφορτώσει κάποιους από τους κόμβους του συστήματος. Στη δημοσίευση αναφέρονται βελτιώσεις στο

χρόνο απόκρισης της εφαρμογής σε ποσοστό έως και 30%. Ο δρομολογητής αυτός όμως έχει και έναν βασικό περιορισμό: η λειτουργία του δεν είναι δυναμική. Έτσι λοιπόν χρειάζεται να τρέξει πρώτα η εφαρμογή και να μαζευτούν τα απαιτούμενα δεδομένα και στη συνέχεια δημιουργείται ο χάρτης δρομολόγησης, ο οποίος μάλιστα παραμένει σταθερός σε όλη τη διάρκεια ζωής της εφαρμογής. Στο πλαίσιο της εργασίας αυτής χρησιμοποιήθηκε μία υλοποίηση του NetMARKS σε Python. Η υλοποίηση αυτή αφορά έναν εξωτερικό δρομολογητή που χρησιμοποιεί τον αλγόριθμο του NetMARKS, χωρίς όμως να επεκτείνει τον kube-scheduler όπως προτείνεται στην αντίστοιχη δημοσίευση.

Custom scheduler: Τελευταίος είναι ο δρομολογητής που υλοποιήθηκε στο πλαίσιο της εργασίας αυτής. Είναι μία δυναμική προσέγγιση στο πρόβλημα της δρομολόγησης, η οποία χρησιμοποιεί μετρικές για δικτυακή κίνηση μεταξύ των pods και επιχειρεί να δρομολογήσει μειώνοντας της επικοινωνίες μεταξύ pods που βρίσκονται σε διαφορετικά node. Στόχος του είναι η μείωση του χρόνου απόκρισης του συστήματος. Για το πειραματικό στάδιο επιλέχθηκε η διαμόρφωση του δρομολογητή έτσι ώστε να μοιράζει με σχετικά ομοιόμορφο τρόπο το φορτίο στα διαφορετικά nodes του συστήματος. Ένα άλλο χαρακτηριστικό που διαχωρίζει την υλοποίηση αυτή από τις υπόλοιπες είναι πως η λειτουργία του είναι δυναμική. Ανταποκρίνεται δηλαδή στις δυναμικές ανάγκες της εφαρμογής και προσαρμόζει τις επιλογές του σχετικά με τη δρομολόγηση αναλόγως.

6.1.3 Cluster

Για την πειραματική διαδικασία δημιουργήθηκε ένα τοπικό cluster από τέσσερις υπολογιστές-κόμβους. Οι κόμβοι αυτοί επιλέχθηκε να έχουν τα ίδια χαρακτηριστικά τόσο σε ότι αφορά το λειτουργικό τους σύστημα όσο και σχετικά με τους υπολογιστικούς πόρους που είχαν στη διάθεσή τους. Το σενάριο αυτό επιλέχθηκε ως το πιο ρεαλιστικό. Σε clusters που στεγάζονται σε τοπικούς servers στο πλαίσιο κάποιας εταιρείας ή οργανισμού και ιδιαίτερα σε αντίστοιχα που παρέχουν οι cloud providers, η ύπαρξη εικονικών υπολογιστών με παρόμοια χαρακτηριστικά είναι ένα αρκετά πιθανό σενάριο. Συνεπώς, η επιλογή αυτή δίνει στα αποτελέσματα των πειραματισμών ακόμα μεγαλύτερη πρακτική αξία.

Η επόμενη επιλογή σχετικά με το τοπικό cluster αφορά την τοποθεσία που βρίσκονται οι υπολογιστές. Το αρχικό σχέδιο ήταν όλα τα nodes να στεγάζονται σαν εικονικά μηχανήματα πάνω στον ίδιο server. Με τον τρόπο αυτό θα εξασφαλιζόταν ότι οι επικοινωνίες μεταξύ των διαφορετικών κόμβων σε κάθε περίπτωση θα εισήγαγαν παρόμοιες καθυστερήσεις. Λόγω περιορισμών υλικού κάτι τέτοιο δεν ήταν εφικτό. Η επόμενη καλύτερη επιλογή ήταν όλα τα μηχανήματα να βρίσκονται σε διαφορετικούς φυσικούς servers που να ισαπέχουν δικτυακά όσο το δυνατόν περισσότερο μεταξύ τους. Για την επίτευξη του στόχου αυτού χρησιμοποιήθηκε το εργαλείο Goldpinger που αναφέρθηκε σε προηγούμενο κεφάλαιο.

6.1.4 Στατικό και Δυναμικό Σενάριο Λειτουργίας

Η πειραματική διαδικασία οργανώθηκε κάτω από δύο βασικά σενάρια, ένα στατικό και ένα δυναμικό.

Το στατικό σενάριο έχει ως στόχο να συγκρίνει τους δρομολογητές για ένα σταθερό παράδειγμα λειτουργίας. Όλες οι λύσεις δρομολογητών που συμμετείχαν στα πειράματα εκτός από την προτεινόμενη έχουν στατική συμπεριφορά. Δηλαδή, δεν ανταποκρίνονται σε δυναμικές αλλαγές που μπορεί να γίνονται σε ό,τι αφορά τον τρόπο λειτουργίας της εφαρμογής. Τέτοιες αλλαγές θα μπορούσαν να προκύψουν τόσο λόγω δομικών αλλαγών στην ίδια την εφαρμογή όσο και λόγω διαφοροποίησης της συμπεριφοράς των χρηστών. Αναλυτικότερα, αν για παράδειγμα σε κάποια καινούργια έκδοση της εφαρμογής εισαχθεί κάποιο νέο pod ή αφαιρεθεί κάποιο άλλο από το σύστημα, ή αν μια προωθητική ενέργεια που τρέχει στην ιστοσελίδα της εφαρμογής δημιουργήσει μεγαλύτερη κίνηση σε ένα συγκεκριμένο τμήμα της, ο γράφος επικοινωνίας μεταξύ των pods ενδέχεται να μεταβληθεί. Αυτό το σενάριο σίγουρα δεν ευνοεί τη στατική προσέγγιση στο θέμα της δρομολόγησης. Κρίθηκε λοιπόν σκόπιμο, πριν το δυναμικό σενάριο, να μελετηθεί η συμπεριφορά των δρομολογητών σε ένα στατικό. Τα δεδομένα αυτά θα είναι χρήσιμα τόσο σε θεωρητικό επίπεδο για τη γενικότερη κατανόηση του προβλήματος και των προτεινόμενων δρομολογητών όσο και σε πρακτικό κυρίως για τις εφαρμογές εκείνες που είναι προδιαγεγραμμένο πως θα έχουν στατικό τρόπο λειτουργίας.

Το δυναμικό σενάριο λειτουργίας έχει ως στόχο να αξιολογήσει την αποδοτικότητα της προτεινόμενη λύσης και παράλληλα να αναδείξει την αδυναμία των υπόλοιπων σε ένα αρκετά ρεαλιστικό σενάριο. Βασική απαίτηση για πολλές σύγχρονες αρχιτεκτονικές είναι η συνεχής διαθεσιμότητά τους. Για την επίτευξη του στόχου αυτού έχει καθιερωθεί και ο όρος Rolling Update ή Rolling Release. Η ιδέα είναι ότι σε περιπτώσεις αναβαθμίσεων τα παλιά pods του συστήματος αντικαθίστανται σταδιακά από καινούρια που φέρουν τη νέα έκδοση, χωρίς όμως ενδιάμεσα να υπάρχουν αδρανή διαστήματα. Στόχος είναι η ελαχιστοποίηση του downtime. Επιπλέον, η ανάπτυξη εφαρμογών με αρχιτεκτονικές microservices αυξάνει τη συχνότητα των αναβαθμίσεων -μειώνοντας το μέγεθός τους-, αφού διαφορετικές υπηρεσίες μπορεί να αναβαθμίζονται ξεχωριστά από διαφορετικές ομάδες. Έτσι λοιπόν ένας στατικός δρομολογητής, όπως ο προτεινόμενος από το NetMARKS, που απαιτεί δοκιμαστική περίοδο συλλογής δεδομένων και επανεκκίνηση ολόκληρης της εφαρμογής για να λειτουργήσει, έρχεται αντίθετος με την παραπάνω επιδίωξη. Σε τέτοια περιβάλλοντα ο NetMARKS θα μπορούσε να δουλέψει χρησιμοποιώντας μόνο την αρχική του βέλτιστη προσέγγιση, σταθερή σε όλη τη διάρκεια ζωής της εφαρμογής. Το δυναμικό σενάριο έρχεται να αναδείξει πως μια δυναμική λύση δρομολόγησης θα μπορούσε να έχει σημαντικές βελτιώσεις απέναντι σε μία στατική σε ότι αφορά το μέσο χρόνο απόκρισης της εφαρμογής.

6.1.5 Locust

Το εργαλείο που επιλέχθηκε για την πραγματοποίηση των πειραμάτων είναι το Locust. Το Locust [34] είναι ένα ισχυρό εργαλείο αυτοματοποιημένων δοκιμών φόρτου (load testing) που χρησιμοποιείται για την αξιολόγηση της απόδοσης και της αντοχής εφαρμογών. Το εργαλείο αυτό επιλέχθηκε τόσο γιατί είναι εύκολο στη χρήση του όσο και για το ότι προσφέρει τη δυνατότητα για πολύπλοκα σενάρια δοκιμών. Με τη χρήση του μπορεί να αναπαρασταθεί η συμπεριφορά των χρηστών και να πραγματοποιηθούν ρεαλιστικά σενάρια δοκιμών.

Συγκεκριμένα με χρήση κώδικα Python ο προγραμματιστής μπορεί να χτίσει σενάρια που αποτελούνται από διαφορετικά requests προς την εφαρμογή. Η δυνατότητα δημιουργίας τους με χρήση της Python και των βιβλιοθηκών της προσφέρει σημαντικές διευκολύνσεις και δυνατότητες. Κατά τη διάρκεια εκτέλεσης είτε μέσω command line είτε μέσω του γραφικού περιβάλλοντος ο χρήστης μπορεί να ξεκινήσει τη λειτουργία του εργαλείου καθορίζοντας τον αριθμό των εικονικών χρηστών που θα δημιουργηθούν, τη ταχύτητα δημιουργίας τους, το σημείο που βρίσκεται η εφαρμογή και την επιθυμητή διάρκεια της δοκιμαστικής διαδικασίας. Στη συνέχεια όλοι οι χρήστες, λειτουργώντας παράλληλα, χρησιμοποιούν το σενάριο που έχει οριστεί και προβαίνουν στα αντίστοιχα requests μέχρι το τέλος της δοκιμαστικής διαδικασίας. Η συνολική λειτουργία έχει ως στόχο να προσομοιώσει την παράλληλη αλληλεπίδραση πολλών χρηστών με την εφαρμογή όπως θα γινόταν σε ένα παραγωγικό περιβάλλον. Τα τελικά αποτελέσματα παρουσιάζονται μέσω ενός αρχείου csv με ξεκάθαρο και οργανωμένο τρόπο που διευκολύνει την κατανόηση και την ανάλυσή τους.

Για τη συγκεκριμένη πειραματική διαδικασία δημιουργήθηκε ένα ρεαλιστικό σενάριο χρήσης της εφαρμογής. Στο σενάριο αυτό καλούνται με τυχαίο τρόπο, λαμβάνοντας υπόψη τα διαφορετικά βάρη, υπορουτίνες που περιέχουν τα requests που αντιστοιχούν σε κινήσεις του χρήστη στην ιστοσελίδα. Τα βάρη για τις διαφορετικές κινήσεις ορίστηκαν κατάλληλα έτσι ώστε η εκτέλεση να προσομοιάζει τη μέση αλληλεπίδραση ενός χρήστη με την εφαρμογή.

Επιπλέον, ορίστηκε σταθερός ρυθμός requests από κάθε χρήστη προς την εφαρμογή. Ο λόγος είναι ότι ο μέσος χρόνος απόκρισης και ο αριθμός των requests που μπορεί να διαχειριστεί μια εφαρμογή είναι ποσά αντιστρόφως ανάλογα. Αν οι χρήστες δεν είχαν σταθερό ρυθμό στα requests που εκτελούν τότε οι καλύτεροι χρόνοι απόκρισης θα οδηγούσαν σε αύξηση του συνολικού αριθμού των requests και συνεπώς σε μεγαλύτερη επιβάρυνση του συστήματος και αντίστοιχη μείωση των χρόνων απόκρισης. Με την επιλογή του σταθερού ρυθμού κρατάμε ίδιο τον αριθμό των συνολικών requests και την επιβάρυνση του συστήματος, και συνεπώς τα αποτελέσματα που προκύπτουν είναι πιο ξεκάθαρα και εύκολα στην ανάλυσή τους.

Μια ακόμα σημαντική σχεδιαστική απόφαση είναι να τρέξει το service που θα παράγει τα requests μέσα από το cluster. Με τον τρόπο αυτό μειώνεται η πιθανότητα να επηρεαστούν τα τελικά αποτελέσματα από αστάθειες της τοπικής σύνδεσης του υπολογιστή που χρησιμοποιήθηκε . Από την άλλη, η σύνδεση στο κέντρο όπου στεγάζονται οι υπολογιστές του cluster έχει αυξημένες πιθανότητες να είναι πιο σταθερή.

6.2 Πειραματική διαδικασία

Η πειραματική διαδικασία χωρίστηκε σε δύο τμήματα, όπως αναφέρθηκε και στην προηγούμενη ενότητα. Η προετοιμασία και για τις δύο περιπτώσεις ήταν η ίδια. Η βασική διαφοροποίηση που υπήρξε ήταν στο σενάριο δοκιμών, το οποίο είχε κάποιες παραλλαγές για το δεύτερο τμήμα των πειραμάτων έτσι ώστε να προσομοιώσει τη δυναμική συμπεριφορά.

6.2.1 Προετοιμασία Περιβάλλοντος

Για το πειραματικό στάδιο χρησιμοποιήθηκε το τοπικό περιβάλλον που έχει αναφερθεί και σε προηγούμενα κεφάλαια. Εν συντομία, το περιβάλλον αυτό διαμορφώθηκε ως εξής:

- Τέσσερα εικονικά μηχανήματα αποτελούν τους κόμβους του cluster. Κάθε υπολογιστής έχει για λειτουργικό σύστημα Linux αρχιτεκτονικής amd64 και τρέχει τα Ubuntu 22.04.2 LTS. Επιπλέον, διαθέτει δύο εικονικούς επεξεργαστές (vCPUs), 4GB μνήμη και 20GB αποθηκευτικό χώρο. Τα μηχανήματα αυτά βρίσκονται σε διαφορετικούς φυσικούς εξυπηρετητές και ισαπέχουν δικτυακά, όσο το δυνατόν περισσότερο μας επιτρέπουν οι περιορισμοί του περιβάλλοντος που χρησιμοποιούμε.
- Τα μηχανήματα αυτά δημιουργούν το cluster. Ενορχηστρωτής του cluster είναι το Kubernetes και συγκεκριμένα είναι εγκατεστημένη η έκδοση Microk8s. Το Microk8s είναι μία ελαφριά διανομή του Kubernetes που διευκολύνει την εγκατάστασή του και αναπτύχθηκε από την εταιρεία Canonical.
- Επιπλέον, έχει εγκατασταθεί το Istio Service Mesh που λειτουργεί σαν sidecar injection πάνω στο Kubernetes.
- Μέσω του API που προσφέρει το Istio έχουν εγκατασταθεί τα εργαλεία Prometheus, Kiali και Grafana.
- Επιπλέον, έχει εγκατασταθεί και το εργαλείο Goldpinger που παρακολουθεί τους χρόνους απόκρισης των nodes.
- Η εφαρμογή που τρέχει είναι η Online Boutique της Google και συγκεκριμένα η έκδοση που βρίσκεται στο project microservices-demo. Τα services της εφαρμογής εγκαταστάθηκαν στο cluster με τη βοήθεια του εργαλείου Kustomize και των αντίστοιχων YAML αρχείων που τα περιγράφουν.
- Για τις δοκιμές χρησιμοποιήθηκε το εργαλείο Locust με ένα ρεαλιστικό σενάριο χρήσης της εφαρμογής. Οι ενέργειες εκκινούν μέσω της γραφικής διεπαφής, από όπου συλλέγονται και τα τελικά αποτελέσματα σε αρχείο μορφής CSV.
- Το service loadgenerator που τρέχει το Locust καθώς και τα αντίστοιχα pods επιλέχθηκε να βρίσκονται συνεχώς σε έναν από τους τέσσερις κόμβους του συστήματος μαζί με το control plane που εποπτεύει τη λειτουργία του cluster. Στον κόμβο αυτό δεν είχαν τη δυνατότητα να στηθούν άλλα pods και συνεπώς η δρομολόγηση ήταν δυνατή μόνο στα υπόλοιπα τρία nodes. Με τον τρόπο αυτό εξασφαλίσαμε το ότι δεν θα ευνοείται κάποιο request μέσω του Locust περισσότερο από τα υπόλοιπα.

Αξίζει να σημειωθεί ότι τόσο στο στατικό όσο και στο δυναμικό σενάριο, τα πειράματα εκκίνησαν με τα pods δρομολογημένα με τον βέλτιστο τρόπο τόσο για το δρομολογητή του NetMARKS όσο και για τον custom. Η επιλογή αυτή διευκόλυνε σε μεγάλο βαθμό τη διαδικασία σύγκρισης των αποτελεσμάτων.

6.2.2 Στατικό Σενάριο

Για το στατικό σενάριο χρησιμοποιήθηκαν και οι πέντε διαθέσιμοι δρομολογητές. Το αρχείο Locust επιχείρησε να αναπαράγει τη μέση αλληλεπίδραση ενός χρήστη με την εφαρμογή. Επειδή πρόκειται για μία σελίδα e-commerce, οι βασικές ενέργειες που μπορούν να κάνουν οι χρήστες είναι οι εξής: περιήγηση στα προϊόντα και προβολή λεπτομερειών για κάποια από αυτά, προσθήκη στο καλάθι, αλλαγή νομίσματος προβολής των τιμών, ολοκλήρωση αγοράς, πληρωμή και παρακολούθηση της αγοράς. Σε ένα ρεαλιστικό σενάριο με πραγματικούς χρήστες οι παραπάνω κινήσεις δεν εμφανίζονται με την ίδια συχνότητα. Έτσι, χρησιμοποιήθηκαν βάρη στις ενέργειες για να καθορίσουν τις αντίστοιχες πιθανότητες εμφάνισης τους.
Για κάθε δρομολογητή έτρεξαν δέκα ίδια πειράματα, από τα οποία προέκυψε ο μέσος όρος που έδωσε τον τελικό χρόνο απόκρισης για τη συγκεκριμένη λύση. Το κάθε πείραμα διήρκεσε πέντε λεπτά. Για κάθε πείραμα δημιουργήθηκαν μέσω της γραφικής διεπαφής είκοσι χρήστες με ρυθμό δημιουργίας τεσσάρων χρηστών το δευτερόλεπτο και σταθερό ρυθμό παραγωγής μίας κίνησης το δευτερόλεπτο. Ο ρυθμός αυτός οδήγησε τα περισσότερα πειράματα τελικά να έχουν από 7300 έως 7400 requests, αφού μερικές κινήσεις περιείχαν πάνω από ένα request.

6.2.3 Δυναμικό Σενάριο

Για το δυναμικό σενάριο επιλέχθηκε να μην χρησιμοποιηθούν όλοι οι διαθέσιμοι δρομολογητές. Η επιλογή αυτή έγινε για τον εξής λόγο: Οι δύο τυχαίες λύσεις του Random scheduler και του Round-Robin scheduler καθώς και ο default kube-scheduler δρομολογούν τα pods αντιμετωπίζοντας το cluster σαν μαύρο κουτί ως προς το δυναμικό τρόπο λειτουργίας του. Έτσι λοιπόν οι επιλογές για τη δρομολόγηση παραμένουν οι ίδιες ανεξάρτητα από τη δομή της εφαρμογής και από το πώς οι χρήστες την χρησιμοποιούν (ο kube-scheduler όπως αναλύθηκε σε προηγούμενο κεφάλαιο επιλέγει χρησιμοποιώντας μόνο στατικά δεδομένα που έχει στη διάθεσή του). Με βάση το σκεπτικό αυτό σε ένα δυναμικό σενάριο αναμένεται η αποτελεσματικότητά τους συγκριτικά με μία δυναμική λύση να είναι ίδια σε κάθε περίπτωση ακριβώς επειδή οι διαφορετικές καταστάσεις δεν επηρεάζουν με κάποιο τρόπο τη διαδικασία επιλογής τους. Για τον λόγο αυτό και σε συνδυασμό με το ότι τα πειράματα για το δυναμικό μέρος είχαν υπερδιπλάσια διάρκεια για να μπορέσουν να πραγματοποιηθούν οι αλλαγές, επιλέχθηκε οι πειραματισμοί να γίνουν για τους δρομολογητές NetMARKS, static-custom και custom. Ο static custom είναι ο δρομολογητής που υλοποιήθηκε στο πλαίσιο της εργασίας αυτής αν του αφαιρεθεί ο δυναμικός του χαρακτήρας. Ουσιαστικά αυτός ο δρομολογητής δοκιμάστηκε και στο στατικό σενάριο αφού η σταθερή συμπεριφορά της εφαρμογής δεν δημιούργησε ανάγκη για δυναμικές παρεμβάσεις. Ο λόγος που επιλέχθηκε να συμπεριληφθεί και αυτός ο δρομολογητής στα πειράματα είναι για να αναδειχθεί με τον πιο ξεκάθαρο τρόπο η επιρροή στο χρόνο απόκρισης που μπορεί να έχει ένας δυναμικός χρονοδρομολογητής σε ένα τέτοιο σενάριο.
Ο τρόπος που επιλέχθηκε για να προσομοιωθεί η δυναμική συμπεριφορά της εφαρμογής είναι η αλλαγή του τρόπου αλληλεπίδρασης των χρηστών με αυτή μέσω αλλαγής του σεναρίου του

Locust. Οι χρήστες, αφού παρέλθει ένα συγκεκριμένο χρονικό διάστημα, αλλάζουν τον τρόπο με τον οποίο χρησιμοποιούν την εφαρμογή. Έτσι, αλλάζει ο γράφος επικοινωνίας μεταξύ των pods και συνεπώς η βέλτιστη λύση δρομολόγησης. Όμως, το να πραγματοποιηθεί μία τέτοια αλλαγή που να επηρεάζει το γράφο επικοινωνίας χωρίς τη δημιουργία νέων pods δεν είναι εύκολο. Οι επικοινωνίες μεταξύ των services είναι συγκεκριμένες. Έτσι λοιπόν μετά από πολλούς πειραματισμούς προέκυψε η ιδέα που χρησιμοποιήθηκε τελικά.
Στην εκκίνηση της διαδικασίας κάποια από τα services παραμένουν εντελώς ανενεργά. Συνεπώς οι αλγόριθμοι των δρομολογητών δεν τα λαμβάνουν υπόψιν στη διαδικασία εύρεσης της βέλτιστης λύσης. Τα services αυτά τελικά δρομολογούνται με τυχαίο τρόπο. Όταν παρέλθει ένα συγκεκριμένο χρονικό διάστημα, οι χρήστες εκκινούν τα requests και προς τα services αυτά. Η διαδικασία αυτή αλλάζει τα δεδομένα και δημιουργεί νέες λύσεις δρομολόγησης, στις οποίες μόνο μια δυναμική προσέγγιση μπορεί να ανταποκριθεί. Για την υλοποίηση της ιδέας αυτής τροποποιήθηκε το αρχείο του Locust που χρησιμοποιήθηκε στο στατικό σενάριο με τρόπο τέτοιο ώστε οι χρήστες να αγνοούν κάποιες κινήσεις για τα τρία πρώτα λεπτά. Στη συνέχεια τις ενσωματώνουν στη συμπεριφορά τους μέχρι την ολοκλήρωση του πειράματος.
Για το δυναμικό κομμάτι των πειραμάτων χρησιμοποιήθηκαν είκοσι χρήστες με ρυθμό παραγωγής κινήσεων μία κίνηση ανά δευτερόλεπτο και ρυθμό δημιουργίας χρηστών τους τέσσερις χρήστες το δευτερόλεπτο. Τα πειράματα έτρεξαν για δέκα λεπτά έτσι ώστε να προλάβουν να γίνουν οι αλλαγές, να ανταποκριθούν ή όχι οι δρομολογητές, να προβούν στις αντίστοιχες κινήσεις και να αποτυπωθούν οι κινήσεις αυτές στα αποτελέσματα. Όπως και στο στατικό σενάριο, έτσι και στο δυναμικό έτρεξαν δέκα πειράματα για κάθε δρομολογητή, και το τελικό αποτέλεσμα προέκυψε ως μέσος όρος των πειραμάτων αυτών.

6.3 Αποτελέσματα

Παρακάτω παρουσιάζονται τα αποτελέσματα για τα δύο σενάρια πειραμάτων, η ανάλυση των αποτελεσμάτων τους, καθώς και γενικά σχόλια και παρατηρήσεις.

6.3.1 Στατικό Σενάριο

Ο πίνακας με τους μέσους όρους των αποτελεσμάτων για τις δέκα επαναλήψεις των πειραμάτων για κάθε δρομολογητή είναι ο παρακάτω:

Scheduler Name	Request Count	Failure Count	Median Response Time	Average Response Time	Min Response Time	Max Response Time	Average Content Size	Requests/s
Round-Robin	7346	0	120.5	163.314592	43.6852335	3703.072	10212.1618	24.0939159
Random	7333.8	0	139	181.648753	46.281812	3883.28496	10198.8131	24.0475546
Default	7308.2	0	146	188.881094	46.3594536	3829.47861	10238.5188	23.9647616
Netmarks	7412.4	0	97.3	134.83639	31.9418505	1711.41103	10279.6057	24.3066011
Custom	7399.2	0	101.7	136.077671	35.4282652	2016.18896	10204.228	24.2670663

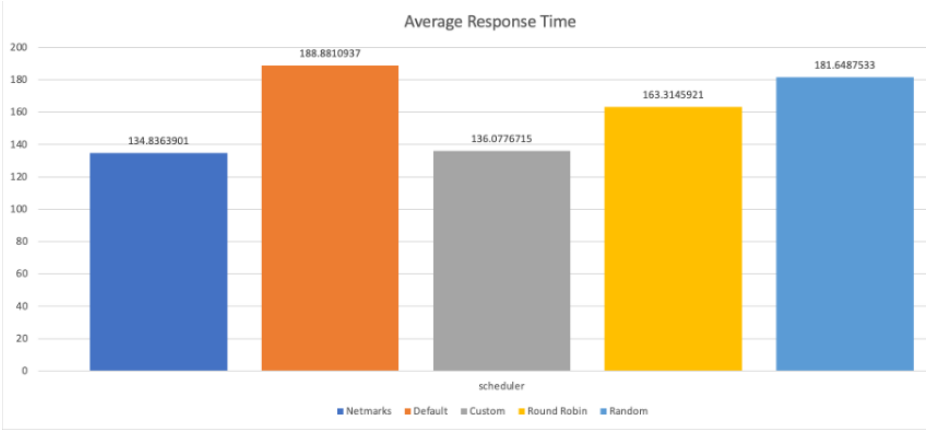
Πίνακας 2. Γενικά αποτελέσματα πειραμάτων για το στατικό σενάριο

Και ο αντίστοιχος πίνακας με τα percentiles για τους χρόνους απόκρισης:

Scheduler Name	50%	66%	75%	80%	90%	95%	98%	99%	99.90%	99.99%	100%
Round-Robin	120.5	155	175	190	254	384	697	927	2650	3700	3700
Random	139	176	200	216	287	407	727	941	2750	3880	3880
Default	146	187	209	228	300	411	738	987	2780	3830	3830
Netmarks	97.3	122	145	163	219	330	643	903	1430	1710	1710
Custom	101.7	128	148	163	212	313	655	877	1630	2020	2020

Πίνακας 3. Αποτελέσματα σε μορφή percentiles για το στατικό σενάριο πειραμάτων

Για την οπτικοποίηση των αποτελεσμάτων σχετικά με το χρόνο απόκρισης της εφαρμογής σε κάθε σενάριο παρατίθεται το ακόλουθο διάγραμμα:



Εικόνα 13. Μέσοι χρόνοι απόκρισης εφαρμογής ανά χρονοδρομολογητή στο στατικό σενάριο πειραμάτων

Εκ πρώτης όψης τα αποτελέσματα φαίνονται ιδιαίτερα θετικά σε ότι αφορά το δρομολογητή της διπλωματικής αυτής. Η απόδοση για το συγκεκριμένο σενάριο έχει βελτίωση της τάξεως του 28% έναντι του default kube-scheduler ενώ βρίσκεται πολύ κοντά με την αντίστοιχη του NetMARKS. Πιο αναλυτικά:

Ο default scheduler φαίνεται να παρουσιάζει τα χειρότερα αποτελέσματα ακόμα και σε σύγκριση με τις τυχαίες λύσεις. Η παρατήρηση αυτή επιβεβαιώνει την αντίληψη γύρω από την οποία δομήθηκε η εργασία αυτή. Τα στατικά δεδομένα που έχει στη διάθεση του δεν παρέχουν ικανή πληροφορία ώστε να οδηγήσουν σε αποδοτικές επιλογές δρομολόγησης, τουλάχιστον σε ότι αφορά το χρόνο απόκρισης της εφαρμογής. Το γεγονός μάλιστα ότι η απόδοσή του είναι χειρότερη ακόμα και από την εντελώς τυχαία προσέγγιση καταδεικνύει το ότι όχι μόνο οι επιλογές που κάνει δεν είναι ιδανικές, αλλά μπορούν μέχρι και να έχουν τα αντίθετα αποτελέσματα.
Σε ότι αφορά τους δύο τυχαίους δρομολογητές παρατηρείται μια σημαντική απόκλιση μεταξύ τους γύρω στο 10%. Το πιθανότερο σενάριο για την απόκλιση αυτή είναι το ότι ο δρομολογητής Round-Robin δρομολογεί μεν με τυχαίο τρόπο αλλά μπορεί οι τυχαίες επιλογές να παραμένουν ίδιες για όλα τα πειράματα. Ένα σενάριο στο οποίο θα μπορούσε να γίνει αυτό θα ήταν αν τα pods και τα nodes συλλέγονταν με αλφαβητική σειρά και συνεπώς η διαδικασία δρομολόγησής τους ήταν συνεχώς η ίδια. Με τον τρόπο αυτό μία καλή τυχαία πρώτη επιλογή δρομολόγησης που επαναλήφθηκε και στα υπόλοιπα πειράματα θα μπορούσε να εξηγήσει τα απρόσμενα καλά αποτελέσματα του δρομολογητή αυτού. Το σενάριο αυτό το επιβεβαιώνουν σε κάποιο βαθμό και οι αναλυτικοί χρόνοι των πειραμάτων στους οποίους οι διακυμάνσεις μεταξύ των δέκα πειραμάτων είναι πολύ μικρότερες σε σχέση με τις αντίστοιχες του Random scheduler. Αντιθέτως ο Random scheduler δρομολογούσε με τυχαίο τρόπο σε κάθε περίπτωση και για τον λόγο αυτό ίσως είναι ο καταλληλότερος για να αποτελέσει μέσο σύγκρισης μαζί με τον default.
Για τον δρομολογητή του NetMARKS επιβεβαιώθηκαν τα πολύ καλά αποτελέσματα έναντι του kube-scheduler που παρουσιάζονται και στην αντίστοιχη δημοσίευση. Συγκεκριμένα η βελτίωση που πέτυχε σχετικά με το χρόνο απόκρισης έναντι του kube-scheduler είναι κοντά στο 29% (στη δημοσίευση παρουσιαζόταν βελτίωση έως και 30%) . Οι αντίστοιχες βελτιώσεις απέναντι στον Random και στον Round-Robin είναι περίπου 26% και 17.5%. Τα αποτελέσματα αυτά αναδεικνύουν την ανάγκη για δυναμικά δεδομένα και δρομολόγηση με βάση αυτά.
Τέλος, τα αποτελέσματα της προτεινόμενης λύσης φαίνεται να είναι σχεδόν εξίσου καλά με τα αντίστοιχα του NetMARKS. Συγκεκριμένα η βελτίωση απέναντι στον kube-scheduler είναι σχεδόν 28%, απέναντι στους Random και Round-Robin 25% και 17% αντιστοίχως ενώ η επιβάρυνσή του σε σχέση με τον NetMARKS είναι μόλις 1%. Παρότι η βελτίωση των αποτελεσμάτων ήταν αναμενόμενη λόγω της προσέγγισης που έγινε στο θέμα της δρομολόγησης η τελική εικόνα ξεπέρασε τις προσδοκίες. Επιτεύχθηκε μέσα από μία μη άπληστη λύση δρομολόγησης να προκύψουν αποτελέσματα πολύ κοντά στα εντυπωσιακά αποτελέσματα του NetMARKS.

Αξίζει να σημειωθεί πως σε όλα τα πειράματα οι χρησιμοποιήσεις των nodes δεν ξεπέρασαν σε κανένα χρονικό σημείο το 85% και συνεπώς μπορούμε να αποκλείσουμε το ενδεχόμενο δημιουργίας bottlenecks. Η επιλογή για είκοσι χρήστες και στις δύο περιπτώσεις δεν έγινε αυθαίρετα. Στόχος ήταν οι δοκιμές να γίνουν με τρόπο τέτοιο ώστε να μην δημιουργούνται για κανένα δρομολογητή bottlenecks στους κόμβους λόγω περιορισμών του υλικού. Με τον τρόπο αυτό κρίθηκε πως τα αποτελέσματα θα ήταν πιο αντικειμενικά και εύκολα ως προς τη σύγκρισή τους. Ο δρομολογητής ο οποίος παρουσίαζε πιο εύκολα τέτοιους περιορισμούς λόγω της προσέγγισής του είναι ο NetMARKS. Συνεπώς μέσω δοκιμών με τον δρομολογητή αυτό προέκυψαν οι είκοσι χρήστες ως ανώτερο επιτρεπτό όριο. Η πληροφορία όμως σχετικά με το μέγιστο αριθμό χρηστών που μπορούσαν να εξυπηρετηθούν σε κάθε περίπτωση με βάση το υφιστάμενο cluster από μόνη της παρουσιάζει ενδιαφέρον. Παρόλο που ο NetMARKS παρουσίαζε χαμηλά ποσοστά χρησιμοποίησης σε κάποιους από τους κόμβους του συστήματος, και συνεπώς θα μπορούσε να παρουσιαστεί σαν επιχείρημα πως για τους κόμβους αυτούς, οι ανάγκες του ήταν μικρότερες σε σχέση με τους άλλους δρομολογητές, η δυνατότητα του να εκμεταλλευτεί ένα cluster από πανομοιότυπους υπολογιστές δεν είναι ιδιαίτερα ικανοποιητική. Αντιθέτως, η προτεινόμενη αρχιτεκτονική με τις συγκεκριμένες ρυθμίσεις πέτυχε πιο ομοιόμορφα ποσοστά χρησιμοποίησης και μέσα από τον πειραματισμό φάνηκε ότι μπορεί να διαχειριστεί πολλαπλάσιο αριθμό χρηστών χωρίς bottlenecks. Στον συγκεκριμένο τομέα η πιο αποδοτική προσέγγιση είναι αυτή του kube-scheduler ο οποίος έχει την δυνατότητα προσαρμογής και ομοιόμορφης χρησιμοποίησης σε όλων των ειδών τα cluster με βάση τα στατικά δεδομένα που χρησιμοποιεί και τον τρόπο που παίρνει αποφάσεις για την δρομολόγηση.

6.3.2 Δυναμικό Σενάριο

Ο πίνακας με τους μέσους όρους των αποτελεσμάτων για τις δέκα επαναλήψεις των πειραμάτων για κάθε δρομολογητή είναι ο παρακάτω:

Scheduler Name	Request Count	Failure Count	Median Response Time	Average Response Time	Min Response Time	Max Response Time	Average Content Size	Requests/s
Netmarks	10374	0	128	168.832665	38.073944	2001.85446	10023.892	17.1499318
Static-Custom	10388.7	0	111.8	148.872538	38.8841039	2617.76053	10021.9386	17.1736914
Custom	10393.9	3.2	84.6	119.819662	34.5181055	2051.10289	10014.9436	17.1822622

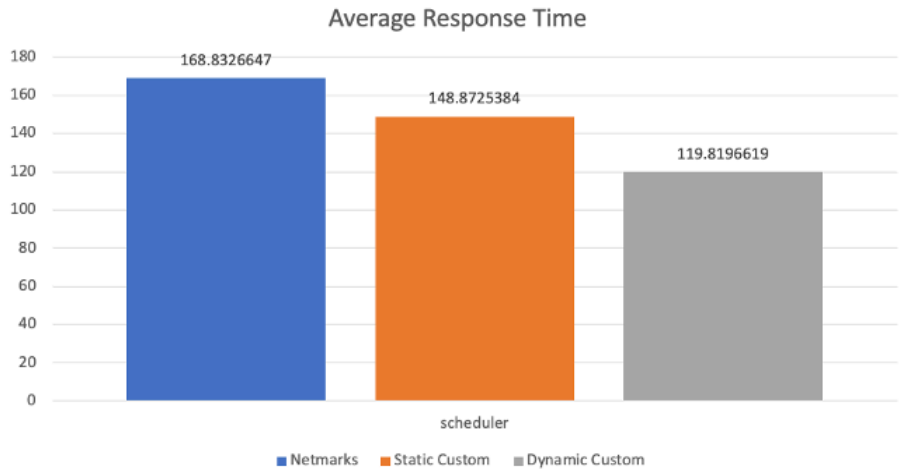
Πίνακας 4. Γενικά αποτελέσματα πειραμάτων για το δυναμικό σενάριο

Και ο αντίστοιχος πίνακας με τα percentiles για τους χρόνους απόκρισης

Scheduler Name	50%	66%	75%	80%	90%	95%	98%	99%	99.90%	99.99%	100%
Netmarks	128	164	192	214	285	387	732	949	1550	1900	2010
Static-Custom	111.8	141	167	185	243	339	635	886	1610	2480	2620
Custom	84.6	106	123	134	181	272	637	884	1490	1980	2060

Πίνακας 5. Αποτελέσματα σε μορφή percentiles για το δυναμικό σενάριο πειραμάτων

Για την οπτικοποίηση των αποτελεσμάτων σχετικά με το χρόνο απόκρισης της εφαρμογής σε κάθε σενάριο παρατίθεται το ακόλουθο διάγραμμα:



Εικόνα 14. Μέσοι χρόνοι απόκρισης εφαρμογής ανά χρονοδρομολογητή στο δυναμικό σενάριο πειραμάτων

Εκ πρώτης όψης και σε αυτή τη περίπτωση τα αποτελέσματα του δρομολογητή που δημιουργήθηκε φαίνεται να είναι ιδιαίτερα καλά. Συγκεκριμένα, ο πλήρης δρομολογητής με τη δυνατότητα για δυναμική ανταπόκριση στις αλλαγές της εφαρμογής παρουσιάζει βελτίωση απέναντι στη στατική έκδοσή του της τάξεως του 19.5% ενώ απέναντι στο NetMARKS βελτίωση της τάξεως του 29%. Πιο αναλυτικά:

Ο NetMARKS scheduler παρουσιάζει τα χειρότερα αποτελέσματα ανάμεσα στους τρεις δρομολογητές. Η εξέλιξη αυτή ήταν αναμενόμενη καθώς όπως έχει αναφερθεί και σε προηγούμενα κεφάλαια η προσέγγιση του NetMARKS είναι ιδιαίτερα άπληστη. Υπερφορτώνει δηλαδή συγκεκριμένα nodes του συστήματος με στόχο την ελαχιστοποίηση του χρόνου απόκρισης, θυσιάζοντας έτσι τον ισομοιρασμό του φόρτου. Το σενάριο πειραμάτων που δημιουργήθηκε είχε στόχο να εκθέσει την προσέγγιση αυτή. Στο αρχικό στάδιο ο NetMARKS δρομολόγησε όλα τα ενεργά pods στον ίδιο κόμβο και τα μη ενεργά στους υπόλοιπους. Όταν η συμπεριφορά των χρηστών άλλαξε και ενεργοποιήθηκαν και τα υπόλοιπα pods, που είχαν υψηλές ανάγκες επικοινωνίας με τα αρχικά, ήταν δρομολογημένα σε διαφορετικούς κόμβους και συνεπώς όλες οι επικοινωνίες τους καθυστερούσαν. Έτσι λοιπόν οι αρχικές βέλτιστες επιλογές ήταν αυτές που οδήγησαν σε άσχημα αποτελέσματα με την

αλλαγή του σεναρίου. Επιπλέον, το γεγονός ότι η δεύτερη κατάσταση διήρκεσε στο 70% του συνολικού χρόνου του πειράματος, οδήγησε σε ακόμα χειρότερα αποτελέσματα. Στην πραγματικότητα ένα τόσο ακραίο σενάριο ίσως και να μην είναι ιδιαίτερα ρεαλιστικό ή να αφορά πολύ συγκεκριμένες περιπτώσεις εφαρμογών. Η επιδίωξη των πειραμάτων ήταν να αναδείξουν το πρόβλημα της συγκεκριμένης λύσης σε μία ακραία του κατάσταση. Η κρισιμότητά του όμως παραμένει μεγάλη. Η κακή απόδοση του NetMARKS, και οποιασδήποτε άλλης μη δυναμικής λύσης, σε περιπτώσεις τόσο δραστικών αλλαγών, δεν έχει τρόπο να βελτιωθεί και συνεπώς χωρίς νεκρά διαστήματα και αλλαγές δεν θα μπορούσε να διορθωθεί το πρόβλημα αυτό με τη χρήση του συγκεκριμένου δρομολογητή.

Σημειώνεται εδώ πως η επιλογή να μπουν τα αρχικά ανενεργά pods σε διαφορετικό κόμβο έγινε στοχευμένα για δύο λόγους. Αρχικά, ο NetMARKS, όπως παρουσιάζεται στην αντίστοιχη δημοσίευση, δεν έχει πρόβλεψη για pods που δεν έχουν ενεργές επικοινωνίες. Έτσι σε συνδυασμό με το ότι είναι υλοποιημένος σαν επέκταση του kube-scheduler, την απόφαση για τα υπόλοιπα pods καλείται να την πάρει ο kube-scheduler, που θα κρίνει με βάση στατικά χαρακτηριστικά όπως για παράδειγμα την χρησιμοποίηση των nodes. Η άπληστη προσέγγιση του NetMARKS επιλέγει να ομαδοποιεί τα περισσότερα pods της εφαρμογής και οδηγεί συγκεκριμένα nodes να λειτουργούν κοντά στα όρια τους. Η αναμενόμενη συμπεριφορά του kube-scheduler σε τέτοια σενάρια είναι να δρομολογήσει με τρόπο τέτοιο ώστε να ισορροπήσει όσο το δυνατόν περισσότερο την κατανομή του φόρτου. Ο δεύτερος λόγος είναι πως το σενάριο πειραματισμού είχε συγκεκριμένο στόχο: να δημιουργήσει μία συνθήκη κατά την οποία ο τρόπος λειτουργίας της εφαρμογής αλλάζει δραστικά. Χωρίς όμως την εκ νέου υλοποίηση η σημαντική τροποποίηση της εφαρμογής δεν θα μπορούσε με φυσικό τρόπο να αλλάξει σε αρκετά μεγάλο βαθμό τον τρόπο με τον οποίον τα pods επικοινωνούν μεταξύ τους. Έτσι λοιπόν επιλέχθηκε να επιτευχθεί ο στόχος αυτός τεχνητά. Η συμπεριφορά αυτή σε συνδυασμό με την αρχική επιλογή για δρομολόγηση των ανενεργών pods σε ελεύθερο node προσομοιώνει ένα σενάριο πραγματικής αλλαγής του γράφου επικοινωνιών της εφαρμογής.

Ο στατικός custom δρομολογητής από την άλλη πλευρά φαίνεται να έχει καλύτερη απόδοση στο σενάριο αυτό. Ο βασικός λόγος είναι πως κατά την αρχική επιλογή δρομολόγησης τα ενεργά pods ισομοιράστηκαν σε κάποιο βαθμό μεταξύ των nodes. Αντίστοιχα, και τα ανενεργά pods δρομολογήθηκαν με Round-Robin τρόπο στους κόμβους αφού δεν υπήρχε κάποια πληροφορία για αυτά. Όταν πλέον τα αρχικά ανενεργά ξεκίνησαν να λαμβάνουν κίνηση και συνεπώς ξεκίνησαν να δημιουργούνται requests μεταξύ των διαφορετικών pods, δεν ήταν δεδομένο πως τα αρχικά ανενεργά και τα αντίστοιχα αρχικά ενεργά που επικοινωνούσαν θα βρίσκονταν σε διαφορετικό κόμβο αλλά εξαρτιόταν από την αρχική τυχαία τοποθέτηση του Round-Robin. Για το λόγο αυτό τα αποτελέσματα ήταν καλύτερα συνολικά από τα αντίστοιχα του NetMARKS, παρόλο που για το αρχικό 30% της διάρκειας παρατηρήθηκε ότι ο NetMARKS ήταν πιο αποδοτικός.

Σημειώνεται ότι στην περίπτωση της στατικής λύσης, σε αντίθεση με το NetMARKS, επιλέχθηκε να δρομολογηθούν τα αρχικά ανενεργά pods με Round-Robin τρόπο. Η απόφαση αυτή προέκυψε αφενός γιατί η συμπεριφορά της προτεινόμενης αρχιτεκτονικής καθορίζει κάτι τέτοιο, σε αντίθεση με του NetMARKS που δεν έχει αντίστοιχη πρόβλεψη, και αφετέρου διότι ακόμα και αν επιλεγόταν ένας από τους κόμβους, αφού όλοι είχαν ήδη δρομολογημένα pods

με τυχαίο τρόπο, θα καθοριζόταν και πάλι πόσο αποδοτική θα ήταν η δρομολόγηση για το δεύτερο τμήμα του πειράματος.

Ο δυναμικός custom δρομολογητής παρουσιάζει τα καλύτερα αποτελέσματα από τις τρεις λύσεις. Αρχικά, συγκρίνοντάς τον με την στατική αντίστοιχη έκδοσή του παρουσιάζεται βελτίωση κοντά στο 19.5%. Η βελτίωση αυτή είναι ιδιαίτερα σημαντική και μπορεί να μεταφραστεί με πολλούς τρόπους ανάλογα με την στόχευση κάθε ομάδας/εταιρείας που αναπτύσσει μία εφαρμογή. Μια τέτοια βελτίωση θα μπορούσε να οδηγήσει σε σημαντική μείωση του κόστους για υλικό. Με τις ίδιες προδιαγραφές η εφαρμογή που θα έτρεχε με τη δυναμική λύση θα είχε σημαντικά καλύτερη απόδοση σε ότι αφορά το χρόνο απόκρισης, ο οποίος με τη σειρά του θα μεταφραζόταν σε δυνατότητα διαχείρισης μεγαλύτερου αριθμού requests ανά μονάδα χρόνου, άρα και υποστήριξη μεγαλύτερου αριθμού χρηστών. Από την άλλη πλευρά αν στόχος είναι η βελτιστοποίηση της εμπειρίας του χρήστη, τότε με τη χρήση των ίδιων εγκαταστάσεων η εφαρμογή θα μπορούσε να πετύχει πολύ καλύτερα αποτελέσματα στην κατεύθυνση αυτή.

Συγκρίνοντας με τον NetMARKS η βελτίωση είναι ακόμα καλύτερη και αγγίζει το 29%. Αξίζει μάλιστα να σημειωθεί πως αν η διάρκεια των πειραμάτων επεκτεινόταν ακόμα περισσότερο και το αρχικό τμήμα στο οποίο ο NetMARKS απέδιδε καλύτερα περιοριζόταν από το 30% σε ένα μικρότερο νούμερο, η διαφορά ενδέχεται να ήταν ακόμα μεγαλύτερη. Το ίδιο ισχύει και στη σύγκριση με τη στατική λύση που αρχικά απέδιδε με παρόμοιο τρόπο με τη δυναμική. Επιπλέον, μεγάλη αξία παρουσιάζει και το γεγονός πως η δυναμική δρομολόγηση πέτυχε το στόχο της για συνεχή διαθεσιμότητα της εφαρμογής. Στα δέκα λεπτά λειτουργίας στα οποία κλήθηκε ο δρομολογητής να κάνει αρκετές αλλαγές για να ανταποκριθεί, κατά μέσο όρο απέτυχαν 3.2 requests από τα συνολικά 10393.9 που επιχειρήθηκαν, που είναι ένα αμελητέο ποσό. Όπως αναφέρθηκε εκτενώς και σε προηγούμενα κεφάλαια, η επίτευξη του στόχου αυτού είναι ιδιαίτερα σημαντική αφού η συνεχής διαθεσιμότητα της εφαρμογής αποτελεί βασική ανάγκη σε πολλές περιπτώσεις.

ΚΕΦΑΛΑΙΟ 7 – Συμπεράσματα και Μελλοντικές Επεκτάσεις

Μέσα από την εργασία αυτή αναδεικνύεται η ανάγκη που υπάρχει τη δεδομένη χρονική στιγμή για λύσεις σχετικά με τη δρομολόγηση σε περιβάλλοντα Kubernetes. Η συνεχώς μεγαλύτερη υιοθέτηση των αρχιτεκτονικών microservices λόγω των πλεονεκτημάτων που προσφέρουν καθιστά ακόμα πιο επιτακτική την ανάγκη αυτή.

Τα ιδιαίτερα θετικά αποτελέσματα που παρουσίασε η προτεινόμενη λύση της εργασίας σε σχέση με τους υπόλοιπους δρομολογητές δείχνουν πως η συγκεκριμένη κατεύθυνση βελτιστοποίησης είναι αρκετά ελπιδοφόρα. Ακόμα και συγκριτικά με τη λύση του NetMARKS, η οποία αποτελεί σημαντική βελτίωση σε σχέση με τον kube-scheduler του Kubernetes, τα περιθώρια βελτίωσης είναι πολύ μεγάλα, κυρίως σε ότι αφορά δυναμικά σενάρια δρομολόγησης. Όταν πρόκειται για πραγματικές εφαρμογές σε περιβάλλοντα παραγωγής, τέτοια μεγέθη βελτίωσης δημιουργούν σημαντικά περιθώρια κέρδους και συνεπώς έχουν μεγάλη αξία.

Πάνω στη βάση που δημιουργήθηκε στο πλαίσιο της εργασίας αυτής θα μπορούσαν να πραγματοποιηθούν προεκτάσεις σε μεταγενέστερο στάδιο με στόχο να επιβεβαιωθεί και να ενισχυθεί η αποτελεσματικότητα και η χρησιμότητα της τελικής λύσης. Μερικές τέτοιες μελλοντικές επεκτάσεις παρουσιάζονται παρακάτω.

Μελέτη σε διαφορετικά cluster και πραγματικό περιβάλλον παραγωγής

Το περιβάλλον του εργαστηρίου στο οποίο έγιναν οι δοκιμές είχε συγκεκριμένους περιορισμούς τόσο σχετικά με τη μη σταθερή παροχή υπολογιστικών πόρων όσο και με την αδυναμία του να συστεγάσει σε έναν server όλους τους υπολογιστές του cluster. Έτσι λοιπόν η δοκιμή σε ένα περιβάλλον που θα προσέφερε τις δυνατότητες αυτές θα προσέδιδε μεγαλύτερη αντικειμενικότητα στα αποτελέσματα. Επιπλέον, η δυνατότητα δοκιμής σε πραγματικό περιβάλλον παραγωγής θα ήταν το επόμενο βήμα για την καλύτερη αξιολόγηση των αποτελεσμάτων. Η συμπεριφορά των χρηστών δεν μπορεί εύκολα να προσομοιωθεί με ακριβή τρόπο, ενώ μέσα από ένα τέτοιο περιβάλλον θα μπορούσε να μελετηθεί καλύτερα και η συμπεριφορά του δρομολογητή σε περιπτώσεις αναβαθμίσεων και αλλαγών στην εφαρμογή.

Μελέτη για διαφορετικές εφαρμογές

Η εφαρμογή που επιλέχθηκε για το πειραματικό στάδιο αποτελεί μία καλή ένδειξη για τις βελτιώσεις που θα μπορούσε να προσφέρει η προτεινόμενη λύση και σε άλλες περιπτώσεις. Παρόλα αυτά, κάθε εφαρμογή έχει τις δικές της ξεχωριστές ανάγκες και τους δικούς της περιορισμούς οπότε η δοκιμή τόσο με διαφορετικές εφαρμογές όσο και γενικότερα με διαφορετικά είδη, θα δημιουργούσε μια καλύτερη εικόνα σχετικά με τα οφέλη της λύσης για διαφορετικά σενάρια.

Δημιουργία χρονοδρομολογητή σαν επέκταση του kube-scheduler με χρήση του kube descheduler

Ο δρομολογητής της εργασίας αυτής δημιουργήθηκε από την αρχή και λειτουργεί ανεξάρτητα από τον kube-scheduler του Kubernetes. Το επόμενο στάδιο για τη βελτίωση της λύσης θα ήταν η υλοποίηση του δρομολογητή σαν επέκταση του υπάρχοντος kube-scheduler μέσω των δυνατοτήτων που προσφέρει. Για την επίτευξη του στόχου αυτού χρειάζεται να γίνει και η αντίστοιχη επέκταση της λύσης του kube descheduler [35]. Ο kube descheduler είναι το εργαλείο που προσφέρει η κοινότητα του Kubernetes για την κατάργηση pods που έχουν δρομολογηθεί με βάση παλαιότερες επιλογές και πλέον χρειάζεται να καταργηθούν. Ο kube descheduler χρειάζεται τον kube-scheduler για την εκ νέου δρομολόγηση των pods αυτών. Έτσι λοιπόν με την χρήση και των δύο εργαλείων θα μπορούσε να επιτευχθεί η πλήρης λειτουργικότητα της προτεινόμενης λύσης. Με τον τρόπο αυτό το τελικό αποτέλεσμα θα ήταν μια ολοκληρωμένη πρόταση δρομολογητή που θα εκμεταλλευόταν τις δυνατότητες της προτεινόμενης λύσης για πιο αποδοτική και δυναμική δρομολόγηση, ενώ παράλληλα θα διέθετε την πλήρη λειτουργικότητα και τις δυνατότητες που προσφέρουν ο kube-scheduler και ο kube descheduler.

Βιβλιογραφία

[1] S. Bennett, “Container Orchestration Statistics 2024 - Everything You Need to Know,” WebinarCare, Mar. 23, 2024. https://webinarcare.com/best-container-orchestration-software/container-orchestration-statistics/#:~:text=It (accessed Mar. 31, 2024).

[2] Ł. Wojciechowski et al., "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh," IEEE INFOCOM 2021 - IEEE Conference on Computer Communications, Vancouver, BC, Canada, 2021, pp. 1-9, doi: 10.1109/INFOCOM42981.2021.9488670.

[3] M. Lin, J. Xi, W. Bai and J. Wu, "Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud," in IEEE Access, vol. 7, pp. 83088-83100, 2019, doi: 10.1109/ACCESS.2019.2924414.

[4] S. Richmond, “Council Post: Uncovering The True Costs Of IT Infrastructure,” Forbes. https://www.forbes.com/sites/forbestechcouncil/2021/11/03/uncovering-the-true-costs-of-it-infrastructure/ (accessed Mar. 31, 2024).

[5] Atlassian, “5 Advantages of Microservices [+ Disadvantages],” Atlassian. https://www.atlassian.com/microservices/cloud-computing/advantages-of-microservices (accessed Mar. 31, 2024).

[6] “Transitioning from Monolithic to Microservices Architecture: A Strategic Move in Retail,” www.ciklum.com. https://www.ciklum.com/resources/blog/transitioning-from-monolithic-to-microservices-architecture-a-strategic-move-in-retail (accessed Mar. 31, 2024).

[7] “The evolution of containers: Docker, Kubernetes and the future,” SearchITOperations. https://www.techtarget.com/searchitoperations/feature/Dive-into-the-decades-long-history-of-container-technology (accessed Mar. 31, 2024).

[8] Amazon, “What is Cloud Computing? - Amazon Web Services,” Amazon Web Services, Inc., 2023. https://aws.amazon.com/what-is-cloud-computing/ (accessed Mar. 31, 2024).

[9] “Microservices vs Monolithic Architecture: Is Deploying Faster and Scalable Applications Your Priority?” SourceFuse, Jan. 21, 2021. https://www.sourcefuse.com/resources/blog/microservices-vs-monolithic-architecture-is-deploying-faster-and-scalable-applications-your-priority/ (accessed Mar. 31, 2024).

[10] VMware, “What is a Virtual Machine? | VMware Glossary,” VMware, Jan. 27, 2022. https://www.vmware.com/topics/glossary/content/virtual-machine.html (accessed Mar. 31, 2024).

[11] “What are containers?” Google Cloud. https://cloud.google.com/learn/what-are-containers (accessed Mar. 31, 2024).

[12] Docker, “What is a Container?” Docker, 2023. https://www.docker.com/resources/what-container/ (accessed Mar. 31, 2024).

[13] [x]cube LABS, “The advantages and disadvantages of containers.,” [x]cube LABS, Feb. 23, 2023. https://www.xcubelabs.com/blog/the-advantages-and-disadvantages-of-containers/ (accessed Mar. 31, 2024).

[14] “IBM Developer,” developer.ibm.com. https://developer.ibm.com/articles/true-benefits-of-moving-to-containers-1/ (accessed Mar. 31, 2024).

[15] “Containers and VMs Together | Docker,” www.docker.com, Apr. 08, 2016. https://www.docker.com/blog/containers-and-vms-together/ (accessed Mar. 31, 2024).

[16] Kubernetes, “Overview,” Kubernetes, Sep. 19, 2023. https://kubernetes.io/docs/concepts/overview/ (accessed Mar. 31, 2024).

[17] “Scheduling Framework,” Kubernetes. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ (accessed Mar. 31, 2024).

[18] “The Istio service mesh,” Istio. https://istio.io/latest/about/service-mesh/ (accessed Mar. 31, 2024).

[19] Prometheus, “Overview | Prometheus,” Prometheus.io, 2012. https://prometheus.io/docs/introduction/overview/ (accessed Mar. 31, 2024).

[20] AWS, “What are Microservices?” Amazon Web Services, Inc., 2019. https://aws.amazon.com/microservices/ (accessed Mar. 31, 2024).

[21] “Deployments,” Kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (accessed Mar. 31, 2024).

[22] “GoogleCloudPlatform/microservices-demo,” GitHub, Nov. 19, 2021. https://github.com/GoogleCloudPlatform/microservices-demo (accessed Mar. 31, 2024).

[23] “Python,” Wikipedia, Mar. 09, 2024. https://el.wikipedia.org/wiki/Python (accessed Mar. 31, 2024).

[24] “Docker,” Wikipedia, Nov. 02, 2021. https://el.wikipedia.org/wiki/Docker (accessed Mar. 31, 2024).

[25] “Docker Engine overview,” Docker Documentation, Apr. 09, 2020. https://docs.docker.com/engine/ (accessed Mar. 31, 2024).

[26] “Documentation,” kiali.io. https://kiali.io/docs/ (accessed Mar. 31, 2024).

[27] “Documentation,” Grafana Labs. https://grafana.com/docs/ (accessed Mar. 31, 2024).

[28] “Grafana Labs Observability Survey 2023 Finds Centralization Saves Time and Money for an Industry Plagued by Tool and Data Source Overload,” Grafana Labs. https://grafana.com/about/press/2023/03/08/grafana-labs-observability-survey-2023-finds-centralization-saves-time-and-money-for-an-industry-plagued-by-tool-and-data-source-overload/#:~:text=Grafana%20(94%25)%20and%20Prometheus (accessed Mar. 31, 2024).

[29] “What are Kubernetes Services? | vmware glossary,” https://www.vmware.com/topics/glossary/content/kubernetes-services.html (accessed Mar. 31, 2024).

[30] Redis, “Redis,” redis.io, 2023. https://redis.io/ (accessed Mar. 31, 2024).

[31] “Kustomize - Kubernetes native configuration management,” kustomize.io. https://kustomize.io/ (accessed Mar. 31, 2024).

[32] “YAML Tutorial: Everything You Need to Get Started in Minutes,” CloudBees, Mar. 08, 2023. https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started (accessed Mar. 31, 2024).

[33] “bloomberg/goldpinger,” GitHub, Mar. 28, 2024. https://github.com/bloomberg/goldpinger (accessed Mar. 31, 2024).

[34] “What is Locust? — Locust 1.6.0 documentation,” docs.locust.io. https://docs.locust.io/en/stable/what-is-locust.html (accessed Mar. 31, 2024).

[35] “Descheduler for Kubernetes,” GitHub, Oct. 31, 2022. https://github.com/kubernetes-sigs/descheduler (accessed Mar. 31, 2024).