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

Ενεργειακά Συνειδητός και Αποδοτικός Χρονοδρομολογητής στο Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αθανάσιου Πυλιώτη

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΣΗΜΜΥ Ε.Μ.Π.

Συν-επιβλέπων: Αλέξανδρος Σούμπλης
Υποψήφιος Διδάκτορας ΣΗΜΜΥ Ε.Μ.Π.

Αθήνα, Ιούνιος 2025

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

Ενεργειακά Συνειδητός και Αποδοτικός Χρονοδρομολογητής στο Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αθανάσιου Πυλιώτη

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΣΗΜΜΥ Ε.Μ.Π.

Συν-επιβλέπων: Αλέξανδρος Σούμπλης
Υποψήφιος Διδάκτορας ΣΗΜΜΥ Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 20η Ιουνίου 2025.

(Ονόματα καθηγητών επιτροπής)

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
…… Παναγιώτης Τσανάκας Καθηγητής Ε.Μ.Π.	…… Δημήτριος Σούντρης Καθηγητής Ε.Μ.Π.	…… Χρήστος Παυλάτος Επίκουρος Καθηγητής Σχολή Ικάρων

Αθήνα, Ιούνιος 2025

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

Υπεύθυνη Δήλωση

Βεβαιώνω ότι είμαι συγγραφέας αυτής της διπλωματικής εργασίας, και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην διπλωματική εργασία. Επίσης έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης, βεβαιώνω ότι αυτή η διπλωματική εργασία προετοιμάστηκε από εμένα προσωπικά για τις απαιτήσεις του προγράμματος σπουδών του τμήματος Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών.

(Υπογραφή)

……

Πυλιώτης Αθανάσιος

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

Περίληψη

Η παρούσα διπλωματική εργασία εκπονήθηκε στα πλαίσια του προπτυχιακού κύκλου σπουδών της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Ο πυρήνας της εργασίας είναι η υλοποίηση ενός ενεργειακά αποδοτικού χρονοδρομολογητή (scheduler) για τη πλατφόρμα Kubernetes. Πιο συγκεκριμένα, στόχος είναι η αποδοτική κατανομή φόρτου εργασίας ώστε να ελαχιστοποιηθεί η κατανάλωση ενέργειας, λαμβάνοντας υπόψιν παράλληλα χαρακτηριστικά όπως η χρήση CPU, η διαθεσιμότητα πόρων, η σταθερότητα των κόμβων και οι πηγές ενέργειας.

Η υλοποίηση βασίστηκε σε ένα scheduler profile με συνδυασμό υπαρχόντων και νέων scoring plugins. Τα προτεινόμενα plugins αξιοποιούν τεχνικές όπως τον αλγόριθμο Best-First του Bin Packing, πολυκριτήρια βελτιστοποίηση με χρήση διανυσμάτων, ενεργειακά μοντέλα (όπως το EATSVM) και δυναμική διαχείριση ενέργειας μέσω του uptime των κόμβων. Η εργασία τεκμηριώνει το θεωρητικό υπόβαθρο, την υλοποίηση, και τον συνδυασμό των plugins μέσω προσομοιώσεων δικών μας δρομολογήσεων. Ο χρονοδρομολογητής που είχε την υψηλότερη ενεργειακή συνείδηση και βέλτιστη απόδοση είναι ο εξατομικευμένος μας χρονοδρομολογητής, με βάρη που ευνοούν τα plugins που βασίζονται σε ενεργειακά δεδομένα. Εν τέλει καταλήγει σε ένα παραμετροποιημένο χρονοδρομολογητή ικανό να λαμβάνει ενεργειακά συνειδητές και αποδοτικές αποφάσεις κατανομής φόρτου σε ένα Microk8s περιβάλλον cluster.

Στο τέλος της διπλωματικής εργασίας παρέχονται και κατάλληλα παραρτήματα. Στα παραρτήματα αυτά περιλαμβάνεται καθοδήγηση για την ορθή εγκατάσταση του συστήματος Kubernetes/Microk8s με τις απαραίτητες εντολές, καθώς και καθοδήγηση για την ορθή δημιουργία ενός καινούριου εξατομικευμένου χρονοδρομολογητή για μελλοντική, ξεχωριστή υλοποίηση.

Λέξεις Κλειδιά

Εξατομικευμένος Χρονοδρομολογητής, Kubernetes, Ενεργειακή Αποδοτικότητα, Βιώσιμη Υπολογιστική, Κατανομή Φόρτου Εργασίας, Δρομολόγηση Pods, Bin Packing, Πολυκριτήρια Βελτιστοποίηση, Διανυσματική ομοιότητα, Ενεργειακά μοντέλα, EATSVM, Δυναμική Διαχείριση ενέργειας (DPM), Microk8s, Plugins του Kubernetes

Abstract

This thesis was conducted as part of the undergraduate program of the School of Electrical and Computer Engineering at the National Technical University of Athens. The core objective of the work is the implementation of an energy-efficient scheduler for the Kubernetes platform. More specifically, the aim is to allocate workload efficiently to minimize energy consumption, while considering factors such as CPU usage, resource availability, node stability, and energy sources.

The implementation is based on a customized scheduler profile that combines existing and newly developed scoring plugins. The proposed plugins incorporate techniques such as the Best-Fit Bin Packing algorithm, multi-criteria optimization using resource vectors, energy estimation models (such as EATSVM), and dynamic power management based on node uptime. The thesis documents the theoretical background, the implementation process, and the combination of plugins through simulation experiments of custom scheduling scenarios. The scheduler that demonstrated the highest energy awareness and optimal performance is our customized scheduler, with weights favoring plugins based on energy-related data. Ultimately, it delivers a parameterized scheduler capable of making energy-aware and efficient workload placement decisions in a MicroK8s cluster environment.

At the conclusion of this thesis, relevant appendices are provided. These appendices include guidance for the proper installation of the Kubernetes/Microk8s system along with the necessary commands, as well as instructions for correctly creating a new custom scheduler for future, standalone implementations.

Keywords

Custom Scheduler, Kubernetes, Energy Efficiency, Energy Awareness, Sustainable Computing, Workload Allocation, Pod Scheduling, Bin Packing, Multi-criteria Optimisation, Vector Similarity, Energy Models, EATSVM, Dynamic Power Management (DPM), Microk8s, Kubernetes Plugins

Στους γονείς μου, την οικογένεια μου, και τους φίλους μου

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Καθηγητή Κ. Παναγιώτη Τσανάκα που μου έδωσε την ευκαιρία να διεκπεραιώσω αυτή τη διπλωματική, παρά τις όλες καθυστερήσεις που προέκυψαν. Επίσης, ένα ιδιαίτερο ευχαριστώ στον συνεπιβλέπων Αλέξανδρο Σούμπλη που ήταν παρών από τη πρώτη στιγμή και με καθοδηγούσε. Η καθοδήγηση και η κατανόηση που επέδειξε με βοήθησαν να συνεχίσω τη διπλωματική και να λάβω και μαθήματα για το πως να αντιμετωπίζω διάφορες καταστάσεις στη ζωή. Αλέξανδρε, ευχαριστώ πάρα πολύ που ήσουν εκεί για εμένα.

Μετά θα ήθελα να πω ένα μεγάλο ευχαριστώ στους γονείς μου. Ίσως κάποιες φορές να έχουμε τα πάνω και τα κάτω μας, αλλά η διαρκής τους συμπαράσταση και πίστη στις ικανότητες μου με βοήθησε να φτάσω στο σημείο που είμαι τώρα, ως διπλωματούχος της Σχολής ΗΜΜΥ του Εθνικού Μετσόβιου Πολυτεχνείου. Θα σας είμαι για πάντα ευγνώμων. Είμαι ευγνώμων και στα υπόλοιπα μέλη της οικογένειας μου που ήταν εκεί καθ'όλη τη χρονιά και με υποστήριζαν με τον τρόπο τους, ενεργά ή πιο σιωπηλά.

Τέλος, θα ήθελα να ευχαριστήσω ιδιαιτέρως τους φίλους μου, παλιούς και καινούριους, που ήταν εκεί καθ'όλη τη διάρκεια και με στήριξαν τόσο με ιδέες όσο και ψυχολογικά προκειμένου να φτάσουμε στο τέλος. Δεν θα αναφέρω ονόματα, αλλά χωρίς τη διαρκή στήριξη και παρουσία τους δεν θα ήμουν εδώ σήμερα και το εκτιμώ. Ευχαριστώ για όλες τις συμβουλές, όλη τη συμπαράσταση, όλες τις συζητήσεις και τις ώρες που μου κρατούσατε παρέα κατά την διεκπεραίωση αυτής της διπλωματικής.

Θα αφήσω και δύο φράσεις από το βιβλίο «Ο Αλχημιστής» τους Paolo Coelho που με άγγιξαν και τις οποίες έχω πάντα στο μυαλό μου:

«Υπάρχει μονάχα ένα πράγμα που μπορεί να κάνει ένα όνειρο αδύνατο να διεκπεραιωθεί: Ο φόβος της αποτυχίας.»

«Όταν θέλεις κάτι, όλο το σύμπαν συνωμοτεί για να το αποκτήσεις.»

Paolo Coelho, Ο Αλχημιστής.

Μακάρι όπως ο νεαρός Αλχημιστής του βιβλίου να ζήσουμε μια υπέροχη περιπέτεια μέχρι να φτάσουμε στον θησαυρό και να κάνουμε τα όνειρα μας πραγματικότητα.

Αθήνα, Ιούνιος 2025

Πυλιώτης Αθανάσιος

Περιεχόμενα

Κατάλογοι

Κατάλογος Εικόνων

Κατάλογος Πινάκων

Κατάλογος με κώδικες

1. Εισαγωγή

Η παρούσα διπλωματική εργασία εκπονήθηκε στα πλαίσια ολοκλήρωσης του προπτυχιακού κύκλου σπουδών της σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Η ανάγκη για καλύτερη διαχείριση των πόρων του Εθνικού Μετσόβιου Πολυτεχνείου προέκυψε από τις υψηλές απαιτήσεις των εφαρμογών στη σημερινή τεχνολογική καθημερινότητα, καθώς και την ύπαρξη πόρων οι οποίοι παραμένουν ανενεργοί παρά την απόκτηση τους. Συνεπώς, προτάθηκε να αξιοποιηθεί η καινοτόμα πλατφόρμα ανοιχτού κώδικα για container orchestration Kubernetes[1] (από το ελληνικό Κυβερνήτης) για την πιο αποδοτική παραχώρηση πόρων για τις εφαρμογές που χρειάζεται το Εθνικό Μετσόβιο Πολυτεχνείου για τη λειτουργία των ψηφιακών εφαρμογών του. Έτσι, εγκαθιστούμε τη πλατφόρμα και τη διατηρούμε όπως είναι, επεξεργαζόμενοι μονάχα το κομμάτι της βαθμολόγησης των nodes κατά τη διάρκεια της χρονοδρομολόγησης. Η πλειοψηφία από τα αρχεία που αναφέρονται στην εργασία μπορούν να βρεθούν στο Github repository[2].

1.1 Σκοπός της Διπλωματικής Εργασίας

Η παρούσα διπλωματική εργασία παρέχει καθοδήγηση για την εγκατάσταση του Kubernetes μέσω του lightweight Kubernetes distribution microk8s μαζί με έναν χρονοδρομολογητή που παρέχει πιο αποδοτική και ενεργειακά εφήμερη κατανομή πόρων. Η διπλωματική εστιάζει σε 2 βασικούς άξονες:

- **Δημιουργία πρόσθετων scoring plugins για τον χρονοδρομολογητή του Kubernetes:** Ο χρονοδρομολογητής του Kubernetes[3] έχει μερικά έτοιμα plugins για να βρίσκει το κατάλληλο node για κάθε pod. Σε αυτά θα προστεθούν δικά μας plugins με σκοπό την ελάχιστη κατανάλωση ενέργειας με την υψηλότερη αποδοτικότητα.
- **Δοκιμασία διαφορετικών configurations του χρονοδρομολογητή:** Πολλαπλές δοκιμές με διαφορετικά plugins και διαφορετικά βάρη για το binding. Η μέτρηση της απόδοσης θα γίνει με στατιστικά από τα metrics.k8s.io[4], Prometheus[5] και Kepler[6].
- **Πρόταση συγκεκριμένου τελικού χρονοδρομολογητή:** Τελικό προϊόν της διπλωματικής θα είναι ένας βελτιωμένος χρονοδρομολογητής μαζί με τα αρχεία που απαιτούνται και όλη τη διαδικασία εγκατάστασης του σε ένα σύστημα.

Η εργασία δεν υποστηρίζει απλώς τη θεωρητική βάση του έργου, αλλά παρέχει και πρακτικές υλοποιήσεις που μπορούν να εφαρμοστούν στα συστήματα του Εθνικού Μετσόβιου Πολυτεχνείου, ή οποιουδήποτε άλλου ιδρύματος, με τη κατάλληλη προεργασία.

1.2 Μεθοδολογική προσέγγιση της εργασίας

Η παρούσα διπλωματική εργασία υλοποιήθηκε ακολουθώντας μια μεθοδολογική προσέγγιση η οποία συνδυάζει βιβλιογραφική, θεωρητική, τεχνική και πειραματική έρευνα. Ο σκοπός της προσέγγισης είναι η εις βάθος μελέτη και αξιολόγηση ενεργειακά αποδοτικών τεχνικών στην δρομολόγηση Pods σε συστήματα Kubernetes.

- **Βιβλιογραφική και θεωρητική προσέγγιση:** Προτού ξεκινήσει η εργασία έγινε εκτενής ανασκόπηση της σχετικής βιβλιογραφίας, προκειμένου να εντοπιστούν οι σύγχρονες τεχνικές του ενεργειακά συνειδητού προγραμματισμού, σε επίπεδο αλγορίθμων κυρίως αλλά και σε επίπεδο υποδομών. Μελετήθηκαν τόσο αλγοριθμικοί μέθοδοι για ενεργειακά συνειδητή δρομολόγηση, όπως το Vector Bin Packing[7] και το EATSVM[8], όσο και χαμηλού επιπέδου τεχνικές εξοικονόμησης ενέργειας, όπως το DVFS[9], [10] και η αξιοποίηση στοιχείων για τους κόμβους, όπως η αξιοποίηση των ανανεώσιμων πηγών ενέργειας[11].
- **Τεχνική και σχεδιαστική προσέγγιση:** Αφού εντοπίστηκαν οι κατάλληλες τεχνικές, υλοποιήθηκαν τα plugins για τον χρονοδρομολογητή του Kubernetes, εντός του περιβάλλοντος του microk8s[12]. Η τεχνική εργασία περιλαμβάνει την επιλογή και ρύθμιση των κατάλληλων scoring plugins, με χρήση των κατάλληλων αλγορίθμων αξιολόγησης, καθώς και τη δημιουργία εναλλακτικών profiles. Παράλληλα, έγινε εύρεση κατάλληλων εργαλείων για μετρικές του συστήματος μας όσον αφορά πόρους και κατανάλωση ενέργειας.
- **Πειραματική προσέγγιση:** Κατά τη πειραματική προσέγγιση σχεδιάστηκε ένα σύνολο προσομοιώσεων διαφορετικής έντασης φόρτου. Τα Pods δημιουργήθηκαν με ελεγχόμενο προφίλ κατανάλωσης πόρων και δρομολογήθηκαν από τον προεπιλεγμένο και τον εξατομικευμένο χρονοδρομολογητή. Οι μετρικές αξιολόγησης συμπεριλάμβαναν την τελική δρομολόγηση των Pods ως προς τους κόμβους στους οποίους δρομολογήθηκαν, τη χρήση CPU, RAM[5] και τη κατανάλωση ενέργειας[6].
- **Ερευνητική διάσταση και προοπτικές:** Η εργασία επιδιώκει να απαντήσει κατά πόσο ένας ενεργειακά συνειδητός χρονοδρομολογητής μπορεί να συνεισφέρει στη μείωση της κατανάλωσης ενέργειας, χωρίς να υποβαθμίσει την απόδοση του συστήματος. Ταυτόχρονα, προτείνει μελλοντικές κατευθύνσεις έρευνας, όπως την εφαρμογή δυναμικής προσαρμογής των βαρών[15], [13], [14], μηχανισμού rescheduling αλλά και τεχνικών DVFS.

1.3 Δομή Διπλωματικής Εργασίας

Η διπλωματική οργανώνεται σε 5 κύρια κεφάλαια:

- **Κεφάλαιο 2: Τεχνολογίες που χρησιμοποιούμε**
  Γίνεται αναφορά σε τεχνολογίες τις οποίες χρειάστηκε να μελετήσουμε προκειμένου να στήσουμε το σύστημα, να παίρνουμε μετρήσεις και να φτιάξουμε τον χρονοδρομολογητή.
- **Κεφάλαιο 3: Θεωρητικό υπόβαθρο**
  Παρέχονται πληροφορίες σχετικά με το θεωρητικό υπόβαθρο της αποδοτικής χρονοδρομολόγησης. Έχει πληροφορίες σχετικά με bin packing αλγορίθμους, στρατηγικές δρομολόγησης που εξαρτώνται από την ενέργεια αλλά και έξυπνες τεχνικές δρομολόγησης.

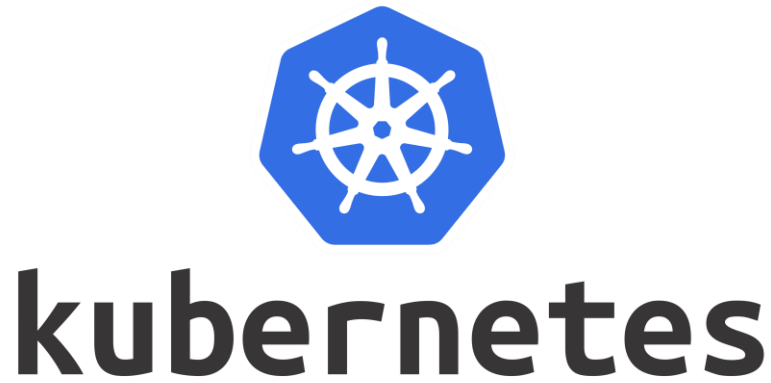
- **Κεφάλαιο 4: Υλοποίηση και Σχεδιασμός**
  Παρουσιάζεται η υλοποίηση, μαζί με παροχή κώδικα και επεξήγηση για τη λειτουργία του, καθώς και ο αρχιτεκτονικός σχεδιασμός του συστήματος.
- **Κεφάλαιο 5: Προσομοίωση και Αξιολόγηση**
  Παρουσιάζονται τα αποτελέσματα από διάφορες δοκιμές συνδυασμών διαφορετικών plugins με διαφορετικούς αλγορίθμους απόφασης. Συγκρίνουμε τα αποτελέσματα με δεδομένα από το Kubernetes ώστε να καταλήξουμε στο αποδοτικότερο ενεργειακά συνδυασμό.
- **Κεφάλαιο 6: Τελικά Συμπεράσματα**
  Συντάσσεται μια κριτική αξιολόγηση των αποτελεσμάτων και προτείνεται ο βέλτιστος χρονοδρομολογητής.

Κάθε κεφάλαιο αντιστοιχεί σε ένα βασικό κομμάτι κατά την εκπόνηση της διπλωματικής εργασίας. Στο τέλος της εργασίας σε αντίστοιχο παράρτημα θα παρατεθούν οδηγίες για την εγκατάσταση όλων των απαιτούμενων τεχνολογιών με βήματα για διευκόλυνση στη μελλοντική αξιοποίηση των αποτελεσμάτων της διπλωματικής εργασίας.

2. Τεχνολογίες που θα χρησιμοποιηθούν

Στην αρχή αυτού του κεφαλαίου θα παρουσιαστούν οι διάφορες τεχνολογίες που χρειάστηκε να αξιοποιήσουμε για να φτιάξουμε τη λύση μας. Σε αυτές συμπεριλαμβάνονται η πλατφόρμα Kubernetes (the Cloud Native Computing Foundation (CNCF[1]), n.d.) με γενικές πληροφορίες, αλλά και πιο συγκεκριμένα για τον χρονοδρομολογητή. Στη συνέχεια παρέχονται πληροφορίες σχετικά με το microk8s, ένα ελαφρύ σύστημα που εγκαθιστά γρήγορα τη πλατφόρμα Kubernetes. Τέλος θα παρουσιαστούν οι τεχνολογίες που μας παρέχουν μετρικές για το σύστημα μας, τα metrics.k8s.io, Prometheus και kepler.

2.1 Kubernetes



Εικόνα 2.1 Kubernetes Logo

2.1.1 Γενικές πληροφορίες

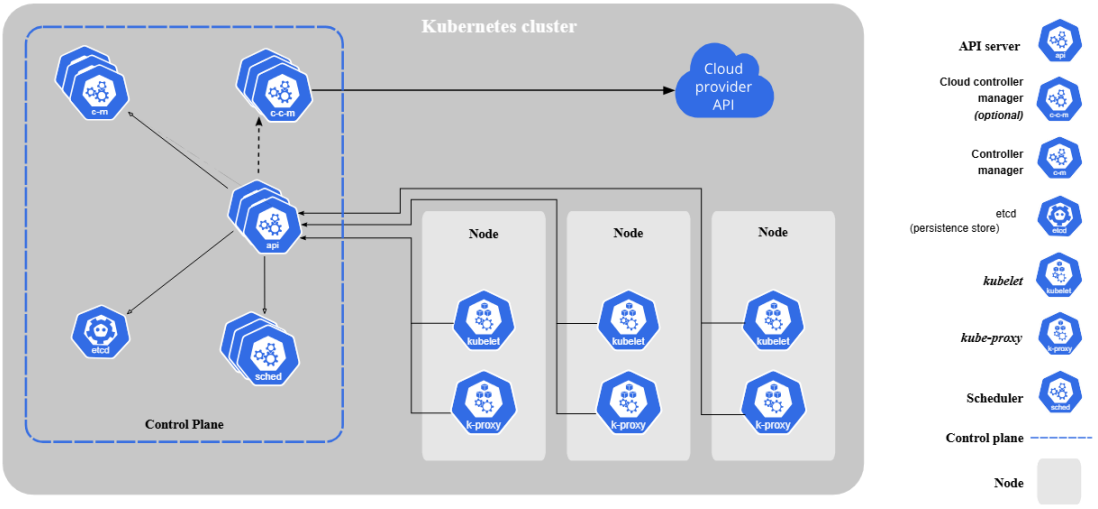
Η πλατφόρμα Kubernetes[1] είναι ένα σύστημα ανοιχτού κώδικα για αυτοματοποιημένο deployment, scaling και διαχείριση εφαρμογών σε containers. Μπορεί να τρέξει σε όλα τα λειτουργικά και είναι πολύ ευέλικτο ώστε να εξυπηρετεί σταθερά και εύκολα τις ανάγκες των χρηστών, ιδιαίτερα για τη χρονοδρομολόγηση υπηρεσιών και άλλων εργασιών. Τα βασικά στοιχεία αποτελούνται από τα στοιχεία του επιπέδου ελέγχου και τα στοιχεία των κόμβων. Παρακάτω φαίνονται πιο αναλυτικά μαζί με τις λειτουργίες τους.
Τα βασικά στοιχεία του επιπέδου ελέγχου είναι:

- kube-apiserver: εκθέτει το HTTP API σύστημα του Kubernetes.
- etcd: Συνεπής και υψηλής διαθεσιμότητας χώρος αποθήκευσης κλειδιών-τιμών για όλα τα δεδομένα διακομιστή API.
- kube-scheduler[3]: Ψάχνει για Pods που δεν είναι ακόμα συνδεδεμένα σε ένα κόμβο και αντιστοιχεί κάθε pod στο κατάλληλο κόμβο.
- kube-controller-manager: Τρέχει τους ελεγκτές για την υλοποίηση του API του Kubernetes.

Και σε κάθε κόμβο υπάρχουν τα παρακάτω στοιχεία:

- kubelet: Διασφαλίζει πως τα pods και τα containers τους είναι ενεργά και υγιή.
- kube-proxy: Διατηρεί τους κανόνες δικτύου στον κόμβο για να ενσωματώσει υπηρεσίες και εφαρμογές.
- container runtime: Λογισμικό υπεύθυνο για την λειτουργία των containers.

Επίσης το Kubernetes έχει και άλλες επιπρόσθετες λειτουργίες όπως το DNS resolution, Διαχείριση Πόρων Container αλλά και logging σε επίπεδο Cluster.



Εικόνα 2.2 Ένδειξη στοιχείων ενός Kubernetes Cluster

Κόμβοι (Nodes)

Ένας κόμβος (Node) μπορεί να είναι οποιοδήποτε ψηφιακό ή φυσικό μηχάνημα, αναλόγως τον cluster. Το επίπεδο ελέγχου διαχειρίζεται κάθε κόμβο. Κάθε κόμβος περιλαμβάνει τις απαραίτητες υπηρεσίες για τη λειτουργία των pods. Συνήθως κάθε cluster έχει αρκετούς κόμβους, οι οποίοι μπορούν να προστεθούν με δύο τρόπους, είτε αυτόματα από το kubelet είτε κατευθείαν από τον χρήστη. Το όνομα κάθε κόμβου είναι ένα DNS subdomain και δύο κόμβοι δεν μπορούν να έχουν το ίδιο όνομα. Τέλος, προκειμένου να γίνουν σημαντικές ανανεώσεις σε ένα κόμβο θα χρειαστεί να διαγραφεί εντελώς και να ξανά εισαχθεί στο cluster.

Pods

Ένα pod στο Kubernetes είναι η μικρότερη μονάδα deployment και η μικρότερη μονάδα που μπορεί να αναφερθεί σε υπολογισμούς. Είναι μια ομάδα από ένα ή περισσότερα containers που μοιράζονται χώρο αποθήκευσης και πόρους δικτύου. Τα pods δρομολογούνται μόνο μία φορά κατά την δημιουργία τους. Το Kubernetes μπορεί να λειτουργήσει το pod μόνο στο συγκεκριμένο κόμβο στον οποίο έχει ανατεθεί. Αν το pod χαλάσει ή παρουσιάσει πρόβλημα που δεν μπορεί να επιδιορθωθεί από το ίδιο Kubernetes, αυτό διαγράφεται. Αν ο κόμβος στον οποίο έχει ανατεθεί αποτύχει, τότε και τα pods που έχουν ανατεθεί στον κόμβο αυτόν διαγράφονται. Ο μοναδικός τρόπος με τον οποίο γίνεται η επαναδρομολόγηση είναι με ένα καινούριο pod με όλα τα χαρακτηριστικά εκτός το .metadata.uid , το οποίο είναι μοναδικό για κάθε pod.

2.1.2 Ο χρονοδρομολογητής (Scheduler) στο Kubernetes

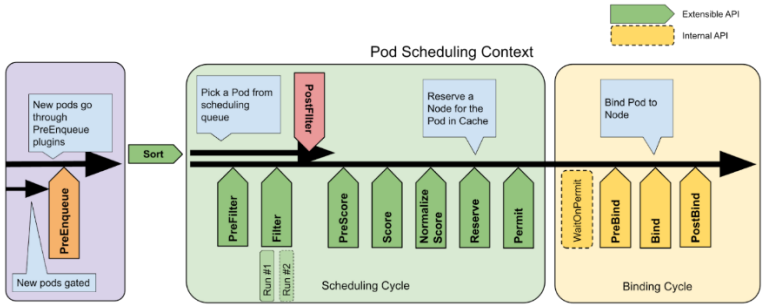
Ο χρονοδρομολογητής του Kubernetes, ή kube-scheduler, αποτελεί τμήμα του επιπέδου ελέγχου. Είναι σχεδιασμένος ώστε να μπορεί ο κάθε διαχειριστής να γράψει τα δικά του τμήματα για χρονοδρομολόγηση και να χρησιμοποιήσει αυτά αντί για το προκαθορισμένο. Παρακάτω θα αξιοποιηθεί αυτή η ικανότητα για Co-scheduling.

Για τη χρονοδρομολόγηση, ο kube-scheduler επιλέγει ένα ικανό κόμβο για να λειτουργήσει καινούρια ή ακόμα χωρίς κόμβο pods. Σε πρώτη φάση φιλτράρει τους κόμβους βάσει των απαιτήσεων στα containers των pods και αφαιρεί όσους δεν μπορούν να εξυπηρετήσουν τις απαιτήσεις. Διαφορετικά, είναι πιθανό σε pod που δημιουργούμε να ζητήσουμε συγκεκριμένο κόμβο, αλλά αυτό ούτε συνηθίζεται ούτε προτείνεται. Στη συνέχεια, ο kube-scheduler τρέχει συναρτήσεις που αξιολογούν τους εναπομείναντες κόμβους βάσει ορισμένων χαρακτηριστικών όπως η εγγύτητα των κόμβων ή η ύπαρξη των containers που χρειάζεται το pod. Τελικά το pod ανατίθεται στον κόμβο με την υψηλότερη βαθμολογία. Αν υπάρχει ισότητα στη βαθμολογία τότε η επιλογή γίνεται τυχαία.

Άλλη ενδιαφέρουσα ιδιότητα του kube-scheduler είναι το όριο κόμβων για βαθμολόγηση. Προκειμένου να βελτιωθεί η απόδοση κατά τη διάρκεια της χρονοδρομολόγησης, ο kube-scheduler μπορεί να σταματήσει να ψάχνει για περισσότερους κόμβους από ένα συγκεκριμένο πλήθος. Σε υπερμεγέθη clusters αυτό μειώνει κατά πολύ το χρόνο που χρειάζεται για τη δρομολόγηση ενός pod. Στη περίπτωση μας ωστόσο ενδιαφερόμαστε περισσότερο για μακροχρόνια, σταθερή λειτουργία, ενώ αυτή η ιδιότητα είναι αρκετά χρήσιμη για πιο σύντομες εργασίες.

Ο έλεγχος των κόμβων γίνεται με βάση τις αρχές του round robin ώστε να έχουν όλοι οι κόμβοι τη πιθανότητα να επιλεγούν για τη δρομολόγηση κάποιου pod. Αν οι κόμβοι είναι σε μια σειρά παραταγμένοι, τη πρώτη ο kube-scheduler θα ξεκινήσει από τον πρώτο κόμβο, και θα καταλήξει σε ένα κόμβο Υ που θα ελέγξει, αν αξιοποιήσουμε την παραπάνω ιδιότητα με το όριο κόμβων για βαθμολόγηση. Την επόμενη φορά που θα ανατεθεί pod για να τρέξει, ο kube-scheduler θα ξεκινήσει την αναζήτηση από τον επόμενο κόμβο του Υ και συνεχίζει από εκεί στην γραμμή. Αν οι κόμβοι είναι σε διαφορετικές ζώνες, τότε ο kube-schduler θα ελέγξει κόμβους από ποικίλες ζώνες για να βεβαιωθεί πως και οι κόμβοι από διαφορετικές ζώνες έχουν τη πιθανότητα να τους επιλέξουν.

Scheduling Framework[3]



Εικόνα 2.3 Scheduling Framework

Όπως φαίνεται στη παραπάνω εικόνα, στο scheduling framework υπάρχουν 2 κύκλοι: ο κύκλος της χρονοδρομολόγησης και ο κύκλος της δέσμευσης. Στον κύκλο της χρονοδρομολόγησης λαμβάνουν μέρος το φιλτράρισμα και η αξιολόγηση των κόμβων μέχρι την επιλογή και προσωρινή δέσμευση του πιο ιδανικού. Στη συνέχεια στον κύκλο δέσμευσης, δεσμεύεται επίσημα ο κόμβος για το pod που χρονοδρομολογείται.

Στη φάση του pre-score δημιουργείται μια κατάσταση κοινής χρήσης που θα αξιοποιηθεί από τα scoring plugins. Αν επιστρέψει σφάλμα, διακόπτεται όλος ο κύκλος χρονοδρομολόγησης.

Στη φάση του score, καλούνται όλα τα scoring plugins και αξιολογούν τους διάφορους κόμβους στους οποίους θα μπορούσε να λειτουργεί το pod. Τέλος, υπάρχει η φάση της κανονικοποίησης του score, μετά την οποία επιλέγεται ο κόμβος με την υψηλότερη συνολική βαθμολογία.

Προεπιλεγμένος kube-scheduler

Ο προεπιλεγμένος kube-scheduler λειτουργεί όπως αναφέρεται παραπάνω στο scheduling framework με τα plugins που θα παρουσιαστούν παρακάτω. Δίνονται λίγες πληροφορίες για το καθένα ώστε να υπάρχει μια βασική κατανόηση του πως λειτουργεί σε κάθε φάση της δρομολόγησης. Τα plugins μπορούν να λειτουργούν σε μία ή περισσότερες φάσεις της χρονοδρομολόγησης. Στη παρούσα διπλωματική θα εξετάσουμε πιο αναλυτικά τα plugins που έχουν να κάνουν με την αξιολόγηση των κόμβων, διατηρώντας όσα βοηθάνε στους στόχους μας και απενεργοποιώντας τις υπόλοιπες λειτουργίες.

- **ImageLocality:** Προτιμά κόμβους που έχουν ήδη τις εικόνες για τα containers που χρειάζεται το Pod.
  Σημεία επέκτασης: score
- **TaintToleration:** Υλοποιεί τα taints και τα tolerations.
  Σημεία επέκτασης: filter, preScore, score
- **NodeName:** Ελέγχει αν το nodeName του Pod ταιριάζει με το τρέχον node.
  Σημεία επέκτασης: filter
- **NodePorts:** Ελέγχει αν ένας κόμβος έχει διαθέσιμες πόρτες για τις πόρτες του Pod.
  Σημεία επέκτασης: preFilter, filter
- **NodeAffinity:** Υλοποιεί τα node selectors και την affinity των κόμβων.
  Σημεία επέκτασης: filter, score
- **PodTopologySpread:** Υλοποιεί την εξάπλωση Pod στην τοπολογία του cluster.
  Σημεία επέκτασης: preFilter, filter, preScore, score
- NodeUnschedulable: Αποκλείει κόμβους με .spec.unschedulable: true.
  Σημεία επέκτασης: filter
- **NodeResourcesFit:** Ελέγχει αν ο κόμβος διαθέτει όλους τους πόρους που ζητάει το Pod. Η βαθμολόγηση μπορεί να χρησιμοποιήσει μία από τις εξής στρατηγικές:
  - LeastAllocated (προεπιλογή)
  - MostAllocated
  - RequestedToCapacityRatio

  Σημεία επέκτασης: preFilter, filter, score
- **NodeResourcesBalancedAllocation:** Προτιμά κόμβους που εξισορροπούν καλύτερα τη χρήση πόρων όταν το Pod προγραμματιστεί σε αυτούς.
  Σημεία επέκτασης: score
- **VolumeBinding:** Ελέγχει αν ο κόμβος έχει ή μπορεί να κάνει bind τα ζητούμενα volumes.
  Σημεία επέκτασης: preFilter, filter, reserve, preBind, score
- **VolumeRestrictions:** Ελέγχει αν τα volumes που προσαρτώνται πληρούν περιορισμούς του provider.
  Σημεία επέκτασης: filter

- **VolumeZone:** Ελέγχει αν τα ζητούμενα volumes πληρούν τις απαιτήσεις ζώνης.
  Σημεία επέκτασης: filter
- **NodeVolumeLimits:** Ελέγχει αν τα όρια των volumes τύπου CSI μπορούν να ικανοποιηθούν στον κόμβο.
  Σημεία επέκτασης: filter
- **EBSLimits:** Ελέγχει αν τα όρια των AWS EBS volumes μπορούν να ικανοποιηθούν.
  Σημεία επέκτασης: filter
- **GCEPDLimits:** Ελέγχει αν τα όρια των GCP Persistent Disks μπορούν να ικανοποιηθούν.
  Σημεία επέκτασης: filter
- **AzureDiskLimits:** Ελέγχει αν τα όρια των Azure Disks μπορούν να ικανοποιηθούν.
  Σημεία επέκτασης: filter
- **InterPodAffinity:** Υλοποιεί affinity και anti-affinity μεταξύ Pods.
  Σημεία επέκτασης: preFilter, filter, preScore, score
- **PrioritySort:** Παρέχει την προεπιλεγμένη ταξινόμηση κατά προτεραιότητα.
  Σημεία επέκτασης: queueSort
- **DefaultBinder:** Παρέχει τον προεπιλεγμένο μηχανισμό binding.
  Σημεία επέκτασης: bind
- **DefaultPreemption:** Παρέχει τον προεπιλεγμένο μηχανισμό προεκτοπισμού (preemption).
  Σημεία επέκτασης: postFilter

Από τα παραπάνω plugins μονάχα 8 επηρεάζουν την φάση αξιολόγησης και θα επεξεργαστούμε αυτά, ενώ τα υπόλοιπα θα διατηρηθούν ως έχουν. Οι επιλογές σχετικά με την διατήρηση ή όχι των plugins θα γίνουν στο κεφάλαιο 4 που περιέχει τον σχεδιασμό του χρονοδρομολογητή μας.

2.2 microk8s

Το **microK8s**[12] είναι μια ελαφριά, πλήρως συμβατή διανομή του Kubernetes, σχεδιασμένη για εύκολη εγκατάσταση και χρήση σε διάφορα περιβάλλοντα, από προσωπικούς υπολογιστές έως συσκευές IoT και υποδομές αιχμής. Αναπτύχθηκε από την Canonical (την εταιρεία πίσω από το Ubuntu) και προσφέρει μια απλοποιημένη εμπειρία Kubernetes με ελάχιστες απαιτήσεις διαχείρισης .

Το microK8s είναι μια διανομή του Kubernetes που παρέχει:

- **Εύκολη εγκατάσταση**: Μπορεί να εγκατασταθεί με μία μόνο εντολή σε Linux, Windows και macOS.
- **Μικρό αποτύπωμα**: Καταναλώνει λιγότερους πόρους, καθιστώντας το ιδανικό για ανάπτυξη σε περιβάλλοντα με περιορισμένους πόρους, όπως Raspberry Pi ή άλλες συσκευές IoT.

- **Υψηλή διαθεσιμότητα**: Υποστηρίζει αυτόματη ανάκαμψη και ενημερώσεις over-the-air για αξιόπιστη λειτουργία σε παραγωγικά περιβάλλοντα.
- **Επεκτασιμότητα με πρόσθετα**: Παρέχει μια σειρά από πρόσθετα (addons) που μπορούν να ενεργοποιηθούν ή να απενεργοποιηθούν ανάλογα με τις ανάγκες, όπως το Istio, το Prometheus και το Knative .

Το microK8s προσφέρει μια απλή και αποτελεσματική λύση για όσους επιθυμούν να αξιοποιήσουν τις δυνατότητες του Kubernetes χωρίς την πολυπλοκότητα της παραδοσιακής εγκατάστασης και διαχείρισης.

Στην παρούσα εργασία θα αξιοποιηθεί το microk8s για γρήγορη και εύκολη εγκατάσταση του Kubernetes. Παράλληλα, θα εξαρτηθούμε σε αυτό και για όλα τα addons τα οποία προσφέρει, όπως για τις μετρικές των κόμβων και το Prometheus, αλλά και για το μικρό αποτύπωμα που έχει στα μηχανήματα που θα χρειαστεί να το εγκαταστήσουμε.

Χρονοδρομολογητής στο microk8s

Ο snap.microk8s.daemon-scheduler είναι η υπηρεσία που αντιπροσωπεύει τον **scheduler του Kubernetes** μέσα στο MicroK8s. Ο scheduler είναι υπεύθυνος για την ανάθεση Pods σε κόμβους του cluster όπως είδαμε παραπάνω, με βάση διάφορα κριτήρια: απαιτήσεις πόρων, ποιότητα υπηρεσίας (QoS), περιορισμούς υλικού ή λογισμικού, affinity/anti-affinity μεταξύ pods, τοποθεσία δεδομένων και χρονικά περιθώρια (deadlines).

Από την έκδοση **1.21 και μετά**, το MicroK8s ενοποίησε τον scheduler μέσα στο daemon-kubelite, ένα ελαφρύ εκτελέσιμο που περιλαμβάνει όλα τα βασικά Kubernetes components. Η συμπεριφορά του scheduler διαμορφώνεται μέσω του αρχείου ρυθμίσεων ${SNAP_DATA}/args/kube-scheduler, όπου ορίζονται τα απαραίτητα arguments σύμφωνα με τις ανάγκες του συστήματος. Αυτά τα ορίσματα είναι πλήρως τεκμηριωμένα στην επίσημη τεκμηρίωση του Kubernetes και επιτρέπουν την ευέλικτη παραμετροποίηση του προγραμματισμού των pods.

2.3 Μετρικές: metrics.k8s.io, Prometheus και Kepler

Προκειμένου να ελέγξουμε την απόδοση του συστήματος θα χρειαστεί να αξιοποιήσουμε συστήματα παρακολούθησης μετρήσεων. Το Kubernetes και το microk8s μας παρέχουν αρκετές επιλογές, με αυτές που αξιοποιήθηκαν να παρουσιάζονται περιληπτικά παρακάτω.

Prometheus

Το Prometheus[5] είναι ένα ισχυρό σύστημα παρακολούθησης και συλλογής μετρήσεων (monitoring & metrics collection) που χρησιμοποιείται ευρέως στο οικοσύστημα του Kubernetes. Συλλέγει δεδομένα από endpoints μέσω HTTP (pull-based μοντέλο), τα αποθηκεύει σε time-series βάση, και επιτρέπει πολύπλοκα ερωτήματα μέσω της γλώσσας PromQL. Στο Kubernetes, μπορεί να χρησιμοποιηθεί για την παρακολούθηση pods, nodes,

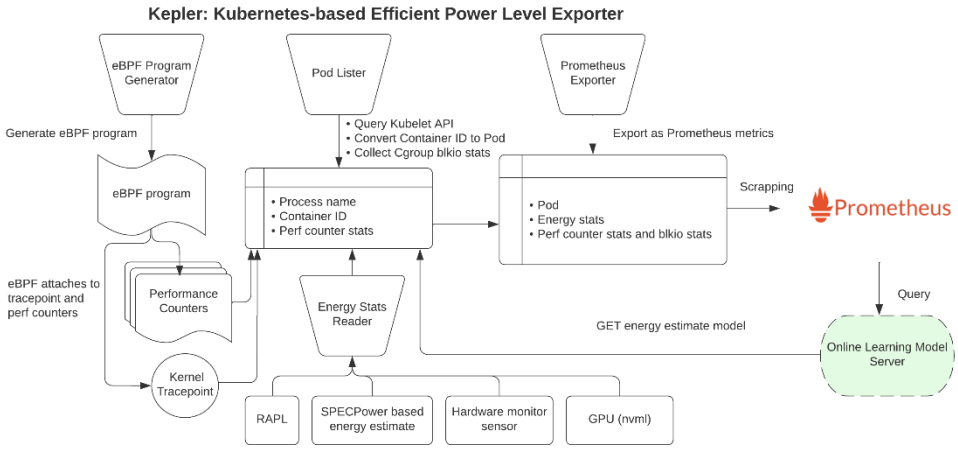
workloads, ακόμα και του ίδιου του scheduler ή custom metrics από εφαρμογές. Συνδυάζεται συχνά με το Grafana για οπτικοποίηση.

metrics.k8s.io API

Το metrics.k8s.io[4] είναι ένα standard Kubernetes API που χρησιμοποιείται για την έκθεση βασικών μετρήσεων χρήσης πόρων από Pods και Nodes, όπως CPU και μνήμη. Παρέχεται μέσω ενός add-on που ονομάζεται Metrics Server και αποτελεί τη βάση για εργαλεία όπως το kubectl top. Δεν αποθηκεύει ιστορικά δεδομένα — παρέχει μόνο τρέχουσες/στιγμιαίες τιμές και χρησιμοποιείται κυρίως για autoscaling (HPA) και debugging.

Kepler (Kubernetes-based Efficient Power Level Exporter)

Το Kepler[6] είναι ένα νεότερο open-source εργαλείο που αναπτύχθηκε από την κοινότητα CNCF για τη μέτρηση και έκθεση ενεργειακής κατανάλωσης ανά workload σε Kubernetes. Χρησιμοποιεί δεδομένα από το hardware (π.χ. RAPL, eBPF) και τα συνδυάζει με πληροφορίες από το cgroup του Linux για να υπολογίσει εκτιμώμενη ή πραγματική κατανάλωση ενέργειας από Pods, Containers και Nodes. Όταν αυτές οι μετρήσεις δεν είναι άμεσα διαθέσιμες, το Kepler χρησιμοποιεί εκπαιδευμένα μοντέλα τα οποία βασίζονται κυρίως στη χρήση CPU, μνήμης και I/O για να εκτιμήσει την ενεργειακή κατανάλωση. Τα μοντέλα έχουν εκπαιδευτεί εκ των προτέρων με διαφορετικά δεδομένα και πόρους, επιτρέποντας στον KEPLER να παρέχει ενεργειακές μετρικές ακόμα και σε συστήματα χωρίς ενσωματωμένους μετρητές ισχύος, όπως το δικό μας. Είναι πολύ χρήσιμο για ενεργειακό scheduling, βελτιστοποίηση απόδοσης/ενέργειας και περιβαλλοντική παρακολούθηση.



Εικόνα 2.4 Σχεδιασμός του KEPLER

3. Θεωρητικό Υπόβαθρο

Σε αυτό το κεφάλαιο θα παρουσιαστεί το θεωρητικό υπόβαθρο που αιτιολογεί τις αποφάσεις που πάρθηκαν αργότερα σχετικά με την υλοποίηση. Παρουσιάζεται το Bin packing πρόβλημα, μαζί με του κλασικούς online αλγορίθμους που το επιλύουν, τόσο σε μονοδιάστατο όσο και σε πολυδιάστατο επίπεδο. Μετά θα επικεντρωθούμε σε στρατηγικές για ενεργειακά ενήμερη δρομολόγηση αλλά και τεχνικές έξυπνης δρομολόγησης. Τέλος, θα υπάρξουν τα συμπεράσματα καθώς και οι αποφάσεις σχετικά με την υλοποίηση.

3.1 Τεχνικές Bin Packing

Το πρόβλημα του Bin Packing[16] έχει ως στόχο, δεδομένων n αντικειμένων με διαφορετικά βάρη και δοχείων (bins) με χωρητικότητα c, να τοποθετηθούν τα αντικείμενα στα δοχεία έτσι ώστε να ελαχιστοποιηθεί ο συνολικός αριθμός των χρησιμοποιούμενων δοχείων. Το πρόβλημα είναι NP-δύσκολο, και επομένως η εύρεση της βέλτιστης λύσης απαιτεί εκθετικό χρόνο. Για το λόγο αυτό, εφαρμόζονται προσεγγιστικοί αλγόριθμοι, που αποδίδουν "καλές" λύσεις σε πολυωνυμικό χρόνο.

Η κατώτερη θεωρητική τιμή (Lower Bound) του αριθμού των απαιτούμενων bins είναι το ταβάνι της διαίρεσης του συνολικού βάρους όλων των αντικειμένων δια του c. Δηλαδή, κανένας αλγόριθμος δεν μπορεί να χρησιμοποιήσει λιγότερα από αυτά τα bins.

Η πιο απλή εκδοχή του προβλήματος αφορά τη μονοδιάστατη περίπτωση, όπου το μόνο κριτήριο είναι το βάρος. Σε αυτή την περίπτωση, μερικοί κλασικοί αλγόριθμοι είναι:

Κλασικοί Αλγόριθμοι Μονοδιάστατου Bin Packing

- **Next Fit (NF):** Το κάθε αντικείμενο προσπαθεί να χωρέσει στο ίδιο bin με το προηγούμενο. Αν δεν χωράει, ανοίγεται νέο bin. Απλός αλγόριθμος με πολυπλοκότητα O(n), αλλά μπορεί να οδηγήσει σε μεγάλη σπατάλη χώρου (χρησιμοποιεί μέχρι και 2M bins, όπου M είναι το βέλτιστο πλήθος).
- **First Fit (FF):** Το κάθε αντικείμενο τοποθετείται στο πρώτο bin που έχει αρκετό χώρο. Αν κανένα δεν χωράει, δημιουργείται νέο bin. Η πολυπλοκότητά του είναι $O(n^2)$, αλλά μπορεί να υλοποιηθεί με έξυπνες δομές ώστε να μειωθεί ο χρόνος. Δεν χρησιμοποιεί περισσότερα από 1.7M bins.
- **Best Fit (BF):** Επιλέγεται το bin που θα αφήσει το λιγότερο υπόλοιπο **χώρο** μετά την προσθήκη του αντικειμένου. Συνήθως οδηγεί σε καλύτερη συμπίεση. Χρησιμοποιεί το πολύ 1.7M bins.
- **Worst Fit (WF):** Επιλέγει το bin που έχει τον περισσότερο διαθέσιμο χώρο, ώστε να κατανέμει πιο ομοιόμορφα τα φορτία. Έχει παρόμοια συμπεριφορά με το NF και χρήση μέχρι 2M–2 bins.

Στην offline εκδοχή του προβλήματος, δηλαδή όταν είναι γνωστά όλα τα αντικείμενα εκ των προτέρων, μπορεί να εφαρμοστεί ο First Fit Decreasing (FFD): Τα αντικείμενα ταξινομούνται φθίνουσα ως προς το βάρος και τοποθετούνται χρησιμοποιώντας τον FF. Αυτός θεωρείται από τους πιο αποτελεσματικούς απλούς αλγορίθμους.

Στην παρούσα εργασία, καθώς τόσο ο First Fit όσο και ο Best Fit έχουν παρόμοια θεωρητικά όρια, αξιολογούνται και οι δύο ώστε να αναδειχθεί ποιος προσφέρει καλύτερη κατανομή φόρτου σε πρακτικές συνθήκες. Δεν μπορεί να χρησιμοποιηθεί κάποιος offline αλγόριθμος λόγω της ανάγκης για διαρκή χρονοδρομολόγηση σε μεγάλο διάστημα, συνεπώς δεν θα μπορούμε να γνωρίζουμε όλα τα pods που θα χρειάζονται δρομολόγηση. Ωστόσο, στη περίπτωση που εφαρμοστούν τεχνικές rescheduling και έχουμε γνώση όλων των pods θα ήταν ενδιαφέρον αν, παρά την σειριακή ανάθεση pod σε κόμβο, ήταν δυνατό να τους αναθέτει με την βέλτιστη σειρά.

3.1.1 Διανυσματικό Bin Packing (Vector Bin Packing)

Στο Vector Bin Packing[7] (VBP), το πρόβλημα γενικεύεται σε πολλαπλές διαστάσεις. Αντί να θεωρούμε μόνο το "βάρος" ως ένα αριθμό, κάθε αντικείμενο περιγράφεται ως ένα διάνυσμα σε d διαστάσεις, π.χ. (CPU, RAM, Ενέργεια). Κάθε bin έχει αντίστοιχα όρια ανά διάσταση. Η τοποθέτηση ενός αντικειμένου είναι εφικτή μόνο αν σε κάθε διάσταση η συνολική χρήση δεν υπερβαίνει την χωρητικότητα. Αυτό δεν θα παραβιαστεί καθώς θα έχει προηγηθεί ο έλεγχος από τον κύκλο φιλτραρίσματος του χρονοδρομολογητή.

Η μοντελοποίηση αυτή ταιριάζει ιδανικά σε σύγχρονα υπολογιστικά συστήματα όπως το Kubernetes, όπου κάθε workload έχει απαιτήσεις σε πολλούς πόρους ταυτόχρονα.

Στο paper αναλύουν προσεκτικά πολλές παραλλαγές αλγορίθμων VBP και καταλήγουν σε χρήσιμες ευρετικές:

- **Norm-based heuristics:** Επιλογή bin με βάση το άθροισμα ή το μέγιστο της υπολειπόμενης χωρητικότητας.
- **Dot Product heuristics:** Επιλογή bin που μεγιστοποιεί το εσωτερικό γινόμενο με το διάνυσμα των απαιτήσεων (αντιπροσωπεύει την "καταλληλότητα").
- **Cosine Similarity heuristics:** Χρήση της συνημίτονου γωνίας μεταξύ διανυσμάτων για πιο κανονικοποιημένη αντιστοίχιση, ειδικά όταν οι πόροι είναι ανομοιογενείς.

Αυτές οι τεχνικές μπορούν να προσαρμοστούν στον προγραμματισμό plugins στο Kubernetes, μετατρέποντας κάθε plugin (π.χ., CPUFit, EnergyAware, BalancedAllocation) σε μία διάσταση του διανύσματος. Η επιλογή του κόμβου τότε μπορεί να γίνεται με dot product ή cosine similarity — επιλέγοντας τον "πιο κοντινό" ή "πιο συμβατό" κόμβο για το pod. Θα αναλυθούν περισσότερο παρακάτω στις τεχνικές έξυπνης δρομολόγησης.

3.2 Energy-aware Προγραμματισμός

Καθώς η κατανάλωση ενέργειας αποτελεί κρίσιμο ζήτημα στα σύγχρονα υπολογιστικά συστήματα, έχουν προταθεί διάφορες στρατηγικές που στοχεύουν στην ελαχιστοποίηση της ενεργειακής σπατάλης και τη μεγιστοποίηση της απόδοσης ανά Watt. Στο πλαίσιο του Kubernetes, αυτές οι στρατηγικές μπορούν να υλοποιηθούν μέσω εξειδικευμένων plugins και ευρετικών προσεγγίσεων κατά τον προγραμματισμό των workloads.

3.2.1 Energy-aware Best-Fit Heuristic

Μία από τις βασικές στρατηγικές που εξετάζονται είναι η υλοποίηση ενός ενεργειακά ενήμερου Best-Fit[17] αλγορίθμου ως scoring plugin. Ο στόχος του είναι να:

- Δίνει προτεραιότητα σε ήδη ενεργά (awake) nodes,
- Ελαχιστοποιεί τον αριθμό των ενεργών κόμβων,
- Βελτιστοποιεί τη χρήση της ενέργειας.

Η πιο απλή προσέγγιση υπολογισμού του score βασίζεται στη χρήση CPU:

$$Score = 100 * \frac{usedCPU}{totalCPU}$$

Όσο υψηλότερη είναι η χρήση CPU, τόσο πιο επιθυμητός θεωρείται ο κόμβος. Με αυτόν τον τρόπο, ευνοούνται οι "φορτωμένοι" κόμβοι, επιτυγχάνοντας συγκέντρωση φορτίου και αποφυγή ενεργοποίησης επιπλέον κόμβων.

Για πιο ακριβή προσέγγιση, μπορούμε να μοντελοποιήσουμε την στατική (βάση) και δυναμική κατανάλωση ενέργειας:

```
baseEnergy := 50.0 // Watts στο idle
dynamicEnergy := 100.0 * cpuUtil
totalEnergy := baseEnergy + dynamicEnergy
energyPerCore := totalEnergy / (usedCPU + 1)
score := 100 - normalize(energyPerCore)
```

Κώδικας 3.1 Θεωρητική προσομοίωση ενεργειακής κατανάλωσης

Η ιδέα είναι να ευνοούνται οι κόμβοι που καταναλώνουν λιγότερη ενέργεια ανά πυρήνα, αφού προσμετρηθεί το φορτίο. Σε επεκτάσεις που περιλαμβάνουν εναλλακτικές πηγές ενέργειας (όπως ανανεώσιμες)[18], [19], το score μπορεί να προσαρμοστεί:

$$Score = (cpuUtil * greenFactor) - (carbonPenalty * carbonIntensity)$$

Για τους σκοπούς της παρούσας εργασίας, θα κάνουμε παραδοχές σχετικά με τις παραπάνω τιμές ώστε να τις αξιολογήσουμε. Θα χρησιμοποιήσουμε μονάχα τις τιμές με το greenFactor και το carbonPenalty ώστε να έχουμε πιο ρεαλιστικές τιμές.

3.2.2 Απλοποιημένο EATSVM Μοντέλο

Σύμφωνα με το EATSVM[8] (Energy-Aware Task Scheduling on Virtual Machines), το οποίο σχεδιάστηκε για σύντομα και παράλληλα workloads, το ενεργειακό κόστος ανά κόμβο υπολογίζεται με βάση την τρέχουσα χρήση CPU και την ταχύτητα ρολογιού (clock speed):

Θεωρούμε πως:

- U_i: τρέχουσα χρήση CPU στον κόμβο i,
- ΔU_j: χρήση CPU του workload j,
- CS_i: ταχύτητα CPU του κόμβου i.

Τότε, η συνολική ισχύς γίνεται:

$$P_{total_i} = P(U_i + \Delta U_j, CS_i)$$

Για την απλοποιημένη έκδοση θα θεωρήσουμε τον παρακάτω κώδικα που υπολογίζει το Power, βάσει των παραπάνω:

$$P_i = C_i * U_i$$

Όπου P_i η συνολική ισχύς που καταναλώνεται, C_i η ταχύτητα του ρολογιού σε GHz και U_i η χρήση της CPU αν δρομολογηθεί το Pod σε αυτό το Node.

Η εργασία EATSVM συγκρίθηκε με τον αλγόριθμο ECTC στην εργασία, που χρησιμοποιείται σε δίκτυα αισθητήρων. Δυστυχώς δεν έχουμε πρόσβαση σε paper σχετικά με τον ECTC. Αν και το EATSVM στοχεύει σε workloads μικρής διάρκειας, μπορούμε να απλοποιήσουμε τη λογική του για σταθερά workloads, βασισμένοι στο utilization.

3.2.3 Τεχνικές Ευρετικής και Πολυκριτηριακής Βελτιστοποίησης στον Ενεργειακό Προγραμματισμό

Ο ενεργειακά ενήμερος προγραμματισμός εξελίσσεται συνεχώς με στόχο τη βελτιστοποίηση της κατανάλωσης ενέργειας χωρίς σημαντικές επιπτώσεις στην απόδοση. Πέρα από απλούς αλγορίθμους όπως το Best-Fit, χρησιμοποιούνται πιο πολύπλοκες ευρετικές και πολυκριτηριακές τεχνικές.

Energy-aware Greedy Scheduling

Απλή και γρήγορη μέθοδος που επιλέγει το node που:

- Είναι ήδη ενεργό

- Έχει την υψηλότερη αξιοποίηση CPU (άρα λειτουργεί κοντά στο κρίσιμο σημείο ενεργειακής απόδοσης)
- Αποφεύγει την αφύπνιση νέων κόμβων

Μπορεί να χρησιμοποιηθεί με διαφορετικά κριτήρια: CPU, RAM, disk I/O, GPU utilization, ή ακόμη και συνδυαστικά.
Η χρήση ενός σταθμισμένου score όπως:

$$Score = w_1 * CPU + w_2 * RAM + w_3 * EnergyCost$$

δίνει τη δυνατότητα προσαρμογής ανάλογα με τις απαιτήσεις.

Energy Delay Product (EDP) και Επεκτάσεις

Οι μετρικές τύπου EDP[10] επιτρέπουν συζυγή ανάλυση κατανάλωσης και απόδοσης. Ορίζονται ως:

- **EDP**: $EDP = Energy * time^w$ $(w = 1,2,3)$

Χρησιμοποιείται όταν ο χρόνος εκτέλεσης είναι κρίσιμος.

- **EDS**: $EDS = \alpha * Energy * \beta * Time$
- **EDD** (Euclidean Distance Delay): $EDD = \sqrt{(Energy^2 + (\beta * Time)^2)}$

Αυτές μπορούν να ενσωματωθούν σε score plugins ως πολυκριτηριακή συνάρτηση κόστους, ώστε να αξιολογούνται ταυτόχρονα η απόδοση και η ενεργειακή συμπεριφορά. Εφόσον ο χρόνος για εμάς δεν είναι ιδιαιτέρως χρήσιμος, ενδεχομένως να μην αξιοποιηθεί στην παρούσα έκδοση.

Knapsack-based Energy Scheduling

Το πρόβλημα του scheduling μπορεί να μοντελοποιηθεί ως πρόβλημα σακιδίου[20] (0-1 Knapsack), όπου:

- Κάθε workload έχει "βάρος" = ενεργειακό κόστος
- Κάθε κόμβος έχει "χωρητικότητα" = διαθέσιμο ενεργειακό περιθώριο
- Ο στόχος είναι να μεγιστοποιηθεί η "αξία" (π.χ. CPU usage, SLA value) υπό ενεργειακό περιορισμό

Η τεχνική αυτή είναι χρήσιμη για εφαρμογές με ενεργειακό budget ή για off-peak placement όταν η ενέργεια είναι φθηνότερη.

3.2.4 Τεχνικές Χαμηλού Επιπέδου Εξοικονόμησης Ενέργειας

Ορισμένες τεχνικές που εφαρμόζονται σε επίπεδο υλικού ή υποδομής έχουν άμεσο αντίκτυπο στην ενεργειακή συμπεριφορά. Μπορούν να συνδυαστούν ή να προσομοιωθούν στα score plugins ενός χρονοδρομολογητή.

DVFS (Dynamic Voltage and Frequency Scaling)

Η τεχνική DVFS[9] ρυθμίζει δυναμικά την τάση και τη συχνότητα λειτουργίας της CPU/GPU, μειώνοντας σημαντικά την κατανάλωση ενέργειας. Όμως, οδηγεί σε μικρότερη υπολογιστική ισχύ.

- Σε scheduling, μπορούμε να ενσωματώσουμε την καμπύλη:

$Power \propto V^2 * f$

- Η μείωση συχνότητας επιδρά με γραμμικό τρόπο στον χρόνο, αλλά εκθετικά στην κατανάλωση.

Αν βρεθεί τρόπος για δυναμική αλλαγή τάσης και συχνότητας CPU/GPU μια ενδιαφέρουσα υλοποίηση θα ήταν η αδρανοποίηση των κόμβων που δεν αξιοποιούνται.

DPM (Dynamic Power Management)

Η τεχνική DPM[21] αυτή θέτει αδρανείς κόμβους ή συσκευές σε κατάσταση χαμηλής ισχύος ή πλήρους απενεργοποίησης. Σε scheduling:

- Μπορεί να δοθεί ποινή (penalty) σε κόμβους που απαιτούν “wake-up”
- Ευνοούνται ενεργοί κόμβοι για τοποθέτηση pods

Σε συνδυασμό με το DVFS θα μας επιτρέψουν υψηλότερο ενεργειακό έλεγχο στο σύστημα μας. Μια πιθανή υλοποίηση θα μπορούσε να είναι με τα μετα-δεδομένα των κόμβων.

Power Capping

Το Power Capping[22] αφορά τον περιορισμό της μέγιστης επιτρεπόμενης κατανάλωσης ενέργειας ανά κόμβο. Εάν τοποθέτηση ενός pod οδηγεί σε υπέρβαση, ο κόμβος αποκλείεται ή δέχεται ποινή.

- Ιδιαίτερα χρήσιμο σε υπερθερμαινόμενα συστήματα ή υποδομές με συμβόλαια ισχύος.

Χρήσιμη τεχνική για την πραγματική ισχύ ενός κόμβου. Ειδικά σε clusters με παλαιότερα μηχανήματα θα είναι ιδιαίτερα χρήσιμο να γνωρίζουμε και λαμβάνουμε υπόψη για τα πρακτικά όρια του μηχανήματος και όχι τα θεωρητικά.

Πράσινες Μετρικές (REE, TGI, PpW)

Για να ενσωματώσουμε την έννοια της βιωσιμότητας και “πράσινης” τεχνολογίας[11]:

- **REE**: Relative Energy Efficiency
- **TGI**: Time Green Intensity — ποσοστό χρόνου με χρήση πράσινης ενέργειας
- **PpW**: Performance per Watt

Μπορούν να αξιοποιηθούν ως κριτήρια αξιολόγησης clusters και όχι μόνο nodes, ενισχύοντας την απόφαση στο ανώτερο επίπεδο, αλλά δεν είναι στα πλαίσια αυτής της εργασίας. Πιο πιθανό να εισαχθούν ως ετικέτες για το NodeAffinity.

3.3 Ευφυείς Τεχνικές Ενεργειακού Προγραμματισμού

Οι σύγχρονες τεχνικές ενεργειακά ενήμερου προγραμματισμού αξιοποιούν ευφυείς αλγορίθμους που προσαρμόζονται σε πολύπλοκες συνθήκες λειτουργίας. Στόχος είναι η αποδοτική κατανομή πόρων, η αποφυγή bottlenecks, και η ελαχιστοποίηση της κατανάλωσης ενέργειας, ακόμα και σε περιβάλλοντα με αβέβαιες ή μεταβαλλόμενες απαιτήσεις.

3.3.1 Ιστορική Τοποθέτηση (History-Based Placement)

Η τοποθέτηση με βάση την μέχρι τώρα συμπεριφορά αποτελεί μια τεχνική όπου το σύστημα "μαθαίνει" από προηγούμενες συμπεριφορές, και επιχειρεί να τοποθετεί παρόμοια workloads σε κόμβους που σίγουρα τα διαχειρίζονται αποδοτικά. Αυτό μπορεί να επιτευχθεί με:

- **Προφίλ workloads**: καταγραφή μέσης και μέγιστης χρήσης CPU, μνήμης, και χρόνου ζωής.
- **Προφίλ κόμβων**: παρακολούθηση της απόδοσης σε συνθήκες φόρτου (π.χ. evictions).
- **Καταγραφή παρεμβολών (interference history)**: μελέτη των περιπτώσεων όπου η συνύπαρξη workloads οδηγεί σε επιβράδυνση ή υπερφόρτωση.

Με την κατάλληλη συλλογή μετρικών (π.χ. Prometheus), μπορούν να δημιουργηθούν προφίλ κατηγοριοποίησης (π.χ. “ευαίσθητο node”, “Υπερθερμαίνεται εύκολα”) και να υλοποιηθεί ένας score-based μηχανισμός επιλογής. Είναι κατάλληλο για συστήματα με σταθερά ή επαναλαμβανόμενα workloads, όπως web servers. Ειδικά στο σύστημα μας θα βοηθούσε με πληροφορίες σχετικά με ενεργειακή κατανάλωση σε συνδυασμό με το NodeAffinity ή τα Taints και Tolerations.

3.3.2 Σταθμισμένη βαθμολόγηση με Lookahead

Η προσέγγιση αυτή συνδυάζει διαφορετικά κριτήρια προγραμματισμού (π.χ. χρήση CPU, ενεργειακό κόστος, ιστορική συμπεριφορά) με σταθμισμένες τιμές (weights). Το συνολικό score ενός κόμβου δίνεται από:

$$Score_{total} = \sum_{i} w_i * S_i$$

όπου:

- w_i: βάρος του κριτηρίου (π.χ. 0.5 για CPU, 0.3 για ενέργεια)

- S_i: μεμονωμένο score (π.χ. κανονικοποιημένη χρήση CPU)

Η τεχνική lookahead επεκτείνει το σύστημα προσθέτοντας προσομοίωση του "μέλλοντος": υπολογίζει πώς θα επηρεαστεί ο κόμβος αν το workload τοποθετηθεί εκεί. Αν προκαλεί bottleneck, η βαθμολογία προσαρμόζεται αρνητικά. Θα μπορούσε να παρουσιαστεί αυτό με τη προσθήκη ενός penalty.

Παρότι το lookahead είναι πιο χρήσιμο σε σενάρια μεταβαλλόμενων workloads, μπορεί να βοηθήσει και σε συστήματα μακράς διάρκειας, αποτρέποντας την υπερφόρτωση κόμβων που ήδη βρίσκονται κοντά στα όρια.

3.3.3 Ευφυείς Αλγόριθμοι Εξερεύνησης

Οι παρακάτω αλγόριθμοι είναι αρκετά πιο απαιτητικοί υπολογιστικά, αλλά αντιστοιχούν σε ιδιαιτέρως καινοτόμες ιδέες. Παρέχονται περισσότερο σαν ιδέες για μελλοντική υλοποίηση, καθώς είναι πιθανό να μην είναι δυνατό στο τωρινό σύστημα.

Hybrid Particle Swarm Optimization (H-PSO)

Το H-PSO[13], [14] βασίζεται στη θεωρία σμηνών (swarm intelligence), όπου κάθε "σωματίδιο" αντιπροσωπεύει μία πιθανή τοποθέτηση workloads και κινείται στον χώρο λύσεων ανάλογα με προσωπικές και συλλογικές εμπειρίες. Συνδυάζεται συχνά με τοπική βελτιστοποίηση ή greedy βήματα για βελτίωση σύγκλισης.

- Έχει αποδειχθεί ότι υπερτερεί του greedy scheduling έως και 45% σε εξοικονόμηση ενέργειας.
- Κατάλληλο για αργά μεταβαλλόμενα workload patterns.

Εξελικτικοί Αλγόριθμοι (Genetic, NSGA-II)

Οι εξελικτικοί αλγόριθμοι (GA)[15] λειτουργούν με βάση πληθυσμούς λύσεων που εξελίσσονται μέσω διασταύρωσης και μετάλλαξης. Οι πιο αποδοτικές λύσεις (τοποθετήσεις pods σε κόμβους) επιλέγονται για την επόμενη γενιά.

- Το NSGA-II επιτρέπει βελτιστοποίηση πολλών στόχων (π.χ., latency, energy).
- Χρήσιμο για παραμετροποίηση weights στο τελικό scoring plugin, π.χ., εκπαίδευση του αλγορίθμου εκτός παραγωγής.

Ενδεχομένως να μπορεί να αξιοποιηθεί σε συνδυασμό με τη σταθμισμένη βαθμολόγηση με lookahead ώστε να γίνονται ακόμα πιο αποδοτικά τα lookaheads.

3.3.4 Dot Product και Cosine Similarity Fit

Δύο μαθηματικά score plugins βασισμένα στη γεωμετρία των πόρων[7] είναι τα παρακάτω:

Dot Product Fit:

Θεωρούμε:

- Διάνυσμα workload: $W = [CPU_{req}, MEM_{req}]$
- Διαθέσιμοι πόροι κόμβου: $N = [CPU_{avail}, MEM_{avail}]$

Υπολογίζουμε:

$Score = W * N = \sum_i W_i * N_i$

Όσο μεγαλύτερο το εσωτερικό γινόμενο, τόσο καλύτερη η "ευθυγράμμιση" — άρα καλύτερη αξιοποίηση. Στη περίπτωση που θέλουμε ομοιότητα μεταξύ κόμβου και pod όσον αφορά τους πόρους θα ήταν χρήσιμο να αξιοποιηθεί.

Cosine Similarity Fit:

Μετράμε τη γωνία μεταξύ των διανυσμάτων:

$$\cos(\theta) = \frac{W * N}{|W| * |N|}$$

Όσο πιο κοντά στο 1 (δηλαδή γωνία ≈ 0°), τόσο πιο συμβατοί οι πόροι.

Αν και το cosine similarity προσφέρει καλύτερη "κανονικοποίηση", το dot product είναι πιο αποτελεσματικό στον best-fit ενεργειακό σχεδιασμό, καθώς προτιμά κόμβους με ελάχιστο κενό στους πόρους.

Μπορούν και τα 2 να αξιολογηθούν με διαφορετική σημαντικότητα. Επίσης, πέρα από τη CPU και τη μνήμη μπορούμε να χρησιμοποιήσουμε και ενεργειακές προδιαγραφές των κόμβων και του pod για τον υπολογισμό.

4. Υλοποίηση και Σχεδιασμός

Σε αυτό το κεφάλαιο θα παρουσιαστεί η υλοποίηση και ο σχεδιασμός του χρονοδρομολογητή μας. Στην αρχή θα παρουσιαστούν τα ήδη υπάρχοντα plugins αξιολόγησης που θα αξιοποιήσουμε στην εφαρμογή μας. Στη συνέχεια θα παρουσιαστούν περιληπτικά τα καινούρια plugins που θα υλοποιήσουμε, καθώς και ο τελικός χρονοδρομολογητής. Θα παρουσιαστούν τα plugins που σχεδιάστηκαν στα πλαίσια της διπλωματικής, με αιτιολόγηση για την επιλογή τους και παράθεση κώδικα.

4.1 Ο χρονοδρομολογητής και ο τελικός σχεδιασμός

Ενώ ο προεπιλεγμένος χρονοδρομολογητής κάνει εξαιρετική δουλειά για τις περισσότερες περιπτώσεις, το Kubernetes μας δίνει την ευελιξία να τον επεκτείνουμε με δικά μας plugins και προφίλ (profiles), ώστε να προσαρμόσουμε τη συμπεριφορά του στις μοναδικές ανάγκες του cluster μας. Αυτή η προσαρμογή μας επιτρέπει να εφαρμόσουμε προηγμένες στρατηγικές, όπως η ενεργειακή απόδοση ή η βελτιστοποίηση της χρήσης πόρων.

4.1.1 Επιλογή Προεπιλεγμένων Plugins για τον Χρονοδρομολογητή

Ο προεπιλεγμένος Kubernetes Scheduler περιλαμβάνει μια σειρά από plugins που καλύπτουν γενικές ανάγκες. Για να βελτιστοποιήσουμε τον χρονοδρομολογητή σύμφωνα με τους ειδικούς στόχους της διπλωματικής μας, θα επιλέξουμε προσεκτικά ποια από αυτά θα ενεργοποιήσουμε και ποια θα απενεργοποιήσουμε, συμπληρώνοντάς τα φυσικά με τα δικά μας plugins.

Θα επιτρέψουμε και θα χρησιμοποιήσουμε τα ακόλουθα 6 προεπιλεγμένα plugins:

1. **TaintToleration**
    - **Γιατί το επιτρέπουμε:** Αυτό το plugin είναι απαραίτητο για τη διαχείριση των **Taints και Tolerations** στους κόμβους. Στο πλαίσιο της βιωσιμότητας και της ενεργειακής απόδοσης, μπορούμε να αξιοποιήσουμε τα Taints με χρήσιμο και ενεργειακό τρόπο. Για παράδειγμα, μπορούμε να εισαγάγουμε taints όπως "low-energy-node", "legacy-hardware", ή "renewable-energy-source". Ενδεχομένως να μην εφαρμοστεί στη παρούσα υλοποίηση, αλλά αφήνεται για μεταγενέστερες.
2. **NodeAffinity**
    - **Γιατί το επιτρέπουμε:** Το NodeAffinity μας δίνει τη δυνατότητα να χρησιμοποιούμε labels στους κόμβους για να εκφράζουμε προτιμήσεις ή απαιτήσεις. Αυτό είναι εξαιρετικά χρήσιμο για τους στόχους μας, καθώς μπορούμε να "μαρκάρουμε" τους κόμβους βάσει κριτηρίων. Αυτά θα μπορούσαν να είναι: Πηγή ενέργειας, όπως energy-source=solar, energy-source=grid, Περιοχή/Δημοτικότητα όπως, region=europe-east,

region=datacenter-a, και Ειδικός εξοπλισμός όπως, hardware-gen=latest, power-mode=eco. Το διατηρούμε καθώς αν δεν υπάρχουν labels δεν μας επηρεάζει, αλλά μπορεί να χρησιμοποιηθεί σε μελλοντικές χρήσεις.

3. **ImageLocality**
    - **Γιατί το επιτρέπουμε:** Αν και δεν είναι άμεσα συνδεδεμένο με τους στόχους βιωσιμότητας ή ενεργειακής απόδοσης, συμβάλλει στην αποτελεσματικότητα και τη μείωση του χρόνου εκκίνησης των Pods. Μειώνοντας τον χρόνο που απαιτείται για το pull των images, μειώνεται και ο συνολικός "άεργος" χρόνος των κόμβων, συμβάλλοντας έμμεσα στην αποδοτικότητα του cluster. Το διατηρούμε ενεργό για τη γενική βελτίωση της απόδοσης, ακόμα κι αν δεν συμβάλει ιδιαιτέρως ενεργειακά.

4. **NodeResourcesBalancedAllocation**
    - **Γιατί το επιτρέπουμε:** Αυτό το plugin στην ουσία, προτιμά κόμβους όπου το Pod θα αυξήσει την CPU και τη μνήμη σε παρόμοιες αναλογίες, ώστε να διατηρηθεί μια ισορροπημένη κατανομή των πόρων. Αυτή η λογική είναι πολύ συμβατή με τον στόχο του "best-fit" και του "bin-packing" που επιδιώκουμε. Ενώ τα δικά μας plugins (όπως το ResourceVectorSimilarity, το Dot Product και το Cosine Similarity) εστιάζουν σε συγκεκριμένα "ταιριάσματα", το NodeResourcesBalancedAllocation παρέχει μια γενική κατεύθυνση για την αποφυγή μονομερούς υπερβολικής χρήσης του ενός πόρου, συμπληρώνοντας τη στρατηγική μας για βέλτιστη αξιοποίηση.

5. **VolumeBinding**
    - **Γιατί το επιτρέπουμε:** Το VolumeBinding είναι απαραίτητο μόνο στην περίπτωση που τα workloads μας χρησιμοποιούν **PersistentVolumes (PVs)** και **PersistentVolumeClaims (PVCs)**. Εφόσον δεν θα τα χρησιμοποιήσουμε, θα μπορούσε να απενεργοποιηθεί. Ωστόσο, αφού δεν θα επηρεάσει κάπως ενεργειακά το σύστημα μας, και υπάρχει το ενδεχόμενο να υπάρξει ανάγκη για PVs στο μέλλον, θα διατηρηθεί.

6. **InterPodAffinity**
    - **Γιατί το επιτρέπουμε:** Το InterPodAffinity και InterPodAntiAffinity χρησιμοποιούνται για να εκφράζουν προτιμήσεις ή απαιτήσεις σχετικά με την τοποθέτηση Pods σε σχέση με άλλα Pods. Θα μπορούσε σε μελλοντικές υλοποιήσεις να χρησιμοποιηθεί για τη διαχείριση "θορυβωδών" (noisy) ή "ευαίσθητων" (sensitive) workloads, αυτή τη στιγμή δεν αποτελεί βασικό στόχο της βελτιστοποίησής μας για βιωσιμότητα και χρήση πόρων. Επομένως, εκτός αν κριθεί απαραίτητο αργότερα θα το διατηρήσουμε για τη περίπτωση που κάποιος χρήστης επιθυμήσει να το χρησιμοποιήσει, π.χ. για παράδειγμα δρομολόγησης ίδιας ιστοσελίδας σε διαφορετικά μηχανήματα.

Προεπιλεγμένα Plugins που θα απενεργοποιήσουμε και γιατί:

1. **LeastAllocated**
    - **Γιατί το απενεργοποιούμε:** Το LeastAllocated βαθμολογεί υψηλότερα τους κόμβους που έχουν τη μεγαλύτερη ποσότητα ελεύθερων (least allocated) πόρων. Αυτό σημαίνει ότι τείνει να διασπείρει τα Pods σε όσο το δυνατόν περισσότερους κόμβους, αφήνοντας τους κόμβους λιγότερο φορτωμένους. Αυτή η συμπεριφορά είναι αντίθετη με τον κύριο στόχο της διπλωματικής, που είναι η πυκνή συσκευασία των Pods σε λιγότερους κόμβους με σκοπό την ελάχιστη κατανάλωση ενέργειας. Η πυκνή συσκευασία είναι κρίσιμη για τη μείωση της συνολικής ενεργειακής κατανάλωσης, καθώς επιτρέπει στους υπόλοιπους κόμβους να παραμένουν σε χαμηλή κατάσταση λειτουργίας ή να απενεργοποιούνται.
2. **PodTopologySpread**
    - **Γιατί το απενεργοποιούμε:** Το PodTopologySpread έχει ως στόχο να διασπείρει τα Pods σε διαφορετικές ζώνες διαθεσιμότητας, κόμβους ή άλλες τοπολογικές μονάδες, για να βελτιώσει την ανθεκτικότητα και τη διαθεσιμότητα των εφαρμογών. Ωστόσο, η λογική της διασποράς έρχεται σε ευθεία αντίθεση με τη στρατηγική της πυκνής συσκευασίας που επιδιώκουμε για την ενεργειακή απόδοση. Για τη βέλτιστη εκμετάλλευση θέλουμε να συγκεντρώνουμε τους φόρτους εργασίας και όχι να τους διασπείρουμε. (Θα μπορούσε βέβαια να είναι ενδιαφέρον να εξεταστούν οι επιπτώσεις του σε ένα σενάριο όπου η διαθεσιμότητα είναι υψηλότερη προτεραιότητα, και να υπολογίσουμε τη συνολική ενεργειακή κατανάλωση σε αυτές τις περιπτώσεις).

Με αυτή την επιλογή, ο χρονοδρομολογητής θα είναι προσαρμοσμένος ώστε να προωθεί τη συγκέντρωση των φόρτων εργασίας (πυκνή συσκευασία), να εκμεταλλεύεται τα ενεργειακά χαρακτηριστικά των κόμβων μέσω labels και taints, και να λαμβάνει υπόψη την ισορροπία των πόρων και την σταθερότητα των κόμβων, όλα προς τον γενικότερο στόχο της ενεργειακής απόδοσης και βιωσιμότητας. Ακόμα, διατηρούνται οι λειτουργίες για PVs και InterPodAffinity ώστε να μπορούν να εφαρμοστούν εύκολα από μελλοντικούς χρήστες αν χρειαστεί.

4.2 Υλοποίηση Εξατομικευμένων Plugins

Σε αυτή την ενότητα θα παρουσιαστούν η σχεδίαση και η υλοποίηση, μαζί με τον κώδικα, των εξατομικευμένων plugins που αναπτύχθηκαν στο πλαίσιο της παρούσας διπλωματικής. Στόχος των plugins είναι η επίτευξη ενεργειακά αποδοτικής δρομολόγησης φορτίων αλλά και η βέλτιστη τοποθέτηση τους σε κατάλληλους κόμβους. Θα παρουσιαστούν 5 plugins των οποίων το θεωρητικό περιεχόμενο παρουσιάστηκε στο κεφάλαιο 3.

4.2.1 Vector Bin Packing

Το παρακάτω plugin βασίστηκε στους αλγορίθμους και τη λογική του Vector Bin Packing[16] με γενικότερη αξιοποίηση του Best fit, σε όσο καλύτερη εφαρμογή γίνεται στο σειριακό τρόπο που υλοποιείται η χρονοδρομολόγηση στο Kubernetes.[7] Συγκεκριμένα, η θεωρητική υλοποίηση είναι στο υποκεφάλαιο 3.1. Επιλέχθηκε καθώς η στρατηγική Best-Fit οδηγεί σε ελαχιστοποίηση του μη χρησιμοποιούμενου χώρου, το οποίο συνδέεται με την ενεργειακή αποδοτικότητα καθώς επιτρέπει την απενεργοποίηση των κόμβων.

Στην τελική υλοποίηση το plugin υπολογίζει το εναπομείναν διανυσματικό «κενό» σε CPU και μνήμη στην περίπτωση που το Pod τοποθετηθεί στον κόμβο. Όσο μικρότερο το Vector gap, τόσο υψηλότερη και η βαθμολογία. Παρακάτω εμφανίζεται ο κώδικας με τα απαραίτητα σχόλια:

Σημείωση: σε αυτό το κομμάτι θα εμφανιστούν όλα τα κομμάτια του κώδικα, ωστόσο για τα επόμενα plugins θα παρουσιαστεί μονάχα η συνάρτηση Score() και όποια άλλη φτιάξαμε που είναι ξεχωριστή για το Plugin. Διαφορετικά, το μόνο που αλλάζει είναι η ονομασίες στις συναρτήσεις.

```
package vectorbinpacking

// Όλα τα imports γίνονται εδώ. Τα περισσότερα χρειάζονται για την
επικοινωνία με το Kubernetes. Υπάρχουν 2 frameworks για το kube-
scheduler καθώς πλέον το CycleState παρέχεται από το k8s.io/kube-
scheduler/framework
import (
    "context"
    "fmt"
    "math"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/runtime"
    framework "k8s.io/kubernetes/pkg/scheduler/framework"
    fwk "k8s.io/kube-scheduler/framework"
)

const Name = "VectorBinPacking"

// Δημιουργούμε το Struct για να μπορούμε να το καλέσουμε αργότερα
type VectorBinPacking struct {
      handle framework.Handle
}

var _ framework.ScorePlugin = &VectorBinPacking{}

// Η συνάρτηση New δημιουργεί το instance του plugin μας στον kube-
scheduler. Το μήνυμα στο logger υπάρχει ώστε να είμαστε σίγουρη για την
ορθή εκκίνηση του plugin.
func New(_ context.Context, _ runtime.Object, _ framework.Handle)
(framework.Plugin, error) {
```

```
    fmt.Println("⚠ VectorBinPacking NEW() CALLED!")
    return &VectorBinPacking{}, nil
}

// Επιστρέφει το όνομα του plugin
func (v *VectorBinPacking) Name() string {
    return Name
}

// Χρησιμοποιούμε αυτή τη συνάρτηση για να πάρουμε τα Pod Requested
Resources καθώς η συνάρτηση που ήταν υλοποιημένη δεν λειτουργεί, οπότε
την καλούμε για να γνωρίζουμε πως γίνεται με σωστό τρόπο.
func getPodResourceRequest(pod *v1.Pod) *framework.Resource {
    result := &framework.Resource{}
    // Υπολογίζετια η συνολική CPU που ζητείται από τα containers αλλά
και η συνολική μνήμη που ζητείται
    for _, c := range pod.Spec.Containers {
        result.MilliCPU += c.Resources.Requests.Cpu().MilliValue()
        result.Memory += c.Resources.Requests.Memory().Value()
    }
    // Στη περίπτωση που υπάρχει InitContainer υποχρεωτικά ζητείται
τουλάχιστον η CPU και η μνήμη που απαιτεί αυτός. Αν τα υπόλοιπα
containers απαιτούν λιγότερη, η δική του υπερισχύει διότι διαφορετικά
δεν μπορεί να λειτουργήσει το Pod.
    for _, c := range pod.Spec.InitContainers {
        cpu := c.Resources.Requests.Cpu().MilliValue()
        mem := c.Resources.Requests.Memory().Value()
        if cpu > result.MilliCPU {
            result.MilliCPU = cpu
        }
        if mem > result.Memory {
            result.Memory = mem
        }
    }
    return result
}

// Η συνάρτηση αξιολόγησης. Για τα επόμενα plugins κυρίως αυτή θα
παρέχεται. Πρώτα ορίζουμε τις μεταβλητές και στη συνέχεια αρχικοποιούμε
το σύστημα μας βάσει των πληροφοριών περί CPU και μνήμη κόμβου και Pod.
Το τελικό αποτέλεσμα δείχνει κατά πόσο το μήκος του vector είναι κοντά
στο 0. Όσο πιο κοντά στο 0 είναι, τόσο περισσότερο το προτιμάμε.
func (v *VectorBinPacking) Score(
    ctx context.Context,
    state fwk.CycleState,
    pod *v1.Pod,
    nodeInfo *framework.NodeInfo,
) (int64, *framework.Status) {
    // Πρώτα υπολογίζουμε πόση CPU και μνήμη μπορεί να αξιοποιηθεί
συνολικά, και πόση αξιοποιείται ήδη.
    allocatable := nodeInfo.Allocatable
```

```
    requested := nodeInfo.Requested

    // Στη συνέχεια υπολογίζουμε πόση CPU και μνήμη απαιτείται από το
Pod που δρομολογείται.
    podReq := getPodResourceRequest(pod)

    // Υπολογίζεται η CPU και η μνήμη που όντως απομένει στον κόμβο για
να παρέχει.
    cpuRemaining := allocatable.MilliCPU - requested.MilliCPU
    memRemaining := allocatable.Memory - requested.Memory

    // Υπολογίζεται η CPU και η μνήμη που θα απομένει στον κόμβο μετά τη
προσθήκη του Pod.
    cpuLeft := cpuRemaining - podReq.MilliCPU
    memLeft := memRemaining - podReq.Memory

    if cpuLeft < 0 || memLeft < 0 {
        return 0, framework.NewStatus(framework.Success)
    }

    // Vector bin packing: όσο πιο κοντά στο 0 είναι το διάνυσμα, τόσο
το καλύτερο. Η διαίρεση γίνεται με την υψηλότερη δυνατή τιμή από το
άθροισμα των τετραγώνων για την συνολική CPU και τη συνολική μνήμη που
μπορεί να δοθεί.
    cpuLeftF = float64(cpuLeft)
    memLeftF = float64(memLeft)/(1024.0*1024.0)
    allocCPUF = float64(allocatable.MilliCPU)
    allocMemF = float64(allocatable.Memory)/(1024.0*1024.0) // Μετατροπή
των bytes σε MiB


    denominator := math.Sqrt(allocCPUF*cpuLeftF + allocMemF*memLeftF)
    numerator := math.Sqrt(cpuLeftF*cpuLeftF + memLeftF *memLeftF)


    scoreF := 100.0 - (numerator / denominator)*100.0

    if scoreF < 0 {
        scoreF = 0
    } else if scoreF > 100 {
        scoreF = 100
    }
    score = int64(scoreF)

    // Παρουσιάζουμε τα αποτελέσματα στο Log για τη δυνατότητα ελέγχου
του plugin.
    fmt.Printf("Scoring Node %s: CPU Allocatable = %dm, Mem Allocatable
= %dMi, CPU left = %dm, Mem left = %dMi, Score = %d\n",
nodeInfo.GetName(), allocatable.MilliCPU, allocatable.Memory, cpuLeft,
memLeft/(1024*1024), score)

    return score, framework.NewStatus(framework.Success)
```

```
}

// Συνάρτηση που χρειάζεται για ScoreExtensions αλλά δεν απαιτεί
παραπάνω κώδικα. Δεν θα παρουσιαστεί για τα επόμενα plugins.
func (v *VectorBinPacking) ScoreExtensions() framework.ScoreExtensions {
    return nil
}
```

Κώδικας 4.1 Κώδικας plugin για το Vector Bin Packing

Η αναμενόμενη συμπεριφορά με το παραπάνω plugin είναι να τοποθετούνται τα pod στο κόμβο με το καλύτερο fit, αυτούς που θα έχουν το μικρότερο τελικό κενό. Έτσι συγκεντρώνεται ο φόρτος και οι άλλοι κόμβοι παραμένουν κενοί και μπορούν να απενεργοποιηθούν.

4.2.2 Ενεργειακά Συνειδητό Plugin για πράσινη ενέργεια και βιωσιμότητα

Το μοντέλο που χρησιμοποιείται για το συγκεκριμένο plugin αναπτύχθηκε θεωρητικά στο υποκεφάλαιο 3.2.1. Βασίζεται στη λογική πως η ενεργειακή κατανάλωση είναι ανάλογη της χρήσης CPU, αλλά εξαρτάται παράλληλα και από την πηγή ενέργειας του μηχανήματος/κόμβου. Επιλέχθηκε ώστε να ελαχιστοποιηθεί η κατανάλωση ενέργειας σε συστήματα που τροφοδοτούνται από ανανεώσιμες πηγές ενέργειας αλλά και για την μείωση παραγωγής διοξειδίου του Άνθρακα. [11], [18], [19]

Κατά την υλοποίηση υπολογίζει σαν τελικό αποτέλεσμα πόσο καλό ενεργειακά είναι το μηχάνημα. Όσο μεγαλύτερο είναι το αποτέλεσμα, τόσο καλύτερο είναι το μηχάνημα και προτιμάται για δρομολόγηση.

Ο κώδικας, με τον απαραίτητο σχολιασμό, παρέχεται παρακάτω:

```
import (
    "context"
    "fmt"
    "math"
    "strconv"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/runtime"
    framework "k8s.io/Kubernetes/pkg/scheduler/framework"
    fwk "k8s.io/kube-scheduler/framework"
)

// Στο Plugin αυτό υπολογίζουμε την ενέργεια που καταναλώθηκε και πόσο
sustainable είναι ο κόμβος βάση δεικτών για προβλεπόμενη ενεργειακή
κατανάλωση.
func (v *EnergyAwareUsage) Score(
    ctx context.Context,
    state fwk.CycleState,
    pod *v1.Pod,
```

```
    nodeInfo *framework.NodeInfo,
) (int64, *framework.Status) {
      allocatableMilliCPU := float64(nodeInfo.Allocatable.MilliCPU)

      requestedMilliCPU := float64(nodeInfo.Requested.MilliCPU)

      podReq := getPodResourceRequest(pod)
      podReqMilliCPU := float64(podReq.MilliCPU)

      usedMilliCPU := requestedMilliCPU + podReqMilliCPU
      cpuUtil := 0.0
      if allocatableMilliCPU > 0 {
            cpuUtil = usedMilliCPU / allocatableMilliCPU
      }

      // Προεπιλεγμένες τιμές
      greenFactor := 1.0
      carbonPenalty := 1.0
      carbonIntensity := 1.0

      // Διαβάζουμε τα annotations που προσθέσαμε στους κόμβους
      if gfStr, ok := nodeInfo.Node().Annotations["greenFactor"]; ok {
            if val, err := strconv.ParseFloat(gfStr, 64); err == nil {
                  greenFactor = val
            }
      }
      if cpStr, ok := nodeInfo.Node().Annotations["carbonPenalty"]; ok {
            if val, err := strconv.ParseFloat(cpStr, 64); err == nil {
                  carbonPenalty = val
            }
      }

      // Διαβάζουμε τα annotations που προσθέσαμε στα Pods
      if ciStr, ok := pod.Annotations["carbonIntensity"]; ok {
            if val, err := strconv.ParseFloat(ciStr, 64); err == nil {
                  carbonIntensity = val
            }
      }

      // Υπολογίζουμε η τελική αξιολόγηση βιωσιμότητας
      rawScore := (cpuUtil * greenFactor) - (carbonPenalty *
carbonIntensity)

      // Κανονικοποιούμε το rawScore στο 1-100
      // Θεωρούμε πως οι πιθανές τιμές είναι μεταξύ -2 και 2 βάσει
υπολογισμών στα μηχανήματα.
      minRaw := -2.0
      maxRaw := 2.0
      clamped := math.Max(minRaw, math.Min(maxRaw, rawScore))
      normalized := (clamped - minRaw) / (maxRaw - minRaw)
      finalScore := int64(1 + normalized*99)
```

```
	fmt.Printf("Scoring Node %s: cpuUtil=%.2f, greenFactor=%.2f,
carbonPenalty=%.2f, carbonIntensity=%.2f → Score = %d\n",
		nodeInfo.GetName(), cpuUtil, greenFactor, carbonPenalty,
carbonIntensity, finalScore)

	return finalScore, framework.NewStatus(framework.Success)
}
```

Κώδικας 4.2 Plugin για πράσινη ενέργεια και βιωσιμότητα

Η αναμενόμενη συμπεριφορά είναι η αποφυγή κόμβων που έχουν μεγαλύτερο πρόστιμο για παραγωγή διοξειδίου του άνθρακα και δεν είναι ιδιαίτερα φιλικοί προς το περιβάλλον βάσει των πηγών τροφοδοσίας τους. Προκειμένου να δουλέψει ορθά αυτό το plugin χρειάζεται να έχουν παραχωρηθούν χειροκίνητα οι παραπάνω τιμές τόσο στους κόμβους όταν τους εισαγάγουμε στο σύστημα, όσο και στα Pods που θέλουμε να δρομολογήσουμε.

4.2.3 Ενεργειακά Συνειδητό EATSVM Plugin

Το θεωρητικό υπόβαθρο για το παρόν plugin παρουσιάστηκε στο υποκεφάλαιο 3.2.2. Επιλέχθηκε καθώς το μοντέλο EATSVM[8] υποθέτει πως η ενεργειακή κατανάλωση είναι συνάρτηση της χρήσης CPU, της αρχιτεκτονικής και της ταχύτητας του ρολογιού. Επιπρόσθετα, παρέχει ένα ημι-αναλυτικό τρόπο πρόβλεψης κατανάλωσης ισχύος και είναι κατάλληλο για τις περιπτώσεις στις οποίες δεν υπάρχει πρόσβαση σε πραγματικές μετρικές.

Κατά την υλοποίηση, το plugin υπολογίζει την μελλοντική αξιοποίηση της CPU εάν το Pod δρομολογηθεί στον κόμβο που εξετάζεται και εκτιμά την ισχύ ως το γινόμενο της προβλεπόμενης χρήσης της CPU επί τους κύκλους του ρολογιού σε GHz. Η τιμή αυτή στη συνέχεια κανονικοποιείται μεταξύ των 2 ακραίων τιμών, 0 και μέγιστη ταχύτητα ρολογιού. Στη συνέχεια αντιστρέφεται προκειμένου να δώσει τον υψηλότερο βαθμό σε κόμβους με χαμηλότερη εκτιμώμενη κατανάλωση. Αυτή η δρομολόγηση ευνοεί τους κόμβους που θα καταναλώσουν λιγότερη ισχύ με την προσθήκη του Pod.

Ο κώδικας που χρησιμοποιήθηκε είναι ο παρακάτω:

```
import (
    "context"
    "fmt"
    "math"
    "strconv"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/runtime"
    framework "k8s.io/Kubernetes/pkg/scheduler/framework"
    fwk "k8s.io/kube-scheduler/framework"
)
```

```
const nodeClockSpeed = 2.67 // Οι κύκλοι του ρολογιού σε GHz


// Αντίστοιχη συνάρτηση αξιολόγησης. Παίρνουμε δεδομένα χρήσης από τον
κόμβο και το Pod στην αρχή και στη συνέχεια βλέπουμε τη προβλεπόμενη
χρήση της CPU με την προσθήκη του Pod και υπολογίζουμε την ισχύ που θα
καταναλώνει. Υπολογίζουμε το τελικό score και όσο μεγαλύτερο είναι, τόσο
το καλύτερο.
func (v *EnergyAwareEATSVM) Score(
      ctx context.Context,
      state fwk.CycleState,
      pod *v1.Pod,
      nodeInfo *framework.NodeInfo,
) (int64, *framework.Status) {

      // Έχουμε σε μεταβλητές την μέγιστη πιθανή CPU, την χρήση CPU από
τον κόμβο αλλά και τη χρήση που ζητάει το pod.
      allocatableMilliCPU := float64(nodeInfo.Allocatable.MilliCPU)
      requestedMilliCPU := float64(nodeInfo.Requested.MilliCPU)

      podReq := getPodResourceRequest(pod)
      podReqMilliCPU := float64(podReq.MilliCPU)

      // Η προβλεπόμενη χρήση CPU είναι η CPU που χρησιμοποιείται ήδη
μαζί με τη CPU που ζητείται από το Pod, και μετά τη βρίσκουμε ως ποσοστό
προς τη συνολική CPU που μπορεί να δοθεί
      predictedUsedMilliCPU := requestedMilliCPU + podReqMilliCPU
      predictedCPUUtil := 0.0
      if allocatableMilliCPU > 0 {
            predictedCPUUtil = predictedUsedMilliCPU /
allocatableMilliCPU
      }

      // Παίρνουμε τη τιμή της ταχύτητας του ρολογιού από τα annotations
      clockSpeed := nodeClockSpeed
      if val, ok := nodeInfo.Node().Annotations["clockSpeed"]; ok {
          if parsed, err := strconv.ParseFloat(val, 64); err == nil {
              clockSpeed = parsed
          }
      }

      // Η συνολική προβλεπόμενη ισχύ υπολογίζεται ως το ποσοστό χρήσης
της CPU επί τους κύκλους του ρολογιού, που μπορεί να είναι από 0 έως και
τους κύκλους του ρολογιού
      predictedTotalPower := clockSpeed * predictedCPUUtil

      minExpectedPower := 0.0
      maxExpectedPower := nodeClockSpeed * 1.0
      clampedPower := math.Max(minExpectedPower,
math.Min(maxExpectedPower, predictedTotalPower))
```

```
	// Κανονικοποιούμε την τιμή μεταξύ του 0 και του 1 πριν την
επιστρέψουμε στον χρονοδρομολογητή.
	normalizedValue := 0.0
	if maxExpectedPower > minExpectedPower {
		normalizedValue = (clampedPower - minExpectedPower) /
(maxExpectedPower - minExpectedPower)
	}

	// Κανονικοποιούμε την τιμή μεταξύ του 1 και του 100, αλλά επειδή
θέλουμε η δρομολόγηση να ευνοεί τους κόμβους με καλύτερη λιγότερη
παραγωγή ισχύος, το αντιστρέφουμε.
	finalScore := int64(1 + (1.0-normalizedValue)*99.0)
	finalScore = int64(math.Max(1, math.Min(100,
float64(finalScore))))

	fmt.Printf("Scoring Node %s: Clock Speed = %.2f, Predicted CPU
Util = %.2f%%, Predicted Total Power = %.4f, Score = %d\n",
		nodeInfo.GetName(), clockSpeed, predictedCPUUtil*100,
predictedTotalPower, finalScore)

	return finalScore, framework.NewStatus(framework.Success)
}
```

Κώδικας 4.3 Plugin με χρήση EATSVM

Η αναμενόμενη συμπεριφορά του plugin είναι να προτιμά κόμβους που καταναλώνουν λιγότερη ισχύ για την εκτέλεση ενός Pod. Αυτό οδηγεί σε πιο καλή ενεργειακή απόδοση και κατανομή, χωρίς ωστόσο να απαιτεί πρόσβαση σε πραγματικά hardware metrics που ενδέχεται να μην μπορούμε να λάβουμε.

4.2.4 Plugin που βασίζεται στην ομοιότητα διανυσμάτων

Το θεωρητικό υπόβαθρο για το συγκεκριμένο plugin υπάρχει στο υποκεφάλαιο 3.3.4 καθώς βασίζεται στο Dot Product και το Cosine Similarity. Επιλέχθηκε καθώς η ομοιότητα μεταξύ των απαιτήσεων ενός Pod με την διαθεσιμότητα ενός κόμβου μπορεί να εκφραστεί διανυσματικά και επιτρέπει ένα ακριβέστερο ταίριασμα. Αυτό συμβαδίζει με τις τεχνικές Best-Fit από το Bin Packing[7], [23] που θέλουμε να χρησιμοποιήσουμε.

Κατά την υλοποίηση υπολογίζει το Dot Produce αλλά και το Cosine Similarity στα διανύσματα CPU και μνήμης για τους πόρους που ζητάει το Pod αλλά και για τους πόρους που διαθέτει διαθέσιμους ο κόμβος. Ο συνδυασμός των 2 οδηγεί σε ένα ζυγισμένο βαθμό στο τέλος, με έμφαση στο Dot Product (3/4) έναντι του Cosine Similarity (1/4). Επιλέξαμε αυτό το ποσοστό επειδή προτιμάμε να υπάρχει υψηλό dot product για να έχουμε περισσότερους διαθέσιμους πόρους, αλλά παρότι θέλουμε να υπάρχει ομοιομορφία κατά την ανάθεση, δεν την απαιτούμε τόσο πολύ.

Ο κώδικας που χρησιμοποιήθηκε φαίνεται παρακάτω:

```
import (
```

```
    "context"
    "fmt"
    "math"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/runtime"
    framework "k8s.io/Kubernetes/pkg/scheduler/framework"
    fwk "k8s.io/kube-scheduler/framework"
)


// Σε αυτή τη συνάρτηση αξιολόγησης φτιάχνουμε 2 διανύσματα, το ένα με
βάση την CPU και τη μνήμη που απομένει στον κόμβο και το άλλο με βάση
την CPU και τη μνήμη που ζητάει το Pod.
func (v *ResourceVectorSimilarity) Score(
      ctx context.Context,
      state fwk.CycleState,
      pod *v1.Pod,
      nodeInfo *framework.NodeInfo,
) (int64, *framework.Status) {

      // Βάζουμε στις μεταβλητές τις τιμές που έχουμε από τον κόμβο και
το Pod σχετικά με τη CPU και τη μνήμη.
      allocatableMilliCPU := float64(nodeInfo.Allocatable.MilliCPU)
      allocatableMemory := float64(nodeInfo.Allocatable.Memory)

      requestedMilliCPU := float64(nodeInfo.Requested.MilliCPU)
      requestedMemory := float64(nodeInfo.Requested.Memory)

      podReq := getPodResourceRequest(pod)
      podReqMilliCPU := float64(podReq.MilliCPU)
      podReqMemory := float64(podReq.Memory)


      // Υπολογίζουμε τη CPU και τη μνήμη που απομένει στο κόμβο
      // cpuRemaining := allocatableMilliCPU - requestedMilliCPU
      // memRemaining := allocatableMemory - requestedMemory

      // Βάζουμε τις τιμές που θέλουμε στα διανύσματα μας
      vectorA_CPU := requestedMilliCPU
      vectorA_Mem := requestedMemory /(1024*1024)

      vectorB_CPU := podReqMilliCPU
      vectorB_Mem := podReqMemory /(1024*1024)

      // Υπολογίζουμε το Dot Product
      dotProduct := (vectorA_CPU * vectorB_CPU) + (vectorA_Mem *
vectorB_Mem)

      // Υπολογίζουμε το Cosine Similarity
      magnitudeA := math.Sqrt(vectorA_CPU*vectorA_CPU +
vectorA_Mem*vectorA_Mem)
```

```
	magnitudeB := math.Sqrt(vectorB_CPU*vectorB_CPU +
vectorB_Mem*vectorB_Mem)

	cosineSimilarity := 0.0
	if magnitudeA > 0 && magnitudeB > 0 {
		cosineSimilarity = dotProduct / (magnitudeA * magnitudeB)
	}

		// minexpecteddotproduct := 0.0
		maxExpectedDotProduct := (allocatableMilliCPU *
podReqMilliCPU) + (allocatableMemory * podReqMemory)

		normalizedDotProduct := math.Min(1.0, dotProduct /
maxExpectedDotProduct)
		normalizedDotProduct = math.Max(0.0, normalizedDotProduct)

	// Υπολογίζουμε το τελικό αποτέλεσμα με ποσοστό 3/4 για το dot product και 1/4 για
το Cosine similarity
	finalScore := int64(1 + normalizedDotProduct*75 +
cosineSimilarity*25)

	finalScore = int64(math.Max(1, math.Min(100,
float64(finalScore))))

	fmt.Printf("📊 Scoring Node %s: Rem CPU = %dm, Rem Mem = %dMi |
Pod Req CPU = %dm, Pod Req Mem = %dMi | Dot Product = %.2f, Cosine
Similarity = %.4f, Score = %d\n",
		nodeInfo.GetName(), int64(vectorA_CPU),
int64(vectorA_Mem)/(1024*1024),
		int64(vectorB_CPU), int64(vectorB_Mem)/(1024*1024),
		dotProduct, cosineSimilarity, finalScore)

	return finalScore, framework.NewStatus(framework.Success)
}
```

Κώδικας 4.4 Plugin για ομοιότητα διανυσμάτων

Η συμπεριφορά που αναμένουμε είναι ο χρονοδρομολογητής να προτιμά κόμβους που ταιριάζουν διανυσματικά οι διαθέσιμοι τους πόροι με τις απαιτήσεις του Pod. Έτσι, θα υπάρχει μια ισορροπία και αποδοτικότητα στους κόμβους, ενώ παράλληλα έχουμε ως προτεραιότητα μας την ενεργειακή απόδοση.

4.2.5 Dense Packing DPM Plugin

Αυτό το plugin παρουσιάζεται θεωρητικά στο κεφάλαιο 3.2.4 και βασίζεται στις τεχνικές DPM[21] και DVFS[9], [24] σε συνδυασμό με τη πυκνή συσκευασία και τη χρήση σταθερότητας του κόμβου. Το επιλέξαμε καθώς θέλουμε να έχουμε όσο λιγότερους κόμβους γίνεται ενεργούς και να έχουμε όσο πιο πυκνή δρομολόγηση Pods γίνεται. Γενικά, υπάρχει προτίμηση για κόμβους που είναι σταθερά ενεργοί για μεγαλύτερο χρονικό διάστημα και που χρησιμοποιούνται έντονα.

Κατά την υλοποίηση υπολογίζουμε 2 συνιστώσες, την προβλεπόμενη αξιοποίηση πόρων (CPU και μνήμη) καθώς και το «uptime» του κόμβου. Ο τελικός βαθμός είναι σταθμισμένος ως συνδυασμός αυτών των 2. Με αυτό το τρόπο επιτυγχάνουμε να ενσωματώσουμε εμμέσως την ενεργειακή απόδοση, αφού οι κόμβοι που ήδη λειτουργούν εντατικά αποδίδουν καλύτερα ανά watt σε σχέση με την ενεργοποίηση νέων κόμβων. Δίνεται μεγαλύτερη βάση στην προβλεπόμενη αξιοποίηση των πόρων ώστε αυτή να είναι όσο μεγαλύτερη γίνεται, ενώ παράλληλα δίνεται λιγότερη στο uptime. Ενώ προτιμάμε κόμβους που είναι ενεργοί για μεγαλύτερο χρονικό διάστημα, θέλουμε παράλληλα να γεμίσουμε τους κόμβους που χρησιμοποιούν ήδη περισσότερους πόρους. Για αυτό δίνουμε βάρος 0.5 στη χρήση μνήμης και CPU και 0.5 στο uptime.

Ο κώδικας που χρησιμοποιούμε για το plugin είναι παρακάτω:

```
import (
    "context"
    "fmt"
    "math"
    "time"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/runtime"
    framework "k8s.io/Kubernetes/pkg/scheduler/framework"
    fwk "k8s.io/kube-scheduler/framework"
)


// Η συνάρτηση getUptimeScoreComponent υπολογίζει ένα βαθμό στο διάστημα
(0.0 προς 1.0) βασισμένο στο uptime του κόμβου.
// Όσο περισσότερο είναι ενεργός ένας κόμβος, τόσο μεγαλύτερος ο βαθμός
func getUptimeScoreComponent(node *v1.Node) float64 {

      // Θεωρούμε πως κατά το χειρότερο ένας κόμβος τρέχει για 1 ώρα και
όποιος κόμβος είναι ενεργός για 15 μέρες θεωρείται πολύ καλός.
Προτείνεται σε μελλοντικές υλοποιήσεις να είναι υψηλότερος ο αριθμός.
      minUptimeDuration := 1 * time.Hour
      maxUptimeDuration := 15 * 24 * time.Hour

      // Βρίσκουμε αν ο κόμβος είναι έτοιμος και από πότε.
      var readyTime time.Time
      foundReady := false
      for _, condition := range node.Status.Conditions {
            if condition.Type == v1.NodeReady && condition.Status ==
v1.ConditionTrue {
                  readyTime = condition.LastTransitionTime.Time
                  foundReady = true
                  break
            }
      }
      // Αν δεν βρεθεί ready state δεν θεωρείται καλό για να
δρομολογηθεί εκεί ένα Pod.
```

```
	if !foundReady {
		return 0.0
	}

	// Υπολογίζουμε το uptime
	uptime := time.Since(readyTime)

	if uptime < minUptimeDuration {
		return 0.0
	}
	if uptime >= maxUptimeDuration {
		return 1.0
	}

	// Βάζουμε γραμμικά το uptime μεταξύ των τιμών 0.0 και 1.0
	return float64(uptime-minUptimeDuration) /
float64(maxUptimeDuration-minUptimeDuration)
}


// Η συνάρτηση αξιολόγησης αξιολογεί τους κόμβους βάση της μέγιστης
χρήσης CPU και μνήμης αλλά και με βάση το uptime τους.
func (v *DensePackingDPM) Score(
	ctx context.Context,
	state fwk.CycleState,
	pod *v1.Pod,
	nodeInfo *framework.NodeInfo,
) (int64, *framework.Status) {
	// Βάζουμε τις τιμές που χρειαζόμαστε από το κόμβο και το Pod στις
μεταβλητές
	allocatableMilliCPU := float64(nodeInfo.Allocatable.MilliCPU)
	allocatableMemory := float64(nodeInfo.Allocatable.Memory)

	requestedMilliCPU := float64(nodeInfo.Requested.MilliCPU)
	requestedMemory := float64(nodeInfo.Requested.Memory)

	podReq := getPodResourceRequest(pod)
	podReqMilliCPU := float64(podReq.MilliCPU)
	podReqMemory := float64(podReq.Memory)

	// Υπολογίζουμε το dense packing με βάση τη χρήση των πόρων.
	predictedUsedMilliCPU := requestedMilliCPU + podReqMilliCPU
	predictedUsedMemory := requestedMemory + podReqMemory

	// Υπολογίζουμε τις προβλεπόμενες χρήσης της CPU και της μνήμης
	predictedCPUUtil := 0.0
	if allocatableMilliCPU > 0 {
		predictedCPUUtil = predictedUsedMilliCPU /
allocatableMilliCPU
	}

	predictedMemUtil := 0.0
```

```
	if allocatableMemory > 0 {
		predictedMemUtil = predictedUsedMemory / allocatableMemory
	}

	// Η κατά μέσο όρο χρησιμοποίηση CPU και μνήμης (0.0 - 1.0)
	resourceUtilScoreComponent := (predictedCPUUtil +
predictedMemUtil) / 2.0


	// Υπολογίζουμε το uptime
	uptimeScoreComponent := getUptimeScoreComponent(nodeInfo.Node())


	// Θέτουμε τα βάρη για τα διαφορετικά Score.
	const resourceWeight = 0.50
	const uptimeWeight = 0.50

	// Ο μέσος όρος των τιμών μετά το πολλαπλασιασμό τους με τα
προεπιλεγμένα βάρη. (το καθένα είναι 0.0-1.0)
	combinedScoreRaw := (resourceUtilScoreComponent * resourceWeight)
+ (uptimeScoreComponent * uptimeWeight)

	// Κανονικοποιούμε το score στη κλίμακα από το 1 ως το 100, με τη
μονάδα στο combinesScoreRaw να αντιστοιχεί στο 100 και το 0 στο 1.
	finalScore := int64(1 + combinedScoreRaw*99.0)

	finalScore = int64(math.Max(1, math.Min(100,
float64(finalScore))))

	fmt.Printf("Scoring Node %s (Dense Packing + Uptime): Current CPU
Util = %.2f%%, Mem Util = %.2f%% | Pred CPU Util = %.2f%%, Mem Util =
%.2f%% | Resource Comp = %.2f, Uptime Comp = %.2f, Final Score = %d\n",
		nodeInfo.GetName(),
		(requestedMilliCPU/allocatableMilliCPU)*100,
(requestedMemory/allocatableMemory)*100,
		predictedCPUUtil*100, predictedMemUtil*100,
		resourceUtilScoreComponent, uptimeScoreComponent,
		finalScore)

	return finalScore, framework.NewStatus(framework.Success)
}
```

Κώδικας 4.5 Plugin βασισμένο στο DPM

Η αναμενόμενη συμπεριφορά του plugin είναι να συγκεντρώνει τα Pods σε ενεργούς και σταθερούς κόμβους. Με αυτό το τρόπο προωθεί την απενεργοποίηση των λιγότερο χρήσιμων κόμβων και την αποφυγή διασποράς των φορτίων.

4.3 Τελικός συνδυασμός των Plugin

Αφού παρουσιάστηκαν και τεκμηριώθηκαν τόσο τα προεπιλεγμένα όσο και τα εξατομικευμένα plugins, σε αυτή την ενότητα περιγράφεται συνοπτικά ο τρόπος με τον οποίο συνδυάζονται στον τελικό χρονοδρομολογητή. Η επιλογή των plugins καθώς και η ανάθεση βαρών θα γίνεται δυναμικά ανάλογα με το εκάστοτε πειραματικό σενάριο, με στόχο τη μελέτη της απόδοσης διαφορετικών στρατηγικών τοποθέτησης. Θα εξεταστούν διαφορετικοί συνδυασμοί ώστε να έχουμε τον πιο ενεργειακά συνειδητό χρονοδρομολογητή.[13], [14], [15]

Τα plugins ενεργοποιούνται μέσω κατάλληλης ρύθμισης ενός custom scheduler profile (KubeSchedulerConfiguration) και χρησιμοποιούνται με το αντίστοιχο schedulerName στα Pods. Αναλυτικά παραδείγματα διαμόρφωσης θα παρουσιαστούν στο επόμενο κεφάλαιο (Κεφ. 5), στο οποίο παρέχεται και το configuration που θα χρησιμοποιήσουμε αλλά και τα pods για τις όποιες προσομοιώσεις. Περισσότερες πληροφορίες για το πως φτιάχνεται ένας χρονοδρομολογητής kube-scheduler παρουσιάζονται στο Παράρτημα 1.

5. Προσομοίωση και Αξιολόγηση

Σε αυτό το κεφάλαιο θα πραγματοποιηθεί η τελική πειραματική αξιολόγηση του χρονοδρομολογητή μας. Θα συγκριθεί με τον προεπιλεγμένο χρονοδρομολογητή, καθώς και με άλλα profiles από τον δικό μας με διαφορετικά plugins ενεργά ή διαφορετικά δοσμένα βάρη. Στη συνέχεια από κάθε profile θα πάρουμε πληροφορίες από το ίδιο σύστημα με τους ίδιους κόμβους και τα ίδια Pods. Στο τέλος θα ελέγξουμε εάν η δρομολόγηση μας είναι όπως την θέλουμε ιδανικά (πυκνή και ενεργειακά συνειδητή) με βάση την κατανομή των Pods στους κόμβους αλλά και τις μετρικές που εμφανίζουν οι μετρικές των metrics.k8s.io, Prometheus και Kepler (αν δουλέψει).

5.1 Περιβάλλον πειραματικής αξιολόγησης

Το περιβάλλον που χρησιμοποιήθηκε για την πειραματική αξιολόγηση αποτελείται από ψηφιακά μηχανήματα (Virtual Machines), χρησιμοποιεί το σύστημα Microk8s, το οποίο περιεγράφηκε στο κεφάλαιο 2. Επίσης, θα χρησιμοποιήσουμε διαφορετικά profiles ώστε να συγκρίνουμε διαφορετικές εκδοχές του χρονοδρομολογητή μας και να βρούμε την βέλτιστή και πιο ενεργειακά συνειδητή.

5.1.1 Οι κόμβοι

Συνολικά στο cluster μας έχουμε 3 κόμβους, οι προδιαγραφές των οποίων φαίνονται παρακάτω:

	Κόμβος 1	**Κόμβος 2**	**Κόμβος 3**
Public IP	Ναι	Όχι	Όχι
Private IP	172.16.102.233	172.16.102.153	172.16.102.227
Number of cores	8 (1 per socket)	8 (1 per socket)	8 (1 per socket)
Threads per core	1	1	1
Clock speed (MHz) (δικά μας νούμερα)	3265.908	2665.908	1885.908
Total disk size	38 GB	38 GB	38 GB
Total RAM	7.8 GiB	7.8 GiB	7.8 GiB
greenFactor	1.8 (solar)	1.6 (wind)	0.8 (petroleum)
carbonPenalty	1.2	0.8	1.5

Πίνακας 5.1 Κόμβοι στο πειραματικό περιβάλλον

Τα νούμερα σχετικά με τους κύκλους ρολογιού, τον πράσινο παράγοντα και το πέναλτυ για παραγωγή διοξειδίου του άνθρακα είναι δικά μας θεωρητικά. Μπορεί ο κάθε χρήστης να τα

θέσει για τα μηχανήματα που θέλει. Σε περίπτωση που δεν τεθούν η προεπιλεγμένη τιμή είναι 1.0 και για τις δύο τιμές, ενώ για τους κύκλους ρολογιού η προεπιλεγμένη τιμή είναι 2.67 GHz.

Ο τρόπος με τον οποίο εισάγουμε τις τιμές αυτές στα nodes είναι μέσω του παρακάτω script σε bash:

```
# soumplis-piliotis-1
microk8s kubectl annotate node soumplis-piliotis-1 greenFactor="1.8" --overwrite
microk8s kubectl annotate node soumplis-piliotis-1 carbonPenalty="1.2" --overwrite
microk8s kubectl annotate node soumplis-piliotis-1 clockSpeed="3.27" --overwrite

# soumplis-piliotis-2
microk8s kubectl annotate node soumplis-piliotis-2 greenFactor="1.6" --overwrite
microk8s kubectl annotate node soumplis-piliotis-2 carbonPenalty="0.8" --overwrite
microk8s kubectl annotate node soumplis-piliotis-2 clockSpeed="2.67" --overwrite

# soumplis-piliotis-3
microk8s kubectl annotate node soumplis-piliotis-3 greenFactor="0.8" --overwrite
microk8s kubectl annotate node soumplis-piliotis-3 carbonPenalty="1.5" --overwrite
microk8s kubectl annotate node soumplis-piliotis-3 clockSpeed="1.89" --overwrite
```

Κώδικας 5.1 Script για εισαγωγή παραμέτρων στους κόμβους

Όπως μπορεί να φανεί από τις τιμές, ο κόμβος 2 είναι ο πιο καλός για δρομολόγηση για τον δρομολογητή μας. Συνολικά θέλουμε να έχει υψηλό green factor, χαμηλό carbon penalty και όσο πιο χαμηλό clock speed γίνεται. Ο κόμβος 2 είναι ο καλύτερος συγκριτικά.

5.1.2 Προσομοιώσεις που θα τρέξουν με τα αντίστοιχα pods

Σε αυτό το κομμάτι θα παρουσιαστούν περιληπτικά οι προσομοιώσεις που θα τρέξουμε για να ελέγξουμε πόσο καλά λειτουργεί το σύστημα μας. Θα τρέξει η κάθε προσομοίωση για καθένα από τα παρακάτω profiles. Οι προσομοιώσεις αυξάνονται σε απαιτήσεις, με την πρώτη να είναι η πιο ήπια και τη τελευταία να είναι η πιο απαιτητική.

Στις πρώτες 4 προσομοιώσεις αξιοποιούμε τη συνάρτηση stress για να δούμε όντως πως θα διαφέρει η χρήση και πόση ενέργεια θα καταναλώνεται, πάντα χρησιμοποιώντας αυστηρά λιγότερη μνήμη από όση δεσμεύεται. Διαφορετικά, με μια παθητική συνάρτηση όπως η sleep δεν θα εμφανίζονταν οι διαφορές που θέλουμε. Στη τελευταία μονάχα προσομοίωση θα δούμε με ποια σειρά γίνεται η δρομολόγηση και ποια προτιμάμε.

Οι προσομοιώσεις στη βάση τους είναι αρκετά τυχαίες. Φτιάχτηκαν με σκοπό να δούμε κατά κύριο λόγο την δρομολόγηση που θα κάνει ο εκάστοτε χρονοδρομολογητής. Για τον Προεπιλεγμένο χρονοδρομολογητή αναμένουμε να κάνει ισορροπημένη κατανομή σε όλους τους κόμβους, ή στους πιο «καλούς» κόμβους, βάσει των δικών του κριτηρίων. Ο δικός μας χρονοδρομολογητής, όντας πιο ενεργειακά συνειδητός και έχοντας τη λογική πυκνής δρομολόγησης σε ενεργειακά συμφέροντες κόμβους, αναμένουμε να δρομολογήσει τους κόμβους είτε στον κόμβο 1 είτε στον κόμβο 2, οι οποίοι θεωρούμε είναι πιο ενεργειακά συμφέροντες. Συνεπώς περιμένουμε πρώτα να γεμίσει τον κόμβο 2 για παράδειγμα, και στη συνέχεια να προχωρήσει στον επόμενο, πιο ενεργειακά συμφέρων κόμβο.

Επιπροσθέτως, οι προσομοιώσεις αυξάνονται σε ένταση αλλά διατηρούν τη λογική τους: μερικά Pods πολύ ήπια και άλλα που όσο πάει είναι όλο και πιο έντονα. Η ένταση δεν φτάνει καμία στιγμή σε βαθμό που δεν μπορεί να διαχειριστεί ένας κόμβος στις 4 πρώτες προσομοιώσεις. Συνεπώς, προκειμένου να ελέγξουμε συνολικά το σύστημα, τρέχουμε και μια πιο απλή προσομοίωση με περισσότερα Pods τα οποία δεσμεύουν τη μνήμη, αλλά δεν τη χρησιμοποιούν όντως. Με αυτό θα δούμε ποια θα είναι η προτεραιότητα του εξατομικευμένου χρονοδρομολογητή και αν πηγαίνει από ένα κόμβο σε ένα άλλο όπως θα θέλαμε, ή αν αρχίζει να ισορροπεί περισσότερο.

Στις υπόλοιπες μετρικές που θα συλλέξουμε, θέλουμε ιδανικά ο δικός μας χρονοδρομολογητής να χρησιμοποιεί λιγότερους από τους πόρους. Ωστόσο, όπως είναι λογικό, δεν μπορεί να υπάρξει μεγάλη διαφορά. Αν ένα Pod απαιτεί ένα πυρήνα για να εκτελεστεί, θα απαιτεί τόσο σε όποιο κόμβο κι αν καταλήξει. Όπως και να διαμοιραστούν τα Pods, το λογικό είναι να η συνολική CPU και μνήμη που χρησιμοποιούνται, αλλά και η ενέργεια που παράγεται – εφόσον βασίζεται σε αυτές τις τιμές – να είναι περίπου ίσες στον αριθμό.

Προσομοίωση 1 – Ήπια

Pod Name	CPU	Memory	Carbon Intensity
pod-1	50m	1536Mi	0.48
pod-2	250m	128Mi	0.62
pod-3	2000m	64Mi	0.88
pod-4	50m	64Mi	1.43

Πίνακας 5.2 Pods προσομοίωσης 1

Αυτή είναι η πιο ήπια προσομοίωση μονάχα με 4 μικρά σχετικά Pods. Το αναμενόμενο θα ήταν να πάνε όλα στο μηχάνημα 1.

Για παράδειγμα πως οργανώνονται τα αρχεία της προσομοίωσης, θα παρατεθεί το yaml αρχείο της προσομοίωσης 1:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  namespace: experiment
```

```
  labels:
    app: whispers
    carbonIntensity: "0.48"
spec:
  schedulerName: my-default-scheduler
  containers:
  - name: container
    image: polinux/stress
    command: ["stress", "--cpu", "1", "--vm", "1", "--vm-bytes", "1500M", "--timeout",
"3600s"]
    resources:
      requests:
        memory: "1536Mi"
        cpu: "50m"
      limits:
        memory: "1536Mi"
        cpu: "50m"

---
apiVersion: v1
kind: Pod
metadata:
  name: pod-2
  namespace: experiment
  labels:
    app: whispers
    carbonIntensity: "0.62"
spec:
  schedulerName: my-default-scheduler
  containers:
  - name: container
    image: polinux/stress
    command: ["stress", "--cpu", "1", "--vm", "1", "--vm-bytes", "120M", "--timeout",
"3600s"]
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "250m"

---
apiVersion: v1
kind: Pod
metadata:
  name: pod-3
  namespace: experiment
  labels:
    app: whispers
```

```
    carbonIntensity: "0.88"
spec:
  schedulerName: my-default-scheduler
  containers:
  - name: container
    image: polinux/stress
    command: ["stress", "--cpu", "2", "--vm", "1", "--vm-bytes", "60M", "--timeout",
"3600s"]
    resources:
      requests:
        memory: "64Mi"
        cpu: "2000m"
      limits:
        memory: "64Mi"
        cpu: "2000m"

---
apiVersion: v1
kind: Pod
metadata:
  name: pod-4
  namespace: experiment
  labels:
    app: whispers
    carbonIntensity: "1.43"
spec:
  schedulerName: my-default-scheduler
  containers:
  - name: container
    image: polinux/stress
    command: ["stress", "--cpu", "1", "--vm", "1", "--vm-bytes", "60M", "--timeout",
"3600s"]
    resources:
      requests:
        memory: "64Mi"
        cpu: "50m"
      limits:
        memory: "64Mi"
        cpu: "50m"
```

Κώδικας 5.2 Αρχείο YAML για τη προσομοίωση 1

Όπως φαίνεται παραπάνω όλα τα pods είναι ίδια για λόγους ευκολίας. Αυτά που αλλάζουν είναι οι ονομασίες τους και τα requests τους. Επίσης, ανάλογα με τον χρονοδρομολογητή που εξετάζουμε, θα αλλάζει και το schedulerName στον αντίστοιχο χρονοδρομολογητή. Κυρίως θέλουμε να εξετάσουμε τη δέσμευση πόρων στο σύστημα και αν η δρομολόγηση είναι επιθυμητή.

Προσομοίωση 2 – Μέτρια-Ήπια

Pod Name	CPU	Memory	Carbon Intensity
pod-1	500m	512Mi	0.71
pod-2	200m	750Mi	1.26
pod-3	500m	256Mi	0.99
pod-4	500m	750Mi	1.65
pod-5	200m	1500Mi	0.87

Πίνακας 5.3 Pods προσομοίωσης 2

Αυτή η προσομοίωση είναι λίγο πιο απαιτητική και με ένα παραπάνω Pod από τη προηγούμενη. Επίσης αυξάνεται η μνήμη και η CPU που απαιτείται, καθώς και το Carbon Intensity. Χρησιμοποιεί την εντολή stress με τις απαιτήσεις για να εξετάσουμε τη κατανάλωση ενέργειας.

Προσομοίωση 3 – Μέτρια

Pod Name	CPU	Memory	Carbon Intensity
pod-1	200m	256Mi	1.48
pod-2	750m	128Mi	1.05
pod-3	1250m	2048Mi	1.42
pod-4	500m	2048Mi	1.02
pod-5	1500m	1024Mi	1.06
pod-6	1000m	128Mi	1.28

Πίνακας 5.4 Pods προσομοίωσης 3

Προσομοίωση 4 – Μέτρια-Έντονη

Pod Name	CPU	Memory	Carbon Intensity
pod-1	750m	1024Mi	1.42
pod-2	100m	128Mi	1.70
pod-3	500m	128Mi	1.81
pod-4	250m	512Mi	1.47
pod-5	750m	128Mi	1.52
pod-6	250m	1536Mi	1.46
pod-7	1000m	1024Mi	1.68
pod-8	2000m	1536Mi	1.45

Πίνακας 5.5 Pods προσομοίωσης 4

Προσομοίωση 5 – Έντονη

Parameter	Value
Number of Pods	30
CPU per Pod	600m
Memory per Pod	420Mi

Parameter	Value
Total CPU Requested	600m × 30 = 18,000m (~18 cores)
Total Memory Requested	420Mi × 30 = 12,600Mi (~12.3Gi)
Namespace	experiment
Carbon Intensity	Διαφέρει, από 0 έως 2

Πίνακας 5.6 Pods προσομοίωσης 5

Αυτή η προσομοίωση θα χρησιμοποιήσει το command sleep, καθώς το stress δεν θα το άντεχε το σύστημα μας. Αυτό που θέλουμε να δούμε σε αυτή τη προσομοίωση είναι σε ποιους κόμβους θα δρομολογήσει κάθε κόμβος τα pods. Γενικά οι απαιτήσεις είναι ιδιαιτέρως υψηλές για CPU

5.1.3 Profiles του χρονοδρομολογητή

Σε αυτό το κομμάτι θα παρουσιαστούν τα διάφορα profiles που θα τρέξουν στη προσομοίωση. Παρακάτω θα δείτε το KubeSchedulerConfiguration αρχείο και στη συνέχεια θα γίνει η ανάλυση των διαφορετικών profiles. Τα profiles που θα υπάρχουν είναι τα εξής:

- Προεπιλεγμένος Χρονοδρομολογητής
- Εξατομικευμένος ενεργειακά συνειδητός χρονοδρομολογητής με όλα τα προεπιλεγμένα plugins και έμφαση στα ενεργειακά μέσω των βαρών

Οι διαφορές θα φανούν μέσα στο configuration αρχείο. Τα παραπάνω είναι ο καλύτερος συνδυασμός που φαινόταν να έχει το καλύτερο επιθυμητό αποτέλεσμα. Ο τελικός σκοπός εξάλλου είναι να παρέχουμε ένα χρονοδρομολογητή που να μπορεί να εκτελέσει και τα 2, τον δικό μας χρονοδρομολογητή και τον προεπιλεγμένο, σε ικανοποιητικό βαθμό και κάθε χρήστης να επιλέγει ποιο θα χρησιμοποιεί, με προτίμηση για τον εξατομικευμένο ενεργειακό χρονοδρομολογητή.

kube-scheduler-config.yaml

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
profiles:
  - schedulerName: my-default-scheduler

  - schedulerName: my-scheduler
    plugins:
      score:
        disabled:
          - name: "*"
        enabled:
          - name: NodeResourcesBalancedAllocation
            weight: 1
          - name: ImageLocality
            weight: 2
          - name: NodeAffinity
            weight: 1
```

```
        - name: TaintToleration
          weight: 1
        - name: VolumeBinding
          weight: 1
        - name: InterPodAffinity
          weight: 1
        - name: VectorBinPacking
          weight: 1
        - name: ResourceVectorSimilarity
          weight: 1
        - name: EnergyAwareUsage
          weight: 2
        - name: EnergyAwareEATSVM
          weight: 3
        - name: DensePackingDPM
          weight: 3
    pluginConfig:
      - name: VectorBinPacking
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
      - name: ResourceVectorSimilarity
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
      - name: EnergyAwareUsage
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
      - name: EnergyAwareEATSVM
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
      - name: DensePackingDPM
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
```

Κώδικας 5.3 Configuration αρχείο του χρονοδρομολογητή

Με το παραπάνω αυτό που επιτυγχάνουμε είναι να δούμε αν θα έχουμε τα επιθυμητά αποτελέσματα από κάποιο από τα δύο profiles που έχουμε. Θα βγάλουμε τα αποτελέσματα όπως θα παρουσιαστούν παρακάτω στις μετρικές αξιολόγησης. Η προσέγγιση αυτή βασίζεται στην ιδέα πως θα θέλαμε να δώσουμε περισσότερη έμφαση στα ενεργειακά συνειδητά plugins που φτιάξαμε. Επιπροσθέτως είναι προτιμότερο να δοθεί περισσότερη βάση σε plugins που εξαρτώνται περισσότερο από τη κατάσταση του συστήματος.

5.1.4 Μετρικές Αξιολόγησης

Οι μετρικές με τις οποίες θα γίνει η αξιολόγηση είναι η τελική δρομολόγηση, η συνολική χρήση της CPU και της μνήμης, αλλά και η συνολική κατανάλωση ενέργειας.

Προκειμένου να έχουμε πρόσβαση από το τοπικό μηχάνημα στο Prometheus[5] θα κάνουμε το παρακάτω:

```
microk8s kubectl port-forward -n observability prometheus-kube-prom-stack-kube-prome-prometheus-0 9090

ssh -L 9090:localhost:9090 ubuntu@147.102.74.106
```

Κώδικας 5.4 Port-forward για τοπική πρόσβαση στις μετρικές του Prometheus

Για να πάρουμε τιμές σχετικά με την ενέργεια αντίστοιχα από το KEPLER[6] θα χρειαστεί να χρησιμοποιήσουμε τα κατάλληλα queries στο Prometheus, εφόσον μπορεί να δει τις μετρικές του KEPLER.

Μέσω της διεύθυνσης localhost:9090 μπορούμε να δούμε όλες τις μετρικές που θέλουμε και να βγάλουμε τα αποτελέσματα από τα πειράματα μας. Έτσι στο 5.2 κομμάτι θα βγουν τα αποτελέσματα από τα πειράματα με τη δρομολόγηση των pods και θα δούμε τον βέλτιστο χρονοδρομολογητή για ενεργειακή συνείδηση και πυκνή δρομολόγηση.

Τα queries που θα τρέξουμε κατά τη πειραματική μας αξιολόγηση είναι τα παρακάτω:

```
# Χρήση CPU

#1 Ποσοστό χρήσης της CPU σε κάθε κόμβο

100 * avg by (instance) (1 - rate(node_cpu_seconds_total{mode="idle"}[5m]))

#2 Πλήθος από πυρήνες (στο περίπου) που χρησιμοποιούνται ανά κόμβο

sum by (instance) (rate(node_cpu_seconds_total{mode!="idle"}[5m]))

#Μνήμη RAM

#1 Χρήση μνήμης RAM σε κάθε κόμβο

(node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes) / 1024^3

#2 Ελεύθερη μνήμη RAM ανά κόμβο

node_memory_MemAvailable_bytes / 1024^3

# Ενέργεια

#1 Ενέργεια που καταναλώθηκε από το CPU package ανά δευτερόλεπτο ανά κόμβο

sum by (instance) (rate(kepler_node_package_joules_total[5m]))

#2 Ενέργεια που καταναλώθηκε από το DRAM ανά δευτερόλεπτο ανά κόμβο

sum by (instance) (rate(kepler_node_dram_joules_total[5m]))

#3 Ενέργεια που καταναλώθηκε από όλες τις πλατφόρμες ανά δευτερόλεπτο ανά κόμβο

sum by (instance) (rate(kepler_node_platform_joules_total[5m]))
```

Κώδικας 5.5 Queries για υπολογισμό μετρικών στο Prometheus

Τα παραπάνω queries επιλέχθηκαν ώστε να έχουμε μια καλή εικόνα από το σύστημα και τη χρήση των διαφορετικών πόρων που γίνεται. Η μέτρηση γίνεται κυρίως σε τιμές των τελευταίων λεπτών, συνεπώς μετά από κάθε δρομολόγηση pods θα αναμένουμε 5 λεπτά έως ότου να τρέξουμε τα queries και να πάρουμε τις μετρήσεις. Με αυτό το τρόπο τα αποτελέσματα θα είναι πιο αξιόπιστα.

5.2 Πειραματική Αξιολόγηση Επίδοσης του Χρονοδρομολογητή

Σε αυτό το κομμάτι θα παρουσιαστούν τα αποτελέσματα από τα πειράματα για τα διαφορετικά Profiles καθώς και η αξιολόγηση τους. Πρώτα θα δοθούν στατιστικά για την αρχική κατάσταση του συστήματος.

5.2.1 Αρχική Κατάσταση Συστήματος

Προκειμένου να πάρουμε την αρχική κατάσταση των κόμβων στο σύστημα θα τρέξουμε την εντολή για να πάρουμε πληροφορίες για τα pods αλλά σε ολόκληρο το cluster. Στη συνέχεια θα δοθούν στατικά για τη χρήση cpu, μνήμης και ενέργειας σε κάθε ξεχωριστό κόμβο. Με αυτό το τρόπο θα καταφέρουμε να δούμε τις διαφορές στο σύστημα όταν θα δρομολογήσουμε τα pods με τους διαφορετικούς χρονοδρομολογητές.

Τα pods στο σύστημα:

NAMESPACE	NAME	NODE
Ingress	nginx-ingress-microk8s-controller	Node 1
Kepler	kepler-exporter	Node 1
Kube-system	calico-kube-controllers	Node 1
Kube-system	calico-node	Node 1
Kube-system	coredns	Node 1
Kube-system	dashboard-metrics-scraper	Node 1
Kube-system	hostpath-provisioner	Node 1
Kube-system	kubernetes-dashboard	Node 1
Kube-system	metrics-server	Node 1
Observability	alertmanager-kube-prom-stack-kube-prome-alertmanager-0	Node 1
Observability	kube-prom-stack-grafana	Node 1
Observability	kube-prom-stack-kube-prome-operator	Node 1
Observability	kube-prom-stack-kube-state-metrics	Node 1

Observability	kube-prom-stack-prometheus-node-exporter	Node 1
Observability	loki-0	Node 1
Observability	loki-promtail	Node 1
Observability	prometheus-kube-prom-stack-kube-prome-prometheus-0	Node 1
Observability	tempo-0	Node 1
Ingress	nginx-ingress-microk8s-controller	Node 2
Kepler	kepler-exporter	Node 2
Kube-system	calico-node	Node 2
Observability	kube-prom-stack-prometheus-node-exporter	Node 2
Observability	loki-promtail	Node 2
Ingress	nginx-ingress-microk8s-controller	Node 2
Kepler	kepler-exporter	Node 3
Kube-system	calico-node	Node 3
Kube-system	custom-scheduler	Node 3
Observability	kube-prom-stack-prometheus-node-exporter	Node 3
Observability	loki-promtail	Node 3

Πίνακας 5.7 Pods στο σύστημα κατά την αρχή των πειραμάτων

Όπως φαίνεται, ήδη υπάρχουν πολλά Pods που παίρνουν αρκετό χώρο από τους κόμβους. Συγκεκριμένα, στον κόμβο 1 υπάρχουν 18 Pods, στον κόμβο 2 υπάρχουν 6 Pods και στον κόμβο 3 υπάρχουν 7 Pods. Όσον αφορά την χρήση CPU και μνήμης έχουμε τα παρακάτω αποτελέσματα από το Prometheus.

Οι μετρικές που έχουν στην αρχική κατάσταση είναι:

Node	Used (cores)	Total (cores)	Used (%)	Free (cores)	Free (%)
2	0.4681	8	6.88%	7.5319	93.12%
3	0.4563	8	6.56%	7.5437	93.44%
1	0.7842	8	11.64%	7.2158	88.36%

Πίνακας 5.8 Χρήση CPU στο αρχικό σύστημα

Στη συνέχεια για τη μνήμη RAM, οι εντολές και τα αποτελέσματα:

Node	Used (GiB)	Total (GiB)	Used (%)	Free (GiB)	Free (%)
2	1.534	7.754	19.66%	6.223	80.34%
3	1.570	7.754	20.25%	6.198	79.75%
1	4.670	7.754	60.11%	3.100	39.89%

Πίνακας 5.9 Χρήση μνήμης RAM ανά κόμβο

Τέλος, από τον KEPLER θα πάρουμε την χρήση ενέργειας. Αντίστοιχα θα πάρουμε την ενέργεια που καταναλώνεται ανά κόμβο στα τελευταία 5 λεπτά μέσω του Prometheus:

Node	Ισχύς CPU (Package)	Ισχύς DRAM	Ισχύς πλατφόρμας	Σύνολο ανά κόμβο
1	148.0615	18.7334	84.0159	**250.8108**
2	147.2889	18.7120	80.3766	**246.3776**
3	147.2964	18.7120	80.3838	**246.3922**
Σύνολο (Όλοι οι κόμβοι)	**442.6468**	**56.1574**	**244.7764**	**743.5806**

Πίνακας 5.10 Κατανάλωση ενέργειας μέχρι τώρα ανά κόμβο βάσει του KEPLER

Συνεπώς θα παρατηρήσουμε για τις αυξήσεις στις παραπάνω τιμές ανάλογα με το εκάστοτε πείραμα και θα βγάλουμε τα συμπεράσματα μας. Θα συγκρίνουμε τις διαφορές που προκαλεί ο κάθε χρονοδρομολογητής και τις διαφορές μεταξύ τους. Αυτό που θέλουμε από το χρονοδρομολογητή μας είναι να επιλέγει τον κόμβο που είναι πιο ενεργειακά συνειδητός και να τον γεμίζει προτού να προχωράει στον επόμενο κόμβο.

5.2.2 Προσομοιώσεις και σχολιασμός

Σε αυτό το κομμάτι θα παρουσιάσουμε τα αποτελέσματα των προσομοιώσεων. Θα τρέξουμε μία φορά για κάθε profile με τις μετρικές που προαναφέρθηκαν. Θα τις παρουσιάσουμε σε σύγκριση με την αρχική κατάσταση και τον ένα χρονοδρομολογητή με τον άλλο ανά προσομοίωση.

Προσομοίωση 1 – Χαμηλής Έντασης Φόρτος

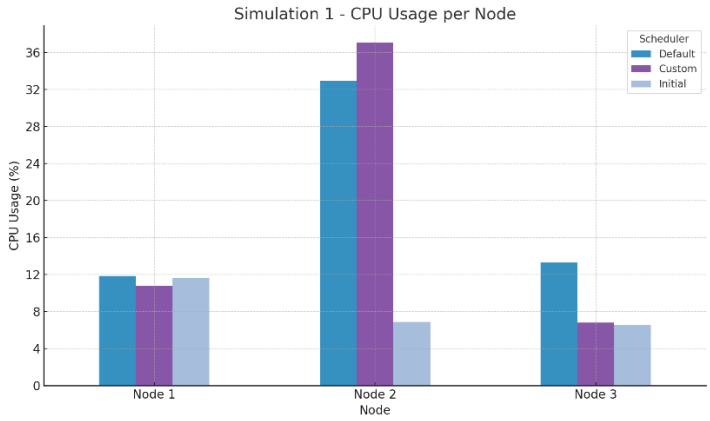
Κατανομή Pods σε κόμβους	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	0	3	1
Custom Scheduler	0	4	0

Πίνακας 5.11 Προσομοίωση 1 – Κατανομή Pods σε κόμβους

Ο προτεινόμενος μας χρονοδρομολογητής συσσώρευσε όλα τα pods στον κόμβο 2, όπως αναμενόταν από μία στρατηγική πυκνής δρομολόγησης. Εφόσον ο 2 είναι πιο ενεργειακά συμφέρων, είναι λογικό να τον προτιμάει το σύστημα, ακόμα κι αν δεν είναι ο πιο «γεμάτος» κόμβος. Ο προεπιλεγμένος χρονοδρομολογητής επιχειρεί και ισορροπεί το φορτίο στους 2 καλύτερους για αυτόν κόμβους, όπως αναμέναμε.

Ποσοστό χρήσης CPU	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	11.85%	32.92%	13.30%
Custom Scheduler	10.78%	37.07%	6.80%

Πίνακας 5.12 Προσομοίωση 1 – Χρήση CPU ανά κόμβο με διαφορετικό χρονοδρομολογητή

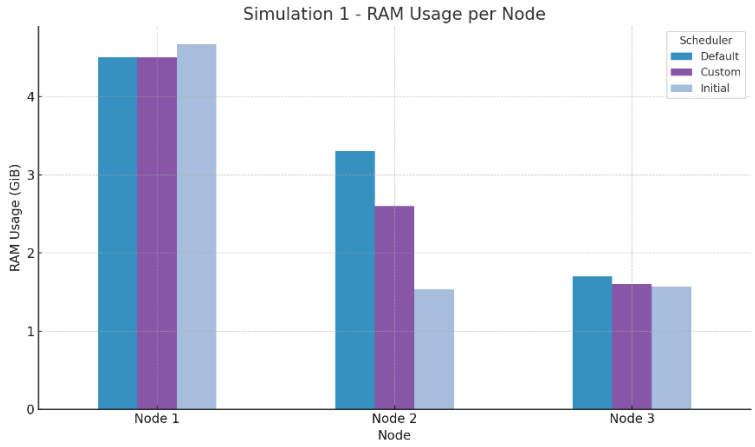


Εικόνα 5.1 Προσομοίωση 1 - Ποσοστό χρήσης CPU ανά κόμβο

Παρατηρούμε πως ο προεπιλεγμένος χρονοδρομολογητής μοιράζει την χρήση CPU σε 2 κόμβους, ενώ ο εξατομικευμένος την επικεντρώνει στον κόμβο 2. Συνολικά χρησιμοποιούν περίπου ίδιο ποσοστό CPU από το σύστημα, ωστόσο αφήνοντας τον κόμβο 3 αδρανή μας δίνει τη δυνατότητα να εφαρμόσουμε DVFS σε μελλοντική υλοποίηση.

Χρήση RAM	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	4.5GiB	3.3GiB	1.7GiB
Custom Scheduler	4.5GiB	2.6GiB	1.6GiB

Πίνακας 5.13 Προσομοίωση 1 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή



Εικόνα 5.2 Προσομοίωση 1 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως ο προεπιλεγμένος χρονοδρομολογητής καταναλώνει περίπου 1GB παραπάνω RAM από τον εξατομικευμένο. Αυτό συμβαίνει λόγω του διαμοιρασμού των pods. Στον εξατομικευμένο είναι όλα μαζί και έχουν τα απαραίτητα images ήδη κατεβασμένα σε ένα κόμβο, συνεπώς απαιτείται και λιγότερη μνήμη.

Κατανάλωση Watt	Κόμβος 1	Κόμβος 2	Κόμβος 3	Σύνολο
Default Scheduler	250.95	283.08	255.70	**787.73**
Custom Scheduler	250.07	288.31	246.64	**785.02**

Πίνακας 5.14 Προσομοίωση 1 – Κατανάλωση Watt ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως η αύξηση της κατανάλωσης Watt με τη προσθήκη Pods σε ένα κόμβο δεν είναι πολύ μεγάλη. Τα Watt στον κόμβο 2 και στις 2 περιπτώσεις είναι υψηλά, αρκετά υψηλότερα στην δεύτερη περίπτωση λόγω των περισσότερων Pods. Στον κόμβο 3 ωστόσο πάλι υπάρχει μια σημαντική αύξηση κατανάλωσης Watt στη πρώτη περίπτωση. Συνολικά, η ενέργεια που καταναλώνεται είναι περισσότερη στον προεπιλεγμένο χρονοδρομολογητή είναι ελάχιστα μεγαλύτερη, αλλά θα υπήρχε μεγάλη διαφορά με αδρανοποίηση του κόμβου 3 στη δεύτερη περίπτωση.

Συνολικά στη πρώτη προσομοίωση βλέπουμε πως ο εξατομικευμένος μας χρονοδρομολογητής είναι ελάχιστα πιο ενεργειακά συνειδητός και αποδοτικός. Οι διαφορές από μόνες τους δεν είναι αρκετές για να αιτιολογήσουν τη προτίμηση μας για αυτόν, αλλά διατηρώντας αδρανή τα μηχανήματα, μας επιτρέπει να εφαρμόσουμε DVFS σε μελλοντικές υλοποιήσεις και να μειώσουμε την κατανάλωση ενέργειας. Επιπροσθέτως, οι κόμβοι 1 και 2 είναι οι πιο παλιοί στο σύστημα, άρα και οι πιο σταθεροί, ενώ παράλληλα έχουμε υποθέσει πως είναι και αυτοί των οποίων η ενέργεια προέρχεται από ανανεώσιμες πηγές ενέργειας. Συνεπώς, συνολικά, ο χρονοδρομολογητής μας είναι ιδιαίτερα πιο ενεργειακά συνειδητός.

Προσομοίωση 2 – Μέτριος προς χαμηλό φόρτος

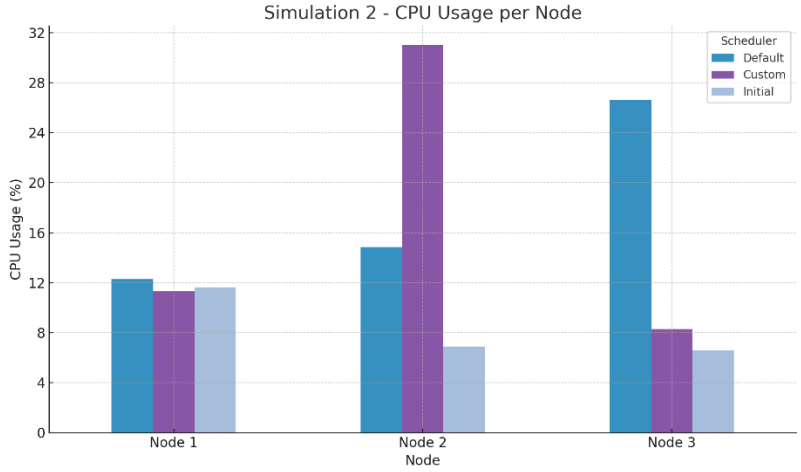
Κατανομή Pods σε κόμβους	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	0	2	3
Custom Scheduler	0	5	0

Πίνακας 5.15 Προσομοίωση 2 - κατανομή Pods σε κόμβους

Ο προτεινόμενος μας χρονοδρομολογητής συσσώρευσε όλα τα pods στον κόμβο 2, όπως αναμενόταν από μία στρατηγική πυκνής δρομολόγησης. Εφόσον ο 2 είναι πιο ενεργά συνειδητός και καλός, είναι λογικό να τον προτιμάει το σύστημα, ακόμα κι αν δεν είναι ο πιο «γεμάτος» κόμβος.

Ποσοστό χρήσης CPU	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	12.27%	14.84%	26.62%
Custom Scheduler	11.31%	31.01%	8.28%

Πίνακας 5.16 Προσομοίωση 2 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

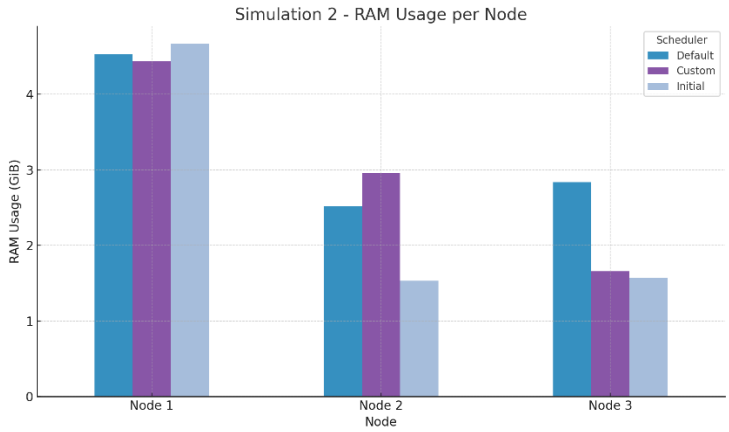


Εικόνα 5.3 Προσομοίωση 2 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως ο προεπιλεγμένος χρονοδρομολογητής έχει πιο ισορροπημένη χρήση του CPU, με τον όγκο να μοιράζεται στους κόμβου 2 και 3 κυρίως. Από την άλλη, ο εξατομικευμένος χρονοδρομολογητής επικεντρώνει όλη τη χρήση στον κόμβο 2, ενώ οι άλλοι 2 χρησιμοποιούνται ελάχιστα. Σε σύστημα με DVFS ενεργό θα ήταν σε κατάσταση αδράνειας, καταναλώνοντας ακόμα λιγότερη ενέργεια.

Χρήση RAM	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	4.53 GiB	2.52 GiB	2.84 GiB
Custom Scheduler	4.44 GiB	2.96 GiB	1.66 GiB

Πίνακας 5.17 Προσομοίωση 2 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή



Εικόνα 5.4 Προσομοίωση 2 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως ο προεπιλεγμένος χρονοδρομολογητής καταναλώνει περίπου 1GB παραπάνω RAM από τον εξατομικευμένο. Αυτό συμβαίνει λόγω του διαμοιρασμού των pods. Στον εξατομικευμένο είναι όλα μαζί και έχουν τα απαραίτητα images ήδη κατεβασμένα σε ένα κόμβο, συνεπώς απαιτείται και λιγότερη μνήμη.

Κατανάλωση Watt	Κόμβος 1	Κόμβος 2	Κόμβος 3	Σύνολο
Default Scheduler	250.37	256.17	273.1	**779.64**
Custom Scheduler	250.4	275.01	247.59	**773**

Πίνακας 5.18 Προσομοίωση 2 – Κατανάλωση Watt ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως η αύξηση της κατανάλωσης Watt με τη προσθήκη Pods σε ένα κόμβο δεν είναι πολύ μεγάλη. Τα Watt στον κόμβο 3 στη πρώτη περίπτωση είναι σχεδόν όσα στον κόμβο 2 στη δεύτερη περίπτωση, παρότι στη δεύτερη περίπτωση όλα τα Pods έχουν δρομολογηθεί στον κόμβο 2. Συνολικά παρατηρούμε πως με τον δικό μας δρομολογητή μειώνεται η κατανάλωση Watt κατά ένα μικρό ποσοστό των 1% περίπου. Ωστόσο, με χρήση του DVFS ενδέχεται να αδρανοποιούνταν ο κόμβος 3, συνεπώς θα μειωνόταν πολύ περισσότερο.

Συνολικά στη δεύτερη προσομοίωση βλέπουμε πως ο εξατομικευμένος μας χρονοδρομολογητής είναι ελάχιστα πιο ενεργειακά συνειδητός και αποδοτικός. Οι διαφορές από μόνες τους δεν είναι αρκετές για να αιτιολογήσουν τη προτίμηση μας για αυτόν, αλλά διατηρώντας αδρανή τα μηχανήματα, μας επιτρέπει να εφαρμόσουμε DVFS σε μελλοντικές υλοποιήσεις και να μειώσουμε την κατανάλωση ενέργειας.

Προσομοίωση 3 – Μέτριος φόρτος

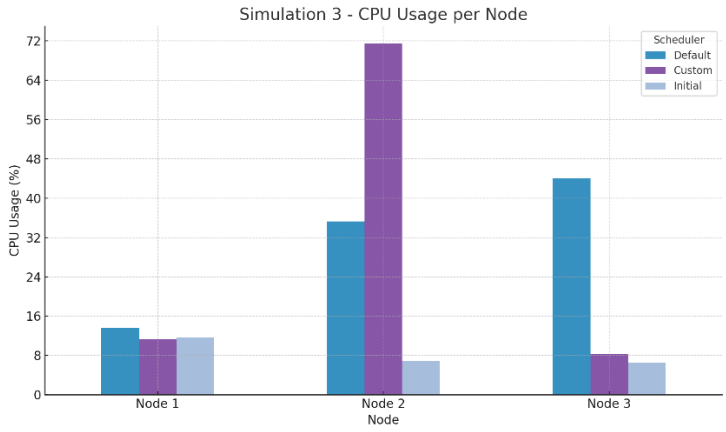
Κατανομή Pods σε κόμβους	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	0	3	3
Custom Scheduler	0	6	0

Πίνακας 5.19 Προσομοίωση 3 - κατανομή Pods σε κόμβους

Αντίστοιχα με τις παραπάνω προσομοιώσεις, στην δεύτερη περίπτωση δεν υπάρχει ισορροπημένη κατανομή, αλλά επικεντρωνόμαστε σε ένα κόμβο. Επίσης, όπως προηγουμένως, το θεωρούμε καλύτερο για την αδρανοποίηση των κόμβων σε μελλοντικές υλοποιήσεις με χρήση DVFS.

Ποσοστό χρήσης CPU	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	13.53%	35.30%	44.06%
Custom Scheduler	11.31%	71.47%	8.28%

Πίνακας 5.20 Προσομοίωση 3 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

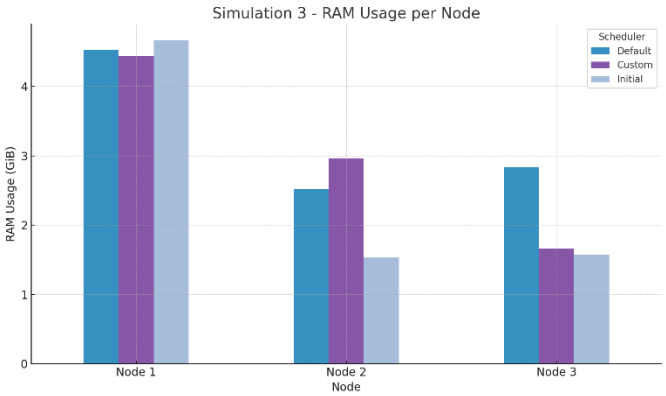


Εικόνα 5.5 Προσομοίωση 3 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Όπως και στις παραπάνω προσομοιώσεις τα ποσοστά είναι περίπου τα ίδια στο σύνολο και στις 2 περιπτώσεις. Στον προεπιλεγμένο χρονοδρομολογητή είναι απλά εξισορροπημένο σε 2 κόμβους, ενώ στον εξατομικευμένο είναι μαζεμένη η χρήση στον κόμβο 2, αλλά εκτελείται κανονικά.

Χρήση RAM	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	4.53 GiB	2.52 GiB	2.84 GiB
Custom Scheduler	4.44 GiB	2.96 GiB	1.66 GiB

Πίνακας 5.21 Προσομοίωση 3 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή



Εικόνα 5.6 Προσομοίωση 3 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε αντίστοιχα αποτελέσματα με τις παραπάνω προσομοιώσεις. Ο εξατομικευμένος χρονοδρομολογητής χρησιμοποιεί περίπου 1GB λιγότερη μνήμη RAM σε σύγκριση με τον προεπιλεγμένο.

Κατανάλωση Watt	Κόμβος 1	Κόμβος 2	Κόμβος 3	Σύνολο
Default Scheduler	250.52	289.36	298.03	**837.91**
Custom Scheduler	250.04	333.93	248.83	**832.80**

Πίνακας 5.22 Προσομοίωση 3 – Κατανάλωση Watt ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Αντίστοιχα με τις προηγούμενες προσομοιώσεις έχουμε μια διαφορά στην κατανάλωση Watt περίπου 1%. Δεν είναι σημαντική από μόνη της, αλλά στην περίπτωση με τον εξατομικευμένο χρονοδρομολογητή θα μπορούσαμε αντίστοιχα να αδρανοποιήσουμε τον κόμβο 3.

Συνολικά και στην τρίτη προσομοίωση βλέπουμε τα ίδια χαρακτηριστικά στο σύστημα όπως και στις 2 πρώτες. Αυτό επιβεβαιώνει πως ο ενεργειακά συνειδητός χρονοδρομολογητής μας λειτουργεί με την λογική της πυκνής δρομολόγησης και με μελλοντική εφαρμογή ενός DVFS controller στο σύστημα.

Προσομοίωση 4 – Μέτριος προς υψηλό φόρτος

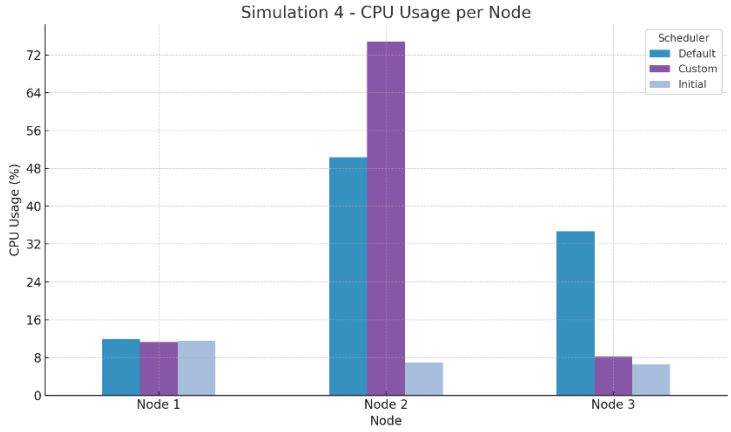
Κατανομή Pods σε κόμβους	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	0	5	3
Custom Scheduler	0	8	0

Πίνακας 5.23 Προσομοίωση 4 - κατανομή Pods σε κόμβους

Τα σχόλια είναι ίδια με τις προηγούμενες προσομοιώσεις, ίδια συμπεριφορά από τους χρονοδρομολογητές.

Ποσοστό χρήσης CPU	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	11.92%	50.39%	34.68%
Custom Scheduler	11.29%	74.76%	8.24%

Πίνακας 5.24 Προσομοίωση 4 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

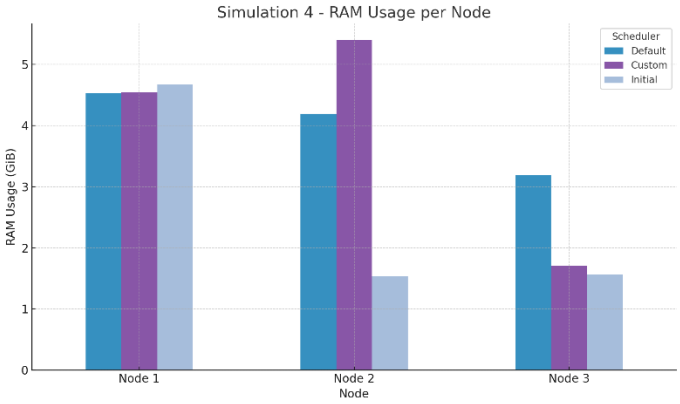


Εικόνα 5.7 Προσομοίωση 4 – Ποσοστό χρήσης CPU ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε πως σε αυτή τη προσομοίωση συνολικά χρησιμοποιείται λίγη περισσότερη CPU κατά την περίπτωση που χρησιμοποιούμε τον προεπιλεγμένο χρονοδρομολογητή, κατά 2%. Παρότι δεν είναι σημαντική η διαφορά, με τον δικό μας χρονοδρομολογητή δρομολογούνται όλα τα Pods στον ίδιο κόμβο, επιτρέποντας μας να είμαστε πιο ευέλικτοι με το υπόλοιπο σύστημα

Χρήση RAM	Κόμβος 1	Κόμβος 2	Κόμβος 3
Default Scheduler	4.53 GiB	4.19 GiB	3.19 GiB
Custom Scheduler	4.54 GiB	5.4 GiB	1.71 GiB

Πίνακας 5.25 Προσομοίωση 4 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή



Εικόνα 5.8 Προσομοίωση 4 – Χρήση RAM ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Αυτή τη φορά η διαφορά στη χρήση RAM είναι μικρότερη από τις προηγούμενες προσομοιώσεις. Παρατηρούμε πως όσο περισσότερα Pods υπάρχουν στο σύστημα, τόσο πιο πιθανό να μην έχουμε συνολική μείωση στη χρήση RAM λόγω των λιγότερων images.

Κατανάλωση Watt	Κόμβος 1	Κόμβος 2	Κόμβος 3	Σύνολο
Default Scheduler	250.31	284.33	308.92	**843.56**
Custom Scheduler	250.30	339.83	247.85	**838.06**

Πίνακας 5.26 Προσομοίωση 4 – Κατανάλωση Watt ανά κόμβο με χρήση διαφορετικού χρονοδρομολογητή

Παρατηρούμε αντίστοιχα και σε αυτή τη προσομοίωση πως η κατανάλωση watt είναι μικρότερη στην περίπτωση του εξατομικευμένου χρονοδρομολογητή, αλλά κατά λίγο λιγότερο πάλι.

Σε αυτή τη προσομοίωση παρατηρούμε μερικές μικροαλλαγές σε σύγκριση με τις προηγούμενες. Συνολικά στη δεύτερη περίπτωση χρησιμοποιούνται λιγότεροι πόροι, αλλά η διαφορά στη χρήση είναι πολύ μικρή. Όπως έχει αναφερθεί ήδη, εφόσον ο χρονοδρομολογητής μας προσπαθεί να διατηρήσει τους ήδη αδρανείς κόμβους αδρανείς, είναι εύκολο να διαχειριστούμε την κατανάλωση ενέργειας τους με ιδέες όπως το DVFS. Συνεπώς, και σε αυτή τη προσομοίωση, ο ενεργειακά συνειδητός χρονοδρομολογητής φαίνεται να είναι καλύτερος.

Προσομοίωση 5 – Υψηλός φόρτος

Σε αυτή τη προσομοίωση θα συγκρίνουμε τη γενικότερη σειρά προτίμησης για την δρομολόγηση των pods από τους χρονοδρομολογητές. Η δρομολόγηση έγινε ως παρακάτω, με σειρά από πάνω προς τα κάτω:

Default Scheduler:

#	Pod	Node	#	Pod	Node	#	Pod	Node	#	Pod	Node	#	Pod	Node
1	pod-0	3	**7**	pod-8	3	**13**	pod-14	2	**19**	pod-20	2	**25**	pod-26	1
2	pod-1	2	**8**	pod-9	2	**14**	pod-15	1	**20**	pod-21	3	**26**	pod-27	2
3	pod-2	3	**9**	pod-10	2	**15**	pod-16	3	**21**	pod-22	3	**27**	pod-28	2
4	pod-3	2	**10**	pod-11	3	**16**	pod-17	2	**22**	pod-23	2	**28**	pod-29	1
5	pod-4	3	**11**	pod-12	1	**17**	pod-18	1	**23**	pod-24	1	**29**	pod-6	1
6	pod-5	2	**12**	pod-13	3	**18**	pod-19	3	**24**	pod-25	3	**30**	pod-7	3

Πίνακας 5.27 Προσομοίωση 5 – Σειρά δρομολόγησης με default scheduler

Custom Scheduler:

#	Pod	Node	#	Pod	Node	#	Pod	Node	#	Pod	Node	#	Pod	Node
1	pod-0	2	7	pod-6	2	13	pod-16	3	19	pod-24	3	25	pod-11	3
2	pod-1	2	8	pod-8	2	14	pod-18	3	20	pod-25	3	26	pod-17	1
3	pod-2	2	9	pod-10	2	15	pod-19	3	21	pod-26	3	27	pod-28	1
4	pod-3	2	10	pod-13	2	16	pod-20	3	22	pod-29	3	28	pod-12	1
5	pod-4	2	11	pod-14	2	17	pod-21	3	23	pod-7	3	29	pod-27	1
6	pod-5	2	12	pod-15	2	18	pod-22	3	24	pod-9	1	30	pod-23	1

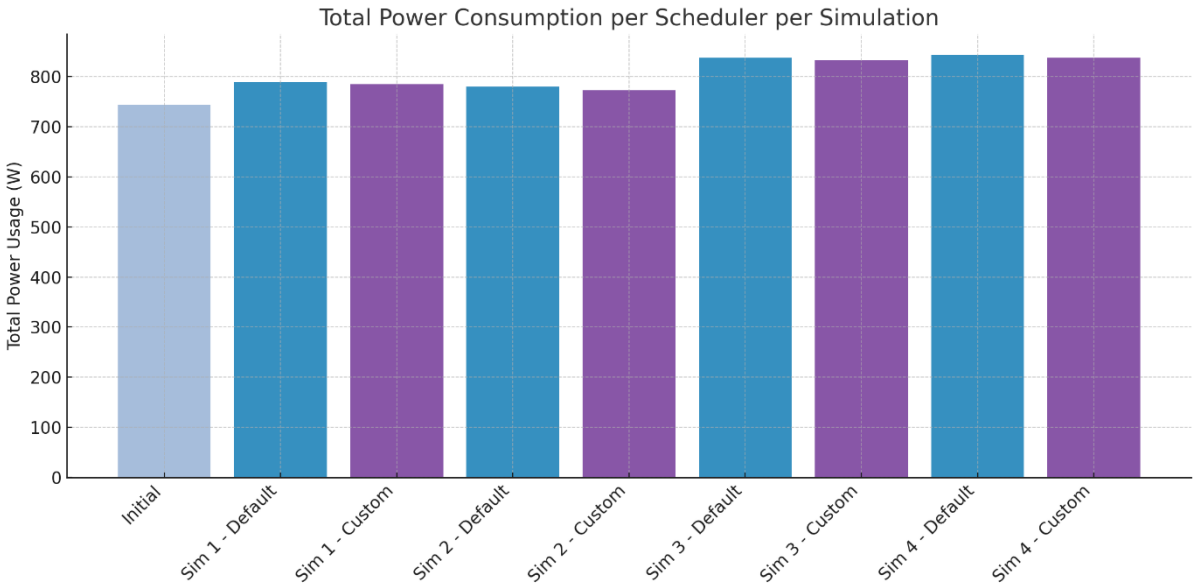
Πίνακας 5.28 Προσομοίωση 5 – Σειρά δρομολόγησης με τον custom scheduler

Αυτό που παρατηρείται είναι πως:

- Με την εφαρμογή του προεπιλεγμένου χρονοδρομολογητή, το σύστημα αναθέτει πόρους μία σε ένα κόμβο, μία σε ένα άλλο. Με αυτό το τρόπο κρατάει το σύστημα σε ισορροπία και δεν επιβαρύνει κάποιον κόμβο περισσότερο από κάποιον άλλο.
- Από την άλλη, με τον ενεργειακά συνειδητό χρονοδρομολογητή δίνεται προτεραιότητα σε ένα κόμβο τη φορά: πρώτα δίνεται στον κόμβο 2, όπως και στις προηγούμενες προσομοιώσεις, στη συνέχεια στον κόμβο 3 και τέλος στον κόμβο 1. Αυτό μας δείχνει

πως πάντα, αν υπάρχει ένας κόμβος που είναι επιθυμητός, θα συνεχίσει να είναι επιθυμητός και στις αμέσως επόμενες δρομολογήσεις, έως ότου να μην χωράει άλλα Pods.

Διάγραμμα συνολικής κατανάλωσης ενέργειας



Εικόνα 5.9 Συνολική κατανάλωση ενέργειας (Watt) σε κάθε προσομοίωση ανά δρομολογητή

Όπως παρατηρούμε από την ενέργεια, οι διαφορές δεν είναι ιδιαίτερα μεγάλες ούτε μεταξύ των προσομοιώσεων, ούτε μεταξύ των χρονοδρομολογητών. Πάντα ο δικός μας ωστόσο είναι ελάχιστα πιο κάτω από τον προεπιλεγμένο, ενώ με τις κατάλληλες προεκτάσεις η διαφορά θα διευρυνόταν ακόμα περισσότερο.

Συνεπώς συμπεραίνουμε πως ο χρονοδρομολογητής μας λειτουργεί όπως επιθυμούμε. Προτιμάει να δρομολογεί πιο πυκνά τα pods, αφήνοντας τη δυνατότητα για αδρανοποίηση των υπόλοιπων κόμβων προτού δρομολογηθούν. Παράλληλα, αυτό το κάνει στον κόμβο 2 που θεωρούμε εμείς τον πιο ενεργά συμφέρων κόμβο εκ των $3^{ων}$. Συνεπώς, και σε αυτή και σε όλες τις προσομοιώσεις τα αποτελέσματα είναι αυτά που αναμέναμε και θετικά για τον χρονοδρομολογητή μας.

6. Τελικά Συμπεράσματα

Σε αυτό το κεφάλαιο γίνεται η τελική αποτίμηση της επίδοσης του χρονοδρομολογητή, παρουσιάζονται τα τελικά συμπεράσματα της ανάλυσης και εξετάζονται πιθανές μελλοντικές κατευθύνσεις για τη περαιτέρω βελτίωση του.

6.1 Συνολική Αξιολόγηση της Υλοποίησης

Η συνολική αξιολόγηση της υλοποίησης βασίζεται στα αποτελέσματα πέντε διαφορετικών προσομοιώσεων, καθεμία από τις οποίες αποκάλυψε τον τρόπο λειτουργίας του εξατομικευμένου ενεργειακά συνειδητού χρονοδρομολογητή. Μέσα από συγκριτικές μελέτες με τον προεπιλεγμένο χρονοδρομολογητή του Kubernetes, προκύπτει ένα σαφές μοτίβο που δικαιολογεί την βελτίωση της δικής μας λύσης ως προς την κατανάλωση ενέργειας κατά μεγάλο βαθμό, και τη διαχείριση των πόρων και την αποδοτικότητα συστήματος κατά μικρότερο βαθμό.

Πυκνή Δρομολόγηση και Ενεργειακή Συνείδηση

Σε όλες τις προσομοιώσεις, ο χρονοδρομολογητής μας επέλεξε να συγκεντρώνει τα Pods σε έναν συγκεκριμένο κόμβο, τον κόμβο 2. Αυτή η επιλογή δεν έγινε τυχαία. Αντίθετα, βασίζεται στο γεγονός πως ο κόμβος 2 θεωρείται πιο σταθερός, πιο ενεργειακά συμφέρων και θεωρήσαμε πως τροφοδοτείται από ανανεώσιμες πηγές ενέργειας και παράγει μικρότερες ποσότητες άνθρακα. Παράλληλα, έχει λιγότερους κύκλους ρολογιού[10] από τον κόμβο 1, που είναι ο δεύτερος πιο ενεργειακά συμφέρων, επομένως θα παράγει ακόμα λιγότερη ενέργεια αν η δρομολόγηση γινόταν σε αυτόν. Συνεπώς, το σύστημα μας φαίνεται να προτιμάει τους πιο οικολογικούς, σταθερούς κόμβους.

Η στρατηγική της πυκνής δρομολόγησης επιτρέπει στο σύστημα να αφήνει τους υπόλοιπους κόμβους σε κατάσταση αδράνειας[9], [11], [21], γεγονός που καθιστά εφικτή την εφαρμογή μεθόδων εξοικονόμησης ενέργειας, όπως το DVFS. Με ένα κατάλληλο controller, επιπλέον του χρονοδρομολογητή, ένας διαχειριστής θα μπορούσε να αδρανοποιεί τους κόμβους που δεν έχουν φορτία πέρα των απαραίτητων για τη λειτουργία τους. Συνεπώς θα μειωνόταν αισθητά η κατανάλωση ενέργειας και θα αξιοποιούνταν πρώτα αρκετοί πόροι από ένα μηχάνημα προτού προχωρήσει σε επόμενο.

Κατανάλωση Πόρων

Στο σύνολο των προσομοιώσεων, η χρήση της CPU και της RAM παρουσιάζει συγκεκριμένες διαφοροποιήσεις:

- **Χρήση CPU**: Ο εξατομικευμένος χρονοδρομολογητής επιβαρύνει κυρίως έναν κόμβο, επιτρέποντας στους υπόλοιπους να διατηρούν χαμηλό φορτίο. Αντίθετα, ο προεπιλεγμένος scheduler διαμοιράζει τον φόρτο, επιβαρύνοντας περισσότερους κόμβους χωρίς ουσιαστικό όφελος στην απόδοση. Οι συνολικές διαφορές στο σύστημα δεν είναι σημαντικές.

- **Χρήση RAM**: Ο δικός μας χρονοδρομολογητής, λόγω της συγκέντρωσης των Pods, επιτυγχάνει μείωση στη συνολική απαίτηση και χρήση μνήμης, καθώς τα απαραίτητα container images χρειάζεται να φορτωθούν σε λιγότερους κόμβους.

Αυτό μεταφράζεται σε εξοικονόμηση πόρων, πιο εστιασμένη αξιοποίηση των υποδομών και μελλοντική δυνατότητα για έξυπνη αδρανοποίηση ή ενεργοποίηση κόμβων.

Κατανάλωση Ενέργειας (Watt)

Από την άποψη της κατανάλωσης ενέργειας, παρατηρείται ότι:

- Σε κάθε προσομοίωση, η συνολική κατανάλωση Watt ήταν ελαφρώς μικρότερη με τον εξατομικευμένο χρονοδρομολογητή. Αν και η διαφορά κυμαίνεται γύρω στο 1%, αυτή αποκτά ιδιαίτερη σημασία σε μεγάλης κλίμακας συστήματα. Επιπροσθέτως, αξίζει να σημειωθεί πως ο τρόπος που λαμβάνουμε αυτές τις μετρικές είναι μέσω της χρήσης μνήμης και CPU. Το εκπαιδευμένο μοντέλο του KEPLER πιθανολογεί πόση θα ήταν η κατανάλωση βάσει δεδομένων και προσομοιώσεων από εκτός δικτύου πειράματα. Συνεπώς, εφόσον δεν υπήρχαν ιδιαίτερες διαφορές στην χρήση CPU και μνήμης, δεν αναμέναμε ιδιαίτερες διαφορές στην κατανάλωση watt.
- Επιπλέον, το γεγονός ότι οι υπόλοιποι κόμβοι μένουν σε χαμηλή κατανάλωση ή αδράνεια επιτρέπει την εφαρμογή τεχνικών όπως DPM, DVFS, ακόμα και shutdown κόμβων, εφόσον προβλεφθούν κατάλληλες πολιτικές ελέγχου. Τα μηχανήματα που μένουν ενεργά για μεγαλύτερο χρονικό διάστημα είναι πιο επιθυμητά, ενώ τα υπόλοιπα μπορούμε να τα αδρανοποιήσουμε.

Στρατηγική Δρομολόγησης Pods

Η πιο σημαντική ίσως παρατήρηση έρχεται από την Προσομοίωση 5, όπου παρακολουθούμε αναλυτικά τη σειρά με την οποία δρομολογούνται τα Pods όταν φτάνουμε το σύστημα στα θεωρητικά του όρια.

- Ο προεπιλεγμένος χρονοδρομολογητής λειτουργεί με τη λογική της ισοκατανομής, επιλέγοντας κόμβους εναλλάξ χωρίς να λαμβάνει υπόψη το ενεργειακό κόστος ή την κατάσταση άλλων Pods.
- Ο χρονοδρομολογητής μας επιλέγει πρώτα τον πιο ενεργειακά "ευνοϊκό" κόμβο, εξαντλεί τα περιθώρια του και έπειτα προχωρά στον επόμενο. Αυτή η συμπεριφορά υποδηλώνει συνέπεια με την αρχική φιλοσοφία μας: πρώτα αξιοποίηση του πιο αποδοτικού και ενεργειακά συνειδητού κόμβου και αργότερα διάχυση φόρτου.

Αυτό το χαρακτηριστικό, που προκύπτει έμπρακτα από τις μετρήσεις και τις αναθέσεις Pods, αποτελεί απόδειξη της εστιασμένης και συνειδητής ενεργειακά λειτουργίας του χρονοδρομολογητή μας. Είναι το αποτέλεσμα το οποίο θέλαμε να έχει ο χρονοδρομολογητής μας, και επιτεύγχθηκε.

Συμπεράσματα

Συνολικά, από τις πέντε προσομοιώσεις προκύπτει ότι:

- Ο ενεργειακά συνειδητός χρονοδρομολογητής μας επιτυγχάνει ελάχιστα χαμηλότερη κατανάλωση ενέργειας.
- Συγκεντρώνει τους φόρτους, επιτρέποντας έξυπνη ενεργειακή διαχείριση.
- Μειώνει τις ανάγκες σε RAM και απλοποιεί τη μεταφορά/φόρτωση των images για τα containers.
- Επιτρέπει την εφαρμογή μελλοντικών τεχνικών DVFS/DPM.
- Διατηρεί την επιθυμητή λειτουργικότητα και δεν θυσιάζει την απόδοση.

Ολοκληρώνοντας, η υλοποίηση του χρονοδρομολογητή μας απέδειξε πως είναι εφικτό να συνδυάσουμε την έξυπνη διαχείριση υποδομής με τη βιώσιμη λειτουργία ενός συστήματος Kubernetes. Χωρίς να αλλάξει ο τρόπος που τα Pods εκτελούνται, μετατοπίσαμε τη στρατηγική τοποθέτησης ώστε να συμβαδίζει με τις ανάγκες ενός ενεργειακά αποδοτικού cloud περιβάλλοντος. Οπότε ο χρονοδρομολογητής μας ακολουθεί όντως τις τεχνικές πυκνής δρομολόγησης και είναι αρκετά ενεργειακά συνειδητός. Η προτίμηση των μηχανημάτων που είναι πιο οικολογικά είναι ιδιαιτέρως σημαντική και προωθεί την ενεργειακή συνείδηση, ενώ παράλληλα με άλλες έξυπνες τεχνικές μπορεί να συνδυαστεί και να μειώσει κατά πολύ τη άσκοπη κατανάλωση ενέργειας.

6.2 Μελλοντικές προεκτάσεις

Σε αυτό το κομμάτι θα παρουσιαστούν ιδέες που δεν υλοποιήθηκαν είτε λόγω έλλειψης χρόνου είτε λόγω ανεπάρκειας δεδομένων. Η ανάλυση δεν θα είναι εκτενής, αλλά περισσότερο ως μια απόθεση ιδεών για τη περίπτωση που κάποιος επιθυμεί σε μετέπειτα χρόνο να εξελίξει περαιτέρω τον χρονοδρομολογητή.

6.2.1 Δυναμική Κλιμάκωση Τάσης και Συχνότητας (DVFS)

Η πιο σημαντική τεχνική που θα μπορούσε να αξιοποιηθεί είναι DVFS[9], [10]. Η βάση του είναι η δυναμική ρύθμιση της τάσης και της συχνότητας στη λειτουργία ενός μηχανήματος. Σε συνδυασμό με το σύστημα μας, θα μπορούσε να αξιοποιήσει τη βάση της πυκνής δρομολόγησης. Εφόσον ο χρονοδρομολογητής μας τείνει να αφήνει συγκεκριμένους κόμβους αδρανείς, το επόμενο βήμα θα ήταν η αυτόματη μετάβαση των κόμβων αυτών σε κατάσταση μειωμένης συχνότητας.

Μια πιθανή υλοποίηση θα ήταν με την υλοποίηση ενός controller στη γλώσσα GO. Αυτό ο controller θεωρητικά θα ελέγχει ποιοι κόμβοι τρέχουν μονάχα τα απαραίτητα για τη λειτουργία τους Pods και θα τους βάζει σε μία λίστα, ή θα τους μειώνει τη συχνότητα αυτόματα, ανεβάζοντας την όταν επιλέγονται για δρομολόγηση. Εργαλεία που τη στιγμή της συγγραφής φαίνονται να συνεισφέρουν στην παραπάνω υλοποίηση είναι το cpufreq και το intel_pstate, τα οποία είναι συστήματα της Linux και της Intel αντίστοιχα για την δυναμική ρύθμιση της συχνότητας, με το δεύτερο να εφαρμόζει πιο αποδοτικούς αλγορίθμους. Διαφορετικά, θα

μπορούσε να αξιοποιήσει ένα ενεργειακά συνειδητό agent όπως το Kepler. Με την εφαρμογή αυτή θα επιτυγχάνεται περαιτέρω μείωση κατανάλωσης ενέργειας σε πραγματικό χρόνο.

6.2.2 Γενετικοί αλγόριθμοι για υπολογισμό βαρών

Στην παρούσα υλοποίηση, τα βάρη των plugins επιλέγονται στατικά. Ένας λογικός επόμενος στόχος θα ήταν η χρήση γενετικών αλγορίθμων[13], [14], [15] για αυτόματη εξεύρεση του βέλτιστου συνδυασμού βαρών. Με μια διαδικασία η οποία βασίζεται σε fitness συναρτήσεις, όπως για παράδειγμα η ενεργειακή κατανάλωση και η ισορροπία ή μη της CPU, το σύστημα μπορούσε να εκπαιδεύεται προκειμένου να παράγει διαφορετικά βάρη ανάλογα με τα χαρακτηριστικά των υπάρχοντών κόμβων και pods.

Μια πιθανή υλοποίηση θα ήταν με χρήση της Python, ξεχωριστή από το cluster, με τη βοήθεια ενός εργαλείου που διαβάζει τα logs από το Prometheus ή και το Kepler, ακόμα και τα scores από τον χρονοδρομολογητή, και τα αλλάζει αντίστοιχα. Παράλληλα θα υλοποιηθεί ένας controller στη γλώσσα GO μέσα στο cluster ο οποίος θα αλλάζει δυναμικά τα βάρη του scheduler configuration και θα επαναδημιουργεί το config map κατάλληλα ώστε να αλλάζει το configuration.

6.2.3 Επαναδρομολόγηση (Rescheduling Mechanism)

Αν και η αρχική δρομολόγηση είναι αποδοτική, σε συστήματα με μακροχρόνια φόρτιση ή αργές μεταβολές στη χρήση, η επαναδρομολόγηση Pods θα μπορούσε να ενισχύσει τη βιωσιμότητα. Ένας Rescheduler Controller θα μπορούσε να παρακολουθεί τους κόμβους και να εντοπίζει περιπτώσεις στις οποίες:

- Ένας κόμβος έχει πολύ χαμηλή χρήση για μεγάλο χρονικό διάστημα,
- Ενδεχομένως να υπάρχει η δυνατότητα να στείλουμε τα Pods ενός κόμβου σε άλλο ώστε να απελευθερωθεί τελείως και να τον αδρανοποιεί το σύστημα αυτόματα.

Για την υλοποίηση του θα χρειαστεί να χρησιμοποιηθεί ο descheduler[25]. Στο σύστημα του Kubernetes τα Pods μπορούν να δρομολογηθούν μονάχα σε ένα κόμβο, και αυτό δεν μπορεί να αλλάξει εκτός αν διαγραφεί εντελώς το Pod και επαναδρομολογηθεί. Για τη διευκόλυνση των χρηστών σε αυτό το κομμάτι έρχεται ο Descheduler, τον οποίο θα προσθέσουμε μαζί με δικό μας Controller, και με βάση μετρικές περί ενέργειας και αξιοπιστίας, ή τις περιπτώσεις παραπάνω για παράδειγμα, θα χρησιμοποιεί τον Descheduler για την επαναδρομολόγηση ενός ή πολλαπλών Pods.

Ι. Παράρτημα 1

Υλοποίηση ενός kube-scheduler

Σε αυτό το παράρτημα θα παρουσιαστεί η μέθοδος που χρησιμοποιήθηκε προκειμένου να φτιάξουμε τον kube-scheduler αναλυτικά, ώστε να μπορεί να χρησιμοποιηθεί για περαιτέρω υλοποίηση. Πρώτα θα παρουσιαστεί ο τρόπος με τον οποίο προετοιμάζουμε το σύστημα μας, πως ορίζουμε τα plugins και τι πρέπει να θέσουμε και που προκειμένου να δουλέψει στο τέλος η εντολή make.

Πρώτον, προκειμένου να φτιάξουμε ένα kube-scheduler για το Kubernetes χρειαζόμαστε τα ήδη υπάρχοντα αρχεία για να το φτιάξουμε. Βρέθηκαν 2 τρόποι που θα μπορούσαμε να φτιάξουμε τον χρονοδρομολογητή. Ο πρώτος είναι μέσω του Github Repository του Kubernetes, το οποίο θα χρειάζεται να έχουμε σε ένα από τα μηχανήματα μας. Ο δεύτερος τρόπος είναι να κατεβάσουμε ένα αρχείο που είναι αποκλειστικά για τον χρονοδρομολογητή του Kubernetes. Μετά από αναζήτηση για αρχεία καταλάβαμε πως δεν υπάρχουν τα προεπιλεγμένα plugins όπως τα περιμέναμε με βάση την βιβλιογραφία του Kubernetes. Συνεπώς, προκειμένου να αξιοποιήσουμε τις γνώσεις που είχαν συλλεχθεί, επιλέχθηκε η επιλογή με το Github repository του Kubernetes.

Αυτό επιτυγχάνεται με τη παρακάτω εντολή:

```
# Θεωρούμε πως ξεκινάμε από το ~/
git clone https://github.com/kubernetes/kubernetes.git

cd kubernetes
```

Κώδικας Ι.1 Εγκατάσταση του Github Repository του Kubernetes

Τώρα μπορούμε να εισάγουμε τα plugins μας στο σύστημα. Όλα τα plugins του kube-scheduler βρίσκονται στον φάκελο pkg/scheduler/framework/plugins. Σε αυτό το φάκελο πρέπει να φτιάξουμε καινούριους φακέλους για τα δικά μας plugins και να βάλουμε τον απαραίτητο κώδικα σε αυτά. Για παράδειγμα, αν θέλουμε να βάλουμε το vectorBinPacking plugin θα το κάναμε ως εξής:

```
mkdir pkg/scheduler/framework/plugins/vectorbinpacking

nano pkg/scheduler/framework/plugins/vectorbinpacking/vectorbinpacking.go
```

Κώδικας Ι.2 Δημιουργία των κατάλληλων αρχείων για το plugin

Και στο αρχείο αυτό βάζουμε τον κώδικα όπως στο υποκεφάλαιο 4.2.1. Στη συνέχεια χρειάζεται να προσθέσουμε το plugin μας κατάλληλα στο αρχείο που καταγράφονται όλα τα plugins. Αυτό το αρχείο είναι το pkg/scheduler/framework/plugins/registry.go . Σε αυτό το αρχείο πρέπει να προσθέσουμε τα παρακάτω:

```
# Προστίθεται στα αρχικά import στην αρχή του registry.go. Προστίθεται με το απαραίτητο κενό στην αρχή
```

```
import (
…
…
…
 vectorbinpacking "k8s.io/Kubernetes/pkg/scheduler/framework/plugins/vectorbinpacking"
)


# Στο NewInTreeRegistry(), προστίθεται ένα καινούριο instance του plugin
registry := runtime.Registry{
…
…
…
        vectorbinpacking.Name:              vectorbinpacking.New,
}
```

Κώδικας Ι.3 Εισαγωγή του vectorBinPacking plugin στο registry.go

Επαναλαμβάνουμε την ίδια διαδικασία για κάθε plugin. Όταν προσθέσουμε όλα τα plugins στους κατάλληλους φακέλους και στο registry.go, τότε μπορούμε να φτιάξουμε τον kube-scheduler μας. Αυτό γίνεται με τη χρήση ενός Makefile που παρέχεται από το ίδιο το Kubernetes. Ωστόσο, προκειμένου να μην χρειαστεί πολλή ώρα, και να μην παράγουμε υπερμεγέθη αρχεία, θα χρησιμοποιήσουμε την εντολή make ως φαίνεται παρακάτω για να δημιουργήσουμε μονάχα το αρχείο kube-scheduler:

```
make WHAT=cmd/kube-scheduler
```

Κώδικας Ι.4 make εντολή για τον kube-scheduler

Η ίδια εντολή μπορεί να χρησιμοποιηθεί και για debugging. Στη περίπτωση που επιτύχει, ο καινούριος kube-scheduler θα βρίσκεται μέσα στα αρχεία που κατεβάσαμε από το Kubernetes, στον φάκελο _output/bin/ με το όνομα kube-scheduler. Στη περίπτωση που δεν μπορεί να δημιουργήσει το αρχείο, θα εμφανίσει η παραπάνω εντολή τα σφάλματα που εντόπισε. Όταν επιλυθούν όλα τα σφάλματα, τότε το αρχείο kube-scheduler είναι έτοιμο με όλα τα προεπιλεγμένα και εξατομικευμένα plugins που του ορίσαμε.

ΙΙ. Παράρτημα 2

Εγκατάσταση περιβάλλοντος για χρήση του kube-scheduler

Σε αυτό το παράρτημα θα παρουσιαστούν αναλυτικά τα βήματα που χρειάζεται να προηγηθούν προκειμένου να μπορεί να λειτουργήσει το σύστημα με τον kube-scheduler μας. Θεωρούμε πως έχουμε έτοιμο ένα δικό μας αρχείο kube-scheduler από το Παράρτημα 1.

Εγκατάσταση του microk8s

Προκείμενου να εγκαταστήσουμε το microk8s θα χρειαστεί να εγκαταστήσουμε το snap του. Στη συνέχεια θα χρειαστεί να περιμένουμε προκειμένου να ολοκληρωθεί η εγκατάσταση και να βεβαιωθούμε πως τρέχει. Τέλος, θα δώσουμε κατάλληλη πρόσβαση στον χρήστη «ubuntu» για να μπορεί να τρέχει το microk8s χωρίς να χρειάζεται να προσθέτει sudo.

Οι απαραίτητες εντολές είναι οι παρακάτω:

```
# Κατεβάζει το snap του microk8s. Η έκδοση που χρησιμοποιήθηκε είναι η 1.32.3

sudo snap install microk8s --classic

# Περιμένει μέχρι το microk8s να τρέχει και να είναι έτοιμο για να το χρησιμοποιήσουμε

microk8s status --wait-ready

# Παρέχουν τη δυνατότητα στον χρήστη ubuntu να χρησιμοποιεί το microk8s

sudo usermod -a -G microk8s ubuntu

sudo chown -f -R ubuntu ~/.kube
```

Κώδικας ΙΙ.1 Εγκατάσταση microk8s

Προκειμένου να πραγματοποιηθούν οι αλλαγές μπορείτε είτε να κάνετε reboot είτε να βγείτε και να ξαναμπείτε στο σύστημα. Και έτσι εγκαθιστούμε στο microk8s σε κάποιο μηχάνημα.

Ακόμα, προκειμένου να δουλέψει κατάλληλα, θα χρειαστεί να εγκαταστήσουμε μερικά add-ons (προτείνεται να προστεθούν ένα-ένα) και να μεταφέρουμε το configuration του kubernetes σε σημείο που ενδέχεται να χρειαστεί.

```
# Επιτρέπει τα παρακάτω add-ons
microk8s enable dns ingress metrics-server rbac hostpath-storage dashboard observability
https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md

# Μεταφέρει το configuration του microk8s σε σημείο που βλέπει το Kubernetes

microk8s config > ~/.kube/config
```

Κώδικας ΙΙ.2 Επιτροπή των απαραίτητων add-ons στο microk8s

Οι κωδικοί για πρόσβαση στις μετρικές είναι admin/prom-operator, το οποίο θα εμφανιστεί και όταν ολοκληρωθεί η εγκατάσταση του Observability.

Εγκατάσταση του Συστήματος KEPLER

Προκειμένου να έχουμε καλύτερο έλεγχο της ενεργειακής κατανάλωσης θα χρησιμοποιήσουμε το σύστημα KEPLER (Kubernetes-based Efficient Power Level Exporter). Το KEPLER εμφανίζει μετρικές για την ενεργειακή κατανάλωση και τη κατανάλωση ισχύος, ο καλύτερος τρόπος να ελέγξουμε πόσο καλός είναι τελικά ένας χρονοδρομολογητής. Η εγκατάσταση του είναι πως παρακάτω:

```
# Πρώτα εγκαθιστούμε το Github Repository του KEPLER στο βασικό μας μηχάνημα

git clone https://github.com/sustainable-computing-io/kepler.git

cd kepler

# Στη συνέχεια εκτελούμε τη παρακάτω  εντολή που το κάνει κατάλληλο για το μηχάνημα μας και για να τρέχει στο Prometheus.

make build-manifest OPTS="BM_DEPLOY PROMETHEUS_DEPLOY"

# Μετά εφαρμόζουμε το manifest και κάνουμε deploy το KEPLER

microk8s kubectl create ns kepler

microk8s kubectl apply -f _output/generated-manifest/deployment.yaml

# Και για να ελέγξουμε πως έχει εγκατασταθεί και λειτουργεί μπορούμε να εκτελέσουμε τη
# παρακάτω εντολή

microk8s kubectl get pods -n kepler
```

Κώδικας ΙΙ.1 Εγκατάσταση του συστήματος KEPLER

Τέλος, για να γίνει η επιβεβαίωση πως βγαίνουν τα σωστά αποτελέσματα, μπορείτε να μπείτε στο dashboard του Prometheus και να πάτε στο tab «status». Εκεί πρέπει να υπάρχει ένα κομμάτι μόνο για τον KEPLER με ενεργειακές τιμές. Στη περίπτωση που δεν εμφανίζονται στο Prometheus μπορούν να βρεθούν με ένα port-forward στο localhost.

Δημιουργία Cluster στο microk8s

Το Cluster στο σύστημα του microk8s είναι μια ομάδα από κόμβους που επικοινωνούν μεταξύ τους. Όπως φαίνεται στο κεφάλαιο 2 για το Kubernetes, επικοινωνούν όλοι οι κόμβοι με τον kube-scheduler ο οποίος τρέχει στο επίπεδο ελέγχου. Προκείμενου να τους συνδέσουμε όλους ωστόσο απαιτείται μια προεργασία. Επιλέγουμε ένα κόμβο που θα είναι ο κεντρικός μας και τρέχουμε την εντολή:

```
microk8s add-node

#Λαμβάνουμε το παρακάτω μήνυμα, το οποίο τρέχουμε σε διαφορετικό κόμβο

microk8s join 172.16.102.187:25000/c99ee99933d31a7553b72be919af4e5d/bc8447960289

# Για πιθανό έλεγχο που θα χρειαστεί. Η εντολή επιστρέφει όλους τους κόμβους στο cluster
```

```
microk8s kubectl get nodes
```

Κώδικας ΙΙ.1 Εισαγωγή κόμβου σε cluster

Στην απάντηση παίρνουμε την εντολή που θα χρειαστεί να τρέξει ένας άλλος κόμβος προκειμένου να μπει στο ίδιο cluster με τον αρχικό μας. Η ip που εμφανίζεται στην αρχή της εντολής είναι η private ip του μηχανήματος. Μερικές χρήσιμες πληροφορίες σχετικά με την εισαγωγή κόμβου σε cluster είναι οι εξής:

- Στην περίπτωση που θέλουμε να προσθέσουμε ένα κόμβο που να είναι «worker», δηλαδή να μην τρέχει στο επίπεδο ελέγχου, μπορούμε να προσθέσουμε το flag "--worker".
- Κάθε εντολή μπορεί να χρησιμοποιηθεί μονάχα από έναν άλλο κόμβο. Αν θέλουμε να προσθέσουμε περισσότερους κόμβους χρειάζεται να επαναληφθεί η διαδικασία για κάθε κόμβο.

Ακολουθώντας αυτή τη διαδικασία θα έχουμε το cluster μας στο microk8s ώστε να μπορούμε να τρέξουμε τα workloads που θέλουμε.

Ο χρονοδρομολογητής

Εφόσον στο microk8s δεν μπορούμε να αλλάξουμε τον kube-scheduler, καθώς είναι ενσωματωμένος στην υπηρεσία kubelite, χρειάστηκε να βρεθεί ένας εναλλακτικός τρόπος για να μπορούμε να έχουμε τον δικό μας χρονοδρομολογητή και να μπορούμε να τον επηρεάζουμε με μεγαλύτερη άνεση. Αυτό επιτυγχάνεται με την βοήθεια του Docker και διάφορων αρχείων που προτείνονται εδώ από την επίσημη βιβλιογραφία του Kubernetes με ελάχιστες τροποποιήσεις. Έστω πως είμαστε στο home του user ubuntu. Τότε θα τρέξουμε τα εξής:

```
# Δημιουργούμε φάκελο στον οποίο θα εργαστούμε
mkdir scheduler
cd scheduler
#Δημιουργούμε ένα απλό Dockerfile
nano Dockerfile
```

Κώδικας ΙΙ.1 Δημιουργία αρχείου scheduler και Dockerfile

```
# Dockerfile
FROM registry.k8s.io/kube-scheduler:v1.29.3
COPY kube-scheduler /usr/local/bin/kube-scheduler
```

Κώδικας ΙΙ.2 Dockerfile για δημιουργία εικόνας χρονοδρομολογητή

Αφού είναι έτοιμο το Dockerfile, τότε θα χρειαστεί να χρησιμοποιήσουμε το Docker Hub, το οποίο ήταν ο πιο εύκολος τρόπος που βρέθηκε για τη παρούσα υλοποίηση. Να σημειωθεί πως στο φάκελο scheduler θα πρέπει να υπάρχει το εκτελέσιμο του kube-scheduler που φτιάξαμε στο Παράρτημα 1.

```
# Φτιάχνουμε το image του kube-scheduler και το προωθούμε στο dockerhub προφίλ μας
docker build -t dockerhub_user/custom-scheduler:latest .
docker login
docker push dockerhub_user/custom-scheduler:latest
```

Κώδικας ΙΙ.3 Δημιουργία image του kube-scheduler στο Docker Hub

Σε αυτό το σημείο θα χρειαστεί να φτιάξουμε yaml αρχεία για Service Account και το RBAC. Η προσθήκη τους έγινε καθώς αυτό προτείνεται από το επίσημο documentation του Kubernetes. Κάθε scheduler χρειάζεται ένα ServiceAccount προκειμένου να μπορεί ν εκτελέσει με συγκεκριμένα credentials και να έχει δικαιώματα στο API. Στη συνέχεια μέσω του ClusterRole εκχωρεί στον my-scheduler όλα τα δικαιώματα που έχει και ο προεπιλεγμένος kube-scheduler, όπως ανάγνωση Pods, Nodes, πρόσβαση σε API endpoints για δρομολόγηση και δημιουργία και τροποποίηση binding αποφάσεων. Παράλληλα, εκχωρεί πρόσβαση σε ό,τι αφορά το PersistentVolumeClaim scheduling. Στη περίπτωση μας δεν χρησιμοποιούμε PVC αλλά ενδέχεται να χρειαστεί στο μέλλον σε κάποιο άλλο σύστημα. Τέλος, δίνεται πρόσβαση στον extension-apiserver-authentication-reader config map καθώς αυτό απαιτείται για να διαβάζουν authentication των extended API servers όπως ο metrics server.

Το Yaml αρχείο είναι το παρακάτω:

```
# serviceAccount-rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-volume-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:volume-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: my-scheduler-extension-apiserver-authentication-reader
```

```
  namespace: kube-system
roleRef:
  kind: Role
  name: extension-apiserver-authentication-reader
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
```

Κώδικας ΙΙ.4 Αρχείο για configuration του Service Account και του RBAC

Ο παραπάνω αρχείο δεν θα χρειαστεί να το ξανατρέξουμε, παρά μόνο αν αλλάξουμε ονομασία στον χρονοδρομολογητή μας. Σε αυτό το σημείο θα χρειαστεί να φτιάξουμε το αρχείο για το deployment το οποίο θα κάνει deploy το Pod με τον χρονοδρομολογητή μας. Το αρχείο φαίνεται παρακάτω:

```
# scheduler-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: custom-scheduler
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      component: custom-scheduler
  template:
    metadata:
      labels:
        component: custom-scheduler
    spec:
      serviceAccountName: my-scheduler
      containers:
        - name: kube-scheduler
          image: dockerhub_user/custom-scheduler:latest
          imagePullPolicy: Always
          command:
            - /usr/local/bin/kube-scheduler
            - --config=/etc/scheduler/kube-scheduler-config.yaml
            - --v=3
          volumeMounts:
            - name: config-volume
              mountPath: /etc/scheduler
              readOnly: true
      volumes:
        - name: config-volume
          configMap:
            name: my-scheduler-config
```

Κώδικας ΙΙ.5 Αρχείο yaml για deployment του χρονοδρομολογητή

Ακόμα, μαζί με αυτά χρειάζεται να έχουμε ένα αρχείο για configuration του χρονοδρομολογητή μας. Σε αυτό το αρχείο θα έχουμε όλα τα profiles που θέλουμε. Ένα παράδειγμα του αρχείου είναι το παρακάτω:

```
# kube-scheduler-config.yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
leaderElection:
  leaderElect: false
profiles:
  - schedulerName: my-default-scheduler
  - schedulerName: my-scheduler
    plugins:
      score:
        disabled:
          - name: "*"
        enabled:
          - name: NodeResourcesBalancedAllocation
            weight: 1
          - name: ImageLocality
            weight: 1
          - name: NodeAffinity
            weight: 1
          - name: TaintToleration
            weight: 1
          - name: VectorBinPacking
            weight: 1
    pluginConfig:
      - name: VectorBinPacking
        args:
          apiVersion: kubescheduler.config.k8s.io/v1
          kind: Args
```

Κώδικας ΙΙ.6 Παράδειγμα configuration του εξατομικευμένου χρονοδρομολογητή

Στο παραπάνω configuration φαίνεται μια υλοποίηση με 2 profiles, ένα του προεπιλεγμένου χρονοδρομολογητή και ένα ενός εξατομικευμένου, στο οποίο συμπεριλαμβάνονται και τα βάρη. Κάθε καινούριο plugin που προστίθεται πρέπει να βρίσκεται και στο pluginConfig μαζί με τα απαραίτητα arguments που χρειάζεται. Προκειμένου ωστόσο να χρησιμοποιηθούν όλα τα παραπάνω, χρειάζεται να τα κάνουμε apply στο microk8s, ενώ παράλληλα δημιουργούμε και ένα config map.

```
microk8s kubectl apply -f serviceAccount-rbac.yaml
microk8s kubectl create configmap my-scheduler-config \
  --from-file=kube-scheduler-config.yaml=/home/ubuntu/scheduler/kube-scheduler-
config.yaml \
  -n kube-system
microk8s kubectl apply -f scheduler-deployment.yaml
```

Κώδικας ΙΙ.7 Εφαρμογή απαραίτητων αρχείων για λειτουργεία χρονοδρομολογητή

Και μετά από μερικά δευτερόλεπτα θα τρέχει ο kube-scheduler και θα μπορεί να χρησιμοποιηθεί. Στην περίπτωση που χρειάζεται να αλλάξει ο kube-scheduler, τότε θα πρέπει να εκτελεστούν οι 2 παρακάτω εντολές:

```
microk8s kubectl delete deployment custom-scheduler -n kube-system
microk8s kubectl delete configmap my-scheduler-config -n kube-system
```

Κώδικας ΙΙ.8 διαγραφή deployment και config map

Με αυτό το τρόπο μπορούμε, ακολουθώντας όλη τη διαδικασία ξανά με τις απαραίτητες αλλαγές, αν αλλάξουμε ονομασίες για παράδειγμα, να τρέξουμε καινούρια εκδοχή του χρονοδρομολογητή.

Ακολουθώντας όλα αυτά τα βήματα έχουμε ένα χρονοδρομολογητή που τρέχει και λειτουργεί στο σύστημα.

Βιβλιογραφία - Παραπομπές

[1] ‘Production-Grade Container Orchestration’, Kubernetes. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://kubernetes.io/

[2] A. Pyliotis, *ntua-el19050/Energy-aware-kubernetes-scheduler*. (16 Ιούνιος 2025). Go. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://github.com/ntua-el19050/Energy-aware-kubernetes-scheduler

[3] ‘Scheduling Framework’, Kubernetes. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[4] The Kubernetes Authors, ‘Kubernetes Metrics \textttmetrics.k8s.io API and Metrics Server’. 2023. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://kubernetes.io/docs/reference/external-api/metrics.v1beta1/

[5] ‘Prometheus - Monitoring system & time series database’. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://prometheus.io/

[6] ‘Kepler’. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://sustainable-computing.io/

[7] R. Panigrahy, K. Talwar, L. Uyeda, και U. Wieder, ‘Heuristics for Vector Bin Packing’, Microsoft Research, Technical Report, Ιανουαρίου 2011. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://www.labri.fr/perso/eyraud/pmwiki/uploads/Main/Panigrahy2011-VBPHeuristics.pdf

[8] L. Ismail και H. Materwala, *EATSVM: Energy-Aware Task Scheduling on Cloud Virtual Machines*, τ. 135. 2018, σελ. 258. doi: 10.1016/j.procs.2018.08.172.

[9] W. Gerards και others, ‘Energy-efficient DVFS scheduling for real-time systems’, *ACM Trans. Embed. Comput. Syst.*, τ. 14, τχ. 2, σελ. 1–20, 2015.

[10] T. Horvath και others, ‘Dynamic voltage scaling in real-time systems’, *IEEE Trans. Comput.*, τ. 56, τχ. 4, σελ. 478–493, 2007.

[11] I. Goiri και others, ‘GreenSlot: Scheduling energy consumption in green datacenters’, στο *Proceedings of the ACM/IEEE Supercomputing Conference*, 2011.

[12] ‘MicroK8s - MicroK8s documentation - home | MicroK8s’, microk8s.io. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: http://microk8s.io

[13] Y. Xu και others, ‘Energy-aware VM placement using a hybrid genetic algorithm in cloud’, *J. Specified*, 2018.

[14] C. Li και others, ‘Energy-aware scheduling using hybrid PSO in virtualized data centers’, *J. Specified*, 2020.

[15] K. Deb και others, ‘A fast and elitist multiobjective genetic algorithm: NSGA-II’, *IEEE Trans. Evol. Comput.*, τ. 6, τχ. 2, σελ. 182–197, 2002, doi: 10.1109/4235.996017.

[16] ‘Bin Packing Problem (Minimize number of used Bins)’, GeeksforGeeks. Ημερομηνία πρόσβασης: 17 Ιούνιος 2025. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://www.geeksforgeeks.org/bin-packing-problem-minimize-number-of-used-bins/

[17] A. Beloglazov, J. Abawajy, και R. Buyya, ‘Energy-aware resource allocation heuristics for efficient management of data centers’, *Future Gener. Comput. Syst.*, τ. 28, τχ. 5, σελ. 755–768, 2012, doi: 10.1016/j.future.2011.04.017.

[18] S. Garg και others, ‘Environment-conscious scheduling of HPC applications on cloud’, *J. Specified*, 2011.

[19] D. Kliazovich και others, ‘Energy-aware scheduling for cloud datacenters using workload profiling’, *J. Specified*, 2013.

[20] M. Stillwell και others, ‘Resource Allocation Algorithms for Virtualized Service Hosting Platforms’, *J. Parallel Distrib. Comput.*, τ. 70, τχ. 9, σελ. 962–974, 2010.
[21] L. Benini, A. Bogliolo, και G. De Micheli, ‘A Survey of Design Techniques for System-Level Dynamic Power Management’, *IEEE Trans. VLSI*, τ. 8, τχ. 3, σελ. 299–316, 2000.
[22] P. Ranganathan και others, ‘Ensemble-level power management for dense blade servers’, στο *Proceedings of ACM SIGARCH*, 2006.
[23] A. Bhardwaj και others, ‘Resource-aware Scheduling Using Similarity Metrics in Cloud Environments’, *J. Specified*, 2022.
[24] B. Kocot, P. Czarnul, και J. Proficz, ‘Energy-Aware Scheduling for High-Performance Computing Systems: A Survey’, *Energies*, τ. 16, τχ. 2, σελ. 890, 2023, doi: 10.3390/en16020890.
[25] K. Authors, ‘Descheduler for Kubernetes’. 2024. [Έκδοση σε ψηφιακή μορφή]. Διαθέσιμο στο: https://kubernetes.io/docs/concepts/scheduling-eviction/descheduler/