Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Επικοινωνιων, Ηλεκτρονικης
Και Συστηματων Πληροφορικης

Κατανεμημένη διαχείριση υποστηρικτικών υπηρεσιών (service mesh) για τη βέλτιστη παροχή εφαρμογών υπολογιστικού νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιλτιάδης, Γ Κοπαλίδης

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Αθήνα, Απρίλιος 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ
ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Κατανεμημένη διαχείριση υποστηρικτικών υπηρεσιών (service mesh) για τη βέλτιστη παροχή εφαρμογών υπολογιστικού νέφους.

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Μιλτιάδης, Γ Κοπαλίδης

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Απριλίου 2023.

Αθήνα, Απρίλιος 2023

....	....	....
Συμεών Παπαβασιλείου	Θεοδώρα Βαρβαρίγου	Γεώργιος Ματσόπουλος
Καθηγητής Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

....

Κοπαλίδης Μιλτιάδης

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

Copyright © Κοπαλίδης Μιλτιάδης
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η αρχιτεκτονική των μικρουπηρεσιών έφερε πολλές νέες προκλήσεις στην βιομηχανία, που σχετίζονται κυρίως με την διαχείριση της δικτυακής κίνησης μεταξύ ανεξάρτητων υπηρεσιών (east-west traffic). Η απάντηση σε αυτές τις προκλήσεις ήρθε μέσω της τεχνολογίας του service mesh. Το service mesh αποτελεί ένα ανεξάρτητο στρώμα (independent infrastructure layer) κάτω από το στρώμα της εφαρμογής (application layer) το οποίο, όχι μόνο καταφέρνει να επιλύσει αδιάφανα το πρόβλημα της μεγάλης δικτυακής κίνησης μεταξύ των υπηρεσιών, αλλά μπορεί και παρέχει μηχανισμούς παρατηρησιμότητας (observability), ασφάλειας (security) και αξιοπιστίας (reliability). Στην παρούσα εργασία εξηγούνται οι λόγοι που οδήγησαν στην υιοθέτηση της service mesh τεχνολογίας από την βιομηχανία, παρουσιάζονται βασικές ιδέες και εργαλεία που συνδυάζονται με την service mesh τεχνολογία, αναλύεται η βασική αρχιτεκτονική της service mesh τεχνολογίας και συγκρίνονται τα πιο διάσημα υφιστάμενα service meshes. Σε αυτό το πλαίσιο, για να υπάρξει πρακτική εξοικείωση με τον μηχανισμό του service mesh χρησιμοποιείται το Linkerd και δοκιμάζονται ο διαμοιρασμός δικτυακής κίνησης (traffic split), οι μηχανισμοί των retries και timeouts και η αυτόματη κλιμάκωση (autoscaling) με βάση τις μετρήσεις που συλλέγει το Linkerd. Για να δοκιμαστούν, να τρέξουν και να διαπιστωθεί η σημασία των μηχανισμών αυτών δημιουργείται ένα Kubernetes cluster δύο φυσικών κόμβων (bare metal Kubernetes cluster) στο οποίο εγκαθίστανται δύο εισαγωγικές (demo) εφαρμογές που αναδεικνύουν την χρησιμότητα των μηχανισμών.

Λέξεις Κλειδιά

Containers, Container-Orchestrators, Kubernetes, Service mesh, Linkerd, αρχιτεκτονική μικρουπηρεσιών, οριζόντια κλιμάκωση, διαμοιρασμός δικτυακής κίνησης, retries, timeouts

Abstract

The microservices architecture generated new challenges in industry of software applications, most of which derived from the need of network traffic management between microservices (east-west traffic). The answer to these problems was a new technology called, service mesh. Service mesh not only solves the network traffic management issue but also offers traffic observability, security and reliability mechanisms. In the current project we explain the reasons why service mesh technology was adopted by the software applications industry, present the core ideas and tools that are needed for service mesh technology, analyze the most common service mesh architecture and compare a few popular service meshes. In order to gain hands-on experience with service mesh, we use Linkerd and try some of its features like TrafficSplit, retries and timeouts, and autoscaling based on its metrics. To do that we establish a bare metal Kubernetes cluster with two nodes in which we setup two demo applications (emojivoto and booksapp).

Keywords

Containers, Container-Orchestrators, Kubernetes, Service mesh, Linkerd, microservices architecture, HorizontalPodAutoscaling, Traffic split, retries, timeouts

Περιεχόμενα

Περίληψη 5
Abstract 6
Περιεχόμενα 7
Λίστα Σχημάτων 8
Λίστα Εικόνων 9
Κεφάλαιο 1 Εισαγωγή 10
1.1 Στόχος της Εργασίας 10
1.2 Δομή της εργασίας 10
Κεφάλαιο 2 Θεωρία 12
2.1 Αρχιτεκτονικές ανάπτυξης εφαρμογών 12
2.1.1 Μονολιθική Αρχιτεκτονική 12
2.1.2 Αρχιτεκτονική των μικρουπηρεσιών (Microservices) 12
2.2 Σύγχρονες τεχνολογίες με εφαρμογή στην αρχιτεκτονική των μικρουπηρεσιών 16
2.2.1 Containers 16
2.2.2 Container Orchestration 17
2.3 Service Mesh 23
2.3.1 Ιστορική αναδρομή 23
2.3.2 Τι είναι το Service Mesh 23
2.3.3 Η αρχιτεκτονική του service mesh 24
2.3.4 Σύγκριση Διάσημων Service meshes 25
2.4 Ανάλυση βιβλιογραφίας 36
Κεφάλαιο 3 Υλοποίηση 38
3.1 Περιγραφή Διάταξης 38
3.1.1 Εισαγωγή 38
3.1.2 Περιγραφή Εφαρμογών 38
3.1.3 Εγκατάσταση Linkerd 39
3.2 Αυτόματη Κλιμάκωση (Autoscaling) 39
3.2.1 Περιγραφή υπηρεσίας 39
3.2.2 Χρησιμότητα της Υπηρεσίας 40
3.2.3 Υλοποίηση 40
3.2.4 Επέκταση 43
3.3 Διαμοιρασμός Δικτυακής κίνησης 43
3.3.1 Περιγραφή Υπηρεσίας 43
3.3.2 Χρησιμότητα της Υπηρεσίας 43
3.3.3 Υλοποίηση 44
3.3.4 Επέκταση 44
3.4 Retries and Timeouts 45
3.4.1 Περιγραφή Υπηρεσίας 45
3.4.2 Χρησιμότητα Υπηρεσίας 45
3.4.3 Υλοποίηση 46
3.4.4 Επέκταση 47
Κεφάλαιο 4 Συμπεράσματα 48
Συμπεράσματα και μελλοντικές επεκτάσεις 48
Βιβλιογραφία 49

Λίστα Σχημάτων

Κεφάλαιο 2

Σχήμα 2.1: Κλιμάκωση των δύο αρχιτεκτονικών [3] 13
Σχήμα 2.2: Εικονικοποίηση των containers και των VMs [12] 16
Σχήμα 2.3: Επικοινωνία του kubelet με το container runtime [16] 21
Σχήμα 2.4: HorizontalPodAutoscaling [17] 21
Σχήμα 2.5: Επικοινωνία μεταξύ χρήστη και extension Apiserver [18] 22
Σχήμα 2.6: Αρχιτεκτονική Service mesh [20] 24
Σχήμα 2.7: Αρχιτεκτονική Linkerd [22] 26
Σχήμα 2.8: Αρχιτεκτονική Istio [24] 27
Σχήμα 2.9: Αρχιτεκτονική Consul Connect [51] 28
Σχήμα 2.10: Αρχιτεκτονική του Kuma [53] 29
Σχήμα 2.13: Ροή Πληροφορίας στο Traefik mesh [55] 30
Σχήμα 2.14: Αρχιτεκτονική του Control plane του OSM [57] 31
Σχήμα 2.15: Αρχιτεκτονική του data plane του OSM [57] 31

Κεφάλαιο 3

Σχήμα 3.1: Φαινομενική χρήση της CPU vs Κανονική Χρήση της CPU [42] 40
Σχήμα 3.2: Ροή πληροφορίας του HPA [44] 41

Λίστα Εικόνων

Κεφάλαιο 3

Εικόνα 3.1: Αιτήματα του Vegeta 42
Εικόνα 3.2: Υπέρβαση του κατωφλιού και εντολή για κλιμάκωση από τον HPA 42
Εικόνα 3.3: Διαμοιρασμός δικτυακής κίνησης μεταξύ των δύο services 44
Εικόνα 3.4: Ποσοστά επιτυχίας πριν την ενεργοποίηση των retries και των timeouts 46
Εικόνα 3.5: Ποσοστά επιτυχίας μετά τα retries και πριν τα timeouts 46
Εικόνα 3. 6: Ποσοστά επιτυχίας μετά τα retries και μετά τα timeouts 46

Κεφάλαιο 1 Εισαγωγή

1.1 Στόχος της Εργασίας

Η πρόοδος και η συνεχή εξέλιξη της τεχνολογίας αλλάζει συνεχώς την κοινωνία. Επίσης ο παγκόσμιος χαρακτήρας της σημερινής κοινωνίας επηρεάζει τις απαιτήσεις των εφαρμογών καθώς το πλήθος των ανθρώπων που έχει πρόσβαση είναι πολύ μεγάλο και γεωγραφικά κατανεμημένο. Η ανάγκη για εφαρμογές που να μπορούν να κλιμακώνονται ώστε να εξυπηρετούν πάρα πολλά αιτήματα από πολλούς χρήστες ταυτόχρονα σε συνδυασμό με την προσπάθεια για καλύτερη οργάνωση του κώδικα των εφαρμογών αποτέλεσαν τα κύρια αίτια για την «ανατροπή» του μοντέλου της μονολιθικής αρχιτεκτονικής και την δημιουργία του μοντέλου των μικρουπηρεσιών. Το μοντέλο αυτό υιοθετήθηκε πριν αρκετά χρόνια από την βιομηχανία και πλέον έχει εδραιωθεί ως το βασικό μοντέλο ανάπτυξης λογισμικού.
Οι σημερινές εφαρμογές καλούνται να ανταποκριθούν στην πολύ μεγάλη ζήτηση, στους μεγάλους όγκους δεδομένων, σε χρονικά ελάχιστες καθυστερήσεις και όλα αυτά πρέπει να γίνονται συνέχεια χωρίς καθόλου διακοπές (αδιάληπτα) και πάντα φροντίζοντας για την ασφάλεια των δεδομένων που ανταλλάσσονται. Αξίζει να παρατηρήσει κανείς ότι όλες οι παραπάνω απαιτήσεις δεν σχετίζονται καθόλου με το περιεχόμενο και τις λειτουργίες της εφαρμογής αλλά με την καλή της υγεία, την διαθεσιμότητά της, την επίδοση και την ασφάλειά της. Το μοντέλο των μικρουπηρεσιών παρόλο που αποτέλεσε την λύση ώστε οι εφαρμογές να μπορούν να ανταπεξέλθουν στις απαιτήσεις της κλιμάκωσης δημιούργησε νέες προκλήσεις οι οποίες αποδίδονται στην μεγάλη αύξηση της δυσκολίας και της πολυπλοκότητας των απαιτήσεων που δεν σχετίζονται με την λογική (business logic) της εφαρμογής (επίδοση, διαθεσιμότητα, ασφάλεια κλπ.).
Για να μπορέσουν να αντιμετωπιστούν αυτές οι απαιτήσεις, νέες τεχνολογικές ιδέες εμφανίστηκαν. Οι πιο διάσημες από αυτές, οι οποίες σήμερα χρησιμοποιούνται κατά κόρον στην βιομηχανία, είναι τα Containers που επιτρέπουν το πακετάρισμα και την εκτέλεση του κώδικα με τον ίδιο τρόπο ανεξαρτήτως υλικού (hardware) και λογισμικού (software), η ενορχήστρωση των Containers (container orchestration) που απευθύνεται σε εργαλεία τα οποία μπορούν και διαχειρίζονται μεγάλα σύνολα από Containers και το Service mesh, το οποίο έρχεται να λειτουργήσει συμπληρωματικά στην ενορχήστρωση των Containers αναλαμβάνοντας να διαχειρίζεται όλη την επικοινωνία μεταξύ των Containers παρέχοντας μάλιστα επιπλέον δυνατότητες παρακολούθησης και ασφάλειας της επικοινωνίας μεταξύ των υπηρεσιών.
Οι νέες πρακτικές που προσπαθούν να αυτοματοποιήσουν τις ενδιάμεσες διαδικασίες που χρειάζεται να περάσει ο κώδικας από την στιγμή που δημιουργείται από τον προγραμματιστή εφαρμογών (developer) μέχρι να βγει στην παραγωγή και παράλληλα φροντίζουν για την υψηλή ποιότητα της εφαρμογής, αναλαμβάνοντας όλες τις απαιτήσεις που σχετίζονται με την επίδοση, την ασφάλεια, την διαθεσιμότητα και την αξιοπιστία της εφαρμογής, δημιούργησαν στην βιομηχανία ένα νέο κλάδο που ονομάζεται DevOps [1].
Έτσι λοιπόν, στις μεγάλες εταιρείες οι προγραμματιστές (developers) μίας εφαρμογής είναι αποκλειστικά υπεύθυνοι για την ανάπτυξη του κώδικα της εφαρμογής και την υλοποίηση της λογικής της (business logic), ενώ όλες τις υπόλοιπες διαδικασίες μαζί με όλες τις απαιτήσεις για την εκτέλεση (deployment) ποιοτικού κώδικα στην παραγωγή τις αναλαμβάνουν οι μηχανικοί DevOps.
Στόχος αυτής της διπλωματικής εργασίας είναι να εισχωρήσουμε στον κόσμο του DevOps εξερευνώντας και αποκτώντας εξοικείωση με το Kubernetes και το Linkerd δύο από τα πιο διάσημα εργαλεία στον κόσμο της ενορχήστρωσης των Containers και του Service mesh.

1.2 Δομή της εργασίας

Η εργασία χωρίζεται σε τέσσερα διαφορετικά κεφάλαια. Το παρόν κεφάλαιο αποτελεί την εισαγωγή της εργασίας δίνοντας μία περιγραφή του τι επιτυγχάνεται στην εργασία. Τα υπόλοιπα κεφάλαια χωρίζονται ως εξής:

- Το κεφάλαιο 2 αποτελεί το θεωρητικό μέρος της εργασίας. Σε αυτό το κεφάλαιο στην ενότητα 2.1 γίνεται μια περιγραφή των αρχιτεκτονικών ανάπτυξης λογισμικού (μονολιθική και αρχιτεκτονική μικρουπηρεσιών) με έμφαση στα αίτια που οδήγησαν στην υιοθέτηση της αρχιτεκτονικής μικρουπηρεσιών από την βιομηχανία. Στην συνέχεια στην ενότητα 2.2 γίνεται μία παρουσίαση των Containers και ενορχήστρωση των Containers. Στην ενότητα της ενορχήστρωσης των Containers γίνεται και μια περιγραφή του πιο διάσημου εργαλείου αυτού του τομέα, του Kubernetes (το οποίο χρησιμοποιήθηκε και στο πρακτικό μέρος της εργασίας). Στην ενότητα 2.3 γίνεται μια ιστορική αναδρομή του Service mesh με στόχο να αναδείξει τους λόγους για τους οποίους δημιουργήθηκε και ενσωματώθηκε από την βιομηχανία. Μετά την ιστορική αναδρομή ακολουθεί ο ορισμός και η παρουσίαση της πιο ευρέως διαδεδομένης αρχιτεκτονικής των Service meshes. Η ενότητα 2.3 ολοκληρώνεται με αναφορά σε δύο έργα (projects) που λειτουργούν συμπληρωματικά στα service meshes, την παρουσίαση της αρχιτεκτονικής των πιο διάσημων service meshes ανοιχτού-κώδικα και ένα μεγάλο πίνακα στον οποίο συγκρίνονται τα χαρακτηριστικά αυτών. Τέλος η ενότητα 2.4 αποτελεί μία ανάλυση βιβλιογραφίας μεταξύ 2021 και 2022. Οι δημοσιεύσεις που αναλύονται έχουν σαν στόχο είτε να βελτιώσουν τα υπάρχοντα service meshes είτε να τα χρησιμοποιήσουν σαν εργαλεία.

- Το κεφάλαιο 3 αποτελεί το πρακτικό μέρος της εργασίας. Σε αυτό το κεφάλαιο περιγράφεται η διάταξη που χρησιμοποιήθηκε για την πρακτική (hands-on) επαφή με το service mesh. Πιο συγκεκριμένα στην ενότητα 3.1 περιγράφονται οι εισαγωγικές (demo) εφαρμογές που χρησιμοποιήθηκαν μαζί με την διάταξη του Kubernetes cluster και την εγκατάσταση του Linkerd (το service mesh που χρησιμοποιήθηκε). Στην ενότητα 3.2 αναλύεται, περιγράφεται και παρουσιάζεται η αυτόματη κλιμάκωση του Kubernetes που παραμετροποιήθηκε ώστε να αξιοποιεί τα δεδομένα που συλλέγει το Linkerd. Στην ενότητα 3.3 γίνεται αντίστοιχα η περιγραφή και η παρουσίαση για τον διαμοιρασμό της δικτυακής κίνησης (traffic split), την δεύτερη υπηρεσία του Linkerd που υλοποιήθηκε. Τέλος στην ενότητα 3.4 προβάλλεται η χρήση των retries και των timeouts που αποτελούν την τρίτη υπηρεσία. Στην τελευταία παράγραφο των ενοτήτων 3.2, 3.3, 3.4 παρουσιάζεται μια επέκταση που θα μπορούσε να γίνει στην εκάστοτε υπηρεσία.

- Στο κεφάλαιο 4 γίνεται μια συνολική αξιολόγηση της διπλωματικής εργασίας και βγαίνουν κάποια συνολικά συμπεράσματα με βάση την δουλειά που έγινε.

Κεφάλαιο 2 Θεωρία

2.1 Αρχιτεκτονικές ανάπτυξης εφαρμογών

2.1.1 Μονολιθική Αρχιτεκτονική

Με τον όρο μονολιθική αρχιτεκτονική χαρακτηρίζουμε το σύνολο των εφαρμογών οι οποίες υλοποιούνται σε ένα μοναδικό πρόγραμμα. Δηλαδή, όλες οι λειτουργίες της εφαρμογής, είτε αυτές αφορούν την λογική της εφαρμογής (business logic) είτε οποιονδήποτε άλλο τομέα (πχ authorization στον τομέα της ασφάλειας), έχουν υλοποιηθεί στο ίδιο πρόγραμμα. Λόγω της κεντρικοποίησης όλων των συστατικών (components) της, μία μονολιθική εφαρμογή είναι σχετικά εύκολο να αναπτυχθεί σε ένα πρώτο επίπεδο. Επίσης σε αρχικό στάδιο, είναι σχετικά εύκολο να ελεγχθεί (test) και τα εκτελεστεί (deploy). Παρόλα αυτά η ύπαρξη όλων των λειτουργιών σε ένα πρόγραμμα και οι εξαρτήσεις που προκύπτουν από αυτή, δημιουργούν πολλά προβλήματα, κάποια εκ των οποίων παρουσιάζονται παρακάτω:

- Επειδή όλες οι υπηρεσίες βρίσκονται στο ίδιο πρόγραμμα αν κάποιος αλλάξει μια υπηρεσία, πρέπει να εγγυηθεί ότι η αλλαγή θα είναι συμβατή με όλες τις υπόλοιπες υπηρεσίες γιατί διαφορετικά η εφαρμογή δεν θα δουλεύει καθόλου (single point of failure).

- Η πολυπλοκότητα του κώδικα της εφαρμογής αυξάνεται σημαντικά όσο προστίθενται νέες υπηρεσίες με αποτέλεσμα η διαδικασία επέκτασης του κώδικα (πχ για την δημιουργία ενός καινούργιου χαρακτηριστικού (feature)) να είναι πολύ πολύπλοκη, χρονοβόρα και κοστοβόρα.

- Οποιαδήποτε νέα έκδοση της εφαρμογής ακόμα και αν το μόνο που αλλάζει είναι μόνο ένα συστατικό (component) της εφαρμογής, απαιτεί επανεκτέλεση (redeployment) όλης της εφαρμογής με αποτέλεσμα όλες οι υπηρεσίες της εφαρμογής να μην είναι διαθέσιμες για ένα διάστημα (availability problem).

- Δεν μπορεί να κλιμακωθεί (scale) αποδοτικά. Συνήθως μία μονολιθική εφαρμογή έχει πολλές υπηρεσίες αλλά μόνο λίγες από αυτές έχουν πολύ μεγάλη ζήτηση. Στην περίπτωση λοιπόν που μόνο μία υπηρεσία (service) χρειάζεται να κλιμακωθεί, η μονολιθική εφαρμογή αναγκάζεται να κλιμακώσει όλες τις υπηρεσίες της με αποτέλεσμα πολλές υπηρεσίες να καταλαμβάνουν πολλούς παραπάνω πόρους (resources) από αυτούς που πραγματικά χρειάζονται [2].

Όλα τα παραπάνω σε συνδυασμό με την πολύ μεγάλη αύξηση της ζήτησης των εφαρμογών και των αυξημένων απαιτήσεων (επιδίωξη μηδενικού νεκρού χρόνου (zero downtime)) αποτέλεσαν τα αίτια για την εγκατάλειψη του μονολιθικού μοντέλου από την βιομηχανία και την επικράτηση του μοντέλου των μικρουπηρεσιών.

2.1.2 Αρχιτεκτονική των μικρουπηρεσιών (Microservices)

Μία πλέον διαδεδομένη αρχιτεκτονική ανάπτυξης λογισμικού είναι αυτή των μικρουπηρεσιών (microservices). Στην αρχιτεκτονική των μικρουπηρεσιών (microservices) η εφαρμογή δομείται από ένα σύνολο ανεξάρτητων υπηρεσιών κάθε μία εκ των οποίων εξυπηρετεί ένα συγκεκριμένο διαφορετικό σκοπό. Η επικοινωνία μεταξύ τους γίνεται με remote procedure calls ή μέσω http request-response [3]. Οι υπηρεσίες συνήθως είναι οργανωμένες με βάση τις επιχειρηματικές

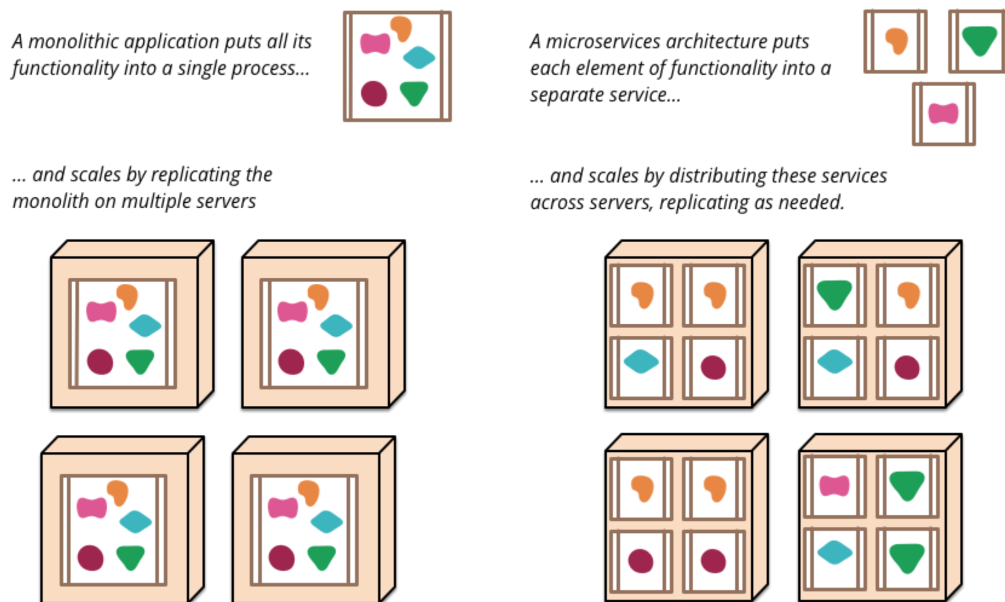
ανάγκες (business capabilities) μιας εταιρείας. Κάθε υπηρεσία μπορεί να έχει το πλήρες πακέτο από διεπαφή χρήστη, λογική εξυπηρετητή και μόνιμο αποθηκευτικό χώρο (user-interface, server logic, persistent storage) και ενδεχομένως κάποια εξωτερική συνεργασία (external collaboration) [3]. Βασική ιδιότητα των μικρουπηρεσιών (microservices) είναι η ανεξαρτησία τους σαν μονάδες, ως προς την ανάπτυξη (development), την εκτέλεση (deployment) και την κλιμακωσιμότητα (scaling). Από αυτή τη βασική ιδιότητα της ανεξαρτησίας προκύπτουν τα παρακάτω χαρακτηριστικά:

- Διαφορετικές ομάδες από developers μπορούν να είναι υπεύθυνες για διαφορετικές υπηρεσίες (services), με αποτέλεσμα να μπορούν να δημιουργούνται παράλληλα χαρακτηριστικά (features) σε διαφορετικές υπηρεσίες (services) [4].

- Όταν ένα νέο χαρακτηριστικό (feature) είναι έτοιμο να ενταχθεί στην εφαρμογή δεν χρειάζεται να γίνει επανεκτέλεση (redeployment) όλης της εφαρμογής (όπως στην μονολιθική αρχιτεκτονική) αλλά μόνο της υπηρεσίας (service) στην οποία ανήκει το συγκεκριμένο χαρακτηριστικό (feature). Συνεπώς, οι χρήστες μπορούν να χρησιμοποιούν τις άλλες υπηρεσίες κατά την διάρκεια της επανεκτέλεσης (redeployment) μίας υπηρεσίας. Επίσης η επανεκτέλεση (redeployment) κρατάει πολύ λίγο οπότε η εφαρμογή έχει μεγάλη διαθεσιμότητα (availability) γιατί το διάστημα που οι χρήστες δεν μπορούν να χρησιμοποιήσουν μία υπηρεσία είναι αρκετά μικρό.

- Κάθε υπηρεσία μπορεί να χρησιμοποιεί διαφορετική γλώσσα προγραμματισμού και διαφορετική βάση δεδομένων [3] ανάλογα με τις ανάγκες της.

- Η κλιμάκωση (scale) των υπηρεσιών (services) μπορεί να γίνει ξεχωριστά για την κάθε μία. Σε αντίθεση με το μονολιθικό μοντέλο που πρέπει να πολλαπλασιάζονται όλα τα συστατικά της εφαρμογής, ακόμα και αν πολλά δεν χρειάζονται, στην αρχιτεκτονική των μικρουπηρεσιών (microservices architecture) μπορούν να πολλαπλασιαστούν μόνο εκείνες οι υπηρεσίες (services) που δέχονται το αυξημένο φορτίο.



Σχήμα 2.1: Κλιμάκωση των δύο αρχιτεκτονικών [3]

- Όταν μία υπηρεσία έχει κάποιο πρόβλημα αυτό δεν μεταβιβάζεται στις υπόλοιπες οι οποίες συνεχίζουν και λειτουργούν κανονικά.

- Είναι εύκολο για τις διάφορες ομάδες προγραμματιστών (developers) να έχουν εποπτεία του κώδικα για τον οποίο είναι υπεύθυνοι και άρα είναι ευκολότερη η συντήρηση του λογισμικού (easy maintenance) [5].

Εκτός όμως από πολλά πλεονεκτήματα η αποκεντροποίηση και η αποδόμηση ενός ενιαίου προγράμματος σε πολλές μικρές ανεξάρτητες υπηρεσίες που επικοινωνούν μεταξύ τους μέσω δικτύου δημιουργεί και νέες προκλήσεις. Παρακάτω παρατίθενται κάποιες από αυτές τις προκλήσεις [6].

- Μη αξιόπιστο δίκτυο
  Πλέον οι διαφορετικές λειτουργικότητες εξυπηρετούνται από διαφορετικές μικρουπηρεσίες, που σημαίνει ότι αναγκαστικά θα πρέπει να υπάρχει επικοινωνία μέσω δικτύου (δεν συνέβαινε στην μονολιθική αρχιτεκτονική). Κάθε στιγμή οποιαδήποτε υπηρεσία μπορεί να αποτύχει, οπότε, θα πρέπει να υπάρχει πλεονασμός (redundancy) των υπηρεσιών (εξασφαλίζεται συνήθως μέσω του replication factor των kubernetes), έτσι ώστε αν ένα στιγμιότυπο (instance) αποτύχει, κάποιο άλλο να είναι διαθέσιμο.

- Μη ασφαλές δίκτυο
  Επειδή η επικοινωνία μεταξύ των μικρουπηρεσιών γίνεται μέσω του δικτύου είναι απαραίτητη η διαδικασία της αυθεντικοποίησης (authentication), ώστε να συνδεόμαστε πράγματι με αυτόν που πιστεύουμε ότι συνδεόμαστε και όχι με κάποιο κακόβουλο χρήστη. Εξίσου απαραίτητη είναι και η διαδικασία του ελέγχου δικαιωμάτων (authorization), ώστε ο κάθε χρήστης να έχει πρόσβαση μόνο σε πληροφορίες που τον αφορούν. Επίσης σε αυτές τις συνθήκες δεν νοείται επικοινωνία χωρίς κρυπτογράφηση (encryption) γιατί τότε θα υπήρχε έκθεση δεδομένων που θα έπρεπε να παραμείνουν κρυφά (π.χ. προσωπικοί κωδικοί).

- Συνεχείς τοπολογικές αλλαγές
  Στην αρχιτεκτονική των μικρουπηρεσιών σε ένα περιβάλλον νέφους (cloud) , η εφαρμογή τρέχει κατανεμημένα σε ένα cluster μηχανημάτων, και κάθε υπηρεσία έχει πολλά δικά της στιγμιότυπα (instances). Σε μια τέτοια δομή, συνεχώς στιγμιότυπα (instances) από υπηρεσίες αποτυγχάνουν και νέα στιγμιότυπα «σηκώνονται» (γεννιούνται), πιθανόν και σε διαφορετικά μηχανήματα.

- Πολλοί διαχειριστές (administrators)
  Βασικό πλεονέκτημα της αρχιτεκτονικής που βασίζεται σε πολλές ανεξάρτητες μικρουπηρεσίες είναι ότι κάθε ομάδα από προγραμματιστές ανάπτυξης εφαρμογών (developers) είναι υπεύθυνη αποκλειστικά για την μικρουπηρεσία που της έχει ανατεθεί. Αυτό συνεπάγεται πολλούς προνομιούχους (privileged) χρήστες με διαφορετικά δικαιώματα ο καθένας.

- Κόστος μετάδοσης της επικοινωνίας
  Η επικοινωνία έχει ένα κόστος γιατί τα δεδομένα για να μεταφερθούν μέσω του δικτύου χρειάζεται πρώτα να σειριοποιηθούν (serialization) και αφού φτάσουν στον προορισμό τους, για να μπορέσουν να διαβαστούν χρειάζεται να αποσειριοποιηθούν (deserialization).

- Μη ομογενές δίκτυο
  Το υλικό μηχάνημα (hardware) που τρέχει η κάθε υπηρεσία δεν είναι απαραίτητα ίδιο. Επίσης τρέχουν διαφορετικά πρωτόκολλα επικοινωνίας ανάμεσα στις υπηρεσίες. Αυτή η ποικιλομορφία θα πρέπει να έχει ληφθεί υπόψη εκ των προτέρων, ειδάλλως το σύστημα θα οδηγείται σε πολλαπλές αποτυχίες.

- Απόδοση πόρων (Resources provisioning)
  Ένα μεγάλο πλεονέκτημα της αρχιτεκτονικής των μικρουπηρεσιών είναι ότι κάθε υπηρεσία μπορεί να κλιμακωθεί μόνη της ατομικά, χωρίς να χρειάζεται να κλιμακωθεί όλο το σύστημα. Αυτό σημαίνει ότι ανά πάσα στιγμή μπορεί να χρειαστεί να αποδοθούν περισσότεροι πόροι (μνήμη και υπολογιστική ισχύ) σε μία υπηρεσία. Άρα θα πρέπει να υπάρχει ένας μηχανισμός που να κάνει αποτελεσματικά την προετοιμασία (provisioning) των πόρων (resources) για την υπηρεσία που πρέπει να κλιμακωθεί.

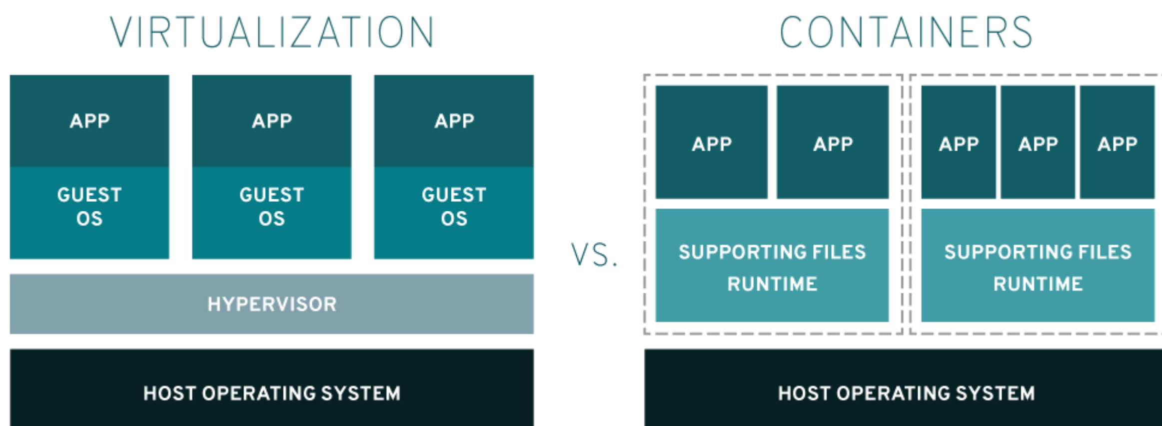
- Παρακολούθηση και εύρεση σφαλμάτων (Monitoring and Debugging)
  Η παρακολούθηση της εφαρμογής είναι πολύ διαφορετική σε σχέση με την μονολιθική αρχιτεκτονική - λόγω της συνεχούς μεταβαλλόμενης κατάστασης των στιγμιότυπων (instances) που τρέχουν τις υπηρεσίες - παραμένοντας όμως ιδιαίτερα σημαντική. Εξίσου σημαντική αλλά και δύσκολη είναι και η εύρεση σφαλμάτων (debugging), γιατί θα πρέπει να μπορούν να συσχετιστούν εγγραφές (logs) από διαφορετικές υπηρεσίες ώστε να διορθωθούν πιθανά προβλήματα.

2.2 Σύγχρονες τεχνολογίες με εφαρμογή στην αρχιτεκτονική των μικρουπηρεσιών

2.2.1 Containers

Τα containers είναι μονάδες κώδικα στις οποίες πακετάρονται το κομμάτι κώδικά της εφαρμογής μαζί με όλες τις εξαρτήσεις του, έτσι ώστε να μπορεί η εφαρμογή να τρέχει με τον ίδιο τρόπο οπουδήποτε και αν προσπαθήσουμε να την τρέξουμε (windows, linux, private datacenter, cloud, personal laptop) [7] [8].
Συχνά δημιουργείται σύγχυση μεταξύ των όρων container και container image. Για να λύσουμε αυτή την σύγχυση μπορούμε να θεωρήσουμε το container image σαν τον πηγαίο κώδικα (source code) και το container σαν την τρέχουσα διεργασία (process). Πιο τυπικά το container image είναι ένα σύνολο αρχείων το οποίο περιέχει τον κώδικα της εφαρμογής μαζί με όλες τις εξαρτήσεις (runtime, system tools, system libraries and settings) [7] της και μετατρέπεται σε container καθόλη την διάρκεια που τρέχει (runtime). Μπορούμε δηλαδή να πούμε ότι το container αποτελεί ένα στιγμιότυπο (instantiation) της container image [9] του. Από τα παραπάνω φαίνεται ότι τα containers προσφέρουν ένα είδος εικονικοποίησης (virtualization) όπως κάνουν και τα VMs. Παρακάτω βλέπουμε κάποια πλεονεκτήματα των containers.

- Τα containers μοιράζονται τον πυρήνα του λειτουργικού συστήματος (OS kernel) του μηχανήματος που τρέχουν ενώ οι εικονικές μηχανές (VMs) εικονικοποιούν το υλικό (hardware). Αυτό τα κάνει πιο "ελαφριά", δηλαδή πιάνουν λιγότερο χώρο, σε σχέση με τα VMs τα οποία χρειάζονται ένα ολόκληρο αντίγραφο του λειτουργικού συστήματος (OS) [10].



Σχήμα 2.2: Εικονικοποίηση των containers και των VMs [12]

Το μέγεθος των container images κυμαίνεται συνήθως σε μερικά megabytes ενώ το μέγεθος των εικονικών μηχανών (VMs) κυμαίνεται σε δεκάδες gigabytes [11].

- Το σχετικά μικρό μέγεθος των containers τους επιτρέπει:
    - Να δημιουργούνται γρήγορα (quickly spin up) και άρα να υποστηρίζουν καλύτερα τις cloud-native εφαρμογές που κλιμακώνονται οριζόντια (horizontal scaling), δηλαδή αυξάνουν και μικραίνουν το πλήθος των στιγμιότυπων (instances) τους [10].
    - Σε ένα φυσικό μηχάνημα μπορούν να τρέχουν παραπάνω containers από εικονικές μηχανές (VMs) [11].

- Τα container images κουβαλούν μαζί τους όλες τις εξαρτήσεις που χρειάζεται για να τρέξει ο κώδικας της εφαρμογής. Αυτό τα καθιστά φορητά και ανεξάρτητα της εκάστοτε πλατφόρμας. Μάλιστα τα container images μπορούν να διατίθενται εύκολα σε μία ομάδα χρηστών μέσω διακομιστών μητρώου (registry servers).

- Με την εικονικοποίηση της ΚΜΕ, της μνήμης (CPU, memory) κλπ. σε επίπεδο λειτουργικού συστήματος, παρέχουν απομόνωση μεταξύ των εφαρμογών [8].

- Βοηθούν στον διαχωρισμό των ευθυνών, με τις ομάδες των προγραμματιστών ανάπτυξης εφαρμογής (developers) να αφοσιώνονται αποκλειστικά στην ανάπτυξη κώδικα της εφαρμογής ενώ οι ομάδες από DevOps μηχανικούς να είναι υπεύθυνες για την εκτέλεση (deployment) και την αυτοματοποίηση της εκτέλεσης (deployment) της εφαρμογής στην παραγωγή.

- Ταιριάζουν με την αρχιτεκτονική των μικρουπηρεσιών αφού επιτρέπουν την κλιμάκωση σε ειδικό επίπεδο (granular scalability) [10].

2.2.2 Container Orchestration

Οι εφαρμογές που ακολουθούν την αρχιτεκτονική των μικρουπηρεσιών συνήθως αποτελούνται από ένα cluster εκατοντάδων υπηρεσιών που τρέχουν σε containers. Ιδανικά, αυτό το cluster από containers θα πρέπει να είναι ανθεκτικό σε λάθη (fault tolerant), συνεχώς διαθέσιμο (available) στους πελάτες, κλιμακώσιμο (scalable), αξιόπιστο και ενδεχομένως γεωγραφικά κατανεμημένο. Όσο πιο μεγάλη είναι η εφαρμογή τόσο πιο πολύπλοκη γίνεται η διαχείριση του cluster των containers.
Οι container orchestrators μπορούν να οριστούν, σαν ένα λογισμικό για την διαχείριση μεγάλου πλήθους containers. Είναι υπεύθυνοι για την αρχική εκτέλεση (deployment) των containers και για την απλοποίηση της διαχείρισης τους όσων αφορά δικτυακά ζητήματα αλλά και ζητήματα διαθεσιμότητας (availability), κλιμακωσιμότητας (scalability) και ανοχής σε λάθη (fault tolerance). Κάποιες από τις βασικές τους δυνατότητες είναι:

- Διαχείριση της κατάστασης του cluster (cluster state) και δρομολόγηση νέου φορτίου (schedule new workload).

    - Οι orchestrators τρέχουν αλγόριθμους που κάνουν προσπάθεια λαμβάνοντας κάποια κριτήρια υπόψη τους να αναθέσουν τα νέα containers όσο το δυνατό καλύτερα στους ήδη υπάρχοντες φυσικούς κόμβους του συμπλέγματος (current nodes of cluster).

    - Είναι υπεύθυνοι για την αξιόπιστη ανακατανομή των πόρων ανάλογα με τα containers που δημιουργούνται ή καταστρέφονται.

    - Ενημερώνουν τα εξαρτώμενα συστήματα για τις αλλαγές που συμβαίνουν στο σύμπλεγμα (cluster) [13].

- Υψηλή διαθεσιμότητα των υπηρεσιών στους πελάτες και ανοχή στα σφάλματα (availability, fault tolerance). Αυτό επιτυγχάνεται με την μέθοδο του πλεονασμού (redundancy).

- Ασφάλεια στο σύμπλεγμα (cluster) με τα containers. Χρησιμοποιούν μεθόδους αυθεντικοποίησης (authentication) και ελέγχου δικαιωμάτων (authorization).

- Απλοποίηση της διαχείρισης της δικτυακής κίνησης στο (cluster).

- Δυνατότητα για ανακάλυψη των υπηρεσιών μεταξύ τους (service discovery). Στην αρχιτεκτονική των μικρουπηρεσιών επειδή δεν τρέχουν όλες οι υπηρεσίες στο ίδιο πρόγραμμα και επειδή η φύση των υπηρεσιών είναι δυναμική, είναι απαραίτητος ένας μηχανισμός μέσω του οποίου οι υπηρεσίες θα μαθαίνουν πως μπορούν να επικοινωνήσουν με την υπηρεσία που αναζητούν. Οι δύο πιο διαδεδομένες αρχιτεκτονικές υλοποίησης αυτού του μηχανισμού εύρεσης υπηρεσιών (service discovery) είναι στην πλευρά του πελάτη (client-based) και στην πλευρά του εξυπηρετητή (server-based).

- Καθιστά δυνατή την συνεχή εκτέλεση υπηρεσιών. Η διαδικασία εκτέλεσης (deployment) του νέου κώδικα είναι τυποποιημένη και μπορεί να επαναλαμβάνεται χωρίς να απαιτεί την διακοπή της εργασίας των διάφορων ομάδων ανάπτυξης της εφαρμογής.

- Παρέχει την δυνατότητα παρακολούθησης των υπηρεσιών.

Μερικά από τα πιο διάσημα εργαλεία στον τομέα του container orchestration είναι τα:

- Kubernetes
- Apache Mesos
- AWS Elastic Container Service
- Nomad

Από τα παραπάνω αυτό που χρησιμοποιήθηκε στην συγκεκριμένη εργασία και ίσως το πιο δημοφιλές από όλα είναι το Kubernetes.

Kubernetes

Το Kubernetes είναι ένα ανοιχτού κώδικα (open source) σύστημα για την αυτοματοποίηση της εκτέλεσης, της κλιμάκωσης και της διαχείρισης των εφαρμογών που τρέχουν σε containers (containerized applications) [14]. Η λέξη Kubernetes προκύπτει από την ελληνική λέξη “κυβερνήτης”. Μπορούμε να σκεφτόμαστε το Kubernetes σαν ένα καπετάνιο σε ένα πλοίο γεμάτο από containers. Οι ρίζες του Kubernetes βρίσκονται στο container orchestrator σύστημα της Google ονόματι Borg και τώρα πλέον αποτελεί project του Cloud Native Computing Foundation (CNCF) [15]. Μερικά από τα πιο βασικά χαρακτηριστικά που προσφέρει το Kubernetes είναι τα εξής [16] :

- Έξυπνη ανάθεση των containers

  - Δρομολογεί τα containers με βάση τους πόρους (resources) που χρειάζονται και τους περιορισμούς που έχουν. Πετυχαίνει να αυξήσει την χρησιμοποίηση (utilization) του υλικού χωρίς να μικρύνει την διαθεσιμότητα.

- Διαχείριση μη λειτουργικών containers

  - Αντικαθιστά (ή επανεκκινεί) τα containers στους κόμβους (nodes) που αποτυγχάνουν.

  - Σκοτώνει (ή επανεκκινεί) τα containers που δεν απαντούν στους ελέγχους της υγείας τους (health checks).

  - Δεν επιτρέπει η δικτυακή κίνηση να φτάσει σε containers που αποτυγχάνουν.

- Οριζόντια κλιμάκωση (Horizontal scaling)
    - Μπορεί και πολλαπλασιάζει τα στιγμιότυπα της εφαρμογής με βάση μετρήσεις της ΚΜΕ, της μνήμης αλλά και με άλλες κατά παραγγελία μετρήσεις (custom metrics).
- Εύρεση υπηρεσιών και ισοστάθμιση φορτίου (Service Discovery και Load Balancing)
    - Αναθέτει IPs στα Pods
    - Αναθέτει ένα μοναδικό όνομα σε ένα σύνολο από containers πετυχαίνοντας τον διαμοιρασμό της δικτυακής κίνησης μεταξύ αυτών (load balancing).
- Αυτόματη αναβάθμιση και αναίρεση αναβάθμισης (Automated rollouts and rollbacks)
    - Μπορεί και κάνει με διαφανή τρόπο αναβαθμίσεις (updates) και αναιρέσεις αναβαθμίσεων (rollbacks) στην εφαρμογή χωρίς εκείνη να επηρεάζεται. Με αυτόν τον τρόπο επιτυγχάνεται μηδενικός νεκρός χρόνος (zero downtime).
- Διαχείριση μυστικών (Secret Configuration and management)
    - Διαχειρίζεται με ειδικό τρόπο την ευαίσθητη πληροφορία (πχ κωδικοί) ώστε να παραμένει κρυφή.
- Ενορχήστρωση αποθηκευτικού χώρου (Storage Orchestration)
    - Μπορεί και αντιστοιχίζει αυτόματα εικονικό αποθηκευτικό χώρο (software defined storage) στα containers.

Αρχιτεκτονική του Kubernetes

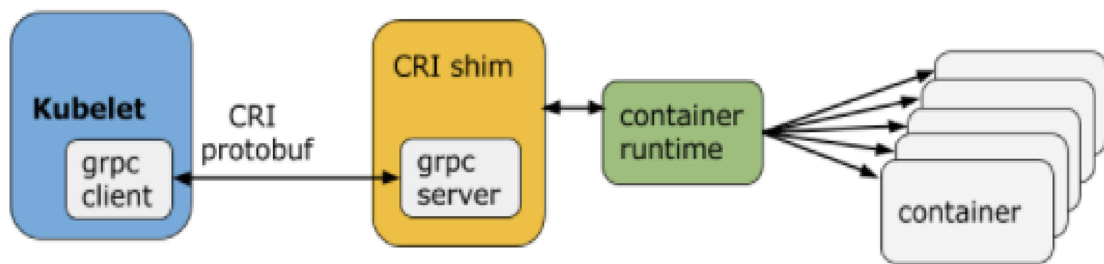
Η αρχιτεκτονική του Kubernetes δομείται από ένα ή περισσότερους μάστερ κόμβους (master nodes) και από έναν ή περισσότερους κόμβους εργάτες (worker nodes). Ο μάστερ (master) αποτελεί τον εγκέφαλο της διάταξης και σε αυτόν τρέχουν τα συστατικά (components) του control plane. Αυτά είναι [16] :

- API Server
    - Δέχεται restful calls και αφού ελέγξει την εγκυρότητα τους (authentication, authorization) τα επεξεργάζεται.
    - Κατά την επεξεργασία των κλίσεων διαβάζει την τρέχουσα κατάσταση του cluster από το etcd εκτελεί το αίτημα και μετά ενημερώνει το etcd για την καινούργια κατάσταση.
    - Μπορεί και μιλάει κατευθείαν στο etcd για αυτό και πολλές χρησιμοποιείται από άλλα συστατικά (components) ως μεσάζοντας στην επικοινωνία με το etcd.
    - Υποστηρίζει οριζόντια κλιμάκωση με βάση τις μετρικές της χρησιμοποίησης της ΚΜΕ και της κατανάλωσης της μνήμης (CPU, memory) και μπορεί να επεκταθεί με extension APIs ώστε υποστηρίζει οριζόντια κλιμάκωση (horizontal autoscaling) και με κατά παραγγελία μετρικές (custom metrics).

- Scheduler
    - Είναι υπεύθυνος για την ανάθεση των νέων Kubernetes αντικειμένων στους κόμβους του cluster. Για να το πετύχει αυτό ακολουθεί τα παρακάτω βήματα:
        - Παίρνει από το etcd, μέσω του API Server, την πληροφορία για την τρέχουσα χρήση των πόρων (resources) σε κάθε κόμβο εργάτη (worker node).
        - Παίρνει από τον API Server τις απαιτήσεις των νέων Kubernetes αντικειμένων.
        - Αφού λάβει υπόψη του όλες τις παραπάνω πληροφορίες (πχ data locality, affinity, anti-affinity, taints, tolerations, cluster topology) αποφασίζει σε ποιο κόμβο θα αναθέσει το νέο αντικείμενο και επικοινωνεί το αποτέλεσμα του στον API Server.
- Controller Managers
    - Τρέχουν controllers για να ρυθμίσουν την κατάσταση του Kubernetes cluster. Οι controllers είναι λούπες που τρέχουν συνεχώς, συγκρίνουν την επιθυμητή κατάσταση του cluster με την τρέχουσα κατάσταση του και αν διαφέρουν προσπαθούν να τις κάνουν να ταιριάζουν.
- etcd
    - Αποτελεί τον αποθηκευτικό σύστημα του cluster
    - Αποθηκεύει την κατάσταση του cluster σε ζευγάρια κλειδί-τιμή (key-value pairs)
    - Το etcd CLI προσφέρει δυνατότητες backup, snapshot και restore
    - Βασίζεται στον αλγόριθμο συμφωνίας Raft (Raft Consensus Algorithm), ο οποίος επιτρέπει σε ένα σύνολο από κόμβους να συνεχίσει να λειτουργεί ακόμα και αν κάποιοι από αυτούς αποτύχουν

Οι κόμβοι εργάτες (worker nodes) εκπροσωπούν το data plane. Σε αυτούς ανατίθενται και τρέχουν τα pods (λογική συλλογή από containers , η μικρότερη μονάδα που διαχειρίζονται οι Kubernetes). Τα βασικά του συστατικά (components) είναι:

- Container Runtime
    - Επειδή το Kubernetes δεν μπορεί να επικοινωνήσει κατευθείαν με τα containers χρησιμοποιεί το container runtime για να τα διαχειρίζεται.
- Kubelet
    - Το kubelet είναι ένας δαίμονας (daemon) που τρέχει σε κάθε εργάτη (worker) και ο βασικός του ρόλος είναι η επικοινωνία με τον μάστερ (master). Δέχεται περιγραφές από pods (pods definitions) και επικοινωνεί με το Container Runtime ώστε να ξεκινήσουν να τρέχουν τα containers που υπάρχουν στις αντίστοιχες περιγραφές. Για να μπορέσει να επικοινωνήσει με το Container Runtime χρησιμοποιείται το Container Runtime Interface Shim, το οποίο αποτελεί ένα επίπεδο αφαίρεσης μεταξύ του kubelet και του Container Runtime.

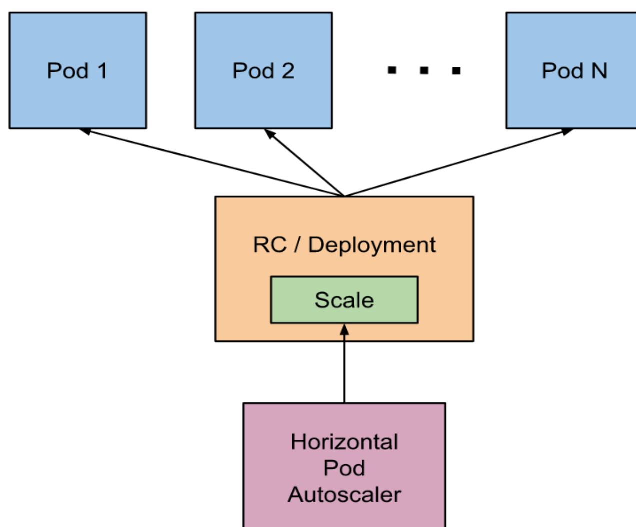


Σχήμα 2.3: Επικοινωνία του kubelet με το container runtime [16]

- kube-proxy
    - Είναι υπεύθυνο για την δυναμική αναβάθμιση και διατήρηση των δικτυακών κανόνων σε ένα κόμβο
    - Υλοποιεί τους κανόνες προώθησης που θέτουν οι χρήστες μέσω Service API αντικειμένων.

Αυτόματη κλιμάκωση (Autoscaling)

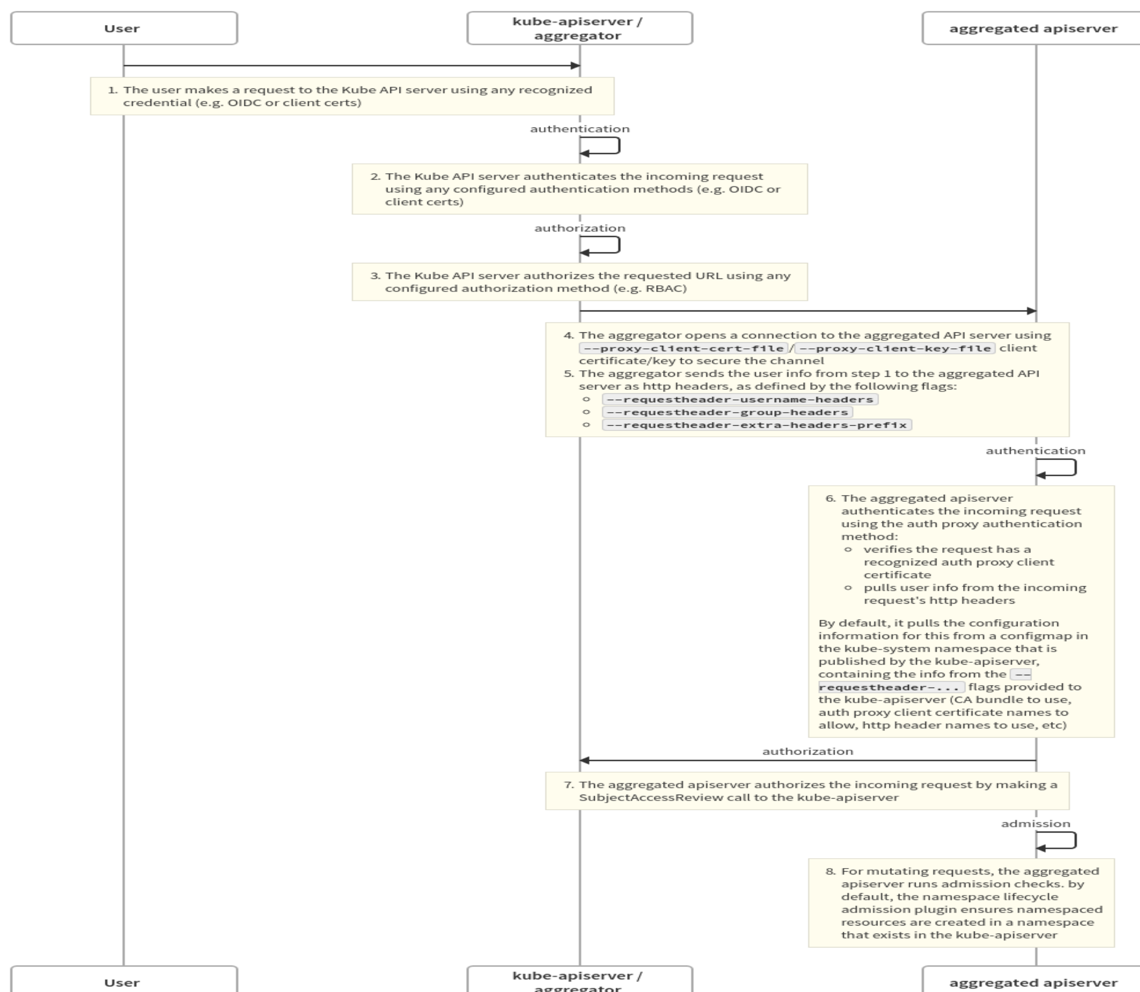
Ένα από τα βασικά και σημαντικά χαρακτηριστικά (features) του Kubernetes είναι η αυτόματη κλιμάκωση. Το Kubernetes μπορεί να παραμετροποιηθεί έτσι ώστε να συγκρίνει τις τιμές μετρικών με ένα κατώφλι και αυτόματα να παίρνει την απόφαση για κλιμάκωση της εφαρμογής. Η κλιμάκωση της εφαρμογής μπορεί να γίνει με δύο διαφορετικές τεχνικές: την κάθετη κλιμάκωση (vertical scaling) και την οριζόντια κλιμάκωση (horizontal scaling). Η κάθετη κλιμάκωση γίνεται με την αύξηση/μείωση των πόρων (π.χ ΚΜΕ, μνήμη) που χρησιμοποιούν τα pods της εφαρμογής. Η οριζόντια κλιμάκωση γίνεται με τον πολλαπλασιασμό ή την μείωση των pods.



Σχήμα 2.4: HorizontalPodAutoscaling [17]

Κάθε κλιμάκωση γίνεται με το αντίστοιχο Kubernetes αντικείμενο (HorizontalPodAutoscaler, VerticalPodAutoscaler). Παρακάτω θα εστιάσουμε στον HorizontalPodAutoscaler ο οποίος χρησιμοποιείται και στην εργασία.

Ο HorizontalPodAutoscaler ελέγχει την κλιμάκωση μίας εκτέλεσης (deployment) και του ReplicaSet του. Η εκτέλεση (deployment) που ο autoscaler καλείται να κλιμακώσει ορίζεται στο πεδίο ScaleTargetRef. Για να πάρει την απόφαση για κλιμάκωση ο autoscaler συγκρίνει τις τιμές μίας μετρικής με την τιμή ενός κατωφλιού που του έχουμε ορίσει. Αν η τιμή της μετρικής ξεπεράσει την τιμή του κατωφλιού τότε δίνεται η εντολή για αυτόματη κλιμάκωση. Οι μετρικές μπορεί να απευθύνονται είτε στην χρησιμοποίηση της ΚΜΕ (CPU) και την κατανάλωση της μνήμη (memory) είτε σε άλλες κατά παραγγελία μετρικές (custom metrics).
Οι μετρικές για την χρησιμοποίηση της ΚΜΕ (CPU) και την κατανάλωση της μνήμης (memory) γίνονται διαθέσιμες μέσω ενός extension API Server, τον Metrics server, ο οποίος δίνει τις μετρήσεις στο metrics.k8s.io API. Οι κατά παραγγελία μετρήσεις (custom metrics) γίνονται διαθέσιμες μέσω άλλων extension apiservers οι οποίοι υλοποιούν το custom.metrics.k8s.io API [17]. Για να μπορέσουν οι extension apiservers να λειτουργήσουν κανονικά πρέπει να διαμορφωθεί το στρώμα συνάθροισης (aggregation layer). Ο ρόλος του aggregation layer είναι η καλή επικοινωνία μεταξύ του Kubernetes apiserver και του extension apiserver και συνήθως χρησιμοποιείται για να παρέχονται οι τιμές διάφορων μετρικών στον HorizontalPodAutoscaler.
Ο HorizontalPodAutoscaler στέλνει αίτημα στον Kubernetes apiserver για να λάβει τις τιμές των μετρικών. Εκείνος αφού ελέγξει την αυθεντικότητα του αιτήματος (authenticate) και ελέγξει και τα δικαιώματα του χρήστη (authorize) στέλνει (μέσω του aggregation layer) το αίτημα στον extension apiserver. Ο extension apiserver με την σειρά του ελέγχει την αυθεντικότητα του Kubernetes apiserver ελέγχει τα δικαιώματα του χρήστη του αιτήματος και μετά το εκτελεί. Αφού λάβει τα αποτελέσματα, αυτά επιστρέφονται μέσω του αντίστροφου μονοπατιού στον HPA [18]. Η επικοινωνία μεταξύ του χρήστη (πιθανός χρήστης είναι ο HorizontalPodAutoscaler) και του extension Apisever μαζί με όλους τους βασικούς μηχανισμούς ασφάλειας (authentication και authorization) απεικονίζονται στο παρακάτω σχήμα.



Σχήμα 2.5: Επικοινωνία μεταξύ χρήστη και extension Apiserver [18]

2.3 Service Mesh

2.3.1 Ιστορική αναδρομή

Στα μέσα της δεκαετίας του 2010-2020 οι εταιρείες διεθνούς κλίμακας όπως η Twitter, η Netflix και η Google προκειμένω να μπορέσουν να ανταπεξέλθουν στην ολοένα και αυξανόμενη ζήτηση των υπηρεσιών τους, αποφασίζουν να εγκαταλείψουν το μονολιθικό μοντέλο υπηρεσιών και να υιοθετήσουν την αρχιτεκτονική των μικρουπηρεσιών στις εφαρμογές τους. Ξεκίνησαν λοιπόν να αποδομούν την μία κεντρική μονολιθική εφαρμογή σε πολλές μικρότερες υπηρεσίες καθεμία εκ των οποίων μπορεί να είναι κλιμακώσιμη και, να αναπτύσσεται και να διαχειρίζεται ανεξάρτητα από τις υπόλοιπες, χωρίς όμως να σταματήσει να επικοινωνεί μαζί τους.
Αυτή η αλλαγή οδήγησε σε μεγάλη αύξηση της επικοινωνίας (μέσω δικτύου) μεταξύ των μικρουπηρεσιών αλλά και σε ένα ευρύτερο σύνολο προκλήσεων που έπρεπε να αντιμετωπιστούν (π.χ. μη αξιόπιστο δίκτυο, μη ασφαλές δίκτυο, διαφορετικές γλώσσες προγραμματισμού ανά υπηρεσία κλπ.). Πολλές από τις προκλήσεις λυνόντουσαν μέσω των Kubernetes (π.χ. η αποτελεσματική προετοιμασία πόρων (provisioning of resources) και η αυτόματη κλιμάκωση (autoscaling) μίας υπηρεσίας) αλλά για πολλές που σχετίζονταν με την επικοινωνία μεταξύ των μικρουπηρεσιών μέσω του δικτύου, δεν υπήρχαν έτοιμες λύσεις.
Έτσι, για την διαχείριση αυτής της επικοινωνίας, οι μεγάλες εταιρείες ξεκίνησαν να δημιουργούν ειδικές βιβλιοθήκες. Παρόλο που οι βιβλιοθήκες αυτές κατάφεραν σε ένα βαθμό να διαχειριστούν την επικοινωνία μεταξύ των υπηρεσιών, είχαν κάποια σοβαρά μειονεκτήματα. Η κάθε μικρουπηρεσία ήταν γραμμένη σε διαφορετική γλώσσα προγραμματισμού και αυτό δημιουργούσε πρόβλημα στην μεταξύ τους επικοινωνία. Επίσης οι βιβλιοθήκες ήταν στενά συνδεδεμένες με την μικρουπηρεσία που τις χρησιμοποιούσε με αποτέλεσμα μία αλλαγή σε μία βιβλιοθήκη να απαιτεί επανεκτέλεση (redeployment) όλων των μικρουπηρεσιών που την χρησιμοποιούσαν.
Στην προσπάθεια να αντιμετωπιστούν αυτά και άλλα προβλήματα η βιομηχανία οδηγήθηκε στην έννοια του Service Mesh. Η ιδέα στην οποία στηρίχθηκε ήταν η ενσωμάτωση των λειτουργιών (που μέχρι τότε παρεχόντουσαν από τις βιβλιοθήκες) μέσα σε proxies, με τέτοιο τρόπο ώστε να υπάρχει ανεξαρτησία από την γλώσσα προγραμματισμού που χρησιμοποιήθηκε για την εκάστοτε υπηρεσία. Έτσι, η οποιαδήποτε αναβάθμιση των proxies θα ήταν διαφανής για την εφαρμογή επιτρέποντας της να συνεχίσει να λειτουργεί κανονικά χωρίς νεκρό χρόνο (downtime).
Το service mesh φαινόταν ότι θα πετύχαινε να αντιμετωπίσει τις δικτυακές προκλήσεις που προκύπταν από την αρχιτεκτονική μικρουπηρεσιών και να λύσει τα προβλήματα που δημιουργούνταν από την χρήση των βιβλιοθηκών. Το μόνο πρόβλημα που έπρεπε να αντιμετωπιστεί με την παραπάνω ιδέα, ήταν η διαχείριση και η εκτέλεση (deployment) πολλών proxies. Η λύση δόθηκε από την ανάπτυξη των containers και των containers orchestrators. Ο proxy πακετάρετε μέσα σε ένα container και ο orchestrator τον εκτελεί (deploy) ομοιόμορφα για όλη την εφαρμογή. Με λίγα λόγια λοιπόν η στροφή στην αρχιτεκτονική των μικρουπηρεσιών δημιούργησε την ανάγκη για το service mesh και η ανάπτυξη των containers και των containers orchestrators έκανε την υλοποίηση του εφικτή [19].

2.3.2 Τι είναι το Service Mesh

Ένας ορισμός που έχει δοθεί στο Service Mesh από τον William Morgan [58] είναι ο παρακάτω:

Το Service Mesh είναι ένα τμήμα της υποδομής που δεσμεύεται για να τρέξει μία εφαρμογή (dedicated infrastructure layer), υπεύθυνο για την διαχείριση της δικτυακής επικοινωνίας μεταξύ των μικρουπηρεσιών. Είναι υπεύθυνο για την αξιόπιστη παράδοση των αιτημάτων που στέλνονται από και προς τις υπηρεσίες που δομούν μία εφαρμογή στο νέφος (cloud). Συνήθως δομείται από ένα σύνολο από proxies που τρέχουν δίπλα στην εφαρμογή χωρίς να επηρεάζουν την λειτουργικότητα της. Η κύρια αξία του Service Mesh είναι ότι προσφέρει παρατηρησιμότητα (observability),

ασφάλεια (security) και αξιοπιστία (reliability) σε επίπεδο πλατφόρμας (platform layer) ενώ αν δεν υπήρχε, τα παραπάνω θα υλοποιούνταν στο επίπεδο της εφαρμογής (application layer) [20]. Ας δούμε αυτούς τους τρεις άξονες λίγο πιο αναλυτικά.

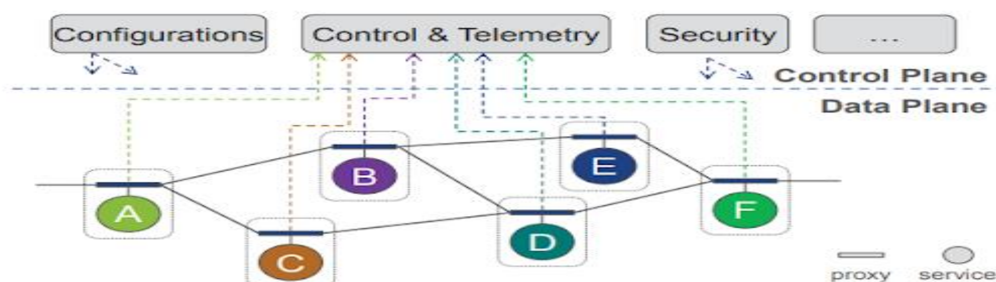
- Παρατηρησιμότητα (Observability)
  Το service mesh μέσω των proxies έχει την δυνατότητα παρατηρώντας μόνο την έξοδο (output) των μικρουπηρεσιών να συλλέγει τις επονομαζόμενες «Χρυσές Μετρήσεις» (Golden Metrics). Μας ενημερώνει δηλαδή για τον χρόνο που απαιτείται για να απαντήσει μία μικρουπηρεσία σε ένα αίτημα (Latency), το πλήθος επιτυχημένων απαντήσεων σε σχέση με το συνολικό πλήθος των αιτημάτων (Success Rate), το τον ρυθμό με τον οποίο δέχεται αιτήματα μία μικρουπηρεσία (Traffic) και τους πόρους που χρησιμοποιεί έντονα η εφαρμογή (Saturation).
  Αυτές οι μετρήσεις μπορούν να αποθηκευτούν (συνήθως στο Prometheus) και μετέπειτα μπορούν να οπτικοποιηθούν, να επεξεργαστούν με σκοπό την εξαγωγή στατιστικών (π.χ. με το Grafana) αλλά και να χρησιμοποιηθούν βοηθητικά στην διαδικασία εύρεσης σφαλμάτων της εφαρμογής (debugging).

- Ασφάλεια (Security)
  Στις «cloud native» εφαρμογές κυριαρχεί το μοντέλο της μηδενικής εμπιστοσύνης (zero-trust model). Η βασική ιδέα του μοντέλου αυτού, είναι ότι τα ασφαλιστικά μέτρα (security measures) εφαρμόζονται στο μικρότερο και πιο ειδικό δυνατό επίπεδο. Για παράδειγμα σε ένα κέντρο δεδομένων (data center) στο οποίο τρέχουν κάποιες εφαρμογές, δεν χρησιμοποιείται ένα τείχος προστασίας (firewall) μπροστά από το κέντρο δεδομένων (data center) αλλά μπροστά από κάθε εφαρμογή ξεχωριστά.
  Το service mesh λοιπόν, είναι εναρμονισμένο με αυτό το μοντέλο αφού χρησιμοποιεί mTLS και έλεγχο δικαιωμάτων (authorization) στην επικοινωνία μεταξύ οποιονδήποτε δύο μικρουπηρεσιών. Το mTLS διασφαλίζει ότι γίνεται επικύρωση της ταυτότητας των δύο υπηρεσιών (services) που επικοινωνούν και ότι η επικοινωνία μεταξύ τους είναι αμφίδρομα κρυπτογραφημένη.

- Αξιοπιστία (Reliability)
  Για να υπάρχει αξιοπιστία στην εφαρμογή θα πρέπει εκείνη να μπορεί δίνει απαντήσεις ακόμα και αν υπάρχουν εσωτερικά «μερικές αποτυχίες» (partial failures). Επίσης στο κομμάτι της αξιοπιστίας είναι η τήρηση της διαθεσιμότητας. Δηλαδή ένα σύστημα που θέλουμε να έχει διαθεσιμότητα 99.9999% είναι αξιόπιστο μόνο αν αποτυγχάνει μία στις 999999. Το Service Mesh μέσα από μηχανισμούς όπως retries, timeouts, load balancing και traffic shifting μπορεί και βελτιώνει την αξιοπιστία της εφαρμογής.

2.3.3 Η αρχιτεκτονική του service mesh

Ας εμβαθύνουμε λίγο στην δομή του Service Mesh. Συνήθως τα service-meshes αποτελούνται από ένα data plane και ένα control plane, όπως στο παραπάνω σχήμα [20].



Σχήμα 2.6: Αρχιτεκτονική Service mesh [20]

Το data plane αποτελείται από ένα σύνολο από proxies που όπως φαίνεται και στο σχήμα τρέχουν «δίπλα» στην εφαρμογή (σε ένα δεύτερο container στο ίδιο Pod της κάθε μικρουπηρεσίας) γνωστοί και ως sidecar proxies. Οι proxies χωρίς να επηρεάζουν την λειτουργικότητα της μικρουπηρεσίας συλλέγουν, επεξεργάζονται και δρομολογούν την δικτυακή κίνηση από και προς την εκάστοτε υπηρεσία. Επίσης χάρις τους proxies το service mesh μπορεί και κρυπτογραφεί την δικτυακή κίνηση (network traffic), και εφαρμόζει αυθεντικοποίηση (authentication) και έλεγχο δικαιωμάτων (authorization). Με απλά λόγια μπορούμε να φανταστούμε τους proxies σαν εργάτες οι οποίοι με την κατάλληλη τεχνογνωσία μπορούν να υλοποιούν κάποιες εργασίες (π.χ. mTLS).
Το control plane λειτουργεί σαν ένας εγκέφαλος ο οποίος δίνει την δυνατότητα στον χρήστη, μέσα από ένα σύνολο από APIs, να μπορεί να διαχειρίζεται τους proxies του data plane , να συλλέγει τις μετρήσεις αυτών και να τους παραμετροποιεί κατάλληλα. Θα λέγαμε δηλαδή, ότι το control plane δίνει στους proxies την τεχνογνωσία για να εκτελούν τις εργασίες που ζητάει κάθε φορά ο χρήστης.

2.3.4 Σύγκριση Διάσημων Service meshes

Στον χώρο των service meshes δραστηριοποιούνται πολλές μεγάλες εταιρείες όπως η Google , η IBM, η HashiCorp, η Microsoft κλπ. Η συμμετοχή όλων αυτών των εταιρειών σε συνδυασμό με το νεαρό της τεχνολογίας και του διαφορετικού προσανατολισμού αυτής έχει δημιουργήσει πολλά διαφορετικά έργα (projects). Εκτός από τα έργα (projects) που αποτελούν ολοκληρωμένα service meshes υπάρχουν πολλά τα οποία εστιάζουν στη επίλυση συγκεκριμένων προβλημάτων ή στην βελτίωση της επίδοσης. Δύο από αυτά είναι τα παρακάτω:

- Aeraki mesh
  Πρόκειται για ένα ανοιχτού-κώδικα έργο (open-source project) του CNCF το οποίο εστιάζει στην διαχείριση οποιουδήποτε πρωτοκόλλου επιπέδου εφτά (layer-7 protocol) και όχι μόνο του HTTP [26].

- Merbridge
  Ένα ανοιχτού-κώδικα έργο (open-source project) του CNCF που στόχος του είναι η επιτάχυνση της προώθησης της δικτυακής κίνησης [27]. Για να το πετύχει αυτό αντικαθιστά τα iptables με τον μηχανισμό eBPF (extended Berkeley Packet Filter) [28]

Τα παραπάνω λειτουργούν συμπληρωματικά στα ολοκληρωμένα service meshes. Τα πιο διάσημα ανοιχτού-κώδικα (open-source) service meshes που ανήκουν στον CNCF είναι τα: Istio, Linkerd, ConsulConnect, Kuma, Traefik, και OpenServiceMesh. Παρακάτω παρουσιάζεται η αρχιτεκτονική καθενός από αυτά τα service meshes και έπειτα ακολουθεί ένας συγκριτικός πίνακας αυτών [29].

Linkerd

Ένα από τα πιο δημοφιλή service meshes είναι το Linkerd. Πρόκειται για ένα ανοιχτού-κώδικα (open-source) service mesh και ένα προχωρημένο έργο (graduated project) του CNCF που αναπτύσσεται από την Buoyant. Το Linkerd ακολουθεί την πιο διαδεδομένη αρχιτεκτονική που αποτελείται από ένα control plane και ένα data plane.
Το data plane αποτελείται από proxies που τρέχουν δίπλα στις υπηρεσίες της εφαρμογής (sidecar proxies). Μέσα από αυτούς τους proxies περνάει όλη η δικτυακή κίνηση μεταξύ των υπηρεσιών. Το Linkerd χρησιμοποιεί ένα proxy για κάθε στιγμιότυπο (instance) της κάθε υπηρεσίας. Επειδή με αυτό τον τρόπο αναγκάζεται να χρησιμοποιεί πάρα πολλούς proxies πρέπει να φροντίσει αυτοί να είναι πολύ γρήγοροι (δηλαδή να μην δημιουργούν καθυστέρηση στην επικοινωνία μεταξύ των υπηρεσιών) και πολύ μικροί σε μέγεθος (δηλαδή να χρειάζονται λίγους πόρους όσον αφορά την κατανάλωση της ΚΜΕ (CPU) και της μνήμης). Για να τα πετύχει αυτά τα δύο και σε συνδυασμό με

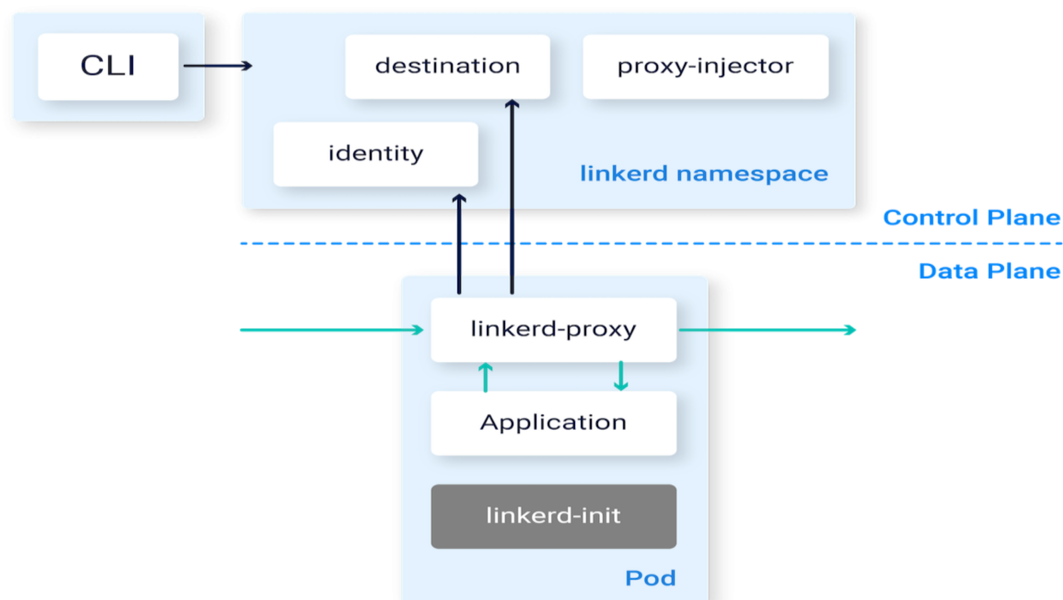
τον στόχο του έργου (project) να δώσει έμφαση στην ταχύτητα και την ασφάλεια, το Linkerd χρησιμοποιεί τους “micro-proxy” Linkerd2 [21].
Το control plane αποτελείται από κάποια συστατικά (components), όπως φαίνεται και στο σχήμα που ακολουθεί, ο ρόλος των οποίων είναι η συντονισμένη και ομαλή λειτουργία του data plane.

Τα συστατικά του control plane όπως φαίνεται και στο σχήμα είναι:

- Destination
  Το Destination χρησιμοποιείται για την πληροφορία εύρεσης υπηρεσιών (που βρίσκεται η υπηρεσία που θέλουμε να επικοινωνήσουμε και τι TLS ταυτότητα να περιμένουμε στην άλλη πλευρά). Επίσης από το Destination μπορεί να γίνει η ανάκτηση πληροφορίας σχετικά με την πληροφορία του service profile μίας υπηρεσίας [22].

- Identity
  Το Identity λειτουργεί σαν αρχή πιστοποίησης (TLS Certificate authority). Λαμβάνει αιτήματα πιστοποίησης (Certificate Signing Requests) και τα υπογράφει. Έτσι ο κάθε proxy έχει ένα πιστοποιητικό (certificate) που χρησιμοποιεί στην mTLS επικοινωνία με τους υπόλοιπους. Η διαδικασία έκδοσης πιστοποιητικών ξεκινάει από τον κάθε proxy κατά την διαδικασία της αρχικοποίησής του [22].

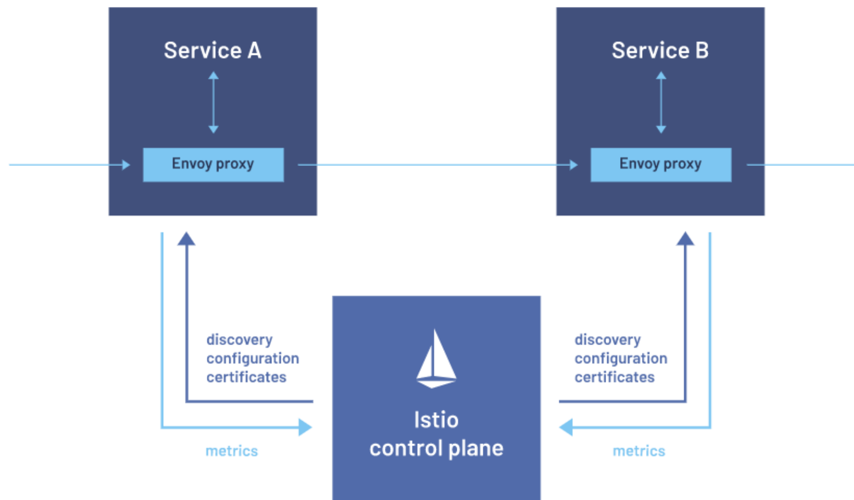
- proxy-injector
  Ο proxy-injector είναι ένας Admission Controller [23] ο οποίος λαμβάνει ένα αίτημα κάθε φορά που δημιουργείται ένα pod. Αν στον ορισμό του pod υπάρχει το annotation “linkerd.io/inject: enabled” τότε ο proxy-injector μορφοποιεί τον ορισμό του pod προσθέτοντας δύο containers, το proxy-init και το linkerd-proxy [22].



Σχήμα 2.7: Αρχιτεκτονική Linkerd [22]

Istio

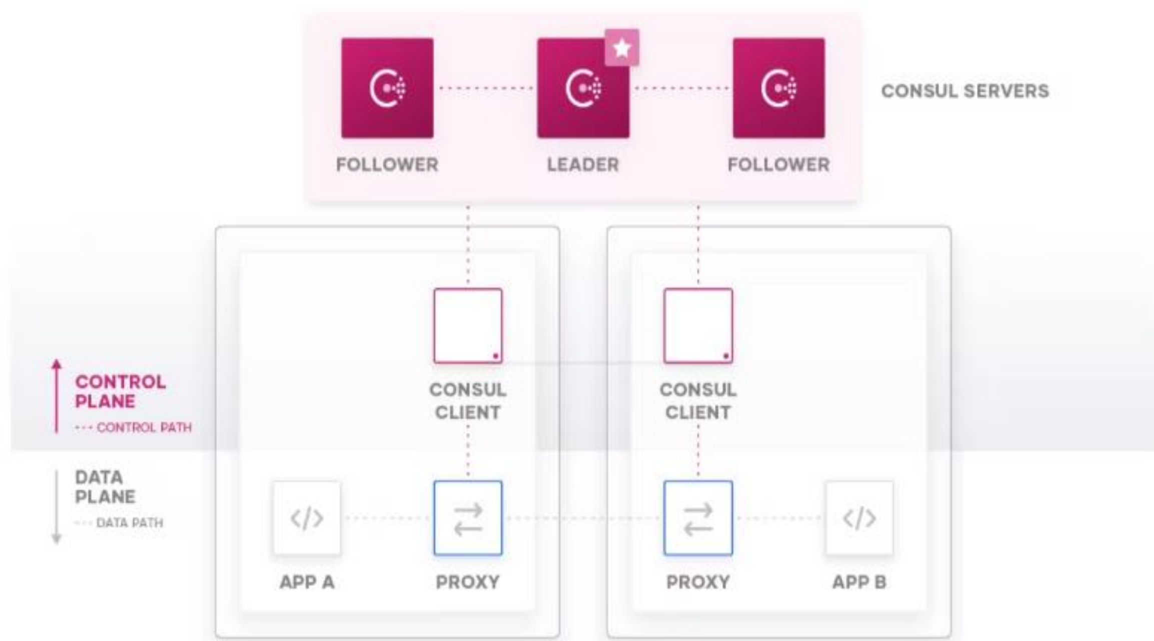
Εξίσου δημοφιλές με το Linkerd είναι και το Istio. Το Istio είναι και αυτό ένα ανοιχτού-κώδικα (open-source) service mesh πρόσφατα (28/09/2022) ενταγμένο στο CNCF, το οποίο ξεκίνησε από ομάδες της Google και της IBM σε συνεργασία με την ομάδα Envoy της Lyft [24]. Η αρχιτεκτονική του είναι πολύ παρόμοια με αυτή του Linkerd. Η μόνη διαφορά είναι ότι το data plane δεν αποτελείται από Linkerd2 proxies αλλά από Envoy proxies. Οι Envoy proxies δεν είναι το ίδιο "ελαφριοί" και γρήγοροι με τους Linkerd2 proxies αλλά είναι πιο γενικού σκοπού και μπορούν να κάνουν πολλά περισσότερα πράγματα (dynamic configuration, HTTP L7 routing, L3/L4 filter [25]). Η παρακάτω εικόνα δείχνει σε υψηλό επίπεδο (high level) την αρχιτεκτονική του Istio.



Σχήμα 2.8: Αρχιτεκτονική Istio [24]

Consul Connect

Το Consul αποτελεί ένα έργο (project) της HashiCorp που ο αρχικός του ρόλος ήταν να παρέχει έναν έξυπνο μηχανισμό εύρεσης υπηρεσιών (Service discovery) σε εφαρμογές που είναι αναπτυγμένες με βάση την αρχιτεκτονική των μικρουπηρεσιών. Αυτός ο πρωταρχικός στόχος απαιτεί να μπορούν να επικοινωνούν οι υπηρεσίες μεταξύ τους σε ένα δυναμικό περιβάλλον χωρίς σταθερές διευθύνσεις. Για να γίνει αυτό, όλες οι υπηρεσίες της εφαρμογής υπάρχουν καταγεγραμμένες σε ένα κεντρικό κατάλογο-καταχωρητή ο οποίος πάντα ενημερώνεται για το ποιες υπηρεσίες λειτουργούν ή αντιμετωπίζουν πρόβλημα. Έτσι όταν μία υπηρεσία θέλει να μιλήσει με μια άλλη μαθαίνει που πρέπει να απευθυνθεί μέσω του κεντρικού καταλόγου-καταχωρητή.
Η HashiCorp προχώρησε το Consul και το εξέλιξε από έναν μηχανισμό εύρεσης υπηρεσιών (Service Discovery μηχανισμός) σε ένα ολοκληρωμένο Service mesh, το Consul Connect. Το Consul Connect πέρα από τα βασικά οφέλη του Service Discovery μπορεί επιπλέον να παρέχει την δυνατότητα διαχείρισης και παρακολούθησης της δικτυακής επικοινωνίας, μπορεί να την κρυπτογραφήσει και να εφαρμόσει αυθεντικοποίηση και έλεγχο δικαιωμάτων στην επικοινωνία μεταξύ των μικρουπηρεσιών. Η παρακάτω εικόνα μας δίνει μία γενική ιδέα της αρχιτεκτονικής του Consul Connect.

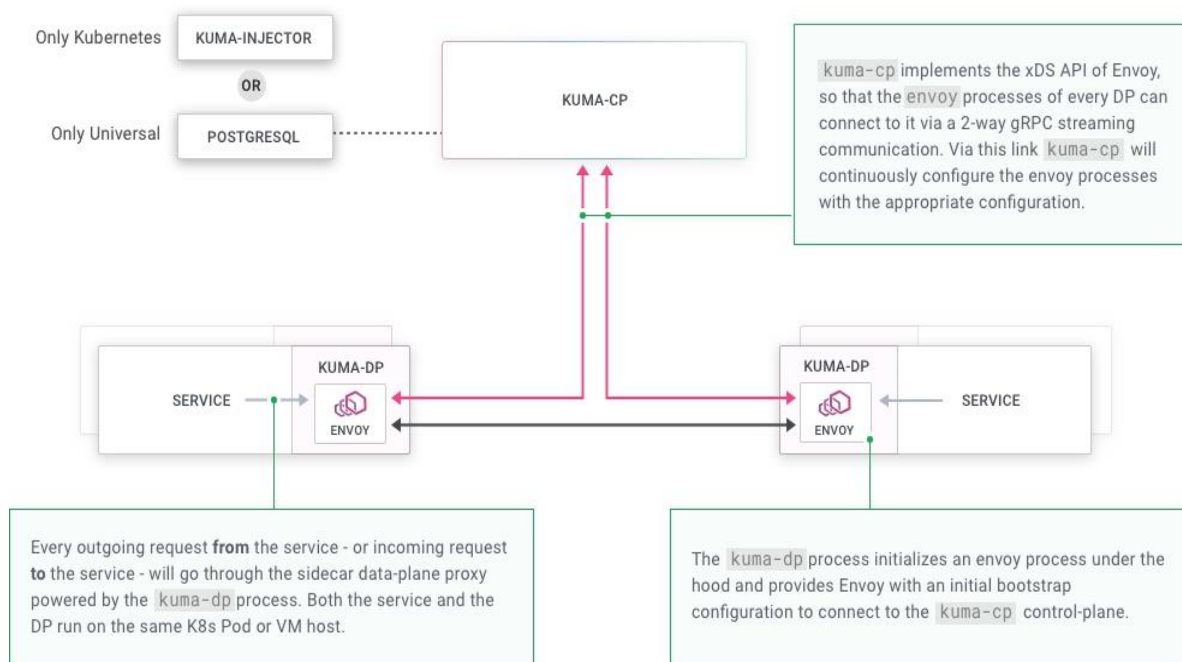


Σχήμα 2.9: Αρχιτεκτονική Consul Connect [51]

Το control plane του Consul Connect αποτελείται από ένα ή περισσότερα κέντρα δεδομένων (datacenters ή ισοδύναμα clusters). Κάθε Cluster (ή datacenter) αποτελείται από δύο ειδών στοιχεία (components), τους πράκτορες-εξυπηρετητές (Server Agents) και τους πράκτορες-πελάτες (Client Agents).
Οι πρώτοι είναι υπεύθυνοι να κρατούν την κατάσταση που βρίσκονται ανά πάσα στιγμή οι υπηρεσίες, δηλαδή ποιες υπηρεσίες είναι λειτουργικές, σε ποιες διευθύνσεις βρίσκονται κλπ. Οι πράκτορες-εξυπηρετητές (Servers Agents) συνήθως είναι περισσότεροι από ένας. Ένας είναι ο αρχηγός (leader) ο οποίος είναι υπεύθυνος για την διαχείριση και απάντηση των αιτημάτων ενώ οι υπόλοιποι ονομάζονται ακόλουθοι (followers) και ο ρόλος τους, όσο υπάρχει αρχηγός, είναι να προωθούν τα αιτήματα που λαμβάνουν από τους πράκτορες-πελάτες (Client Agents) στον αρχηγό (leader). Ο αρχηγός (leader) φροντίζει να αντιγράφει την κατάσταση του σε όλους τους ακόλουθους (followers) έτσι ώστε αν για κάποιο λόγω αντιμετωπίσει πρόβλημα να μπορεί να αντικατασταθεί από έναν από αυτούς.
Οι πράκτορες-πελάτες (Client Agents) από την άλλη είναι υπεύθυνοι να ενημερώνουν τους πράκτορες-εξυπηρετητές (Server Agents) για την κατάσταση των μικρουπηρεσιών [51] (διεύθυνση, υγειά κλπ.). Αξίζει να σημειωθεί ότι για να διατηρείται η ομαλή λειτουργία του Cluster εφαρμόζεται ένας μηχανισμός για τον περιορισμό της εντροπίας (anti-entropy mechanism) [52]. Ο μηχανισμός αυτός συγχρονίζει την κατάσταση του κεντρικού καταλόγου-καταχωρητή με αυτή των πρακτόρων-πελατών (Client Agents) ώστε να υπάρχει συνοχή στο Cluster. Σε περίπτωση που κατά την διαδικασία συγχρονισμού υπάρξει διαφωνία μεταξύ της κατάστασης των πρακτόρων-πελατών (Clients Agents) και του κεντρικού καταλόγου καταχωρητή δίνεται προτεραιότητα στην κατάσταση των πρακτόρων-πελατών (Client Agents).
Το data plane αποτελείται από proxies οι οποίοι τρέχουν δίπλα σε κάθε υπηρεσία και προωθούν την δικτυακή κίνηση που λαμβάνουν από και προς αυτή. Μέσω των proxies δίνεται η δυνατότητα για κρυπτογράφηση (encryption), αυθεντικοποίηση (authentication), παρατηρησιμότητα (observability) και διαχείριση της δικτυακής κίνησης (traffic management). Είναι σημαντικό να αναφερθεί ότι όλοι οι proxies μπορούν παραμετροποιηθούν ταυτόχρονα από το Control plane όπως και στα άλλα Service meshes. Το Consul Connect έχει δικούς του (built in) proxies αλλά μπορεί εναλλακτικά να λειτουργήσει και με Envoy proxies.

Kuma

Το Kuma όπως και τα υπόλοιπα είναι ένα ανοιχτού-κώδικα (open-source) Service meshes έργο του CNCF (Sandbox project) το οποίο αποτελείται και αυτό από ένα control plane (kuma-cp) και ένα data plane. Το data plane αποτελείται από proxies μέσα από τους οποίους διέρχεται όλη η επικοινωνία μεταξύ των υπηρεσιών.
Το control plane λειτουργεί ανεξάρτητα του data plane, δεν αλληλοεπιδρά άμεσα με την δικτυακή κίνηση μεταξύ των υπηρεσιών αλλά μπορεί να δέχεται από τους χρήστες πολιτικές ρύθμισης της δικτυακής κίνησης (policies) και να παραμετροποιεί κατάλληλα τους proxies του data plane. Έτσι μία εκτέλεση (deployment) του Kuma αποτελείται από τουλάχιστον ένα control plane (kuma-cp) και τουλάχιστον ένα kuma-dp για κάθε υπηρεσία που ανήκει στο mesh. Το kuma-dp είναι μία διεργασία που αρχικοποιεί έναν envoy proxy παραμετροποιώντας τον κατάλληλά ώστε να μπορεί να επικοινωνήσει με το control plane [53].



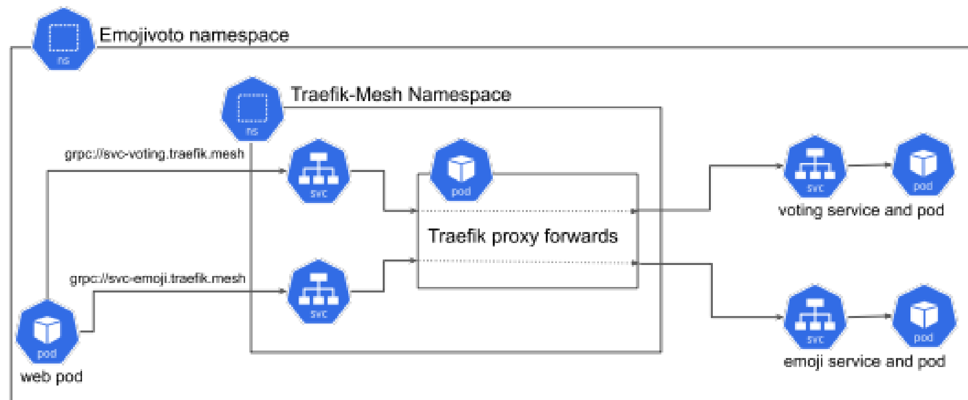
Σχήμα 2.10: Αρχιτεκτονική του Kuma [53]

Το Kuma έχει το πλεονέκτημα ότι μπορεί να τρέξει και σε Kubernetes αλλά και σε εικονικές μηχανές (VMs), φυσικούς κόμβους (bare metal) και υβριδικά περιβάλλοντα. Αν τρέχει σε Kubernetes mode μπορεί και χρησιμοποιεί τον Kubernetes API Server για να αποθηκεύει την κατάσταση του ενώ αν τρέχει σε Universal mode αξιοποιεί την PostgreSQL σαν αποθηκευτικό χώρο για την κατάστασή του. Να σημειωθεί ότι έχει την δυνατότητα να τρέχει σε Kubernetes αλλά να χρησιμοποιεί το Universal mode. Σε αυτή την περίπτωση χρησιμοποιεί και PostgreSQL για να αποθηκεύει την κατάσταση του.

Traefik Mesh

Το Traefik mesh είναι και αυτό ένα ανοιχτού-κώδικα (open source) Service mesh που στεγάζεται στο CNCF. Το Traefik mesh λειτουργεί σε Kubernetes και η αρχιτεκτονική του είναι διαφορετική σε σχέση με τα προηγούμενα meshes που αναφέρονται. Εδώ δεν χρησιμοποιούνται sidecar proxies δίπλα σε κάθε μικρουπηρεσία της εφαρμογής. Αντίθετα το Traefik mesh χρησιμοποιεί proxy

endpoints [54] για κάθε κόμβο του Kubernetes cluster και μια δική του DNS ζώνη. Με αυτόν τον τρόπο το mesh είναι λιγότερο επεμβατικό καθώς δεν αλλάζει Kubernetes αντικείμενα των μικρουπηρεσιών ούτε τα IPtables των Pods [55]. Ένα χαρακτηριστικό παράδειγμα ροής της δικτυακής κίνησης στο Traefik mesh απεικονίζεται στο παρακάτω σχήμα



Σχήμα 2.11: Ροή Πληροφορίας στο Traefik mesh [55]

Στο σχήμα χρησιμοποιείται η ανοιχτού-κώδικα (open-source) εφαρμογή της Buoyant ονόματι emojivoto [56]. Αξίζει να παρατηρηθεί ότι τα αιτήματα από την web μικρουπηρεσία δεν περνάνε από κάποιο sidecar proxy και καταλήγουν σε endpoints που ανήκουν στην DNS ζώνη .cluster.local αλλά ανήκουν στην DNS ζώνη .traefik.mesh (η DNS ζώνη του Traefik mesh) [55]. Από εκεί ο Traefik proxy του κάθε κόμβου τα δρομολογεί κατάλληλα στις άλλες μικρουπηρεσίες.

Open Service Mesh (OSM)

Το OSM είναι ένα ανοιχτού-κώδικα (open-source) service mesh και έργο (sandbox project) του CNCF. Η αρχιτεκτονική του είναι παρόμοιας φιλοσοφίας με τα περισσότερα service mesh, έχει δηλαδή ένα control plane, που το ονομάζει service mesh controller, και ένα data plane που αποτελείται από sidecar envoy proxies που τρέχουν δίπλα στις μικρουπηρεσίες και διαχειρίζονται την δικτυακή κίνηση μεταξύ αυτών. Τα βασικά στοιχεία του control plane είναι:

- Το Proxy Control plane, το οποίο είναι υπεύθυνο για την συνεχή αναβάθμισή της παραμετροποίησης των proxies. Όλοι οι proxies έχουν μία mTLS σύνδεση με το Proxy Control plane.

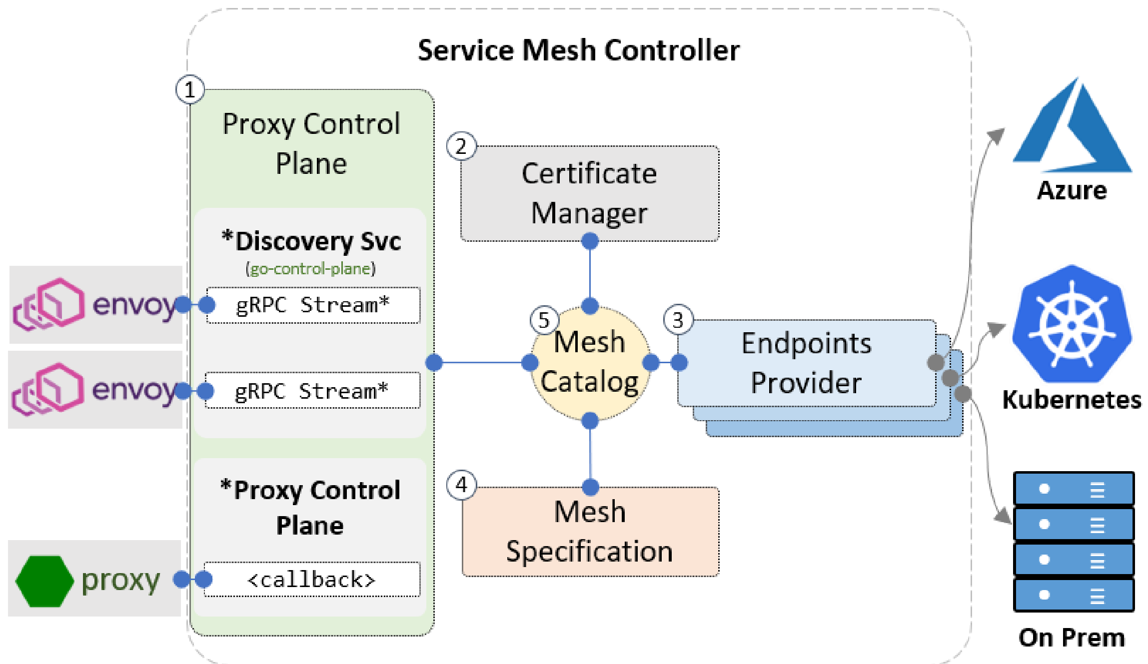
- Τον Certificate Manager, ο οποίος είναι υπεύθυνος για την έκδοση ενός TLS πιστοποιητικού (certificate) για την κάθε μικρουπηρεσία. Το πιστοποιητικό αυτό χρησιμοποιείται για την υλοποίηση (κρυπτογράφηση και αυθεντικοποίηση) της mTLS επικοινωνίας με τις άλλες μικρουπηρεσίες.

- Οι Endpoint Providers, οι οποίοι είναι υπεύθυνοι για την αντιστοίχιση των ονομάτων των μικρουπηρεσιών με τα σύνολα των IP διευθύνσεων που βρίσκονται αυτές.

- Το mesh Specification, το οποίο λειτουργεί να ένα περικάλυμμα (wrapper) των SMI ορισμών (SMI specification components) [57].

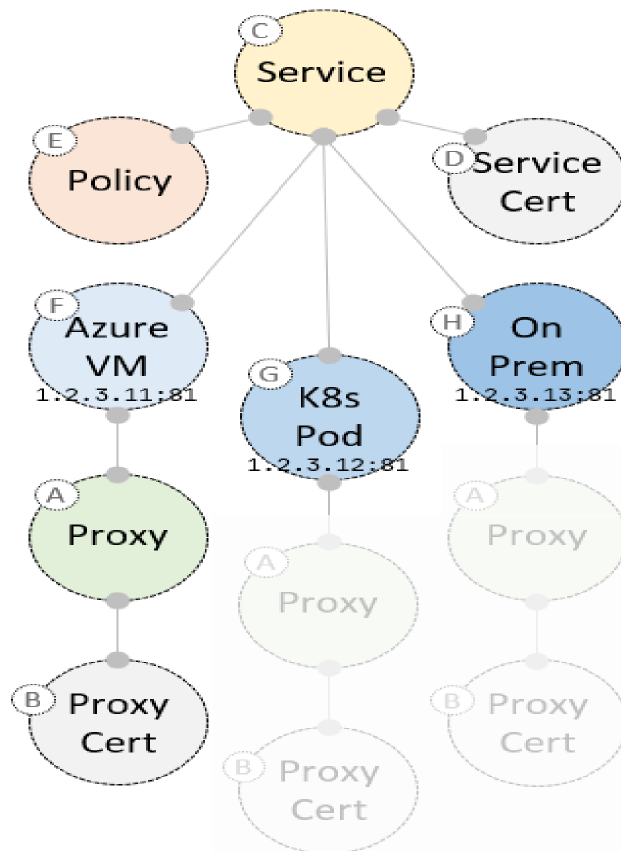
- Τον mesh catalog, ο οποίος λαμβάνει τις εξόδους των προηγούμενων στοιχείων (Certificate Manage, Endpoint Provides, Mesh Specification) , τις συνδυάζει και με βάση αυτές δημιουργεί μία παραμετροποίηση των proxies η οποία μεταδίδεται σε αυτούς μέσω του Proxy Control plane.

Όλα τα παραπάνω φαίνονται στο παρακάτω σχήμα.



Σχήμα 2.12: Αρχιτεκτονική του Control plane του OSM [57]

Το Σχήμα 2.14 δείχνει την πλευρά του control plane. Για καλύτερη κατανόηση του data plane υπάρχει το παρακάτω σχήμα.



Σχήμα 2.13: Αρχιτεκτονική του data plane του OSM [57]

Από αυτό το σχήμα βλέπουμε ότι ο proxy διαχειρίζεται την δικτυακή κίνηση που προορίζεται για την μικρουπηρεσία (όπου και αν τρέχει αυτή), και έχει το δικό του ξεχωριστό πιστοποιητικό (proxy certificate) το οποίο το χρησιμοποιεί για να επικοινωνήσει με το control plane.

Service Mesh	Istio	Linkerd	Consul	Kuma	Traefik Mesh	Open Service Mesh (OSM)
Current Version	1.15.3	2.12	1.13	2.0	1.4	1.0
License	Apache License 2.0	Apache License 2.0	Mozilla License	Apache License 2.0	Apache License 2.0	Apache License 2.0
Initiated by	IBM, Google, Lyft	Buoyant	HashiCorp	Kong	Traefik Labs	Microsoft
Service Proxy	Envoy	Linkerd2-proxy	Defaults to Envoy, Exchangeable	Envoy	Traefik Proxy on each node	Envoy
Ingress Controller	Envoy, Support for Kubernetes API Gateway	Any	Envoy, Support for Kubernetes API Gateway	any	any	Contour
Used in Production	yes	yes	yes	no	no	no
Supported Protocols						
TCP	yes	yes	yes	yes	yes	yes
HTTP/1.1+	yes	yes	yes	yes	yes	
HTTP/2	yes	yes	yes	yes	yes	yes
gRPC	yes	yes	yes	yes	yes	yes
Sidecar/Dataplane						
Automatic Sidecar Injection	yes	yes	yes	yes	yes (per Node)	yes
CNI plugin	yes	yes	yes	yes	no	no

Platform and Extensibility						
Platform	Kubernetes	Kubernetes	Kubernetes, Nomad, VMs, ECS	Kubernetes	Kubernetes	Kubernetes
Mesh Expansion Extension of the Mesh by containers/VMs outside the cluster	yes	no	yes	yes	no	no
Multi-Cluster Mesh Control and observe multiple clusters	yes	yes	yes	yes	no	planned
Service mesh Interface Compatibility						
Traffic Access Control	yes	no	yes	no	yes	yes
Traffic Specs	yes	no	no	no	yes	yes
TrafficSplit	yes	yes	no	no	yes	yes
Traffic Metrics	yes	yes	no	no	no	yes
Monitoring Features						
Access log Generation	yes	yes	yes	yes	yes	yes
Golden Signal Metrics Generation	yes	yes	yes	yes	yes	yes
Per Route metrics	yes	yes	depends on proxy	no	no	no
Dashboard	yes	yes	yes	yes	no	no

Distributed Tracing	Jaeger, Zipkin	OpenCensus	Datadog, Zipkin, OpenTracing	Jaeger, Zipkin	Zipkin, Jaeger, Datadog, Instana, Elastic, Haystack	Jaeger
Routing Features						
LoadBalancing	Round Robin, RandomWeighted, Least Request	EWMA	Round Robin, Random Weighted, Least Request, Ring hash, Manglev	Round Robin, Least Request, Ring hash, Manglev	yes	yes
Percentage-Based TrafficSplit	yes	yes	yes	yes	yes	yes
Resilience Features						
Circuit breaking	yes	no	yes	yes	yes	yes
Retry and Timeout	yes	yes	yes	yes	yes	no
Fault Injection	yes	yes	no	yes	no	no
Security Features						
mTLS	yes	yes	yes	yes	no	yes
mTLS Enforcement	yes	yes	yes	yes	no	yes

Πίνακας 2.1: Σύγκριση των service meshes [29]

2.4 Ανάλυση βιβλιογραφίας

Το service mesh σαν τεχνολογία είναι αρκετά καινούργια. Παρόλα αυτά υπάρχουν αρκετές σύγχρονες δημοσιεύσεις (papers) με αυτό, οι οποίες είτε προσπαθούν να βελτιώσουν κάποια χαρακτηριστικά του ίδιου του service mesh είτε το χρησιμοποιούν για την συγκομιδή πληροφορίας μεταξύ των μικρουπηρεσιών.
Ένα κλασικό παράδειγμα χρήσης των service meshes είναι σε δουλειές που αποσκοπούν στην βελτίωση του Kubernetes scheduler. Βασικό μειονέκτημα του Kubernetes scheduler είναι η αδυναμία του να αξιοποιεί σαν κριτήριο την επικοινωνία μεταξύ των pods, κατά την διαδικασία απόφασης της ανάθεσης ενός pod σε έναν κόμβο. Στο [30] με αφορμή την ελάχιστη καθυστέρηση των 5G δικτύων και την παραπάνω αδυναμία του Kubernetes scheduler παρουσιάζεται μία δουλειά που έχει γίνει πάνω στον Kubernetes scheduler με σκοπό την δημιουργία ενός νέου βελτιωμένου scheduler του NETMARKS (Network Metrics Aware K8s Scheduler). Ο NETMARKS χρησιμοποιεί το Istio για να λαμβάνει ποσοτικά δεδομένα (bytes) για την επικοινωνία μεταξύ των μικρουπηρεσιών. Με αυτόν τον τρόπο βρίσκει ποια pods επικοινωνούν περισσότερο μεταξύ τους και τα τοποθετεί στους ίδιους κόμβους. Η τοποθέτηση των pods στον ίδιο κόμβο μειώνει την συνολική καθυστέρηση (latency) και αυξάνει το εύρος ζώνης (bandwidth) της μεταξύ τους επικοινωνίας. Έτσι αναθέτοντας δύο pods με μεγάλη μεταξύ τους επικοινωνία στον ίδιο κόμβο ωφελείται περισσότερο η μείωση της συνολικής καθυστέρησης (latency) της εφαρμογής. Παρόμοια δουλειά γίνεται και στο [31] όπου προτείνεται μία λύση για την καλύτερη ανάθεση μικρουπηρεσιών στους κόμβους που βελτιώνει την ρυθμαπόδοση, την μέση καθυστέρηση, την μέγιστη καθυστέρηση κλπ. εξασφαλίζοντας παράλληλα ότι ικανοποιούνται όλες οι απαιτήσεις των μικρουπηρεσιών σχετικά με τους πόρους που χρειάζονται. Σε αυτή την λύση χρησιμοποιείται το Istio για την συλλογή δεδομένων σχετικά με την επικοινωνία μεταξύ των υπηρεσιών. Σκοπός είναι η δημιουργία ενός γράφου που κάθε ακμή του δείχνει την καθυστέρηση (latency) και την «ένταση» επικοινωνίας (traffic intensity) μεταξύ των μικρουπηρεσιών και κάθε κορυφή του συμβολίζει μία μικρουπηρεσία μαζί με τους πόρους που χρειάζεται. Αυτός ο γράφος διαμερίζεται σε επιμέρους τομές με τις μικρουπηρεσίες της κάθε τομής να τοποθετούνται στον ίδιο κόμβο. Οι δύο προηγούμενες προτάσεις έχουν ελεγχθεί σε υπάρχουσες εφαρμογές όπου φαίνεται η σημαντική μείωση στην καθυστέρηση (latency).
Εκτός από τους Kubernetes schedulers ένα άλλο πεδίο στο οποίο έχει εφαρμογή το service mesh είναι η απομακρυσμένη υπολογισιμότητα (edge computing). Στο [32] αντιμετωπίζεται η περίπτωση γεωγραφικά κατανεμημένων clusters που αποτελούνται από απομακρυσμένους εξυπηρετητές (edge servers) με περιορισμένους πόρους. Σε αυτή την διάταξη δημιουργείται ένας Service mesh Controller που χρησιμοποιεί το Istio για να κάνει το επονομαζόμενο cooperative load balancing, δηλαδή, επειδή οι εξυπηρετητές (servers) έχουν περιορισμένο αριθμό πόρων, όταν κάποιος από αυτούς υπερφορτωθεί (ξεπεραστεί δηλαδή ένα κατώφλι χρησιμοποίησης πόρων), τα επόμενα αιτήματα που προορίζονταν για αυτόν δεν δρομολογούνται σε ένα τυχαίο εξυπηρετητή του cluster αλλά δρομολογούνται σε έναν κοντινό του γεωγραφικά εξυπηρετητή (server) ώστε να κρατηθεί χαμηλά η καθυστέρηση (latency). Η συνεισφορά του service mesh σε απομακρυσμένα περιβάλλοντα αναλύεται και στο [61] όπου εξετάζεται η αρχιτεκτονική πολυπλοκότητα της εκτέλεσης ενός Service mesh σε τέτοιο περιβάλλον αλλά και η επίδρασή του στην δικτυακή κίνηση μεταξύ των υπηρεσιών του cluster (east-west traffic) αλλά και στην εισερχόμενη και εξερχόμενη κίνηση (north-south traffic). Στο [62] προτείνεται η αντικατάσταση της στατικής δρομολόγησης με έναν round robin αλγόριθμο που τρέχει πάνω στο Istio και αλλάζει δυναμικά βάρη. και τρέχει πάνω στο Istio και επαληθεύεται η χρησιμότητα του καθώς το φορτίο κατανέμεται με μεγαλύτερη ισονομία και η καθυστέρηση (response time) μικραίνει.
Άλλες δουλειές σχετικά με την επίδοση είναι, στο [33] η εκπαίδευση ενός πράκτορα με ενισχυτική μάθηση (reinforcement learning agent) ο οποίος χρησιμοποιεί μετρήσεις που συλλέγονται από το Istio για να παράγει τις βέλτιστες πολιτικές (policies) για μια σειρά στόχων (πχ μεγιστοποίηση της ρυθμαπόδοσης διατηρώντας την από άκρο σε άκρο καθυστέρηση πεπερασμένη), και στο [34] η πρόταση για συνδυαστική χρήση των service meshes με το eBPF (extended Berkeley Packet Filter)

επιτρέποντας την μείωση του επιπλέον κόστους (overhead) των service meshes μέσω της αξιοποίησης του πυρήνα (kernel) και της στιγμιαίας μεταγλώττισης (Just-in-Time Compilation).
Σε θέματα επίδοσης εστιάζει και το [36], στο οποίο παρουσιάζεται ένας ελεγκτής (controller), που μπορεί να εφαρμοστεί σε οποιαδήποτε υπηρεσία ανεξάρτητα της λειτουργίας της, ο οποίος επεκτείνει τις λειτουργίες του στατικού μηχανισμού διακοπής κυκλώματος (circuit breaker) αποφεύγοντας την υπερφόρτωση του συστήματος και περιορίζοντας τις αντίστοιχες αποτυχίες που προκύπτουν από αυτή, και εκμεταλλευόμενος το Istio προσπαθεί να μεγιστοποιεί την ρυθμαπόδοση (throughput) φροντίζοντας να κρατάει χαμηλά την καθυστέρηση (tail latency). Στο [37] χρησιμοποιείται το Istio για την εισαγωγή σφαλμάτων στην εφαρμογή και την ανάλυση των τρόπων αντιμετώπισής τους. Συγκεκριμένα εισάγονται στο δίκτυο σφάλματα σχετικά με την καθυστέρηση των αιτημάτων και την εγκατάλειψη της HTTP σύνδεσης (HTTP abortion). Μετά την εισαγωγή των σφαλμάτων δοκιμάζονται σαν τρόποι αντιμετώπισης η κλιμάκωση (scaling), η αλλαγή σε υγιή σύνδεση (failover) και η χρήση μηχανισμού διακοπής κυκλώματος (circuit breaker). Στο [63] προτείνεται μία υψηλής επίδοσης αρχιτεκτονική ειδικά για IoT περιβάλλοντα ενώ στο [64] γίνεται μία συστηματική και ποσοτική ανάλυση της σύγχρονης κατάστασης των Service meshes με στόχο να προσδιοριστούν οι ανάγκες διαχείρισης της δικτυακής κίνησης στις εφαρμογές που έχουν απαιτήσεις υψηλής επίδοσης.
Πολύ σημαντικό ρόλο στην βελτίωση και την ανάπτυξη του service mesh είναι η δυνατότητα αξιολόγησής του. Είναι χρήσιμο δηλαδή μία εταιρεία να γνωρίζει το κόστος (overhead) που προσθέτει η υιοθέτηση ενός service mesh στην εφαρμογή της. Στο [35] παρουσιάζεται το εργαλείο MeshInsight. Το MeshInsight είναι ένα εργαλείο που στόχος του είναι να μπορεί να ενημερώνει τους χρήστες του για το επιπλέον κόστος (στην δημοσίευση σαν κόστος ορίζεται η αύξηση της καθυστέρησης και η επιπλέον κατανάλωση της ΚΜΕ που απαιτείται για την επεξεργασία μηνυμάτων από το service mesh) που προσθέτει ένα service mesh, να βρίσκει τους κύριους παράγοντες στους οποίους οφείλεται το κόστος (πχ σε λειτουργία HTTP proxy το έξτρα κόστος οφείλεται στο διάβασμα των μηνυμάτων) και να ποσοτικοποιεί τις βελτιστοποιήσεις που γίνονται στα service meshes. Στην ποσοτικοποίηση της επίδρασης συγκεκριμένα του Istio, εστιάζει και το [60].
Αρκετές δημοσιεύσεις (papers) σχετίζονται και με το κομμάτι της ασφάλειας. Τα service meshes έχουν την δυνατότητα κατά την διάρκεια εκτέλεσης της εφαρμογής να εφαρμόζουν κανόνες που ορίζονται από τους ανθρώπινους χρήστες. Στο [38] γίνεται προσπάθεια να ενταχθεί στο Istio ένα συστατικό (component), που ονομάζεται AAF (Assurance Assessment Framework), το οποίο με βάση τις αλλαγές στο περιβάλλον της εφαρμογής μπορεί να εκτιμάει δυναμικά το ρίσκο (δηλαδή την επικινδυνότητα) που βρίσκεται μία υπηρεσία και να το περιορίζει αλλάζοντας αυτόματα τους κανόνες της δικτυακής κίνησης (traffic flow configuration). Στο [39] προτείνεται μία διάταξη για κρυπτογράφηση των μηνυμάτων που χρησιμοποιεί το service mesh για τον έλεγχο και την παρακολούθηση της δικτυακής κίνησης έτσι ώστε να μην μπορούν να τα διαβάζουν οι άλλοι ενοικιαστές (tenants) της εφαρμογής. Σε περιβάλλοντα με πολλούς ενοικιαστές αναφέρεται και η Baranova Oksana στο [59] που προτείνει ένα τρόπο διαχείρισης πιστοποιητικών στο Istio για να αντιμετωπίσει το πρόβλημα ασφάλειας μεταξύ των διάφορων ενοικιαστών. Πολύ μεγάλο ενδιαφέρον έχει στο [40] η δουλειά που γίνεται για την προφύλαξη του service mesh από έναν κακόβουλο διαχειριστή του cluster (rogue cluster administrator). Σε αυτή την δημοσίευση προτείνεται το data plane του service mesh να μην εμπιστεύεται 100% το control plane αλλά να δέχεται διαφορετικά configurations μόνο όταν αυτά είναι υπογεγραμμένα από τον ίδιο τον ιδιοκτήτη της εφαρμογής (application owner).

Κεφάλαιο 3 Υλοποίηση

3.1 Περιγραφή Διάταξης

3.1.1 Εισαγωγή

Η διάταξη πάνω στην οποία εργαζόμαστε βασίζεται σε ένα Kubernetes cluster δύο κόμβων. Ο ένας κόμβος έχει τον ρόλο του μάστερ (master node). Σε αυτόν τρέχει το control plane του Kubernetes. Ο άλλος κόμβος είναι ένας απλός εργάτης (worker node) που εκπροσωπεί το data plane του cluster. Το cluster χρησιμοποιεί το container runtime CRI-O και το calico σαν CNI. Στο cluster τρέχουν το linkerd (το service mesh του cluster), το linkerd-viz (ένα extension του linkerd) και δύο (demo) εφαρμογές (οι emojivoto και booksapp). Καθένα από τα παραπάνω τρέχει σε ένα ξεχωριστό namespace οπότε σύνολο έχουμε πέντε namespaces τα linkerd, linkerd-viz, emojivoto, booksapp, kube-system (το namespace του kubernetes).

3.1.2 Περιγραφή Εφαρμογών

Η εφαρμογή emojivoto σου επιτρέπει να ψηφίσεις το αγαπημένο σου emoji από κάποια συγκεκριμένη λίστα αλλά και να δεις την μέχρι τώρα βαθμολογία για το κάθε emoji. Η εφαρμογή αποτελείται συνολικά από τέσσερεις μικρουπηρεσίες (microservices) καθεμία εκ των οποίων έχει ένα διαφορετικό ρόλο.

- Η μικρουπηρεσία web αποτελεί το frontend της εφαρμογής. Παρουσιάζει στον χρήστη το σύνολο με τα emoji από το οποίο ο χρήστης μπορεί να ψηφίσει. Φροντίζει να επικοινωνήσει με τις κατάλληλες μικρουπηρεσίες (emoji , voting) ώστε να απαντήσει στο εκάστοτε αίτημα του χρήστη.

- Η μικρουπηρεσία emoji μπορεί να επιστρέψει μια λίστα με όλα τα emoji και να βρει κάποιο συγκεκριμένο emoji με βάση το shortcode του. Συνήθως απαντάει σε αιτήματα της μικρουπηρεσίας web.

- Η μικρουπηρεσία voting είναι υπεύθυνη για την ψηφοφορία των emoji αυξάνει δηλαδή το πλήθος των ψήφων του emoji που ψηφίστηκε. Συνήθως απαντάει σε αιτήματα της μικρουπηρεσίας web.

- Η μικρουπηρεσία vote-bot δημιουργεί τεχνητή δικτυακή κίνηση (traffic). Στέλνει δηλαδή αιτήματα στην web μικρουπηρεσία.

Να σημειωθεί ότι όταν ψηφίζεται το emoji Doughnut επιστρέφεται επίτηδες error. Η εφαρμογή τρέχει πάνω στο Kubernetes cluster. Για την κάθε υπηρεσία υπάρχει ένα service, ένα deployment και ένα pod.

Η εφαρμογή booksapp σου δίνει την δυνατότητα να διαχειριστείς την βιβλιοθήκη σου προσθέτοντας και διαγράφοντας βιβλία και συγγραφείς σε ένα υπάρχων σύνολο με βιβλία και συγγραφείς. Η δομή της είναι αντίστοιχη με αυτή της emojivoto. Αποτελείται και αυτή από τέσσερις μικρουπηρεσίες.

- Η μικρουπηρεσία webapp λειτουργεί αντίστοιχα με την μικρουπηρεσία web και αποτελεί το frontend της εφαρμογής. Για να μπορέσει να απαντήσει επιτυχώς στα αιτήματα που δέχεται, επικοινωνεί με τις μικρουπηρεσίες books και authors.

- Η μικρουπηρεσία books διαχειρίζεται τα βιβλία, δέχεται αιτήματα από την μικρουπηρεσία webapp και επικοινωνεί με την μικρουπηρεσία authors ώστε να συσχετιστεί το βιβλίο με έναν συγγραφέα.

- Η μικρουπηρεσία authors αντίστοιχα, δέχεται και αυτή αιτήματα από την webapp και επικοινωνεί και αυτή με την μικρουπηρεσία books.

- Η μικρουπηρεσία traffic είναι αντίστοιχη της vote-bot και παράγει τεχνητή δικτυακή κίνηση (traffic). Στέλνει δηλαδή αιτήματα στην μικρουπηρεσία webapp.

Στην εφαρμογή booksapp όταν γίνονται αιτήματα από την μικρουπηρεσία books στην μικρουπηρεσία authors το ποσοστό επιτυχίας είναι επίτηδες 50% (τεχνητό transient error).

3.1.3 Εγκατάσταση Linkerd

Στο kubernetes cluster έχουμε εγκαταστήσει το service mesh linkerd. Το control plane και το data plane του linkerd τρέχουν και τα δύο στο data plane του kubernetes cluster δηλαδή, στον κόμβο εργάτη (worker node). Κάθε εκτέλεση (deployment) έχει παραμετροποιηθεί με την εντολή linkerd inject έτσι ώστε να έχει στα pod specs το annotation linkerd.io/inject: enabled. Αυτό επιτρέπει στο linkerd, όταν δημιουργείται ένα pod, να δημιουργεί μέσα σε αυτό ένα επιπλέον container το οποίο θα έχει τον ρόλο του data plane proxy (sidecar proxy). Κάνοντας επανεκκίνηση όλες τις εκτελέσεις (deployments) των εφαρμογών πετυχαίνουμε να έχουμε σε όλα τα pods δύο containers, ένα στο οποίο θα τρέχει η εφαρμογή και ένα στο οποίο θα τρέχει ο linkerd proxy.
Πέρα από το κυρίως linkerd στο cluster εγκαθιστούμε και ένα βασικό add-on του linkerd, το linkerd-viz. Το linkerd viz μας δίνει πρόσβαση σε περισσότερες εντολές ενισχύοντας τις δυνατότητες παρακολούθησης (telemetry).

3.2 Αυτόματη Κλιμάκωση (Autoscaling)

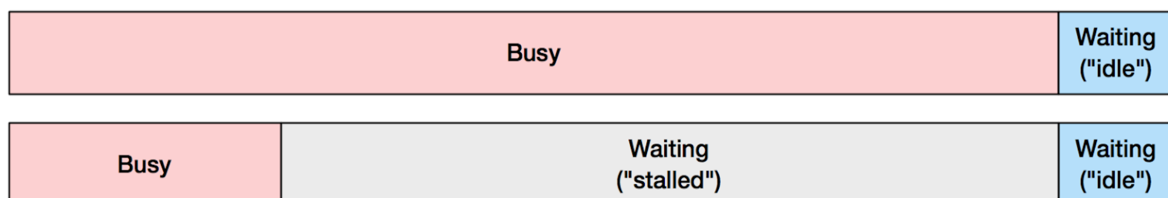
3.2.1 Περιγραφή υπηρεσίας

Όπως αναφέρεται και στην περιγραφή της διάταξης κάθε container της εφαρμογής συνοδεύεται από ένα container που τρέχει ο linkerd proxy. Όλη η δικτυακή κίνηση που προορίζεται για (ή προέρχεται από) ένα container της εφαρμογής περνάει πρώτα από τους linkerd proxies. Αυτό σημαίνει ότι το linkerd έχει πρόσβαση σε όλη την πληροφορία που ανταλλάσσεται μεταξύ δύο μικρουπηρεσιών. Μέσω αυτής της πληροφορίας το linkerd μπορεί και συλλέγει τιμές από μετρικές (πχ latency, requests-per-second, success rate κλπ.) σχετικά με τις μικρουπηρεσίες που επικοινωνούν (telemetry). Οι παραπάνω μετρήσεις αποθηκεύονται προσωρινά με την μορφή χρονοσειρών σε ένα προϋπάρχον (build in) Prometheus στιγμιότυπο (instance) το οποίο υπάρχει στο linkerd-viz namespace.
Το Kubernetes έχει την δυνατότητα για κλιμάκωση (scale) με διαφορετικούς τρόπους και διαφορετικές παραμέτρους. Όσον αφορά τα pods μπορεί να γίνει κλιμάκωση (scaling) χρησιμοποιώντας είτε τον vertical pod autoscaler, με αύξηση των πόρων (resources) του κάθε pod, είτε τον horizontal pod autoscaler με πολλαπλασιασμό του πλήθους των pods. Ο HorizontalPodAutoscaler (ο οποίος και χρησιμοποιήθηκε στην συγκεκριμένη υπηρεσία) μπορεί να κλιμακώσει μία εκτέλεση (deployment) με βάση μετρικές που δεν περιορίζονται στην κατανάλωση της μνήμης (memory) και της ΚΜΕ (CPU, default μετρήσεις του kubernetes), όπως είναι για παράδειγμα η χρονική καθυστέρηση (latency). Οι μετρικές που σχετίζονται με τα pods αλλά δεν σχετίζονται με την μνήμη (memory) ή την ΚΜΕ (CPU) ονομάζονται κατά παραγγελία μετρικές (custom metrics) και λαμβάνονται από το custom.metrics.k8s.io API [41].
Ο στόχος της συγκεκριμένης υπηρεσίας είναι να μπορεί να κλιμακώνεται η εκτέλεση (deployment) web της εφαρμογής emojivoto με βάση τις μετρήσεις που συλλέγει το linkerd (ανήκουν στις custom

metrics) και συγκεκριμένα με βάση την χρονική καθυστέρηση (latency) της μικρουπηρεσίας. Για να επιτευχθεί αυτό γίνονται όλες οι απαραίτητες ενέργειες ώστε τα δεδομένα που αποθηκεύονται στο Prometheus από το linkerd, να φτάνουν με την σωστή μορφή στον HPA. O HPA τα αξιοποιεί για να κλιμακώσει την εκτέλεση (deployment) web όταν η χρονική της καθυστέρηση (latency) ξεπεράσει το κατώφλι (threshold) που του έχουμε θέσει.

3.2.2 Χρησιμότητα της Υπηρεσίας

Το kubernetes δίνει την δυνατότητα της κλιμάκωσης με βάση, το φορτίο που διαχειρίζεται η κεντρική μονάδα επεξεργασίας (CPU) και την κατανάλωση της μνήμης (memory). Αυτές οι δύο μετρικές δεν είναι πάντα ο καλύτερος τρόπος να αποφασιστεί αν χρειάζεται να υπάρξει κλιμάκωση μίας υπηρεσίας.
Η μετρική της κατανάλωσης της μνήμης είναι πολύ ειδική. Η χρησιμότητα της είναι εξαρτημένη από το φορτίο και δεν αντικατοπτρίζει πάντα αν μία υπηρεσία χρειάζεται να κλιμακωθεί ή όχι. Όταν μία υπηρεσία κάνει έντονη χρήση της μνήμης (memory intensive service) τότε η μετρική της μνήμης είναι χρήσιμη. Υπάρχουν όμως φορές που μία υπηρεσία δεν κάνει έντονη χρήση μνήμης αλλά χρειάζεται να κλιμακωθεί (π.χ. λόγω έντονης χρήσης της κεντρικής μονάδας επεξεργασίας, CPU intensive service).
Η μετρική της κατανάλωσης της κεντρικής μονάδας επεξεργασίας (CPU) έχει και αυτή πρόβλημα γιατί δεν είναι ακριβής. Ο Brendan Gregg εξηγεί [42] ότι η μετρική της χρησιμοποίησης της ΚΜΕ (CPU utilization) δηλώνει το ποσοστό του χρόνου κατά το οποίο η ΚΜΕ (CPU) δεν τρέχει το idle thread. Τονίζει όμως ότι αυτό το διάστημα του χρόνου δεν σημαίνει ότι η ΚΜΕ (CPU) είναι απασχολημένη καθώς μπορεί να έχει σταματήσει λόγω διάφορων αναβολών (stalls). Τα παραπάνω συνοψίζονται και στο παρακάτω σχήμα.



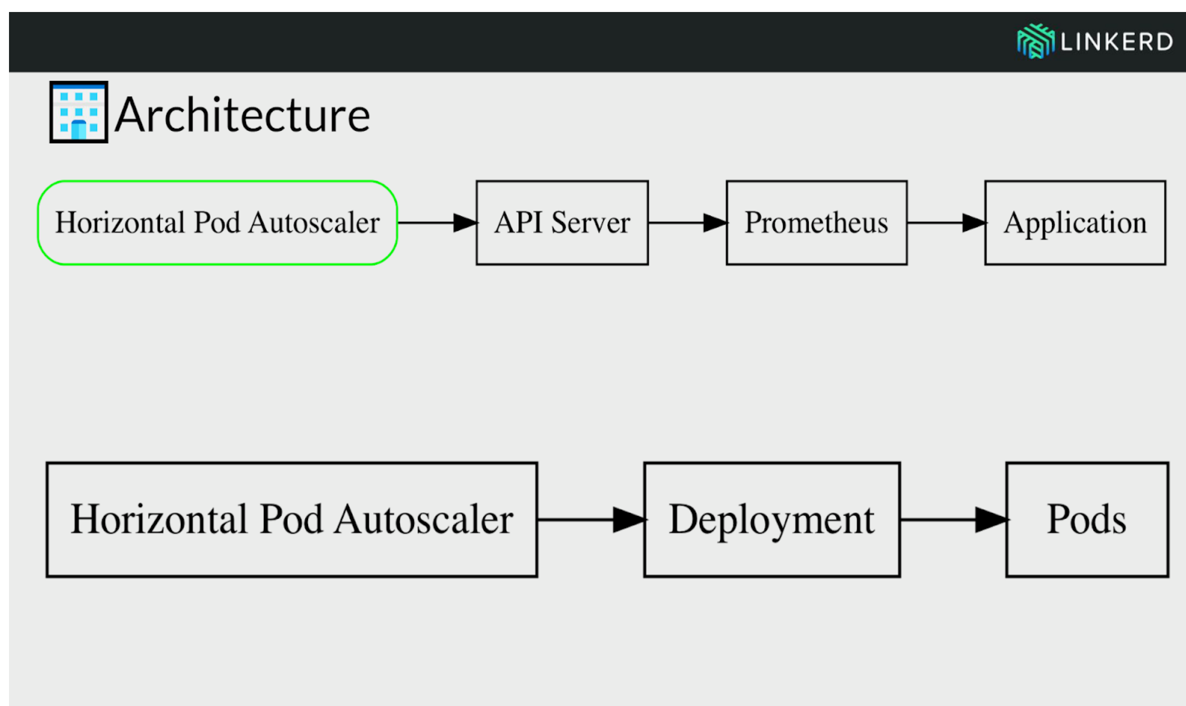
Σχήμα 3.1: Φαινομενική χρήση της CPU vs Κανονική Χρήση της CPU [42]

Το συμπέρασμα από τις παραπάνω διαπιστώσεις είναι ότι μία υπηρεσία μπορεί να έχει μεγάλη χρησιμοποίηση της ΚΜΕ (CPU utilization) χωρίς όμως το πραγματικό πρόβλημα να είναι οι απαιτήσεις της ΚΜΕ (δηλαδή χωρίς να είναι CPU bound).
Οι ιδιαιτερότητες των προηγούμενων μετρικών δίνουν αξία στην εύρεση και άλλων, νέων μετρικών. Οι χρήστες μίας εφαρμογής δεν βλέπουν την μνήμη ή την χρησιμοποίηση της ΚΜΕ. Η μετρική η οποία επηρεάζει άμεσα την εμπειρία του πελάτη είναι ο χρόνος που περιμένει μέχρι να λάβει απάντηση στο αίτημά του, δηλαδή η χρονική καθυστέρηση (latency). Η χρησιμότητα της συγκεκριμένης υπηρεσίας βρίσκεται στην χρήση της χρονικής καθυστέρησης (latency) σαν μετρική που θα καθορίζει την κλιμάκωση της υπηρεσίας web. Επίσης δείχνουμε πως μπορούμε να χρησιμοποιούμε το linkerd για να πετύχουμε κλιμάκωση με βάση διαφορετικές μετρικές από τις προκαθορισμένες (default) (memory, CPU) του Kubernetes.

3.2.3 Υλοποίηση

Το linkerd συλλέγει αυτόματα δεδομένα σχετικά με την χρονική καθυστέρηση (latency), τον ρυθμό άφιξης αιτημάτων (requests per second), και τα σφάλματα (errors) για κάθε υπηρεσία. Αυτά τα δεδομένα αποθηκεύονται με την μορφή χρονοσειρών στο Prometheus στιγμιότυπο (instance) που

τρέχει στο namespace linkerd-viz. Για την υλοποίηση της υπηρεσίας πρέπει να ταΐσουμε τα δεδομένα, που συλλέγει το linkerd σχετικά με την χρονική καθυστέρηση (latency) (και συγκεκριμένα αυτά που αφορούν το latency της εκτέλεσης (deployment) web), στον HorizontalPodAutoscaler, ο οποίος θα κλιμακώσει την εκτέλεση (deployment) web. Πρώτο βήμα είναι η παραμετροποίηση του HPA ώστε να χρησιμοποιεί την μετρική της χρονικής καθυστέρησης (latency) για την κλιμάκωση. Επειδή η μετρική της χρονικής καθυστέρησης (latency) δεν ανήκει στις προκαθορισμένες (default) μετρικές του kubernetes (memory, CPU) ο HPA θα παίρνει τις μετρήσεις από το custom.metrics.k8s.io API. Το custom.metrics.k8s.io API είναι ένα extension API του kubernetes. Για να λειτουργήσει ένα extension API πρέπει να διαμορφώσουμε (configure) το aggregation layer [43] και να καταχωρήσουμε το extension API.
Έχοντας κάνει τις απαραίτητες ενέργειες για την σωστή λειτουργία του extension API πρέπει να τροφοδοτήσουμε τα δεδομένα από το Prometheus στο custom.metrics.k8s.io API. Το API αυτό δεν μπορεί να επικοινωνήσει κατευθείαν με το Prometheus για αυτό εγκαθιστούμε τον Prometheus adapter. Ο Prometheus adapter κάνει τα ερωτήματα (queries) που του ορίζουμε στο Prometheus και τα αποτελέσματα που λαμβάνει τα στέλνει στο custom.metrics.k8s.io. Επειδή ο Prometheus adapter χρειάζεται διάφορα kubernetes objects δεν είναι εύκολο να τον εγκαταστήσουμε μέσω ενός αρχείου με την εντολή kubectl apply -f. Για την εγκατάσταση του χρησιμοποιείται ο package manager του kubernetes, το Helm. Μέσω του Helm με την εντολή
helm install kopa-prometheus prometheus-community/prometheus-adapter -f prometheus-adapter.yaml -n linkerd-viz εγκαθιστούμε το Prometheus adapter και μάλιστα το παραμετροποιούμε με τα ερωτήματα (queries) που μας χρειάζονται προς το Prometheus. Με τον Prometheus adapter εγκατεστημένο ο HPA μπορεί και βλέπει πλέον τις μετρήσεις για την χρονική καθυστέρηση (latency) που μάζεψε το linkerd. Η ροή της πληροφορίας φαίνεται στο παρακάτω σχήμα.



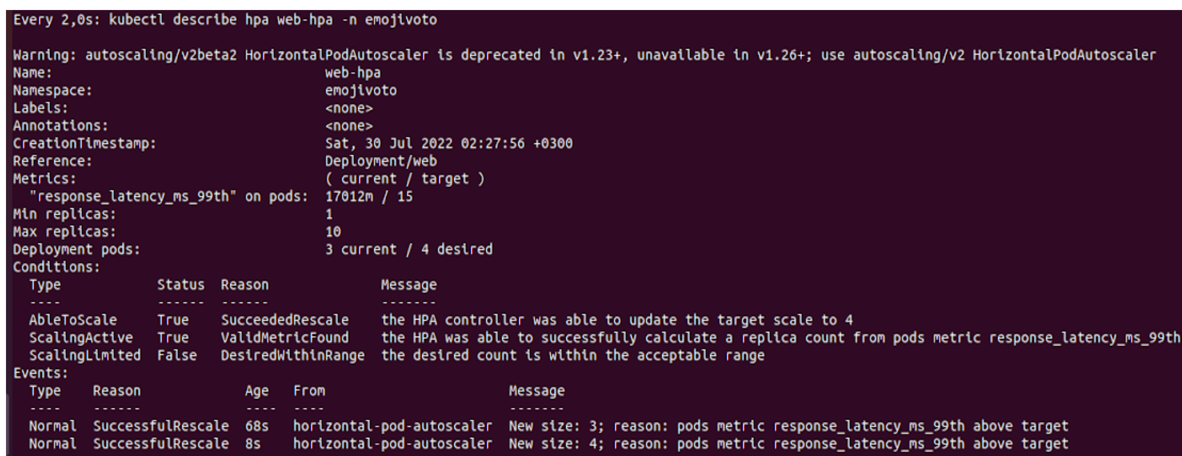
Σχήμα 3.2: Ροή πληροφορίας του HPA [44]

Ο HPA κάνει αίτημα για την μέτρηση από το kubernetes API, αυτό στέλνει ως διαμεσολαβητής το αίτημα στο custom.metrics.k8s.io extension API το οποίο πρακτικά υλοποιείται από τον Prometheus adapter ο οποίος κάνει το κατάλληλο ερώτημα (query) στον Prometheus και επιστρέφει τα δεδομένα στον HPA. Ο HPA συγκρίνει την τιμή της χρονικής καθυστέρησης (latency) που παίρνει από το Prometheus adapter με το κατώφλι (threshold) των 15 ms που του έχουμε θέσει. Αν η τιμή που μετράει ξεπεράσει τα 15 ms τότε αυτός κλιμακώνει (scales) την web εκτέλεση (deployment) πολλαπλασιάζοντας τον αριθμό των pods.

Έλεγχος σωστής λειτουργίας

Για να βεβαιώσουμε ότι η υπηρεσία πράγματι λειτουργεί χρησιμοποιούμε το load testing εργαλείο Vegeta με το οποίο βομβαρδίζουμε με αιτήματα ένα route (endpoint+method) της εκτέλεσης (deployment) web. Πριν όμως ξεκινήσουμε αυτή την διαδικασία πρέπει να έχουμε πρόσβαση στα αντίστοιχα routes. Επειδή η εκτέλεση τρέχει εσωτερικά στο kubernetes cluster δεν έχει επαφή με τον κόσμο έξω από αυτό. Για να λύσουμε αυτό το πρόβλημα δημιουργούμε ένα service (είναι Kubernetes object) τύπου nodePort το οποίο εκθέτει (exposes) την εκτέλεση (deployment) web εκτός cluster επιτρέποντας την πρόσβαση στα επιθυμητά routes. Έπειτα ξεκινάμε να στέλνουμε συνεχόμενα αιτήματα στο endpoint http://192.168.1.20:30630/api/list.

Εικόνα 3.1: Αιτήματα του Vegeta



Εικόνα 3.2: Υπέρβαση του κατωφλιού και εντολή για κλιμάκωση από τον HPA

Όταν η χρονική καθυστέρηση (latency) ξεπερνάει το φράγμα των 15 ms παρατηρούμε ότι ο HPA δίνει εντολή για την δημιουργία καινούργιων pods για την εκτέλεση (deployment) web, που σημαίνει ότι ο HPA έκανε επιτυχώς την επιθυμητή κλιμάκωση (scale).

3.2.4 Επέκταση

Το linkerd μας δίνει την δυνατότητα μέσω των service profiles να συλλέγουμε μετρήσεις ειδικά για συγκεκριμένα routes (method and endpoint) μίας υπηρεσίας. Αν έχουμε μία εφαρμογή η οποία έχει κάποια κρίσιμα για την λειτουργικότητα της (mission-critical) endpoints τότε μπορούμε να αξιοποιήσουμε τα service profiles σε συνδυασμό με όλη την προηγούμενη διάταξη ώστε να κλιμακώνουμε την συγκεκριμένη υπηρεσία με βάση μετρικές που θα είναι εξειδικευμένες στα κρίσιμα (mission critical) endpoints. Η παραπάνω λειτουργικότητα μπορεί να εξελιχθεί περεταίρω. Το Prometheus μέσω της συνάρτησης predict_linear() της PromQl έχει την δυνατότητα να κάνει γραμμική παλινδρόμηση (simple linear regression), δηλαδή να κάνει μια εκτίμηση για τις επόμενες τιμές της μετρικής που χρησιμοποιούμε. Μπορεί λοιπόν το Prometheus αξιοποιώντας την συγκεκριμένη συνάρτηση της PromQl να προβλέψει ότι οι τιμές της χρονικής καθυστέρησης (latency) που ακολουθούν θα ξεπεράσουν το κατώφλι (threshold) που έχουμε ορίσει στον HPA. Έτσι ο HPA βλέποντας τις μελλοντικές τιμές, ξεκινάει την κλιμάκωση πριν πραγματικά ξεπεραστεί το κατώφλι (threshold) πετυχαίνοντας οι τιμές της χρονικής καθυστέρησης (latency) να μην γίνουν πολύ υψηλές μέχρι να ετοιμαστούν τα καινούργια pods.

3.3 Διαμοιρασμός Δικτυακής κίνησης

3.3.1 Περιγραφή Υπηρεσίας

Στο kubernetes cluster στην εφαρμογή emojivoto θέλουμε να χρησιμοποιήσουμε μία νέα έκδοση (version) της εκτέλεσης (deployment) web η οποία έχει μία αλλαγή στην αρχική ιστοσελίδα που εμφανίζει. Αφού ανεβάσουμε την νέα εκτέλεση (deployment) στο cluster χρησιμοποιούμε το trafficsplit χαρακτηριστικό (feature) του linkerd [45] για να μετακινήσουμε ένα μέρος της δικτυακής κίνησης από την παλιά έκδοση της εκτέλεσης (deployment) web στην νέα έκδοση.

3.3.2 Χρησιμότητα της Υπηρεσίας

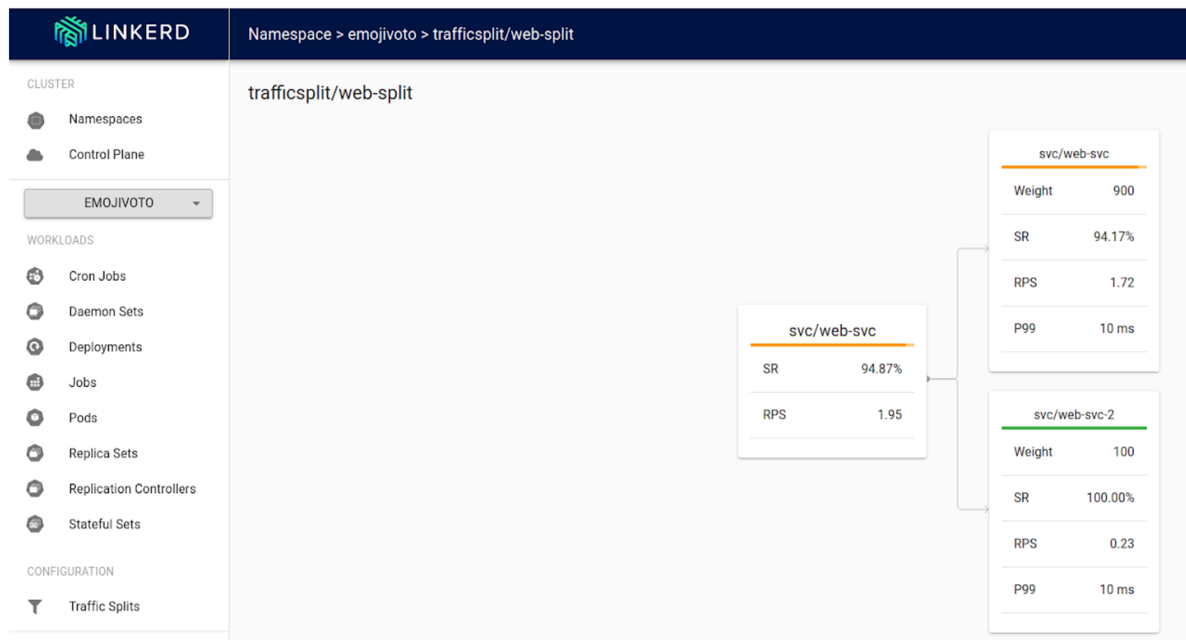
Πολλές φορές χρειάζεται να αναβαθμιστεί η έκδοση του κώδικα που βρίσκεται στην παραγωγή (production) σε μία νέα έκδοση (π.χ. λόγω ενός καινούργιου χαρακτηριστικού (new feature), λόγω διόρθωσης προηγούμενων προβλημάτων (bugs) κλπ.). Αυτή η μετάβαση δεν είναι πάντα απλή γιατί μπορεί να εισάγει νεκρό χρόνο (downtime) στους χρήστες της εφαρμογής μέχρι να αρχίσουν να δρομολογούνται στην νέα έκδοση. Επίσης σε περίπτωση που ο νέος κώδικας παρουσιάσει προβλήματα οι πελάτες θα έχουν κακή εμπειρία με την εφαρμογή μέχρι αυτά να διορθωθούν. Το linkerd με την υπηρεσία TrafficSplit έρχεται να επιλύσει τα παραπάνω προβλήματα. Η βασική ιδέα στην οποία στηρίζεται και υλοποιεί το TrafficSplit είναι η ανεξαρτητοποίηση της φάσης της εκτέλεσης (deployment) του κώδικα από την φάση της διάθεσής του στην παραγωγή (production release). Τα πλεονεκτήματα είναι πολλά:

- Ευκολία στην μετάβαση των χρηστών στην νέα έκδοση που συνεπάγεται ελαχιστοποίηση του νεκρού χρόνου (downtime).

- Σταδιακή μετάβαση των χρηστών στην νέα έκδοση.
    - Επιτρέπει την έγκαιρη αναγνώριση των προβλημάτων που προκύπτουν χωρίς να χρειάζεται να εκτεθούν όλοι οι πελάτες σε αυτά.
    - Επιτρέπει την γρήγορη απόσυρση των πελατών από τον νέο προβληματικό κώδικα και την επιστροφή τους στην παλιά λειτουργική έκδοση.

3.3.3 Υλοποίηση

Στο namespace emojivoto θέλουμε να μετακινήσουμε την δικτυακή κίνηση από την παλιά εκτέλεση (deployment) web στην νέα web-svc-2. Για την μετακίνηση αυτή χρησιμοποιείται το TrafficSplit CRD (Custom Resource Definition) του kubernetes. Για να μπορέσει το linkerd να κατανοεί και να διαμορφώνει το TrafficSplit CRD εγκαθιστούμε το Linkerd-SMI [46]. Έχοντας εγκαταστήσει το Linkerd-SMI δημιουργούμε ένα TrafficSplit CRD στο οποίο ορίζουμε δύο βάρη τα οποία θα χωρίζουν την δικτυακή κίνηση σε 90% να πηγαίνει στην παλιά υπηρεσία (service) web-svc και το 10% να πηγαίνει στην καινούργια υπηρεσία (service) την web-svc-2.



Εικόνα 3.3: Διαμοιρασμός δικτυακής κίνησης μεταξύ των δύο services

Όταν εφαρμόζουμε αυτό το CRD (apply), αυτόματα δημιουργείται ένα Service Profile για την υπηρεσία (service) web με το οποίο το linkerd θα δρομολογεί το 90% της δικτυακής κίνησης στην υπηρεσία (service) web-svc η οποία με την σειρά της θα στέλνει την κίνηση στην παλιά εκτέλεση (deployment) web και το 10% στην υπηρεσία (service) web-svc-2 η οποία με την σειρά της θα στέλνει την δικτυακή κίνηση στην εκτέλεση (deployment) web-svc-2. Στην συνέχεια για να μετακινήσουμε επιπλέον κομμάτι της δικτυακής κίνησης αλλάζουμε το TrafficSplit CRD.

3.3.4 Επέκταση

Για να επεκταθεί η παραπάνω υπηρεσία βάζοντας ένα ακόμα επίπεδο αυτοματισμού πρέπει να γίνει χρήση του Flagger [47]. Αφού εγκατασταθεί το flagger μας δίνει πρόσβαση στο Canary CRD [48]. Χρησιμοποιώντας το Canary CRD ορίζονται οι κανόνες με βάση τους οποίους θα διεξαχθεί η

μετάβαση από την παλιά έκδοση στην καινούργια. Μόλις γίνει η εκτέλεση (deploy) του Canary CRD το flagger δημιουργεί μία καινούργια εκτέλεση (deployment) στην οποία απευθύνεται το Canary CRD την οποία ονομάζει με το ίδιο όνομα -primary και στην οποία μεταφέρεται όλη η δικτυακή κίνηση. Έτσι υπάρχουν δύο εκτελέσεις (deployments) η κύρια εκτέλεση (primary deployment), η οποία δέχεται όλη την δικτυακή κίνηση , και η εκτέλεση καναρίνι (canary deployment), η οποία δέχεται μηδενική δικτυακή κίνηση. Οποτεδήποτε γίνει κάποια αναβάθμιση στην υπηρεσία και άρα και στην αντίστοιχη εκτέλεση (deployment) το flagger την αντιλαμβάνεται και πυροδοτεί την μετάβαση με βάση τους κανόνες που έχουν οριστεί στο Canary CRD. Κατά την μετάβαση μετακινούνται ποσοστά δικτυακής κίνησης από την κύρια εκτέλεση στην εκτέλεση καναρίνι (canary deployment), γίνονται οι έλεγχοι που έχουν οριστεί στο Canary CRD και αν δεν αποτύχουν η διαδικασία συνεχίζεται μέχρι να μεταφερθεί όλη η δικτυακή κίνηση στην εκτέλεση καναρίνι (canary deployment). Μόλις τελειώσει η διαδικασία της μετάβασης το flagger αντιγράφει την αλλαγή στην κύρια εκτέλεση (primary deployment) και όλη η δικτυακή κίνηση μεταφέρεται στην κύρια εκτέλεση. Τα πλεονεκτήματα σε σχέση με το απλό TrafficSplit είναι η αυτοματοποίηση της σταδιακής μετάβασης της δικτυακής κίνησης και οι αυτόματοι έλεγχοι κατά την διάρκεια της μετάβασης. Ο νέος κώδικας δηλαδή εκτίθεται στους πελάτες ταυτόχρονα(με την εκτέλεση του) αφού πρώτα έχει ελεγχθεί.

3.4 Retries and Timeouts

3.4.1 Περιγραφή Υπηρεσίας

Στο kubernetes cluster τρέχει η εφαρμογή booksapp [49]. Στην εφαρμογή αυτή παρατηρούμε ότι η μικροϋπηρεσία books έχει ποσοστό επιτυχίας μικρότερο από 100%. Το σφάλμα (error) στο οποίο οφείλεται το <100% ποσοστό επιτυχίας (success rate) είναι παροδικό (transient) δηλαδή μπορεί σε δύο διαδοχικά ίδια αιτήματα (requests) την μία φορά να εμφανιστεί και την άλλη όχι. Στην συγκεκριμένη υπηρεσία χρησιμοποιούνται οι μηχανισμοί retry και timeout για να βελτιώσουν τα ποσοστά επιτυχίας.

3.4.2 Χρησιμότητα Υπηρεσίας

Η χρησιμότητα του μηχανισμού των retries αναδεικνύεται όταν το σφάλμα (error) που έχει μία εφαρμογή είναι παροδικό (transient), δηλαδή μπορεί το ίδιο αίτημα που απέτυχε τώρα μερικά δευτερόλεπτα μετά να επιτύχει. Σε τέτοιες περιπτώσεις δεν χρειάζεται ο πελάτης (client) να στείλει νέο το αίτημα αλλά μπορεί να ξανασταλεί το ίδιο αίτημα το οποίο έχει αποθηκευτεί σε ενδιάμεσο διαμεσολαβητή (proxy). Με αυτό τον τρόπο μεγαλώνει το ποσοστό επιτυχίας (success rate) στην πλευρά του πελάτη (effective success) γιατί ο πελάτης δεν βλέπει όλα τα ενδιάμεσα retries αλλά μόνο αυτό που πέτυχε. Αντίθετα η πλευρά του εξυπηρετητή (actual traffic) δεν βλέπει αύξηση στο ποσοστό επιτυχίας γιατί ο εξυπηρετητής βλέπει τις αποτυχίες όλων των ενδιάμεσων αιτημάτων.
Επειδή τα συνεχόμενα retries παίρνουν χρόνο ενδέχεται να αυξηθεί σημαντικά η χρονική καθυστέρηση (latency) της υπηρεσίας. Αυτό μπορεί να έχει σαν αποτέλεσμα να υπάρχουν φορές που χρειάζεται να περιμένουμε μη βιώσιμους χρόνος (πχ 10min). Για να αποφευχθούν οι μεγάλοι αυτοί χρόνοι χρησιμοποιείται ο μηχανισμός των timeout. Με αυτό τον μηχανισμό θέτουμε έναν χρόνο ο οποίος αν ξεπεραστεί ο πελάτης (client) σταματάει να αναμένει και στέλνει νέο αίτημα (request). Με αυτόν τον τρόπο καταφέρνουμε και αποφεύγουμε τους μεγάλους και απαγορευτικούς χρόνους αναμονής.

3.4.3 Υλοποίηση

Αρχικά γίνεται η παρατήρηση της ύπαρξης σφαλμάτων (errors) στην υπηρεσία books. Στην συνέχεια για να υπάρχει καλύτερη κατανόηση και συγκεκριμενοποίηση του προβλήματος δημιουργούμε service profiles για την κάθε υπηρεσία ξεχωριστά. Με τα service profiles έχουμε πληροφορία για το κάθε route της κάθε υπηρεσίας. Μέσω των Service profiles βλέπουμε ότι κάποια από τα αιτήματα από την υπηρεσία books στο route HEAD /authors/{id}.json της υπηρεσίας authors αποτυγχάνουν.

```
miltos@miltos-HP-Spectre-Notebook:~$ linkerd viz routes deploy/webapp --to deploy/books -n booksapp -o wide
ROUTE                     SERVICE   EFFECTIVE_SUCCESS   EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json     books             100.00%          0.6rps          100.00%       0.6rps           6ms          10ms          10ms
GET /books.json             books             100.00%          1.1rps          100.00%       1.1rps           8ms          18ms          20ms
GET /books/{id}.json        books             100.00%          2.4rps          100.00%       2.4rps           6ms          10ms          10ms
POST /books.json            books              46.98%          2.5rps           46.98%       2.5rps          14ms          19ms          20ms
PUT /books/{id}.json        books              49.32%          1.2rps           49.32%       1.2rps          70ms          97ms          99ms
[DEFAULT]                   books                   -               -                -            -             -             -             -

miltos@miltos-HP-Spectre-Notebook:~$ linkerd viz routes deploy/books --to deploy/authors -n booksapp -o wide
ROUTE                       SERVICE   EFFECTIVE_SUCCESS   EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /authors/{id}.json   authors                   -               -                -            -             -             -             -
GET /authors.json           authors                   -               -                -            -             -             -             -
GET /authors/{id}.json      authors                   -               -                -            -             -             -             -
HEAD /authors/{id}.json     authors              51.42%          3.5rps           51.42%       3.5rps           4ms           6ms           9ms
POST /authors.json          authors                   -               -                -            -             -             -             -
[DEFAULT]                   authors                   -               -                -            -             -             -             -

miltos@miltos-HP-Spectre-Notebook:~$
```

Εικόνα 3.4: Ποσοστά επιτυχίας πριν την ενεργοποίηση των retries και των timeouts

Για αυτό τον λόγο πειράζουμε το service profile της υπηρεσίας authors και κάνουμε το συγκεκριμένο route retryable.

```
Terminal                                   Οκτ 17 19:38
                                           miltos@miltos-HP-Spectre-Notebook: ~
miltos@miltos-HP-Spectre-Notebook:~$ linkerd viz routes deploy/books --to deploy/authors -n booksapp -o wide
ROUTE                       SERVICE   EFFECTIVE_SUCCESS   EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /authors/{id}.json   authors                   -               -                -            -             -             -             -
GET /authors.json           authors                   -               -                -            -             -             -             -
GET /authors/{id}.json      authors                   -               -                -            -             -             -             -
HEAD /authors/{id}.json     authors             100.00%          2.3rps           54.26%       4.3rps           6ms          20ms          28ms
POST /authors.json          authors                   -               -                -            -             -             -             -
[DEFAULT]                   authors                   -               -                -            -             -             -             -

miltos@miltos-HP-Spectre-Notebook:~$ linkerd viz routes deploy/webapp --to deploy/books -n booksapp -o wide
ROUTE                     SERVICE   EFFECTIVE_SUCCESS   EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json     books             100.00%          0.8rps          100.00%       0.8rps           7ms          10ms          10ms
GET /books.json             books             100.00%          1.6rps          100.00%       1.6rps           8ms          12ms          18ms
GET /books/{id}.json        books             100.00%          2.2rps          100.00%       2.2rps           5ms           9ms          10ms
POST /books.json            books             100.00%          1.5rps          100.00%       1.5rps          17ms          33ms          39ms
PUT /books/{id}.json        books             100.00%          0.8rps          100.00%       0.8rps          75ms          98ms         100ms
[DEFAULT]                   books                   -               -                -            -             -             -             -

miltos@miltos-HP-Spectre-Notebook:~$
```

Εικόνα 3.5: Ποσοστά επιτυχίας μετά τα retries και πριν τα timeouts

Μετά την αλλαγή παρατηρούμε ότι το Effective_Success στο συγκεκριμένο route γίνεται 100%. Επίσης 100% γίνεται και το Effective_Success και το Actual_Success στα routes POST /boks.json και PUT /books/{id}.json τα οποία πριν την αλλαγή ήταν κάτω από 100% (γιατί αυτά τα δύο routes εξαρτιόντουσαν το ένα από το άλλο). Επειδή το latency route PUT /books/{id}.json είναι μεγάλο, ορίζουμε σε αυτό ένα timeout μικρότερου χρόνου. Έτσι πετυχαίνουμε να μην περιμένουμε πολύ μεγάλους χρόνους.

```
                                           miltos@miltos-HP-Spectre-Notebook: ~
miltos@miltos-HP-Spectre-Notebook:~$ watch linkerd viz routes deploy/webapp --to deploy/books -n booksapp -o wide
miltos@miltos-HP-Spectre-Notebook:~$ linkerd viz routes deploy/webapp --to deploy/books -n booksapp -o wide
ROUTE                     SERVICE   EFFECTIVE_SUCCESS   EFFECTIVE_RPS   ACTUAL_SUCCESS   ACTUAL_RPS   LATENCY_P50   LATENCY_P95   LATENCY_P99
DELETE /books/{id}.json     books             100.00%          0.8rps          100.00%       0.8rps           8ms          10ms          10ms
GET /books.json             books             100.00%          1.6rps          100.00%       1.6rps           8ms          17ms          19ms
GET /books/{id}.json        books             100.00%          2.4rps          100.00%       2.4rps           6ms          10ms          10ms
POST /books.json            books             100.00%          1.6rps          100.00%       1.6rps          18ms          32ms          38ms
PUT /books/{id}.json        books              96.00%          0.8rps          100.00%       0.8rps          71ms          97ms          99ms
[DEFAULT]                   books                   -               -                -            -             -             -             -

miltos@miltos-HP-Spectre-Notebook:~$
```

Εικόνα 3. 6: Ποσοστά επιτυχίας μετά τα retries και μετά τα timeouts

Βλέπουμε όμως ότι το Effective_Success πέφτει κάτω από 100% ενώ το Actual_Success παραμένει 100%. Αυτό οφείλεται στο γεγονός ότι η πλευρά του πελάτη webapp στην συγκεκριμένη περίπτωση βλέπει αποτυχία επειδή λήγει το timeout ενώ ο εξυπηρετητής δηλαδή η υπηρεσία books στην συγκεκριμένη περίπτωση απαντάει σωστά (δεν επηρεάζεται από τα timeouts).

3.4.4 Επέκταση

Ο μηχανισμός των retries κρύβει τον κίνδυνο του retry storm [50] (μία υπηρεσία αποτυγχάνει λόγω των πολλών αιτημάτων και έχει ως αποτέλεσμα την αλυσιδωτή αποτυχία περισσότερων). Το linkerd για να αποφύγει αυτόν τον κίνδυνο έχει την παράμετρο του retry budget με την οποία μπορεί να οριστεί το πλήθος των retries σαν ποσοστό των συνολικών αιτημάτων συν έναν σταθερό αριθμό από retries/sec. Με αυτόν τον τρόπο διασφαλίζεται ότι το μέγιστο πλήθος από retries δεν θα ξεπεράσει ένα άνω όριο.

Κεφάλαιο 4 Συμπεράσματα

Συμπεράσματα και μελλοντικές επεκτάσεις

Βασικός στόχος της εργασίας υπήρξε η εξερεύνηση του χώρου του service mesh και η πρακτική εξοικείωση με την τεχνολογία αυτή μέσω εισαγωγικών (demo) εφαρμογών. Στήθηκε ένα Kubernetes cluster δύο κόμβων και πάνω του εγκαταστάθηκαν οι δύο εφαρμογές και το service mesh Linkerd. Μέσω αυτής της διάταξης διαπιστώθηκαν τα οφέλη του service mesh και πως αυτά μπορούν να αξιοποιηθούν συνδυαστικά με το Kubernetes (βλέπε αξιοποίηση της τηλεμετρίας και της συλλογής δεδομένων σε συνδυασμό με την αυτόματη κλιμάκωση του HPA). Επίσης είδαμε κάποια ιδιαίτερα χαρακτηριστικά του Linkerd όπως η ταχύτητα που έχει και την αμελητέα χρονική καθυστέρηση (latency) που προσδίδει (το συνολικό latency στην εφαρμογή emojivoto κυμαινόταν κοντά στα 10ms). Κατανοήθηκε πρακτικά η έννοια της διαφάνειας (transparency) και εμπεδώθηκε καλύτερα η διακριτότητα των ρόλων προγραμματιστής εφαρμογής (Developer) και μηχανικός DevOps.

Μελλοντικά θα μπορούσαν να υλοποιηθούν οι επεκτάσεις της κάθε υπηρεσίας. Θα μπορούσε για παράδειγμα να αξιοποιηθεί η συνάρτηση predict_linear() της PromQL η οποία χρησιμοποιεί γραμμική παλινδρόμηση (linear regression) για να προβλέπει εκ των προτέρων τις μελλοντικές τιμές της καθυστέρησης (latency). Με αυτόν τον τρόπο θα βρίσκονται έγκαιρα οι τιμές που ξεπερνούν το κατώφλι του HPA και η κλιμάκωση του συστήματος θα γίνεται αρκετά νωρίς μειώνοντας έτσι το συνολικό διάστημα υψηλών καθυστερήσεων. Επίσης στον διαμοιρασμό της δικτυακής κίνησης θα μπορούσε να χρησιμοποιηθεί το Flagger ώστε να επιτευχθεί ολοκληρωτική αυτοματοποίηση της εκτέλεσης (deployment) των καινούργιων εκδόσεων μίας μικρουπηρεσίας.

Πηγαίνοντας ακόμα ένα βήμα πιο πέρα θα μπορούσαν να υλοποιηθούν οι συγκεκριμένες (ή ισοδύναμες) υπηρεσίες με διαφορετικά service meshes όπως το Istio και το Consul για να γίνει σύγκριση του τρόπου υλοποίησης των υπηρεσιών αυτών από το κάθε service mesh αναδεικνύοντας τα πλεονεκτήματα και τα μειονεκτήματα του καθενός.

Βιβλιογραφία

[1] Len Bass, Ingo Weber, Liming Zhu "DevOps A Software Architect's Perspective" book

[2] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas , Santiago Gil, "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud" in 2015 10th Computing Colombian Conference (10CCC)

[3] James Lewis, and Martin Fowler, "Microservices: a definition of this new architectural term", https://martinfowler.com/articles/microservices.html, March 2014

[4] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, Stefan Tilkov, "Microservices: The Journey So Far and Challenges Ahead" in IEEE Software (Volume: 35, Issue: 3, May/June 2018)

[5] Konrad Gos, Wojciech Zabierowski, "The Comparison of MIcroservice and Monolithic Architecture" in 2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)

[6] Rahul Sharma, Avinash Singh, Getting Started with Istio Service Mesh (book)

[7] https://www.docker.com/resources/what-container/

[8] https://cloud.google.com/learn/what-are-containers

[9] https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#container

[10] https://www.ibm.com/cloud/learn/containers#toc-benefits-o-AcskyjM7

[11] https://www.vmware.com/topics/glossary/content/vms-vs-containers.html

[12] https://www.redhat.com/en/topics/containers/whats-a-linux-container

[13] Asif Khan, "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application" in IEEE Cloud Computing (Volume: 4, Issue: 5, September/October 2017)

[14] https://kubernetes.io/

[15] Abhisek Verma , Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Erik Tune, John Wilkes " Large-scale cluster management at Google with Borg" in Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France (2015)

[16] LinuxFoundationX's Introduction to Kubernetes and Cloud Native Technologies

[17] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[18] https://kubernetes.io/docs/tasks/extend-kubernetes/configure-aggregation-layer/

[19] Course: Introduction to Service Mesh with Linkerd (LFS143x)

[20] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, Yanbo Han, "Service Mesh: Challenges, State of the Art, and Future Research Opportunities" in 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)

[21] https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/?_gl=1*1x9wcmt*_ga*OTU5ODI5OTYzLjE2NDUxMDc0NDA.*_ga_TV358ZPK6D*MTY2NzMzMjU0MS42LjEuMTY2NzMzMjYxOS4wLjAuMA..#_ga=2.163085050.77300688.1667320504-959829963.1645107440

[22] https://linkerd.io/2.12/reference/architecture/

[23] https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

[24] https://istio.io/latest/about/service-mesh/

[25] https://www.envoyproxy.io/docs/envoy/v1.24.0/intro/what_is_envoy

[26] https://github.com/aeraki-mesh/aeraki

[27] https://merbridge.io/docs/overview/

[28] https://ebpf.io/

[29] https://servicemesh.es/

[30] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, Moonki Hong "NetMARKS: Network Metrics-AwaRe Kubernetes Scheduler Powered by Service Mesh" in IEEE INFOCOM 2021 - IEEE Conference on Computer Communications

[31] Lianjie Cao, Puneet Sharma "Co-locating Containerized Workload Using Service Mesh Telemetry" in Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies

[32] Toru Furusawa, Hiroshi Abe, Kazuya Okada, Akihiro Nakao, "Service Mesh Controller for Cooperative Load Balancing among Neighboring Edge Servers" in 2022 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)

[33] Forough Shahab Samani, Rolf Stadler, "Dynamically meeting performance objectives for multiple services on a service mesh" in 18th International Conference on Network and Service Management

[34] Mohammad Reza Saleh Sedghpour, Paul Townend "Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions" in 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)

[35] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, Ratul Mahajan "Dissecting Service Mesh Overheads" Cornell University

[36] Mohammad Reza Saleh Sedghpour, Cristian Klein, Johan Tordsson "Service mesh circuit breaker: From panic button to performance management tool" in Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems

[37] Rupesh Raj Karn, Jukka Heikkonen, Rammi Das, Rajeev Kanth, Dibakar Raj Pan "Automated Testing and Resilience of Microservice's Network-link using Istio Service Mesh" in Proceeding of the 31st conference of fruct association

[38] Rami Alboqmi, Sharmin Jahan, Rose F. Gamble "Toward Enabling Self-Protection in the Service Mesh of the Microservice Architecture" in 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)

[39] Moonjoong Kang, Jun-Sik Shin, JongWon Kim "Protected Coordination of Service Mesh for Container-based 3-tier Service Traffic" in 2019 International Conference on Information Networking (ICOIN)

[40] Constantin Adam, Abdulhamid Adebayo, Hubertus Franke, Edward Snible, Tobin Feldman-Fitzthum, James Cadden, Nerla Jean-Louis "Partially Trusting the Service Mesh Control Plane" Cornell University

[41] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[42] https://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html

[43] https://kubernetes.io/docs/tasks/extend-kubernetes/configure-aggregation-layer/

[44] https://github.com/grampelberg/talks/blob/master/kubecon-12-2018/slides.pdf

[45] https://linkerd.io/2.11/features/traffic-split/

[46] https://linkerd.io/2.11/tasks/linkerd-smi/

[47] https://flagger.app/

[48] https://linkerd.io/2.11/tasks/canary-release/

[49] https://linkerd.io/2.11/tasks/books/#retries

[50] https://linkerd.io/2.11/features/retries-and-timeouts/

[51] https://developer.hashicorp.com/consul/docs/architecture

[52] https://developer.hashicorp.com/consul/docs/architecture/anti-entropy

[53] https://kuma.io/docs/2.0.x/explore/overview/

[54] https://doc.traefik.io/traefik-mesh/

[55] Marius Jøsok Nesset, Supervisor: Erik Hjelmås, "Evaluating Performance And Security Characteristics Of Service Mesh Technologies In A Rancher 2.X Environment", Bachelor's project in IT-Operations and Information Security, May 2021

[56] https://github.com/BuoyantIO/emojivoto

[57] https://release-v1-2.docs.openservicemesh.io/docs/overview/osm_components/

[58] https://linkerd.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

[59] Oksana Baranova, "Multi-Tenant Isolation in a Service Mesh", Thesis in Master's Programme in Computer, Communication and Information Sciences Aalto University

[60] L. Larsson, W. Tarneberg, C. Klein, E. Elmroth, and M. Kihl, "Impact of etcd deployment on Kubernetes, Istio, and application performance," Software: Practice and Experience, 2020

[61] M. Ganguli, S. Ranganath, S. Ravisundar, A. Layek, D. Ilangovan, and E. Verplanke, "Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge," in 2021 IEEE International Conference on Edge Computing (EDGE)

[62] Z. Sun, "Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment," 2019

[63] X. He and F. Deng, "Research on architecture of internet of things platform based on service mesh," in 2020 12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), 2020

[64] A. O. Duque, C. Klein, J. Feng, X. Cai, B. Skubic, and E. Elmroth, "A Qualitative Evaluation of Service Mesh-based Traffic Management for Mobile Edge Cloud," arXiv preprint arXiv:2205.06057, 2022