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

Μελέτη και Ανάπτυξη Μηχανισμών Ενορχήστρωσης Εφαρμογών σε Ετερογενείς Υπολογιστικές Υποδομές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθηνά Σ. Κυριάκου

Επιβλέπων : Εμμανουήλ Βαρβαρίγος
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

Η σελίδα αυτή είναι σκόπιμα λευκή.

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

Μελέτη και Ανάπτυξη Μηχανισμών Ενορχήστρωσης Εφαρμογών σε Ετερογενείς Υπολογιστικές Υποδομές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Αθηνά Σ. Κυριάκου

Επιβλέπων : Εμμανουήλ Βαρβαρίγος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 10η Νοεμβρίου 2021.

.....	.....	.....
Εμμανουήλ Βαρβαρίγος	Θεοδώρα Βαρβαρίγου	Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

.....
Αθηνά Σ. Κυριάκου

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

Περίληψη

Λόγω των διαρκώς αυξανόμενων απαιτήσεών τους, οι σύγχρονες εφαρμογές λογισμικού αναπτύσσονται ως ομάδες χαλαρά συζευγμένων υπηρεσιών και για την εκτέλεσή τους χρησιμοποιούνται υπολογιστικοί πόροι διαφορετικών χαρακτηριστικών και δυνατοτήτων. Προκειμένου να εξυπηρετηθούν οι ανάγκες των εφαρμογών, οι πόροι τοποθετούνται σε διάφορα σημεία του συστήματος, σε διαφορετική εγγύτητα προς τους χρήστες, όπως σε υποδομές υπολογιστικού νέφους ή και σε συσκευές περιορισμένης υπολογιστικής ισχύος στα άκρα του δικτύου. Σε αυτό το αποτελούμενο από πολλαπλές ετερογενείς υποδομές περιβάλλον, η αποδοτική κατανομή των πόρων στις εφαρμογές και η ενορχήστρωση της εκτέλεσής τους, σύμφωνα με τις απαιτήσεις των χρηστών, την κατάσταση των υποδομών και πολλαπλά κριτήρια βελτιστοποίησης, συνιστούν μια σύνθετη διαδικασία λήψης αποφάσεων. Ύστερα από τη μελέτη διαδεδομένων εργαλείων ανοιχτού κώδικα για την ενορχήστρωση των εφαρμογών μεμονωμένα σε κάθε υποδομή, σχεδιάστηκε ένα κατανεμημένο σύστημα με μηχανισμούς ενορχήστρωσης σε δύο επίπεδα για τη διαχείριση των εφαρμογών συνολικά στους ετερογενείς πόρους, ανεξάρτητα από τις επιμέρους ιδιαιτερότητες που αυτοί παρουσιάζουν. Βάσει της υλοποίησης μιας απλουστευμένης εκδοχής του συστήματος, αξιολογήθηκε η δυνατότητα υλοποίησης της προτεινόμενης λύσης στην πράξη για τη διαφανή ενορχήστρωση εφαρμογών σε ετερογενείς υπολογιστικούς πόρους.

Λέξεις Κλειδιά

εγγενής εφαρμογή υπολογιστικού νέφους, υποδομές υπολογιστικού νέφους, πόροι στα άκρα του δικτύου, τεχνολογίες ενορχήστρωσης containers, ετερογενείς πόροι

Abstract

Due to their ever-increasing requirements, software applications nowadays are developed as groups of loosely coupled services and numerous computing resources, usually with different characteristics and capabilities, are used for their execution. In order to fulfil the applications' needs, computing resources are placed in varying proximity to the end users, such as in cloud infrastructures or embedded in edge devices. In such heterogeneous environments, the efficient allocation of resources and the orchestration of submitted applications throughout their lifecycle, according to users' specifications, the system's state and multiple optimization criteria, constitute a complex decision-making process. After examining existing open-source tools for the orchestration of applications in each infrastructure independently, a distributed system with orchestration mechanisms in two levels is designed, aiming to manage the submitted applications across all heterogeneous infrastructures, regardless of their varying features. Based on a simplified implementation of this system, the practical feasibility of the proposed solution is evaluated, towards the transparent deployment and orchestration of applications in systems with heterogenous computing resources.

Keywords

cloud-native application, cloud computing, edge computing, container orchestration, heterogeneous resources

Ευχαριστίες

Με την ολοκλήρωση αυτής της εργασίας ολοκληρώνεται και ένα σημαντικό κεφάλαιο στην ακαδημαϊκή και προσωπική μου πορεία, αυτό των προπτυχιακών σπουδών. Έχοντας ήδη αρχίσει να ξεχνάω τις δυσκολίες που παρουσιάστηκαν, χρειάζεται να αναφερθώ εδώ στους ανθρώπους που ήταν κοντά μου τα τελευταία έξι χρόνια, μετατρέποντας κάθε πρόκληση σε εμπειρία που θα αναπολώ για πολύ καιρό ακόμα.

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα της διπλωματικής μου και καθηγητή στη ΣΗΜΜΥ κ. Μάνο Βαρβαρίγο για όλες τις δυνατότητες για έρευνα που μου δόθηκαν στο εργαστήριο Επικοινωνιακών Δικτύων Υψηλής Ταχύτητας, πέρα από τις ανάγκες μιας εργασίας. Νιώθω πολύ τυχερή για τις εμπειρίες που αποκόμισα από τον ακαδημαϊκό ερευνητικό χώρο όντας ακόμα φοιτήτρια. Ευχαριστώ επίσης τον κ. Αριστοτέλη Κρέτση και τον κ. Παναγιώτη Κόκκινο για την καθοδήγηση και τη συνεχή υποστήριξή τους όλους τους προηγούμενους μήνες.

Τίποτα δε θα ήταν το ίδιο ωστόσο χωρίς ανθρώπους όπως η Αλίκη, η Θεοδώρα, η Φωτεινή, η Αντωνία και η Νεφέλη, ο Αλέξανδρος, ο Ορέστης, ο Γιώργος, ο Πάνος, η Νίκη, η Ξανθή, η Κύρκη και ο Taiyo, οι οποίοι ανελλιπώς ήταν δίπλα μου, παρά τη γεωγραφική απόσταση που ενίοτε υπήρχε ανάμεσά μας. Όπου και αν βρεθούμε τα επόμενα χρόνια, θα είμαι μία κλήση μακριά να συνεχίσουμε να μοιραζόμαστε στιγμές.

Και τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου, ειδικά τους γονείς μου και τον αδερφό μου Γιώργο. Σας ευχαριστώ που πάντοτε με στηρίζετε, ό,τι και αν βάλω στο μυαλό μου.

Περιεχόμενα

Κεφάλαιο 1: Εισαγωγή

Οι διαρκώς αυξανόμενες λειτουργικές και μη λειτουργικές απαιτήσεις των σύγχρονων εφαρμογών λογισμικού έχουν συμβάλει στη σταδιακή μετάβαση από ιεραρχικές αρχιτεκτονικές κεντρικού ελέγχου μεμονωμένων και πολύ μεγάλων υποδομών σε ομάδες χαλαρά συζευγμένων (loosely coupled) κατανεμημένων συστημάτων. Σε αυτό το ετερογενές περιβάλλον, οι υπολογιστικοί, αποθηκευτικοί και δικτυακοί πόροι τοποθετούνται σε διάφορα σημεία της συνολικής υποδομής, σε διαφορετική εγγύτητα προς τους χρήστες για την εξυπηρέτηση διαφορετικών αναγκών των εφαρμογών. Διαθέσιμοι πόροι μπορούν να βρίσκονται σε υποδομές υπολογιστικού νέφους (cloud computing) για την εξυπηρέτηση εφαρμογών με αυξημένες απαιτήσεις σε υπολογιστική ισχύ και χωρητικότητα και ανάγκες κλιμάκωσης. Ταυτόχρονα, κόμβοι υπολογισμού μπορούν να είναι ενσωματωμένοι σε συσκευές περιορισμένης υπολογιστικής ισχύος στα άκρα του δικτύου (edge computing) για την άμεση επεξεργασία των συλλεγόμενων δεδομένων και την ικανοποίηση των αιτημάτων που απαιτούν μικρούς χρόνους απόκρισης. Παράλληλα, για την εκτέλεση εργασιών πολύ μεγάλου όγκου διαχειριζόμενων δεδομένων και υψηλού υπολογιστικού φορτίου μπορούν να χρησιμοποιούνται κεντρικές υποδομές υπέρ-υψηλών επιδόσεων (high performance computing).

Στο εισαγωγικό κεφάλαιο περιγράφονται τα δύο πιο σύγχρονα μοντέλα χρήσης πόρων που αξιοποιούνται σε αυτό το ετερογενές περιβάλλον· αρχικά η χρήση πόρων που βρίσκονται σε κεντρικές υποδομές υπολογιστικού νέφους και στη συνέχεια σε συσκευές περιορισμένης υπολογιστικής ισχύος στα άκρα του δικτύου. Ιδιαίτερη αναφορά γίνεται επίσης στην αρχιτεκτονική των εκτελέσιμων εφαρμογών. Τέλος, παρουσιάζονται το αντικείμενο και η συνεισφορά της διπλωματικής εργασίας στη δημιουργία μιας πλατφόρμας που θα διευκολύνει την ανάπτυξη, εκτέλεση και ενορχήστρωση και παρακολούθηση εφαρμογών σε αυτά τα σύνθετα περιβάλλοντα υπολογισμού, ανεξάρτητα από τις ιδιαιτερότητες των επιμέρους συστημάτων.

1.1 Μοντέλα Χρήσης Πόρων

1.1.1 Υπηρεσίες Υπολογιστικού Νέφους

Το μοντέλο υπολογιστικού νέφους περιγράφει τη διάθεση πόρων και υπηρεσιών κατ' αίτημα των χρηστών (on-demand) μέσω διαδικτύου. Οι διατιθέμενοι πόροι μπορεί να είναι υπολογιστικές, αποθηκευτικές ή και δικτυακές μονάδες οι οποίες βρίσκονται σε απομακρυσμένες κεντρικές υποδομές (data centres). Εξωτερικοί πάροχοι είναι υπεύθυνοι για τη διαχείριση των πόρων και υπηρεσιών, περιορίζοντας το κόστος και την πολυπλοκότητα διαχείρισης συγκριτικά με τις ιδιόκτητες από τους χρήστες υποδομές και πλατφόρμες (on-premise). Η πρόσβαση στους πόρους

και στις υπηρεσίες πραγματοποιείται συνήθως με χρηματικό αντίτιμο, το οποίο ορίζεται σε μηνιαία βάση ή εξαρτάται από τον βαθμό χρησιμοποίησής τους.

Η διάθεση των πόρων από τους παρόχους επιτυγχάνεται συνήθως μέσω του μηχανισμού της εικονικοποίησης (virtualization). Συνοπτικά, αποκρύπτοντας λεπτομέρειες υλοποίησης και λειτουργίας, η εικονικοποίηση επιτρέπει την εικονική διαίρεση των πόρων υλικού (π.χ. εικονικοποίηση επεξεργαστή, μνήμης, δίσκου), λογισμικού (π.χ. λειτουργικού συστήματος, εφαρμογών) ή και των δικτυακών υποδομών ενός μηχανήματος. Ο διαμοιρασμός αυτός επιτυγχάνεται με τη χρήση ειδικού λογισμικού, του υπέρ επόπτη (hypervisor). Κατά αυτόν τον τρόπο, η εικονικοποίηση συμβάλλει στην ευκολότερη και αποδοτικότερη διαχείριση των πόρων, στον περιορισμό των διακοπών λειτουργίας τους (downtime) και στην ταχύτερη διάθεσή τους από τους παρόχους

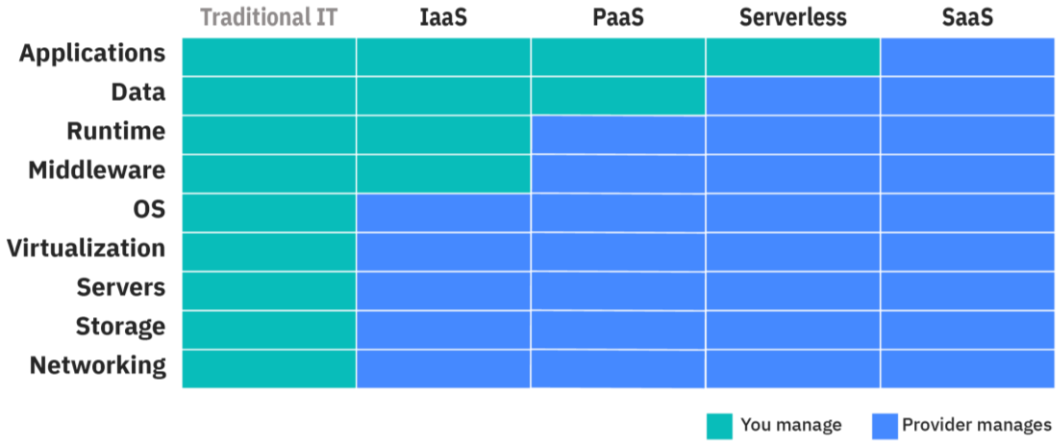
Η πιο συνήθης μορφή εικονικοποίησης πραγματοποιείται σε επίπεδο διακομιστή (server virtualization). Σε αυτόν τον τύπο εικονικοποίησης, οι φυσικοί πόροι ενός υπολογιστικού μηχανήματος διαιρούνται σε πολλαπλά εικονικά μηχανήματα (virtual machines), καθένα από τα οποία αποτελείται συνήθως από ένα εξωτερικό λειτουργικό σύστημα (guest operating system) και ένα εικονικό τμήμα του διαθέσιμου υλικού. Έτσι, κάθε εικονικό μηχάνημα προσομοιώνει μία υπολογιστική μονάδα [1].

Ανάλογα με τους παρεχόμενους πόρους υλικού και λογισμικού και τον βαθμό ελέγχου της λειτουργίας τους από τους χρήστες, το μοντέλο υπολογιστικού νέφους διακρίνεται σε επιμέρους κατηγορίες. Αρχικά, το μοντέλο της «Υποδομής ως Υπηρεσία» (Infrastructure-as-a-Service, IaaS) παρέχει κατ' αίτημα πρόσβαση σε θεμελιώδεις πόρους υλικού, όπως διακομιστές (servers), αποθηκευτικές μονάδες και υποδομές δικτύου. Ένα άλλο μοντέλο είναι αυτό της «Πλατφόρμας ως Υπηρεσία» (Platform-as-a-Service, PaaS), σύμφωνα με το οποίο οι πάροχοι διαθέτουν πλατφόρμες υλικού και λογισμικού για την εκτέλεση εφαρμογών. Οι πάροχοι συντηρούν και διαχειρίζονται τους πόρους υλικού (π.χ. διακομιστές, αποθηκευτικές μονάδες, δικτυακές υποδομές) και λογισμικού (π.χ. λειτουργικά συστήματα, βάσεις δεδομένων) για τη λειτουργία των πλατφορμών. Όπως φαίνεται στην Εικόνα 1.1, οι χρήστες είναι υπεύθυνοι σε μεγαλύτερο βαθμό για τη διαχείριση και τη λειτουργία των πόρων στις υπηρεσίες IaaS και ακολούθως στις PaaS, συγκριτικά με τις υπόλοιπες κατηγορίες υπηρεσιών του μοντέλου υπολογιστικού νέφους.

Συνεχίζοντας, με τις serverless υπηρεσίες οι χρήστες αποκτούν πρόσβαση σε πόρους μόνο κατά την εκτέλεση των εφαρμογών τους. Οι πάροχοι είναι υπεύθυνοι για όλες τις διαδικασίες διαχείρισης των υποδομών, όπως προμήθεια και συντήρηση, ανάθεση των εργασιών σε πόρους για εκτέλεση και αυτόματη οριζόντια κλιμάκωση των πόρων αν απαιτείται. Οι εφαρμογές εκτελούνται στις υποδομές κατ' αίτημα των χρηστών και έτσι η κοστολόγηση είναι ανάλογη των πόρων που χρησιμοποιούνται αποκλειστικά στον χρόνο εκτέλεσης. Μια διαδεδομένη υποκατηγορία serverless είναι το μοντέλο

της «Συνάρτησης ως Υπηρεσία» (Function-as-a-Service, FaaS), σύμφωνα με το οποίο οι χρήστες μπορούν να εκτελέσουν τμήματα του κώδικα μιας εφαρμογής, τις συναρτήσεις, με γνώμονα συγκεκριμένα συμβάντα.

Τέλος, ένα ακόμα μοντέλο υπηρεσιών υπολογιστικού νέφους είναι αυτό του «Λογισμικού ως Υπηρεσία» (Software-as-a-Service, SaaS). Στις υπηρεσίες αυτές παρέχονται στους χρήστες προϊόντα λογισμικού. Παραδείγματα τέτοιων προϊόντων είναι εφαρμογές για αποθήκευση δεδομένων σε υποδομές υπολογιστικού νέφους, εφαρμογές επικοινωνίας κλπ. Η πρόσβαση στις υπηρεσίες αυτές γίνεται μέσω του φυλλομετρητή (web browser), μίας εφαρμογής ή μιας διεπαφής προγραμματισμού εφαρμογών (Application Programming Interface, API).



Εικόνα 1.1: Βαθμός διαχείρισης των διαθέσιμων πόρων και υπηρεσιών από τους χρήστες και τους παρόχους στις δικτυακές υπηρεσίες υπολογιστικού νέφους

Τα περιβάλλοντα υπολογιστικού νέφους διακρίνονται επίσης σε δημόσια και ιδιωτικά ανάλογα με τον βαθμό στον οποίο οι χρήστες επιλέγουν να είναι προσβάσιμοι οι πόροι που χρησιμοποιούν, άμεσα ή έμμεσα, από άλλους. Από τη μία πλευρά, στις υπηρεσίες δημόσιου υπολογιστικού νέφους οι υποδομές των παρόχων είναι προσβάσιμες από πολλαπλές διαφορετικές ομάδες χρηστών (multi-tenant environment). Από την άλλη, στο ιδιωτικό μοντέλο υπολογιστικού νέφους, μόνο εξουσιοδοτημένοι χρήστες έχουν πρόσβαση στις παρεχόμενες υποδομές και υπολογιστικούς πόρους. Ωστόσο, ένας οργανισμός μπορεί να επιλέξει να έχει πρόσβαση τόσο σε δημόσιους όσο και ιδιωτικούς πόρους, δημιουργώντας υβριδικά περιβάλλοντα υπολογιστικού νέφους (hybrid cloud). Για ακόμα μεγαλύτερη ευελιξία και πρόσβαση σε μεγαλύτερο εύρος υπηρεσιών, συχνά χρησιμοποιούνται ταυτόχρονα υποδομές διαφορετικών παρόχων (multi cloud), αυξάνοντας ωστόσο την πολυπλοκότητα διαχείρισης του συνολικού συστήματος.

Το μοντέλο υπολογιστικού νέφους παρουσιάζει πολλαπλά οφέλη για έναν οργανισμό, συγκριτικά με τη χρήση ιδιόκτητων πόρων. Μεταξύ άλλων, ανάλογα με το μέγεθος του οργανισμού, περιορίζονται συνήθως τα λειτουργικά κόστη και ο χρόνος που απαιτείται για την εγκατάσταση, διαχείριση και συντήρηση των υποδομών. Οι υπηρεσίες υπολογιστικού νέφους ενισχύουν επίσης την ευελιξία και

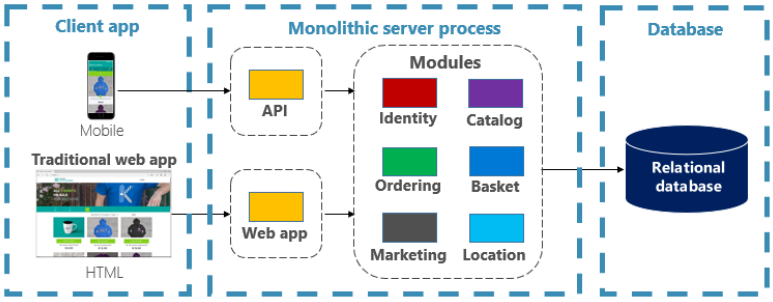
την παραγωγικότητα των ομάδων ανάπτυξης, δίνοντάς τους άμεση πρόσβαση σε μεγάλο εύρος υπηρεσιών. Ακόμη, διευκολύνεται η κλιμάκωση των χρησιμοποιούμενων πόρων, ανάλογα με τις ανάγκες των εφαρμογών, και η απρόσκοπτη λειτουργία τους σε παγκόσμιο επίπεδο [2], [3].

Ωστόσο, η άμεση ή έμμεση χρήση πόρων τη διαχείριση των οποίων αναλαμβάνουν τρίτοι, παρουσιάζει εγγενείς περιορισμούς που χρειάζεται να ληφθούν υπόψη κατά τον σχεδιασμό συστημάτων σε περιβάλλοντα υπολογιστικού νέφους. Αρχικά, οι χρήστες μπορεί να αντιμετωπίζουν δυσκολίες κατά την ανάπτυξη των εφαρμογών σε πόρους που δε διαχειρίζονται αποκλειστικά οι ίδιοι. Επίσης, η ανομοιογένεια στον σχεδιασμό, στις παρεχόμενες λειτουργίες και διεπαφές και στις συμβάσεις χρήσης των υποδομών διαφορετικών παρόχων, δυσχεραίνει την ενσωμάτωση στα συστήματα των χρηστών εξωτερικών υπηρεσιών και την μετεγκατάσταση των εφαρμογών τους από την πλατφόρμα ενός παρόχου σε μια άλλη. Ταυτόχρονα, δεδομένου ότι οι υπηρεσίες του μοντέλου υπολογιστικού νέφους είναι προσβάσιμες μέσω διαδικτύου, οι χρήστες χρειάζεται να λαμβάνουν υπόψη στον σχεδιασμό των συστημάτων τους πιθανές διακοπές λειτουργίας, όπως και ζητήματα ασφαλείας ειδικά κατά τη διαχείριση ευαίσθητων δεδομένων [4].

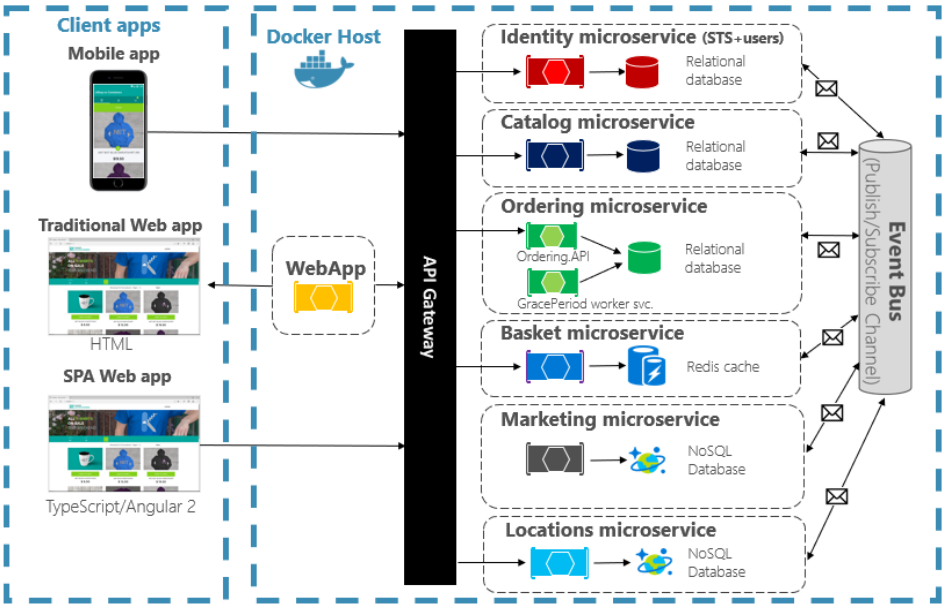
Ιδιαίτερη αναφορά χρειάζεται να γίνει ακόμη στην αρχιτεκτονική των εφαρμογών που εκτελούνται σε περιβάλλοντα υπολογιστικού νέφους. Μία εγγενής εφαρμογή υπολογιστικού νέφους (cloud native application) αποτελείται από πολλές επιμέρους υπηρεσίες, τις μικροϋπηρεσίες (microservices), οι οποίες συνδέονται χαλαρά μεταξύ τους. Κάθε μικροϋπηρεσία υλοποιεί μία συγκεκριμένη λειτουργία της εφαρμογής και έτσι αναπτύσσεται, εκτελείται, αναβαθμίζεται και κλιμακώνεται ανεξάρτητα από τις υπόλοιπες. Επίσης, οι μικροϋπηρεσίες των εγγενών εφαρμογών υπολογιστικού νέφους εκτελούνται σε containers, για τη διαχείριση των οποίων και τη βέλτιστη αξιοποίηση των διαθέσιμων πόρων υλικού και λογισμικού είναι υπεύθυνα προηγμένα εργαλεία ενορχήστρωσης. Όπως γίνεται κατανοητό, η αρχιτεκτονική αυτή σχεδιασμού των εφαρμογών ως σύνολα από microservices διευκολύνει την ανάπτυξη, τον έλεγχο και την ευρεία κλιμάκωσή τους [5], [6].

Συγκρίνοντας την υλοποίηση μιας μονολιθικής εφαρμογής ιστού (web application) (Εικόνα 1.2) με αυτή βάσει του μοντέλου των μικροϋπηρεσιών (Εικόνα 1.3) παρατηρούνται τα παρακάτω. Στη δεύτερη περίπτωση κάθε μικροϋπηρεσία χρησιμοποιεί τις κατάλληλες για τη λειτουργία της τεχνολογίες και το δικό της μοντέλο δεδομένων. Οι μικροϋπηρεσίες επικοινωνούν μεταξύ τους μέσω διεπαφών προγραμματισμού εφαρμογών βασισμένων σε περιγραφή πόρων (RESTful, REST APIs) και ροών συμβάντων (event streaming) [7]. Συνήθεις εφαρμογές υπολογιστικού νέφους είναι αυτές με μεγάλες απαιτήσεις σε υπολογιστική ισχύ και αποθηκευτικές μονάδες, πέρα συνήθως από τους διαθέσιμους πόρους σε έναν οργανισμό, όπως επεξεργασία και ανάλυση μεγάλου όγκου δεδομένων (big data analytics), εφαρμογές του Διαδικτύου των Πραγμάτων (Internet of Things, IoT) και εφαρμογές μηχανικής (machine learning) και βαθιάς μηχανικής μάθησης (deep learning) [2].

Πληθώρα τεχνολογιών έχει δημιουργηθεί για την ανάπτυξη και διαχείριση των εγγενών εφαρμογών υπολογιστικού νέφους. Για την καλύτερη κατανόηση των δυνατοτήτων των τεχνολογιών αυτών, έχει προταθεί από τον οργανισμό Cloud Native Computing Foundation (CNCF) η ομαδοποίηση των τεχνολογιών στο CNCF Cloud Native Landscape[1], ανάλογα με τη λειτουργία τους. Όπως φαίνεται στην Εικόνα 1.4, η ομαδοποίηση αποτελείται από τέσσερα οριζόντια και δύο κατακόρυφα επίπεδα. Τα τέσσερα οριζόντια επίπεδα ομαδοποιούν τις τεχνολογίες για την εγκατάσταση και διαχείριση των υποδομών που απαιτούνται για την εκτέλεση των εφαρμογών (Provisioning), τις τεχνολογίες για την εκτέλεση (Runtime) και την ενορχήστρωση και διαχείρισή τους (Orchestration & Management) και τέλος τις τεχνολογίες για τον ορισμό και την ανάπτυξη (Application Definition & Development). Τα κατακόρυφα επίπεδα αποτελούνται από τεχνολογίες για την παρακολούθηση της λειτουργίας του συστήματος (Observability & Analysis) και πλατφόρμες με υπηρεσίες για τον συντονισμό των διαφορετικών τεχνολογιών στα οριζόντια επίπεδα (Platforms). Τα κατακόρυφα επίπεδα συμπληρώνουν τις λειτουργίες των τεχνολογιών των οριζοντίων επιπέδων σε ένα σύστημα υπολογιστικού νέφους [8].

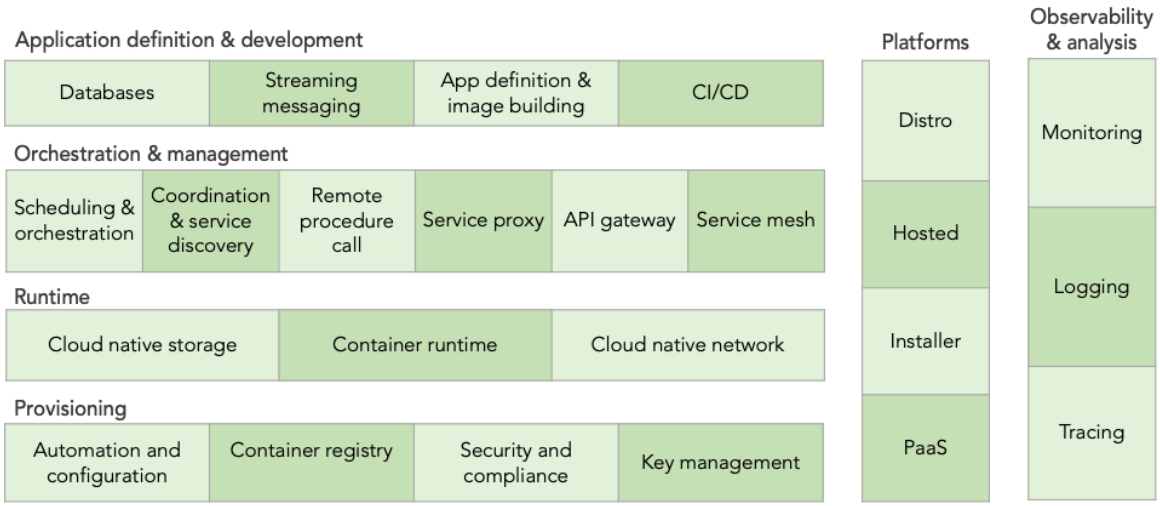


Εικόνα 1.2: Μονολιθική υλοποίηση εφαρμογής ιστού



Εικόνα 1.3: Υλοποίηση εφαρμογής ιστού με μικροϋπηρεσίες

[1] https://landscape.cncf.io/



Εικόνα 1.4: Το CNCF Cloud Native Landscape

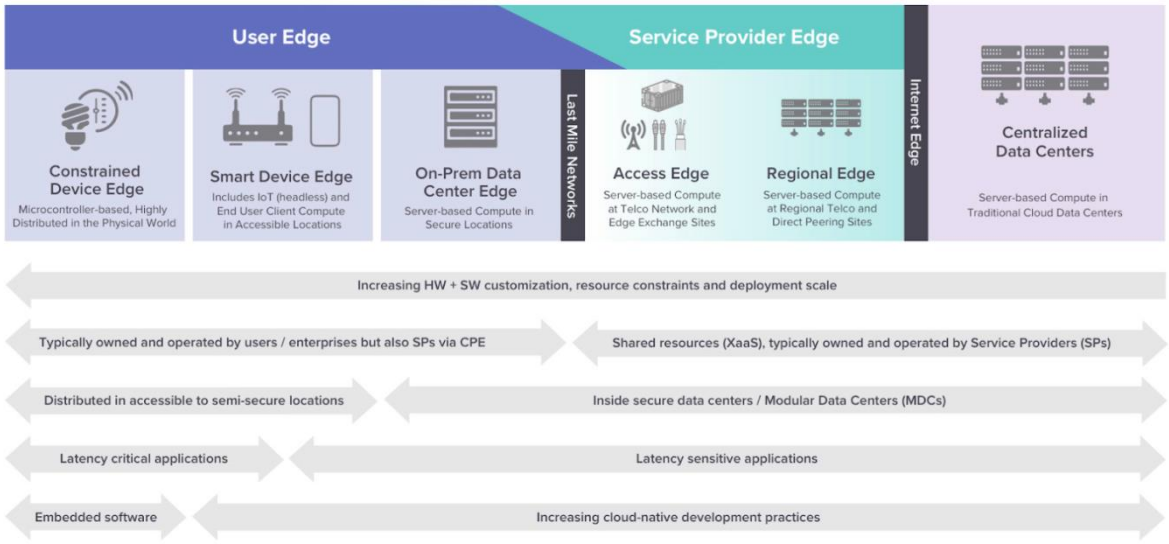
1.1.2 Υπηρεσίες Διάθεσης Πόρων στα Άκρα του Δικτύου

Οι υπηρεσίες διάθεσης πόρων στα άκρα του δικτύου (edge computing) αφορούν ένα κατανεμημένο μοντέλο χρήσης υπολογιστικών πόρων το οποίο αναπτύχθηκε στην προσπάθεια καλύτερης διαχείρισης του μεγάλου όγκου παραγόμενων δεδομένων από συσκευές IoT. Σύμφωνα με το μοντέλο, η επεξεργασία και η αποθήκευση των δεδομένων μιας εφαρμογής πραγματοποιούνται κυρίως στα άκρα του δικτύου, όσο πιο κοντά γίνεται στο γεωγραφικό σημείο δημιουργίας των δεδομένων και χρήσης των συμπερασμάτων που εξάγονται από την ανάλυσή τους. Στην πράξη, οι υπολογιστικοί και αποθηκευτικοί πόροι στις τοπολογίες edge computing βρίσκονται τόσο σε συσκευές περιορισμένης υπολογιστικής ισχύος στα άκρα του δικτύου, όσο και σε υποδομές υπολογιστικού νέφους και σε ιδιόκτητα data centers, η σύνδεση μεταξύ των οποίων γίνεται μέσω τοπικών δικτύων και του διαδικτύου.

Η Εικόνα 1.5 απεικονίζει την κατανομή των φυσικών πόρων σε μία τοπολογία edge computing, όπως έχει προταθεί από το μοντέλο LF Edge του οργανισμού Linux Foundation [9]. Το μοντέλο LF Edge κατανείμει τις υπολογιστικές μονάδες σε διάφορα σημεία ανάμεσα στις συσκευές στα άκρα του δικτύου και στις κεντρικές υποδομές ή στις υποδομές υπολογιστικού νέφους.

Όπως φαίνεται στην Εικόνα 1.5, το μοντέλο LF Edge περιγράφει δύο επίπεδα υποδομών edge computing τα οποία συνδέονται με πολύ απομακρυσμένα δίκτυα (last mile networks). Στη μία πλευρά του απομακρυσμένου δικτύου, πιο κοντά στο γεωγραφικό σημείο συλλογής δεδομένων, ορίζεται το Επίπεδο Χρήστη (User Edge). Το επίπεδο αυτό περιλαμβάνει δισεκατομμύρια ετερογενείς και εξειδικευμένες ως προς τη λειτουργία συσκευές περιορισμένων υπολογιστικών πόρων, όπως αισθητήρες και ελεγκτές, οι οποίες τοποθετούνται κοντά στους τελικούς χρήστες για τη συλλογή δεδομένων. Επίσης, στο Επίπεδο Χρήστη υπάρχουν ευφυείς πύλες δικτύου (edge gateways), όπως συσσωρευτές IoT δεδομένων και συσκευές μεταγωγής και ανάθεσης των εργασιών

σε πόρους για εκτέλεση, οι οποίες μπορούν να αποθηκεύουν μέρος των δεδομένων και πραγματοποιούν περιορισμένες αναλύσεις σε πραγματικό χρόνο. Οι πύλες συνδέονται μέσω τοπικών δικτύων σε ιδιόκτητες πλατφόρμες διακομιστών (on-premise server platforms) σε ασφαλείς τοποθεσίες. Οι πλατφόρμες διακομιστών εκτελούν τους απαιτούμενους από τον διακομιστή και είναι υπεύθυνοι για την εκτέλεση και διαχείριση των εφαρμογών στις συσκευές στα άκρα του δικτύου.



Εικόνα 1.5: Το μοντέλο LF Edge για τοπολογίες edge computing

Στην άλλη πλευρά του απομακρυσμένου δικτύου, ορίζεται σύμφωνα με το μοντέλο LF Edge το Επίπεδο του Παρόχου Υπηρεσιών (Service Provider Edge). Το επίπεδο αυτό συντονίζει τους υπολογισμούς στους διακομιστές του τηλεπικοινωνιακού δικτύου (telco network) και στα σημεία ανταλλαγής στα άκρα του δικτύου (edge exchange sites) με αυτούς στο περιφερειακό δίκτυο (regional network). Το Επίπεδο του Παρόχου Υπηρεσιών λειτουργεί ως μια τοπική υποδομή υπολογιστικού νέφους. Πιο συγκεκριμένα, χρησιμοποιείται από τις συσκευές του Επιπέδου Χρήστη για ανταλλαγή μηνυμάτων, για πιο υπολογιστικά απαιτητικές εργασίες και για την αποθήκευση μεγαλύτερου όγκου δεδομένων. Οι παρεχόμενες στους χρήστες υπηρεσίες σε αυτό το επίπεδο είναι επίσης πιο ασφαλείς, συγκριτικά με τις δημόσιες υπηρεσίες υπολογιστικού νέφους, καθώς η ανταλλαγή δεδομένων γίνεται μέσω ιδιωτικών τοπικών δικτύων.

Τα Επίπεδα Χρήστη και Παρόχου Υπηρεσιών στα άκρα του δικτύου συνδέονται με τις κεντρικές υποδομές υπολογισμού και αποθήκευσης μέσω διαδικτύου. Οι κεντρικές αυτές υποδομές μπορεί να βρίσκονται είτε σε υποδομές υπολογιστικού νέφους είτε και να είναι ιδιόκτητες και διαθέτουν πρακτικά απεριόριστους πόρους και υπολογιστική ισχύ, συγκριτικά με τις συσκευές στα άκρα του δικτύου. Αξίζει τέλος να σημειωθεί ότι οι κατάλληλα τοποθετημένες υπολογιστικές και αποθηκευτικές μονάδες που βρίσκονται ανάμεσα στον τελικό χρήστη και στις υποδομές

υπολογιστικού νέφους ή και κεντρικών μονάδων αναφέρονται συνήθως στη βιβλιογραφία ως fog computing [10].

Οι τοπολογίες edge computing παρουσιάζουν πολλαπλά πλεονεκτήματα. Πιο συγκεκριμένα, επιδιώκοντας την επεξεργασία και αποθήκευση των δεδομένων στα λογικά άκρα της τοπολογίας, δηλαδή είτε κοντά στους χρήστες είτε στις κεντρικές υποδομές υπολογισμού και αποθήκευσης, περιορίζονται οι ανάγκες σε εύρος δικτύου (bandwidth), μειώνονται τα λειτουργικά κόστη και ενισχύεται η αυτονομία μεταξύ εφαρμογών και η αξιοπιστία των λειτουργιών του συστήματος. Επιπλέον, οι υπηρεσίες edge computing ενισχύουν την επεξεργασία δεδομένων σε σχεδόν πραγματικό, συμβάλλοντας κατά συνέπεια στη βελτίωση του χρόνου απόκρισης και της εμπειρίας αλληλεπίδρασης των χρηστών με το σύστημα. Αξίζει να σημειωθεί επίσης ότι η γεωγραφική εγγύτητα των υπολογισμών στους τελικούς χρήστες, μπορεί να ενισχύσει τις πολιτικές απορρήτου για την ασφάλεια των δεδομένων. Αυτό γιατί τα δεδομένα δε χρειάζεται να μεταφερθούν αναγκαστικά για ανάλυση στις κεντρικές υποδομές μέσω του δημοσίου μη αξιόπιστου δικτύου.

Παρ’ όλα αυτά, οι τοπολογίες edge computing παρουσιάζουν σημαντικούς περιορισμούς, οι οποίοι σχετίζονται κυρίως με τα χαρακτηριστικά των συσκευών στα άκρα του δικτύου. Μεταξύ άλλων, όπως προαναφέρθηκε, οι συσκευές αυτές έχουν συνήθως μειωμένες υπολογιστικές δυνατότητες, χωρητικότητα αποθήκευσης και εύρος δικτύου προς τις κεντρικές υποδομές επεξεργασίας, με αποτέλεσμα συχνά μην είναι εφικτή η εκτέλεση εξειδικευμένου λογισμικού (π.χ. ενορχήστρωσης και παρακολούθησης εφαρμογών). Οι συσκευές αυτές παρουσιάζουν επίσης μεγάλη ετερογένεια δυσχεραίνοντας την ενσωμάτωση και τη διαχείρισή τους στο σύστημα. Ταυτόχρονα, μπορεί να παρουσιάζουν απότομες διακοπές τροφοδοσίας και συχνότερες απώλειες συνδεσιμότητας, συγκριτικά με τους κόμβους των κεντρικών υποδομών επεξεργασίας. Τέλος, οι κόμβοι στις κεντρικές υποδομές ενδέχεται να αντιμετωπίζουν δυσκολίες σύνδεσης με τους κόμβους στα άκρα του δικτύου λόγω της ύπαρξης τειχών προστασίας (firewalls). Για την αντιμετώπιση των εγγενών περιορισμών των τοπολογιών edge computing και τη διευκόλυνση της επεξεργασίας δεδομένων και της λήψης αποφάσεων στα άκρα του δικτύου αναπτύσσονται ποικίλα νέα προϊόντα και πραγματοποιείται προσπάθεια επέκτασης των πλατφορμών που αρχικά σχεδιάστηκαν για υποδομές υπολογιστικού νέφους [11], [12], [13].

Ωστόσο, συγκριτικά με τις εγγενείς εφαρμογές υπολογιστικού νέφους, μέχρι σήμερα δεν υπάρχει αυστηρή προτυποποίηση για τις εφαρμογές που εκτελούνται σε τοπολογίες edge computing. Σύμφωνα με τους [10], [14], οι εγγενείς εφαρμογές edge computing χρειάζεται να λαμβάνουν υπόψη τόσο τους περιορισμούς στα άκρα του δικτύου (π.χ. περιορισμένοι υπολογιστικοί και αποθηκευτικοί πόροι, αυξημένο κόστος διαχείρισης κατανεμημένων υποδομών συγκριτικά με κεντρικές υποδομές), όσο και να αξιοποιούν τις δυνατότητες αυτού του σχεδιασμού (π.χ. μικροί χρόνοι απόκρισης, επεκτασιμότητα εύρους ζώνης, ασφάλεια δεδομένων). Ταυτόχρονα όμως, υπογραμμίζουν ότι οι εγγενείς εφαρμογές edge computing χρειάζεται να αξιοποιούν και τις δυνατότητες των πόρων στα

άκρα του δικτύου και σε μικρότερο βαθμό, μόνο όταν είναι απαραίτητο, να χρησιμοποιούν τους πόρους υπολογιστικού νέφους και ιδιόκτητων κεντρικών υποδομών για τη βελτίωση των παρεχόμενων υπηρεσιών στους χρήστες.

Μάλιστα, ανάλογα με τον βαθμό χρησιμοποίησης των πόρων στα άκρα του δικτύου και των κεντρικών υποδομών, οι Satyanarayanan, Klas, Silva και Mangiante [14] κατατάσσουν τις εφαρμογές σε εφαρμογές συσκευών, σε εγγενείς εφαρμογές υπολογιστικού νέφους ενισχυόμενες από υποδομές edge, σε εγγενείς εφαρμογές συσκευών ενισχυόμενες από υποδομές edge και σε εγγενείς εφαρμογές edge. Σύμφωνα με αυτή την κατάταξη, οι εφαρμογές συσκευών έχουν πρόσβαση αποκλειστικά μόνο στους πόρους των συσκευών στα άκρα του δικτύου. Οι εγγενείς εφαρμογές υπολογιστικού νέφους ενισχυόμενες από υποδομές edge αξιοποιούν τους πόρους στα άκρα του δικτύου όταν είναι διαθέσιμοι, αλλά έχουν ικανοποιητική λειτουργία και όταν εκτελούνται αποκλειστικά σε υποδομές υπολογιστικού νέφους. Συνεχίζοντας, οι εγγενείς εφαρμογές συσκευών ενισχυόμενες από υποδομές edge εκτελούνται σε συσκευές στα άκρα του δικτύου όμως παρέχουν βελτιωμένες υπηρεσίες όταν έχουν πρόσβαση σε πόρους fog. Τέλος, οι εγγενείς εφαρμογές edge χρησιμοποιούν σχεδόν αποκλειστικά τους πόρους των συσκευών στα άκρα του δικτύου και τους πόρους fog.

Συνεπώς, οι εγγενείς εφαρμογές edge εκτελούνται κυρίως στις συσκευές στα άκρα του δικτύου και σε fog πόρους και χρησιμοποιούν τις υπολογιστικές και αποθηκευτικές δυνατότητες κεντρικών υποδομών μόνο όταν απαιτείται. Το σημείο στο οποίο θα εκτελεστεί ο κώδικας, ή τμήμα του κώδικα, μια εφαρμογής σε μία τοπολογία edge εξαρτάται μεταξύ άλλων από τις υπολογιστικές και αποθηκευτικές απαιτήσεις του, από το πλήθος των χρηστών που χρειάζεται να έχουν πρόσβαση στα αποτελέσματα εκτέλεσής του, από το αν πρέπει να ληφθεί υπόψη η κατάσταση ενός υποσυστήματος ή του συνολικού συστήματος της τοπολογίας και από θέματα που αφορούν την ασφάλεια των διαχειριζόμενων δεδομένων [15]. Συνήθεις εφαρμογές edge computing είναι αυτές που διαχειρίζονται δεδομένα IoT, επαυξημένης και εικονικής πραγματικότητας και τηλεπικοινωνιών οι οποίες απαιτούν μειωμένους χρόνους απόκρισης και λήψη αποφάσεων σε πραγματικό χρόνο [16].

1.2 Το Αντικείμενο της Διπλωματικής Εργασίας

Οι αυξανόμενες απαιτήσεις των σύγχρονων εφαρμογών λογισμικού έχουν οδηγήσει στην ανάπτυξη εφαρμογών ως ομάδων χαλαρά συζευγμένων υπηρεσιών και έχουν δημιουργήσει την ανάγκη χρήσης πόρων διαφορετικών χαρακτηριστικών και δυνατοτήτων τους οποίους διαχειρίζονται πολλοί ανεξάρτητοι πάροχοι. Συνδυαστικά με την ευρεία ανάπτυξη λογισμικού ανοιχτού κώδικα (open-source), έχει δημιουργηθεί πληθώρα τεχνολογιών και εργαλείων για την αυτοματοποίηση της διαχείρισης των εφαρμογών στο επίπεδο των μεμονωμένων υποδομών. Ωστόσο, η εκτέλεση

εφαρμογών σε κατανεμημένα συστήματα που ενσωματώνουν ετερογενείς πόρους παρουσιάζει ακόμα αρκετές δυσκολίες.

Έτσι, η παρούσα διπλωματική εργασία αποσκοπεί στη μελέτη των υπαρχόντων τεχνολογιών και στον σχεδιασμό των απαραίτητων μηχανισμών που θα εξασφαλίζουν τη διαφανή και αποτελεσματική ενορχήστρωση και εκτέλεση εγγενών εφαρμογών υπολογιστικού νέφους σε ένα σύστημα με ετερογενείς υπολογιστικές υποδομές, ανεξάρτητα από τις επιμέρους ιδιαιτερότητές τους. Πιο συγκεκριμένα, θεωρούμε ένα σύστημα με επιμέρους πολλαπλές διαθέσιμες edge και cloud υποδομές και στόχος είναι η αποδοτική ανάθεση των ετερογενών υπολογιστικών και αποθηκευτικών πόρων τους στις εφαρμογές των τελικών χρηστών. Η ύπαρξη edge υποδομών θα εξασφαλίζει την διάθεση των κατάλληλων πόρων πολύ κοντά στους τελικούς χρήστες και στα σημεία δημιουργίας των δεδομένων. Αντίστοιχα, οι cloud υποδομές θα αξιοποιούνται για πιο κρίσιμες εφαρμογές με μεγαλύτερο υπολογιστικό φορτίο και ανάγκες κλιμάκωσης. Με αυτόν τον τρόπο, οι υπηρεσίες που απαιτούν άμεση απόκριση του συστήματος θα εξυπηρετούνται κοντά στα σημεία δημιουργίας των σχετικών δεδομένων, ενώ οι εφαρμογές μεγάλου όγκου δεδομένων και υψηλού υπολογιστικού φορτίου θα κατανέμονται έξυπνα στις cloud υποδομές.

Στο πλαίσιο της διπλωματικής εργασίας, αρχικά μελετήθηκαν και αξιολογήθηκαν ως προς τις κυριότερες λειτουργίες τους οι πιο διαδεδομένοι μηχανισμοί ενορχήστρωσης εγγενών εφαρμογών υπολογιστικού νέφους (container orchestrators) ανοιχτού κώδικα οι οποίοι θα μπορούσαν να είναι υπεύθυνοι για την τοπική ενορχήστρωση των εφαρμογών στις επιμέρους υποδομές cloud του συστήματος (Κεφάλαιο 2). Μεταξύ αυτών, με κριτήρια την ετερογένεια, την προτυποποίηση και τον βαθμό χρήσης τους στο σύγχρονο επιχειρησιακό περιβάλλον επιλέχθηκαν οι SwarmKit και Kubernetes και μελετήθηκαν περαιτέρω ως προς την αρχιτεκτονική τους και τα πρότυπα που ακολουθούν για την περιγραφή των εφαρμογών (Κεφάλαιο 3). Επίσης, αναλύθηκαν οι μηχανισμοί ανάθεσης των εργασιών σε πόρους που χρησιμοποιούν τα εργαλεία και εντοπίστηκαν οι δυνατότητες επέκτασης των λειτουργιών τους, προκείμενου να υλοποιηθούν παραμετροποιήσιμοι και εύκολα επεκτάσιμοι μηχανισμοί κατανομής των πόρων στο συνολικό σύστημα (Κεφάλαιο 4).

Τη θεωρητική μελέτη συμπλήρωσε ο σχεδιασμός της αρχιτεκτονικής του κατανεμημένου συστήματος εκτέλεσης εφαρμογών στο οποίο μπορούν να ενσωματώνονται ετερογενείς υποδομές cloud και edge, αλλά και να χρησιμοποιούνται, στο επίπεδο των επιμέρους πόρων αυτών των υποδομών, διαφορετικοί μηχανισμοί ενορχήστρωσης. Μάλιστα, υλοποιήθηκαν οι μηχανισμοί αυτού του συστήματος για την υποβολή και τον τερματισμό εφαρμογών σε cloud υποδομές με χρήση των εργαλείων SwarmKit και Kubernetes (Κεφάλαιο 5). Τέλος, βάσει αυτής της υλοποίησης, εντοπίστηκαν πρακτικές δυσκολίες διαχείρισης εγγενών εφαρμογών υπολογιστικού νέφους σε ετερογενείς πόρους από πολλαπλές διαφορετικές πλατφόρμες ενορχήστρωσης, καθώς και σημεία μελλοντικής έρευνας για την περαιτέρω θεωρητική θεμελίωση του προβλήματος (Κεφάλαιο 6).

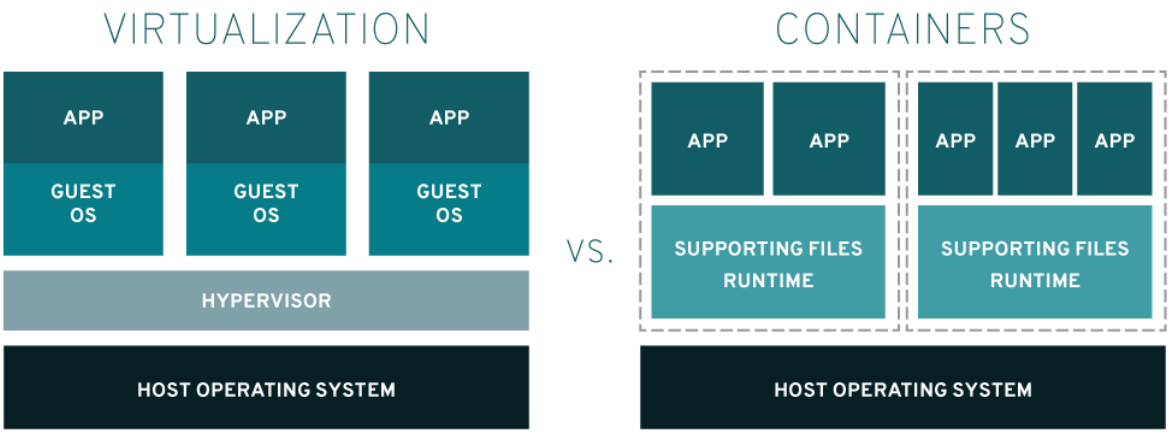
Κεφάλαιο 2: Containers και Τεχνολογίες Ενορχήστρωσης

2.1 Containers

Τα containers είναι εκτελέσιμες μονάδες λογισμικού που χρησιμοποιούνται συνήθως για την ανάπτυξη και την εκτέλεση των μικροϋπηρεσιών των εγγενών εφαρμογών υπολογιστικού νέφους. Η τεχνολογία των containers βασίζεται σε εικονικοποίηση σε επίπεδο λειτουργικού συστήματος. Πιο συγκεκριμένα, το λογισμικό των containers εγκαθίσταται πάνω από τον πυρήνα (kernel) του λειτουργικού συστήματος ενός υπολογιστικού μηχανήματος (host operating system). Επιπλέον, πέρα από τον διαμοιρασμό του υλικού και του λειτουργικού συστήματος ενός μηχανήματος, τα containers μπορούν να διαμοιράζονται και δυαδικά αρχεία (binaries) και βιβλιοθήκες κώδικα. Κάθε container ορίζεται βάσει ενός αρχείου κώδικα, της εικόνας (image), το οποίο περιλαμβάνει τον πηγαίο κώδικα (source code) μιας εφαρμογής ή τμήματός της και τις απαιτούμενες βιβλιοθήκες, εξαρτήσεις και αρχεία παραμετροποίησης για την εκτέλεσή του. Χάρη στην εικονικοποίηση πολλαπλές εφαρμογές σε containers μπορούν να εκτελούνται απομονωμένα η μία από την άλλη και με σχετικά μικρό υπολογιστικό φόρτο σε διαφορετικές πλατφόρμες, όπως εικονικά μηχανήματα, κεντρικές υποδομές και υποδομές υπολογιστικού νέφους [17], [18].

Η ανάπτυξη και εκτέλεση εφαρμογών με χρήση containers (containerization) είναι πιο αποδοτική αυτής με χρήση εικονικών μηχανών. Μεταξύ άλλων, όπως φαίνεται στην Εικόνα 2.1, το λογισμικό του hypervisor αντικαθίσταται στα containers από μικρά σε μέγεθος αρχεία παραμετροποίησης και από το λογισμικό του μηχανισμού εκτέλεσης των containers (container runtime engine). Επίσης, χάρη στην εικονικοποίηση, δε δημιουργείται ένα αντίγραφο του λειτουργικού συστήματος και του πυρήνα του για κάθε εφαρμογή ή μικροϋπηρεσία (guest operating system).

Κατά αυτόν τον τρόπο, η τεχνολογία των containers παρουσιάζει πολλαπλά πλεονεκτήματα. Αρχικά, συμβάλλει στην πιο αποδοτική διαχείριση των διαθέσιμων πόρων και στον περιορισμό των σφαλμάτων των εφαρμογών, αφού κάθε εφαρμογή εκτελείται σε ένα υπολογιστικό μηχάνημα απομονωμένη από τις υπόλοιπες. Συγκριτικά μάλιστα με τα εικονικά μηχανήματα, η χρήση containers διευκολύνει τη φορητότητα των εφαρμογών στις διάφορες πλατφόρμες και μειώνει το μέγεθος των εκτελέσιμων μονάδων. Ταυτόχρονα, αυξάνει την ταχύτητα εκτέλεσης των εφαρμογών γιατί, δεδομένου ότι δε χρησιμοποιείται αντίγραφο του λειτουργικού συστήματος, οι χρόνοι έναρξης των containers είναι μικροί [19], [17], [20].



Εικόνα 2.1: Σύγκριση των εικονικών μηχανών με τα containers

Η τεχνολογία των containers έχει αναπτυχθεί εδώ και πολλές δεκαετίες. Ωστόσο, διάδοση της χρήσης τους παρατηρήθηκε μετά το 2008, όταν το Linux ενσωμάτωσε τη λειτουργία των containers στον πυρήνα του, όπως και το 2013, οπότε το λογισμικό των Docker containers έγινε ανοιχτού κώδικα. Έτσι, μέχρι σήμερα έχουν αναπτυχθεί ποικίλες τεχνολογίες containers πέρα από τα Docker containers, όπως τα containerd, CoreOS rkt, Mesos Containerizer, LXC Linux Containers, OpenVZ και crio-d. Μάλιστα, για την προτυποποίηση των λειτουργιών των διαφόρων τεχνολογιών και την περαιτέρω διευκόλυνση της φορητότητας των εφαρμογών αναπτύχθηκε και καθιερώθηκε το 2015 το πρωτόκολλο Open Container Initiative (OCI) [20]. Στο πλαίσιο της διπλωματικής εργασίας, χρησιμοποιήθηκε η ανοιχτού κώδικα τεχνολογία των Docker containers ως η πιο διαδεδομένη και προτυποποιημένη από το πρωτόκολλο OCI και τον οργανισμό CNCF λύση.

2.2 Τεχνολογίες Ενορχήστρωσης Containers

Οι τεχνολογίες ενορχήστρωσης containers αναπτύχθηκαν για να διευκολύνουν τη διαχείριση σύνθετων εφαρμογών σε containers σε όλο τον κύκλο ζωής τους. Πρόκειται για πακέτα λογισμικού τα οποία αυτοματοποιούν σε μεγάλο βαθμό απαραίτητες διαδικασίες για την ανάπτυξη, εκτέλεση, κλιμάκωση και παρακολούθηση containerized εφαρμογών που εκτελούνται κατανεμημένα, σε συμπλέγματα υπολογιστικών πόρων (clusters). Τα εργαλεία ενορχήστρωσης ενσωματώνουν επίσης ευφυείς λειτουργίες, καθώς και παρέχουν δυνατότητες επέκτασης των υλοποιημένων μηχανισμών για την καλύτερη εξυπηρέτηση των αιτημάτων των χρηστών. Όπως θα αναλυθεί στην Ενότητα 2.3, έχουν αναπτυχθεί πολυάριθμες λύσεις ενορχήστρωσης containers, όπως τα Apache Mesos, Kubernetes, Nomad και Swarm [21], [22].

Παρά τις επιμέρους διαφορές στην υλοποίηση και στις παρεχόμενες δυνατότητες των εργαλείων ενορχήστρωσης, η διαδικασία ενορχήστρωσης των εφαρμογών περιλαμβάνει πάντοτε τα ακόλουθα τέσσερα στάδια. Πρώτον, δημιουργείται ένα σύμπλεγμα υπολογιστικών μηχανημάτων για την εκτέλεση των εφαρμογών. Τα μηχανήματα αυτά μπορεί να είναι πραγματικά ή εικονικά και να

βρίσκονται είτε σε ιδιόκτητες υποδομές είτε σε υποδομές υπολογιστικού νέφους. Δεύτερον, ορίζεται η προς εκτέλεση σε containers εφαρμογή, στις περισσότερες τεχνολογίες με δηλωτικό τρόπο. Η περιγραφή της εφαρμογής γίνεται συνήθως σε ένα αρχείο μορφοτύπου YAML ή JSON, στο οποίο προσδιορίζονται, μεταξύ άλλων, η εικόνα των containers και η επιθυμητή κατάστασή τους, οι πόροι υλικού και λογισμικού που θα χρησιμοποιηθούν και οι απαραίτητες διασυνδέσεις δικτύου. Τρίτον, ο αλγόριθμος ανάθεσης των εργασιών σε πόρους του μηχανισμού ενορχήστρωσης επιλέγει τους υπολογιστικούς κόμβους του cluster στους οποίους θα εκτελεστούν τα containers της εφαρμογής, λαμβάνοντας υπόψη ποικίλα κριτήρια όπως θα αναλυθεί στο Κεφάλαιο 4, και τα αναθέτει για εκτέλεση σε αυτούς. Τέταρτον, το εργαλείο ενορχήστρωσης είναι υπεύθυνο για τη διαχείριση των containers σε όλη τη διάρκεια του κύκλου ζωής της εφαρμογής, βάσει της επιθυμητής κατάστασης που προσδιορίστηκε κατά τον ορισμό τους. Στις λειτουργίες διαχείρισης περιλαμβάνονται η προς τα πάνω, ή προς τα κάτω, κλιμάκωση της εφαρμογής με αύξηση, ή αντίστοιχα μείωση, του πλήθους των containers της, η δυναμική ανάθεση των containers για εκτέλεση στους πόρους του συστήματος ανάλογα με τις εκάστοτε απαιτήσεις της εφαρμογής, ο έλεγχος της διαθεσιμότητάς τους και η πραγματοποίηση των απαραίτητων ενημερώσεων λογισμικού [22], [23], [24].

2.3 Σύγκριση των Τεχνολογιών Ενορχήστρωσης Containers Ανοιχτού Κώδικα

Όπως προαναφέρθηκε, έχουν αναπτυχθεί πολυάριθμα εργαλεία ενορχήστρωσης εφαρμογών σε containers. Σε αυτή την ενότητα συγκρίθηκαν οι πιο διαδεδομένες τεχνολογίες ενορχήστρωσης ανοιχτού κώδικα [25] ως προς τις κυριότερες ενσωματωμένες (built-in) λειτουργίες τους, όπως περιγράφονται στη βιβλιογραφία [23], [26], [27], [28], [29] μέχρι τις εκδόσεις του 2019.

Πιο συγκεκριμένα, συγκρίθηκαν τα εργαλεία SwarmKit[2], Kubernetes[3], Apache Mesos[4], DC/OS[5] και Nomad[6], καθώς και το πλαίσιο (framework) των Apache Mesos και DC/OS, Marathon[7]. Πρώτον, το SwarmKit δημιουργήθηκε από το Docker για τη χρονοδρομολόγηση εργασιών σε κατανεμημένα συστήματα. Πιο συχνά στη βιβλιογραφία γίνεται αναφορά στο Swarm mode, μία ενσωματωμένη στο Docker Engine λειτουργία η οποία χρησιμοποιεί βιβλιοθήκες του SwarmKit για την ενορχήστρωση containers [30]. Δεύτερον, το Kubernetes αναπτύχθηκε αρχικά από την Google για την αυτοματοποιημένη εκτέλεση, κλιμάκωση και διαχείριση εφαρμογών σε containers. Τρίτον, το Apache Mesos είναι το πρώτο εργαλείο που υλοποιήθηκε για τη διαχείριση εφαρμογών και frameworks σε κατανεμημένα συστήματα και αναπτύχθηκε από το Πανεπιστήμιο του Berkeley.

[2] https://github.com/docker/swarmkit#swarmkit
[3] https://kubernetes.io/docs/home/
[4] http://mesos.apache.org/documentation/latest/
[5] https://docs.d2iq.com/mesosphere/dcos/2.2/
[6] https://www.nomadproject.io/docs
[7] https://mesosphere.github.io/marathon/docs/

Συνεχίζοντας, βασιζόμενο στο Mesos και επεκτείνοντας τις λειτουργίες του, το Distributed Cloud Operating System (DC/OS) είναι μια κατανεμημένη πλατφόρμα για την ενορχήστρωση εργασιών που αναπτύχθηκε από την D2iQ, πρώην Mesosphere. Επιπλέον, το Marathon είναι ένα από τα πιο διαδεδομένα frameworks ενορχήστρωσης. Μπορεί να εκτελεστεί τόσο στην πλατφόρμα του Apache Mesos, όσο και σε αυτή του DC/OS. Στο πλαίσιο της σύγκρισης που πραγματοποιήθηκε σε αυτή την ενότητα, μελετήθηκε ως εργαλείο ενορχήστρωσης εκτελούμενο στο Apache Mesos. Τέλος, το Nomad είναι ένας ενορχηστρωτής που σχεδιάστηκε από την HashiCorp για την εκτέλεση και διαχείριση διαφορετικών τύπων εργασιών, όπως εφαρμογών σε containers, εφαρμογών που αναπτύχθηκαν βάσει της αρχιτεκτονικής των μικροϋπηρεσιών και εργασιών που υποβάλλονται για εκτέλεση σε δέσμες (batch).

Τα κριτήρια σύγκρισης των τεχνολογιών ενορχήστρωσης ομαδοποιήθηκαν στις κατηγορίες Γενική Αρχιτεκτονική και Εγκατάσταση, Περιγραφή, Εκτέλεση και Αναβάθμιση Εφαρμογών, Διαχείριση και Παρακολούθηση Εφαρμογών και Συστήματος, Έλεγχος Συστήματος και Εξασφάλιση Ποιότητας Παρεχόμενων Υπηρεσιών, Παραμετροποίηση και Επεκτασιμότητα, Διασύνδεση και Ασφάλεια. Ωστόσο, λόγω της ταχείας ανάπτυξης αυτών των τεχνολογιών, με τη δημιουργία νέων χαρακτηριστικών και την επέκταση ή και κατάργηση υπαρχόντων, η δημιουργία επικαιροποιημένης αναφοράς σύγκρισης των λειτουργιών τους είναι μία ιδιαίτερα σύνθετη και απαιτητική διαδικασία. Έτσι, τα συμπεράσματα που έχουν εξαχθεί σε αυτή την ενότητα είναι ενδεικτικά και λεπτομερή στον βαθμό που ήταν εφικτό στο πλαίσιο μιας διπλωματικής εργασίας. Αξίζει να σημειωθεί ωστόσο ότι ο αυξημένος αυτός ρυθμός ανάπτυξης που παρατηρήθηκε, συνδυαστικά με τη σχετική ετερογένεια των εργαλείων ενισχύουν την ιδέα δημιουργίας μιας γενικής πλατφόρμας, η οποία δε θα αναπτυχθεί βασιζόμενη σε συγκεκριμένες τεχνολογίες αλλά θα υλοποιεί διεπαφές για τη διαφανή και εύκολη ενσωμάτωση των εκάστοτε κάθε φορά χρησιμοποιούμενων τεχνολογιών.

Γενική Αρχιτεκτονική και Εγκατάσταση

Ο Πίνακας 2.1 συνοψίζει τα κυριότερα χαρακτηριστικά της αρχιτεκτονικής και τις εναλλακτικές δυνατότητες εγκατάστασης των εργαλείων ενορχήστρωσης. Αρχικά, όλες οι συγκρινόμενες τεχνολογίες υλοποιούν την αρχιτεκτονική master-worker και υποστηρίζουν τη δημιουργία πολλαπλών κεντρικών κόμβων υπολογισμού (masters) για υψηλή διαθεσιμότητα. Η ενορχήστρωση με Apache Mesos και Marathon ή και με Nomad προτιμάται για clusters περισσότερων υπολογιστικών κόμβων, συγκριτικά με τα SwarmKit και Kubernetes. Ταυτόχρονα, ως προς την προσβασιμότητα, όλες οι τεχνολογίες διαθέτουν HTTP APIs για πρόσβαση στους κεντρικούς κόμβους υπολογισμού και βιβλιοθήκες για την προγραμματιστική διαχείριση του cluster (client libraries).

Ωστόσο, ως προς την προτυποποίηση, το Kubernetes υποστηρίζει τα πρωτόκολλα Open Container Initiative (OCI) και Containerd για την εκτέλεση containers, καθώς και τα Container Network

Interface (CNI) για δικτυακές υπηρεσίες και Common Storage Interface (CSI) για μόνιμες δομές αποθήκευσης. Επίσης, το Kubernetes έχει προτυποποιηθεί από το CNCF και υποστηρίζει τις περισσότερες δυνατότητες εγκατάστασης σε διαφορετικούς υπολογιστικούς πόρους.

Περιγραφή, Εκτέλεση και Αναβάθμιση Εφαρμογών

Ο Πίνακας 2.2 παρουσιάζει τις λειτουργίες των εργαλείων που αφορούν την εκτέλεση, την κλιμάκωση και την αναβάθμιση των εφαρμογών και τη μόνιμη αποθήκευση δεδομένων. Αρχικά, μόνο τα Kubernetes, DC/OS και Nomad υποστηρίζουν εφαρμογές που διατηρούν (stateful) ή και δε διατηρούν (stateful) την κατάστασή τους, εργασίες που υποβάλλονται για εκτέλεση σε δέσμες (batch jobs), καθώς και εργασίες που εκτελούνται ανά προκαθορισμένα χρονικά διαστήματα (cron jobs). Ακόμη, με εξαίρεση το SwarmKit, όλα τα υπόλοιπα εργαλεία προτιμώνται κυρίως για την εκτέλεση σύνθετων εφαρμογών με αυξημένες απαιτήσεις ενορχήστρωσης, όπως εγγενών εφαρμογών υπολογιστικού νέφους που αποτελούνται από πολλές μικροϋπηρεσίες. Η περιγραφή των εφαρμογών γίνεται σε όλες τις τεχνολογίες με δηλωτικό τρόπο.

Συνεχίζοντας, για τον καλύτερο έλεγχο των εφαρμογών σε όλο τον κύκλο ζωής τους, οι ομάδες ανάπτυξης χρειάζεται να κατανοήσουν την ορολογία που χρησιμοποιείται στις διάφορες πλατφόρμες. Για παράδειγμα, σε όλα τα συστήματα εκτός από το Kubernetes, η μικρότερη μονάδα εκτέλεσης είναι το task. Στο SwarmKit ένα task αποτελείται από ένα container, ενώ στα Mesos και Nomad το task μπορεί να αποτελείται από ένα container ή και από ολόκληρες containerized και non-containerized εφαρμογές. Στο Kubernetes, η μικρότερη μονάδα εκτέλεσης είναι το Pod, μία ομάδα containers που εκτελούνται ταυτόχρονα υλοποιώντας συνήθως μία συγκεκριμένη λογική λειτουργία του συστήματος.

Βάσει της βιβλιογραφίας επίσης, μόνο τα Kubernetes και Nomad υποστηρίζουν την αυτόματη κλιμάκωση των πόρων ανάλογα με τις απαιτήσεις των εφαρμογών. Τέλος, όπως παρουσιάζεται πιο αναλυτικά στον πίνακα, τα Kubernetes, DC/OS και Nomad υποστηρίζουν τις περισσότερες λειτουργίες για την αναβάθμιση των εφαρμογών.

Διαχείριση και Παρακολούθηση Εφαρμογών και Συστήματος

Ο Πίνακας 2.3 συνοψίζει τις λειτουργίες των εργαλείων ενορχήστρωσης που σχετίζονται με τη διαχείριση και την παρακολούθηση των εφαρμογών σε όλο τον κύκλο της ζωής τους, αλλά και του συστήματος συνολικά. Αρχικά, όλα τα υπό εξέταση εργαλεία ενσωματώνουν διεπαφή γραμμής εντολών (Command Line Interface, CLI) για πρόσβαση στο API του cluster και, με εξαίρεση το SwarmKit, δίνεται και δυνατότητα δικτυακής πρόσβασης σε αυτήν. Επιπλέον, στην ανοιχτή έκδοση των Kubernetes, DC/OS και Nomad υποστηρίζεται CLI και γραφική διεπαφή χρήστη (Graphical User Interface, GUI) για την παρακολούθηση των πόρων του συστήματος. Ακόμη, όλα τα εργαλεία διαθέτουν μηχανισμούς ανίχνευσης σφαλμάτων και αλλαγών στα υπάρχοντα containers και με

εξαίρεση το Nomad, τα υπόλοιπα εργαλεία ενσωματώνουν μηχανισμούς παρακολούθησης της διαθεσιμότητας των containers. Τέλος, όλες οι υπό εξέταση τεχνολογίες υποστηρίζουν καταγραφή των συμβάντων (logging) που παρατηρούνται στις ενσωματωμένες λειτουργίες τους, αλλά μόνο τα Kubernetes, DC/OS και Nomad διαθέτουν εργαλεία για την ανάλυση των σχετικών συλλεγόμενων δεδομένων στις εκδόσεις ανοιχτού τους κώδικα. Όπως φαίνεται στον πίνακα, τα DC/OS και Nomad υποστηρίζουν τις περισσότερες λειτουργίες αυτής της κατηγορίας.

Έλεγχος Συστήματος και Εξασφάλιση Ποιότητας Παρεχόμενων Υπηρεσιών

Ο Πίνακας 2.4 παρουσιάζει τις λειτουργίες που αποσκοπούν στον καλύτερο έλεγχο των πόρων του συστήματος. Όπως φαίνεται, μόνο τα Kubernetes, Mesos και DC/OS υποστηρίζουν τον ορισμό ομάδων χρηστών (user groups) για τον περιορισμό της πρόσβασης στις υπηρεσίες των εφαρμογών, στους υπολογιστικούς κόμβους και στις αποθηκευτικές μονάδες του cluster. Ωστόσο, μόνο τα Kubernetes, Mesos και Nomad υποστηρίζουν τον προσδιορισμό μεγίστων ορίων χρήσης τόσο του επεξεργαστή, όσο και της μνήμης και του δίσκου.

Ως προς τις λειτουργίες για την εξασφάλιση υψηλής ποιότητας παρεχόμενων υπηρεσιών τα SwarmKit, Kubernetes και Nomad εξασφαλίζουν την κατ' ελάχιστη πρόσβαση κάθε εφαρμογής στον επεξεργαστή και στη μνήμη των κόμβων. Όλα τα υπό εξέταση εργαλεία υποστηρίζουν δυνατότητες ελέγχου των κόμβων που θα επιλεχθούν για την εκτέλεση κάθε εφαρμογής βάσει των απαιτήσεών της, καθώς και επανάληψης της προσπάθειας ανάθεσης των εργασιών σε πόρους σε περίπτωση σφάλματος.

Παραμετροποίηση και Επεκτασιμότητα

Ο Πίνακας 2.5 συγκεντρώνει τις λειτουργίες που αφορούν την παραμετροποίηση και την επεκτασιμότητα των μηχανισμών του συστήματος. Η πλατφόρμα του Kubernetes υποστηρίζει όλες τις υπό εξέταση λειτουργίες παρέμβασης στα αρχεία παραμετροποίησης στο επίπεδο των containers και επεκτασιμότητας των μηχανισμών του συστήματος μέσω πρόσθετων λογισμικών (plugins).

Διασύνδεση

Ο Πίνακας 2.6 συνοψίζει τις λειτουργίες που αφορούν τη διασύνδεση των υπηρεσιών μιας εφαρμογής και τον εντοπισμό των εφαρμογών που έχουν υποβληθεί στο σύστημα. Αρχικά, όλα τα υπό εξέταση εργαλεία εκτός του Nomad υποστηρίζουν και τους τρεις τρόπους επικοινωνίας των υπηρεσιών (services) των εφαρμογών, χωρίς να υποστηρίζεται ωστόσο σε κάποιες περιπτώσεις η χρήση στατικού ονόματος Domain Name System (DNS) για τον προσδιορισμό συγκεκριμένης υπηρεσίας. Επίσης, το σύνολο αυτών των εργαλείων υποστηρίζει εσωτερικό μηχανισμό DNS για την ανάθεση αναγνωριστικών στις υπηρεσίες, αλλά και δυνατότητα εξωτερικής πρόσβασης σε αυτές μέσω του πλέγματος δρομολόγησης (routing mesh). Ωστόσο, αν επιλεχθεί το Nomad, απαιτείται

επιπλέον η χρήση του εργαλείου Consul[8] για τον εντοπισμό (service discovery) και τη διασύνδεση (service networking) των υπηρεσιών.

Ασφάλεια

Ο Πίνακας 2.7 παρουσιάζει τις λειτουργίες που σχετίζονται με την ασφάλεια του συστήματος και των containers των εφαρμογών. Αξίζει να σημειωθεί ότι οι εκδόσεις των SwarmKit και Kubernetes δε διαφέρουν σημαντικά ως προς τις λειτουργίες που εξετάστηκαν αφού και οι δύο υποστηρίζουν την ταυτοποίηση και εξουσιοδότηση των χρηστών και την ταυτοποίηση των workers κόμβων μέσω του API του manager κόμβου, όπως και την κωδικοποίηση των ανταλλασσόμενων μηνυμάτων μεταξύ των εφαρμογών, τον περιορισμό της εξωτερικής πρόσβασης στις θύρες των υπηρεσιών και την προστασία των ευαίσθητων δεδομένων στα containers των εφαρμογών. Ακόμη, πολλές από τις υπό εξέταση λειτουργίες δεν υποστηρίζονται από τα Mesos, Mesos με χρήση του Marathon και από την έκδοση ανοιχτού κώδικα του DC/OS, ενώ για κάποιες από τις λειτουργίες δε βρέθηκε αν υποστηρίζονται από το Nomad. Τέλος, αξίζει να σημειωθεί ότι μόνο το Kubernetes διαθέτει μηχανισμούς απομόνωσης ενός container από άλλες εργασίες που εκτελούνται στον ίδιο κόμβο με αυτό, λειτουργία που είναι σημαντική στην περίπτωση που οι χρησιμοποιούμενες υποδομές είναι προσβάσιμες από πολλαπλές διαφορετικές ομάδες χρηστών.

[8] https://www.consul.io

Πίνακας 2.1: Γενική Αρχιτεκτονική και Εγκατάσταση

Λειτουργία	**SwarmKit**	**Kubernetes**	**Mesos**	**Mesos & Marathon**	**DC/OS**	**Nomad**
✓: η λειτουργία υποστηρίζεται - : η λειτουργία δεν υποστηρίζεται / δεν είναι εφαρμόσιμη (κενό): δεν υπάρχει αναφορά στη βιβλιογραφία για τη συγκεκριμένη λειτουργία						
Γενική Αρχιτεκτονική						
Τύπος αρχιτεκτονικής	Master-Worker	Master-Worker	Master-Worker	Master-Worker	Master-Worker	Master-Worker
Μέγεθος cluster	Μέχρι 1,000 κόμβους	Μέχρι 5,000 κόμβους	-	Μέχρι 10,000 κόμβους	Δε βρέθηκε σχετική πληροφορία	Πάνω από 10,000 κόμβους
Υποστηριζόμενα πρότυπα container runtime	Open Container Initiative (OCI), containerd	OCI, containerd	-	-	-	Απαιτείται η χρήση κατάλληλων drivers (π.χ. Podman Task Driver, Docker Driver)
Υποστηριζόμενα πρότυπα αρχιτεκτονικής plugins των υπηρεσιών δικτύου	CNM (Docker Container Network Model)	CNI (Container Network Interface)	CNI, CNM	CNI, CNM	CNI, CNM	
Υποστηριζόμενα πρότυπα μόνιμων δομών αποθήκευσης	Docker Volume Plugin System	CSI (Common Storage Interface)	Docker Volume Plugin System, CSI	Docker Volume Plugin System	Docker Volume Plugin System, CSI	
Υποστηριζόμενοι Container Engines	Υποστηριζόμενοι από το OCI (π.χ. runC, crun), υποστηριζόμενοι από το CRI (π.χ. containerd, cri-o), Docker Engine	Υποστηριζόμενοι από το OCI, Docker Engine	Docker Engine, Mesos Containerizer	Docker Engine, Mesos Containerizer	Docker Engine, Mesos Containerizer	Docker Engine
Ορισμός πολλαπλών κεντρικών κόμβων υπολογισμού για υψηλή διαθεσιμότητα	✓	✓	✓	✓	✓	✓
Αυτοματοποιημένος ορισμός κεντρικών κόμβων υψηλής διαθεσιμότητας	✓	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	-	✓	✓	-

HTTP API για πρόσβαση στους υπολογιστικούς κόμβους	✓	✓	✓	✓	✓	✓
Client βιβλιοθήκες	✓	✓	✓	✓	✓	✓
Εγκατάσταση						
Εγκατάσταση και χρήση του λογισμικού σε Docker containers	-	✓	✓	✓	-	-
Εγκατάσταση και χρήση του λογισμικού σε virtual machines	-	✓	✓	✓	✓	✓
Εγκατάσταση του λογισμικού από πακέτα Linux μέσω της διεπαφής γραμμής εντολών (Command Line Interface, CLI)	-	✓	✓	✓	✓	✓
Εγκατάσταση μέσω εργαλείων λογισμικού (π.χ. Puppet, Chef)	-	✓	✓	-	-	Δε βρέθηκε σχετική πληροφορία
Εγκατάσταση με χρήση εργαλείων και APIs των cloud providers	✓ (Microsoft Azure)	✓	-	-	✓ (Microsoft Azure)	Δε βρέθηκε σχετική πληροφορία
Εργαλεία εγκατάστασης για τα Microsoft Windows / τον Windows Server	✓	✓	✓	-	-	✓

Πίνακας 2.2: Περιγραφή, Εκτέλεση και Αναβάθμιση Εφαρμογών

Λειτουργία	**SwarmKit**	**Kubernetes**	**Mesos**	**Mesos & Marathon**	**DC/OS**	**Nomad**
Τύπος Εφαρμογών						
Καταλληλότερος τύπος εφαρμογών και εργασιών	Απλές, περιορισμένου μεγέθους cloud-native εφαρμογές	Σύνθετες, υψηλών απαιτήσεων και μεγάλου μεγέθους cloud-native εφαρμογές, batch και cron εργασίες	Εφαρμογές που εκτελούνται για μεγάλο χρονικό διάστημα και σε πολλά αντίγραφα σε πολλαπλά υπολογιστικά μηχανήματα			Σύνθετες, υψηλών απαιτήσεων και μεγάλου μεγέθους cloud-native εφαρμογές, batch και cron εργασίες
Υποστήριξη stateful εφαρμογών	✓	✓	✓			✓
Υποστήριξη stateless εφαρμογών	✓	✓	✓			✓
Υποστήριξη container-based εργασιών	-	✓ (σειριακή ή παράλληλη εκτέλεση, υποστήριξη batch και cron εργασιών)	-	-	✓	✓ (batch και cron εργασίες)
Υποστήριξη κατανεμημένων εφαρμογών πολλαπλών επίπεδων με αλληλοεξαρτήσεις (multi-tier)	✓	- (υποστήριξη χρήσης εξωτερικών εργαλείων π.χ. Kompose, Helm)	-	✓ (Application Groups)	✓	✓
Περιγραφή Εφαρμογών						
Τρόπος περιγραφής	Δηλωτικός	Δηλωτικός	-	Δηλωτικός	Δηλωτικός	Δηλωτικός
Μορφότυπο περιγραφής	YAML	YAML (προτεινόμενο), JSON	-	JSON	JSON	Γλώσσα HashiCorp
Εκτέλεση Εφαρμογών						
Μικρότερη μονάδα εκτέλεσης	Task (αποτελείται από ένα container)	Pod (ομάδα containers που αποτελούν μία λογική μονάδα και εκτελούνται ταυτόχρονα)	Task (αποτελείται από τουλάχιστον ένα container, μπορεί να περιλαμβάνει και μη containerized διαδικασίες)			Task (μπορεί να είναι μία εντολή, ένα container, μία υπηρεσία, μία containerized ή non containerized εφαρμογή)

Υποστήριξη ομαδοποίησης των containers σε Pod	-	✓	✓	✓	✓	-
Υποστήριξη Global Containers	✓ (μέσω των global services)	✓ (μέσω των daemon sets)	-	-	-	-
Μόνιμες Δομές Αποθήκευσης (Persistent Volumes)						
Τοπικές δομές αποθήκευσης: αποτελούμενες από μονάδες δίσκου στον κόμβο εκτέλεσης του container	✓	✓	✓	✓	✓	✓
Ανάθεση των containers σε κόμβο που διαθέτει συγκεκριμένη τοπική δομή αποθήκευσης	-	✓	✓	✓	✓	✓
Μοιραζόμενες δομές μόνιμης αποθήκευσης μεταξύ των containers	✓	✓	✓	-	-	-
Υποστήριξη εξωτερικών δομών μόνιμης αποθήκευσης	✓	✓	✓	✓	✓	✓
Υποστήριξη plugin αρχιτεκτονικών δομών αποθήκευσης	✓ (Docker Engine Plugin)	✓ (CSI υποστηριζόμενα plugins και plugins παρόχων AWS, Microsoft Azure, Google Cloud)	✓ (Docker Engine και CSI υποστηριζόμενα plugins)	✓ (Docker Engine Plugin)	✓ (Docker Engine και CSI υποστηριζόμενα plugins)	✓ (CSI υποστηριζόμενα plugins)
Υποστήριξη εγκατάστασης δομών αποθήκευσης κατά τον χρόνο εκτέλεσης	✓	✓	✓	✓	✓	✓
Υποστήριξη αυτόματης δημιουργίας μόνιμων δομών αποθήκευσης κατά τη δημιουργία ενός container	✓	✓	✓	✓	✓	✓
Κλιμάκωση Εφαρμογών						
Οριζόντια αύξηση/μείωση των containers	✓	✓	-	✓	✓	✓

Αυτόματη αύξηση/μείωση των containers	-	✓	-	- (δυνατή με παρέμβαση του προγραμματιστή ή χρήση εξωτερικών εργαλείων)		✓
Αναβάθμιση Υπηρεσιών Εφαρμογών						
Σταδιακή αναβάθμιση (rolling upgrade) υπηρεσιών κατά την επανεκκίνηση του container με νέο image	✓	✓	-	✓	✓	✓
Καθορισμός των ενεργειών που πραγματοποιούνται κατά τη σταδιακή αναβάθμιση	✓	✓	-	✓	✓	✓
Μηχανισμοί παρακολούθησης της σταδιακής αναβάθμισης	✓	✓	-	✓	✓	✓
Επαναφορά στην προηγούμενη κατάσταση της υπηρεσίας αν αποτύχει μια αναβάθμιση	✓	✓	-	-	✓	✓
Καθορισμός ελέγχων διαθεσιμότητας των υπηρεσιών	-	✓	-	✓	✓	✓ (από την έκδοση Nomad 1.1)
Canary deployments	-	✓	-	-	✓	✓
Αναβάθμιση των ρυθμίσεων μιας υπηρεσίας επιτόπου (in place)	✓	✓	-	-	✓	✓
In place αναβάθμιση μιας υπηρεσίας χωρίς να επηρεάζεται η εκτέλεσή της	✓	✓	-	-	-	✓
Αναβάθμιση του Docker daemon σε ένα κόμβο χωρίς να επηρεαστούν τα containers που εκτελούνται στον κόμβο	✓	✓ (με χρήση του kubeadm)	✓	✓	✓	-

Πίνακας 2.3: Διαχείριση και Παρακολούθηση Εφαρμογών και Συστήματος

Λειτουργία	SwarmKit	Kubernetes	Mesos	Mesos & Marathon	DC/OS	Nomad
Διαχείριση των Εφαρμογών και του Συστήματος						
CLI για πρόσβαση στο API του cluster	✓	✓	✓	✓	✓	✓
Ενσωμάτωση διαδικτυακής διεπαφής χρήστη	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓	✓	✓	✓	✓
Χρήση ετικετών για ομαδοποίηση των API αντικειμένων του συστήματος	✓	✓	✓	✓	✓	✓
Παρακολούθηση Συστήματος						
Διεπαφή γραμμής εντολών και γραφική διεπαφή χρήστη (graphical user interface) για παρακολούθηση της χρήσης πόρων σε όλο το σύστημα	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓	-	-	✓	✓
Κεντρικοί μηχανισμοί παρακολούθησης της χρήσης πόρων από τα containers και τις υπηρεσίες των εφαρμογών	- (υποστήριξη χρήσης εξωτερικών εργαλείων όπως τα InfluxDV, Grafana, cAdvisor κλπ)	✓ (επιπλέον υποστήριξη χρήσης εξωτερικών εργαλείων όπως τα ElasticSearch / Kibana, InfluxDB, Grafana, Sysdig)	✓	-	✓	✓
Κεντρικοί μηχανισμοί παρακολούθησης της χρήσης πόρων από τον μηχανισμό ενορχήστρωσης	- (υποστήριξη χρήσης εξωτερικών εργαλείων όπως το Prometheus)	✓	✓	✓	✓	- (υποστήριξη χρήσης εξωτερικών εργαλείων όπως το Prometheus)

Μηχανισμοί παρακολούθησης της διαθεσιμότητας των containers	✓	✓	✓	✓	✓	-
Μηχανισμοί εντοπισμού νέων αιτημάτων δημιουργίας containers και υπηρεσιών εφαρμογών, αλλαγών στα υπάρχοντα containers και σφαλμάτων	✓	✓	✓	✓	✓	✓
Συντήρηση του Cluster						
Καταγραφή της κατάστασης του cluster για ανάκτησή της στην περίπτωση σφάλματος (backup)	✓	- (δυνατή με χρήση εξωτερικών εργαλείων)	✓	✓	✓	✓
CLI εντολή για την μετακίνηση των containers ενός κόμβου για συντήρησή του	✓	✓	✓	✓	✓	✓
Μηχανισμοί για τη συλλογή και διαγραφή containers και images που δε χρησιμοποιούνται (garbage collection)	✓	✓	✓	-	✓	✓
Καταγραφή Συμβάντων (Logging) και Εντοπισμός Σφαλμάτων (Debugging)						
Υποστήριξη logging λειτουργιών στο επίπεδο των containers μέσω του CLI ή και ενός πίνακα ελέγχου	✓	✓	✓	-	✓	-
Υποστήριξη logging λειτουργιών στο επίπεδο των μηχανισμών του Container Orchestrator	✓	✓	✓	✓	✓	✓

Υποστήριξη συστημάτων συγκέντρωσης και ανάλυσης των καταγεγραμμένων συμβάντων	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓	-	-	✓	✓
Υποστήριξη Πολλαπλών Υποδομών Cloud						
Δυνατότητα δημιουργίας cluster σε πολλαπλές ζώνες διαθεσιμότητας (availability zones)	✓	✓ (περιορισμένες δυνατότητες στην έκδοση ανοιχτού κώδικα, ανάλογα με τον πάροχο των cloud υποδομών)	✓	✓	✓	✓
Δυνατότητα διαχείρισης cluster σε πολλαπλές υποδομές cloud	✓	✓	-	-	✓	✓
Υποστήριξη εκτέλεσης εργασιών σε πολλαπλές ζώνες διαθεσιμότητας	✓	✓	✓	✓	✓	✓

Πίνακας 2.4: Έλεγχος Συστήματος και Εξασφάλιση Ποιότητας Παρεχόμενων Υπηρεσιών

Λειτουργία	SwarmKit	Kubernetes	Mesos	Mesos & Marathon	DC/OS	Nomad
Έλεγχος Πρόσβασης στους Πόρους του Συστήματος από Ομάδες Χρηστών (User Groups)						
Καθορισμός ομάδων χρηστών για πρόσβαση σε υπηρεσίες εφαρμογών, υπολογιστικούς κόμβους, αποθηκευτικές μονάδες κλπ (API objects)	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓	✓	-	✓	- (υποστήριξη namespaces για τον καθορισμό ομάδων εργασιών στις οποίες έχει πρόσβαση ένας χρήστης)
Ορισμός μεγίστου αριθμού API objects στα οποία έχει πρόσβαση μια ομάδα χρηστών	-	✓	✓	-	-	-
Ορισμός Ορίων Χρήσης Πόρων στο Επίπεδο των Containers, Υπηρεσιών, Εφαρμογών						
Ορισμός ορίων χρήσης του επεξεργαστή, της μνήμης και του δίσκου για κάθε ομάδα χρηστών	-	✓ (ορισμός ελάχιστων και μέγιστων ορίων χρήσης του επεξεργαστή και της μνήμης και μέγιστου ορίου για χρήση του σκληρού δίσκου ανά ομάδα πόρων αποθήκευσης)	✓ (ορισμός ελάχιστου ορίου για τον επεξεργαστή, τη μνήμη και τον σκληρό δίσκο και χρήση βαρών για τον καταμερισμό των πόρων)	-	-	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα
Ελάχιστη εγγύηση πρόσβασης στον επεξεργαστή	✓	✓	✓	✓	✓	✓
Ελάχιστη εγγύηση πρόσβασης στη μνήμη	✓	✓	-	-	-	✓
Ορισμός μέγιστου ορίου χρήσης του επεξεργαστή	✓	✓	✓	-	-	✓
Ορισμός μέγιστου ορίου χρήσης της μνήμης	✓	✓	✓	✓	✓	✓
Όρια χρήσης NVDIA GPU	-	-	✓	✓	✓	-
Όρια χρήσης πόρων δίσκου	-	✓ (μόνο για τον τοπικό δίσκο)	✓	✓	✓	✓

Έλεγχος της Ανάθεσης Εργασιών σε Πόρους μέσω Περιορισμών στο Επίπεδο Υπολογιστικών Κόμβων						
Ορισμός ετικετών στους κόμβους για περιγραφή περιορισμών ή/και προτιμήσεων ανάθεσης εργασιών σε αυτούς	✓	✓	✓	✓	✓	✓
Ορισμός συνδυαστικών περιορισμών ή/και προτιμήσεων (π.χ. με χρήση λογικών τελεστών, κανονικών εκφράσεων	✓	✓	-	✓	✓	✓
Λειτουργίες Επανεκκίνησης της Διαδικασίας Ανάθεσης των Containers σε Πόρους για την Εξασφάλιση της Ποιότητας Παρεχόμενων Υπηρεσιών						
Χρήση προτεραιοτήτων ανάθεσης μεταξύ των containers, τερματισμός των containers χαμηλότερης προτεραιότητας στην περίπτωση που ο κόμβος πρέπει να χρησιμοποιηθεί από container υψηλότερης προτεραιότητας	-	✓	-	-	-	✓
Χρήση προτεραιοτήτων μεταξύ των εγγυήσεων παρεχόμενης ποιότητας υπηρεσιών, τερματισμός των containers των υπηρεσιών με χαμηλότερη προτεραιότητα αν οι πόροι του κόμβου στον οποίο έχουν δρομολογηθεί εξαντληθούν	-	✓	-	-	-	✓
Στην περίπτωση αποτυχίας ενός υπολογιστικού κόμβου, ανάθεση των containers του σε άλλον	✓	✓	✓	✓	✓	✓
Διαχείριση του κύκλου ζωής των containers ως αλληλουχία καταστάσεων	✓	✓	✓	✓	✓	✓
Αναδιανομή των containers στους διαθέσιμους υπολογιστικούς κόμβους σε περιπτώσεις διαταραχής της ισορροπίας	✓	-	-	-	-	-

Πίνακας 2.5: Παραμετροποίηση και Επεκτασιμότητα

Λειτουργία	SwarmKit	Kubernetes	Mesos	Mesos & Marathon	DC/OS	Nomad
Στο Επίπεδο των Containers						
Παραμετροποίηση των μεταβλητών περιβάλλοντος (environment variables)	✓	✓	✓	✓	✓	✓
Διαχωρισμός των ρυθμίσεων και των δεδομένων παραμετροποίησης από την container image	✓	✓	-	-	-	-
Δυνατότητα καθορισμού της εντολής που θα εκτελεστεί κατά την έναρξη μιας Docker image και των ορισμάτων της στο χρόνο εκτέλεσης	✓	✓	✓	✓	✓	✓
Δυνατότητα υποστήριξης πολλαπλών τύπων containers και images	✓	✓	✓	✓	✓	-
Συστήματος						
Υποστήριξη plugins για την επέκταση των μηχανισμών ενορχήστρωσης	✓	✓	✓	✓	✓	-
Υποστήριξη plugins για επέκταση των μηχανισμών και των αλγορίθμων ανάθεσης εργασιών σε πόρους	-	✓	✓	✓	✓	✓

Πίνακας 2.6: Διασύνδεση

Λειτουργία	SwarmKit	Kubernetes	Mesos	Mesos & Marathon	DC/OS	Nomad
Επικοινωνία μεταξύ των Υπηρεσιών (Services)						
Αναγνωριστικό υπηρεσίας: μια συγκεκριμένη θύρα στους κόμβους στους οποίους εκτελείται Αναγνωριστικό container: συνδυασμός της διεύθυνσης IP και μιας θύρας του κόμβου στον οποίο τρέχει Load Balancer: εξυπηρετεί τα αιτήματα μιας υπηρεσίας σε έναν ή περισσότερους κόμβους του cluster	✓	✓	✓	✓	✓	Απαιτείται χρήση του Consul για τη διασύνδεση των υπηρεσιών
Αναγνωριστικό υπηρεσίας: ένα στατικό όνομα DNS ή μία διεύθυνση IP Αναγνωριστικό container: η διεύθυνση IP ενός εικονικού δικτύου (virtual network) Load Balancer: για τα IPs των υπηρεσιών Εσωτερικός μηχανισμός: αντιστοιχίζει κάθε όνομα DNS στην IP διεύθυνση μιας υπηρεσίας αν υπάρχει, αλλιώς σε λίστα με τις IP διευθύνσεις των containers της υπηρεσίας	✓	✓	✓ (χωρίς υποστήριξη στατικού ονόματος DNS ως αναγνωριστικό μιας υπηρεσίας)	✓ (χωρίς υποστήριξη στατικού ονόματος DNS ως αναγνωριστικό μιας υπηρεσίας)	✓	
Αναγνωριστικό υπηρεσίας: ένα στατικό όνομα DNS Οι θύρες των containers γίνονται διαθέσιμες μέσω μίας κεντρικής θύρας Εσωτερικός μηχανισμός: αποθηκεύονται οι IP διευθύνσεις των κόμβων στους οποίους έχουν δρομολογηθεί τα containers των υπηρεσιών, σε κάθε αίτημα επιστρέφεται μία διεύθυνση με round-robin τρόπο	✓	✓ (χωρίς υποστήριξη στατικού ονόματος DNS ως αναγνωριστικό μιας υπηρεσίας)	✓	✓	✓	

Ανάθεση Θυρών (Ports)						
Υποστήριξη δυναμικής ανάθεσης θυρών	✓	-	✓	✓	✓	✓
Διαχείριση συγκρούσεων μεταξύ στατικά προσδιορισμένων θυρών στον ίδιο κόμβο	✓	✓	-	-	-	-
Εντοπισμός Υπηρεσιών και Προσβασιμότητα						
Εσωτερικός μηχανισμός DNS	✓ (κατανεμημένη)	✓ (κεντρική)	✓ (κεντρική)	✓ (κεντρική)	✓ (κατανεμημένη και κεντρική)	Απαιτείται χρήση του Consul για τον εντοπισμό των υπηρεσιών
Εξωτερική πρόσβαση στις υπηρεσίες μέσω του πλέγματος δρομολόγησης	✓	✓	✓	✓	✓	
Μία υπηρεσία μπορεί να προσδιορίζεται τόσο από την IP διεύθυνσή της όσο και από το port που χρησιμοποιεί	✓	✓	-	-	-	

Πίνακας 2.7: Ασφάλεια

Λειτουργία	SwarmKit	Kubernetes	Mesos	Mesos & Marathon	DC/OS	Nomad
Ασφάλεια Συστήματος						
Ταυτοποίηση χρηστών μέσω του API του manager κόμβου	✓	✓	✓	✓	✓	✓
Εξουσιοδότηση χρηστών μέσω του API του manager κόμβου	✓	✓	✓	✓	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	Δε βρέθηκε σχετική πληροφορία
Ταυτοποίηση των workers κόμβων μέσω του API του manager κόμβου	✓	✓	✓	-	✓	Δε βρέθηκε σχετική πληροφορία
Κωδικοποίηση των μηνυμάτων ελέγχου	✓	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	-	-	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓
Κωδικοποίηση των ανταλλασσόμενων μηνυμάτων μεταξύ των εφαρμογών	✓	✓	-	-	Δεν υποστηρίζεται στην έκδοση ανοιχτού κώδικα	✓
Περιορισμός της εξωτερικής πρόσβασης στις θύρες των υπηρεσιών	✓	✓	-	-	✓	Δε βρέθηκε σχετική πληροφορία
Ασφάλεια Containers						
Προστασία των ευαίσθητων δεδομένων και των ιδιόκτητων λογισμικών στα containers ως ιδιωτικά κλειδιά	✓	✓	✓	✓	✓	✓
Χρήση Docker container images από ιδιωτικά αποθετήρια εικόνων (image registry)	✓	✓	✓	✓	✓	✓
Υποστηριζόμενοι μηχανισμοί απομόνωσης των λειτουργιών κάθε container από τις άλλες λειτουργίες του κόμβου στον οποίο εκτελείται	-	✓ (ορισμός Linux δυνατοτήτων, SELinux ετικετών, AppArmor profiles, seccomp profiles ανά container)	-	-	-	Δε βρέθηκε σχετική πληροφορία

Κεφάλαιο 3: Επισκόπηση των Εργαλείων Ενορχήστρωσης

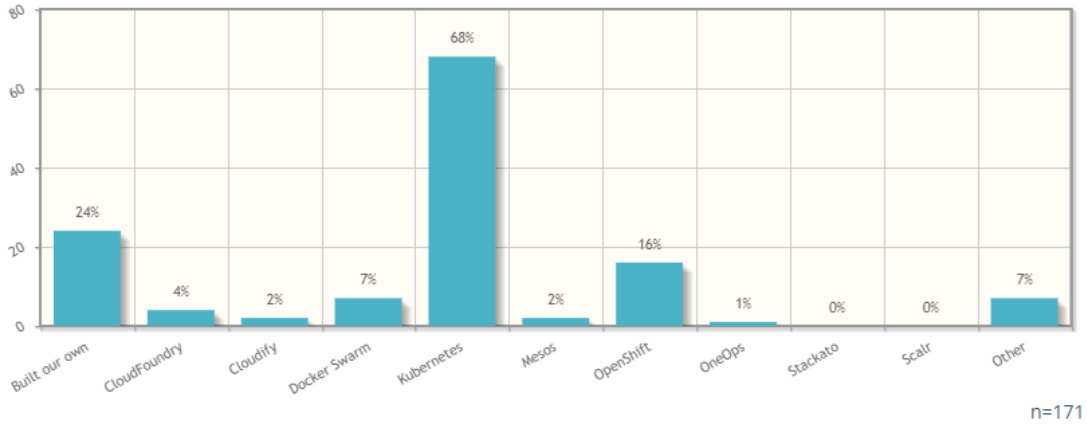
3.1 Κριτήρια Επιλογής

Ύστερα από τη σύγκριση των κυριότερων λειτουργιών των πιο διαδεδομένων ανοιχτού κώδικα εργαλείων ενορχήστρωσης εγγενών εφαρμογών υπολογιστικού νέφους όπως παρουσιάστηκε στην Ενότητα 2.3, επιλέχθηκαν για περαιτέρω μελέτη τα SwarmKit και Kubernetes. Η επιλογή αυτή έγινε με κριτήρια την ετερογένεια, την προτυποποίηση και τον βαθμό χρήσης των τεχνολογιών ενορχήστρωσης που εξετάστηκαν στο σύγχρονο επιχειρησιακό περιβάλλον.

Πρώτον, ως προς την ετερογένεια αξίζει να σημειωθούν τα παρακάτω. Βάσει της αναφοράς [26], σημαντικές διαφορές στους υποστηριζόμενους μηχανισμούς παρουσιάζονται μεταξύ των SwarmKit και Kubernetes, των SwarmKit και DC/OS, του Kubernetes και του Mesos, καθώς και του Kubernetes και του Mesos με χρήση του Marathon. Αντίθετα, δεν εμφανίζεται σημαντική διαφοροποίηση μεταξύ του Kubernetes και του DC/OS. Επίσης, το SwarmKit αποτελεί το λιγότερο γενικευμένο εργαλείο ενώ το Kubernetes αυτό που υποστηρίζει αποκλειστικά τις περισσότερες λειτουργίες και παρέχει πληθώρα δυνατοτήτων επέκτασής τους.

Δεύτερον, ως προς την προτυποποίηση όπως παρουσιάστηκε στην υπό ενότητα Γενική Αρχιτεκτονική και Εγκατάσταση, τα SwarmKit και Kubernetes υποστηρίζουν τα περισσότερα πρότυπα διαχείρισης containers, plugins υπηρεσιών δικτύου και μόνιμων δομών αποθήκευσης. Η υποστήριξη διαδεδομένων προτύπων είναι ενδεικτική της ωριμότητας των σχετικών λειτουργιών και περιορίζει τις πιθανότητες κατάργησής τους σε μελλοντικές εκδόσεις του εργαλείου. Κατά αυτόν τον τρόπο, διευκολύνεται η συντήρηση του συστήματος και του λογισμικού που έχει αναπτυχθεί σε βάθος χρόνου.

Τρίτον, δεδομένης της ταχείας ανάπτυξης που παρατηρείται σε αυτές τις τεχνολογίες, είναι σύνηθες τα εργαλεία που δεν ικανοποιούν πλέον τις ανάγκες των εφαρμογών να αντικαθίστανται από νέα σε σύντομο χρονικό διάστημα, δημιουργώντας προβλήματα στις ομάδες ανάπτυξης που τα χρησιμοποιούν. Για αυτόν τον λόγο, καθίστανται αναγκαία η επιλογή ευρέως διαδεδομένων εργαλείων με μεγάλη κοινότητα χρηστών. Στην Εικόνα 3.1 απεικονίζονται τα στατιστικά του OpenStack User Survey 2021 για τα PaaS εργαλεία που χρησιμοποιούνται για τη διαχείριση εφαρμογών σε containers [31], με το Kubernetes να αποτελεί την πιο διαδεδομένη πλατφόρμα.



Εικόνα 3.1: Στατιστικά χρήσης PaaS εργαλείων διαχείρισης εφαρμογών

3.2 Αρχιτεκτονική

3.2.1 Docker SwarmKit και Swarm Mode

Όπως προαναφέρθηκε, το SwarmKit είναι μία πλατφόρμα ανοιχτού κώδικα του Docker για την εκτέλεση εργασιών σε συμπλέγματα κόμβων υπολογισμού (clusters). Αποτελεί διάδοχο του Docker Classic Swarm. Πιο συχνά στη βιβλιογραφία γίνεται αναφορά στο swarm mode, ένα σύνολο ενσωματωμένων λειτουργιών στο Docker Engine, οι οποίες χρησιμοποιούν βιβλιοθήκες του SwarmKit για τη χρονοδρομολόγηση και ενορχήστρωση εφαρμογών σε containers. Η ενσωμάτωση του swarm mode στο Docker Engine έγινε από την έκδοση 1.12.

Ένα σύνολο υπολογιστικών κόμβων με Docker Engines σε swarm mode ομαδοποιούνται σε ένα swarm για την ανάπτυξη, εκτέλεση και διαχείριση εφαρμογών σε μεγάλη κλίμακα [32]. Μεταξύ άλλων, η πλατφόρμα Docker SwarmKit, συμπεριλαμβανομένων των λειτουργιών swarm mode, παρέχει δυνατότητες κλιμάκωσης των εφαρμογών και λειτουργίες για την ασφαλή επικοινωνία των εκτελούμενων υπηρεσιών και την ανταλλαγή δεδομένων. Επιπλέον, διαθέτει μηχανισμούς αποκατάστασης σφαλμάτων ώστε να διατηρηθεί η επιθυμητή κατάσταση του συστήματος σε περίπτωση αποκλίσεων. Η πλατφόρμα SwarmKit υποστηρίζει ακόμη τη δικτύωση πολλαπλών κόμβων υπολογισμού (multi-host networking) όταν απαιτούνται περισσότεροι υπολογιστικοί πόροι από αυτούς που είναι διαθέσιμοι σε έναν κόμβο. Επίσης, διαθέτει μηχανισμούς εντοπισμού των εκτελούμενων εργασιών (service discovery) και εξισορρόπησης και διαχείρισης του εσωτερικού και εξωτερικού φορτίου του συστήματος (load balancing). Τέλος, η πλατφόρμα SwarmKit υποστηρίζει παραμετροποιήσιμες κυλιόμενες ενημερώσεις (rolling updates), με δυνατότητα επαναφοράς του συστήματος σε προηγούμενη κατάσταση αν προκληθεί σφάλμα [33].

Ένα swarm περιλαμβάνει δύο τύπους υπολογιστικών κόμβων, τον manager και τους workers, ακολουθώντας το μοντέλο ελέγχου και επικοινωνίας master-slave όπως φαίνεται στην Εικόνα 3.2. Οι

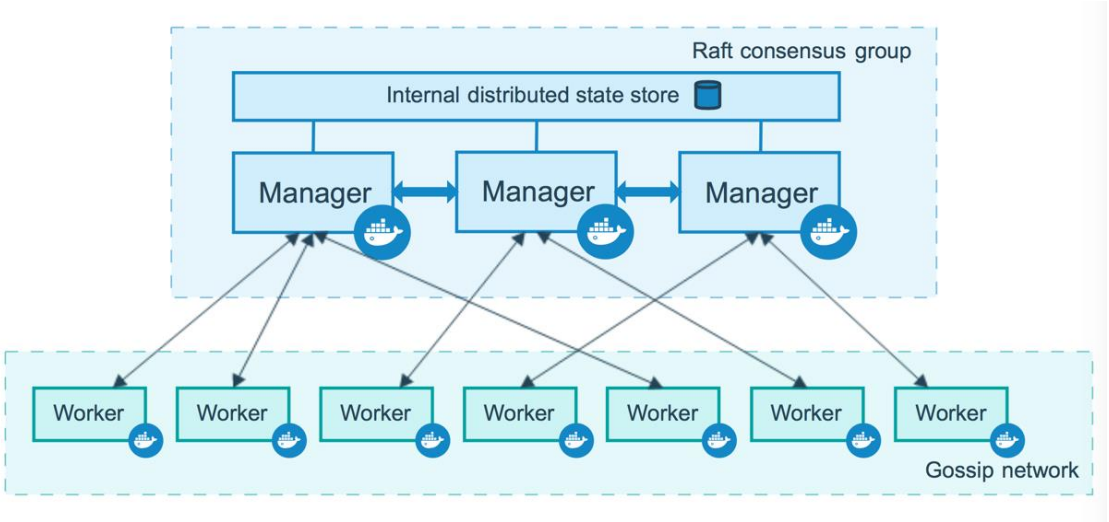
κόμβοι μπορούν να βρίσκονται σε φυσικά ή εικονικά μηχανήματα, σε υποδομές των χρηστών ή υπολογιστικού νέφους. Εξ ’ορισμού κάθε manager λειτουργεί και ως worker, επιτρέποντας τη δημιουργία ενός swarm με ένα μόνο κόμβο υπολογισμού. Η συγκεκριμένη τοπολογία προτιμάται κυρίως για τον έλεγχο του συστήματος και των εφαρμογών. Ένας κόμβος worker μπορεί να προαχθεί σε manager και αντιστρόφως [34].

Πιο συγκεκριμένα, από τη μία πλευρά οι workers κόμβοι είναι υπεύθυνοι για την εκτέλεση των εφαρμογών των χρηστών. Η μικρότερη μονάδα χρονοδρομολόγησης στο swarm είναι το task. Στην τρέχουσα έκδοση του SwarmKit, κάθε task αποτελείται από ένα Docker container και τις προς εκτέλεση σε αυτό εντολές. Έτσι, κάθε εργασία μιας εγγενούς εφαρμογής υπολογιστικού νέφους περιγράφεται ως μία ομάδα από tasks, το service, ή περισσότερες. Μελλοντικά, η έννοια του task στο SwarmKit αναμένεται να επεκταθεί ώστε να υποστηρίζεται η εκτέλεση και άλλων τύπων εργασιών, πέρα από αυτές που περιγράφονται ως σύνολα από containers [35]. Αφού οι χρήστες υποβάλλουν στο swarm μια εφαρμογή για εκτέλεση, όπως θα αναλυθεί στην Ενότητα 3.3.1, οι workers λαμβάνουν σειριακά τα tasks των εργασιών της από τον manager και τα εκτελούν ανεξάρτητα. Ειδικό λογισμικό σε κάθε worker, ο agent, ενημερώνει τον manager για την κατάσταση των tasks που του έχουν ανατεθεί για εκτέλεση.

Από την άλλη πλευρά, ο manager είναι υπεύθυνος για τη συνολική διαχείριση των υπολογιστικών κόμβων του cluster και την ενορχήστρωση της εκτέλεσης των εφαρμογών. Μεταξύ άλλων, ο manager εξυπηρετεί τα αιτήματα των χρηστών και των εξωτερικών συστημάτων στα HTTP API endpoints του swarm και δρομολογεί τα tasks των υποβληθέντων εφαρμογών στους workers. Είναι αρμόδιος επίσης για τη διατήρηση της επιθυμητής κατάστασης των εφαρμογών, όπως έχει προσδιοριστεί από τους χρήστες. Αξίζει να σημειωθεί ότι οι μηχανισμοί ενορχήστρωσης και χρονοδρομολόγησης του manager δε γνωρίζουν το είδος των tasks που έχουν υποβληθεί για εκτέλεση, σχεδιαστική απόφαση που θα διευκολύνει τη μελλοντική επέκταση του swarm mode και σε άλλους τύπους εργασιών πέρα από αυτές που περιγράφονται ως σύνολα από containers. Ο manager επικοινωνεί ακόμη με μία εσωτερική κατανεμημένη βάση δεδομένων στην οποία αποθηκεύονται πληροφορίες για την κατάσταση του swarm και των εκτελούμενων υπηρεσιών [32].

Για μεγαλύτερη ανοχή σε σφάλματα, προτείνεται η δημιουργία πολλαπλών managers κόμβων. Ωστόσο, σε αυτή την περίπτωση, επιλέγεται μόνο ένας ως υπεύθυνος για τις λειτουργίες ενορχήστρωσης. Σε swarm mode, οι managers υλοποιούν τον αλγόριθμο Raft consensus [36] για να διασφαλιστεί ότι γνωρίζουν την ίδια κατάσταση του συστήματος και των εκτελούμενων εργασιών. Σε περίπτωση σφάλματος, ο Raft επιτρέπει σε οποιονδήποτε manager να αποκαταστήσει τα tasks και τα services που απέτυχαν ή διακόπηκαν. Για N managers, ο αλγόριθμος μπορεί να εντοπίσει μέχρι $\frac{N-1}{2}$ σφάλματα, ενώ για την αναγνώριση μιας κατάστασης ως εσφαλμένη χρειάζεται να συμφωνήσουν τουλάχιστον $\frac{N}{2}+1$ από τους managers. Η δημιουργία πολλαπλών managers συμβάλει στην

αποκατάσταση σφαλμάτων με περιορισμένο χρόνο διακοπής λειτουργίας. Ωστόσο, η πλεονάζουσα αύξηση του αριθμού τους δυσχεραίνει την απόδοση και την κλιμάκωση του swarm [37].



Εικόνα 3.2: Τα συστατικά στοιχεία ενός swarm

3.2.2 Kubernetes

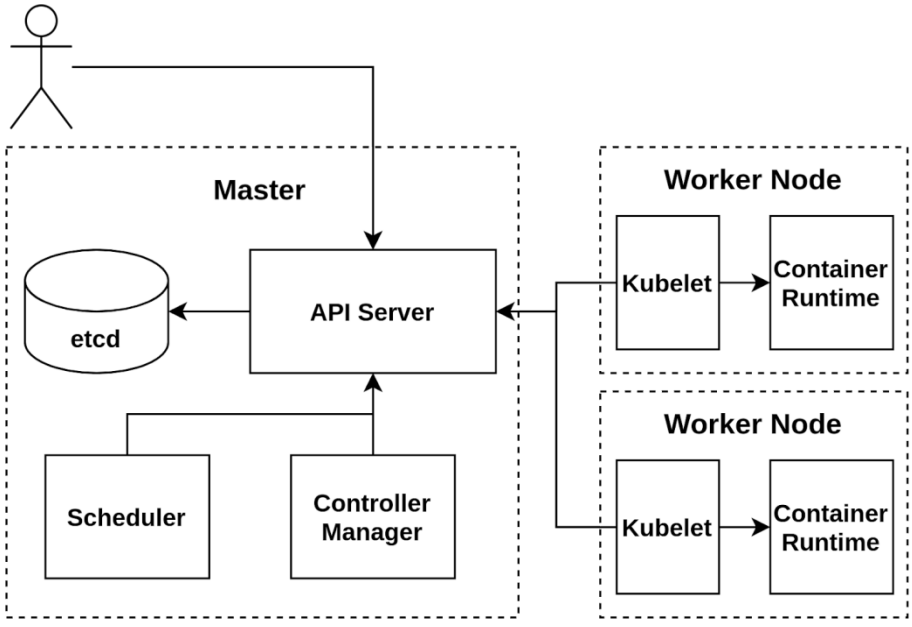
Το Kubernetes (K8s) είναι ένα σύστημα ανοιχτού κώδικα που αναπτύχθηκε αρχικά από την Google για την αυτοματοποίηση της εκτέλεσης, διαχείρισης και κλιμάκωσης εφαρμογών σε συμπλέγματα υπολογιστικών πόρων. Μεταξύ άλλων, η πλατφόρμα υποστηρίζει επεκτάσιμους αλγορίθμους χρονοδρομολόγησης των εφαρμογών και ευέλικτες λειτουργίες ενορχήστρωσης των containers τους σε κατανεμημένες υποδομές, λαμβάνοντας υπόψη τις προτιμήσεις των χρηστών. Διαθέτει ακόμα μηχανισμούς εντοπισμού των εκτελούμενων εργασιών, ασφαλούς διαχείρισης και ανταλλαγής δεδομένων μεταξύ τους και εξισορρόπησης φορτίου. Επίσης, παρέχει δυνατότητες αυτοματοποίησης των ενημερώσεων, επαναφοράς του διαχειριζόμενου υλικού και λογισμικού σε προηγούμενες καταστάσεις σε περίπτωση σφάλματος και αποκατάστασης σφαλμάτων [38]. Για την υποστήριξη αυτών των λειτουργιών, το Kubernetes ακολουθεί το μοντέλο ελέγχου και επικοινωνίας master-slave.

Κάθε διαχειριζόμενο από το Kubernetes σύμπλεγμα υπολογιστικών πόρων αποτελείται από τους workers και το Control Plane. Πιο συγκεκριμένα, σε κάθε worker κόμβο εκτελούνται οι υποβληθείσες από τους χρήστες εφαρμογές ως σύνολα από Pods. Το Pod είναι η μικρότερη μονάδα εκτέλεσης στο Kubernetes και συνίσταται από ένα ή περισσότερα containers, τα οποία δρομολογούνται από κοινού για εκτέλεση σε έναν υπολογιστικό κόμβο, ο οποίος μπορεί να είναι ένας server ή ένα εικονικό μηχάνημα. Με αυτόν τον τρόπο, τα containers του Pod διαμοιράζονται τους αποθηκευτικούς και δικτυακούς πόρους του κόμβου και έτσι διευκολύνεται η μεταξύ τους επικοινωνία και ανταλλαγή δεδομένων [6].

Για τον συντονισμό της εκτέλεσης των εφαρμογών, κάθε worker διαθέτει τους μηχανισμούς Kubelet, Network Proxy και container runtime. Το Kubelet λαμβάνει τις προδιαγραφές κάθε Pod από το Control Plane και εξασφαλίζει ότι τα containers που το αποτελούν είναι υγιή και εκτελούνται βάσει

των προδιαγραφών των χρηστών. Το Network Proxy είναι υπεύθυνο για τη διαχείριση και των έλεγχο των αιτημάτων άλλων κόμβων ή εξωτερικών συστημάτων στα Pods που έχουν δρομολογηθεί στον worker. Τέλος, το container runtime εκτελεί τα containers των Pods [39].

Στους υπολογιστικούς κόμβους με ρόλο Control Plane εκτελούνται οι μηχανισμοί API Server, Scheduler, Controller Manager και προαιρετικά ο Cloud Control Manager για τον ορισμό και την εκτέλεση των εφαρμογών και την ενορχήστρωση και διαχείριση των containers που τις συνιστούν. Πρώτον, ο API Server, εξυπηρετώντας τα REST αιτήματα στο API των μηχανισμών του Kubernetes (Kubernetes API), αποτελεί το σημείο αλληλεπίδρασης των χρηστών με τους υπολογιστικούς κόμβους του συμπλέγματος αλλά και των μηχανισμών του συστήματος μεταξύ τους, όπως φαίνεται στην Εικόνα 3.3 [40]. Χάρει στον σχεδιασμό του, διευκολύνεται η δυναμική ανανέωση του Kubernetes API, βάσει των αλλαγών στον τρόπο περιγραφής των πόρων του συστήματος. Ο API Server εντοπίζει επίσης τις αλλαγές που πραγματοποιούνται στο σύστημα και είναι υπεύθυνος για την ενημέρωση των μηχανισμών που επηρεάζονται από αυτές. Ακόμη, διαχειρίζεται τις λειτουργίες ταυτοποίησης και εξουσιοδότησης για την πρόσβαση των χρηστών στους πόρους υλικού και λογισμικού του συστήματος. Αφού ένα αίτημα χρήστη ταυτοποιηθεί και εξουσιοδοτηθεί επιτυχώς, ο API Server εκτελεί μηχανισμούς απόρριψης ή αποδοχής του αιτήματος, τους Admission controllers.

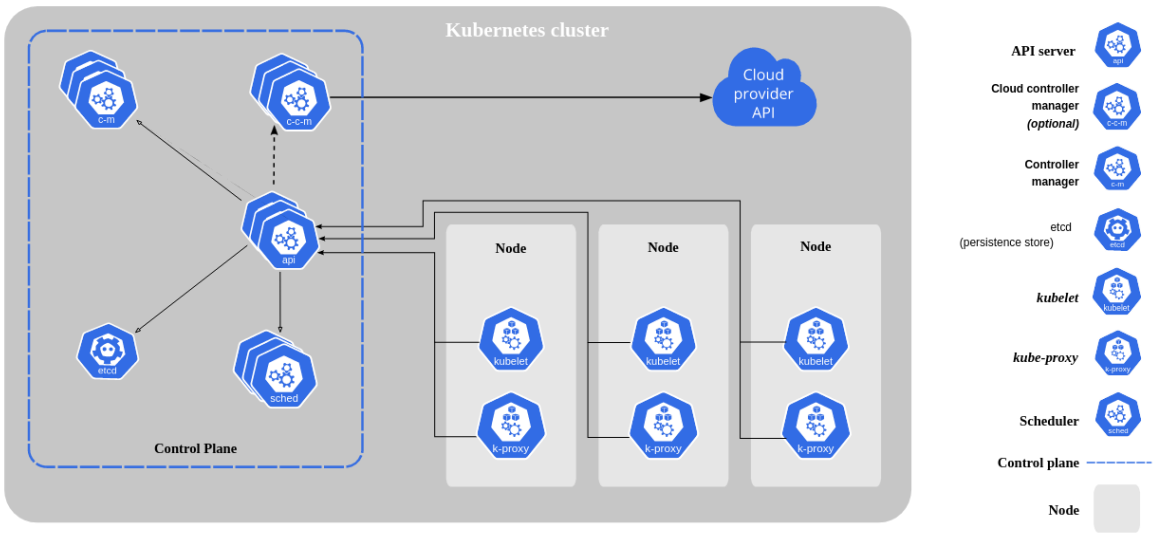


Εικόνα 3.3: Η επικοινωνία μεταξύ των μηχανισμών του Kubernetes

Δεύτερον, ο Scheduler εντοπίζει τα Pods τα οποία δεν έχουν δρομολογηθεί για εκτέλεση και τα αναθέτει σε υπολογιστικούς κόμβους του συμπλέγματος, όπως θα αναλυθεί περαιτέρω στην Ενότητα 4.1.2. Τρίτον, ο μηχανισμός Controller Manager του Control Plane παρακολουθεί τα διάφορα μέρη του συστήματος, όπως τους υπολογιστικούς κόμβους, τις υποβληθείσες εργασίες και τις διεπαφές, και πραγματοποιεί αλλαγές ώστε να διασφαλιστεί η επιθυμητή κατάσταση των πόρων υλικού και

λογισμικού και των εφαρμογών. Συνεχίζοντας, ο μηχανισμός Cloud Control Manager χρησιμοποιείται στο Control Plane μόνο όταν το Kubernetes εκτελείται σε υποδομές υπολογιστικού νέφους. Ο Cloud Control Manager συνδέει το σύμπλεγμα υπολογιστικών πόρων με το API του παρόχου, επιτρέπει την αλληλεπίδραση με τους πόρους του και εκτελεί τους μηχανισμούς ελέγχου του. Τέλος, για την αξιόπιστη διαχείριση της κατάστασης του συστήματος και τη διατήρηση αντιγράφων των δεδομένων υπάρχει η ανάγκη ύπαρξης μηχανισμού μόνιμης αποθήκευσης στο Control Plane. Η προτεινόμενη ενσωματωμένη λύση στο Kubernetes είναι η κατανεμημένη βάση δεδομένων. Στην etcd τα δεδομένα αποθηκεύονται σε ζεύγη κλειδιού-τιμής (key-value). Ο μόνος μηχανισμός του συστήματος που επικοινωνεί με τη βάση δεδομένων είναι ο API Server. Για καλύτερη ανοχή σε σφάλματα και υψηλή διαθεσιμότητα, οι μηχανισμοί του Control Plane εκτελούνται συνήθως σε πολλαπλά υπολογιστικά μηχανήματα.

Η Εικόνα 3.4 [39] συνοψίζει τους μηχανισμούς ενός συμπλέγματος υπολογιστικών κόμβων διαχειριζόμενου από το Kubernetes. Συνήθως, στο πλαίσιο εκμάθησης του Kubernetes και σε περιβάλλοντα ελέγχου (testing environments) οι κόμβοι υπολογισμού βρίσκονται σε τοπικά μηχανήματα, ενώ στο παραγωγικό περιβάλλον (production environment) σε υποδομές υπολογιστικού νέφους και σε on-premise data centers. Επίσης, επί πληρωμή υπηρεσίες Kubernetes, όπως τα Amazon Elastic Kubernetes Service (Amazon EKS), Azure Kubernetes Service (AKS) και Google Kubernetes Engine (GKE), μπορούν να χρησιμοποιηθούν για πιο αποδοτική διαχείριση των υποδομών, βελτιωμένη διαθεσιμότητα, μικρότερα λειτουργικά κόστη και ταχύτερη ανάπτυξη [41].



Εικόνα 3.4: Οι μηχανισμοί ενός συμπλέγματος υπολογιστικών κόμβων διαχειριζόμενου από το Kubernetes

3.3 Περιγραφή και Δημιουργία Αιτήματος Εκτέλεσης Εφαρμογής

Κάθε εργαλείο διαχείρισης και ενορχήστρωσης εφαρμογών, ανάλογα με την αρχιτεκτονική και τα σχεδιαστικά πρότυπα ανάπτυξής του, αλλά και τους τύπους των εφαρμογών των οποίων τις ανάγκες

στοχεύει να ικανοποιήσει βέλτιστα, ακολουθεί διαφορετικές συμβάσεις και παρέχει στους χρήστες διαφορετικές δυνατότητες για την περιγραφή των εφαρμογών. Όπως θα γίνει κατανοητό στις επόμενες υποενότητες, ο τρόπος περιγραφής των εγγενών εφαρμογών υπολογιστικού νέφους διαφέρει σημαντικά στα Docker SwarmKit και Kubernetes και αν και πολλές φορές φαινομενικά χρησιμοποιούνται ίδιοι όροι, οι λειτουργίες και οι μηχανισμοί που περιγράφονται από αυτούς δεν είναι οι ίδιοι. Οι διαφορές αυτές χρειάζεται να μελετηθούν εκτενέστερα και να ενταθούν οι προσπάθειες για τη δημιουργία κοινών συμβάσεων περιγραφής, ώστε να επιτευχθεί μελλοντικά η ανάπτυξη και εκτέλεση εφαρμογών σε ετερογενή περιβάλλοντα τα οποία ενδεχομένως συνδυάζουν ετερογενή εργαλεία και πλατφόρμες ενορχήστρωσης.

3.3.1 Docker SwarmKit

Η πλατφόρμα Docker SwarmKit παρέχει ένα δηλωτικό τρόπο περιγραφής μιας εγγενούς εφαρμογής υπολογιστικού νέφους και της επιθυμητής κατάστασής της σε όλο τον κύκλο ζωής της. Πιο συγκεκριμένα, για την περιγραφή των εφαρμογών ως σύνολα από containers και των εξαρτήσεών τους με άλλα συστήματα, ανεξάρτητα από τις υποκείμενες πλατφόρμες υλικού και λογισμικού, χρησιμοποιείται το εργαλείο Docker Compose. Σύμφωνα με αυτό, οι χρήστες περιγράφουν κάθε εφαρμογή σε ένα ή περισσότερα αρχεία κειμένου YAML. Κάθε αρχείο YAML συνίσταται από τα επιμέρους πεδία `services`, `networks`, `volumes`, `configs` και `secrets`.

Τα υπολογιστικά στοιχεία μιας εφαρμογής περιγράφονται στο πεδίο `services`. Για την περίπτωση των εγγενών εφαρμογών υπολογιστικού νέφους, κάθε service αντιστοιχεί συνήθως σε μία μικροϋπηρεσία [42], όπως περιγράφηκε στην Ενότητα 1.1.1, και αποτελείται από ένα ή περισσότερα tasks, όπως αναφέρθηκε στην Ενότητα 3.2.1. Σε κάθε αρχείο Docker Compose ορίζεται υποχρεωτικά το πεδίο `services`. Κάτω από αυτό, το όνομα κάθε service της εφαρμογής αντιστοιχίζεται στον ορισμό του, δηλαδή ένα σύνολο ρυθμίσεων και περιορισμών για τη δημιουργία, την εκτέλεση, την κλιμάκωση και τον τερματισμό της λειτουργίας των containers που το συνιστούν [43].

Πιο συγκεκριμένα, για την δημιουργία και εκτέλεση των containers που συνιστούν ένα service, ο χρήστης προσδιορίζει στον ορισμό του μία Docker image και ορίσματα εκτέλεσης. Συνεχίζοντας, στον ορισμό ενός service, ομαδοποιούνται οι περιορισμοί και οι απαιτήσεις που περιγράφουν την ανάθεση πόρων, τις λειτουργίες του service ως λογική οντότητα, την παραμετροποίησή του και τη διασύνδεσή του με άλλα περιγραφόμενα services (π.χ. τα υποπεδία `deploy`, `runtime`, `configs`, `networks`, `extends`, `depends_on`). Επίσης, οι περιορισμοί και οι απαιτήσεις μπορούν να αφορούν τις επιμέρους λειτουργίες, την παραμετροποίηση και τη διασύνδεση των containers του service (π.χ. τα υποπεδία `image`, `command`, `labels`, `ports`, `restart`). Τέλος, στον ορισμό ενός service μπορούν να περιγράφονται οι διαθέσιμοι πόροι υλικού και η παραμετροποίησή τους (π.χ. τα υποπεδία `cpu_count`, `cpuset`, `mem_swappiness`) [44]. Το μοντέλο ορισμού των services με το Docker Compose είναι δηλωτικό, δεδομένου ότι ο χρήστης περιγράφει την επιθυμητή κατάστασή του σε όλα

τα στάδια του κύκλου ζωής του και η πλατφόρμα του Docker είναι υπεύθυνη για την αυτοματοποιημένη διατήρηση της επιθυμητής αυτής κατάστασης [45].

Αξίζει να σημειωθεί ακόμη ότι σε ένα swarm, τα εκτελούμενα containers μπορεί να μην ανήκουν σε κάποιο service, αλλά να έχουν οριστεί και να εκτελούνται αυτόνομα (standalone containers). Μία ειδοποιός διαφορά μεταξύ των αυτόνομων containers και αυτών των services είναι ότι τα αυτόνομα containers μπορούν να εκτελούνται τόσο στους managers όσο και σε workers κόμβους του swarm, ενώ μόνο οι managers κόμβοι υπολογισμού μπορούν να διαχειρίζονται τα containers των services. Ωστόσο, ένα σημαντικό πλεονέκτημα της ομαδοποίησης των containers σε services είναι ότι μπορεί να τροποποιηθεί η παραμετροποίηση ενός service και να ανανεωθούν αντίστοιχα τα containers του χωρίς την παρέμβαση του χρήστη [43]. Κατά την ανανέωση ενός service, το Docker σταματάει την εκτέλεση των containers του και τα εκκινεί ξανά με τη νέα παραμετροποίηση [45].

Σε ένα YAML αρχείο του Docker Compose μπορεί να εμφανίζεται επίσης το πεδίο `networks`. Στο πεδίο αυτό προσδιορίζονται οι χαμηλού επιπέδου, συγκεκριμένες για κάθε πλατφόρμα, παράμετροι δικτύωσης που επιτρέπουν την επικοινωνία των services μεταξύ τους και με εξωτερικά συστήματα. Η διασύνδεση των containers ενός service με άλλα ή με εξωτερικούς του συστήματος πόρους γίνεται μέσω μια IP σύνδεσης. Ένα service μπορεί να συνδεθεί σε ένα δίκτυο, προσδιορίζοντας στο υποπεδίο `networks` του ορισμού του το όνομα του δικτύου.

Στο YAML αρχείο μπορεί να οριστεί το πεδίο `volumes` για την περιγραφή του συγκεκριμένου για κάθε πλατφόρμα τρόπου μόνιμης αποθήκευσης και διαμοιρασμού δεδομένων. Οι περιγραφόμενες σε αυτό το πεδίο δομές αποθήκευσης μπορούν να χρησιμοποιούνται και να διαμοιράζονται από πολλαπλά services [44].

Συνεχίζοντας, η παραμετροποίηση ενός service μπορεί να καθορίζεται στον χρόνο εκτέλεσης ή να εξαρτάται από τα επιμέρους χαρακτηριστικά κάθε πλατφόρμας. Το πεδίο `configs` χρησιμοποιείται για τον προσδιορισμό μη ευαίσθητων πληροφοριών παραμετροποίησης ως σύνολα δυαδικών ή μη συμβολοσειρών, οι οποίες μπορεί να βρίσκονται σε κάποιο εξωτερικό αρχείο. Μπορούν να χρησιμοποιηθούν συνδυαστικά με μεταβλητές περιβάλλοντος (environment variables) ή ετικέτες (labels). Οι πληροφορίες αυτές αποθηκεύονται ως ένα νέο αρχείο στο σύστημα αρχείων των containers του service, δηλαδή εκτός της Docker image του. Έτσι, οι Docker images διατηρούνται γενικές και δε χρειάζεται να δημιουργηθούν ξανά σε περίπτωση αλλαγών στην παραμετροποίηση του service. Επίσης, κατά αυτόν τον τρόπο, οι ρυθμίσεις παραμετροποίησης που έχουν οριστεί μπορούν να διαμοιράζονται από πολλαπλά services, καθώς και να προστεθούν σε ή να αφαιρεθούν από ένα service οποιαδήποτε στιγμή. Όταν σταματήσει η εκτέλεση ενός container, το αρχείο configs που έχει δημιουργηθεί διαγράφεται από το σύστημα αρχείων στη μνήμη του.

Αξίζει να σημειωθεί ακόμη ότι ένας υπολογιστικός κόμβος έχει πρόσβαση στα configs που έχουν οριστεί μόνο εάν είναι manager ή αν σε αυτόν εκτελούνται tasks με εξουσιοδοτημένη πρόσβαση σε

αυτά. Όταν προστίθεται ένα νέο config στο swarm, το Docker το αποστέλλει στον manager κόμβο μέσω μίας TLS σύνδεσης. Από εκεί διαμοιράζεται κωδικοποιημένο στους υπόλοιπους managers προκειμένου να διασφαλιστεί υψηλή διαθεσιμότητα πρόσβασης όλων των managers σε αυτό [44], [46].

Τέλος, το πεδίο `secrets` του Docker Compose χρησιμοποιείται για τον προσδιορισμό ευαίσθητων δεδομένων παραμετροποίησης, ο διαμοιρασμός των οποίων υπόκεινται περιορισμούς ασφαλείας. Πιο συγκεκριμένα, μπορούν να προσδιοριστούν δεδομένα που χρειάζονται containers κατά τον χρόνο εκτέλεσης αλλά δεν μπορούν να αποθηκευτούν χωρίς κρυπτογράφηση στο Dockerfile της image του service τους ή στον πηγαίο κώδικα της εφαρμογής, όπως ονόματα χρηστών, κωδικοί, πιστοποιητικά και κλειδιά διασύνδεσης. Το Docker εξασφαλίζει την αυτοματοποιημένη διαχείριση των δεδομένων και τη δυνατότητα πρόσβασης σε αυτά μόνο από τα containers εξουσιοδοτημένων services και μόνο κατά την εκτέλεσή τους.

Επίσης, με την αποθήκευση ευαίσθητων δεδομένων στο πεδίο `secrets` επιτυγχάνεται η αποσύνδεση των ευαίσθητων δεδομένων από τα επιμέρους containers των services. Έτσι είναι δυνατή για παράδειγμα η χρήση διαφορετικών πιστοποιητικών και διαπιστευτηρίων ανάλογα με το περιβάλλον εκτέλεσης (π.χ. περιβάλλον ανάπτυξης, ελέγχου, παραγωγής). Στο πεδίο `secrets` μπορούν να οριστούν ακόμη και μη ευαίσθητα δεδομένα, ωστόσο συνήθως αυτά προσδιορίζονται στο πεδίο `configs`.

Η διαχείριση των ορισμένων secrets από το swarm είναι παρόμοια με τη διαχείριση των configs, δηλαδή κάθε οριζόμενο secret κοινοποιείται κρυπτογραφημένο στους manager κόμβους. Τα secrets συνδέονται ως αρχεία στο σύστημα αρχείων των containers, αλλά σε αντίθεση με τα configs αποθηκεύονται στη RAM. Αξίζει να σημειωθεί μάλιστα ότι ο ορισμός των secrets είναι εφικτός μόνο για τα swarm services και όχι για ανεξάρτητα containers [44], [47].

Η δημιουργία αιτήματος εκτέλεσης μιας εφαρμογής στο Docker SwarmKit είναι μια σχετικά απλή διαδικασία. Αρχικά, αν δεν υπάρχει η Docker image ενός service, χρειάζεται να καθοριστούν οι οδηγίες για τη δημιουργία της σε ένα Dockerfile και να εκτελεστεί η εντολή `docker build`. Οι δημιουργούμενες Docker images πρέπει να αποθηκεύονται σε ένα δημόσιο αποθετήριο (registry), όπως το Docker Hub, ή σε ένα ιδιωτικό αποθετήριο στην τοπική υποδομή, ώστε να διαμοιράζονται με αυτοματοποιημένο τρόπο σε όλους τους κόμβους του swarm.

Στη συνέχεια, η εφαρμογή ορίζεται σε ένα YAML αρχείο με τον τρόπο που περιγράφηκε παραπάνω. Αξίζει να σημειωθεί ωστόσο ότι τα αρχεία θα πρέπει να είναι συμβατά με την έκδοση 3.0 του Compose ή μεγαλύτερες και να χρησιμοποιούν πεδία που είναι συμβατά με το σύνολο εντολών `docker stack`. Το όνομα και η κατάληξη του αρχείου ενδείκνυνται να είναι `compose.yaml`, αλλά μπορούν να χρησιμοποιηθούν και τα `compose.yml`, `docker-compose.yaml` και `docker-compose.yml`. Μάλιστα, για τον ορισμό του μοντέλου της εφαρμογής μπορούν να χρησιμοποιηθούν

πολλαπλά αρχεία YAML [44]. Τέλος, με την εντολή `docker stack deploy` αρχικοποιούνται όλα τα services της εφαρμογής, δημιουργείται ένα δίκτυο για την απομονωμένη εκτέλεση των containers τους και δρομολογείται την εκτέλεσή τους στο swarm [35], [48].

Στην Εικόνα 3.5 φαίνεται ένα παράδειγμα εφαρμογής ορισμένη σε ένα αρχείο Compose. Η εφαρμογή αποτελείται από τα services `frontend` και `backend` και ορίζονται επιπλέον τα δίκτυα `front-tier` και `back-tier`, η δομή μόνιμης αποθήκευσης `db-data` και τα δεδομένα παραμετροποίησης `httpd-config` και `server-certificate`. Αν οι Docker images για τα services δεν υπάρχουν ήδη στο αποθετήριο που χρησιμοποιείται χρειάζεται να δημιουργηθούν βάσει ενός Dockerfile, αντίστοιχο αυτού που φαίνεται στην Εικόνα 3.6.

```yaml
services:
  frontend:
    image: awesome/webapp
    ports:
      - "443:8043"
    networks:
      - front-tier
      - back-tier
    configs:
      - httpd-config
    secrets:
      - server-certificate

  backend:
    image: awesome/database
    volumes:
      - db-data:/etc/data
    networks:
      - back-tier

volumes:
  db-data:
    driver: flocker
    driver_opts:
      size: "10GiB"

configs:
  httpd-config:
    external: true

secrets:
  server-certificate:
    external: true

networks:
  # The presence of these objects is sufficient to define them
  front-tier: {}
  back-tier: {}
```

Εικόνα 3.5: Παράδειγμα εφαρμογής ορισμένης σε αρχείο Docker Compose **[44]**

```dockerfile
# syntax=docker/dockerfile:1
FROM node:12-alpine
RUN apk add --no-cache python g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

Εικόνα 3.6: Παράδειγμα Dockerfile για τη δημιουργία Docker image **[107]**

3.3.2 Kubernetes

Λόγω του μεγάλου πλήθους υποστηριζόμενων λειτουργιών και των αυξημένων δυνατοτήτων επεκτασιμότητας, η ουσιαστική χρήση του Kubernetes απαιτεί σε κάποιο βαθμό τη συνολική κατανόηση του συστήματος και των σχεδιαστικών προτύπων που έχουν αναπτυχθεί. Έτσι, σε αυτή την υποενότητα γίνεται αρχικά αναφορά στον τρόπο με τον οποίο το Kubernetes αλληλοεπιδρά με τους διαθέσιμους πόρους υλικού και λογισμικού. Η αναφορά αυτή δεν αποτελεί μια εξαντλητική περιγραφή των μηχανισμών του Kubernetes αλλά συνοψίζει τα κυριότερα χαρακτηριστικά του συστήματος για την καλύτερη κατανόηση του χρησιμοποιούμενου μοντέλου περιγραφής εφαρμογών και των λειτουργιών της πλατφόρμας, τα οποία εξετάζονται στη συνέχεια στο πλαίσιο της διπλωματικής εργασίας.

Kubernetes API και Objects

Κεντρική έννοια στην ορολογία του Kubernetes είναι αυτή του object. Ένα object είναι μία μόνιμη οντότητα στο σύστημα, μέσω της οποίας ο χρήστης προσδιορίζει την επιθυμητή κατάσταση του cluster, υπό το πρίσμα της εκτέλεσης μιας εφαρμογής [49]. Πιο συγκεκριμένα, ένα object μπορεί να περιγράφει τις εκτελούμενες εφαρμογές σε containers και τους κόμβους στους οποίους έχουν δρομολογηθεί, τους υπολογιστικούς, αποθηκευτικούς και δικτυακούς πόρους που είναι διαθέσιμοι σε αυτές και την πολιτική διαχείρισης των εφαρμογών σε όλο τον κύκλο ζωής τους.

Η δημιουργία και η διαχείριση objects πραγματοποιείται με κλήσεις στο Kubernetes API, μια διεπαφή προγραμματισμού εφαρμογών βασισμένη σε περιγραφή πόρων (RESTful API), μέσω της οποίας, με HTTP αιτήματα, οι χρήστες αποκτούν πρόσβαση στους παρεχόμενους από το Kubernetes μηχανισμούς. Κεντρική έννοια στην περιγραφή του Kubernetes API είναι αυτή του resource. Τα περισσότερα περιγραφόμενα από το Kubernetes API resources είναι objects ή σύνολα από objects, τα οποία αναφέρονται ως lists στην ορολογία του Kubernetes, ενώ ένας μικρότερος αριθμός αφορά εικονικά resources τα οποία περιγράφουν λειτουργίες.

Οι τύποι resources που υποστηρίζονται από το Kubernetes API μπορούν να κατηγοριοποιηθούν ανάλογα με τη χρήση τους σε αυτούς για την περιγραφή εφαρμογών (π.χ. Pod, Deployment και ReplicaSet), για τη δικτύωση και τη διαχείριση του αιτημάτων των εφαρμογών (π.χ. Service, Endpoints, Ingress) και για την παραμετροποίηση των Pods των εφαρμογών και τις δομές αποθήκευσης (π.χ. ConfigMap, Secret, Volume). Υποστηρίζονται ακόμα resources για την πιστοποίηση (π.χ. ServiceAccount, CertificateSigningRequest) και εξουσιοδότηση των χρηστών (π.χ. Role, SelfSubjectAccessReview), για την περιγραφή πολιτικών διαχείρισης των διαθέσιμων πόρων (π.χ. NetworkPolicy, ResourceQuota) και για την περιγραφή του cluster (π.χ. Node, Namespace, Event). Επιπλέον, το Kubernetes διαθέτει resources (π.χ. CustomResourceDefinition) για τη δημιουργία και υποστήριξη και άλλων τύπων resources, ενδεχομένως οριζόμενων από τους χρήστες.

Για τον τρόπο οργάνωσης των resources στο Kubernetes API χρειάζεται να ληφθούν υπόψη τα παρακάτω. Σύνολα από resources, συνήθως ίδιου τύπου (collections), ομαδοποιούνται σε ομάδες (API groups) και η πρόσβαση σε αυτά γίνεται από κοινό σημείο στο API (API endpoint). Κάθε API group, μπορεί να προσδιορίζεται από μία ή περισσότερες εκδόσεις, οι οποίες εξελίσσονται ανεξάρτητα, με την κάθε μία να έχει ένα ή περισσότερα resources. Κάθε resource έχει μία συγκεκριμένη αναπαράσταση σε JSON, ενδεικτική των χαρακτηριστικών και των λειτουργιών του. Τέλος, κάθε resource έχει ένα όνομα το οποίο μπορεί να είναι μοναδικό σε όλο το cluster ή σε ένα συγκεκριμένο σύνολο ονομάτων (namespace). Έτσι, βάσει των παραπάνω, η πρόσβαση σε ένα resource στην πρώτη περίπτωση γίνεται μέσω API endpoint της μορφής `/apis/GROUP/VERSION/RESOURCETYPE/NAME`, ενώ στη δεύτερη περίπτωση με API endpoint της μορφής `/apis/GROUP/VERSION/namespaces/NAMESPACE/RESOURCETYPE/NAME`.

Κλήσεις στο Kubernetes API για την αξιοποίηση των λειτουργιών του μπορούν να γίνουν με τρεις τρόπους [50]. Πρώτον, η δημιουργία αιτήματος στο API μπορεί να γίνει μέσω της διεπαφής γραμμής εντολών (Command-Line Interface, CLI) με χρήση της κατάλληλης HTTP λειτουργίας. Δεύτερον, το αίτημα μπορεί να πραγματοποιηθεί κατευθείαν στον πηγαίο κώδικα μιας εφαρμογής με χρήση κατάλληλης client βιβλιοθήκης. Το Kubernetes υποστηρίζει επίσημα βιβλιοθήκες για τις γλώσσες dotnet, Go, Haskell, Java, JavaScript και Python αλλά υπάρχουν και άλλες βιβλιοθήκες παρεχόμενες από την ευρεία κοινότητα της πλατφόρμας [51]. Τρίτον, μπορούν να χρησιμοποιηθούν CLI εργαλεία, όπως το kubectl[9], τα οποία είναι πιο φιλικά στον χρήστη παρέχοντας περιγραφικές εντολές για τις διάφορες λειτουργίες και πραγματοποιώντας έμμεσα τις απαραίτητες στο API κλήσεις.

Συγκεκριμένα λοιπόν για τα objects, η δημιουργία, τροποποίηση, διαγραφή και ανάκτηση γίνεται χρησιμοποιώντας τις συνήθεις HTTP λειτουργίες `POST`, `PUT`, `PATCH`, `DELETE` και `GET`, ενώ ταυτόχρονα υποστηρίζονται και endpoints για άλλες ενσωματωμένες (built-in) ή ορισμένες από τον χρήστη HTTP λειτουργίες. Όλα τα ανταλλασσόμενα δεδομένα, τόσο στις κλήσεις στο API όσο και στις επιστρεφόμενες απαντήσεις, σειριοποιούνται εξ' ορισμού με τον μορφότυπο JSON και έτσι, η διαχείριση των ρυθμίσεων παραμετροποίησης γίνεται με δηλωτικό τρόπο [52], [53], [54].

Για την περιγραφή ενός object προς δημιουργία χρειάζεται να προσδιοριστούν τα πεδία `apiVersion`, `kind`, `metadata` και `spec`. Αρχικά, στο πεδίο `apiVersion` προσδιορίζεται η έκδοση του Kubernetes API που χρησιμοποιείται για τη δημιουργία και τη διαχείριση αυτού του object. Στο πεδίο `kind` προσδιορίζεται ο τύπος του object. Συνεχίζοντας, στο πεδίο `metadata` περιλαμβάνονται προσδιοριστικά δεδομένα αυτού του object, συγκεκριμένα το όνομά του (υποπεδίο `name`) με προαιρετικά το σύνολο ονομάτων αναφοράς (υποπεδίο `namespace`) και ένας μοναδικός αριθμός για τη διάκριση μεταξύ των objects με το ίδιο όνομα (υποπεδίο `uid`). Αν δεν προσδιοριστεί namespace,

[9] https://kubernetes.io/docs/reference/kubectl/overview/

χρησιμοποιείται το default. Τέλος, στο πεδίο `spec` περιγράφεται η επιθυμητή κατάσταση για αυτό το object. Τα υποπεδία του `spec` διαφέρουν ανάλογα με τον τύπο του αντικειμένου.

Για τη δημιουργία ενός object γίνεται κλήση στο Kubernetes API endpoint βάσει του πεδίου `apiVersion` και των προσδιοριστικών πληροφοριών του object των πεδίων `kind` και `metadata`, όπως αναφέρθηκε προηγουμένως. Το αίτημα αυτό μπορεί να γίνει είτε απευθείας με χρήση `POST` HTTP κλήσης είτε στον πηγαίο κώδικα με χρήση κάποιας εξωτερικής βιβλιοθήκης. Η περιγραφή του object χρειάζεται συμπεριληφθεί στο αίτημα σε μορφότυπο JSON. Αξίζει να σημειωθεί ωστόσο ότι συνηθέστερα, για τη δημιουργία ενός object, χρησιμοποιούνται CLI εργαλεία, όπως το kubectl. Αν χρησιμοποιηθεί το kubectl, η περιγραφή του object μπορεί να γίνει σε ένα YAML αρχείο, όπως αυτό που φαίνεται στην Εικόνα 3.7 για ένα Pod αποτελούμενο από container βασισμένο στην Docker image nginx [55]. Το kubectl μετατρέπει την περιγραφή σε YAML σε JSON κατά την υποβολή του αιτήματος δημιουργίας του object [49], [53].

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
```

Εικόνα 3.7: YAML αρχείο για τη δημιουργία ενός Pod

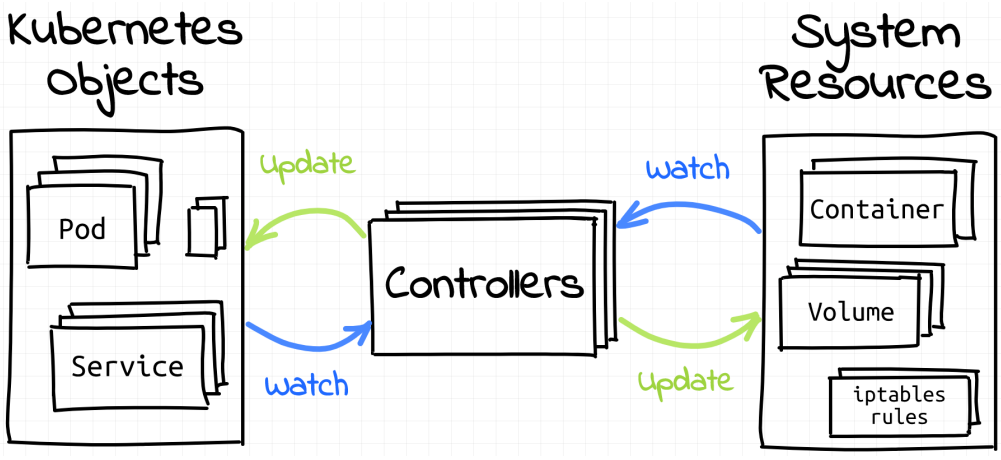
Controllers

Κεντρική έννοια στην ορολογία του Kubernetes είναι και αυτή του controller. Ο controller είναι ένας διαρκώς εκτελούμενος μηχανισμός ο οποίος είναι υπεύθυνος για τη διατήρηση όσο το δυνατόν μικρότερης απόκλισης ανάμεσα στην τρέχουσα και στην επιθυμητή κατάσταση του cluster όπως έχει προσδιοριστεί από τους χρήστες. Πιο συγκεκριμένα, κάθε controller παρακολουθεί το πεδίο `spec` ενός ή περισσοτέρων resources και προχωρά σε αναπροσαρμογές ανάλογα με την παρατηρούμενη απόκλιση. Το Kubernetes διαθέτει πολλαπλούς ενσωματωμένους controllers οι οποίοι εκτελούνται στο Control Plane, αλλά και controllers που εκτελούνται εκτός αυτού επεκτείνοντας τις λειτουργίες της πλατφόρμας. Παρέχει επίσης τη δυνατότητα στους χρήστες να ορίσουν controllers ανάλογα με τις ανάγκες των εκάστοτε εκτελούμενων εφαρμογών.

Ένας controller μπορεί να πραγματοποιήσει τις απαιτούμενες αλλαγές μέσω του API Server ή απευθείας. Η πρώτη περίπτωση είναι πιο συνήθης και αφορά κυρίως τους ενσωματωμένους

controllers του Kubernetes οι οποίοι διαχειρίζονται την κατάσταση του συστήματος αλληλοεπιδρώντας με τους υπόλοιπους μηχανισμός του cluster μέσω του API Server. Στη δεύτερη περίπτωση, ο controller λαμβάνει μόνος του τις απαιτούμενες ενέργειες. Η περίπτωση αυτή παρατηρείται συνήθως σε controllers που ελέγχουν resources εξωτερικά του cluster, οπότε και μαθαίνουν την επιθυμητή κατάστασή τους από τον API Server και επικοινωνούν απευθείας με το εξωτερικό σύστημα για τις απαραίτητες αναπροσαρμογές. Όταν πραγματοποιηθούν οι απαιτούμενες αλλαγές, ο controller ενημερώνει για τη νέα κατάσταση των resources (status) τον API Server [56].

Αξίζει να τονιστεί σε αυτό το σημείο η διάκριση, βάσει της οποίας έχει σχεδιαστεί το Kubernetes API, ανάμεσα στην επιθυμητή κατάσταση ενός resource όπως έχει προσδιοριστεί από τον χρήστη (spec) και στην τρέχουσα κατάστασή του όπως παρατηρείται στο cluster (status). Για την περίπτωση των objects πιο συγκεκριμένα, ο όρος spec, όπως προαναφέρθηκε, αναφέρεται στην επιθυμητή κατάσταση ενός object, προσδιορίζεται από τον χρήστη κατά τη δημιουργία του object στο πεδίο `spec` της περιγραφής και αποθηκεύεται μαζί με το object σε μόνιμη δομή αποθήκευσης. Από την άλλη πλευρά, το state περιγράφει την τρέχουσα κατάσταση του object όπως αυτή διατηρείται και ανανεώνεται από τους μηχανισμούς του Kubernetes, κυρίως από τους controllers εντός και εκτός του Control Plane, ενημερώνοντας το πεδίο `status` της περιγραφής του object. Όπως φαίνεται στην Εικόνα 3.8, οι controllers παρακολουθούν το status των objects. Στην περίπτωση απόκλισής του από το spec, πραγματοποιούν άμεσα ή έμμεσα τις απαραίτητες αλλαγές στους πόρους του cluster και ακολούθως ενημερώνουν το πεδίο `status` των objects με τη νέα τρέχουσα κατάσταση [57]. Συνεπώς, κάθε χρονική στιγμή το spec και το status ενός object δεν είναι κατ' ανάγκη ίδια. Μάλιστα, οι κλήσεις στο Kubernetes API είναι συνήθως διαφορετικές για αλλαγές στο πεδίο `spec` από αυτές για αλλαγές στο `status` [49], [53].



Εικόνα 3.8: Η αλληλεπίδραση των controllers με τα objects και τους πόρους του συστήματος

Περιγραφή και Δημιουργία Αιτήματος Εκτέλεσης Εφαρμογής

Ανάλογα με τις ανάγκες των χρηστών, η πλατφόρμα του Kubernetes ορίζει πληθώρα διαφορετικών τύπων objects για την περιγραφή εφαρμογών. Έτσι, σε αυτή την υποενότητα μελετώνται οι τύποι εκείνοι που χρησιμοποιούνται συνηθέστερα, συγκεκριμένα τα Deployment, Service, Ingress, StatefulSet, ConfigMap και Secret, μέσω ενός παραδείγματος εφαρμογής ιστού [58]. Πιο συγκεκριμένα, η εφαρμογή αυτή, γραμμένη σε Go, λαμβάνει τα μηνύματα των χρηστών και τα αποθηκεύει σε μία μόνιμη δομή αποθήκευσης Redis.

Το πρώτο βήμα για την περιγραφή της εφαρμογής είναι η δημιουργία μιας Docker image αποτελούμενη από τον εκτελέσιμο κώδικα της εφαρμογής (αρχείο `main.go`) και τις εξαρτήσεις για την ορθή εκτέλεσή του (αρχεία `go.mod` και `go.sum`). Για τον σκοπό αυτό, προσδιορίζονται σε ένα Dockerfile, όπως φαίνεται στην Εικόνα 3.9, οδηγίες για τη δημιουργία μιας scratch image[10] με τα αρχεία του κώδικα και των εξαρτήσεων που βρίσκονται στον φάκελο `/go/src/app`. Το μέγεθος των scratch images, συγκριτικά με τους άλλους τύπους container images, είναι πολύ μικρό, μειώνοντας τον απαιτούμενο χρόνο μεταφοράς τους από και προς το αποθετήριο εικόνων και μεταξύ των κόμβων του cluster.

```
FROM golang:alpine AS build-env
RUN mkdir /go/src/app && apk update && apk add git
ADD main.go go.mod go.sum /go/src/app/
WORKDIR /go/src/app
RUN go mod download && CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-extldflags "-static"' -o app .

FROM scratch
WORKDIR /app
COPY --from=build-env /go/src/app/app .
ENTRYPOINT [ "./app" ]
```

Εικόνα 3.9: Dockerfile για τη δημιουργία scratch image της εφαρμογής

Στη συνέχεια, χρειάζεται να δημιουργηθούν και να αρχικοποιηθούν τα Pods για την εκτέλεση της εφαρμογής. Ωστόσο, εξ' ορισμού, το Pod είναι μια εφήμερη δομή εκτέλεσης. Με άλλα λόγια, όταν διακοπεί η εκτέλεσή του, είτε ομαλά είτε λόγω κάποιου σφάλματος, το Pod διαγράφεται από τους υπεύθυνους για την εκτέλεσή του controllers και δεν αντικαθίσταται. Έτσι, για την περιγραφή εφαρμογών στο Kubernetes χρησιμοποιείται συνήθως το object Deployment [59].

Το Deployment συμβάλει στην αυτοματοποιημένη διαχείριση objects τύπου Pod και ReplicaSet. Για τον ορισμό ενός Deployment, δεδομένου ότι είναι object, χρειάζεται να προσδιοριστούν τα πεδία `apiVersion`, `kind`, `metadata` και `spec`. Το `apiVersion` που χρησιμοποιείται είναι το `apps/v1`, το προσδιοριζόμενο `kind` είναι `Deployment` και στο πεδίο `metadata` προσδιορίζονται μεταδεδομένα όπως για κάθε άλλο object (υποπεδία `name`, `namespace`, `uid`).

[10] https://hub.docker.com/_/scratch

Στο πεδίο `spec` περιγράφεται η επιθυμητή κατάσταση του Deployment. Πιο συγκεκριμένα, υποχρεωτικά προσδιορίζεται στο πεδίο `spec` μία ετικέτα (label) βάσει της οποίας αναγνωρίζονται τα Pods που διαχειρίζεται το Deployment (υποπεδίο `selector`). Συνοπτικά, η ετικέτα είναι ένα ζεύγος κλειδιού-τιμής, το οποίο καθορίζεται στο υποπεδίο `labels` του πεδίου `metadata` της περιγραφής ενός ή περισσοτέρων objects κατά τον ορισμό τους, και μπορεί να τροποποιηθεί σε όλη τη διάρκεια του κύκλου ζωής τους. Προκειμένου ο χρήστης να δημιουργήσει ένα object το οποίο χρησιμοποιεί ένα σύνολο από objects που διαμοιράζονται την ίδια ετικέτα, χρειάζεται να προσδιορίσει το κοινό κλειδί και την κοινή τιμή της ετικέτας στο πεδίο `selector` της περιγραφής του [60].

Επίσης, στο πεδίο `spec` καθορίζεται μία περιγραφή σύμφωνα με την οποία ο Deployment controller του Kubernetes θα δημιουργήσει νέα Pods αν χρειαστεί (υποπεδίο `template`), για παράδειγμα σε περίπτωση σφάλματος. Ακόμη, μπορεί να προσδιοριστεί το πλήθος αντιγράφων των Pods που απαιτείται να εκτελούνται κάθε χρονική στιγμή στο cluster (υποπεδίο `replicas`), η πολιτική που χρειάζεται να ακολουθηθεί για την αντικατάσταση υπαρχόντων Pods με νέα σε περίπτωση σφάλματος (υποπεδίο `strategy`) και το πλήθος Pods προηγούμενης έκδοσης από την τρέχουσα που πρέπει να διατηρηθούν για να είναι εφικτή η επιστροφή σε προηγούμενη κατάσταση του cluster αν χρειαστεί (υποπεδίο `revisionHistoryLimit`). Ακόμη, μπορεί να προσδιοριστεί ο χρόνος πέρα από τον οποίο ένα νέο Pod δε θεωρείται πλέον διαθέσιμο (υποπεδίο `minReadySeconds`) ή ένα Deployment θεωρείται ότι έχει αποτύχει (υποπεδίο `progressDeadlineSeconds`).

Στο παράδειγμα της εφαρμογής που εξετάζεται, δημιουργείται ένα Deployment για τις frontend λειτουργίες, το οποίο λαμβάνει τα αιτήματα των χρηστών. Το αρχείο `deployment.yml` απεικονίζεται στην Εικόνα 3.10, όπου έχουν προσδιοριστεί τα υποπεδία `replicas`, `selector` και `template`. Σημειώνεται ότι η apiVersion `extensions/v1beta1` έχει καταργηθεί και χρησιμοποιείται πλέον η `apps/v1` [61].

Στη διάρκεια του κύκλου ζωής μιας εφαρμογής, το status του Deployment μεταβάλλεται από τον Deployment controller. Πιο συγκεκριμένα, το status μπορεί να προσδιορίζεται ως `progressing` όταν δημιουργείται το σύνολο Pods ή το ReplicaSet και όταν τα Pods του Deployment κλιμακώνονται προς τα πάνω ή προς τα κάτω. Μπορεί να είναι επίσης `complete` όταν όλα τα Pods της τρέχουσας έκδοσης του Deployment είναι διαθέσιμα και δεν εκτελούνται Pods προηγούμενων εκδόσεων. Τέλος, μπορεί να είναι `failed` σε περίπτωση σφάλματος, για παράδειγμα κατά την ανάκτηση της Docker image, στην παραμετροποίηση της εφαρμογής ή όταν είναι ελλιπή τα δικαιώματα πρόσβασης στους πόρους που αιτείται το Deployment στην περιγραφή του.

Έτσι, ανακτώντας το πεδίο `status`, όπως αυτό διαμορφώνεται από τους Deployment controllers από την υποβολή του αιτήματος δημιουργίας του στο Kubernetes API, εντοπίζονται σε αυτό τα ακόλουθα υποπεδία μόνο για ανάγνωση (read-only). Στο υποπεδίο `replicas` προσδιορίζεται ο αριθμός των μη τερματισμένων Pods του Deployment και στα υποπεδία `availableReplicas`, `readyReplicas`,

`unavailableReplicas` και `updatedReplicas` ο επιμέρους αριθμός Pods των αντίστοιχων κατηγοριών. Στο υποπεδίο `conditions` αναγράφονται περισσότερες πληροφορίες για την τρέχουσα κατάσταση του Deployment, ενώ στο υποπεδίο `observedGeneration` ο αριθμός της έκδοσης του object όπως παρατηρείται από τον Deployment controller και ανανεώνεται σε κάθε αίτημα ενημέρωσης [53]. Τέλος, στο υποπεδίο `collisionCount` αναγράφεται ο αριθμός των συγκρούσεων ονόματος που παρατηρούνται όταν χρειάζεται να ονοματιστεί ένα νέο ReplicaSet [62].

Ο ορισμός ενός Deployment για τη διαχείριση Pods παρουσιάζει πολλαπλά οφέλη. Παρέχοντας ένα δηλωτικό τρόπο περιγραφής της επιθυμητής κατάστασης των μερών μια εφαρμογής, συνήθως των microservices για μια cloud-native εφαρμογή, ο Deployment controller παρακολουθεί συνεχώς την κατάσταση Pods που έχουν δημιουργηθεί και προχωρά στις απαραίτητες αλλαγές στο cluster ώστε η τρέχουσα κατάστασή τους να αποκλίνει όσο το δυνατόν λιγότερο από την επιθυμητή. Μάλιστα, σε περίπτωση σφάλματος είναι εφικτή η διακοπή της λειτουργίας των Pods για την αποκατάστασή του, η επιστροφή σε προηγούμενες εκδόσεις εκτελούμενου λογισμικού στα Pods και η δημιουργία νέων αν απαιτείται. Επίσης, διευκολύνεται η οριζόντια κλιμάκωση των Pods με την αύξηση ή τη μείωση των replicas σε περιπτώσεις αυξημένου ή μειωμένου φορτίου αντίστοιχα. Μέσω του Deployment controller ακόμα, ο API Server ενημερώνεται αυτόματα για την κατάσταση των Pods του Deployment, και έτσι και όλοι οι υπόλοιποι μηχανισμοί του cluster που ενδιαφέρονται για αυτή την πληροφορία. Συνεπώς, το Deployment object αυτοματοποιεί σημαντικά τη διαχείριση και η κλιμάκωση των Pods σε όλο τον κύκλο ζωής της εφαρμογής και για αυτό ενδείκνυται η χρήση του [63], [64], [59].

Στο παράδειγμα της εφαρμογής ιστού, αφού οριστούν τα απαραίτητα Pods του frontend, χρειάζεται γίνουν προσβάσιμα από τους χρήστες της εφαρμογής. Κάθε δημιουργούμενο Pod στο Kubernetes αποκτά μία μοναδική IP διεύθυνση για την πρόσβαση σε αυτό. Ωστόσο, οι διευθύνσεις αυτές είναι προσβάσιμες μόνο από τους εσωτερικούς μηχανισμούς του cluster. Για να επιτευχθεί η πρόσβαση στα Pods από τους εξωτερικούς ως προς το cluster χρήστες χρησιμοποιούνται τα objects Service και Ingress.

Πιο συγκεκριμένα, το Service είναι ένα object του Kubernetes μέσω του οποίου περιγράφεται η πολιτική πρόσβασης σε ένα σύνολο από Pods για την ικανοποίηση των αιτημάτων των χρηστών της εφαρμογής. Όπως και κάθε άλλο Kubernetes object, ένα Service ορίζεται σε αρχείο YAML ή σε JSON, με `apiVersion v1`, `kind Service` και `metadata` αντίστοιχα με τους άλλους τύπους objects. Στο πεδίο `spec` ορίζονται κυρίως τα Pods που ελέγχονται από το συγκεκριμένο Service (υποπεδίο `selector`), οι θύρες (ports) πρόσβασης στο Service και άρα και στα Pods που ελέγχει (υποπεδίο `ports`) καθώς και ο τύπος του ορισμένου Service (υποπεδίο `type`).

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    labels:
        app: frontend
    name: frontend
spec:
    replicas: 2
    selector:
        matchLabels:
            app: frontend
    template:
        metadata:
            labels:
                app: frontend
        spec:
            containers:
            - image: magalixcorp/sample-api:v1
              imagePullPolicy: IfNotPresent
              name: frontend
              env:
              - name: REDIS_PASSWORD
                valueFrom:
                    secretKeyRef:
                        name: redis-password
                        key: redis-password
              volumeMounts:
              - name: config-volume
                mountPath: /app/config.json
                subPath: config.json
            volumes:
                - name: config-volume
                  configMap:
                    name: app-config
```

Εικόνα 3.10: Deployment object για το frontend της εφαρμογής

Μπορούν να οριστούν οι τέσσερις διαφορετικοί τύποι Service. Ο προεπιλεγμένος τύπος κατά τον ορισμό ενός Service είναι ο `ClusterIP`, σύμφωνα με τον οποίο το Service είναι προσβάσιμο μόνο από τους εσωτερικούς μηχανισμούς του cluster. Συνεχίζοντας, με τον τύπο `ExternalName` το Service προσδιορίζεται από όνομα βάσει του DNS. Ένας άλλος υποστηριζόμενος τύπος είναι ο `LoadBalancer` με τον οποίο το Service αποκτά διεύθυνση IP ώστε είναι προσβάσιμο από εξωτερικούς μηχανισμούς του cluster, χρησιμοποιώντας τον μηχανισμό ισοστάθμισης φορτίου (load balancer) ενός παρόχου υπηρεσιών υπολογιστικού νέφους. Τέλος, ένα Service τύπου `NodePort` είναι εσωτερικά και εξωτερικά προσβάσιμο από συγκεκριμένη θύρα κάθε επιλεγμένου κόμβου του cluster. Η πρόσβαση σε Service αυτού του τύπου από εξωτερικούς μηχανισμούς του cluster γίνεται χρησιμοποιώντας τον συνδυασμό της IP διεύθυνσης (`NodeIP`) και του προκαθορισμένης θύρας (`NodePort`) του κόμβου.

Ανάλογα με τον τύπο του Service, στο πεδίο `spec` μπορούν να οριστούν και άλλα υποπεδία σχετικά με τις διευθύνσεις IP που χρησιμοποιούνται από το Service (π.χ. υποπεδία `ipFamilies`, `clusterIP`, `loadBalancerIP`), με την πολιτική κατανομής του φορτίου στα Pods (π.χ. υποπεδία

externalTrafficPolicy, internalTrafficPolicy, loadBalancerSourceRanges) και με τη διαχείριση των θυρών των κόμβων που χρησιμοποιεί το Service (π.χ. υποπεδία healthCheckNodePort, allocateLoadBalancerNodePorts). Τέλος, στο πεδίο status οι μηχανισμοί ελέγχου προσδιορίζουν κάθε χρονική στιγμή την τρέχουσα κατάσταση του Service [65], [66], [67]. Στην Εικόνα 3.11 περιγράφεται το Service frontend-svc για την εξυπηρέτηση των Pods του frontend μέρους της εφαρμογής, όπως ορίστηκαν από το Deployment στην Εικόνα 3.10.

Για την κατανομή του φορτίου στα Pods του frontend, θα μπορούσε να χρησιμοποιηθεί Service τύπου LoadBalancer. Ωστόσο, η λύση αυτή δυσχεραίνει την κλιμάκωση της εφαρμογής διότι κάθε load balancer συνδέεται αποκλειστικά με τα Pods του συγκεκριμένου Service κι έτσι, στην περίπτωση που χρειαστούν επιπλέον Services για την κάλυψη των αναγκών της εφαρμογής, θα πρέπει να δημιουργηθούν νέοι load balancers και να ανατεθούν επιπλέον διευθύνσεις IP. Για αυτό τον λόγο, χρησιμοποιείται ένα object Ingress.

```
1   apiVersion: v1
2   kind: Service
3   metadata:
4       labels:
5           app: frontend
6       name: frontend-svc
7   spec:
8       ports:
9       - port: 3000
10        protocol: TCP
11        targetPort: 3000
12      selector:
13          app: frontend
14      type: ClusterIP
```

Εικόνα 3.11: Service object για τα Pods του frontend της εφαρμογής

Το Ingress είναι ένα object του Kubernetes API για τη διαχείριση της πρόσβασης στα Services ενός cluster από εξωτερικούς μηχανισμούς, συνήθως μέσω HTTP(s) κλήσεων. Με αυτόν τον τρόπο, μόνο το Ingress είναι προσβάσιμο από εξωτερικούς μηχανισμούς του cluster και όχι τα Services που διαχειρίζεται. Για αυτόν τον λόγο, το Service της εφαρμογής στην Εικόνα 3.11 είναι τύπου ClusterIP.

Για τον ορισμό ενός Ingress χρησιμοποιείται η apiVersion networking.k8s.io/v1, με την extensions/v1beta1 να έχει καταργηθεί, το kind είναι Ingress και τα metadata ορίζονται όπως κάθε object. Στο πεδίο spec, μεταξύ άλλων προσδιορίζονται οι κανόνες βάσει των οποίων κατανέμεται το εξωτερικό φορτίο στα Services που έχουν οριστεί και άρα στα αντίστοιχα Pods (υποπεδίο rules). Για παράδειγμα, στην Εικόνα 3.12, όπου ορίζεται το Ingress για την εφαρμογή, τα αιτήματα στο endpoint /api εξυπηρετούνται από το Service frontend-svc το οποίο ορίστηκε στην Εικόνα 3.11 για το frontend μέρος της εφαρμογής και χρησιμοποιεί τη θύρα 3000 των κόμβων. Αξίζει

να σημειωθεί ότι τα endpoints στο υποπεδίο `rules` πρέπει να ορίζονται από το πιο ειδικό στο πιο γενικό. Αυτό καθώς οι κανόνες εξετάζονται σειριακά και ένας κανόνας εφαρμόζεται μόλις το endpoint του είναι ίδιο με το endpoint του αιτήματος [68], [69].

Για τη διαχείριση ενός Ingress είναι υπεύθυνος ένας Ingress controller. Ωστόσο, σε αντίθεση με τους controllers άλλων τύπων objects, το Kubernetes δε διαθέτει ενσωματωμένους Ingress controllers και για αυτό χρειάζεται να εγκατασταθεί ένας από τον διαχειριστή του cluster, ανάλογα με τις εκάστοτε απαιτήσεις. Το Kubernetes υποστηρίζει του AWS[11], GCE[12] και nginx[13] controllers, άλλα υπάρχουν υλοποιήσεις για την υποστήριξη και άλλων τύπων [70]. Η χρήση του Ingress συνδυαστικά με τα δύο ορισμένα Services απεικονίζεται στην Εικόνα 3.13 [58].

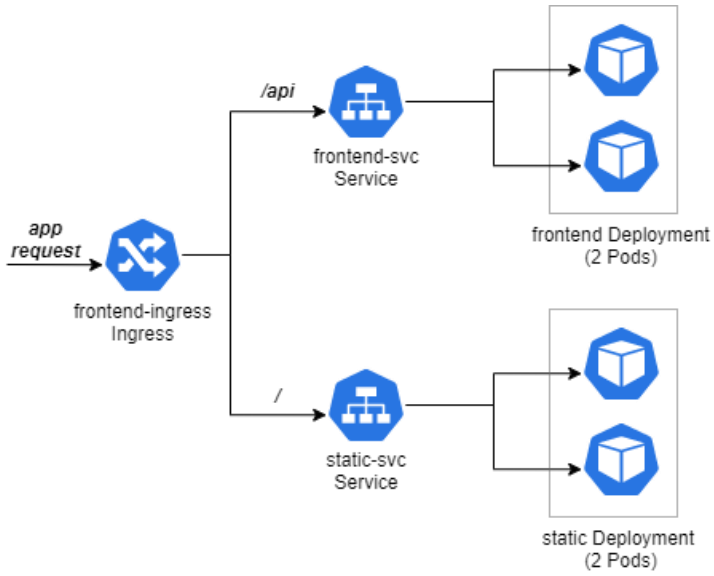
Συνεχίζοντας, στην υπό εξέταση εφαρμογή χρησιμοποιείται για την περιγραφή της μόνιμης δομής αποθήκευσης Redis ένα API object τύπου StatefulSet. Συνοπτικά, το StatefulSet όπως το Deployment διαχειρίζεται ένα σύνολο όμοιων Pods. Ωστόσο, σε αντίθεση με ένα Deployment, τα Pods του StatefulSet προσδιορίζονται μοναδικά από ένα δίκτυο, συγκεκριμένα από μοναδικά ονόματα DNS και hostname, και από μία δομή αποθήκευσης. Κατά αυτόν τον τρόπο, τα Pods του StatefulSet διατηρούν μία μόνιμη κατάσταση, ανεξάρτητα από πιθανές επανεκκινήσεις των μηχανισμών του cluster λόγω σφαλμάτων ή αλλαγές. Έτσι, το object StatefulSet χρησιμοποιείται συνήθως όταν χρειάζονται σταθερά και μοναδικά στοιχεία δικτύου, μόνιμες δομές αποθήκευσης όπως στην προκειμένη περίπτωση, διατεταγμένη κλιμάκωση των Pods και διατεταγμένες και αυτοματοποιημένες ενημερώσεις για την ομαλή εκτέλεση της εφαρμογής. Για τον ορισμό ενός StatefulSet χρησιμοποιείται ως apiVersion η `v1`, ως kind το `StatefulSet` και τα metadata προσδιορίζονται αντίστοιχα με τους άλλους τύπους objects. Στο πεδίο `spec` περιγράφεται το δίκτυο και η δομή αποθήκευσης που προσδιορίζουν μοναδικά τα Pods του οριζόμενου StatefulSet [71], [72].

Τέλος, οι ρυθμίσεις παραμετροποίησης μιας εφαρμογής μπορούν να προσδιοριστούν ξεχωριστά από τον πηγαίο κώδικά της χρησιμοποιώντας το API object ConfigMap. Στο ConfigMap διατηρούνται μη ευαίσθητα δεδομένα παραμετροποίησης σε ζεύγη κλειδιού-τιμής. Τα δεδομένα μπορούν να είναι αποθηκευμένα σε κάποιο άλλο αρχείο και όχι στο αρχείο ορισμού του ConfigMap, όπως απεικονίζεται στην Εικόνα 3.14 όπου τα δεδομένα βρίσκονται στο αρχείο `config.json`. Τα Pods μπορούν να χρησιμοποιήσουν τα δεδομένα των ConfigMaps ως μεταβλητές περιβάλλοντος, εντολές στο CLI ή αρχεία παραμετροποίησης σε δομή αποθήκευσης.

[11] https://github.com/kubernetes-sigs/aws-load-balancer-controller#readme
[12] https://git.k8s.io/ingress-gce/README.md#readme
[13] https://git.k8s.io/ingress-nginx/README.md#readme

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: frontend-ingress
spec:
    rules:
    - http:
        paths:
        - path: /api
          backend:
            serviceName: frontend-svc
            servicePort: 3000
        - path: /
          backend:
            serviceName: static-svc
            servicePort: 80
```

Εικόνα 3.12: Ingress object για τα front-end Services της εφαρμογής



Εικόνα 3.13: Τα Ingress και Service objects της εφαρμογής

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
    name: app-config
data:
  config.json: |-
    {
        "RedisHost": "redis-svc"
    }
```

Εικόνα 3.14: ConfigMap object για τον προσδιορισμό δεδομένων παραμετροποίησης

Αξίζει να σημειωθεί ωστόσο, ότι τα δεδομένα που αποθηκεύονται σε object τύπου ConfigMap δεν είναι κρυπτογραφημένα [73]. Για τον προσδιορισμό ευαίσθητων δεδομένων εκτός του κώδικα της εφαρμογής, όπως κωδικών πρόσβασης, μπορεί να χρησιμοποιηθεί το object Secret. Εξ' ορισμού, τα δεδομένα σε ένα object τύπου Secret αποθηκεύονται μη κρυπτογραφημένα στην etcd. Έτσι, πρόσβαση σε αυτά και δυνατότητα τροποποίησής τους έχουν οι μηχανισμοί με πρόσβαση στη etcd και οι χρήστες που μπορούν να δημιουργήσουν Pods στο συγκεκριμένο namespace. Για την ασφαλή διαχείριση των δεδομένων των Secrets ο διαχειριστής του cluster θα πρέπει να μεριμνήσει για την κρυπτογράφησή τους, δημιουργώντας για παράδειγμα objects τύπου EncryptionConfiguration, ή και ορίσει κανόνες Role-Based Access Control (RBAC) για τον περιορισμό της πρόσβασης σε αυτά [74]. Η δημιουργία και η εισαγωγή των Secrets στα Pods είναι προτιμότερο να γίνει μέσω ενός CLI εργαλείου. Στο kubectl για τον σκοπό αυτό χρησιμοποιείται η εντολή `kubectl create secret`. Τα Secrets προσδιορίζονται στο υποπεδίο `env` του `spec` του object που τα χρησιμοποιεί, όπως φαίνεται στην Εικόνα 3.10 για το Secret `redis-password` που χρησιμοποιείται από το Deployment του frontend της εφαρμογής.

Τέλος, για τη δημιουργία, την εκτέλεση και την αναβάθμιση όλων των επιμέρους objects και controllers της εφαρμογής στο Kubernetes cluster μπορούν να χρησιμοποιηθούν οι σχετικές εντολές ενώ CLI εργαλείου, όπως το kubectl. Ωστόσο, για τον σκοπό αυτό ενδείκνυται η χρήση εργαλείων διαχείρισης Kubernetes εφαρμογών, όπως το Helm[14]. Συνοπτικά, το Helm είναι ένα εργαλείο προτύπων (templating tool) με τη χρήση του οποίου οι απαραίτητες πληροφορίες για τη δημιουργία μιας εφαρμογής Kubernetes ορίζουν ένα Chart. Το Chart σε συνδυασμό με τις ρυθμίσεις παραμετροποίησης (Config) δημιουργούν την εφαρμογή ως εκτελέσιμο αντικείμενο (Release). Η χρήση αυτών των εργαλείων διευκολύνει καθοριστικά τη μεταφορά, εκτέλεση και κλιμάκωση των εφαρμογών σε διαφορετικά περιβάλλοντα, καθώς έτσι δεν απαιτούνται ξεχωριστά αρχεία περιγραφής και παραμετροποίησης για κάθε περιβάλλον εκτέλεσης. Με τη χρήση του Helm οι τιμές που πρέπει να μεταβληθούν κατά τη μεταφορά των εφαρμογών από το ένα περιβάλλον στο άλλο αντικαθίστανται από μεταβλητές (placeholders) που τροποποιούνται κατάλληλα ανάλογα με το εκάστοτε περιβάλλον ανάπτυξης.

3.4 Κύκλος Ζωής Μονάδων Εκτέλεσης Εφαρμογών

3.4.1 Docker SwarmKit και Swarm Mode

Η αλληλουχία ενεργειών που απαιτείται για τη χρονοδρομολόγηση εφαρμογών σε ένα cluster από Docker Engines σε λειτουργία swarm είναι σχετικά απλή, συγκριτικά με τα υπόλοιπα εργαλεία ενορχήστρωσης. Αρχικά, ο χρήστης χρειάζεται να αιτηθεί τη δημιουργία του stack της εφαρμογής, παρέχοντας την αντίστοιχη περιγραφή όπως αναλύθηκε στην Ενότητα 3.3.1. Ο swarm manager

[14] https://helm.sh

λαμβάνει και επεξεργάζεται τα αιτήματα χρηστών μέσω του Docker CLI ή απευθείας στο API. Πιο συγκεκριμένα, στον swarm manager ο μηχανισμός του orchestrator δημιουργεί σειριακά τα tasks που απαιτούνται για κάθε service. Τα δημιουργούμενα tasks μπορεί να αντιστοιχούν είτε σε νέα services, είτε να δημιουργήθηκαν για την κλιμάκωση υπαρχόντων services ή για την αντικατάσταση tasks που δεν εκτελέστηκαν επιτυχώς.

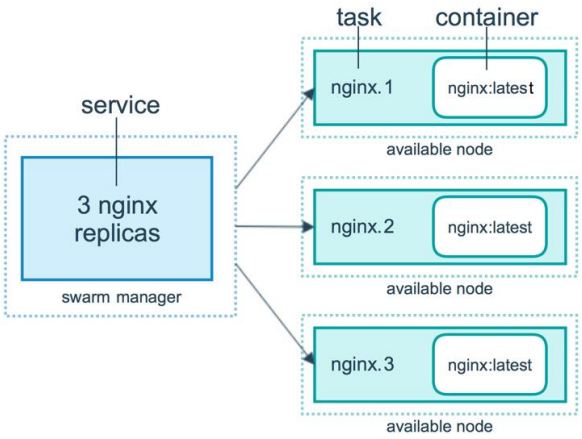
Στη συνέχεια, κάθε δημιουργούμενο task, αφού συσχετιστεί από τον allocator με τους απαραίτητους για την εκτέλεσή του πόρους (π.χ. πόρους δικτύου), δρομολογείται από τον dispatcher και τον scheduler σε έναν κόμβο υπολογισμού για εκτέλεση. Όμοια tasks μπορούν να δρομολογηθούν από κοινού. Ο αλγόριθμος χρονοδρομολόγησης που χρησιμοποιείται για την επιλογή του κόμβου υπολογισμού και οι δυνατότητες παραμετροποίησής του θα αναλυθούν στις Ενότητες 4.1.1 και 4.2.1. Για να εκτελεστεί το container του task, ο worker κόμβος που επιλέχθηκε χρειάζεται να αποδεχθεί τη δρομολόγησή του [42]. Με αυτόν τον τρόπο, τα containers ενός service μπορούν να δρομολογούνται ανεξάρτητα το ένα από το άλλο μέχρι να εκτελεστούν επιτυχώς ή να αποτύχουν. Η Εικόνα 3.15 απεικονίζει την κατανομή των tasks ενός service στους επιλεγμένους κόμβους υπολογισμού και η Εικόνα 3.16 παρουσιάζει τη διαδικασία δημιουργίας ενός service και της χρονοδρομολόγησης των tasks του στους κόμβου υπολογισμού ενός swarm.

Όταν δημιουργηθεί ένα service, τα tasks του μεταβαίνουν από μία κατάσταση μόνο στις επόμενες ενός πεπερασμένου συνόλου καταστάσεων. Πιο συγκεκριμένα, όταν δημιουργείται ένα task από τον orchestrator βρίσκεται στην κατάσταση `NEW`. Συνεχίζοντας, όταν ο allocator ξεκινήσει να επεξεργάζεται το task, η κατάστασή του γίνεται `PENDING`. Το task παραμένει σε κατάσταση `PENDING` μέχρι ο dispatcher και ο scheduler επιλέξουν έναν κόμβο υπολογισμού στον οποίο θα δρομολογηθεί.

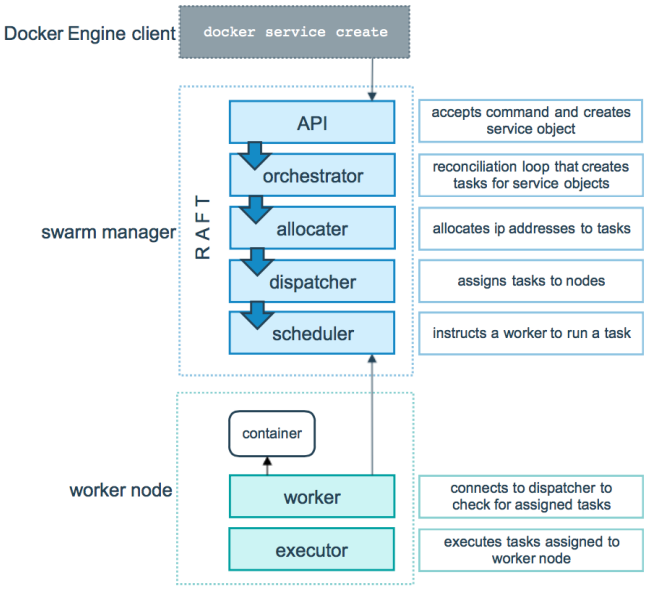
Ωστόσο, ένα service μπορεί να έχει οριστεί έτσι ώστε κανένας κόμβος του swarm εκείνη τη χρονική στιγμή να μην μπορεί να εκτελέσει τα containers των tasks του. Για παράδειγμα, αυτό μπορεί να συμβεί αν η λειτουργία όλων των υπολογιστικών κόμβων έχει διακοπεί ή αν δεν υπάρχει κόμβος που να διαθέτει τους απαιτούμενους πόρους (π.χ. μνήμη) και περιορισμούς τοποθέτησης. Σε αυτή την περίπτωση, τα tasks του service παραμένουν σε κατάσταση `PENDING`. Όταν βρεθεί ένας κόμβος στον οποίο μπορεί να δρομολογηθεί ένα task, η κατάστασή του γίνεται `ASSIGNED`. Εφεξής, ο agent αυτού του κόμβου ενημερώνει τον manager του swarm για τη μετάβαση της κατάστασης του task στις καταστάσεις `ACCEPTED`, `PREPARING`, `STARTING` και από τη `RUNNING` στην `COMPLETE` ή `FAILED`. Αξίζει να σημειωθεί μάλιστα ότι όταν ένα task δρομολογείται σε έναν worker κόμβο, εκτελείται σε αυτόν μόνο μία φορά μέχρι να ολοκληρωθεί η εκτέλεσή του επιτυχώς ή να αποτύχει. Στην περίπτωση αποτυχίας, χρειάζεται να αντικατασταθεί από ένα νέο task. Ο Πίνακας 3.1 συνοψίζει το σύνολο των πιθανών καταστάσεων ενός task [75], [76].

Πίνακας 3.1: Το σύνολο των δυνατών καταστάσεων ενός task στο swarm

Κατάσταση task	Περιγραφή
NEW	Δημιουργία του task από τον orchestrator
PENDING	Κατανέμονται οι απαιτούμενοι πόροι στο task και οι dispatcher και scheduler αναζητούν υπολογιστικό κόμβο για τη δρομολόγησή του
ASSIGNED	Ο scheduler δρομολογεί το task σε έναν υπολογιστικό κόμβο
ACCEPTED	Ο υπολογιστικός κόμβος επιλέγει να εκτελέσει το container του task
REJECTED	Ο υπολογιστικός κόμβος απορρίπτει το task
PREPARING	Το task ετοιμάζεται να εκτελεστεί
STARTING	Ξεκινάει η εκτέλεση του container του task
RUNNING	Εκτελείται το container του task
COMPLETE	Ολοκληρώνεται χωρίς σφάλμα η εκτέλεση του container του task
FAILED	Η εκτέλεση του container του task αποτυγχάνει
SHUTDOWN	Ο orchestrator τερμάτισε το task
ORPHANED	Ο υπολογιστικός κόμβος που επιλέχθηκε για την εκτέλεση του task δεν είναι διαθέσιμος
REMOVED	Το task διαγράφηκε από τον χρήστη ή εξαιτίας της αποκλιμάκωσης του service του



Εικόνα 3.15: Κατανομή των tasks ενός service σε κόμβους υπολογισμού



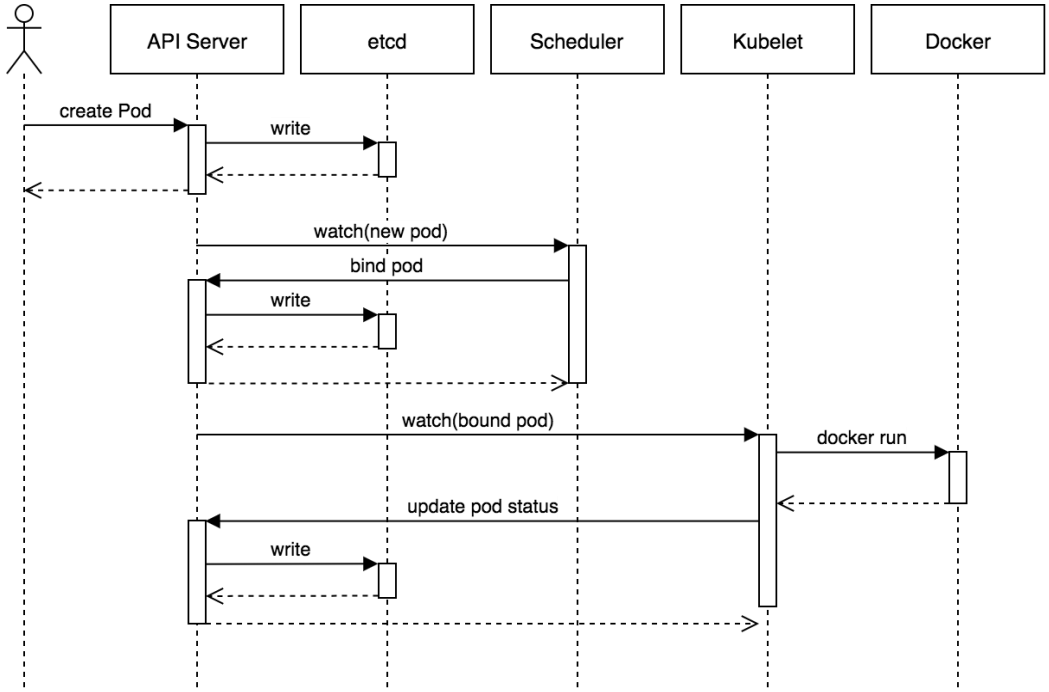
Εικόνα 3.16: Η λειτουργία του swarm manager κατά τη δημιουργία ενός service

3.4.2 Kubernetes

Η Εικόνα 3.17 απεικονίζει την αλληλουχία ενεργειών πραγματοποιούνται από τους μηχανισμούς του Kubernetes για τη δημιουργία και εκτέλεση ενός Pod. Η αλληλουχία αυτή είναι ίδια για τους όλους τους τύπους containers που υποστηρίζονται από την πλατφόρμα. Πιο συγκεκριμένα, αρχικά, ο χρήστης αιτείται τη δημιουργία των απαραίτητων Pods της εφαρμογής στον API Server ορίζοντας συνήθως ένα object τύπου Deployment, όπως αναλύθηκε στην Ενότητα 3.3.2. Στη συνέχεια, ο API Server καταχωρεί κάθε Pod στη βάση etcd και ο Scheduler ειδοποιείται για το Pod που δημιουργήθηκε και ξεκινά την αναζήτηση ενός υπολογιστικού κόμβου για την εκτέλεσή του, όπως θα αναλυθεί στην Ενότητα 4.1.2. Όταν επιλεγεί ένας κόμβος για την εκτέλεση του container του Pod, ο Scheduler ενημερώνει τον API Server για αυτή την επιλογή και ο API Server με τη σειρά του μεταβιβάζει την απόφαση στο Kubelet του επιλεχθέντα κόμβου. Τέλος, το Kubelet αναλαμβάνει την εκτέλεση των containers του Pod, ενημερώνοντας τον API Server για την εκάστοτε κατάσταση κάθε Pod. Όλες οι αλλαγές καταστάσεων καταγράφονται από τον API Server στην etcd, η οποία διατηρεί τη συνολική κατάσταση του cluster [40].

Ο κύκλος ζωής του Pod ορίζεται ως ένα σύνολο διακριτών φάσεων (Pod phases) οι οποίες προσδιορίζονται στο πεδίο status της περιγραφής του. Ο Πίνακας 3.2 περιγράφει αυτές τις φάσεις. Κάθε Pod μεταβαίνει σειριακά από τη μία φάση του κύκλου ζωής στην επόμενη, χωρίς να είναι εφικτή η επιστροφή του σε προηγούμενες φάσεις, μέχρι να ολοκληρωθεί η εκτέλεσή του ή να διακοπεί λόγω σφάλματος. Μάλιστα, σε περίπτωση διακοπής, το Pod δεν ανατίθεται σε νέο κόμβο για εκτέλεση, αλλά διαγράφεται από το cluster και αποδεσμεύονται οι χρησιμοποιούμενοι από αυτό πόροι. Τα Pods μιας εφαρμογής που τερματίστηκαν πρόωρα λόγω σφάλματος αντικαθίστανται από νέα μόνο αν είχαν οριστεί στο πλαίσιο objects όπως το Deployment και το ReplicaSet.

Πέρα από τη φάση ενός Pod, οι μηχανισμοί του Kubernetes ελέγχουν και την κατάσταση των containers του. Τα containers ενός Pod δημιουργούνται αφότου ο Scheduler το αναθέσει για εκτέλεση σε έναν κόμβο, από το Kubelet του επιλεγμένου κόμβου. Στη διάρκεια του κύκλου ζωής του Pod, η κατάστασή τους προσδιορίζεται ως `waiting`, `running` ή `terminated`. Στην κατάσταση `waiting`, το container εκτελεί τις λειτουργίες που απαιτούνται για την εκκίνησή του (π.χ. ανάκτηση της container image ή των Secrets που χρησιμοποιεί). Στην κατάσταση `running` το container εκτελείται χωρίς σφάλματα ενώ τέλος στην `terminated`, η εκτέλεσή του έχει ολοκληρωθεί επιτυχώς ή διακόπηκε λόγω σφάλματος [77].



Εικόνα 3.17: Οι ενέργειες για τη δημιουργία ενός Pod και την εκτέλεση των containers του

Πίνακας 3.2: Οι φάσεις του κύκλου ζωής ενός Kubernetes Pod

Φάση του Pod	**Περιγραφή**
Pending	Το Pod έχει δημιουργηθεί στο cluster, και ή αναμένει να δρομολογηθεί ή κάποια από τα containers του δεν είναι έτοιμα να εκτελεστούν (π.χ. υπάρχει καθυστέρηση ανάκτησης των container images).
Running	Όλα τα containers του Pod έχουν δημιουργηθεί και το Pod έχει ανατεθεί για εκτέλεση σε έναν κόμβο. Τουλάχιστον ένα από τα containers του πρόκειται να ξεκινήσει να εκτελείται, έχει ξεκινήσει ήδη να εκτελείται ή βρίσκεται σε διαδικασία επανεκκίνησης.
Succeeded	Όλα τα containers του Pod ολοκλήρωσαν επιτυχώς την εκτέλεσή τους και δε θα επανεκκινηθούν.
Failed	Έχει διακοπεί η εκτέλεση όλων των containers του Pod και τουλάχιστον ενός λόγω σφάλματος.
Unknown	Η κατάσταση του Pod δεν μπορεί να ανακτηθεί. Συνήθως η φάση αυτή παρατηρείται όταν δεν είναι εφικτή η επικοινωνία με τον κόμβο στον οποίο έχει ανατεθεί η εκτέλεση του Pod.

Κεφάλαιο 4: Μηχανισμοί Χρονοδρομολόγησης Εφαρμογών

4.1 Επισκόπηση των Μηχανισμών Χρονοδρομολόγησης

4.1.1 Docker SwarmKit και Swarm Mode

Όπως προαναφέρθηκε στις Ενότητες 3.2.1 και 3.4.1, ο scheduler σε ένα swarm εκτελείται στους manager κόμβους υπολογισμού και λαμβάνει tasks από τον dispatcher με τη σειρά με την οποία υποβλήθηκαν στο σύστημα (First In First Out, FIFO). Όταν ο scheduler λαμβάνει ένα task, αυτό βρίσκεται σε κατάσταση `PENDING` και έχουν ολοκληρωθεί οι απαραίτητες ενέργειες για να είναι εφικτή η εκτέλεσή του, όπως ανάθεση δικτυακών πόρων. Έτσι, η κύρια αρμοδιότητα του scheduler είναι να επιλέξει έναν worker υπολογιστικό κόμβο όπου μπορεί να εκτελεστεί το container του task [78].

Ο εντοπισμός ενός συνόλου υπολογιστικών κόμβων στους οποίους μπορεί να δρομολογηθεί ένα task και η επιλογή ενός από αυτούς γίνεται σε τρία στάδια. Στο πρώτο στάδιο, ο scheduler επιλέγει ένα υποσύνολο κόμβων του swarm εφαρμόζοντας ένα σύνολο φίλτρων που είναι οργανωμένα σε μια δομή προγραμματισμού struct τύπου `Pipeline`[15], όπως έχει οριστεί στον πηγαίο κώδικα του scheduler. Καθένα από τα φίλτρα υλοποιεί τη διεπαφή (interface) `Filter`[16], η οποία αποτελείται από τις μεθόδους `SetTask`, `Check` και `Explain`. Συγκεκριμένα, η μέθοδος `SetTask`[17] ελέγχει αν το φίλτρο μπορεί να εφαρμοστεί στο task, επιστρέφοντας αναλόγως `True` ή `False`. Επίσης, στη `SetTask` μπορούν να πραγματοποιηθούν ενέργειες προ-επεξεργασίας οι οποίες απαιτούνται για την εκτέλεση του task. Συνεχίζοντας, στη μέθοδο `Check`[18] πραγματοποιείται το φιλτράρισμα των worker κόμβων του swarm, ελέγχοντας κάθε φορά αν το container του task μπορεί να εκτελεστεί στον εκάστοτε κόμβο. Ταυτόχρονα, η μέθοδος `Explain`[19] παράγει μηνύματα που περιγράφουν πιθανά σφάλματα κατά την εκτέλεση της μεθόδου `Check`. Ο Πίνακας 4.1 συνοψίζει τα φίλτρα που χρησιμοποιούνται εξ ’ορισμού στην τρέχουσα έκδοση 20.10 του Docker Engine με τη σειρά που εφαρμόζονται στους κόμβους για κάθε task.

[15] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/pipeline.go#L38
[16] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L14
[17] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L19
[18] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L24
[19] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L27

Πίνακας 4.1: Το σύνολο υλοποιημένων φίλτρων του SwarmKit

Φίλτρο	**Λειτουργία**
ReadyFilter[20]	Ελέγχει αν ο υπό εξέταση υπολογιστικός κόμβος είναι σε κατάσταση `READY` και η διαθεσιμότητά του έχει προσδιοριστεί ως `ACTIVE`
ResourceFilter[21]	Ελέγχει ότι ο κόμβος διαθέτει τους απαραίτητους πόρους επεξεργαστή και μνήμης για την εκτέλεση του task
PluginFilter[22]	Ελέγχει ότι ο κόμβος έχει εγκατεστημένα τα απαραίτητα plugins δομών αποθήκευσης
ConstraintFilter[23]	Ελέγχει ότι ο κόμβος ικανοποιεί τους περιορισμούς τοποθέτησης (placement constraints) του service
PlatformFilter[24]	Ελέγχει ότι ο κόμβος έχει το κατάλληλο λειτουργικό σύστημα και αρχιτεκτονική
HostPortFilter[25]	Ελέγχει ότι οι θύρες που θα χρησιμοποιηθούν από το service δεν είναι δεσμευμένες στον υπό εξέταση κόμβο
MaxReplicasFilter[26]	Απορρίπτει τους κόμβους στους οποίους εκτελούνται περισσότερα tasks ενός service από το μέγιστο πλήθος που έχει οριστεί

Στο δεύτερο στάδιο της χρονοδρομολόγησης, κάθε κόμβος που ικανοποιεί τους ελέγχους των φίλτρων, προστίθεται στην κατάλληλη θέση ενός σωρού μεγίστου (max heap). Πιο συγκεκριμένα, ο scheduler διατηρεί ένα δέντρο σωρών μεγίστου, με κάθε σωρό να αντιστοιχεί σε ένα swarm service και να αποτελείται από τους workers κόμβους στους οποίους θα μπορούσαν να εκτελεστούν τα tasks του service. Στην περίπτωση που ο χρήστης προσδιορίσει περιορισμούς ως προς το σύνολο των κόμβων στους οποίους μπορεί να εκτελεστεί μία εφαρμογή (placement constraints), όπως θα αναλυθεί στην Ενότητα 4.2.1, για κάθε service μπορεί να δημιουργηθούν περισσότεροι του ενός σωροί μεγίστου.

Σε κάθε σωρό, οι κόμβοι ταξινομούνται ανάλογα με τον αριθμό των tasks του service που μπορούν να δρομολογηθούν σε αυτούς και έτσι, ο κόμβος στον οποίο μπορούν να εκτελεστούν τα περισσότερα tasks βρίσκεται στην κορυφή του σωρού. Μάλιστα, το μέγιστο μέγεθος του σωρού καθορίζεται από το πλήθος των tasks που μπορούν να ομαδοποιηθούν από το λειτουργικό σύστημα και να εκτελεστούν χωρίς την παρέμβαση του χρήστη (batch processing) και ανάλογα με την υλοποίηση, η υπέρβαση αυτού του μέγιστου ορίου μπορεί να οδηγήσει στην αντικατάσταση των «χειρότερων» κόμβων, βάσει της στρατηγικής χρονοδρομολόγησης που χρησιμοποιείται, από τους υπό εξέταση «καλύτερους».

Η κατασκευή του δέντρου σωρών μεγίστου είναι μία ακριβή υπολογιστικά διαδικασία με χρονική πολυπλοκότητα $O(\text{αριθμός workers κόμβων})$ για κάθε task. Για να μειωθεί ο περιττός χρόνος επεξεργασίας σε περιπτώσεις που δε χρειάζεται να δημιουργηθεί ένας ξεχωριστός σωρός για κάθε service (π.χ. για replicated services με replicated tasks), μπορεί να σχηματιστεί ένα μόνο δέντρο διαμοιραζόμενο από όμοια tasks, δηλαδή tasks που έχουν ίδια service ID και SpecVersion. Για την

[20] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L31
[21] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L55
[22] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L104
[23] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L219
[24] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L254
[25] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L323
[26] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/filter.go#L323

ομαδοποίηση των όμοιων task και την από κοινού δρομολόγησή τους, ο scheduler του SwarmKit περιμένει μέχρι 50 ms από τη στιγμή που λαμβάνει το πρώτο task μέχρι να το αναθέσει σε κάποιον κόμβο για εκτέλεση [79].

Η τρέχουσα έκδοση του SwarmKit υποστηρίζει μόνο την Spread στρατηγική επιλογής κόμβου, η οποία στοχεύει στον ομοιόμορφο διαμοιρασμό των tasks ενός service στους διαθέσιμους κόμβους υπολογισμού. Πιο συγκεκριμένα, σύμφωνα με αυτή τη στρατηγική, μεταξύ των workers κόμβων που πληρούν τις απαιτήσεις και τους περιορισμούς σε πόρους όπως έχουν καθοριστεί από τον χρήστη, ο scheduler επιλέγει αυτόν στον οποίο εκτελούνται τα λιγότερα σε πλήθος tasks του service. Μάλιστα, σε περίπτωση ισοπαλίας, επιλέγεται ο κόμβος στον οποίο εκτελούνται συνολικά τα λιγότερα tasks, ανεξάρτητα από τα services στα οποία ανήκουν. Έτσι, ο κόμβος με τη μεγαλύτερη προτεραιότητα στον σωρό μεγίστου είναι αυτός στον οποίο εκτελούνται τα περισσότερα tasks ενός service ή συνολικά και βάσει της Spread στρατηγικής πιθανόν δε θα επιλεχθεί [79], [80].

Στο τρίτο στάδιο επιλογής worker κόμβου, οι σωροί μεγίστου μετατρέπονται σε αύξουσα ταξινομημένες λίστες ως προς την καθορισμένη προτεραιότητα των σωρών. Έτσι, ο worker κόμβος στον οποίο εκτελούνται τα λιγότερα tasks του service, ή και επιπλέον τα λιγότερα tasks συνολικά, βρίσκεται στην πρώτη θέση της λίστας, αν χρησιμοποιείται η Spread στρατηγική. Η μετατροπή αυτή γίνεται χωρίς τη χρήση βοηθητικής δομής δεδομένων (in-place), γεγονός που αιτιολογεί και τη χρήση σωρών μεγίστου.

Ακολούθως, ο scheduler χρησιμοποιεί το δέντρο αύξουσα ταξινομημένων λιστών για τη ανάθεση ομάδων tasks σε κόμβους με χρονική καθυστέρηση, όπως προαναφέρθηκε, μέχρι 50 ms. Για κάθε ομάδα tasks προς εκτέλεση, ο scheduler επιλέγει τον υπολογιστικό κόμβο με τα λιγότερα εκτελούμενα tasks για αυτό το service και του αναθέτει τόσα tasks όσα αυτά που εκτελούνται στον επόμενο κόμβο στην ταξινομημένη λίστα. Αν μετά από αυτή τη διαδικασία υπάρχουν ακόμα tasks τα οποία δεν έχουν ανατεθεί σε κάποιον κόμβο, ακολουθείται μία προσέγγιση round-robin[27] για να βρεθούν κατάλληλοι κόμβοι. Δεδομένου μάλιστα ότι το state ενός task μεταβαίνει ακολουθιακά από μία κατάσταση μόνο στις επόμενες ενός πεπερασμένου συνόλου καταστάσεων, όπως περιγράφηκε στην Ενότητα 3.4.1, ο scheduler θα αναθέσει κάθε task για εκτέλεση σε κάποιον κόμβο μόνο μία φορά. Συνεπώς, από τη στιγμή που ληφθεί, η απόφαση χρονοδρομολόγησης δεν αλλάζει.

Σε αυτό το σημείο, αξίζει να σημειωθεί πως ένα πρόβλημα της Spread στρατηγικής είναι ότι μπορεί αέναα να επιλέγει υπολογιστικούς κόμβους που παρουσιάζουν προβλήματα. Αυτό συμβαίνει όταν ένας κόμβος δεν μπορεί να εκτελέσει τα containers που του έχουν ανατεθεί, και έτσι φαίνεται να έχει λιγότερο φορτίο συγκριτικά με τους υπόλοιπους. Για την αντιμετώπιση αυτού του προβλήματος, ο scheduler παρακολουθεί τον αριθμό αποτυχιών εκτέλεσης containers σε κάθε worker. Αν ο αριθμός

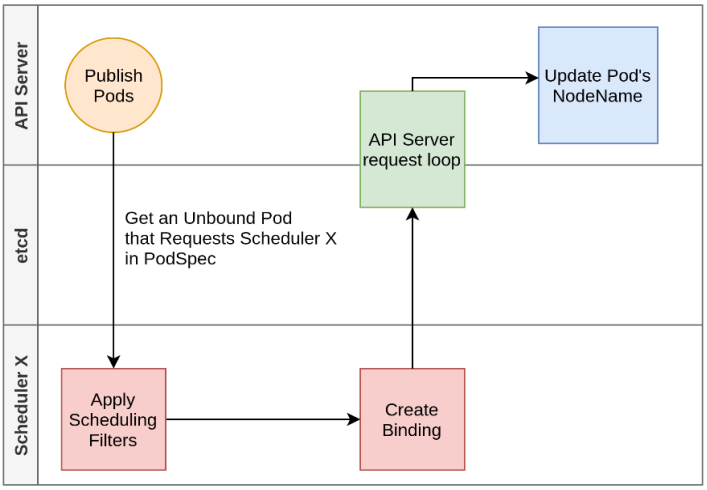
[27] https://avinetworks.com/glossary/round-robin-load-balancing/

αυτός υπερβεί ένα προκαθορισμένο όριο, ο κόμβος σημαίνεται ως ελαττωματικός ώστε να μην επιλέγεται από το δέντρο λιστών [79].

Τέλος, στις μελλοντικές εκδόσεις του Docker Engine θα υποστηρίζονται και άλλες στρατηγικές χρονοδρομολόγησης πέρα από την Spread και θα δίνεται η δυνατότητα στον χρήστη να επιλέξει αυτήν που θα χρησιμοποιηθεί ανάλογα με τις ανάγκες της εφαρμογής. Για παράδειγμα, θα μπορούσαν να υποστηρίζονται μια εκδοχή της Spread στρατηγικής με βάρη, σε περιπτώσεις που μία ομάδα πόρων έχει περισσότερους υπολογιστικούς κόμβους από τις υπόλοιπες, ή μια στρατηγική με την οποία τα tasks θα δρομολογούνται σε κόμβους όπου εκτελούνται παρόμοια tasks [81].

4.1.2 Kubernetes

Ο Scheduler στο Kubernetes είναι ένα μηχανισμός που εκτελείται στο Control Plane, όπως αναφέρθηκε στην Ενότητα 3.2.2, υπεύθυνος για την κατανομή εργασιών σε ένα Kubernetes cluster. Περισσότεροι του ενός διαφορετικοί μηχανισμοί χρονοδρομολόγησης μπορούν να χρησιμοποιούνται ταυτόχρονα σε ένα cluster [82]. Κάθε Scheduler ενημερώνεται από τον API Server για τα Pods που δεν έχουν ανατεθεί σε κάποιον υπολογιστικό κόμβο για εκτέλεση, βρίσκονται δηλαδή σε κατάσταση `PENDING`, και έχουν προσδιορίσει στο πεδίο `spec` τους τη χρήση του συγκεκριμένου Scheduler για τη δρομολόγησή τους. Όπως απεικονίζεται στην Εικόνα 4.1, η ανάθεση ενός Pod σε έναν κόμβο αποτελείται από τη φάση χρονοδρομολόγησης (scheduling phase) και τη φάση ανάθεσης (binding phase), οι οποίες στην ορολογία του Kubernetes αναφέρονται συνολικά ως πλαίσιο χρονοδρομολόγησης (scheduling framework) [83].



Εικόνα 4.1: Το πλαίσιο χρονοδρομολόγησης στο Kubernetes

Στη φάση scheduling, ο Scheduler λαμβάνει το πρώτο Pod, για τη χρονοδρομολόγηση του οποίου είναι υπεύθυνος, από την ουρά χρονοδρομολόγησης και αναζητά έναν worker κόμβο για την εκτέλεσή του. Οι διαθέσιμοι κόμβοι με ρόλο worker στο cluster ελέγχονται έναντι ενός συνόλου προκαθορισμένων κανόνων, τα οποία σχετίζονται με τους πόρους και τους περιορισμούς που περιγράφονται στο πεδίο `spec` του Pod, την τρέχουσα και την επιθυμητή κατάσταση του cluster, τους

περιορισμούς των πόρων υλικού και λογισμικού κ.λπ. [82]. Ο Scheduler λαμβάνει όλες τις απαραίτητες πληροφορίες για τις προδιαγραφές του Pod και την κατάσταση του συστήματος από τον API Server μέσω του Kubernetes API.

Αν μετά από αυτούς τους ελέγχους υπάρξει κόμβος που ικανοποιεί όλες τις απαιτήσεις και τους περιορισμούς, και άρα είναι κατάλληλος για την εκτέλεση των containers του Pod, στη φάση binding ο Scheduler αιτείται την ανάθεση του Pod στον συγκεκριμένο κόμβο με κλήση στο Kubernetes API. Αν το αίτημα ανάθεσης γίνει αποδεκτό, ο API Server ενημερώνει το υποπεδίο `nodeName` στο `spec` του Pod με το όνομα του κόμβου [84]. Μόλις το Kubelet του κόμβου ενημερωθεί για την ανάθεση, λαμβάνει την περιγραφή του Pod και δεσμεύει τους απαραίτητους πόρους [85]. Στην περίπτωση που το Pod δεν μπορεί να εκτελεστεί στον κόμβο, ή αν υπάρξει σφάλμα, το πλαίσιο χρονοδρομολόγησης που βρίσκεται σε εξέλιξη ακυρώνεται και το Pod τοποθετείται πάλι στην ουρά χρονοδρομολόγησης ώστε να επιχειρηθεί εκ νέου αργότερα η ανάθεσή του σε κάποιον άλλον κόμβο. Αξίζει να σημειωθεί ότι οι φάσεις scheduling πραγματοποιούνται σειριακά σε ένα cluster, ενώ οι binding μπορούν να εκτελούνται παράλληλα [86].

Στην περίπτωση που δεν οριστεί η χρήση συγκεκριμένου Scheduler στο `spec` του Pod, τη δρομολόγησή του αναλαμβάνει ο Kube-Scheduler, ο προκαθορισμένος μηχανισμός χρονοδρομολόγησης του Kubernetes. Ο Kube-Scheduler επιδιώκει να κατανείμει ομοιόμορφα τα Pods προς χρονοδρομολόγηση στους διαθέσιμους κόμβους του cluster. Πιο συγκεκριμένα, η scheduling φάση του Kube-Scheduler αποτελείται τα στάδια φιλτραρίσματος (filtering step) και βαθμολόγησης (scoring step). Στο filtering στάδιο, ο Kube-Scheduler φιλτράρει τους κόμβους με ρόλο worker του cluster βάσει ενός συνόλου κατηγορημάτων (predicates) και απορρίπτει αυτούς που δε διαθέτουν τους απαιτούμενους πόρους για τη χρονοδρομολόγηση του Pod, δεν είναι συνδεδεμένοι με τις μόνιμες δομές αποθήκευσης που έχουν αιτηθεί ή έχουν απορριφθεί ρητά στο `spec` του Pod. Ο Πίνακας 4.2 συνοψίζει τα ενσωματωμένα predicates στην τρέχουσα έκδοση του Kubernetes (v.1.21), περιγράφεται η λειτουργία τους, καθώς και προσδιορίζεται η σειρά με την οποία εφαρμόζονται διαδοχικά στους κόμβους [87]. Όπως φαίνεται στον πίνακα, ένα υποσύνολο των υποστηριζόμενων predicates χρησιμοποιείται στον προκαθορισμένο αλγόριθμο χρονοδρομολόγησης [88]. Ωστόσο, ο διαχειριστής του cluster μπορεί να επιλέξει ποια από τα ενσωματωμένα predicates θα χρησιμοποιηθούν, να καθορίσει τη σειρά με την οποία θα εφαρμοστούν αλλά και να ορίσει νέα [89].

Αν μετά το filtering στάδιο δεν έχει βρεθεί κανένας κόμβος που ικανοποιεί τις απαιτήσεις του Pod, το Pod δεν μπορεί να εκτελεστεί και η διαδικασία χρονοδρομολόγησης θα τερματιστεί. Διαφορετικά, ο Kube-Scheduler θα προχωρήσει στο scoring στάδιο, αξιολογώντας το υποσύνολο των εναπομεινάντων του προηγούμενου σταδίου κόμβων για να εντοπίσει τον πιο κατάλληλο για την εκτέλεση του Pod. Πιο συγκεκριμένα, το Kubernetes έχει ενσωματωμένο ένα σύνολο συναρτήσεων προτεραιότητας (priority functions) οι οποίες βαθμολογούν καθέναν από τους κόμβους στην κλίμακα 0-10, ανάλογα με το αν είναι λιγότερο ή περισσότερο κατάλληλος για την εκτέλεση του Pod. Η

βαθμολογία κάθε συνάρτησης σταθμίζεται με έναν θετικό αριθμό, το βάρος της. Η τελική βαθμολογία κάθε κόμβου υπολογίζεται ως το άθροισμα των σταθμισμένων επιμέρους βαθμολογιών των συναρτήσεων και ο Kube-Scheduler αναθέτει το Pod στον κόμβο με την υψηλότερη βαθμολογία. Αν περισσότεροι του ενός κόμβοι έχουν την υψηλότερη βαθμολογία, ο Kube-Scheduler επιλέγει τυχαία έναν [88]. Ο Πίνακας 4.3, συνοψίζει τις ενσωματωμένες συναρτήσεις στην τρέχουσα έκδοση του Kubernetes (v.1.21). Μάλιστα, τα βάρη στις συναρτήσεις προτεραιότητας μπορούν να μεταβληθούν, καθώς και είναι εφικτό να οριστούν νέες προτεραιότητες [82], [83], [87], [90].

Συνολικά, η υλοποίηση του Kube-Scheduler χρησιμοποιείται ως πρότυπο για τη δημιουργία νέων μηχανισμών χρονοδρομολόγησης από τους διαχειριστές του cluster βάσει των απαιτήσεων των εκάστοτε εφαρμογών. Τέλος, αξίζει να σημειωθεί ότι από την έκδοση 1.19 του Kubernetes, τα περισσότερα predicates και priority functions έχουν μετατραπεί σε plugins [28] στα σημεία επεκτασιμότητας Filter και Score αντίστοιχα του πλαισίου χρονοδρομολόγησης [86]. Τα σημεία αυτά θα αναλυθούν στην Ενότητα 4.2.2.

Πίνακας 4.2: Τα προκαθορισμένα predicates στην έκδοση 1.21 του Kubernetes[29, 30]

Predicate	**Λειτουργία**	**Προκαθορισμένη χρήση[31]**	**Προκαθορισμένη σειρά εφαρμογής[32]**
CheckNodeUnschedulable[33]	Απορρίπτει τους κόμβους που έχουν σημανθεί στο spec ως `unschedulable`, εκτός αν το Pod επιτρέπει το taint του κόμβου (Ενότητα 4.2.2)	✓	1
GeneralPredicates	Ελέγχει αν ο κόμβος ικανοποιεί τα non-critical και essential predicates τα οποία έχουν οριστεί. Τα non-critical predicates πρέπει να ικανοποιούνται για τα μη κρίσιμα προς χρονοδρομολόγηση Pods, ενώ τα essential χρειάζεται να ικανοποιούνται από όλα τα Pods	✓	2
HostName[34]	Ελέγχει το hostname των containers του Pod	✓	3

28 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/registry.go

29 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L203

30 https://gowalker.org/k8s.io/kubernetes/pkg/scheduler/algorithm/predicates

31 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L191

32 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L135

33 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/nodeunschedulable/node_unschedulable.go#L30

34 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L93

PodFitsHostPorts[35]	Ελέγχει αν ο κόμβος έχει ελεύθερες θύρες δικτύου σύμφωνα με τις προδιαγραφές του spec του Pod		4
MatchNodeSelector[36]	Ελέγχει αν η ετικέτα του κόμβου αντιστοιχεί σε κάποια από αυτές που ενδεχομένως έχουν προσδιοριστεί στο spec του Pod		5
PodFitsResources[37]	Ελέγχει αν ο κόμβος έχει διαθέσιμους πόρους (π.χ. CPU και μνήμη) για την ικανοποίηση των απαιτήσεων του Pod		6
NoDiskConflict[38]	Αξιολογεί αν ο κόμβος έχει τις κατάλληλες δομές αποθήκευσης για την εκτέλεση του Pod	✓	7
PodToleratesNodeTaints[39]	Ελέγχει αν το Pod επιτρέπει τα taints του κόμβου (Ενότητα 4.2.2)	✓	8
CheckNodeLabelPressence[40]	Ελέγχει αν ο κόμβος έχει όλες τις ετικέτες που έχουν προσδιοριστεί στο spec του Pod, ανεξάρτητα από την τιμή τους		9
CheckServiceAffinity[41]	Ελέγχει αν τα Pods του ίδιου Service έχουν ανατεθεί στον ίδιο κόμβο ή στην ίδια τοπολογία κόμβων		10
MaxCSIVolumeCount[42]	Αποφασίζει το πλήθος των Container Storage Interface (CSI) δομών αποθήκευσης που θα χρησιμοποιηθούν και αν αυτό υπερβαίνει κάποιο προκαθορισμένο όριο	✓	11
CheckVolumeBinding[43]	Ελέγχει αν το Pod μπορεί να εκτελεστεί βάσει των δομών αποθήκευσης που αιτείται	✓	12
NoVolumeZoneConflict[44]	Ελέγχει αν οι δομές αποθήκευσης που αιτείται το Pod είναι διαθέσιμες στον κόμβο	✓	13

35 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/nodeports/node_ports.go#L29

36 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/nodename/node_name.go

37 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/noderesources/fit.go#L49

38 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/volumerestrictions/volume_restrictions.go#L47

39 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/tainttoleration/taint_toleration.go#L31

40 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/nodelabel/node_label.go#L64

41 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/serviceaffinity/service_affinity.go#L89

42 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/nodevolumelimits/csi.go#L48

43 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/volumebinding/volume_binding.go#L71

44 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/volumezone/volume_zone.go#L36

MatchInterPodAffinity[45]	Ελέγχει αν ο κόμβος ικανοποιεί τους affinity/anti-affinity καθορισμένους κανόνες (Ενότητα 4.2.2)	✓	14

Πίνακας 4.3: Οι προκαθορισμένες priority functions στην έκδοση 1.21 του Kubernetes

Priority function	**Λειτουργία**	**Προκαθορισμένη χρήση[46]**	**Προκαθορισμένο βάρος[47]**
EqualPriority	Προσδίδει σε όλους τους κόμβους μοναδιαίο βάρος		-
MostRequestedPriority	Δίνει μεγαλύτερη προτεραιότητα στους κόμβους με τους περισσότερους διαθέσιμους πόρους. Βάσει αυτής της στρατηγικής, τα προς εκτέλεση Pods θα ανατεθούν στον μικρότερο δυνατό αριθμό κόμβων		-
RequestedToCapacityRatioPriority	Δίνει προτεραιότητα στους κόμβους βάσει των χρησιμοποιούμενων πόρων στον κόμβο συγκριτικά με άλλους. Η προκαθορισμένη συμπεριφορά στο σύστημα είναι παρόμοια με τη LeastRequestedPriority priority function		-
SelectorSpreadPriority	Ελαχιστοποιεί το πλήθος των Pods που ανήκουν στο ίδιο Service ή replication controller και ανατίθενται για εκτέλεση στον ίδιο κόμβο	✓	1
InterPodAffinityPriority	Δίνει προτεραιότητα στους κόμβους ανάλογα με το αν ικανοποιούν τους affinity και anti-affinity κανόνες που έχουν καθοριστεί στο spec του Pod (Ενότητα 4.2.2)	✓	1
LeastRequestedPriority	Δίνει προτεραιότητα στους κόμβους στους οποίους χρησιμοποιούνται οι λιγότεροι πόροι	✓	1
BalancedResourceAllocation	Δίνει προτεραιότητα στους κόμβους με ισορροπημένη χρήση πόρων	✓	1
NodeAffinityPriority	Δίνει προτεραιότητα στους κόμβους βάσει των Preferred During Scheduling Ignored During Execution affinity κανόνων που έχουν προσδιοριστεί στο πεδίο spec του Pod (Ενότητα 4.2.2)	✓	1
TaintTolerationPriority	Δίνει προτεραιότητα στους κόμβους βάσει του πλήθους των taints που δεν ικανοποιούνται στον κόμβο (Ενότητα 4.2.2)	✓	1
ImageLocalityPriority	Δίνει προτεραιότητα στους κόμβους που έχουν ήδη αποθηκευμένη στην cache την container image του Pod	✓	1

45 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/interpodaffinity/plugin.go#L45

46 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L207

47 https://github.com/kubernetes/kubernetes/blob/a5cf298a95b4a60789a6ba1caab931bdc980aa2d/pkg/scheduler/framework/plugins/legacy_registry.go#L207

EvenPodsSpreadPriority	Επιδιώκει την επιθυμητή κατανομή, όπως έχει προσδιοριστεί από τον διαχειριστή του cluster, μιας ομάδας Pods στην τοπολογία κόμβων [91]	✓	2

4.2 Παραμετροποίηση και Επεκτασιμότητα των Μηχανισμών Χρονοδρομολόγησης

4.2.1 Docker SwarmKit και Swarm Mode

Το Docker SwarmKit παρέχει στους χρήστες τέσσερις τρόπους παρέμβασης στον αλγόριθμο επιλογής υπολογιστικών κόμβων για τη χρονοδρομολόγηση των tasks κάθε service, σε συμφωνία με τη Spread στρατηγική χρονοδρομολόγησης.

Πρώτον, μπορούν προσδιοριστούν οι απαιτήσεις κάθε service σε CPU και μνήμη (resource requirements), ώστε τα tasks του να εκτελεστούν σε κόμβους που τις ικανοποιούν. Αυτό μπορεί να γίνει, όπως φαίνεται στην Εικόνα 4.2 [92], κατά τον ορισμό ενός service στο YAML αρχείο Docker Compose, ορίζοντας το πεδίο `reservations`, το οποίο ανήκει στο πεδίο `deploy`, και καθορίζοντας τα υποπεδία `cpus` και `memory`. Ο scheduler μηχανισμός απορρίπτει τους κόμβους που δεν ικανοποιούν αυτές τις απαιτήσεις κατά την εφαρμογή του φίλτρου `ResourceFilter`. Επίσης, στο YAML αρχείο δίνεται η δυνατότητα καθορισμού στο υποπεδίο `limits` άνω ορίων CPUs και μνήμης που μπορούν να χρησιμοποιηθούν από ένα δρομολογημένο task. Ωστόσο, ο καθορισμός αυτών των ορίων δεν εγγυάται ότι το Docker Engine δε θα χρησιμοποιήσει περισσότερους πόρους του κόμβου εκτέλεσης, αν χρειαστεί [93].

Δεύτερον, ο χρήστης μπορεί να καθορίσει το σύνολο των κόμβων εκτέλεσης, ορίζοντας περιορισμούς τοποθέτησης (placement constraints) στην περιγραφή του service και τις αντίστοιχες ετικέτες στους κόμβους του swarm. Όπως φαίνεται στην Εικόνα 4.3 [92], ο ορισμός των περιορισμών γίνεται στο υποπεδίο `constraints` του πεδίου `placement`, το οποίο ανήκει στο πεδίο `deploy`. Κατά την επιλογή του κόμβου ανάθεσης του task, ο scheduler ελέγχει αν ο εκάστοτε κόμβος ικανοποιεί τους περιορισμούς όταν εφαρμόζει το φίλτρο `ConstraintFilter`. Μεταξύ των χαρακτηριστικών των κόμβων και του Docker Engine για τα οποία μπορούν να οριστούν περιορισμοί είναι: το ID και το όνομα, ο ρόλος (manager ή worker), το λειτουργικό σύστημα και η αρχιτεκτονική (π.χ. x86_64) του κόμβου, όπως και το λειτουργικό σύστημα και οι drivers του χρησιμοποιούμενου Docker Engine. Οι καθορισμένοι περιορισμοί του service μπορούν είτε να αντιστοιχίζονται (==) με την ετικέτα και την τιμή του υπό εξέταση κόμβου, είτε να τον αποκλείουν (!=). Αν μετά το `ConstraintFilter` δε βρεθεί κόμβος που να ικανοποιεί τους περιορισμούς, ο κύκλος χρονοδρομολόγησης θα αποτύχει και τα tasks θα παραμείνουν σε κατάσταση `PENDING`. Ο scheduler θα επιδιώξει ξανά την εύρεση κόμβων

για την εκτέλεση των tasks και θα τα δρομολογήσει όταν βρεθούν κόμβοι που ικανοποιούν τους περιορισμούς [93], [45].

Τρίτον, ο χρήστης μπορεί να προσδιορίσει στην περιγραφή του service προτιμήσεις τοποθέτησης (placement preferences) [48] για τους κόμβους στους οποίους θα δρομολογηθούν τα tasks. Πιο συγκεκριμένα, αφού ο scheduler απορρίψει τους κόμβους που δεν ικανοποιούν τις απαιτήσεις σε πόρους και τους περιορισμούς των χρηστών, εφαρμόζεται η Spread στρατηγική χρονοδρομολόγησης για να εντοπιστούν οι κόμβοι με το λιγότερο φορτίο και να κατανεμηθούν ομοιόμορφα σε αυτούς τα προς εκτέλεση tasks. Σε αυτό το σημείο ο scheduler λαμβάνει υπόψη τις προτιμήσεις τοποθέτησης των χρηστών.

```
version: "{{ site.compose_file_v3 }}"
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

Εικόνα 4.2: Ορισμός απαιτήσεων σε πόρους στην περιγραφή του service

Οι προτιμήσεις τοποθέτησης λειτουργούν ως «χαλαροί» περιορισμοί, δεδομένου ότι ικανοποιούνται αν είναι εφικτό. Όταν έχει οριστεί μια προτίμηση τοποθέτησης, ο scheduler δίνει προτεραιότητα στους υποψήφιους κόμβους που την ικανοποιούν κατά την ανάθεση των tasks. Ωστόσο, αν το φορτίο δεν μπορεί να κατανεμηθεί ομοιόμορφα μεταξύ αυτών των κόμβων, η διαδικασία χρονοδρομολόγησης του service δεν αποτυγχάνει και τα επιπλέον tasks ανατίθενται για εκτέλεση σε άλλους κόμβους που πληρούν τις απαιτήσεις σε πόρους και τους περιορισμούς τοποθέτησης των χρηστών, ακόμα και αν δεν ικανοποιούν τις προτιμήσεις τους. Αξίζει να σημειωθεί μάλιστα ότι tasks ανατίθενται και σε κόμβους χωρίς ετικέτες, αφού στο Docker SwarmKit η έλλειψη ετικέτας ισοδυναμεί με ετικέτα με μηδενική τιμή.

Οι προτιμήσεις τοποθέτησης αναφέρονται τόσο στα χαρακτηριστικά του κόμβου όσο και του Docker Engine και η χρήση τους προϋποθέτει τον ορισμό κατάλληλων ετικετών. Πιο συγκεκριμένα, δεδομένου ότι οι κόμβοι του swarm ή και το Docker Engine έχουν σημανθεί με ετικέτες, οι χρήστες χρειάζεται να προσδιορίσουν αυτές που τους ενδιαφέρουν στο αρχείο YAML της περιγραφής του service. Όπως φαίνεται στην Εικόνα 4.3, οι επιθυμητές ετικέτες και οι τιμές τους περιγράφονται στο υποπεδίο `preferences` του πεδίου `placement`. Οι περιγραφές των ετικετών είναι της μορφής `node.labels.<ONOMA_ETIKETAΣ>` και `engine.labels.<ONOMA_ETIKETAΣ>` για προτιμήσεις που αφορούν τους κόμβους και τα χαρακτηριστικά του Docker Engine αντίστοιχα. Οι ετικέτες

[48] https://github.com/docker/swarmkit/blob/17d8d4e4d8bdec33d386e6362d3537fa9493ba00/manager/scheduler/scheduler.go#L577

προτιμήσεων του SwarmKit θα μπορούσαν μελλοντικά να αναφέρονται και σε άλλα χαρακτηριστικά του swarm, όπως τους διαθέσιμους πόρους.

```
version: "{{ site.compose_file_v3 }}"
services:
  db:
    image: postgres
    deploy:
      placement:
        constraints:
          - "node.role==manager"
          - "engine.labels.operatingsystem==ubuntu 18.04"
        preferences:
          - spread: node.labels.zone
```

Εικόνα 4.3: Ορισμός περιορισμών και προτιμήσεων στην περιγραφή του service

Μάλιστα, αξίζει να σημειωθεί ότι το SwarmKit επιτρέπει στους διαχειριστές του cluster να ορίσουν μία ιεραρχία προτιμήσεων τοποθέτησης, καθορίζοντας πολλαπλές προτιμήσεις για κάθε service. Οι προτιμήσεις αυτές λαμβάνονται υπόψη από τον scheduler με τη σειρά που έχουν οριστεί στην περιγραφή του service. Ένα τυπικό παράδειγμα χρήσης αυτής της λειτουργίας είναι όταν τα προς εκτέλεση tasks πρέπει να κατανεμηθούν ομοιόμορφα σε μία ιεραρχική τοπολογία X κεντρικών υποδομών, με την κάθε μία να αποτελείται από Y σειρές και Z στοίβες υπολογιστικών κόμβων (racks). Σε αυτή την περίπτωση, χρειάζεται να καθοριστούν τρεις ετικέτες προτιμήσεων τοποθέτησης, αρχικά στο επίπεδο των κεντρικών υποδομών, μετά των σειρών και τέλος των στοιβών τους. Με τη σειρά αυτή, ο scheduler θα επιδιώξει πρώτα τα κατανείμει ομοιόμορφα τα tasks μεταξύ των κεντρικών υποδομών και ακολούθως στις διαθέσιμες σειρές και στις στοίβες υπολογιστικών κόμβων τους [93], [45], [81].

Τέλος, δίνεται η δυνατότητα χρήσης μόνο workers κόμβων του swarm για την εκτέλεση των tasks. Για να επιτευχθεί αυτό, η διαθεσιμότητα των manager κόμβων (πεδίο `availability`) πρέπει να αποκτήσει την τιμή `DRAIN`, με την εντολή `docker node update --availability drain <NODE-ID>`. Μετά τη μεταβολή της διαθεσιμότητας, ο scheduler διακόπτει τα tasks που εκτελούνται σε αυτούς τους κόμβους και επιχειρεί τη δρομολόγησή τους σε κόμβους με διαθεσιμότητα `ACTIVE` [34], [94].

4.2.2 Kubernetes

Μεταξύ των λόγων για τους οποίους το Kubernetes χαρακτηρίζεται ως ένα εξαιρετικά επεκτάσιμο σύστημα ενορχήστρωσης είναι οι δυνατότητες που παρέχει στους χρήστες να παρεμβαίνουν στον αλγόριθμο χρονοδρομολόγησης, χωρίς την τροποποίηση του πηγαίου κώδικα της πλατφόρμας. Σημειώνεται ωστόσο ότι όλες οι λειτουργίες παραμετροποίησης που θα αναλυθούν πρέπει να χρησιμοποιούνται με φειδώ αφού έχουν υψηλές υπολογιστικές απαιτήσεις [83].

Περιγραφή των περιορισμών και των προτιμήσεων των χρηστών για τους κόμβους εκτέλεσης στον ορισμό του Pod

Αρχικά, το Kubernetes επιτρέπει την περιγραφή απαιτήσεων χρονοδρομολόγησης κατά τον ορισμό του Pod χρησιμοποιώντας επιλογείς ετικετών. Ο πιο απλός τρόπος είναι να καθοριστεί το όνομα ενός συγκεκριμένου κόμβου στο υποπεδίο `nodeName` του πεδίου `spec` του Pod, όπως φαίνεται στην Εικόνα 4.4. Σε αυτή την περίπτωση, ο Scheduler δεν εκκινεί πλαίσιο χρονοδρομολόγησης για το Pod, όλες οι υπόλοιπες μέθοδοι επιλογής κόμβου αγνοούνται και το Kubelet του προσδιορισμένου κόμβου επιχειρεί απευθείας την εκτέλεση του Pod. Στην πράξη, αυτή η μέθοδος πρέπει να χρησιμοποιείται μόνο σε περιβάλλοντα ανάπτυξης και ελέγχου λόγω εγγενών περιορισμών της όπως: αποτυχία του Pod αν ο κόμβος δεν υπάρχει ή δε διαθέτει τους απαραίτητους πόρους για την εκτέλεση του container, μη διαχειριζόμενα προβλήματα δικτύου των Pods, πιθανόν άγνωστες συμβάσεις για τα ονόματα των πόρων σε υποδομές εξωτερικών παρόχων.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: kube-01
```

Εικόνα 4.4: Καθορισμός του πεδίου nodeName στο PodSpec

Μέρος αυτών των προβλημάτων μπορεί να αντιμετωπιστεί, χρησιμοποιώντας εναλλακτικά το υποπεδίο `nodeSelector` του `spec` του Pod, όπως φαίνεται στην Εικόνα 4.5. Σε αυτό το υποπεδίο δεν καθορίζεται μονοσήμαντα ένας κόμβος για την εκτέλεση του Pod. Αντίθετα, προσδιορίζεται ένα σύνολο ετικετών και των τιμών τους οι οποίες περιγράφουν συνήθως μια ομάδα υποψήφιων κόμβων.

Συνεχίζοντας, μία πιο περιγραφική μέθοδος, η οποία διευκολύνει τον προσδιορισμό σύνθετων απαιτήσεων χρονοδρομολόγησης, είναι ορίζοντας χαρακτηριστικά προτίμησης (affinity features). Μέσω των χαρακτηριστικών αυτών περιγράφονται τρεις τύποι κανόνων χρονοδρομολόγησης στο πεδίο `affinity` του `spec` του Pod: προτίμηση ως προς τους κόμβους (υποπεδίο `nodeAffinity`), προτίμηση ως προς τα Pods (υποπεδίο `podAffinity`) και απόρριψη ως προς τα Pods (υποπεδίο `podAntiAffinity`).

Το υποπεδίο `nodeAffinity` έχει παρόμοια λειτουργία με το `nodeSelector`. Ορίζοντας επιλογείς ετικετών στο πεδίο `spec` του Pod, οι χρήστες καθορίζουν τους κόμβους στους οποίους μπορεί αυτό να εκτελεστεί. Ωστόσο, το πεδίο `nodeAffinity` υποστηρίζει περισσότερους κανόνες για την αντιστοίχιση των ζευγών κλειδιού-τιμής των ετικετών του Pod με αυτές των κόμβων (υποπεδίο `matchExpressions`) χρησιμοποιώντας τελεστές (υποπεδίο `operators`). Για παράδειγμα πέρα από την απόλυτη συμφωνία των κλειδιών και των τιμών δύο ετικετών, μπορεί να χρησιμοποιηθεί ο

τελεστής `In` και να καθοριστούν περισσότερες από μία αποδεκτές τιμές για μία ετικέτα. Επίσης, στο πεδίο `nodeAffinity` ορίζονται τόσο περιορισμοί, τους οποίους είναι απαραίτητο να πληροί ο κόμβος που θα επιλεχθεί για την εκτέλεση του Pod, όσο και προτιμήσεις των χρηστών οι οποίες ικανοποιούνται αν είναι εφικτό.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Εικόνα 4.5: Καθορισμός του πεδίου nodeSelector στο PodSpec

Πιο συγκεκριμένα, στην τρέχουσα έκδοση του Kubernetes (v.1.21) οι κανόνες που περιγράφονται στο πεδίο `nodeAffinity` μπορούν να ανήκουν είτε στην κατηγορία «Required During Scheduling Ignored During Execution» είτε στην «Preferred During Scheduling Ignored During Execution», όπως φαίνεται στην Εικόνα 4.6. Η πρώτη κατηγορία αφορά τους περιορισμούς που πρέπει να ικανοποιεί ένας κόμβος για του ανατεθεί το προς εκτέλεση Pod (π.χ. εκτέλεση του Pod μόνο στον κόμβο που διαθέτει την X κάρτα γραφικών). Η δεύτερη κατηγορία αφορά τις προτιμήσεις των χρηστών, τις οποίες ο Scheduler θα προσπαθήσει να ικανοποιήσει κατά την επιλογή του κόμβου. Κάθε προτίμηση ορίζεται συνδυαστικά με ένα θετικό αριθμό στο εύρος 1-100, με το 1 να αναφέρεται στην προτίμηση μικρότερης προτεραιότητας. Στη διαδικασία επιλογής κόμβου για την εκτέλεση του Pod, ο Scheduler αρχικά θα απορρίψει τους κόμβους που δεν ικανοποιούν τους αυστηρούς περιορισμούς και στη συνέχεια θα ελέγξει σειριακά τις προτιμήσεις που έχουν προσδιοριστεί, βαθμολογώντας κάθε εναπομείναντα κόμβο με την αντίστοιχη τιμή αν ικανοποιεί την προτίμηση. Ακολούθως, ο Scheduler θα υπολογίσει την τελική βαθμολογία κάθε κόμβου όπως προκύπτει από το άθροισμα των τιμών των προτιμήσεων που ικανοποιεί, καθώς και της βαθμολογίας από την εφαρμογή των συναρτήσεων προτεραιότητας όπως περιεγράφηκαν στην Ενότητα 4.1.2. Τελικά, θα επιλεχθεί για την εκτέλεση του Pod κάποιος από τους κόμβους με την υψηλότερη βαθμολογία.

Ωστόσο, ένας από τους περιορισμούς χρήσης του πεδίου `nodeAffinity` είναι ότι η τρέχουσα έκδοση του Kubernetes υποστηρίζει μόνο κατηγορίες «IgnoredDuringExecution». Με άλλα λόγια, όταν ένα Pod ανατεθεί για εκτέλεση σε έναν κόμβο, θα συνεχίσει να εκτελείται σε αυτόν ακόμα και αν οι ετικέτες του κόμβου αλλάξουν στον χρόνο εκτέλεσης. Σε μελλοντικές εκδόσεις του Kubernetes οι λειτουργίες αυτού του πεδίου αναμένεται να επεκταθούν [95].

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

Εικόνα 4.6: Καθορισμός του πεδίου nodeAffinity στο PodSpec

Δύο άλλα υποπεδία του `affinity` είναι τα `podAffinity` και `podAntiAffinity`. Τα υποπεδία αυτά, εκτός του ότι παρέχουν μια πιο εκφραστική σύνταξη για την περιγραφή των περιορισμών και των προτιμήσεων, επιτρέπουν στους χρήστες να απορρίψουν κόμβους για την εκτέλεση του εκάστοτε Pod βάσει των ετικετών των Pods που εκτελούνται ήδη σε αυτούς. Οι κανόνες που περιγράφονται σε αυτά τα υποπεδία είναι της μορφής «αυτό το Pod πρέπει, για το υποπεδίο `podAffinity` – ή δεν πρέπει, για το υποπεδίο `podAntiAffinity` – να εκτελεστεί στους κόμβους της τοπολογίας X, αν σε αυτούς εκτελούνται ήδη Pods που ικανοποιούν τον κανόνα Y». Η τοπολογία μπορεί να αφορά έναν μεμονωμένο κόμβο, ένα rack ή μία ζώνη του παρόχου υποδομών υπολογιστικού νέφους, και προσδιορίζεται από την τιμή μιας ετικέτας με την οποία έχουν σημανθεί όλοι οι υπολογιστικοί κόμβοι της. Ο κανόνας Y ορίζεται χρησιμοποιώντας επιλογείς ετικετών στην περιγραφή των Pods.

Όπως γίνεται αντιληπτό από την Εικόνα 4.7, οι κατηγορίες «Required During Scheduling Ignored During Execution» για αυστηρούς περιορισμούς (π.χ. τοποθέτηση των Pods του Service A στην ίδια τοπολογία με τα Pods του Service B) και «Preferred During Scheduling Ignored During Execution» για τις προτιμήσεις των χρηστών (π.χ. ομοιόμορφη κατανομή των Pods του Service A στην εκάστοτε τοπολογία, αν είναι εφικτό) υποστηρίζονται και στα υποπεδία `podAffinity` και `podAntiAffinity`. Αντίστοιχα, αναμένεται σε μελλοντικές εκδόσεις της πλατφόρμας η επέκταση σε κατηγορίες «Required During Execution». Αξίζει να σημειωθεί ωστόσο ότι η χρήση αυτών των υποπεδίων προϋποθέτει συνεπή σήμανση των κόμβων και των Pods με ετικέτες, ενώ η επεξεργασία

τους έχει αυξημένες υπολογιστικές απαιτήσεις που ενδέχεται να δυσχεραίνουν τη χρονοδρομολόγηση σε μεγάλα clusters [95].

Τέλος, σύνθετες οδηγίες χρονοδρομολόγησης περιγράφονται χρησιμοποιώντας το πεδίο `taints` στους κόμβους και το πεδίο `tolerations` στα Pods. Μέσω αυτών των πεδίων, ο χρήστης καθορίζει για κάθε κόμβο τα Pods που μπορούν να εκτελεστούν σε αυτόν, αντί να συσχετίζει κάθε Pod με ένα σύνολο κόμβων. Συγκεκριμένα, ένας ή περισσότεροι κανόνες μπορούν να προσδιοριστούν στο πεδίο `taints` ενός υπολογιστικού κόμβου, με τον καθένα να περιγράφεται από τα υποπεδία `key`, `value` και `effect`.

Ένα Pod μπορεί να εκτελεστεί σε έναν κόμβο αν έχει οριστεί toleration που αντιστοιχίζεται σε ένα από τα taints του κόμβου. Το πεδίο `tolerations` περιλαμβάνεται στο `spec` του Pod και αποτελείται από τα υποπεδία `key`, `value`, `effect` και `operator`, όπως φαίνεται στην Εικόνα 4.8. Οι τιμές που λαμβάνει το πεδίο `operator` είναι `Exists` και `Equal`. Όταν η τιμή του `operator` είναι `Exists`, τότε το Pod μπορεί να εκτελεστεί στον κόμβο αν τα αντίστοιχα πεδία `key` και `effect` είναι ίδια. Αν το πεδίο `operator` έχει τιμή `Equal`, πρέπει επιπλέον τα αντίστοιχα πεδία `value` να είναι ίδια.

Συνεχίζοντας, το πεδίο `effect` μπορεί να πάρει μία από τις τιμές `NoSchedule`, `PreferNoSchedule` και `NoExecute`, τα ονόματα των οποίων είναι δηλωτικά των λειτουργιών τους. Συγκεκριμένα, προσδιορίζουν το σύνολο των ενεργειών που θα εκτελέσει ο Scheduler αν τα tolerations του προς εκτέλεση Pod δεν αντιστοιχίζονται με τα taints του υπό εξέταση κόμβου [96]. Αν το πεδίο `effect` έχει τιμή `NoSchedule`, ο Scheduler δε θα αναθέσει το Pod σε κόμβο που δεν ικανοποιεί τα tolerations του, ενώ αν έχει τιμή `PreferNoSchedule` θα επιδιώξει να μην πραγματοποιηθεί αυτή η ανάθεση, αν είναι εφικτό. Στην περίπτωση της τιμής `NoExecute`, το Pod δε θα δρομολογηθεί σε κόμβο που δεν ικανοποιεί τα tolerations του και αν ήδη εκτελείται σε έναν τέτοιο κόμβο, θα διακοπεί η εκτέλεσή του. Τέλος, αξίζει να σημειωθεί ότι τα πεδία `taints` και `tolerations` χρησιμοποιούνται συνήθως όταν υπάρχουν κόμβοι με εξειδικευμένους πόρους (π.χ. GPUs), οι οποίοι είναι προτιμότερο να χρησιμοποιηθούν από Services που αξιοποιούν τις δυνατότητές τους, ή όταν ένα υποσύνολο των κόμβων χρησιμοποιείται αποκλειστικά από συγκεκριμένες ομάδες χρηστών και για συγκεκριμένες εφαρμογές, ή όταν τα Pods δεν πρέπει να ανατεθούν σε κόμβους που έχουν παρουσιάσει προβλήματα λειτουργίας [97], [85].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - web-store
            topologyKey: "kubernetes.io/hostname"
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - store
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-app
        image: nginx:1.16-alpine
```

Εικόνα 4.7: Καθορισμός των πεδίων podAffinity και podAntiAffinity στο PodSpec

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

Εικόνα 4.8: Καθορισμός του πεδίου tolerations στο PodSpec

Επέκταση των προκαθορισμένων μηχανισμών χρονοδρομολόγησης

Πέρα από τον έλεγχο των κόμβων στους οποίους θα εκτελεστεί το Pod βάσει των περιορισμών και των προτιμήσεων που προσδιορίζονται στον ορισμό του, το Kubernetes παρέχει τη δυνατότητα επέκτασης των προκαθορισμένων αλγορίθμων χρονοδρομολόγησης με τέσσερις τρόπους. Οι προσεγγίσεις αυτές δεν είναι αλληλοαποκλειόμενες και μπορούν να χρησιμοποιηθούν συνδυαστικά, ανάλογα με τις απαιτήσεις των εφαρμογών και τις τοπολογίες του cluster.

Πρώτον, ο πιο προφανής τρόπος είναι επεμβαίνοντας στον πηγαίο κώδικα. Για τον σκοπό αυτό, θα πρέπει να δημιουργηθεί αντίγραφο του κώδικα του Kubernetes από το δημόσιο Git repository του[49] και αφού γίνουν οι απαραίτητες αλλαγές, να μεταγλωττιστεί ξανά. Ο τρόπος αυτός δεν προτιμάται λόγω της αυξημένης πολυπλοκότητας ενσωμάτωσης των αλλαγών στις υπάρχουσες λειτουργίες της πλατφόρμας.

Δεύτερον, μία καλύτερη προσέγγιση είναι η δημιουργία webhooks, με τα οποία μπορούν να επεκταθούν οι προκαθορισμένες λειτουργίες των σταδίων Filter, Preempt, Prioritize και Bind της φάσης scheduling του Kube-Scheduler. Σύμφωνα με αυτήν την προσέγγιση, μπορεί να ενταχθεί μετά από το εκάστοτε επιλεγμένο στάδιο του πλαισίου χρονοδρομολόγησης, ένα βήμα επεξεργασίας των υποψήφιων κόμβων που έχουν προκύψει, χρησιμοποιώντας κλήσεις HTTP(s). Για παράδειγμα, μετά το στάδιο Filter, μια λειτουργία επέκτασης με webhook θα μπορούσε να λαμβάνει τους υποψήφιους κόμβους με HTTP(s) κλήσεις και να ελέγχει αν ικανοποιούν και άλλους περιορισμούς ή προτιμήσεις. Συνήθως, τα webhooks χρησιμοποιούνται όταν ο Scheduler καλείται να λάβει αποφάσεις ανάθεσης σε τοπολογίες με πόρους στους οποίους έχει περιορισμένα δικαιώματα πρόσβασης, για παράδειγμα όταν βρίσκονται εκτός του cluster.

Η επέκταση μιας λειτουργίας χρονοδρομολόγησης με χρήση webhooks μπορεί να επιτευχθεί με προγραμματιστικό τρόπο. Αρχικά, δημιουργείται βάσει των συμβάσεων του Kubernetes ένα object τύπου KubeSchedulerConfiguration, προσδιορίζοντας στο υποπεδίο `policy` του πεδίου `algorithmSource` ένα τοπικό αρχείο περιγραφής σε JSON ή ένα object τύπου ConfigMap που έχει ήδη δημιουργηθεί. Στην Εικόνα 4.9 παρουσιάζεται ένα παράδειγμα ορισμού object τύπου KubeSchedulerConfiguration σε αρχείο YAML, όπου στο πεδίο `algorithmSource` έχει προσδιοριστεί ένα τοπικό αρχείο σε JSON. Στη συνέχεια, στο JSON αρχείο ή στο ConfigMap ορίζεται ένα object τύπου Policy. Στο `extenders` πεδίο αυτού του object χρησιμοποιούνται τα κατάλληλα υποπεδία[50] για να προσδιοριστεί το στάδιο της φάσης scheduling που θα επεκταθεί, καθώς και το σημείο που θα αποσταλούν για περαιτέρω επεξεργασία τα αποτελέσματα της προκαθορισμένης λειτουργίας του Kube-Scheduler αυτού του σταδίου. Στην Εικόνα 4.10 απεικονίζεται το JSON αρχείο που χρησιμοποιήθηκε προηγουμένως (Εικόνα 4.9), σύμφωνα με το οποίο οι υποψήφιοι κόμβοι που προκύπτουν μετά τα στάδια Filter και Prioritize αποστέλλονται για περαιτέρω επεξεργασία στα endpoints `<urlPrefix>/<filterVerb>` και `<urlPrefix>/ <prioritizeVerb>` αντιστοίχως. Τέλος, χρειάζεται να υλοποιηθούν στην επιλεγμένη γλώσσα προγραμματισμού οι μέθοδοι του webhook που θα λαμβάνουν και θα ικανοποιούν τις κλήσεις `POST` του σταδίου που επεκτείνεται [98], [99]. Στο GitHub repository[51] παρουσιάζεται ένα παράδειγμα υλοποίησης σε Go.

[49] https://github.com/kubernetes/kubernetes
[50] https://kubernetes.io/docs/reference/config-api/kube-scheduler-config.v1beta1/#kubescheduler-config-k8s-io-v1-LegacyExtender
[51] https://github.com/Huang-Wei/sample-scheduler-extender

```
# content of the file passed to "--config"
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
clientConnection:
  kubeconfig: "/var/run/kubernetes/scheduler.kubeconfig"
algorithmSource:
  policy:
    file:
      path: "/root/config/scheduler-extender-policy.json"
```

Εικόνα 4.9: YAML αρχείο ορισμού του KubeSchedulerConfiguration object

```
{
    "kind" : "Policy",
    "apiVersion" : "v1",
    "extenders" : [{
        "urlPrefix": "http://localhost:8888/",
        "filterVerb": "filter",
        "prioritizeVerb": "prioritize",
        "weight": 1,
        "enableHttps": false
    }]
}
```

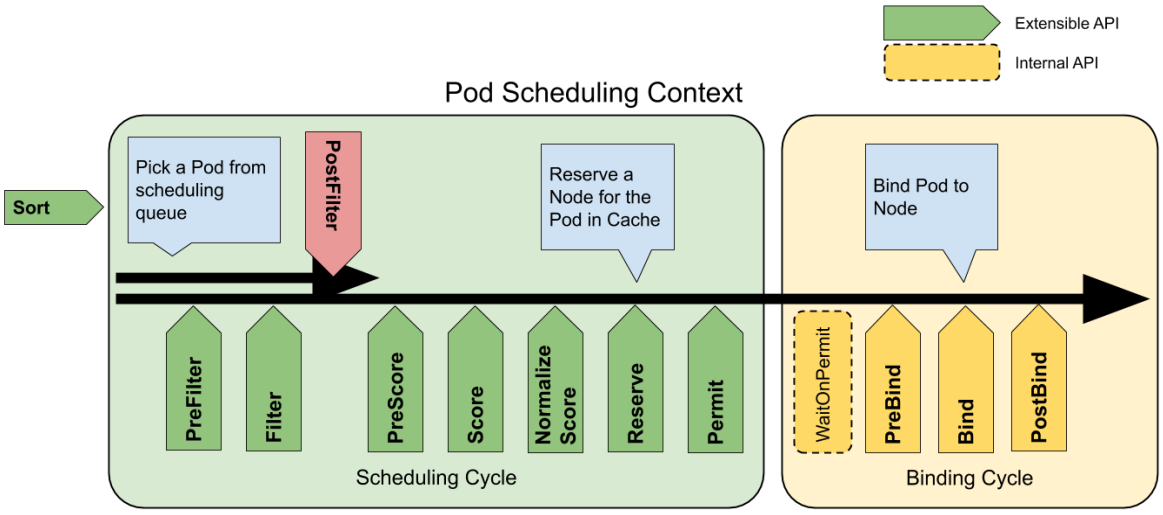
Εικόνα 4.10: JSON αρχείο ορισμού του Policy object

Η χρήση webhooks για την επέκταση των προκαθορισμένων λειτουργιών χρονοδρομολόγησης του Kube-Scheduler έχει ένα σύνολο περιορισμών. Αρχικά, τα σημεία του αλγορίθμου χρονοδρομολόγησης που μπορούν να επεκταθούν είναι περιορισμένα, αφού περαιτέρω επεξεργασία στους εκάστοτε υποψήφιους κόμβους μπορεί να γίνει μόνο στο τέλος των τεσσάρων σταδίων χρονοδρομολόγησης που προαναφέρθηκαν. Επίσης, η χρήση των πόρων δικτύου για τη μεταφορά των δεδομένων μέσω HTTP(s) κλήσεων, καθώς και για τις λειτουργίες σειριοποίησης είναι αυξημένη, συγκριτικά με τις εγγενείς μεθόδους στον πηγαίο κώδικα του Kube-Scheduler. Επιπλέον, αυτός ο τρόπος επέκτασης είναι κατάλληλος όταν απαιτείται περαιτέρω επεξεργασία των υποψήφιων κόμβων που έχουν προκύψει από τις προκαθορισμένες λειτουργίες χρονοδρομολόγησης. Ωστόσο, τα webhooks δεν μπορούν να χρησιμοποιηθούν αν είναι θεμιτό να προστεθούν στους υποψήφιους κόμβους του επόμενου σταδίου χρονοδρομολόγησης νέοι κόμβοι διότι δεν μπορεί να εξασφαλιστεί ότι θα ικανοποιούν τους ελέγχους των προηγούμενων σταδίων. Τέλος, τα webhooks ως εξωτερικοί μηχανισμοί δεν ενημερώνονται με αυτοματοποιημένο τρόπο για την τρέχουσα κατάσταση του cluster από τον API Server. Έτσι, ο διαχειριστής του cluster χρειάζεται μεριμνήσει για την υλοποίηση των απαιτούμενων μηχανισμών ενημερώσεων [99].

Για την επέκταση των προκαθορισμένων λειτουργιών χρονοδρομολόγησης ενδείκνυται, από την έκδοση v.1.15 του Kubernetes[52], η αξιοποίηση του Kubernetes Scheduling Framework. Σύμφωνα με αυτόν τον τρίτο τρόπο επέκτασης, ορίζονται σημεία παρέμβασης στον αλγόριθμο του Kube-Scheduler και υλοποιούνται APIs για την πρόσβαση σε αυτά. Τα APIs με τη σειρά τους χρησιμοποιούνται από plugins, στα οποία υλοποιούνται οι επιθυμητές λειτουργίες επέκτασης. Με αυτόν τον τρόπο, ένα

[52] https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.15.md#additional-notable-feature-updates

plugin μπορεί να συνδέεται σε πολλαπλά σημεία επέκτασης. Όπως προαναφέρθηκε στην Ενότητα 4.1.2, το πλαίσιο χρονοδρομολόγησης ενός Pod αποτελείται από τις φάσεις scheduling και binding. Στην Εικόνα 4.11 απεικονίζονται τα σημεία παρέμβασης στον αλγόριθμο του Kube-Scheduler που υποστηρίζονται από το Kubernetes Scheduling Framework σε αυτές τις δύο φάσεις [86].



Εικόνα 4.11: Το Kubernetes Scheduling Framework

Η πλατφόρμα του Kubernetes υποστηρίζει προκαθορισμένα plugins στα διάφορα σημεία επέκτασης του Scheduling Framework, τα οποία συνοψίζονται στον Πίνακα 4.4 [100]. Μάλιστα, όπως προαναφέρθηκε στην Ενότητα 4.1.2, από την έκδοση 1.19 του Kubernetes, τα περισσότερα predicates και priority functions του Kube-Scheduler έχουν μετατραπεί σε plugins στα σημεία επέκτασης Filter και Score αντίστοιχα.

Ωστόσο, και η διαδικασία δημιουργίας ενός plugin είναι σχετικά απλή. Αρχικά, σε κάθε ορισμένο από τον χρήστη plugin χρειάζεται να υλοποιηθούν οι συναρτήσεις του προγραμματιστικού interface Plugin, καθώς και τα συγκεκριμένα interfaces των σημείων του Scheduling Framework που θα επεκταθούν. Στην Εικόνα 4.12[53] απεικονίζονται το Plugin interface και το interface του σημείου επέκτασης Sort. Στον Πίνακα 4.5 [86], [101], συνοψίζεται για κάθε σημείο επέκτασης η γενική λειτουργία των plugins του και οι συναρτήσεις του interface του που χρειάζεται να υλοποιηθούν. Στη συνέχεια, σε κάθε plugin πρέπει να οριστεί μια συνάρτηση κατασκευαστή (constructor function), η δήλωση της οποίας είναι της μορφής `New (plArgs *runtime.Unknown, handle framework.FrameworkHandle)(framework.Plugin, error)`. Το σύνολο αυτών των συναρτήσεων μπορεί να υλοποιηθεί σε ένα αρχείο της επιλεγμένης γλώσσας προγραμματισμού, για παράδειγμα σε ένα αρχείο Go.

[53] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L17

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

// LessFunc is the function to sort pod info
type LessFunc func(podInfo1, podInfo2 *QueuedPodInfo) bool

// QueueSortPlugin is an interface that must be implemented by "QueueSort" plugins.
// These plugins are used to sort pods in the scheduling queue. Only one queue sort
// plugin may be enabled at a time.
type QueueSortPlugin interface {
        Plugin
        // Less are used to sort pods in the scheduling queue.
        Less(*QueuedPodInfo, *QueuedPodInfo) bool
}
```

Εικόνα 4.12: Τα interfaces Plugin και QueueSortPlugin

Πίνακας 4.4: Τα υλοποιημένα plugins στο Kubernetes Scheduling Framework

Plugin	**Σημεία επέκτασης**	**Λειτουργία**	**Προκαθορισμένη χρήση**
ImageLocality	Score	Δίνεται προτεραιότητα στους κόμβους στους οποίους υπάρχουν ήδη οι container images του Pod	✓
TaintToleration	Filter, PreScore, Score	Υλοποιεί τη λογική των taints και tolerations	✓
NodeName	Filter	Ελέγχει αν το όνομα κόμβου που έχει καθοριστεί στο PodSpec αντιστοιχεί στον υπό εξέταση κόμβο	✓
NodePorts	PreFilter, Filter	Ελέγχει αν οι θύρες δικτύου που έχουν αιτηθεί από το Pod είναι διαθέσιμες στον υπό εξέταση κόμβο	✓
NodePreferAvoidPorts	Score	Βαθμολογεί τους υποψήφιους κόμβους για τη χρονοδρομολόγηση βάσει της σήμανσης `scheduler.alpha.kubernetes.io/preferAvoidPods`	✓
NodeAffinity	Filter, Score	Υλοποιεί τη λογική χρήσης του πεδίου nodeSelector και των υποπεδίων του affinity	✓
PodTopologySpread	PreFilter, Filter, PreScore, Score	Υλοποιεί τους περιορισμούς της τοπολογίας Pod Spread[54]	✓
NodeUnschedulable	Filter	Απορρίπτει τους κόμβους στους οποίους το πεδίο `unschedulable` του `spec` είναι `True`	✓
NodeResourcesFit	PreFilter, Filter, Score	Ελέγχει αν υπό εξέταση κόμβος έχει όλους τους πόρους που αιτείται το Pod	✓
NodeResourcesBalancedAllocation	Score	Δίνεται προτεραιότητα στους κόμβους στους οποίους αν δρομολογηθεί το Pod θα είναι πιο ομοιόμορφη η χρήση των πόρων	✓

[54] https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/

VolumeBinding	PreFilter, Filter, Reserve, PreBind, Score	Ελέγχει αν ο κόμβος έχει ή μπορούν να συνδεθούν σε αυτόν οι δομές αποθήκευσης που αιτείται το Pod	✓
VolumeRestrictions	Filter	Ελέγχει ότι οι δομές αποθήκευσης που είναι συνδεδεμένες στον κόμβο ικανοποιούν τους περιορισμούς του παρόχου που χρησιμοποιείται	✓
VolumeZone	Filter	Ελέγχει ότι οι δομές αποθήκευσης που έχουν αιτηθεί από το Pod ικανοποιούν πιθανούς περιορισμούς της ζώνης των χρησιμοποιούμενων πόρων	✓
NodeVolumeLimits	Filter	Ελέγχει ότι ικανοποιούνται οι περιορισμοί των CSI δομών αποθήκευσης στον κόμβο	✓
EBSLimits	Filter	Ελέγχει ότι ικανοποιούνται οι περιορισμοί των AWS EBS δομών αποθήκευσης στον κόμβο	✓
GCEPDLimits	Filter	Ελέγχει ότι ικανοποιούνται οι περιορισμοί των GCP-PD δομών αποθήκευσης στον κόμβο	✓
AzureDiskLimits	Filter	Ελέγχει ότι ικανοποιούνται οι περιορισμοί των Azure Disk δομών αποθήκευσης στον κόμβο	✓
InterPodAffinity	PreFilter, Filter, PreScore, Score	Υλοποιεί τη λογική των πεδίων `podAffinity` και `podAntiAffinity`	✓
PrioritySort	Sort	Παρέχει τις προκαθορισμένες λειτουργίες της ταξινόμησης βάσει προτεραιοτήτων	✓
DefaultBinder	Bind	Παρέχει τους προκαθορισμένους μηχανισμούς ανάθεσης	✓
DefaultPreemption	PostFilter	Παρέχει τους προκαθορισμένους μηχανισμούς εκτοπισμού του Pod από έναν κόμβο και αντικατάστασής του από άλλο	✓
SelectorSpread	PreScore, Score	Προτιμά την κατανομή των Pods του ίδιου Service, ReplicaSet και StatefulSet σε πολλαπλούς κόμβους	
CinderLimits	Filter	Ελέγχει ότι ικανοποιούνται οι περιορισμοί των OpenStack Cinder δομών αποθήκευσης στον κόμβο	

Πίνακας 4.5: Τα σημεία επέκτασης και τα αντίστοιχα plugins στο Kubernetes Scheduling Framework

Σημείο επέκτασης	**Λειτουργία plugin**	**Συναρτήσεις του interface προς υλοποίηση**
Sort[55]	Ταξινομεί τα Pods στην ουρά χρονοδρομολόγησης βάσει της συνάρτησης Less. Μόνο ένα plugin μπορεί να επεκτείνει αυτό το σημείο.	**Less** (*PodInfo, *PodInfo) bool: για τον καθορισμό της επιθυμητής προτεραιότητας ανάμεσα σε δύο Pods, Επιστέφει τιμή τύπου bool.
PreFilter[56]	Καλείται στην αρχή της φάσης scheduling για την	**PreFilter** (ctx context. Context, state *CycleState,

[55] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L278

	προ επεξεργασία πληροφοριών σχετικών με το Pod ή για τον έλεγχο περιορισμών που πρέπει να ικανοποιεί το Pod ή το cluster συνολικά. Για να δρομολογηθεί το Pod πρέπει όλα τα plugins σε αυτό το σημείο επέκτασης να εκτελεστούν επιτυχώς.	p *v1 Pod) *Status: υλοποιεί τη λογική προ επεξεργασίας.
Filter[57]	Απορρίπτει τους κόμβους στους οποίους δεν μπορεί να δρομολογηθεί το Pod. Εξ ορισμού, σε αυτό το σημείο ο kube-scheduler ελέγχει ένα υποσύνολο τον predicates σε προκαθορισμένη σειρά, όπως περιγράφηκε στην Ενότητα 4.1.2. Αν μετά την εφαρμογή κάποιου plugin δεν υπάρχουν υποψήφιοι κόμβοι για την εκτέλεση του Pod, δεν εφαρμόζονται τα υπόλοιπα plugins. Περισσότεροι του ενός κόμβοι μπορούν να ελέγχονται ταυτόχρονα Filter plugins σε υπορουτίνες. Τα Filter plugins μπορούν να κληθούν περισσότερες από μία φορές στη scheduling φάση του πλαισίου χρονοδρομολόγησης.	**Filter** (ctx context. Context, state *CycleState, pod *v1.Pod, nodeInfo *schedulernodeinfo.NodeInfo) *Status: υλοποιεί το φιλτράρισμα των υποψήφιων για τη χρονοδρομολόγηση κόμβων. Αν ο υπό εξέταση κόμβος είναι κατάλληλος για τη χρονοδρομολόγηση του Pod επιστρέφει "Success", αλλιώς "Unschedulable", "Unschedulable And Unresolvable" ή "Error".
PostFilter[58]	Καλείται όταν δεν υπάρχουν υποψήφιοι κόμβοι για τη χρονοδρομολόγηση του Pod μετά την εφαρμογή των Filter plugins. Τα καθορισμένα σε αυτό το σημείο plugins καλούνται με τη σειρά που έχουν οριστεί. Αν κάποιο από αυτά τα plugins επιλέξει έναν κόμβο για τη χρονοδρομολόγηση του Pod, δε θα κληθούν τα υπόλοιπα PostFilter plugins. Μία τυπική χρήση των PostFilter plugins είναι όταν σταματά η εκτέλεση Pods μικρότερης προτεραιότητας σε έναν κόμβο για να δρομολογηθεί Pod υψηλότερης προτεραιότητας (π.χ. πιο κρίσιμης εφαρμογής).	**PostFilter** (ctx context. Context, state *CycleState, pod *v1.Pod, filteredNodeStatusMap NodeToStatusMap) (*PostFilterResult, *Status): εκτελείται αν δεν έχει βρεθεί κόμβος για τη χρονοδρομολόγηση του Pod. Επιστρέφει "Success", "Unschedulable" ή "Error".
PreScore[59]	Καλείται αν βρεθεί τουλάχιστον ένας υποψήφιος κόμβος για την ανάθεση του Pod μετά τα Filter plugins. Το σημείο χρησιμοποιείται συνήθως για την απόκτηση πληροφοριών ώστε να ενημερωθεί η εσωτερική κατάσταση του cluster και τα αρχεία καταγραφής (logs).	**PreScore** (ctx context. Context, state *CycleState, pod *v1.Pod, nodes []*v1.Node) *Status: υλοποιεί απαραίτητες λειτουργίες ελέγχου του Pod πριν ξεκινήσει το στάδιο Score. Αν δεν επιστρέψει "Success" το πλαίσιο χρονοδρομολόγησης του Pod τερματίζεται ανεπιτυχώς.
Score[60]	Βαθμολογεί του υποψήφιους κόμβους που προέκυψαν από τα Filter plugins. Τα plugins αυτού του σημείου υλοποιούν τις περισσότερες συναρτήσεις προτεραιότητας, όπως περιεγράφηκαν στην Ενότητα 4.1.2. Ο Scheduler θα καλέσει τη συνάρτηση Score κάθε plugin για κάθε υποψήφιο κόμβο και όλα τα plugins πρέπει να εφαρμοστούν επιτυχώς για να μην αποτύχει το πλαίσιο χρονοδρομολόγησης του Pod.	**Score** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string) (int64, *Status): υλοποιεί τη βαθμολόγηση των κόμβων. Αν δεν υπάρξει σφάλμα επιστρέφει τη βαθμολογία του κόμβου και την τιμή "Success".
NormalizeScore[61]	Επεξεργάζεται περαιτέρω τις βαθμολογίες που έχουν ανατεθεί στους κόμβους πριν ο Scheduler υπολογίσει την τελική βαθμολογία κάθε κόμβου. Ο Scheduler θα συνδυάσει κατάλληλα τη βαθμολογία που έχει ανατεθεί σε κάθε κόμβο από κάθε plugin βάσει του καθορισμένου βάρους του plugin.	**NormalizeScore** (ctx context. Context, state *CycleState, p *v1.Pod, scores NodeScoreList) *Status: καλείται μία φορά ανά plugin για όλες τις βαθμολογίες των κόμβων που έχουν προκύψει από την ίδια συνάρτηση Score του plugin. Κανονικοποιεί τις βαθμολογίες των κόμβων κάθε plugin στο εύρος ακέραιων τιμών 0-100 και επιστρέφει "Success" αν δεν υπάρξει σφάλμα.

[56] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L308
[57] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L329
[58] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L347
[59] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L366
[60] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L384
[61] https://github.com/kubernetes/kubernetes/blob/2112bddae101b35934f1d66288449adf31243033/pkg/scheduler/framework/interface.go#L375

Reserve[62]	Σημείο επέκτασης για την ανάληψη πληροφοριών. Τα plugins που υλοποιούνται διατηρούν πληροφορίες για την κατάσταση εκτέλεσης όταν δεσμεύουν πόρους σε έναν κόμβο για το εκάστοτε Pod. Αποτελείται από δύο στάδια: το Reserve και το Unreserve. Το στάδιο Reserve πραγματοποιείται πριν ο Scheduler αναθέσει το Pod στον επιλεγμένο κόμβο και υπάρχει προκειμένου να αποφευχθούν συνθήκες ανταγωνισμού (race conditions) όσο ο Scheduler περιμένει να γίνει δεκτό το αίτημα ανάθεσης. Το στάδιο Reserve θεωρείται επιτυχές αν εκτελεστούν με επιτυχία όλα τα ορισμένα σε αυτό plugins και μόνο τότε εκτελούνται οι υπόλοιπες λειτουργίες του πλαισίου χρονοδρομολόγησης. Το στάδιο Unreserve εκτελείται αν αποτύχει το Reserve στάδιο ή κάποια από τις επόμενες λειτουργίες του πλαισίου χρονοδρομολόγησης. Το στάδιο αυτό υπάρχει για την επαναφορά του cluster στην προηγούμενη κατάστασή του πριν τη δρομολόγηση του Pod.	**Reserve** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string) *Status: καλείται κάθε φορά που η cache του Scheduler ενημερώνεται. Αν η συνάρτηση αποτύψει, ο Scheduler θα καλέσει τη συνάρτηση Unreserve για όλα τα plugins του σταδίου Reserve και το πλαίσιο χρονοδρομολόγησης του Pod θα τερματιστεί ανεπιτυχώς. **Unreserve** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string): καλείται όταν ο κόμβος απέρριψε το αίτημα ανάθεσης του Pod σε αυτόν για εκτέλεση ή η διαδικασία χρονοδρομολόγησης του Pod τερματίστηκε λόγω σφάλματος.
Permit[63]	Καλείται στο τέλος της φάσης χρονοδρομολόγησης ώστε κάθε Pod να αποδεχθεί ή να απορρίψει τον κόμβο που επιλέχθηκε ή να καθυστερήσει την εκτέλεσή του.	**Permit** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string) (*Status, time.Duration): επιστρέφει “Success” ή “Wait” προσδιορίζοντας ένα χρονικό όριο αναμονής αν δεν υπάρξει σφάλμα. Αν επιστραφεί τιμή σφάλματος το πλαίσιο χρονοδρομολόγησης του Pod τερματίζεται ανεπιτυχώς.
PreBind[64]	Εκτελεί τις εργασίες που ενδεχομένως απαιτούνται πριν ανατεθεί το Pod στον κόμβο (π.χ. σύνδεση των απαραίτητων πόρων δικτύου στον επιλεγμένο κόμβο). Όλα τα plugins αυτού του σημείου πρέπει να εκτελεστούν επιτυχώς για να μην αποτύχει το πλαίσιο χρονοδρομολόγησης του Pod.	**PreBind** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string) *Status: υλοποιεί τις απαραίτητες λειτουργίες πριν την ανάθεση του Pod
Bind[65]	Αναθέτει το Pod για εκτέλεση στον επιλεγμένο κόμβο. Τα plugins αυτού του σημείου καλούνται με τη σειρά που έχουν οριστεί και κάθε plugin μπορεί να επιλέξει αν θα εφαρμοστεί ή όχι στο εκάστοτε Pod. Αν ένα plugin επιλέξει να εφαρμοστεί στο Pod, δεν εφαρμόζονται τα επόμενά του σε αυτό το σημείο επέκτασης.	**Bind** (ctx context.Context, state *CycleState, p *v1.Pod, nodeName string) *Status: επιστρέφει “Skip” όταν το plugin δεν μπορεί να εφαρμοστεί στο συγκεκριμένο Pod. Αν επιστρέψει τιμή σφάλματος, η διαδικασία χρονοδρομολόγησης του Pod σταματάει.
PostBind[66]	Σημείο ανάληψης πληροφοριών, τα plugins του οποίου καλούνται αφού ένα Pod ανατεθεί επιτυχώς για εκτέλεση στον κόμβο που επιλέχθηκε. Συνήθως τα plugins σε αυτό το σημείο χρησιμοποιούνται για την αποδέσμευση των πόρων που δεσμεύτηκαν για το συγκεκριμένο πλαίσιο χρονοδρομολόγησης.	**PostBind** (ctx context. Context, state *CycleState, p *v1.Pod, nodeName string): υλοποιεί τις απαραίτητες λειτουργίες μετά την ανάθεση του Pod

Συνοπτικά, οι προγραμματιστικές δομές δεδομένων και οι συναρτήσεις που εντοπίζονται κυρίως στα interfaces των plugins στον κώδικα του Kubernetes σε Go είναι οι παρακάτω. Οι πληροφορίες για το Pod προς χρονοδρομολόγηση περιγράφονται σε αντικείμενα της struct `PodInfo`[67]. Επιπλέον

[62] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L401
[63] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L437
[64] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L417
[65] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L450
[66] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L426
[67] https://github.com/kubernetes/kubernetes/blob/2112bddae101b35934f1d66288449adf31243033/pkg/scheduler/framework/types.go#L122

πληροφορίες που σχετίζονται με την κατάσταση του Pod στην ουρά χρονοδρομολόγησης συγκεντρώνονται σε αντικείμενα της struct `QueuedPodInfo`[68], η οποία λειτουργεί ως wrapper της `PodInfo`. Για κάθε κόμβο, δημιουργείται ένα αντικείμενο της struct `NodeInfo`[69], το οποίο περιέχει πληροφορίες σχετικές για παράδειγμα με τα εκτελούμενα σε αυτόν Pods και τους πόρους που χρησιμοποιεί. Στη struct `CycleState`[70] προσδιορίζονται τα APIs για την πρόσβαση στα δεδομένα του εκάστοτε πλαισίου χρονοδρομολόγησης και ορίζεται ο μηχανισμός ανταλλαγής δεδομένων μεταξύ των υλοποιημένων plugins στα διάφορα σημεία επέκτασης. Επίσης, οι συναρτήσεις του interface `Framework`[71] είναι υπεύθυνες για την εκτέλεση και τον συντονισμό των plugins στα σημεία επέκτασης. Ακόμη, οι συναρτήσεις του interface `Handle`[72] επιτρέπουν την πρόσβαση του plugin στην εσωτερική cache του Scheduler και στον API Server για την ανάκτηση πληροφοριών σχετικών με την κατάσταση του cluster.

Ακολούθως, για τη δημιουργία ενός plugin, αφού δημιουργηθούν οι μέθοδοι των σχετικών interfaces και ο constructor του, χρειάζεται να υλοποιηθεί μία συνάρτηση `main()` η οποία λειτουργεί ως wrapper του Kube-Scheduler. Σε αυτή δηλώνονται τα plugins που έχουν οριστεί, προκειμένου ο κώδικάς τους να μεταγλωττιστεί μαζί με αυτόν του Kube-Scheduler. Στην Εικόνα 4.13 απεικονίζεται ένα παράδειγμα υλοποίησης σε Go. Η μέθοδος `NewSchedulerCommand`[73] του πακέτου λογισμικού `kube-scheduler` ορίζει ένα αντικείμενο `Command`[74] της βιβλιοθήκης `Cobra`[75], το οποίο είναι εκτελέσιμο. Ο πιο απλός τρόπος για τη δήλωση των plugins είναι να προσδιοριστούν σε μία προγραμματιστική δομή τύπου `Registry`. Συγκεκριμένα, για τη δήλωση κάθε plugin καλείται η `WithPlugin`[76], όπως φαίνεται στην Εικόνα 4.13, η οποία καλεί εσωτερικά τη συνάρτηση `Register`[77] (Εικόνα 4.14). Η συνάρτηση `Register` δημιουργεί ένα αντικείμενο τύπου `Registry`[78] με το όνομα και τον constructor του plugin, τα οποία αποτελούν ένα αντικείμενο τύπου `PluginFactory`. Τέλος, η συνάρτηση `Setup`[79] αρχικοποιεί τα plugins που ορίστηκαν, δηλώνοντάς τα στο αντικείμενο τύπου `Registry`[80] του Scheduler [86]. Συνολικά, ένα plugin σε Go υλοποιείται στο GitHub repository scheduler-framework-sample[81].

---

[68] https://github.com/kubernetes/kubernetes/blob/2112bddae101b35934f1d66288449adf31243033/pkg/scheduler/framework/types.go#L93
[69] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/types.go#L358
[70] https://github.com/kubernetes/kubernetes/blob/2112bddae101b35934f1d66288449adf31243033/pkg/scheduler/framework/cycle_state.go#L44
[71] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L463
[72] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/interface.go#L566
[73] https://github.com/kubernetes/kubernetes/blob/master/cmd/kube-scheduler/app/server.go#L65
[74] https://github.com/spf13/cobra#commands
[75] https://github.com/spf13/cobra
[76] https://github.com/kubernetes/kubernetes/blob/master/cmd/kube-scheduler/app/server.go#L299
[77] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/runtime/registry.go#L62
[78] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/runtime/registry.go#L58
[79] https://github.com/kubernetes/kubernetes/blob/8634bc61c635717dec93128f8908ffd20774e66f/cmd/kube-scheduler/app/server.go#L319
[80] https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/framework/runtime/registry.go#L58
[81] https://github.com/angao/scheduler-framework-sample/blob/master/pkg/plugins/sample/sample.go

Τέλος, χρειάζεται να δημιουργηθεί ένα προφίλ χρονοδρομολόγησης (scheduling profile) για να ρυθμιστεί η λειτουργία των plugins στον Kube-Scheduler. Οι απαραίτητες ρυθμίσεις μπορούν να προσδιοριστούν ορίζοντας ένα Kubernetes object τύπου KubeSchedulerConfiguration σε ένα αρχείο YAML[82] και ακολούθως να εφαρμοστούν στον Kube-Scheduler με την εντολή `kube-scheduler --config <ONOMA_APXEIOY_YAML>`. Σε αυτό το αρχείο, καθορίζονται για κάθε σημείο επέκτασης τα plugins που έχουν υλοποιηθεί, ποια από τα προκαθορισμένα plugins του Kube-Scheduler και από αυτά που ορίστηκαν από τους χρήστες θα χρησιμοποιηθούν στα επερχόμενα πλαίσια χρονοδρομολόγησης, καθώς και η σειρά με την οποία αυτά τα plugins θα εφαρμοστούν στους υποψήφιους κόμβους. Προαιρετικά, μπορούν να προσδιοριστούν ορίσματα για τις συναρτήσεις κάθε plugin. Ένα απλό παράδειγμα αυτού του αρχείου απεικονίζεται στην Εικόνα 4.15[83], όπου στο σημείο επέκτασης Score είναι απενεργοποιημένο το προκαθορισμένο plugin `NodeResourcesLeastAllocated` και αντί αυτού θα χρησιμοποιηθούν τα plugins `MyCustomPluginA` και `MyCustomPluginsB`. Μάλιστα, από την έκδοση v.1.19, το Kubernetes παρέχει τη δυνατότητα δημιουργίας περισσότερων του ενός προφίλ χρονοδρομολόγησης [100], [102]. Για παράδειγμα, στην Εικόνα 4.16[84] έχουν οριστεί δύο προφίλ, τα `default-scheduler` και `no-scoring-scheduler`.

```go
import (
    scheduler "k8s.io/kubernetes/cmd/kube-scheduler/app"
)

func main() {
    command := scheduler.NewSchedulerCommand(
            scheduler.WithPlugin("example-plugin1", ExamplePlugin1),
            scheduler.WithPlugin("example-plugin2", ExamplePlugin2))
    if err := command.Execute(); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

Εικόνα 4.13: Παράδειγμα υλοποίησης της συνάρτησης main σε Go για τη δήλωση των ορισμένων από τον χρήστη plugins

```go
// WithPlugin creates an Option based on plugin name and factory. Please don't remove this function: it is used to register out-of-tree plugins,
// hence there are no references to it from the kubernetes scheduler code base.
func WithPlugin(name string, factory runtime.PluginFactory) Option {
	return func(registry runtime.Registry) error {
		return registry.Register(name, factory)
	}
}
```

Εικόνα 4.14: Η υλοποίηση της WithPlugin σε Go

Συνολικά, το Scheduling Framework του Kubernetes διευκολύνει την ανάπτυξη λογικής χρονοδρομολόγησης με άξονα τις ανάγκες των εφαρμογών, αξιοποιώντας ταυτόχρονα στο έπακρο τις δυνατότητες του προκαθορισμένου μηχανισμού χρονοδρομολόγησης της πλατφόρμας. Πρώτον, τα

[82] https://github.com/angao/scheduler-framework-sample/blob/master/deploy/deployment.yaml
[83] https://kubernetes.io/docs/reference/scheduling/config/#extension-points
[84] https://github.com/kubernetes/website/blob/main/content/en/docs/reference/scheduling/config.md

σημεία παρέμβασης στον Kube-Scheduler είναι πολλαπλά, τόσο στη φάση scheduling όσο και στην binding. Δεύτερον, η ανάπτυξη λειτουργιών χρονοδρομολόγησης ως plugins, αποσυνδέει τη λογική χρονοδρομολόγησης που έχει υλοποιηθεί από τον χρήστη για τις ανάγκες μιας συγκεκριμένης εφαρμογής από τις ήδη υλοποιημένες λειτουργίες. Με αυτόν τον τρόπο, ο βασικός αλγόριθμος χρονοδρομολόγησης παραμένει απλός και εύκολα διατηρήσιμος, παρέχοντας ταυτόχρονα τη δυνατότητα στους χρήστες να τον χρησιμοποιούν μαζί με τον κώδικα χρονοδρομολόγησης που έχει αναπτυχθεί για την κάλυψη συγκεκριμένων αναγκών των εφαρμογών. Ο κώδικας του Kube-Scheduler δε χρειάζεται να τροποποιηθεί ή να υλοποιηθεί εκ νέου και χρήστες δεν είναι αναγκαίο να προσαρμόσουν τον κώδικά τους σε ενδεχόμενες αλλαγές του προκαθορισμένου μηχανισμού. Τρίτον, η διαδικασία ανάθεσης των Pods σε κόμβους είναι ταχύτερη με αυτόν τον τρόπο επέκτασης του μηχανισμού χρονοδρομολόγησης, δεδομένου ο κώδικας των υλοποιημένων plugins μεταγλωττίζεται μαζί με τον κώδικα του Kube-Scheduler [98]. Τέλος, συγκριτικά με τις άλλες μεθόδους επέκτασης, είναι πιο εύκολη η συλλογή πληροφοριών στο επίπεδο του plugin για την κατάσταση του cluster και των επιμέρους μηχανισμών, καθώς και η ενημέρωση των συναρτήσεων του plugin για πιθανά σφάλματα που ενδεχομένως οδήγησαν σε διακοπή του πλαισίου χρονοδρομολόγησης του εκάστοτε Pod.

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:
        disabled:
        - name: NodeResourcesLeastAllocated
        enabled:
        - name: MyCustomPluginA
          weight: 2
        - name: MyCustomPluginB
          weight: 1
```

Εικόνα 4.15: Παράδειγμα ορισμού προφίλ χρονοδρομολόγησης

```yaml
apiVersion: kubescheduler.config.k8s.io/v1beta2
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: default-scheduler
  - schedulerName: no-scoring-scheduler
    plugins:
      preScore:
        disabled:
        - name: '*'
      score:
        disabled:
        - name: '*'
```

Εικόνα 4.16: Παράδειγμα ορισμού περισσότερων του ενός προφίλ χρονοδρομολόγησης

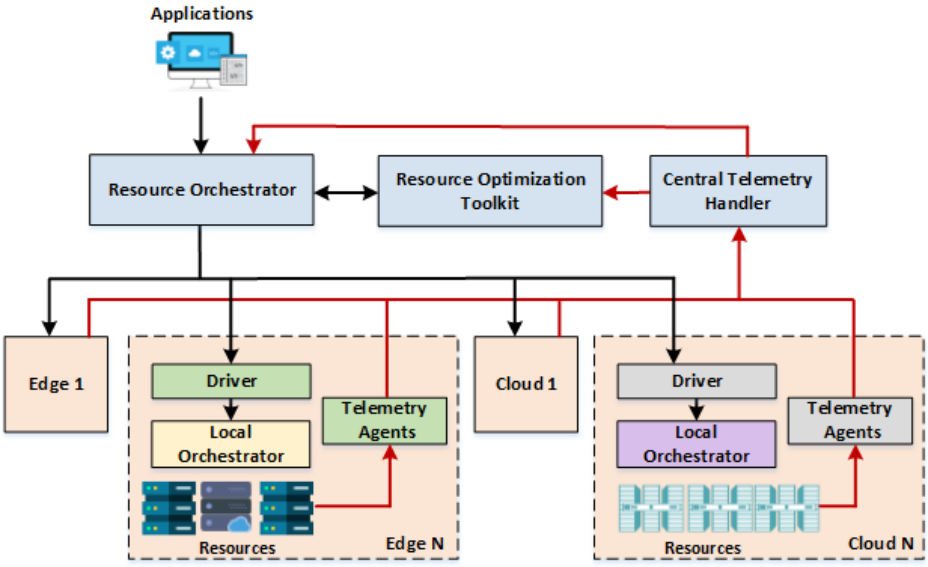
Τέλος, πρέπει να αναφερθεί ότι σε κάποιες περιπτώσεις μπορεί οι διαχειριστές του cluster να χρειαστεί να υλοποιήσουν από την αρχή τον μηχανισμό χρονοδρομολόγησης που θα χρησιμοποιηθεί για την ανάθεση των Pods πολύ συγκεκριμένων εφαρμογών σε εξειδικευμένους πόρους και

τοπολογίες κόμβων. Το εγχείρημα αυτό είναι εξαιρετικά σύνθετο, λαμβάνοντας υπόψη την αυξημένη πολυπλοκότητα των μηχανισμών του Kubernetes. Για αυτόν τον λόγο, ενδείκνυται η εκτενής μελέτη των δυνατοτήτων επέκτασης του προκαθορισμένου αλγορίθμου χρονοδρομολόγησης, κυρίως μέσω plugins, πριν επιχειρηθεί από τις ομάδες ανάπτυξης [83].

Κεφάλαιο 5: Σχεδιασμός και Υλοποίηση Συστήματος Ενορχήστρωσης Εφαρμογών

5.1 Γενική Αρχιτεκτονική Συστήματος Ενορχήστρωσης

Όπως αναφέρθηκε στην Ενότητα 1.2, η διπλωματική εργασία εστιάζει στη μελέτη και στον σχεδιασμό των κατάλληλων μηχανισμών για τη διαφανή ενορχήστρωση εγγενών εφαρμογών υπολογιστικού νέφους σε ένα σύστημα που περιλαμβάνει πολλές ετερογενείς edge και cloud υποδομές. Η Εικόνα 5.1 παρουσιάζει την ιεραρχική και κατανεμημένη οργάνωση των μηχανισμών ενορχήστρωσης που θεωρήθηκε για τη διαχείριση των διαθέσιμων υπολογιστικών και αποθηκευτικών πόρων, βάσει του μοντέλου της ενοποιημένης λειτουργίας υποδομών (federated infrastructures).



Εικόνα 5.1: Ιεραρχική ενορχήστρωση εγγενών εφαρμογών υπολογιστικού νέφους σε ενοποιημένες edge και cloud υποδομές

Σύμφωνα με αυτόν τον σχεδιασμό, η ενορχήστρωση των εφαρμογών στο σύστημα πραγματοποιείται σε δύο επίπεδα, στο χαμηλότερο επίπεδο των Local Orchestrators και στο υψηλότερο του Resource Orchestrator. Πιο συγκεκριμένα, από τη μία πλευρά, κάθε Local Orchestrator είναι υπεύθυνος για την ενορχήστρωση των εφαρμογών στους πόρους των clusters που διαχειρίζεται. Ως Local Orchestrator για υποδομές cloud μπορεί να επιλεχθεί κάποιο από τα εργαλεία ανοιχτού κώδικα που εξετάστηκαν στην Ενότητα 2.3. Όπως φαίνεται στην Εικόνα 5.1, στο σύστημα μπορούν να ενσωματώνονται περισσότεροι του ενός Local Orchestrators, δηλαδή πολλαπλά και διαφορετικά μεταξύ τους εργαλεία ενορχήστρωσης, τόσο για υποδομές cloud όσο και για edge, με καθένα να διαχειρίζεται διαφορετικά και ενδεχομένως ετερογενή ως προς τους πόρους τους clusters.

Από την άλλη πλευρά, στο υψηλότερο επίπεδο ενορχήστρωσης, ο Resource Orchestrator διαχειρίζεται τους διαθέσιμους ετερογενείς πόρους με έναν πιο αφαιρετικό τρόπο συγκριτικά με τους Local Orchestrators. Ο μηχανισμός αυτός λαμβάνει τα αιτήματα των χρηστών του συστήματος για την εκτέλεση, τον τερματισμό ή άλλες λειτουργίες που σχετίζονται με τη διαχείριση μιας εφαρμογής σε όλο τον κύκλο ζωής της και καθοδηγεί κατάλληλα τους Local Orchestrators για την εξυπηρέτηση αυτών των αιτημάτων. Η διασύνδεση του Resource Orchestrator με κάθε Local Orchestrator επιτυγχάνεται υλοποιώντας έναν μηχανισμό Driver.

Ιδιαίτερα στην περίπτωση αιτήματος εκτέλεσης μιας εφαρμογής, το αίτημα των χρηστών χρειάζεται να περιλαμβάνει μια γενική περιγραφή της, η οποία είναι ανεξάρτητη των εργαλείων ενορχήστρωσης που έχουν χρησιμοποιηθεί ως Local Orchestrators και των υποδομών του συστήματος. Στη συνέχεια, ανάλογα με τους περιορισμούς και τις προτιμήσεις των χρηστών, όπως έχουν προσδιοριστεί με γενικό τρόπο στην περιγραφή, καθώς και την εκάστοτε κατάσταση του συστήματος, ο Resource Orchestrator αναθέτει την εκτέλεση της εφαρμογής στους κατάλληλους, ενδεχομένως ετερογενείς, Local Orchestrators και τους καθοδηγεί για την ενορχήστρωσή της τοπικά στα αντίστοιχα clusters. Οι οδηγίες αυτές προκύπτουν από τις πληροφορίες που λαμβάνει ο Resource Orchestrator από τους μηχανισμούς Resource Optimization Toolkit και Central Telemetry Handler και στοχεύουν στην ικανοποίηση των αιτημάτων των χρηστών, επιτυγχάνοντας ταυτόχρονα βέλτιστη κατανομή των διαθέσιμων πόρων. Έκτοτε, την εκτέλεση της εφαρμογής στους υποκείμενους πόρους αναλαμβάνουν οι επιλεγμένοι Local Orchestrators.

Αξίζει να σημειωθεί ότι η βέλτιστη κατανομή των πόρων στην υποβληθείσες εργασίες σε ένα τόσο ετερογενές και δυναμικά μεταβαλλόμενο περιβάλλον συνιστά μια σύνθετη διαδικασία λήψης αποφάσεων σε πραγματικό χρόνο. Λόγω των πολλαπλών, συχνά αντικρουόμενων κριτηρίων που χρειάζεται να ληφθούν υπόψη, η απόφαση αυτή δεν μπορεί να ληφθεί με βέλτιστο τρόπο στο επίπεδο των Local Orchestrators, δηλαδή με άξονα μόνο την κατάσταση των μεμονωμένων υποδομών. Για αυτόν τον λόγο, όπως προαναφέρθηκε, εισάγεται στο σύστημα ο μηχανισμός του Resource Optimization Toolkit. Σε αυτόν υλοποιούνται, ανάλογα με τις παραμέτρους προς βελτιστοποίηση (π.χ. βέλτιστη κατανομή της διαθέσιμης υπολογιστικής ισχύος ή της μνήμης, ελαχιστοποίηση της καταναλωμένης ενέργειας), αλγόριθμοι για την επίλυση του συνολικού προβλήματος βέλτιστης κατανομής των διαθέσιμων πόρων στις υποβληθείσες εφαρμογές βάσει των απαιτήσεων των χρηστών και της κατάστασης του συστήματος. Οι αλγόριθμοι μπορεί να είναι προσεγγιστικοί πολλαπλών κριτηρίων και να συνδυάζουν τεχνικές δυναμικού προγραμματισμού και μηχανικής μάθησης με ευριστικές λύσεις.

Τέλος, στην αρχιτεκτονική περιλαμβάνονται μηχανισμοί εποπτείας και συλλογής πληροφοριών για την τρέχουσα κατάσταση των πόρων του συστήματος, καθώς και των εφαρμογών που έχουν υποβληθεί. Παρομοίως με τους μηχανισμούς ενορχήστρωσης, οι μηχανισμοί αυτοί οργανώνονται σε δύο επίπεδα, με τους Telemetry Agents να συλλέγουν πληροφορίες στο κατώτερο επίπεδο των

υποδομών και τον Central Telemetry Handler να συγκεντρώνει και να επεξεργάζεται αυτές τις πληροφορίες και να τις προωθεί στους μηχανισμούς Resource Optimization Toolkit και Resource Orchestrator στο υψηλότερο επίπεδο.

Σε αυτή την γενική αρχιτεκτονική, η διπλωματική εργασία έχει εστιάσει στους μηχανισμούς ενορχήστρωσης τόσο στο υψηλότερο επίπεδο (Resource Orchestrator) όσο και στις επιμέρους υποδομές (Drivers, Local Orchestrators). Στη συνέχεια παρουσιάζεται το σύστημα που σχεδιάστηκε και αναπτύχθηκε για την περίπτωση χρήσης ετερογενών εργαλείων διαχείρισης πόρων σε cloud υποδομές προκειμένου να αξιολογηθεί η δυνατότητα υλοποίησης της προτεινόμενης γενικής αρχιτεκτονικής στην πράξη (Proof of Concept, PoC). Ο πηγαίος κώδικας της υλοποίησης σε Python 3 είναι διαθέσιμος στο GitHub repository[85].

5.2 Περιγραφή Συστήματος και Χρησιμοποιούμενες Τεχνολογίες

Η αρθρωτή αρχιτεκτονική του συστήματος του Resource Orchestrator που σχεδιάστηκε, καθώς και οι κυριότεροι μηχανισμοί του και η διασύνδεση με το Resource Optimization Toolkit και τους Drivers των Local Orchestrators, συνοψίζονται στη Εικόνα 5.2. Στην Εικόνα 5.3 παρουσιάζεται λεπτομερέστερα η PoC υλοποίηση αυτού του συστήματος.

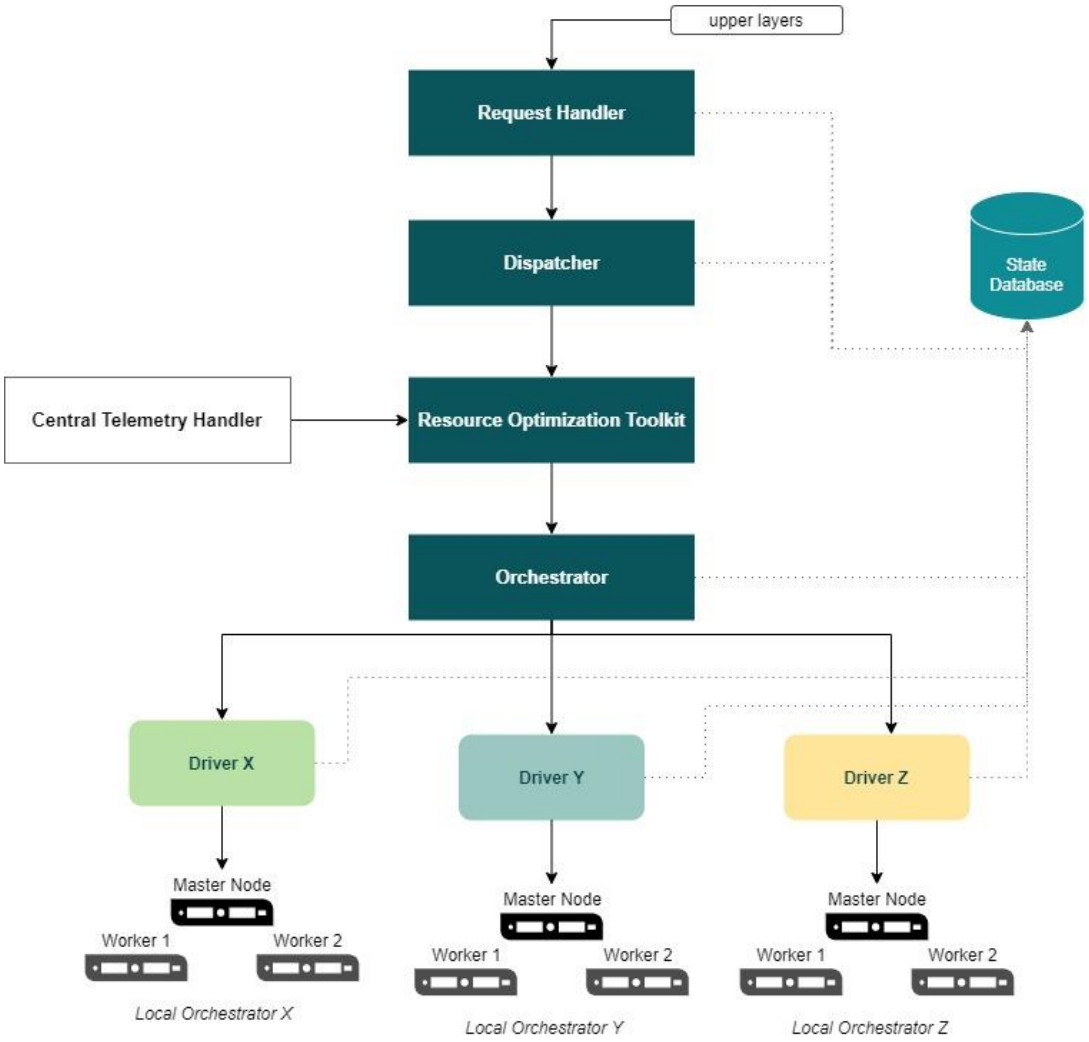
Συγκεκριμένα, για τη λήψη των αιτημάτων των χρηστών από τα ανώτερα επίπεδα της πλατφόρμας σχεδιάστηκε ο μηχανισμός του Request Handler. Πρόκειται για ένα REST API το οποίο δυνητικά υποστηρίζει τις λειτουργίες που συνοψίζονται στον Πίνακας 5.1. Στην περίπτωση αιτήματος για την εκτέλεση μιας εφαρμογής, ο Request Handler πραγματοποιεί επιπλέον κάποιους αρχικούς ελέγχους για σφάλματα σύνταξης στο αρχείο YAML της γενικής και ανεξάρτητης των ενσωματωμένων πόρων υλικού και λογισμικού περιγραφής της εφαρμογής. Για την υλοποίηση του Request Handler στο PoC σύστημα του Resource Orchestrator χρησιμοποιήθηκε η βιβλιοθήκη Flask[86] της Python.

Στη συνέχεια, ο μηχανισμός του Dispatcher είναι υπεύθυνος για τον διαμοιρασμό των αιτημάτων που έχουν υποβληθεί στον Request Handler στους υπόλοιπους μηχανισμούς του συστήματος για να εξυπηρετηθούν. Η απόφαση για τον μηχανισμό στον οποίο θα ανατεθεί το αίτημα για περαιτέρω επεξεργασία εξαρτάται από την πληροφορία του πεδίου `action` που λαμβάνει ο Dispatcher από τον Request Handler. Ο Πίνακας 5.2 συνοψίζει ο σύνολο τιμών αυτού του πεδίου.

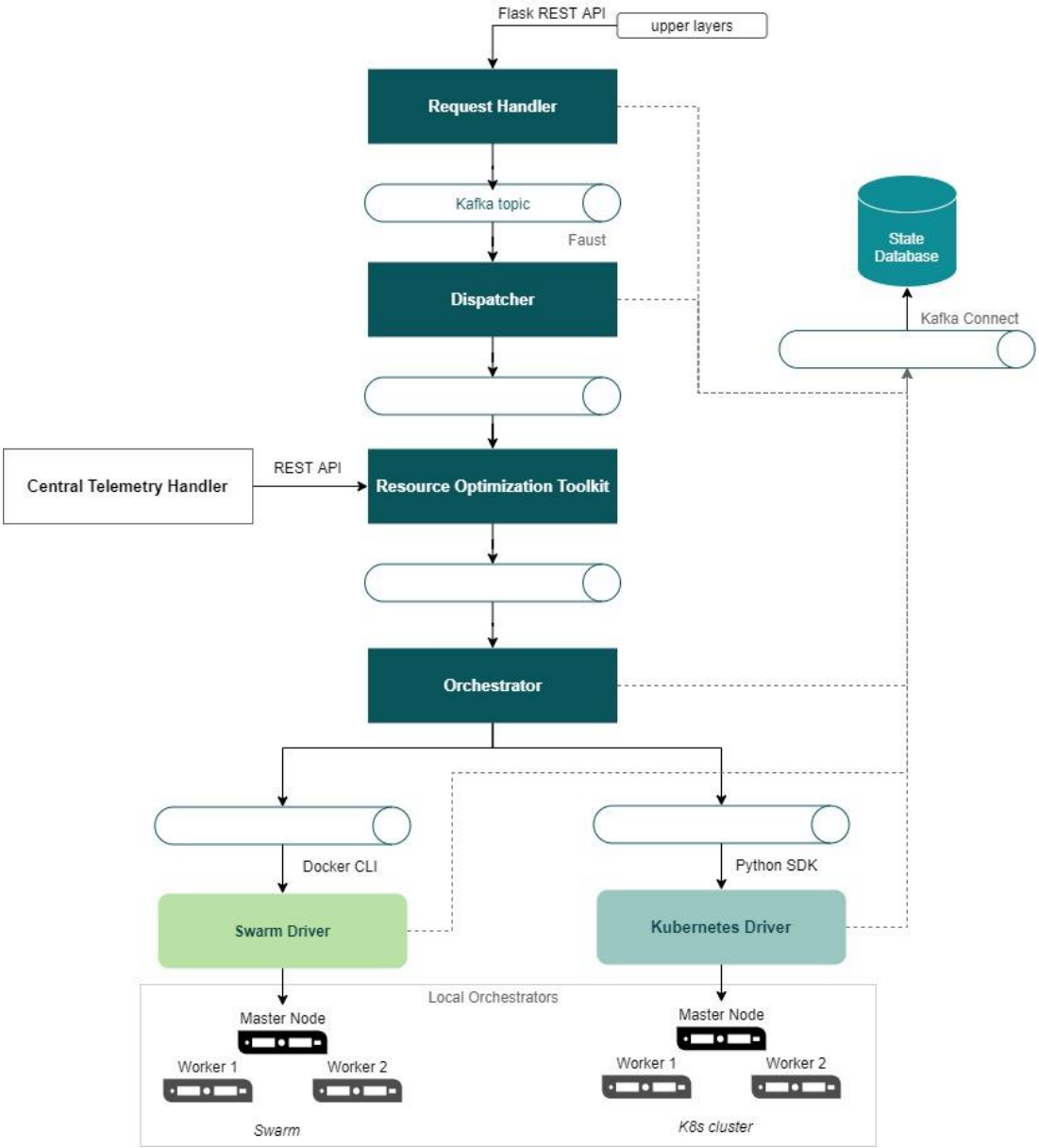
Ακολούθως, στην περίπτωση που η τιμή του πεδίου `action` είναι `deploy` ή `inspect` καλείται ο μηχανισμός του Resource Optimization Toolkit για την εξυπηρέτηση του αιτήματος. Ο μηχανισμός αυτός εκτελεί τον αλγόριθμο χρονοδρομολόγησης που έχει καθοριστεί ώστε, ανάλογα το αίτημα των χρηστών, την τρέχουσα κατάσταση των clusters που διαχειρίζονται οι Local Orchestrators και τα κριτήρια που έχουν επιλεχθεί προς βελτιστοποίηση (π.χ. ελαχιστοποίηση του χρόνου απόκρισης), να

[85] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator
[86] https://flask.palletsprojects.com/en/2.0.x/

επιλεχθεί ο Local Orchestrator στον οποίο θα εκτελεστεί η εφαρμογή. Επίσης, από τον αλγόριθμο χρονοδρομολόγησης προκύπτουν οδηγίες προς τον Driver του επιλεγμένου Local Orchestrator σχετικά με τους τοπικούς κόμβους στους οποίους θα εκτελεστεί η εφαρμογή. Αξίζει να σημειωθεί ότι το Resource Optimization Toolkit λαμβάνει πληροφορίες για την τρέχουσα κατάσταση των υποκείμενων πόρων υλικού και λογισμικού πραγματοποιώντας REST κλήση στο API του Central Telemetry Handler.



Εικόνα 5.2: Η αρθρωτή αρχιτεκτονική του Resource Orchestrator



Εικόνα 5.3: Το PoC σύστημα του Resource Orchestrator

Πίνακας 5.1: Πιθανές λειτουργίες του Request Handler

Λειτουργία	**Περιγραφή λειτουργίας**	**REST κλήση**	**Ορίσματα**
deploy[87]	Υποβολή εφαρμογής	POST	Περιγραφή εφαρμογής σε YAML
track	Ανάκτηση της κατάστασης της εφαρμογής και των μικροϋπηρεσιών της	GET	Αναγνωριστικό εφαρμογής
inspect	Ανάκτηση του πιθανού πλάνου χρονοδρομολόγησης της εφαρμογής, δηλαδή των πόρων στους οποίους θα ανατεθεί η εκτέλεσή της αν υποβληθεί στο σύστημα	GET	Περιγραφή εφαρμογής σε YAML
remove[88]	Τερματισμός εφαρμογής	POST	Αναγνωριστικό εφαρμογής

[87] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/flask_app.py#L23
[88] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/flask_app.py#L62

Πίνακας 5.2: Σύνολο τιμών του πεδίου action[89]

Τιμή	Μηχανισμός επεξεργασίας του αιτήματος μετά τον Dispatcher	Μηχανισμός επεξεργασίας του αιτήματος μετά τον Resource Optimization Toolkit
deploy	Resource Optimization Toolkit	Orchestrator
track	State Database	-
inspect	Resource Optimization Toolkit	-
remove	Orchestrator	-

Εάν το πεδίο `action` έχει τιμή `deploy`, το αίτημα εκτέλεσης της εφαρμογής μεταφέρεται στον μηχανισμό Orchestrator. Ο μηχανισμός αυτός αρχικά δημιουργεί μια περιγραφή της υποβληθείσας για εκτέλεση εφαρμογής βάσει της σύνταξης που χρησιμοποιεί το εργαλείο ενορχήστρωσης του επιλεγμένου cluster. Στη συνέχεια, δεδομένου ότι το cluster είναι διαθέσιμο, προωθεί το αίτημα στον Driver του επιλεγμένου Local Orchestrator μαζί με τις επιπλέον οδηγίες χρονοδρομολόγησης που έχουν προκύψει από τον αλγόριθμο του Resource Optimization Toolkit.

Στην περίπτωση που ο χρήστης έχει αιτηθεί την εκτέλεση ή τον τερματισμό μιας εφαρμογής, δηλαδή το πεδίο `action` έχει τιμή `deploy` ή `remove` αντιστοίχως, το αίτημα επεξεργάζεται και ο Driver μηχανισμός. Συγκεκριμένα, από τη μία πλευρά, στην περίπτωση αιτήματος εκτέλεσης μιας εφαρμογής, ο Driver του Local Orchestrator που έχει επιλεχθεί αρχικά ερμηνεύει και μεταφράζει σε όρους που χρησιμοποιούνται από το εργαλείο ενορχήστρωσής του τις επιπλέον οδηγίες χρονοδρομολόγησης που έχει λάβει από το Resource Optimization Toolkit. Η μετάφραση αυτή πραγματοποιείται δεδομένου ότι οι οδηγίες χρονοδρομολόγησης μπορούν να ικανοποιηθούν από τις υποστηριζόμενες λειτουργίες του εργαλείου ενορχήστρωσης και συμπεριλαμβάνεται στην περιγραφή της εφαρμογής που έχει προκύψει από τον Orchestrator. Στη συνέχεια, ο Driver επικοινωνεί με τον master κόμβο του cluster και υποβάλλει την εφαρμογή για χρονοδρομολόγηση. Από το σημείο αυτό και μετά υπεύθυνο για την ομαλή εκτέλεση της εφαρμογής είναι το εργαλείο ενορχήστρωσης. Από την άλλη πλευρά, στην περίπτωση αιτήματος τερματισμού εφαρμογής, ο Driver επικοινωνεί απευθείας με τον master κόμβο.

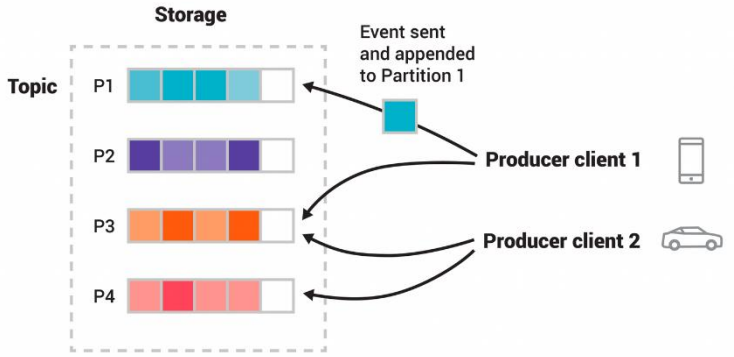
Για κάθε είδος εργαλείου ενορχήστρωσης που χρησιμοποιείται και άρα για κάθε ενσωματωμένο στο σύστημα Local Orchestrator, υλοποιείται ένας Driver μηχανισμός. Δεδομένου ότι, όπως προαναφέρθηκε στις προηγούμενες ενότητες της παρούσας εργασίας, ως Local Orchestrators επιλέχθηκαν το Docker SwarmKit με swarm mode και το Kubernetes, στην προκειμένη περίπτωση έχουν υλοποιηθεί ο Swarm και ο Kubernetes Driver αντιστοίχως. Επίσης, η προγραμματιστική επικοινωνία με τον manager κόμβο του swarm πραγματοποιήθηκε με κλήσεις του Swarm Driver στο Docker CLI (Ενότητα 3.3.1) σε Python subprocess[90], δεδομένου ότι δεν υπάρχει κάποια βιβλιοθήκη

[89] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/swarm_driver_app/src/config.py#L4
[90] https://docs.python.org/3/library/subprocess.html

προγραμματιστικής επικοινωνίας με το swarm σε Python. Στην περίπτωση του Kubernetes, στον Kubernetes Driver χρησιμοποιήθηκε η επίσημη βιβλιοθήκη της Python[91].

Για την επικοινωνία των μηχανισμών του συστήματος μεταξύ τους χρησιμοποιήθηκε η πλατφόρμα Apache Kafka[92]. Συνοπτικά, βάσει του σχεδιασμού της πλατφόρμας, κάθε μήνυμα ορίζει ένα event και σύνολα από events οργανώνονται και αποθηκεύονται μόνιμα σε διαμοιρασμένες στους διαθέσιμους πόρους (partitioned) δομές, τα topics, όπως απεικονίζεται στην Εικόνα 5.4. Οι εφαρμογές που πραγματοποιούν εγγραφές στα topics αναφέρονται ως producers στην ορολογία του Kafka, ενώ εκείνες που διαβάζουν από τα topics ως consumers. Μία από τις κυριότερες σχεδιαστικές επιλογές του Kafka, η οποία διευκολύνει την ευρεία κλιμάκωση των εφαρμογών που το χρησιμοποιούν, είναι ότι δεν υπάρχει καμία εξάρτηση μεταξύ των producers και των consumers [103].

Για να επιτευχθεί ακόμη η σύγχρονη και ασύγχρονη ανταλλαγή μηνυμάτων, χρησιμοποιήθηκε η βιβλιοθήκη επεξεργασίας ροών δεδομένων Faust[93]. Πρόκειται για μία βιβλιοθήκη ανοιχτού κώδικα που αρχικά αναπτύχθηκε από την Robinhood. Η βιβλιοθήκη Faust υλοποιεί σε Python τις περισσότερες από τις υποστηριζόμενες λειτουργίες του πιο διαδεδομένου εργαλείου Kafka Streams[94], δεδομένου ότι αυτές μπορούν να χρησιμοποιηθούν αποδοτικά μόνο από εφαρμογές Java. Ο Πίνακας 5.3 συνοψίζει τα Kafka topics που δημιουργήθηκαν στο σύστημα του Resource Orchestrator με χρήση της Faust, καθώς και το σχήμα των δεδομένων τους.



Εικόνα 5.4: Παράδειγμα εγγραφής μηνυμάτων στο Apache Kafka

Επιπροσθέτως, για τη διατήρηση της κατάστασης των εφαρμογών, στο σύστημα που αναπτύχθηκε περιλαμβάνεται μία βάση δεδομένων, η State Database. Στη βάση αυτή, αποθηκεύονται για κάθε εφαρμογή, πέρα από τα προσδιοριστικά της στοιχεία, και η τρέχουσα κατάστασή της στο σύστημα. Συγκεκριμένα, οι μηχανισμοί του συστήματος επεξεργάζονται το αίτημα υποβολής μιας εφαρμογής, η κατάσταση της εφαρμογής μεταβάλλεται στο σύνολο τιμών που απεικονίζεται στον Πίνακα 5.4. Κάθε φορά που ένας μηχανισμός πραγματοποιεί μια αλλαγή κατάστασης, ενημερώνει το πεδίο state και το

[91] https://github.com/kubernetes-client/python
[92] https://kafka.apache.org
[93] https://faust.readthedocs.io/en/latest/
[94] https://kafka.apache.org/documentation/streams/

πεδίο timestamp της κατάλληλης εγγραφής στη βάση δεδομένων με τη νέα τιμή της κατάστασης και τη χρονική στιγμή που πραγματοποιήθηκε η μεταβολή αντίστοιχα. Μελλοντικά, θα μπορούσε να αναπτυχθεί και ένας μηχανισμός ώστε οι Local Orchestrators να ενημερώνουν τη βάση για την κατάσταση της εφαρμογής στους υπολογιστικούς κόμβους στους οποίους εκτελείται.

Τέλος, στην τρέχουσα υλοποίηση, χρησιμοποιήθηκε η μη-σχεσιακή βάση δεδομένων (non-relational database) MongoDB[95]. Επιλέχθηκε μη-σχεσιακή βάση για μεγαλύτερη ευελιξία ως προς το σχήμα των ροών δεδομένων που αποθηκεύονται και για μεγαλύτερη απόδοση, δεδομένου ότι στο σύστημα δεν πραγματοποιούνται σύνθετα ερωτήματα (queries) στη βάση και δεν είναι αναγκαία η συστηματική οργάνωση των δεδομένων. Επίσης, στην αρχιτεκτονική που αναπτύχθηκε, τα events που εγγράφονται στο topic `db_consumer` καταχωρούνται απευθείας στη βάση, χωρίς να έχει αναπτυχθεί μια ξεχωριστή λειτουργία consumer, χρησιμοποιώντας το open-source εργαλείο Kafka Connect[96].

Πίνακας 5.3: Τα topics στο PoC σύστημα του Resource Orchestrator

Topic[97]	**Producer**	**Consumer**	**Σχήμα δεδομένων**
dispatcher	Request Handler	Dispatcher	ComponentsRecord[98]: • requestUUID (string): μοναδικό προσδιοριστικό του αιτήματος του χρήστη[99] • action (string): καθορίζει τις ενέργειες του συστήματος που θα πραγματοποιηθούν βάσει του αιτήματος του χρήστη • yamlSpec (dict): η περιγραφή της εφαρμογής αν πρόκειται για αίτημα εκτέλεσης εφαρμογής, αλλιώς το πεδίο είναι κενό
resource_optimization_toolkit	Dispatcher	Resource Optimization Toolkit	ComponentsRecord
orchestrator	Resource Optimization Toolkit	Orchestrator	ComponentsRecord
swarm	Orchestrator	Swarm Local Orchestrator	SwarmRecord[100]: • requestUUID (string) • namespace (string): το πεδίο ονομάτων της εφαρμογής, όπως έχει καθοριστεί στην περιγραφή της • name (string): το όνομα της εφαρμογής, όπως έχει καθοριστεί στην περιγραφή της • yamlSpec (dict) • action (string)

[95] https://www.mongodb.com
[96] https://docs.confluent.io/platform/current/connect/index.html
[97] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/config.py#L54
[98] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/models.py#L4
[99] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/flask_app.py#L31
[100] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/models.py#L18

kubernetes	Orchestrator	Kubernetes Local Orchestrator	KubernetesRecord[101]: • requestUUID (string) • namespace (string) • name (string) • yamlSpec (dict) • action (string)
db_consumer	Request Handler, Dispatcher, Orchestrator, Swarm Local Orchestrator, Kubernetes Local Orchestrator	MongoDB	DatabaseRecord[102]: • requestUUID (string) • namespace (string) • name (string) • state (string) • resource (string): ο Local Orchestrator που επιλέχθηκε για την ενορχήστρωση της εφαρμογής, προσδιορίζεται από το Orchestrator • yamlSpec (dict) • timestamp (string): η χρονική στιγμή εγγραφής του συγκεκριμένου event στο topic

Πίνακας 5.4: Το σύνολο τιμών του πεδίου state[103]

Κατάσταση εφαρμογής	**Περιγραφή**
new	Το αίτημα εκτέλεσης της εφαρμογής έχει ληφθεί από τον Request Handler
dispatched	Το αίτημα εκτέλεσης της εφαρμογής έχει επεξεργαστεί από τον Dispatcher
pending	Το αίτημα εκτέλεσης της εφαρμογής έχει ανατεθεί στον Orchestrator
deployed	Η χρονοδρομολόγηση της εφαρμογής στο τοπικό cluster ήταν επιτυχής
failed	Η χρονοδρομολόγηση της εφαρμογής στο τοπικό cluster απέτυχε
removed	Η εφαρμογή τερματίστηκε από τον Driver του Local Orchestrator

5.3 Περιβάλλοντα Ανάπτυξης και Εκτέλεσης

5.3.1 Περιβάλλον Ανάπτυξης

Ως περιβάλλον ανάπτυξης (development environment) αναφέρεται το σύνολο των εργαλείων που χρησιμοποιούνται από τις ομάδες ανάπτυξης για την υλοποίηση και τον τοπικό έλεγχο του συστήματος. Τα εργαλεία αυτά είναι εγκατεστημένα στον τοπικό σταθμό εργασίας, ο οποίος αποτελείται συνήθως από προσωπικούς υπολογιστές και τοπικούς servers. Παρ' όλο που το περιβάλλον ανάπτυξης δεν περιγράφει συνήθως επαρκώς το πραγματικό περιβάλλον εκτέλεσης και τους περιορισμούς του, διευκολύνει σημαντικά την ανάπτυξη, τον έλεγχο και την επέκταση της υλοποίησης.

[101] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/models.py#L26
[102] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/models.py#L9
[103] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/config.py#L9

Το σύστημα του Resource Orchestrator αναπτύχθηκε τοπικά σε Ubuntu 20.04. Για τη δημιουργία του και διαχείριση ενός τοπικού swarm, εγκαταστάθηκαν τα εργαλεία Docker Engine[104] και Docker Compose[105]. Για τη δημιουργία και παραμετροποίηση του Kubernetes cluster χρησιμοποιήθηκε το εργαλείο kind[106].

Αξίζει να σημειωθεί ότι έχουν αναπτυχθεί πολλαπλά εργαλεία για τη διαχείριση ενός τοπικού Kubernetes cluster. Κατά τη δημιουργία του περιβάλλοντος ανάπτυξης, συγκρίθηκαν τα πιο διαδεδομένα από αυτά. Ο Πίνακας 5.5 παρουσιάζει τα αποτελέσματα της σύγκρισης [104]. Τελικά, χρησιμοποιήθηκε το kind (Kubernetes in Docker), ένα open-source εργαλείο που έχει αναπτυχθεί σε Go και σχεδιάστηκε αρχικά για τον έλεγχο της πλατφόρμας του Kubernetes. Το kind επιλέχθηκε μεταξύ άλλων γιατί υποστηρίζει τη δημιουργία συμπλεγμάτων πολλαπλών κόμβων υψηλής διαθεσιμότητας (multi-node highly available clusters) τοπικά, τα οποία χρησιμοποιούνται συνήθως στο περιβάλλον εκτέλεσης. Επίσης, δεδομένου ότι το τοπικό cluster εκτελείται σε Docker containers, μειώνεται σημαντικά ο χρόνος εκκίνησής του. Ακόμη, το kind είναι ένα πιστοποιημένο από το CNCF εργαλείο εγκατάστασης του Kubernetes. Κατά συνέπεια, το kind είναι σε πολύ μεγάλο βαθμό συμβατό με τα σχετικά με την πλατφόρμα εργαλεία και εκτενώς τεκμηριωμένο.

Πίνακας 5.5: Σύγκριση εργαλείων τοπικής δημιουργίας και διαχείρισης Kubernetes clusters

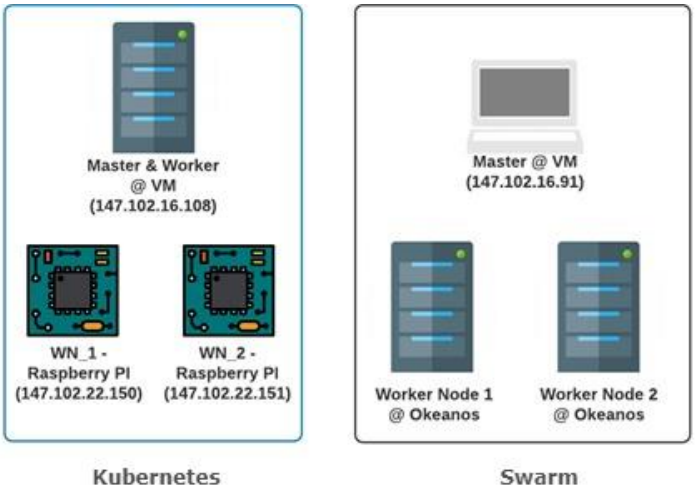
Εργαλείο	**Υποστηριζόμενες τοπολογίες**	**Λογισμικό υλοποίησης υπολογιστικού κόμβου**	**Υποστηριζόμενες πλατφόρμες (χωρίς χρήση εικονικού μηχανήματος)**	**Λειτουργίες για βελτιωμένη εμπειρία χρήσης**
Kind	ενός κόμβου, πολλαπλών κόμβων, πολλαπλών κόμβων υψηλής διαθεσιμότητας	Docker container	Linux	CLI, δυνατότητα παραμετροποίησης του cluster σε YAML αρχείο
K3s[107]	ενός κόμβου, πολλαπλών κόμβων	διεργασία λειτουργικού συστήματος	Linux	-
Minikube[108]	ενός κόμβου	Linux VM	Linux, macOS, Windows	CLI
MicroK8s[109]	ενός κόμβου, πολλαπλών κόμβων, πολλαπλών κόμβων υψηλής διαθεσιμότητας	Ubuntu OS	Ubuntu	- (ενδείκνυται η χρήση του multipass για macOS και Windows)

5.3.2 Περιβάλλον Εκτέλεσης

Το περιβάλλον εκτέλεσης (production environment) συνίσταται από τους πραγματικούς πόρους υλικού και λογισμικού στους οποίους εκτελούνται οι εφαρμογές των χρηστών και τα

[104] https://docs.docker.com/engine/install/ubuntu/
[105] https://docs.docker.com/compose/install/
[106] https://kind.sigs.k8s.io/docs/user/quick-start/
[107] https://k3s.io
[108] https://minikube.sigs.k8s.io/docs/
[109] https://microk8s.io

χρησιμοποιούμενα εργαλεία λογισμικού. Λόγω των περιορισμένων διαθέσιμων πόρων, οι τοπολογίες που αναπτύχθηκαν για το swarm και το Kubernetes cluster είναι πολύ απλές, όπως φαίνεται στην Εικόνα 5.5. Συγκεκριμένα, το swarm αποτελείται από έναν master κόμβο και δύο ξεχωριστούς worker κόμβους. Κάθε worker κόμβος του swarm διαθέτει 1 επεξεργαστή, 8 GB μνήμης RAM και 16 GB δίσκου. Το Kubernetes cluster συνίσταται επίσης από ένα master και δύο workers, με κάθε worker κόμβο να έχει 2 πυρήνες επεξεργαστή, 8 GB μνήμης RAM και 60 GB δίσκου.



Εικόνα 5.5: Το περιβάλλον εκτέλεσης των Local Orchestrators

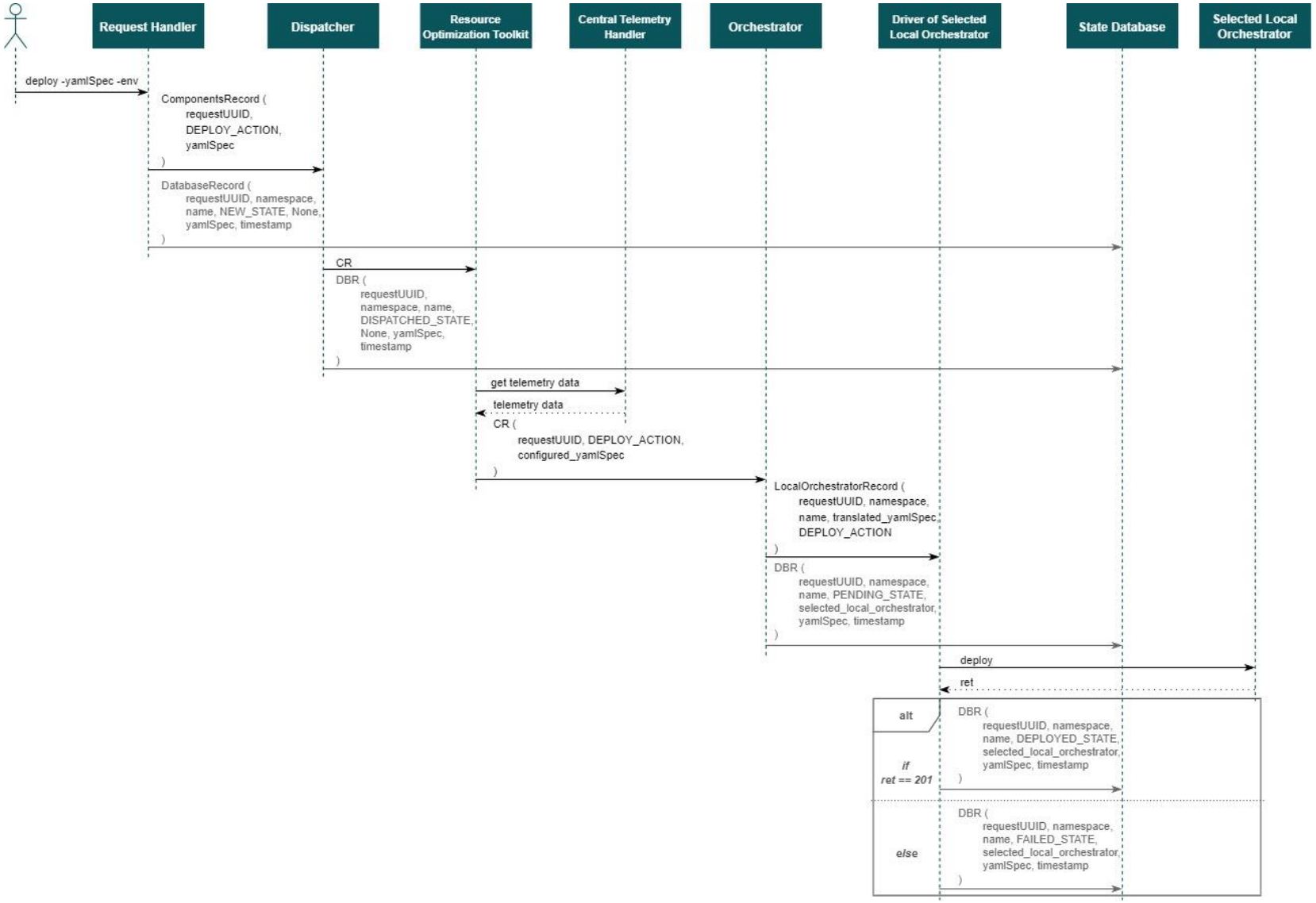
5.4 Υλοποιημένες Λειτουργίες

Στην τρέχουσα PoC υλοποίηση μπορούν να εξυπηρετηθούν μόνο αιτήματα υποβολής μιας εφαρμογής για εκτέλεση και τερματισμού.

5.4.1 Υποβολή Εφαρμογής για Εκτέλεση

Το ακολουθιακό διάγραμμα της Εικόνα 5.6 συνοψίζει τις ενέργειες που πραγματοποιούνται στο σύστημα του Resource Orchestrator για την εξυπηρέτηση ενός αιτήματος υποβολής μιας εφαρμογής για εκτέλεση. Στα βέλη προσδιορίζονται τα δεδομένα που γράφονται στα σχετικά topics, το σχήμα τους (`ComponentsRecord` – CR, `DatabaseRecord` – DBR, LocalOrchestratorRecord), καθώς και αιτήματα μεταξύ των μηχανισμών του συστήματος. Στις Εικόνες Εικόνα 5.9 έως Εικόνα 5.12 απεικονίζονται στιγμιότυπα του συστήματος κατά την εξυπηρέτηση ενός τέτοιου αιτήματος.

Πιο συγκεκριμένα, στην τρέχουσα υλοποίηση, για την εκτέλεση μιας εφαρμογής στο σύστημα καλείται η εντολή `<python3_path> ./deploy.py -yamlSpec -env` στον φάκελο `serrano_app` του GitHub repository. Στο όρισμα `yamlSpec` προσδιορίζεται η απόλυτη τοποθεσία (absolute path) στην οποία βρίσκεται, στο τοπικό μηχάνημα, το YAML αρχείο με τη γενική περιγραφή της εφαρμογής. Το όρισμα `env` παίρνει τιμές `local`, αν πρόκειται να χρησιμοποιηθεί το περιβάλλον ανάπτυξης, ή `prod` για το περιβάλλον εκτέλεσης.



Εικόνα 5.6: Ακολουθιακό διάγραμμα εξυπηρέτησης αιτήματος υποβολής εφαρμογής

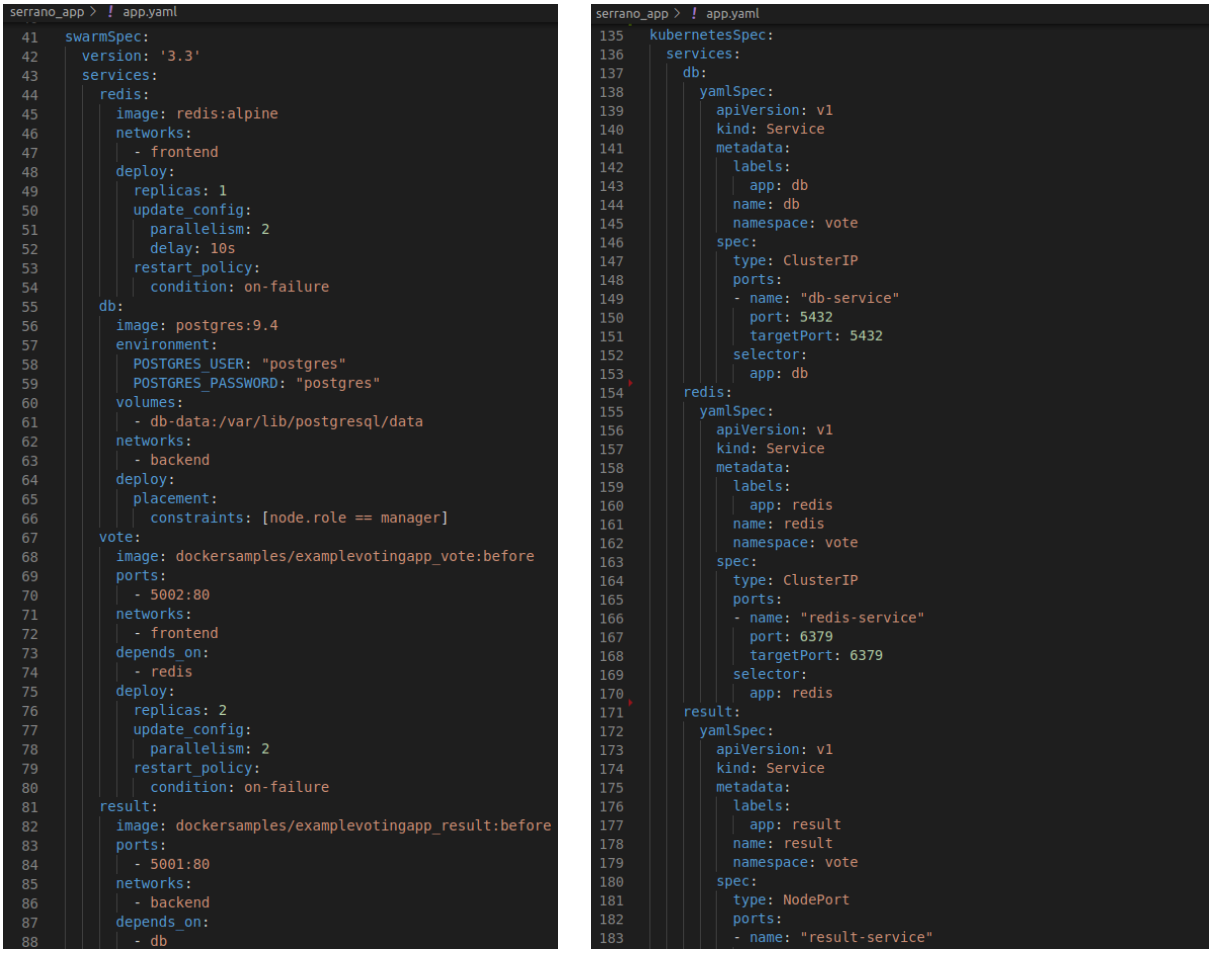
Στην Εικόνα 5.7 παρουσιάζεται το απλό σχήμα του αρχείου YAML που θεωρήθηκε για την περιγραφή των εφαρμογών που υποβάλλονται στο PoC σύστημα του Resource Orchestrator. Συγκεκριμένα, στο πεδίο `name` ο χρήστης προσδιορίζει το όνομα της εφαρμογής και στο `namespace` το σύνολο ονομάτων αναφοράς. Επίσης, όπως θα αναλυθεί παρακάτω, ο αλγόριθμος χρονοδρομολόγησης που υλοποιήθηκε λαμβάνει υπόψη για την επιλογή του κόμβου ανάθεσης τους πόρους επεξεργαστή, μνήμης και δίσκου που αιτούνται για την εφαρμογή συνολικά αλλά και για κάθε microservice της. Αυτές οι πληροφορίες προσδιορίζονται στα πεδία `appCores`, `appRAMGB` και `appStorageGB` για την εφαρμογή συνολικά, και κάτω από το πεδίο `app_comp_reqs` για κάθε microservice της.

Επιπροσθέτως, στο αρχείο YAML θεωρήθηκε ότι θα περιλαμβάνεται μια περιγραφή της εφαρμογής για κάθε Driver και άρα για κάθε εργαλείο ενορχήστρωσης που εξυπηρετεί το σύστημα. Η υπόθεση αυτή λήφθηκε δεδομένου ότι δεν ήταν αντικείμενο της παρούσας εργασίας ο προσδιορισμός μιας γενικής γλώσσας περιγραφής εφαρμογών, η οποία να είναι ανεξάρτητη των ενσωματωμένων εργαλείων ενορχήστρωσης. Επιπλέον, για τις πλατφόρμες ενορχήστρωσης Docker SwarmKit και Kubernetes που επιλέχθηκαν δε βρέθηκε εργαλείο γενικής περιγραφής που να υποστηρίζει επαρκώς τις δυνατότητές τους και να μεταφράζει επιτυχώς την περιγραφή της εφαρμογής για τη μία πλατφόρμα σε περιγραφή που να υποστηρίζεται από την άλλη (εργαλεία που υποστηρίζουν περιορισμένο αριθμό λειτουργιών π.χ. Kompose [110]). Έτσι, στο αρχείο YAML του PoC περιλαμβάνονται τα υποπεδία `swarmSpec` και `kubernetesSpec` για τις αντίστοιχες περιγραφές. Στην Εικόνα 5.8 απεικονίζεται ένα παράδειγμα αυτών των πεδίων για μία εφαρμογή πέντε μικροϋπηρεσιών, με το συνολικό YAML αρχείο να υπάρχει στο GitHub repository[111].

```yaml
# general app information
name: <value>
namespace: <value>

# requested resources for the app in total
appCores: <numerical_value>
appRAMGB: <numerical_value>
appStorageGB: <numerical_value>

# requested resources per app's microservice
app_comp_reqs:
  <add_microservice_name>:
    cores: <numerical_value>
    RAMGB: <numerical_value>
    storageGB: <numerical_value>

# app description, one per type of integrated orchestration mechanism
# for SwarmKit
swarmSpec:
  <swarm_specific_spec>

# for Kubernetes
kubernetesSpec:
  <kubernetes_specific_spec>
```

Εικόνα 5.7: Το αρχείο YAML για την περιγραφή εφαρμογής στο PoC του Resource Orchestrator

[110] https://kompose.io
[111] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/app.yaml



Εικόνα 5.8: Τα πεδία swarmSpec και kubernetesSpec του YAML αρχείου του PoC

Μετά την επεξεργασία του αιτήματος από τον Request Handler και τον Dispatcher, οι σχετικές πληροφορίες προωθούνται στον μηχανισμό Resource Optimization Toolkit. Ο αλγόριθμος χρονοδρομολόγησης [112] του Resource Optimization Toolkit στην PoC υλοποίηση του Resource Orchestrator είναι πολύ απλός και λαμβάνει αποφάσεις σε δύο επίπεδα. Πιο συγκεκριμένα, στο πρώτο επίπεδο, ο αλγόριθμος επιλέγει το cluster στο οποίο θα δρομολογηθεί για εκτέλεση η υποβληθείσα εφαρμογή[113]. Για τη λήψη αυτής της απόφασης, αρχικά εντοπίζει από όλα τα διαθέσιμα clusters αυτά που διαθέτουν περισσότερους πόρους επεξεργαστή, μνήμης και δίσκου από αυτούς που αιτείται η εφαρμογή συνολικά. Ακολούθως, αυτά τα cluster εισάγονται σε μία max heap, με προτεραιότητα που καθορίζεται από την τιμή:

[112] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/resource_optimization_toolkit/rot_algorithm.py#L175
[113] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/resource_optimization_toolkit/rot_algorithm.py#L44

$$value_{cluster} = \frac{w1*normalized[available\_cluster\_RAM - appRAMGB] + w2*normalized[available\_cluster\_storage - appStorageGB]}{w1+w2}$$

Όπως φαίνεται, για κάθε cluster προσδιορίζονται οι εναπομείναντες πόροι μνήμης και δίσκου αν δρομολογηθεί σε αυτό η εφαρμογή και οι οποίοι λαμβάνονται υπόψη με διαφορετικά βάρη w1 και w2. Τελικά, από τη max heap επιλέγεται το cluster μεγαλύτερης προτεραιότητας, δηλαδή αυτό το οποίο εκτιμάται ότι θα έχει τους περισσότερους διαθέσιμους πόρους μνήμης και δίσκου αν εκτελεστεί σε αυτό η εφαρμογή.

Στο δεύτερο επίπεδο[114], για κάθε microservice της εφαρμογής ο αλγόριθμος που υλοποιήθηκε προσδιορίζει έναν κόμβο του επιλεγμένου cluster στον οποίο προτείνεται η εκτέλεση του microservice. Για τον σκοπό αυτό, αρχικά ταξινομούνται τα microservices της εφαρμογής σε φθίνουσα σειρά ως προς την τιμή:

$$value_{service} = \frac{w1*normalized[serviceRAMGB] + w2*normalized[serviceStorageGB]}{w1+w2}$$

Στη συνέχεια, για κάθε microservice από την ταξινομημένη λίστα εντοπίζεται ο κόμβος που διαθέτει πόρους για την εκτέλεση της μικροϋπηρεσίας και αν επιλεχθεί για εκτέλεση, θα έχει τους περισσότερους διαθέσιμους πόρους μνήμης και δίσκου βάση της τιμής:

$$value_{node} = \frac{w1*normalized[available\_node\_RAM - serviceRAMGB] + w2*normalized[available\_node\_storage - serviceStorageGB]}{w1+w2}$$

Στην περίπτωση ισοπαλίας, επιλέγεται ένας κόμβος τυχαία. Επίσης, μόλις επιλεχθεί ένας κόμβος, ανανεώνονται οι διαθέσιμοι πόροι του σε:

$$available\_node\_RAM' = available\_node\_RAM - serviceRAMGB$$

$$available\_node\_storage' = available\_node\_storage - serviceStorageGB$$

Ο αλγόριθμος λαμβάνει τις απαραίτητες πληροφορίες για την κατάσταση των υποκείμενων υποδομών με REST κλήση στον μηχανισμό Central Telemetry Handler. Στο πλαίσιο του PoC συστήματος, για τον σκοπό αυτό υλοποιήθηκε ένα εξωτερικό REST API και ο Πίνακας 5.6 συνοψίζει τα endpoints του.

Μετά την εκτέλεση αυτού του αλγορίθμου, προστίθενται στο YAML αρχείο περιγραφής της εφαρμογής (configured_yamlSpec στην Εικόνα 5.6) το πεδίο `orchestrator` με τον Local Orchestrator του cluster που επιλέχθηκε και ένα πεδίο με τις επιπλέον οδηγίες χρονοδρομολόγησης προς τον μηχανισμό Orchestrator του συστήματος. Στην τρέχουσα υλοποίηση, το πεδίο αυτό ονομάζεται `comp_preferences`. Σε αυτό προσδιορίζεται, για κάποιες ή όλες τις μικροϋπηρεσίες της

[114] https://github.com/AthinaKyriakou/ntua-thesis-orchestrator/blob/main/serrano_app/src/resource_optimization_toolkit/rot_algorithm.py#L94

εφαρμογής, οι διευθύνσεις IP των κόμβων του cluster στους οποίους προτείνεται, αν είναι εφικτό, να ανατεθούν για εκτέλεση. Στην Εικόνα 5.9 φαίνεται ένα παράδειγμα του μηνύματος που αποστέλλει ο μηχανισμός του Resource Optimization Toolkit στο topic του Orchestrator. Για τον σκοπό αυτό, κάθε υπολογιστικός κόμβος του cluster έχει σημανθεί με μια ετικέτα `ip` με τιμή την IP του, όπως φαίνεται στην Εικόνα 5.10 για ένα Kubernetes cluster.

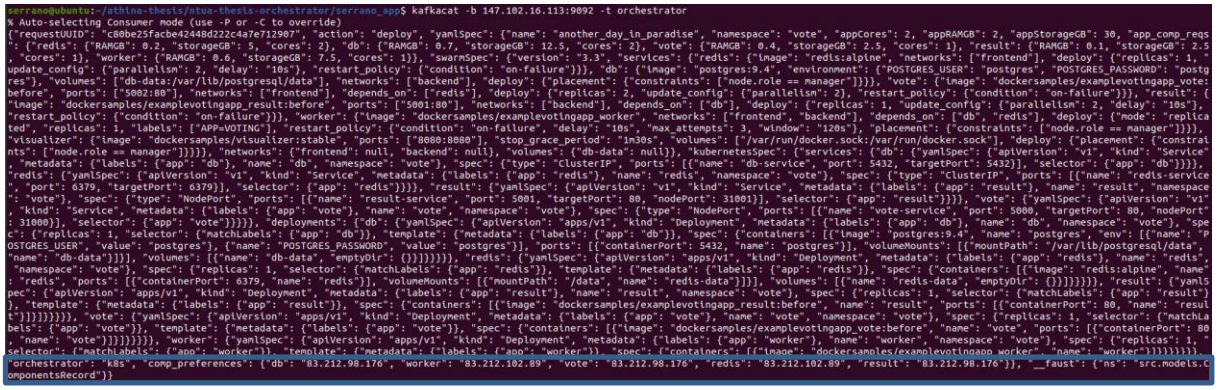
Πίνακας 5.6: Το API του Central Telemetry Handler στο PoC σύστημα

Περιγραφή Λειτουργίας	**REST Κλήση**	**Endpoint**	**Απάντηση**

Λήψη πληροφοριών για τα διαθέσιμα clusters του συστήματος	GET	/telemetry/sites	{
			“timestamp”: η χρονική στιγμή της κλήσης,
			“sites”: [
			// ένα αντικείμενο για κάθε cluster
			]
			}
			
			// αντικείμενο για κάθε cluster
			{
			“id”: προσδιοριστικό του cluster,
			“label”: ετικέτα σήμανσης του cluster,
			“master_node”: {“ip”: η διεύθυνση IP του master κόμβου του cluster},
			“orchestrator”: ο μηχανισμός ενορχήστρωσης του cluster, τιμές “K8S” ή “SWARM”
			}

Λήψη πληροφοριών τους κόμβους του cluster με προσδιοριστικό SITE_ID	GET	/telemetry/site/ SITE_ID	{
			“id”: προσδιοριστικό του cluster,
			“label”: ετικέτα σήμανσης του cluster,
			“total_ram_GB”: πόροι μνήμης του cluster σε GB,
			“total_storage_GB”: πόροι δίσκου του cluster σε GB,
			“total_cpus”: πλήθος επεξεργαστικών πυρήνων του cluster,
			“worker_nodes” : [
			{} // ένα αντικείμενο για κάθε worker κόμβο
			]
			}
			
			// αντικείμενο για κάθε worker κόμβο
			{
			“node_ram_GB”: πόροι μνήμης του κόμβου σε GB,
			“available_ram_GB”: διαθέσιμοι πόροι μνήμης του κόμβου σε GB,
			“node_storage_GB”: πόροι δίσκου του κόμβου σε GB,
			“available_storage_GB”: διαθέσιμοι πόροι δίσκου του κόμβου σε GB,
			“cpu_cores”: πλήθος επεξεργαστικών πυρήνων του κόμβου,
			“containers”: τα containers του κόμβου,
			“ip”: η IP διεύθυνση του κόμβου
			}

Στη συνέχεια, ο Orchestrator μεταφράζει τη γενική περιγραφή της εφαρμογής στη γλώσσα περιγραφής που χρησιμοποιεί το εργαλείο ενορχήστρωσης του Local Orchestrator που επιλέχθηκε (translated_yamlSpec στην Εικόνα 5.6). Στην τρέχουσα υλοποίηση, από το αρχείο YAML της γενικής περιγραφής, ο Orchestrator απομονώνει μόνο την περιγραφή του Local Orchestrator του επιλεγμένου cluster. Ακολούθως, προωθεί τη μεταφρασμένη περιγραφή της εφαρμογής και τις επιπλέον οδηγίες του Resource Optimization Toolkit στον κατάλληλο Driver.

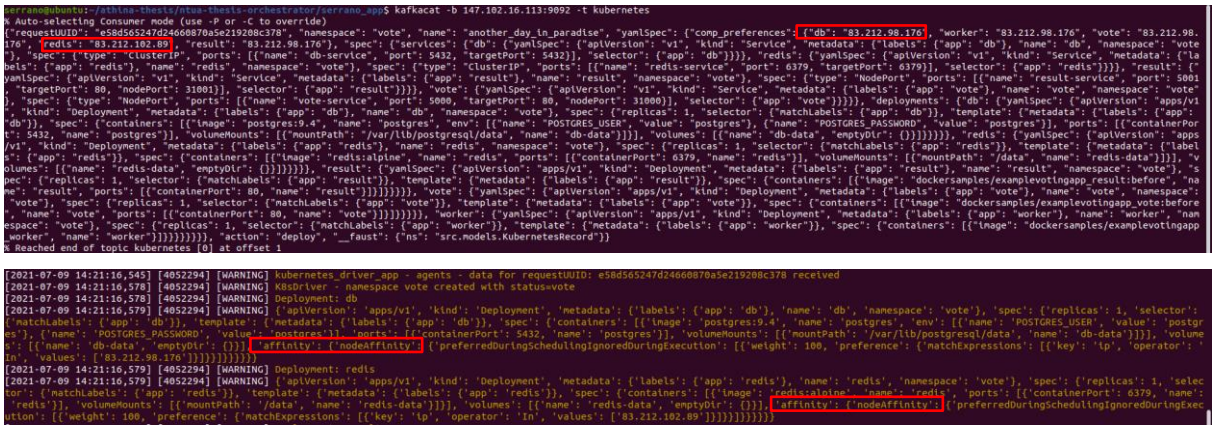


Εικόνα 5.9: Παράδειγμα μηνύματος του Resource Optimization Toolkit στον Orchestrator

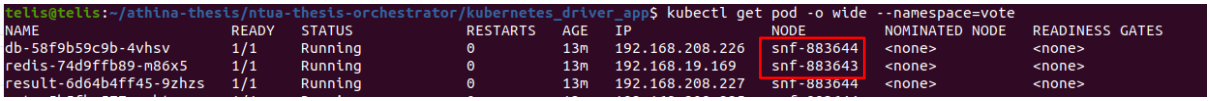
Εικόνα 5.10: Σήμανση των κόμβων ενός Kubernetes cluster με την IP διεύθυνσή τους

Λαμβάνοντας τα δεδομένα αυτά, ο Driver ερμηνεύει πρώτα τις οδηγίες του Resource Optimization Toolkit βάσει των υποστηριζόμενων λειτουργιών του εργαλείου ενορχήστρωσης του Local Orchestrator του και τροποποιεί κατάλληλα την περιγραφή της εφαρμογής. Στην τρέχουσα υλοποίηση, οι οδηγίες στο πεδίο `comp_preferences` αντιστοιχήθηκαν σε placement preferences στο επίπεδο των υπολογιστικών κόμβων στο Docker SwarmKit (Ενότητα 4.2.1) και σε οδηγίες affinity στο Kubernetes (Ενότητα 4.2.2). Ακολούθως, ο Driver υποβάλλει το αίτημα στον master κόμβο του Local Orchestrator. Στην Εικόνα 5.11 φαίνεται η μετάφραση των προτιμήσεων του πεδίου `comp_preferences` για τα microservices `db` και `redis` της εφαρμογής σε nodeAffinity κανόνες στο Kubernetes, όπως πραγματοποιήθηκε από τον Kubernetes Driver που επιλέχθηκε σε αυτό το παράδειγμα.

Τέλος, στην περίπτωση εύρεσης κόμβου για την εκτέλεση κάθε μικροϋπηρεσίας της εφαρμογής, το αίτημα θεωρείται ότι ικανοποιήθηκε επιτυχώς και το πεδίο `state` της εγγραφής στη βάση αποκτά τιμή `deployed`. Αν είναι εφικτό, ικανοποιούνται και οι προτιμήσεις του πεδίου `comp_preferences`, όπως πραγματοποιήθηκε στο παράδειγμα χρονοδρομολόγησης της Εικόνα 5.12 για τα microservices `db` και `redis`.



Εικόνα 5.11: Παράδειγμα αντιστοίχισης των προτιμήσεων του πεδίου comp_preferences σε node affinity κανόνες

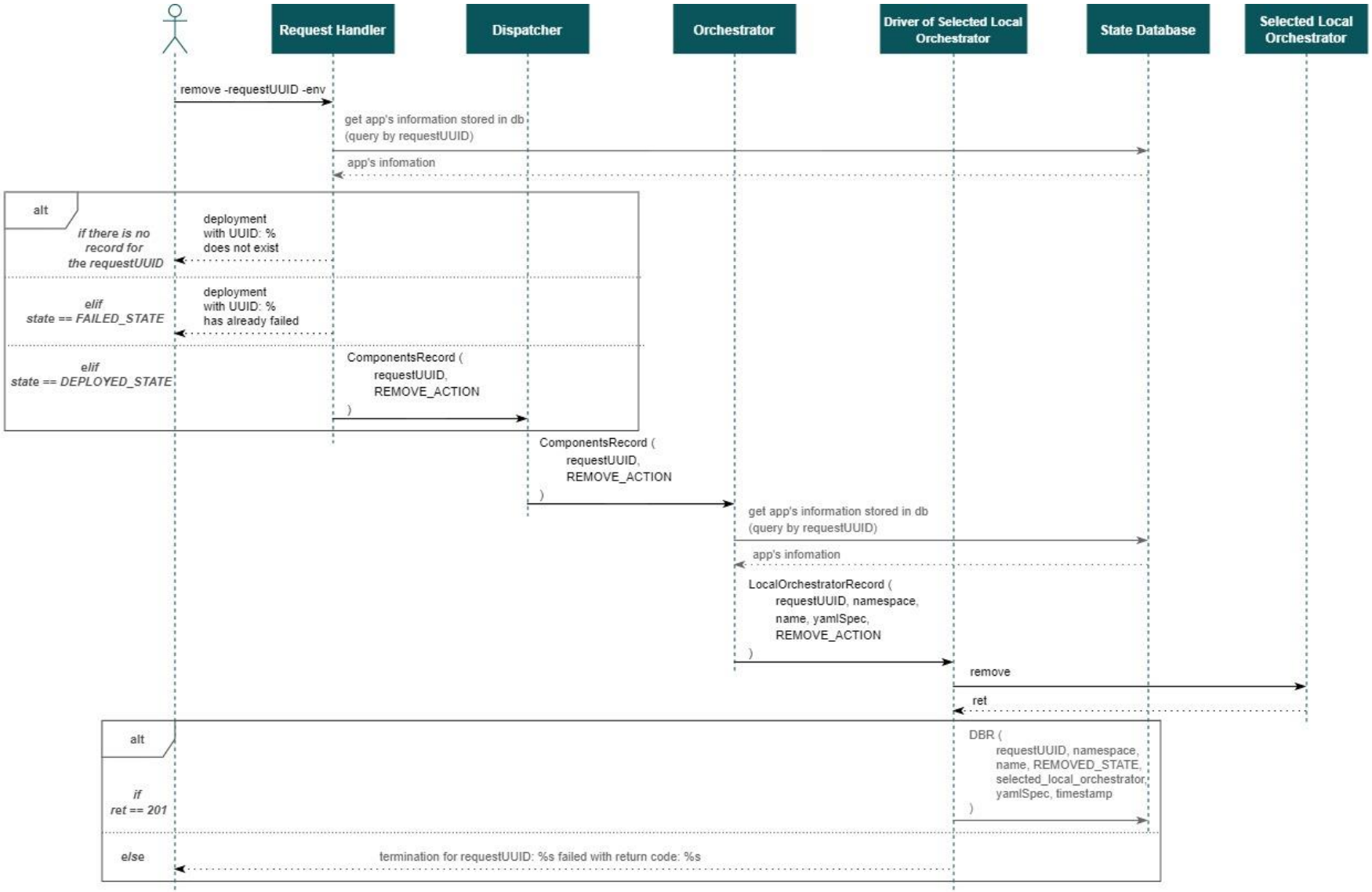


Εικόνα 5.12: Παράδειγμα επιτυχούς χρονοδρομολόγησης εφαρμογής

5.4.2 Τερματισμός Εφαρμογής

Το ακολουθιακό διάγραμμα της Εικόνα 5.13 συνοψίζει τις ενέργειες στο σύστημα του Resource Orchestrator για την εξυπηρέτηση ενός αιτήματος τερματισμού εφαρμογής. Στην τρέχουσα υλοποίηση, ο τερματισμός μιας εφαρμογής πραγματοποιείται βάσει της τιμής του `requestUUID` που έλαβε το αίτημα υποβολής της με κλήση της εντολής `<python3_path> ./remove.py -requestUUID -env` στον φάκελο `serrano_app`.

Το αίτημα τερματισμού προωθείται στον Dispatcher μόνο αν η εφαρμογή εκτελείται σε κάποιο από τα υποκείμενα clusters. Στην περίπτωση που τερματιστεί επιτυχώς η εφαρμογή, η τιμή του πεδίου `state` στη βάση ενημερώνεται σε `removed`, ενώ αν αποτύχει αποστέλλεται ενημερωτικό μήνυμα στον χρήστη.



Εικόνα 5.13: Ακολουθιακό διάγραμμα εξυπηρέτησης αιτήματος τερματισμού εφαρμογής

Κεφάλαιο 6: Αξιολόγηση του Συστήματος Ενορχήστρωσης και Συμπεράσματα

Σκοπός της παρούσας διπλωματικής εργασίας ήταν η μελέτη των υπαρχόντων τεχνολογιών και ο σχεδιασμός των απαραίτητων μηχανισμών για τη διαφανή προς τον τελικό χρήστη και αποτελεσματική ενορχήστρωση και εκτέλεση εγγενών εφαρμογών υπολογιστικού νέφους σε ένα σύστημα με ετερογενείς υπολογιστικές υποδομές, ανεξάρτητα από τις επιμέρους ιδιαιτερότητές τους. Για τον σκοπό αυτό, θεωρήθηκε ένα κατανεμημένο σύστημα με μηχανισμούς ενορχήστρωσης σε δύο επίπεδα και σχεδιάστηκε και υλοποιήθηκε μία απλή εκδοχή του. Βάσει της θεωρητικής μελέτης και του σχεδιασμού και της υλοποίησης αυτής της απλουστευμένης εκδοχής του συστήματος, προέκυψαν τα παρακάτω συμπεράσματα.

Πρώτον, η αρθρωτή αρχιτεκτονική με διεπαφές συμβάλει σημαντικά στην επεκτασιμότητα του συστήματος. Συγκεκριμένα, δίνει τη δυνατότητα στον διαχειριστή του συστήματος να επεκτείνει εύκολα ήδη υλοποιημένους μηχανισμούς, πραγματοποιώντας αλλαγές μόνο στη λογική οντότητα κώδικα (module) του μηχανισμού προς τροποποίηση, χωρίς να επηρεάζονται η λειτουργία ή ο κώδικας των υπόλοιπων μερών του συστήματος. Επίσης, διευκολύνει την ενσωμάτωση νέων εργαλείων, αφού απαιτείται μόνο η επέκταση ήδη υλοποιημένων διεπαφών.

Δεύτερον, αναφορικά με τη χρήση μιας γενικής και ανεξάρτητης των ενσωματωμένων πόρων περιγραφής της εφαρμογής στα κοντινότερα προς τον τελικό χρήστη επίπεδα της πλατφόρμας και αντιστοίχισης αυτής στις υποστηριζόμενες λειτουργίες των ενσωματωμένων εργαλείων ενορχήστρωσης στο επίπεδο του Driver, αξίζει να ληφθούν υπόψη τα ακόλουθα σημεία. Από τη μία πλευρά, σε θεωρητικό επίπεδο η χρήση γενικής περιγραφής απλοποιεί την ενσωμάτωση νέων τοπικών εργαλείων ενορχήστρωσης, καθώς χρειάζεται να υλοποιηθεί μόνο ένας νέος μηχανισμός Driver και να συνδεθεί με τη διεπαφή του Orchestrator, χωρίς να μεταβληθούν οι ήδη υλοποιημένοι μηχανισμοί. Ταυτόχρονα, η γενική περιγραφή των εφαρμογών διευκολύνει την ενσωμάτωση νέων τύπων πόρων στα υποκείμενα clusters, αλλά και την προσαρμογή στις αλλαγές που παρατηρούνται μεταξύ των εκδόσεων των τεχνολογιών ενορχήστρωσης. Παρατηρείται πολύ συχνά μεταξύ διαδοχικών εκδόσεων η δημιουργία νέων λειτουργιών, όπως και η επέκταση και κατάργηση υπαρχόντων, δεδομένου του αυξημένου ρυθμού ανάπτυξης αυτών των εργαλείων.

Ωστόσο, σε πρακτικό επίπεδο, η χρήση μιας γενικής γλώσσας περιγραφής εφαρμογών παρουσιάζει δυσκολίες. Ο σχεδιασμός ενός περιβάλλοντος στο οποίο οι χρήστες θα περιγράφουν με γενικό τρόπο της εφαρμογές τους (π.χ. γραφικό) χρειάζεται να λαμβάνει υπόψη επαρκώς τις λειτουργίες των υποστηριζόμενων συστημάτων και να αναπροσαρμόζεται στις αλλαγές τους. Πέρα από τη σύνθετη αρχική υλοποίηση ενός τέτοιου εργαλείου λογισμικού, κρίνεται αρκετά δύσκολη και η συντήρησή

του, καθώς και η πιθανή ενσωμάτωση σε αυτό νέων πλατφορμών ενορχήστρωσης. Ειδικότερα για τα SwarmKit και Kubernetes που μελετήθηκαν, δε βρέθηκε εργαλείο που να αξιοποιεί επαρκώς τις λειτουργίες των δύο συστημάτων κατά τη μετατροπή από τη μία περιγραφή στην άλλη.

Τρίτον, περιορισμοί εμφανίζονται και στον αλγόριθμο δρομολόγησης του Resource Optimization Toolkit. Πιο συγκεκριμένα, τα κριτήρια βάσει των οποίων λαμβάνεται η απόφαση δρομολόγησης και οι περαιτέρω οδηγίες που παρέχονται στους Local Orchestrators πρέπει να μπορούν να αντιστοιχηθούν στις υποστηριζόμενες λειτουργίες των ενσωματωμένων εργαλείων ενορχήστρωσης. Ωστόσο, αυτό δεν κρίνεται ότι θα αποτελέσει σημαντικό πρόβλημα σε πραγματικά περιβάλλοντα χρήσης, δεδομένου ότι συνήθως στο περιβάλλον εκτέλεσης χρησιμοποιούνται υψηλά επεκτάσιμες πλατφόρμες ενορχήστρωσης με πληθώρα παρεχόμενων λειτουργιών, όπως τα Kubernetes και Nomad, σε αντίθεση με το SwarmKit. Βέβαια, είναι απαραίτητο να προσδιοριστεί ο βαθμός οργάνωσης των μονάδων δρομολόγησης στον οποίο θα λαμβάνεται η απόφαση δρομολόγησης, δεδομένου ότι δεν παρατηρείται πλήρης αντιστοίχιση μεταξύ των τρόπων ομαδοποίησης των εκτελέσιμων μονάδων στα διάφορα εργαλεία ενορχήστρωσης (π.χ. τα task και service στο SwarmKit διαφέρουν από τα Pod και Deployment στο Kubernetes).

Λόγω αυτών των περιορισμών, προτείνεται η διερεύνηση ενός εναλλακτικού σχεδιασμού του συστήματος, ο οποίος ενσωματώνει ετερογενείς πόρους αλλά χρησιμοποιεί μόνο μία πλατφόρμα ενορχήστρωσης. Για παράδειγμα, θα μπορούσε να μελετηθεί ένα σύστημα αποτελούμενο από πολλαπλά ανεξάρτητα μεταξύ τους Kubernetes clusters σε ετερογενείς υποδομές, οι λειτουργίες των οποίων συντονίζονται από έναν κεντρικό ή αποκεντρωμένο μηχανισμό ενορχήστρωσης [105], [106].

Συμπερασματικά, η υλοποίηση ενός κατανεμημένου συστήματος διαχείρισης εφαρμογών σε ετερογενείς υποδομές με ετερογενείς πλατφόρμες ενορχήστρωσης, αν και είναι υποσχόμενη σε θεωρητικό επίπεδο, παρουσιάζει σημαντικές πρακτικές δυσκολίες. Χρειάζεται να μελετηθούν εκτενέστερα τα σημεία που προαναφέρθηκαν και ενδεχομένως να εξεταστούν και εναλλακτικές αρχιτεκτονικές, οι οποίες ενδεχομένως παρουσιάζουν ετερογένεια μόνο στο επίπεδο των πόρων υλικού και λογισμικού και όχι στις τεχνολογίες ενορχήστρωσης.

Βιβλιογραφία

[1] I. C. Education, “Virtualization,” IBM, 19 June 2019. [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide. [Accessed 18 October 2020].

[2] S. Vennam, “Cloud Computing,” IBM, 18 August 2020. [Online]. Available: https://www.ibm.com/cloud/learn/cloud-computing. [Accessed 18 October 2020].

[3] A. W. Services, “What is cloud computing?,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/. [Accessed 18 October 2020].

[4] A. Larkin, “Disadvantages of Cloud Computing,” Cloud Academy Inc., 7 August 2019. [Online]. Available: https://cloudacademy.com/blog/disadvantages-of-cloud-computing/. [Accessed 18 October 2021].

[5] I. C. Team, “Cloud Native Applications,” IBM, 15 May 2019. [Online]. Available: https://www.ibm.com/cloud/learn/cloud-native. [Accessed 19 October 2021].

[6] D. Gannon, R. Barga and N. Sundaresan, “Cloud-Native Applications,” *IEEE Cloud Computing,* vol. 4, no. 5, pp. 16-21, 2017.

[7] “Introduction to cloud-native applications,” Microsoft, 19 January 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduction. [Accessed 2 February 2021].

[8] C. Paganini, “An Introduction to the Cloud Native Landscape,” The New Stack, 21 July 2020. [Online]. Available: https://thenewstack.io/an-introduction-to-the-cloud-native-landscape/. [Accessed 1 December 2020].

[9] T. L. Foundation, “2021 State of the Edge,” The Linux Foundation, 2021.

[10] “Open Glossary of Edge Computing,” The Linux Foundation, 30 June 2020. [Online]. Available: https://github.com/State-of-the-Edge/glossary/blob/master/edge-glossary.md. [Accessed 15 May 2021].

[11] J. Minor, “SMARTER: An Approach to Edge Compute Observability and Performance Monitoring,” Arm Research, 16 April 2020. [Online]. Available: https://community.arm.com/developer/research/b/articles/posts/an-approach-to-edge-compute-observability-and-performance-monitoring. [Accessed 2 April 2021].

[12] A. Iyengar and C. Ouyang, “Edge computing architecture,” IBM, [Online]. Available: https://www.ibm.com/cloud/architecture/architectures/edge-computing/. [Accessed 2 April 2021].

[13] T. L. Foundation, “Sharpening the Edge: Overview of the LF Edge Taxonomy and Framework,” The Linux Foundation, 2020.

[14] M. Satyanarayanan, G. Klas, M. Silva and S. Mangiante, “The Seminal Role of Edge-Native Applications,” in *2019 IEEE International Conference on Edge Computing (EDGE)*, Milan, Italy, 2019.

[15] A. Betts, “Becoming edge native in your application design,” 23 December 2020. [Online]. Available: https://www.fastly.com/blog/becoming-edge-native-in-your-application-design. [Accessed 2 February 2021].

[16] V. Ishu, “Developing at the edge: Best practices for edge computing,” 16 July 2020. [Online]. Available: https://developers.redhat.com/blog/2020/07/16/developing-at-the-edge-best-practices-for-edge-computing#. [Accessed 2 December 2020].

[17] I. C. Education, “Containers,” 23 June 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containers. [Accessed 30 June 2021].

[18] Microsoft, “Containers vs. virtual machines,” 21 October 2019. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/containers-vs-vm. [Accessed 5 December 2020].

[19] R. Hat, “Containers vs VMs,” [Online]. Available: https://www.redhat.com/en/topics/containers/containers-vs-vms. [Accessed 3 December 2020].

[20] I. C. Education, “Containerization,” 23 June 2021. [Online]. Available: https://www.ibm.com/cloud/learn/containerization. [Accessed 2 July 2021].

[21] “What is container orchestration?,” 2 December 2019. [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration. [Accessed 15 December 2020].

[22] I. C. Education, “Container Orchestration,” 27 May 2021. [Online]. Available:

https://www.ibm.com/cloud/learn/container-orchestration. [Accessed 1 July 2021].

[23] L. -. LFS151.x, “Introduction to Cloud Infrastructure Technologies,” 2021.

[24] L. -. LFS158x, “Introduction to Kubernetes,” 2021.

[25] T. O. U. Committee, “2018 OpenStack User Survey Report,” The OpenStack Foundation, 2018.

[26] E. Truyen, D. Van Landuyt, D. Preuveneers, B. Lagaisse and W. Joosen, “A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks,” *Applied Sciences,* vol. 9, no. 5, pp. 931-1007, 2019.

[27] S. Sengupta, “Kubernetes vs Docker Swarm: Comparing Container Orchestration Tools,” 11 November 2020. [Online]. Available: https://www.bmc.com/blogs/kubernetes-vs-docker-swarm/. [Accessed 27 February 2021].

[28] E. Casalicchio, “Container Orchestration: A Survey,” in *Systems Modeling: Methodologies and Tools*, Springer, Cham, 2019, pp. 221-235.

[29] HashiCorp, “Nomad vs. Kubernetes,” 31 March 2021. [Online]. Available: https://www.nomadproject.io/docs/nomad-vs-kubernetes. [Accessed 23 June 2021].

[30] D. Labs, “Understanding Difference between Docker Swarm(Classic), Swarm Mode & SwarmKit,” [Online]. Available: https://dockerlabs.collabnix.com/intermediate/swarm/difference-between-docker-swarm-vs-swarm-mode-vs-swarmkit.html . [Accessed 1 July 2021].

[31] OpenStack, “OpenStack User Survey 2021,” The OpenStack Foundation, 2021.

[32] D. Inc, “Swarm mode key concepts,” 8 April 2020. [Online]. Available: https://docs.docker.com/engine/swarm/key-concepts/. [Accessed 3 April 2021].

[33] D. Inc, “Swarm mode overview,” 25 March 2021. [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed 5 April 2021].

[34] D. Inc, “How nodes work,” 8 April 2020. [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/. [Accessed 5 April 2021].

[35] D. Inc, “Deploy to Swarm,” 5 January 2021. [Online]. Available: https://docs.docker.com/get-started/swarm-deploy/. [Accessed 4 April 2021].

[36] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” in *2014 USENIX Annual Technical Conference*, Philadelphia, PA, 2014.

[37] D. Inc, “Raft consensus in swarm mode,” 13 October 2020. [Online]. Available: https://docs.docker.com/engine/swarm/raft/. [Accessed 6 April 2021].

[38] T. K. Authors, “What is Kubernetes?,” 1 February 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 2 April 2021].

[39] T. K. Authors, “Kubernetes Components,” 3 January 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 2 April 2021].

[40] J. Beda, “Core Kubernetes: Jazz Improv over Orchestration,” 30 March 2017. [Online]. Available: https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca. [Accessed 2 April 2021].

[41] The Kubernetes Authors, “What is Kubernetes?,” 1 February 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed 2 April 2021].

[42] D. Inc., “How services work,” 26 January 2018. [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/. [Accessed 3 April 2021].

[43] Docker Inc., “Key Concepts,” 8 April 2020. [Online]. Available: https://github.com/docker/docker.github.io/blob/master/engine/swarm/key-concepts.md. [Accessed 4 August 2021].

[44] Docker Inc., “The Compose Specification,” 27 April 2021. [Online]. Available: https://github.com/compose-spec/compose-spec/blob/master/spec.md. [Accessed 6 May 2021].

[45] Docker Inc., “Deploy services to a swarm,” 26 October 2020. [Online]. Available: https://docs.docker.com/engine/swarm/services/. [Accessed 6 May 2021].

[46] Docker Inc., “Store configuration data using Docker Configs,” 14 August 2021. [Online]. Available: https://docs.docker.com/engine/swarm/configs/. [Accessed 13 September 2021].

[47] Docker Inc., “About secrets,” 19 August 2021. [Online]. Available: https://docs.docker.com/engine/swarm/secrets/. [Accessed 13 September 2021].

[48] Docker Inc., “Deploy a stack to a swarm,” 6 August 2021. [Online]. Available:

https://docs.docker.com/engine/swarm/stack-deploy/. [Accessed 13 September 2021].

[49] The Kubernetes Authors, “Understanding Kubernetes Objects,” 19 August 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/. [Accessed 10 September 2021].

[50] The Kubernetes Authors, “Kubernetes API,” 19 January 2021. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/. [Accessed 14 September 2021].

[51] The Kubernetes Authors, “Client Libraries,” 26 May 2021. [Online]. Available: https://kubernetes.io/docs/reference/using-api/client-libraries/. [Accessed 14 September 2021].

[52] The Kubernetes Authors, “API Overview,” 5 August 2021. [Online]. Available: https://kubernetes.io/docs/reference/using-api/. [Accessed 9 September 2021].

[53] The Kubernetes Authors, “API Conventions,” 14 September 2021. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md. [Accessed 20 September 2021].

[54] The Kubernetes Authors, “Kubernetes API Concepts,” 5 August 2021. [Online]. Available: https://kubernetes.io/docs/reference/using-api/api-concepts/. [Accessed 24 September 2021].

[55] The Kubernetes Authors, “Create static Pods,” 26 August 2021. [Online]. Available: https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/. [Accessed 26 September 2021].

[56] The Kubernetes Authors, “Controllers,” 14 June 2021. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/. [Accessed 24 September 2021].

[57] The Kubrnetes Authors, “Exploring Kubernetes Operator Pattern,” 31 January 2021. [Online]. Available: https://iximiuz.com/en/posts/kubernetes-operator-pattern/. [Accessed 15 May 2021].

[58] M. Ahmed, “Deploying An Application On Kubernetes From A to Z,” 14 April 2020. [Online]. Available: https://www.magalix.com/blog/deploying-an-application-on-kubernetes-from-a-to-z. [Accessed 28 April 2021].

[59] Google Developers, “Pod,” 5 October 2021. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/pod. [Accessed 9 October 2021].

[60] The Kubernetes Authors, “Labels and Selectors,” 1 July 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/. [Accessed 9 October 2021].

[61] V. Lancey, “Deprecated APIs Removed In 1.16: Here’s What You Need To Know,” 18 July 2019. [Online]. Available: https://kubernetes.io/blog/2019/07/18/api-deprecations-in-1-16/. [Accessed 29 September 2021].

[62] The Kubernetes Authors, “Deployment,” 26 June 2021. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/. [Accessed 30 June 2021].

[63] The Kubernetes Authors, “ReplicaSet,” 29 June 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. [Accessed 30 June 2021].

[64] The Kubernetes Authors, “Deployments,” 18 March 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Accessed 30 June 2021].

[65] The Kubernetes Authors, “Using a Service to Expose Your App,” 22 July 2021. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/. [Accessed 10 August 2021].

[66] The Kubernetes Authors, “Service,” 26 June 2021. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/service-resources/service-v1/. [Accessed 10 August 2021].

[67] The Kubernetes Authors, “Service,” 3 October 2021. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/. [Accessed 7 October 2021].

[68] The Kubernetes Authors, “Ingress,” 26 June 2021. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/service-resources/ingress-v1/. [Accessed 6 September 2021].

[69] The Kubernetes Authors, “Ingress,” 30 June 2021. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/. [Accessed 6 September 2021].

[70] The Kubernetes Authors, “Ingress Controllers,” 21 July 2021. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/. [Accessed 6 September 2021].

[71] The Kubernetes Authors, “StatefulSets,” 28 September 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. [Accessed 1 October 2021].

[72] The Kubernetes Authors, “StatefulSet,” 26 June 2021. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/stateful-set-v1/. [Accessed 1 October 2021].

[73] The Kubernetes Authors, “ConfigMaps,” 25 June 2021. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/. [Accessed 1 October 2021].

[74] The Kubernetes Authors, “Secrets,” 4 August 2021. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/. [Accessed 1 October 2021].

[75] D. Inc., “Swarm task states,” 3 July 2020. [Online]. Available: https://docs.docker.com/engine/swarm/how-swarm-mode-works/swarm-task-states/. [Accessed 3 April 2021].

[76] D. Inc., “SwarmKit task model,” 31 December 2018. [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/task_model.md#task-lifecycle. [Accessed 3 April 2021].

[77] The Kubernetes Authors, “Pod Lifecycle,” 23 July 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. [Accessed 17 September 2021].

[78] T. Menouer, C. Cérin and É. Leclercq, “New Multi-objectives Scheduling Strategies in Docker SwarmKit,” in *Algorithms and Architectures for Parallel Processing. ICA3PP 2018*, 2018.

[79] Docker Inc., “Scheduler design,” 15 July 2017. [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/scheduler.md. [Accessed 10 April 2021].

[80] Docker Inc., “SwarmKit,” 27 April 2017. [Online]. Available: https://github.com/docker/swarmkit/blob/766c68f98957013be854cba423bcb36c1b17802e/README.md#features. [Accessed 10 April 2021].

[81] Docker Inc., “Topology aware scheduling,” 16 March 2017. [Online]. Available: https://github.com/docker/swarmkit/blob/master/design/topology.md. [Accessed 10 April 2021].

[82] The Kubernetes Authors, “Kubernetes Scheduler,” 5 August 2021. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/. [Accessed 2 September 2021].

[83] M. Burillo, “Building a Kubernetes Scheduler using Custom Metrics,” 2018. [Online]. Available: https://static.sched.com/hosted_files/kccnceu18/0e/KubeConEU2018_custom_kubernetes_scheduler.pdf. [Accessed 1 May 2021].

[84] A. Chen and D. Tornow, “The Kubernetes Scheduler,” 27 December 2018. [Online]. Available: https://dominik-tornow.medium.com/the-kubernetes-scheduler-cd429abac02f. [Accessed 16 May 2021].

[85] R. Soboi, “A Deep Dive into Kubernetes Scheduling,” 30 November 2020. [Online]. Available: https://thenewstack.io/a-deep-dive-into-kubernetes-scheduling/. [Accessed 25 April 2021].

[86] The Kubernetes Authors, “Scheduling Framework,” 12 February 2021. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/. [Accessed 26 April 2021].

[87] The Kubernetes Authors, “Scheduling Policies,” 8 July 2021. [Online]. Available: https://kubernetes.io/docs/reference/scheduling/policies/. [Accessed 11 August 2021].

[88] The Kubernetes Authors, “Scheduler Algorithm in Kubernetes,” 31 March 2021. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler_algorithm.md. [Accessed 26 April 2021].

[89] The Kubernetes Authors, “Understanding the Kubernetes Scheduler,” 31 March 2021. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scheduling/scheduler.md. [Accessed 26 April 2021].

[90] Alibaba Cloud, “Getting Started with Kubernetes | Scheduling Process and Scheduler Algorithms,” 4 August 2020. [Online]. Available: https://alibaba-cloud.medium.com/getting-started-with-kubernetes-scheduling-process-and-scheduler-algorithms-847e660533f1. [Accessed 10 October 2021].

[91] The Kubernetes Authors, “Pod Topology Spread Constraints,” 9 September 2021. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/. [Accessed 10 October 2021].

[92] Docker Inc., “Compose file version 3 reference,” 13 September 2021. [Online]. Available: https://github.com/docker/docker.github.io/blob/master/compose/compose-file/compose-file-v3.md#resources. [Accessed 4 October 2021].

[93] Docker Inc., “service create,” 21 August 2021. [Online]. Available:

https://docs.docker.com/engine/reference/commandline/service_create/#specify-memory-requirements-and-constraints-for-a-service---reserve-memory-and---limit-memory. [Accessed 6 May 2021].

[94] Docker Inc., "Drain a node on the swarm," 6 August 2021. [Online]. Available: https://github.com/docker/docker.github.io/blob/master/engine/swarm/swarm-tutorial/drain-node.md. [Accessed 5 October 2021].

[95] The Kubernetes Authors, "Assigning Pods to Nodes," 19 April 2021. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/. [Accessed 28 April 2021].

[96] The Kubernetes Authors, "taint," 2020. [Online]. Available: https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#taint. [Accessed 25 April 2021].

[97] The Kubernetes Authors, "Taints and Tolerations," 21 April 2021. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/. [Accessed 25 April 2021].

[98] The Kubernetes Authors, "Scheduler extender," 29 January 2019. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/design-proposals/scheduling/scheduler_extender.md. [Accessed 26 April 2021].

[99] W. Huang, "Create a custom Kubernetes scheduler," 24 April 2019. [Online]. Available: https://developer.ibm.com/technologies/containers/articles/creating-a-custom-kube-scheduler/. [Accessed 26 April 2021].

[100] The Kubernetes Authors, "Scheduler Configuration," 22 August 2021. [Online]. Available: https://kubernetes.io/docs/reference/scheduling/config/. [Accessed 1 September 2021].

[101] L. Junjiang, "Talking about the implementation of Kubernetes Scheduling-Framework plug-in," [Online]. Available: https://www.programmersought.com/article/80654647561/. [Accessed 26 April 2021].

[102] CERN OpenStack Private Cloud Guide, "Scheduling Policies," [Online]. Available: https://clouddocs.web.cern.ch/containers/tutorials/scheduling.html. [Accessed 28 April 2021].

[103] Apache Software Foundation, "Kafka 3.0 Documentation," [Online]. Available: https://kafka.apache.org/documentation/#gettingStarted. [Accessed 14 October 2021].

[104] A. Pulido, "Navigating the Sea of Local Kubernetes Clusters," The Linux Foundation, 2020.

[105] Kubernetes Special Interest Groups, "Kubernetes Cluster Federation," [Online]. Available: https://github.com/kubernetes-sigs/kubefed. [Accessed 17 September 2021].

[106] K. Dongmin, M. Hanif, K. Eunsam, H. Sumi and L. Choonhwa, "TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform," *Applied Sciences,* vol. 9, no. 1, p. 191, 2019.

[107] Docker Inc., "Sample application," 17 August 2021. [Online]. Available: https://docs.docker.com/get-started/02_our_app/. [Accessed 13 September 2021].

Απόδοση Ξενόγλωσσων Όρων

absolute path: απόλυτη τοποθεσία

affinity feature: χαρακτηριστικό προτίμησης

Application Programming Interface (API): διεπαφή προγραμματισμού εφαρμογών

availability zone: ζώνη διαθεσιμότητας

backup: αποθήκευση πληροφορίας και ανάκτηση σε περίπτωση σφάλματος

bandwidth: εύρος δικτύου

batch job: εργασία που υποβάλλεται για εκτέλεση σε δέσμες

batch processing: ομαδοποίηση εργασιών από το λειτουργικό σύστημα και εκτέλεση χωρίς την παρέμβαση του χρήστη

big data analytics: επεξεργασία και ανάλυση μεγάλου όγκου δεδομένων

binary: δυαδικό αρχείο κώδικα

binding phase: φάση ανάθεσης

built-in functionality: ενσωματωμένη λειτουργία

client library: βιβλιοθήκη προγραμματιστικής διαχείρισης συστήματος

cloud computing: υπηρεσίες και υποδομές υπολογιστικού νέφους

cloud native application: εγγενής εφαρμογή υπολογιστικού νέφους

cluster: σύμπλεγμα υπολογιστικών κόμβων

Command-Line Interface (CLI): διεπαφή γραμμής εντολών

constructor function: συνάρτηση κατασκευαστή

container image: εικόνα του container

container orchestrators: μηχανισμοί ενορχήστρωσης εγγενών εφαρμογών υπολογιστικού νέφους

container runtime engine: μηχανισμός εκτέλεσης containers

containerization: μέθοδος ανάπτυξης και εκτέλεσης εφαρμογών με χρήση containers

cron job: εργασία που εκτελείται ανά προκαθορισμένα χρονικά διαστήματα

data centers: κεντρικές υποδομές

debugging: εντοπισμός σφαλμάτων

deep learning: βαθιά μηχανική μάθηση

development environment: περιβάλλον ανάπτυξης

downtime: διακοπή λειτουργίας

edge computing: πόροι στα άκρα του δικτύου

edge exchange sites: σημεία ανταλλαγής στα άκρα του δικτύου

environment variable: μεταβλητή περιβάλλοντος

environment variables: μεταβλητές περιβάλλοντος

event streaming: ροή συμβάντων

federated infrastructures: μοντέλο ενοποιημένης λειτουργίας υποδομών

filtering step: στάδιο φιλτραρίσματος

firewall: τείχος προστασίας

framework: πλαίσιο

Function-as-a-Service (FaaS): μοντέλο «Συνάρτησης ως Υπηρεσία»

garbage collector: μηχανισμός συλλογής και διαγραφής προγραμματιστικών αντικειμένων που δε χρησιμοποιούνται

gateway: πύλη δικτύου

guest operating system: εξωτερικό λειτουργικό σύστημα

high performance computing: υποδομές υπέρ-υψηλών επιδόσεων

hybrid cloud: υβριδικό περιβάλλον υπολογιστικού νέφους

hypervisor: λογισμικό υπέρ-επόπτη

image registry: αποθετήριο εικόνων

in place operation: λειτουργία που πραγματοποιείται επί τόπου

Infrastructure-as-a-Service (IaaS): μοντέλο «Υποδομής ως Υπηρεσία»

in-place algorithm: αλγόριθμος ο οποίος για τις μετατροπές των δεδομένων εισόδου δε χρησιμοποιεί επιπλέον βοηθητικές δομές δεδομένων

interface: διεπαφή

Internet of Things (IoT): Διαδίκτυο των Πραγμάτων

kernel: πυρήνας

key-value data store: δομή αποθήκευσης δεδομένων σε ζεύγη κλειδιού-τιμής

labels: ετικέτες

last mile network: πολύ απομακρυσμένο δίκτυο

load balancer: μηχανισμός ισοστάθμισης φορτίου

load balancing mechanism: μηχανισμός εξισορρόπησης φορτίου

log: αρχείο καταγραφής

logging: καταγραφή συμβάντων

loosely coupled: χαλαρά συζευγμένο

machine learning: μηχανική μάθηση

master: κεντρικός κόμβος υπολογισμού

max heap: σωρός μεγίστου

microservices: επιμέρους υπηρεσίες μιας εγγενούς εφαρμογής υπολογιστικού νέφους

module: λογική οντότητα κώδικα

multi cloud: υποδομές υπολογιστικού νέφους από πολλαπλούς παρόχους

multi-host networking: δικτύωση πολλαπλών κόμβων υπολογισμού

multi-node highly available cluster: σύμπλεγμα πολλαπλών κόμβων υψηλής διαθεσιμότητας

multi-tenant environment: υποδομές προσβάσιμες από πολλαπλές διαφορετικές ομάδες χρηστών

multi-tier application: εφαρμογή πολλαπλών επίπεδων με αλληλεξαρτήσεις

on-demand resources: διάθεση πόρων κατ' αίτημα των χρηστών

on-premise resources: ιδιόκτητοι πόροι χρηστών

open-source software: λογισμικό ανοιχτού κώδικα

operating system: λειτουργικό σύστημα

persistent volume: μόνιμη δομή αποθήκευσης

placeholder: μεταβλητή αντικατάστασης

placement constraint: περιορισμός τοποθέτησης

placement constraint: περιορισμός τοποθέτησης

placement preference: προτίμηση τοποθέτησης

Platform-as-a-Service (PaaS): μοντέλο «Πλατφόρμας ως Υπηρεσία»

plugin: πρόσθετο λογισμικό

port: θύρα

port: θύρα δικτύου

predicate: κατηγόρημα

priority function: συνάρτηση προτεραιότητας

production environment: παραγωγικό περιβάλλον

production environment: περιβάλλον εκτέλεσης

Proof of Concept (PoC): σύστημα για τον έλεγχο της δυνατότητας πρακτικής υλοποίησης μιας προτεινόμενης γενικότερης λύσης

query database: ερώτημα στη βάση δεδομένων

race condition: συνθήκη ανταγωνισμού

rack: στοίβα υπολογιστικών κόμβων

read-only field: πεδίο μόνο για ανάγνωση

regional network: περιφερειακό δίκτυο

registry: αποθετήριο

RESTful API: διεπαφή προγραμματισμού εφαρμογών βασισμένη σε περιγραφή πόρων

rolling updates: κυλιόμενες ενημερώσεις

rolling upgrade: σταδιακή αναβάθμιση

routing mesh: πλέγμα δρομολόγησης

scheduling framework: πλαίσιο χρονοδρομολόγησης

scheduling phase: φάση χρονοδρομολόγησης

scheduling profile: προφίλ χρονοδρομολόγησης

scoring step: στάδιο βαθμολόγησης

server: διακομιστής

service discovery mechanism: μηχανισμός εντοπισμού των εκτελούμενων εργασιών

service discovery: εντοπισμός υπηρεσίας

service networking: διασύνδεση υπηρεσιών

service: υπηρεσία

Software-as-a-Service (SaaS): μοντέλο «Λογισμικού ως Υπηρεσία»

source code: πηγαίος κώδικας

standalone containers: αυτόνομα εκτελούμενα containers

stateful application: εφαρμογή που διατηρεί την κατάστασή της

stateless application: εφαρμογή που δε διατηρεί την κατάστασή της

telco network: τηλεπικοινωνιακό δίκτυο

templating tool: εργαλείο προτύπων

testing environment: περιβάλλον ελέγχου

user group: ομάδα χρηστών

User Interface (UI): γραφική διεπαφή χρήστη

virtual machine: εικονικό μηχάνημα

virtual network: εικονικό δίκτυο

virtualization: εικονικοποίηση

web application: εφαρμογή ιστού

web browser: φυλλομετρητής

Ακρωνύμια

API: Application Programming Interface

CLI: Command-Line Interface

CNCF: Cloud Native Computing Foundation

CNI: Container Network Interface

CNM: Container Network Model του Docker

CRI: Container Runtime Interface

CSI: Common Storage Interface

DC/OS: Distributed Cloud Operating System

DNS: Domain Name System

FaaS: Function-as-a-Service

IaaS: Infrastructure-as-a-Service

IoT: Internet of Things

OCI: Open Container Initiative

PaaS: Platform-as-a-Service

PoC: Proof of Concept

SaaS: Software-as-a-Service

UI: User Interface

Κατάλογος Εικόνων

Κατάλογος Πινάκων