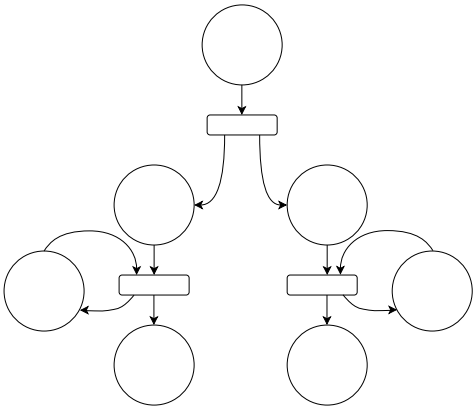
Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Επικοινωνιων, Ηλεκτρονικης & Συστημάτων Πληροφορικης
Εργαστηριο Διαχειρισης & Βελτιστου Σχεδιασμου Δικτυων Τηλεματικης

Μοντελοποίηση Multi-Cluster υποδομών με χρήση Δικτύων Petri

Διπλωματικη Εργασια

του

ΧΡΙΣΤΟΦΟΡΟΥ Ν. ΒΑΡΔΑΚΗ



Επιβλέπων: Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Επικοινωνιων, Ηλεκτρονικης & Συστημάτων Πληροφορικης
Εργαστηριο Διαχειρισης & Βελτιστου Σχεδιασμου Δικτυων Τηλεματικης

Μοντελοποίηση Multi-Cluster υποδομών με χρήση Δικτύων Petri

Διπλωματικη Εργασια

του

ΧΡΙΣΤΟΦΟΡΟΥ Ν. ΒΑΡΔΑΚΗ

Επιβλέπων: Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Ιουλίου 2024.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Συμεών Παπαβασιλείου	Γεώργιος Ματσόπουλος	Ιωάννα Ρουσσάκη
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αναπληρώτρια Καθηγήτρια Ε.Μ.Π.

Αθήνα, Ιούλιος 2024

(Υπογραφή)

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

Περίληψη

Η ανάπτυξη νέων τεχνολογιών, όπως τα δίκτυα 5ης γενιάς και το Διαδίκτυο των Αντικειμένων, οδηγεί στην αντικατάσταση των παραδοσιακών συστημάτων υπολογιστικής Νέφους από το φάσμα του υπολογιστικού νέφους, ένα συνδεδεμένο περιβάλλον αποτελούμενο από διαφόρων ειδών συσκευές που φέρουν την επεξεργασία των δεδομένων πιο κοντά στην πηγή τους. Για την ανάπτυξη των εφαρμογών σε αυτό το περιβάλλον αποτελεί μονόδρομο η χρήση τεχνολογιών εικονικοποίησης και containerisation. Για την ενορχήστρωση αυτού του είδους εφαρμογών σε ολοένα και πολυπλοκότερες δομές, είναι συχνά απαραίτητη η επιστράτευση λύσεων Πολλαπλών Υποδομών (Multi-Cluster). Παρά τα πλεονεκτήματά τους, οι λύσεις αυτές χαρακτηρίζονται από αυξημένη πολυπλοκότητα, δυναμικότητα, ανομοιογένεια, υψηλό κόστος επικοινωνίας και αυξημένες καθυστερήσεις κατά τη διαχείρισή τους. Επιπλέον πρόκληση αποτελεί η προσέγγιση συμπεριφοράς αυτών των συστημάτων προς βελτιστοποίηση της διαχείρισης των διαθέσιμων πόρων, όπου καθορίζονται από ένα μεγάλο σύνολο πιθανών καταστάσεων. Στην παρούσα διπλωματική εργασία μελετάται η χρήση θεωρίας αυτομάτων στοχεύοντας στην μοντελοποίηση τέτοιων συστημάτων μέσω της ανάλυσης των πιθανών καταστάσεων. Συγκεκριμένα, ως μέσω μοντελοποίησης προτείνουμε τα δίκτυα Petri, μία μαθηματική δομή ικανή για αναπαράσταση συστημάτων διακριτών συμβάντων.

Στο πλαίσιο της παρούσας εργασίας, σχεδιάστηκαν μοντελοποιήσεις για τη διαδικασία τοποθέτησης εφαρμογών σε συστήματα τα οποία διαχειρίζεται λογισμικό ενορχήστρωσης πολλαπλών εικονικοποιημένων υποδομών, συγκεκριμένα το λογισμικό Karmada. Μοντελοποιήθηκαν οι βασικές πολιτικές τοποθέτησης εφαρμογών του λογισμικού, μέσω της ανάπτυξης των αντίστοιχων δικτύων. Επίσης, υλοποιήθηκε μία επεκτάσιμη και παραμετροποιήσιμη βιβλιοθήκη, βασισμένη σε αρχές αντικειμενοστραφούς προγραμματισμού, η οποία είναι ικανή για την προγραμματιστική υλοποίηση διαφόρων συστημάτων χρησιμοποιώντας τις μοντελοποιήσεις αυτές. Η βασισμένη στα δίκτυα Petri υλοποίηση αξιολογήθηκε ως προς τη δυνατότητα προσέγγισης της κατάστασης του συστήματος προσομοιώνοντας τις πολιτικές που υποστηρίζονται, συγκρινόμενη με την πραγματική απόκριση σε ένα πειραματικό περιβάλλον νέφους, πολλαπλών υποδομών που διαχειρίζεται το Karmada. Τα αποτελέσματα αναδεικνύουν την ικανότητα των δικτύων για ακριβή πρόβλεψη στην συντριπτική πλειοψηφία των περιπτώσεων καθώς και για παραμετροποίηση με σκοπό την προσαρμογή σε περίπτωση αποτυχίας. Τέλος, η υλοποίηση εξετάστηκε ως προς την χρονική πολυπλοκότητα, παρουσιάζοντας χαμηλό χρόνο εκτέλεσης και πολυωνυμική μεταβολή του χρόνου σε σχέση με την κλιμάκωση των εφαρμογών.

Λέξεις Κλειδιά

Ενορχήστρωση Πολλαπλών Υποδομών, Δίκτυα Petri, Υπολογιστική Νέφους, Υπολογιστική Ομίχλης, Υπολογιστική Άκρων, Kubernetes, Karmada

Abstract

The development of new technologies, such as 5G networks and the Internet of Things, leads to the replacement of traditional Cloud Computing systems by the Cloud Continuum, a connected environment consisting of various devices that bring data processing closer to their source. Developing applications in this environment necessitates the use of virtualization and containerization technologies. Orchestrating such applications into increasingly complex structures often requires the usage of Multi-Cluster solutions. Despite their advantages, such solutions are characterized by increased complexity, dynamic behaviour, heterogeneity, high communication costs, and increased delays in their management. Another challenge constitutes optimizing the behavior of these systems to manage available resources, which are determined by a large set of potential states. This thesis explores the use of automata theory to model such systems by analyzing possible states. Specifically, as a modeling tool we propose Petri Nets, a mathematical structure capable of representing discrete events systems.

Within the scope of this thesis, models were designed for the placement process in systems managed by multi-cluster orchestration software, specifically Karmada. The basic placement policies of the software were modeled through the development of corresponding networks. Additionally, we implemented an extensible and customizable library, based on principles of object-oriented programming, capable of programmatically implementing various systems using these models. The Petri Net-based implementation was evaluated in terms of its ability to approximate the system state by simulating supported policies, comparing it with actual responses in an experimental multi-cluster cloud environment managed by Karmada. The results demonstrate the networks' ability to provide accurate predictions in the overwhelming majority of cases, as well as configuration for adaptation in case of failure. Finally, the implementation was examined in regards to its time complexity, demonstrating low execution time and polynomial change rate of time relative to application scaling.

Keywords

Multi-Cluster Orchestration, Petri Nets ,Cloud Computing, Fog Computing, Edge Computing, Kubernetes, Karmada

στην οικογένειά μου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή κ. Συμεών Παπαβασιλείου για την επίβλεψη αυτής της διπλωματικής εργασίας και για την ευκαιρία που μου έδωσε να ασχοληθώ με ένα τόσο ενδιαφέρον και χρήσιμο αντικείμενο. Ιδιαίτερες ευχαριστίες θα ήθελα να δώσω επίσης τους Δρ. Δημήτριο Δεχουνιώτη και Δρ. Ιωάννη Δημολίτσα για τη συνεχή τους βοήθεια και συνεργασία καθ' όλη τη διάρκεια της διπλωματικής. Τέλος, θα ήθελα να ευχαριστήσω τους γονείς, την οικογένεια και τους φίλους μου για την καθοδήγηση και τη συμπαράσταση που μου προσέφεραν όλα αυτά τα χρόνια, καθώς χωρίς αυτούς η εργασία αυτή δε θα είχε πραγματοποιηθεί.

Αθήνα, Ιούλιος 2024

Χριστόφορος Βαρδάκης

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

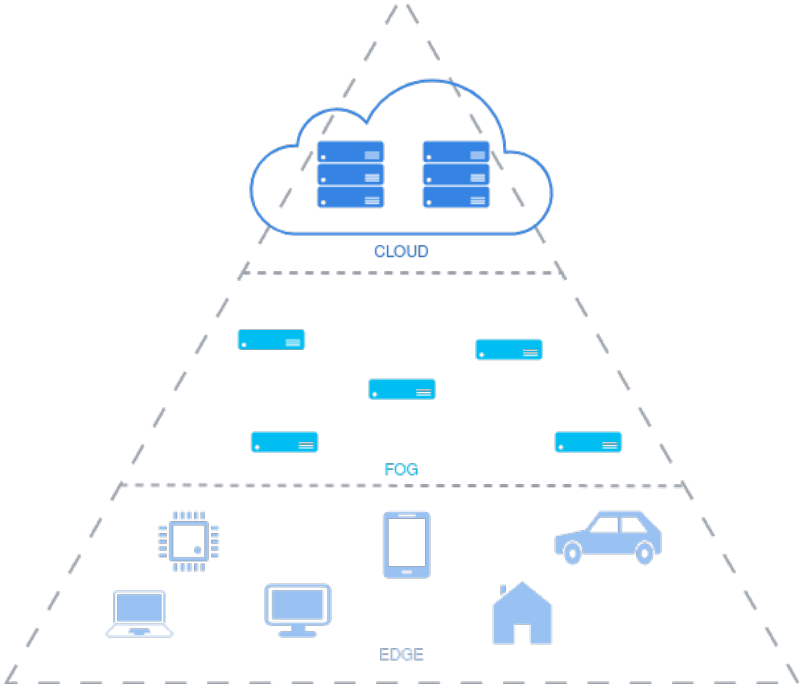
Εισαγωγή

Ο τομέας της πληροφορικής υφίσταται σημαντικές αλλαγές, οι οποίες οδηγούν στην αντικατάσταση του παραδοσιακού μοντέλου της υπολογιστικής νέφους. Αυτές οι αλλαγές οφείλονται σε νέες τεχνολογίες, όπως τα δίκτυα πέμπτης γενιάς (5G) ή το διαδίκτυο των αντικειμένων (Internet of Things - IoT), οι οποίες αυξάνουν εκθετικά τον αριθμό των συνδεδεμένων στο διαδίκτυο συσκευών [1]. Έτσι, παρατηρείται ανάδυση ενός νέου μοντέλου υπολογισμού, του φάσματος του Υπολογιστικού Νέφους (Cloud Continuum) ενός προτύπου που διανέμει τους υπολογιστικούς πόρους σε διάφορα επίπεδα, φέρνοντας την επεξεργασία των δεδομένων πιο κοντά στην πηγή τους [2].

Τα παραδοσιακά κεντρικά μοντέλα αποθήκευσης και επεξεργασίας δεδομένων έχουν δώσει τα τελευταία χρόνια τη θέση τους σε πιο κατανεμημένες αρχιτεκτονικές. Η Υπολογιστική Νέφους (Cloud Computing) παρέχει ευελιξία και δυνατότητα κλιμάκωσης, προσφέροντας δυναμικά πρόσβαση σε υπολογιστικούς πόρους. Ωστόσο, για εφαρμογές που απαιτούν επεξεργασία σε πραγματικό χρόνο και ελάχιστο χρόνο καθυστέρησης, όπως έξυπνες πόλεις, έξυπνη γεωργία, Διαδίκτυο των Ιατρικών Αντικειμένων (Internet of Medical Things), υψηλής ποιότητας video streaming, online gaming, αυτόνομη οδήγηση και έξυπνα σπίτια, οι λύσεις που βασίζονται στο νέφος μπορεί να μην είναι οι ιδανικές[3][4]. Σε αυτό το σημείο έρχονται σε εφαρμογή η υπολογιστική ομίχλης (fog) [5] και η υπολογιστική άκρων (edge) [6]).

Η υπολογιστική ομίχλης (fog computing) επεκτείνει την εμβέλεια του υπολογιστικού νέφους φέρνοντας τις δυνατότητες επεξεργασίας πιο κοντά στα άκρα του δικτύου (Edge). Αντικαθιστά τα παραδοσιακά cloud συστήματα, τα οποία απαρτίζονταν από μεγάλα κεντρικά datacenters, με κατανεμημένους μικρότερους servers οι οποίοι βρίσκονται πιο κοντά στις συσκευές των άκρων (Edge devices) [5]. Αυτό επιτρέπει ταχύτερη επεξεργασία δεδομένων και μειωμένη καθυστέρηση σε σύγκριση με την αποκλειστική χρήση κεντρικών πόρων νέφους. Η υπολογιστική άκρων (Edge Computing) προχωράει αυτή την ιδέα ακόμα περισσότερο, εστιάζοντας στην επεξεργασία δεδομένων σε πραγματικό χρόνο στις συσκευές που βρίσκονται στα άκρα του δικτύου (Edge Devices). Οι συσκευές αυτές μπορεί να είναι είτε οι ίδιες οι συσκευές των χρηστών μίας υποδομής, είτε έξυπνες συσκευές που αποτελούν μέρος μίας ευρύτερης υποδομής, όπως συμβαίνει για παράδειγμα στο Διαδίκτυο των Αντικειμένων (IoT) [6].

Αυτή η κατανεμημένη προσέγγιση επεξεργασίας προσφέρει πολλά πλεονεκτήματα. Πρώτον, μειώνει σημαντικά την καθυστέρηση, καθώς η επεξεργασία των δεδομένων γίνεται πιο κοντά στην πηγή τους, μειώνοντας κατά συνέπεια τον χρόνο που απαιτείται για τη μεταφορά και την επεξεργασία τους. Δεύτερον, βελτιώνει τη συνολική απόδοση των εφαρμογών μειώνοντας τον φόρτο εργασίας στα κεντρικά συστήματα. Επιπλέον, διατηρώντας τα δεδομένα πιο κοντά στην πηγή τους, ενισχύει την ασφάλεια ελαχιστοποιώντας τους κινδύνους που σχετίζονται με τη μετάδοση μέσω δικτύου. Τέλος, η τοπική επεξεργασία μειώνει την ανάγκη για εκτεταμένη μεταφορά δεδομένων, οδηγώντας σε χαμηλότερη



Σχήμα 1: *Το φάσμα του Υπολογιστικού Νέφους*

κατανάλωση ενέργειας.[7][8]

Για τη διαχείριση αυτών των αποκεντρωμένων υποδομών, το Kubernetes (βλ. κεφάλαιο 1.2) αναδεικνύεται ως η de facto λύση. Η εγγενής ευελιξία, η αυτοματοποίηση και η δυνατότητα κλιμάκωσης του Kubernetes το καθιστούν ιδανικό για τέτοιου είδους περιβάλλοντα. Με τη χρήση του Kubernetes, μία συστάδα (cluster) από servers (πραγματικών ή εικονικών) μπορεί να συμπεριφερθεί σαν μία ενιαία υποδομή. Η έμφαση του Kubernetes στην ευελιξία, την αυτοματοποίηση, τη δυνατότητα κλιμάκωσης και την εγγύηση διαθεσιμότητας το καθιστά την ιδανική πλατφόρμα για τη διαχείριση εφαρμογών στο φάσμα του υπολογιστικού νέφους.

Ενώ το Kubernetes αποτελεί εξαιρετική λύση για διαχείριση εφαρμογών στο φάσμα του υπολογιστικού νέφους, η χρήση ενός cluster παρουσιάζει περιορισμούς, ιδιαίτερα σε περιβάλλοντα fog και edge. Ένας μεμονωμένος Kubernetes cluster έχει περιορισμένα όρια σχετικά με τον αριθμό των κόμβων (nodes) και των pods (δηλ. servers και εφαρμογών αντίστοιχα) που μπορεί να διαχειριστεί [9]. Αυτά τα όρια, αν και αρκετά υψηλά, συνήθως δεν επαρκούν για εφαρμογές μεγάλης κλίμακας που απαιτούνται σε περιβάλλοντα ομίχλης και άκρων, όπου τα δεδομένα διανέμονται σε πολλά αποκεντρωμένα σημεία. Συχνά τέτοιες υποδομές μπορεί να έχουν από εκατομμύρια έως και δισεκατομμύρια συνδεδεμένες συσκευές, καθιστώντας τη διαχείριση τους από έναν cluster ανέφικτη [10]. Επιπλέον, ένας μεμονωμένος Kubernetes cluster προϋποθέτει συνήθως μια περιορισμένη γεωγραφική περιοχή. Σε περιβάλλοντα ομίχλης και άκρων, όπου οι συσκευές μπορεί να είναι διασκορπισμένες σε μεγάλη γεωγραφική έκταση, η χρήση ενός cluster μπορεί να οδηγήσει σε αυξημένη καθυστέρηση και μειωμένη απόδοση.

Για την αντιμετώπιση των περιορισμών των single cluster υποδομών Kubernetes, η χρήση υποδομών Multi-Cluster αναδύεται ως μια πολλά υποσχόμενη λύση. Οι υποδομές Multi-Cluster επιτρέπουν τη διαχείριση πολλαπλών Kubernetes clusters σαν ένα ενιαίο σύστημα,

προσφέροντας σημαντικά οφέλη. Η δυνατότητα διαχείρισης πολλαπλών clusters επιτρέπει την κατανομή των πόρων υπολογισμού σε μια ευρύτερη γεωγραφική περιοχή, καθιστώντας τις Multi-Cluster υποδομές ιδανικές για εφαρμογές ομίχλης και άκρων. Επιπλέον, παρέχουν εργαλεία για την κεντρική διαχείριση πολλαπλών clusters, μειώνοντας την πολυπλοκότητα και το χρόνο που απαιτείται για τη διαχείριση απομακρυσμένων υποδομών. Τέλος, η ύπαρξη πολλαπλών clusters αυξάνει την ανθεκτικότητα του συστήματος, καθώς η αποτυχία ενός cluster δεν επηρεάζει τη λειτουργία των υπολοίπων.

Παρότι οι υποδομές Multi-Cluster προσφέρουν σημαντικά οφέλη, η υλοποίησή τους παρουσιάζει και ορισμένες προκλήσεις. Η διαχείριση πολλαπλών clusters εισάγει πρόσθετη πολυπλοκότητα στο σύστημα. Η διαμόρφωση, η ανάπτυξη και η παρακολούθηση εφαρμογών σε πολλούς clusters απαιτεί πρόσθετα εργαλεία και διαδικασίες. Επιπλέον, οι clusters μπορεί να διαφέρουν ως προς το υλικό, το λογισμικό και τη διαμόρφωση. Τέλος, η επικοινωνία μεταξύ απομακρυσμένων clusters μπορεί να εισάγει καθυστερήσεις στην ανάπτυξη και τη διαχείριση των εφαρμογών.

Για την αντιμετώπιση αυτών των προκλήσεων, χρήσιμο εργαλείο αποτελεί η μοντελοποίηση Multi-Cluster συστημάτων. Αυτά τα μοντέλα μπορούν να βοηθήσουν στη βελτίωση της αυτόματης διαχείρισης, της διαχείρισης πόρων και της ανθεκτικότητας σε σφάλματα. Τα τυπικά μοντέλα μπορούν να προσομοιάσουν διαδικασίες και να προβλέψουν τη συμπεριφορά του συστήματος, χωρίς να χρειάζεται να γίνει παρέμβαση στο πραγματικό σύστημα. Έτσι μπορούν να αποφευχθούν ανεπιθύμητες καταστάσεις, να γίνει σύγκριση διαφορετικών εναλλακτικών και γενικότερα να γίνει προληπτική διαχείριση και συντήρηση (predictive management and maintenance), χωρίς να χρειάζεται να γίνουν πραγματικές δοκιμές, οι οποίες θα κοστίσουν χρόνο και υπολογιστικούς πόρους.

Αντικείμενο της διπλωματικής

Αντικείμενο της παρούσας εργασίας είναι η σχεδίαση και υλοποίηση μοντέλων βασισμένων στα Δίκτυα Petri, τα οποία θα μπορούν επιτυχώς να περιγράψουν και να προβλέψουν τη συμπεριφορά ενός Multi-Cluster συστήματος. Αρχικά, γίνεται θεωρητικός σχεδιασμός Δικτύων Petri, τα οποία περιγράφουν τον τρόπο με τον οποίο λαμβάνουν χώρα συμβάντα σε ένα Multi-Cluster περιβάλλον. Συγκεκριμένα, γίνεται εκτενής μοντελοποίηση των πιθανών μεθόδων τοποθέτησης (placement) των διεργασιών στο σύστημα, ακολουθώντας τις πολιτικές διάδοσης (propagation policies) που υλοποιεί το λογισμικό Karmada, το οποίο αποτελεί ένα από τα πλέον διαδεδομένα εργαλεία για τη διαχείριση Multi-Cluster συστημάτων (βλ. Κεφάλαιο 2). Επίσης, γίνεται αναφορά και σε άλλες διεργασίες που λαμβάνουν χώρα σε ένα τέτοιο σύστημα και μπορούν να μοντελοποιηθούν με χρήση Δικτύων Petri.

Πέρα από τη θεωρητική μελέτη του προβλήματος, γίνεται και υλοποίηση των παραπάνω μοντέλων σε γλώσσα προγραμματισμού Python. Συγκεκριμένα, υλοποιήθηκε μία βιβλιοθήκη (python module) η οποία μπορεί να χρησιμοποιηθεί για την προγραμματιστική υλοποίηση των μοντέλων αυτών. Η βιβλιοθήκη αυτή έχει επίσης κατάλληλη δομή, ώστε να είναι παραμετροποιήσιμη και επεκτάσιμη, δίνοντας τη δυνατότητα για προσθήκη νέων μοντέλων, αλλά και συνδυασμό μοντέλων για τη δημιουργία πολυπλοκότερων δομών. Επίσης, υλοποιήθηκαν και κατάλληλες μέθοδοι για την οπτικοποίηση των μοντέλων, την εξαγωγή χρήσιμων μετρικών

και τη δυναμική περιγραφή πραγματικών συστημάτων.

Διάρθρωση τόμου

Η παρούσα εργασία χωρίζεται σε 2 μέρη, κάθε ένα από τα οποία χωρίζεται σε 3 κεφάλαια, με κοινή αρίθμηση. Το πρώτο μέρος (κεφάλαια 1 έως 3) καλύπτει το θεωρητικό υπόβαθρο που χρειάζεται για την εργασία, ενώ το δεύτερο μέρος (κεφάλαια 4 έως 6) περιγράφει αναλυτικά την πρακτική μελέτη του προβλήματος, που αποτελεί και τη συνεισφορά της παρούσας εργασίας. Στο κεφάλαιο 1 μελετάται η χρήση containerisation για την υλοποίηση των σύγχρονων εφαρμογών στο φάσμα του Υπολογιστικού Νέφους, καθώς και η χρήση ενορχηστρωτών για τη διαχείριση τους. Στο κεφάλαιο 2 περιγράφεται η αρχιτεκτονική του λογισμικού Karmada, λογισμικού που χρησιμοποιείται για τη διαχείριση Multi Cluster συστημάτων. Στο κεφάλαιο 3 γίνεται μία εισαγωγή στα Δίκτυα Petri και τις ιδιότητές τους, καθώς γίνεται και ιδιαίτερη αναφορά στα Έγχρωμα Δίκτυα Petri.
Στο κεφάλαιο 4 περιγράφονται αναλυτικά τα Δίκτυα Petri τα οποία σχεδιάστηκαν για να μοντελοποιήσουν συστήματα Multi-Cluster που χρησιμοποιούν το Karmada. Στο κεφάλαιο 5 παρουσιάζεται η υλοποίηση των παραπάνω Δικτύων σε γλώσσα προγραμματισμού Python, δημιουργώντας μία παραμετροποιήσιμη και επεκτάσιμη βιβλιοθήκη. Τέλος, στο κεφάλαιο 6 διενεργούνται πειράματα για την αξιολόγηση των μοντέλων σε σύγκριση με ένα πραγματικό Multi-Cluster σύστημα.

Μέρος Ι

Θεωρητικό Μέρος

Κεφάλαιο 1

Ανάπτυξη και Ενορχήστρωση στο φάσμα του Υπολογιστικού Νέφους

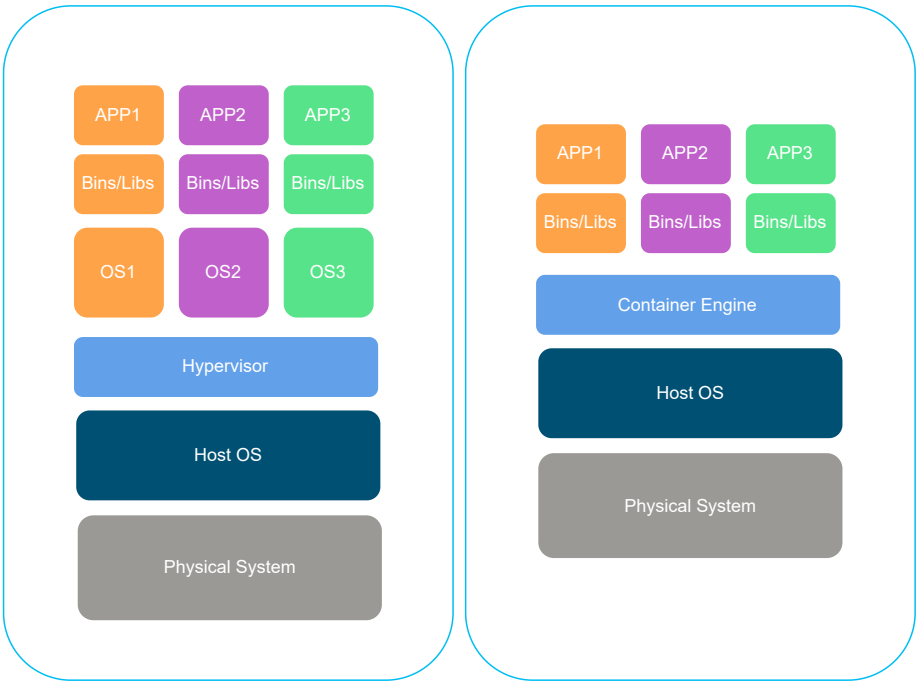
Στο κεφάλαιο αυτό γίνεται μία σύντομη εισαγωγή σε έννοιες σχετικές με την ανάπτυξη και ενορχήστρωση εφαρμογών στο φάσμα του υπολογιστικού Νέφους. Αρχικά, γίνεται μία σύντομη εισαγωγή στις έννοιες της εικονικοποίησης και του containerisation. Εν συνεχεία γίνεται μία εισαγωγή στην ενορχήστρωση containerised εφαρμογών καθώς και περιεκτική παρουσίαση της δομής και της λειτουργίας του λογισμικού Kubernetes. Τέλος, γίνεται αναφορά στη χρήση Multi-Cluster υποδομών και τους λόγους για τους οποίους τα σύγχρονα υπολογιστικά συστήματα τις αξιοποιούν.

1.1 Ανάπτυξη εφαρμογών στο Υπολογιστικό Νέφος

Σε ένα περιβάλλον υπολογιστικής νέφους (και κατά συνέπεια σε όλο το φάσμα του υπολογιστικού νέφους) η παραδοσιακή έννοια του μονολιθικού server ο οποίος εξυπηρετεί μία μόνο εφαρμογή ή ένα σύνολο συμβατών εφαρμογών αποτελεί παρελθόν. Πλέον σε έναν server χρειάζεται να φιλοξενηθούν εφαρμογές που έχουν διαφορετικές απαιτήσεις λογισμικού (dependencies) ή ακόμα και λειτουργικό σύστημα. Επίσης, χρειάζεται συχνά η δυναμική εκτέλεση της εφαρμογής σε διαφορετικούς servers.

1.1.1 Virtualisation

Απάντηση σε αυτό έρχεται να δώσει η τεχνολογία της εικονικοποίησης (virtualisation). Οι εφαρμογές εκτελούνται σε εικονικές μηχανές (Virtual Machines ή VMs), οι οποίες επιτρέπουν την εκτέλεση διαφορετικών λειτουργικών συστημάτων στην ίδια φυσική μηχανή. Για να επιτευχθεί αυτό ο server εκτελεί τον hypervisor, ένα λογισμικό το οποίο μεταφράζει τις εντολές συστήματος των εικονικών μηχανών (guests) σε εντολές του φυσικού συστήματος (host) [11]. Παρά τη χρησιμότητα τους, οι εικονικές μηχανές παρουσιάζουν ορισμένους περιορισμούς. Αρχικά, οι πόροι που χρησιμοποιεί μια εικονική μηχανή είναι στατικοί. Συνεπώς, δεν έχουν δυνατότητα κλιμάκωσης με τις ανάγκες της εφαρμογής που εξυπηρετούν. Παράλληλα, κάθε μηχανή εκτελεί μία πλήρη εγκατάσταση του εκάστοτε λογισμικού, με αποτέλεσμα να καταλαμβάνουν μεγάλο αριθμό υπολογιστικών πόρων. Ειδικά σε εφαρμογές που χρειάζεται να εκτελούνται πολλές από αυτές ταυτόχρονα, ή όπου οι πόροι του host είναι περιορισμένοι, όπως συμβαίνει σε υποδομές fog και edge, η χρήση τους είναι απαγορευτική.[12]



Σχήμα 1.1: *Σύγκριση συστημάτων βασισμένων σε VMs (αριστερά) και Containers (δεξιά)*

1.1.2 Containerisation

Ως λύση στους περιορισμούς των εικονικών μηχανών εμφανίζεται η τεχνολογία containerisation.[13] Σε αντίθεση με τις εικονικές μηχανές, οι εφαρμογές εκτελούνται σε containers, τα οποία αποτελούν μία ελαφρύτερη και δυναμικότερη εκδοχή των πρώτων. Ένα container ενώνει την εφαρμογή μαζί με τις απαιτήσεις της σε ένα πακέτο, το οποίο μπορεί να εκτελεσθεί μαζί με άλλα πακέτα πάνω από ένα κοινό πυρήνα λειτουργικού συστήματος. Αποτελούν δηλαδή χωριστές διεργασίες και όχι αυτόνομους υπολογιστές. Έτσι, κάθε container έχει τη δυνατότητα να καταλαμβάνει δυναμικά πόρους και να έχει μικρότερο μέγεθος από αυτό της αντίστοιχης εικονικής μηχανής, διατηρώντας παράλληλα την απομόνωση από το υπόλοιπο περιβάλλον και τη δυνατότητα μεταφοράς σε διαφορετικά φυσικά συστήματα [14].

1.2 Ενορχήστρωση Containerised Εφαρμογών

Ως ενορχήστρωση (orchestration) ορίζεται η αυτοματοποιημένη διαμόρφωση, ο συντονισμός και η διαχείριση των συστημάτων υπολογιστών και του λογισμικού. Σχετικά με την ανάπτυξη containerised εφαρμογών, η ενορχήστρωση αναφέρεται στην αυτοματοποιημένη διαχείριση των containerized εφαρμογών που εκτελούνται στο σύστημα. Εργασίες που καλύπτονται από την ενορχήστρωση αποτελούν η εκτέλεση (deployment), η τοποθέτηση (placement) και η κλιμάκωση (scaling) των εφαρμογών, αλλά και η διανομή δικτυακής κίνησης (load balancing). Για την ενορχήστρωση containerised εφαρμογών, αναπτύχθηκε και αποτελεί κυρίαρχη λύση στον κλάδο το λογισμικό Kubernetes.

1.2.1 Το λογισμικό Kubernetes

Το Kubernetes είναι ένα σύστημα ανοικτού κώδικα για την αυτοματοποίηση της ανάπτυξης, της κλιμάκωσης και της διαχείρισης containerised εφαρμογών. Αρχικά σχεδιάστηκε από την Google, αλλά το έργο συντηρείται πλέον από μια παγκόσμια κοινότητα συνεργατών και ανήκει στο Cloud Native Computing Foundation. Διαθέτει ένα μεγάλο, ταχέως αναπτυσσόμενο οικοσύστημα. Οι υπηρεσίες, η υποστήριξη και τα εργαλεία του Kubernetes είναι ευρέως διαθέσιμα [9].

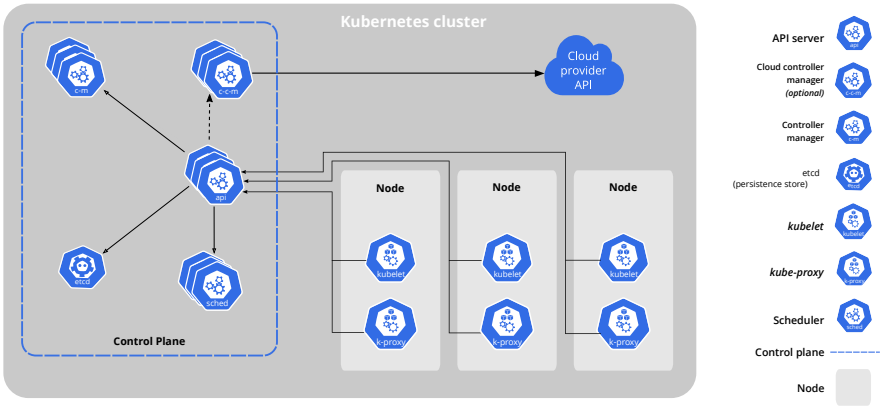
Το Kubernetes συγκεντρώνει έναν ή περισσότερους υπολογιστές, είτε φυσικούς είτε εικονικούς, σε μία συστάδα (cluster) η οποία μπορεί να εκτελεί containerised εφαρμογές σαν ένα ενιαίο σύστημα. Μπορεί να συνδυαστεί με διάφορα προγράμματα εκτέλεσης containerised εφαρμογών (container runtimes), όπως το Docker, το containerd και το CRI-O. Η καταλληλότητά του για την εκτέλεση και διαχείριση φόρτων εργασίας διαφόρων μεγεθών και ειδών καθώς και η συμβατότητα και η επεκτασιμότητα του έχουν οδηγήσει στην ευρεία υιοθέτησή του.

Σημείωση: Για την υλοποίηση και τον έλεγχο της παρούσας εργασίας γίνεται εκτενής χρήση του kubernetes και του περιβάλλοντος του. Για έννοιες που δεν καλύπτονται από το κεφάλαιο αυτό, καθώς και για τη χρήση των εργαλείων και των εννοιών που αναφέρονται παρακάτω ανατρέξτε στην επίσημη τεκμηρίωση του kubernetes, η οποία βρίσκεται στον παρακάτω σύνδεσμο https://kubernetes.io/docs/

1.2.2 Αρχιτεκτονική ενός Kubernetes Cluster

Ένας Kubernetes Cluster αποτελείται από ένα σύνολο υπολογιστικών συστημάτων, που ονομάζονται κόμβοι (nodes ή worker nodes), που εκτελούν τις containerised εφαρμογές. Κάθε Cluster έχει τουλάχιστον έναν worker node.

Οι worker nodes φιλοξενούν τα Pods που αποτελούνται από ένα ή περισσότερα containers. Το επίπεδο ελέγχου (control plane) διαχειρίζεται τους worker nodes και τα Pods μέσα στον cluster. Σε περιβάλλοντα παραγωγής, το control plane εκτελείται συνήθως σε πολλούς υπολογιστές και ένας cluster συνήθως αποτελείται από πολλά nodes, παρέχοντας ανοχή σε σφάλματα και υψηλή διαθεσιμότητα.



Σχήμα 1.2: *Αρχιτεκτονική ενός Kubernetes Cluster*

Αναλυτικότερα τα μέρη που απαρτίζουν τον cluster είναι τα εξής:

Μέρη του Control Plane:

- **kube-apiserver**

  Ο API server είναι ένα στοιχείο του control plane του Kubernetes που εκθέτει το Kubernetes API (βλ. 1.2.3). Ο API server είναι το front-end για το control plane του Kubernetes. Η κύρια υλοποίηση ενός API server του Kubernetes είναι ο kube-apiserver. Ο kube-apiserver έχει σχεδιαστεί για να κλιμακώνεται οριζόντια - δηλαδή, να κλιμακώνεται με την ανάπτυξη περισσότερων instances.

- **etcd**

  Συνεπής και υψηλής διαθεσιμότητας αποθηκευτικός χώρος τύπου key - value που χρησιμοποιείται ως αποθηκευτικός χώρος υποστήριξης του Kubernetes για όλα τα δεδομένα του cluster.

- **kube-scheduler**

  Στοιχείο του control-plane που παρακολουθεί για νέα Pods χωρίς καθορισμένο node και επιλέγει ένα node για να εκτελεσθούν.

  Οι παράγοντες που λαμβάνονται υπόψη για τις αποφάσεις scheduling περιλαμβάνουν: ατομικές και συλλογικές απαιτήσεις πόρων, περιορισμούς υλικού/λογισμικού/πολιτικής, προδιαγραφές affinity και anti-affinity, τοπικότητα δεδομένων, παρεμβολές μεταξύ φόρτου εργασίας και προθεσμίες.

- **kube-controller-manager**

  Στοιχείο του control plane που εκτελεί διεργασίες ελεγκτών (controllers). Σε έναν cluster υπάρχουν πολλοί διαφορετικοί τύποι controllers, όπως παραδείγματος χάριν ο node controller, ο οποίος είναι υπεύθυνος για την παρατήρηση και την ανταπόκριση όταν τα nodes αποτυγχάνουν, ή ο job controller, ο οποίος παρακολουθεί για αντικείμενα διεργασιών Jobs που αντιπροσωπεύουν εφάπαξ εργασίες και στη συνέχεια δημιουργεί Pods για την εκτέλεση αυτών των εργασιών μέχρι την ολοκλήρωσή τους.

  Σε λογικό επίπεδο, κάθε ελεγκτής είναι μια ξεχωριστή διεργασία. Όμως για να μειωθεί η πολυπλοκότητα, μεταγλωττίζονται όλοι σε ένα ενιαίο εκτελέσιμο και εκτελούνται ως μια ενιαία διεργασία.

- **cloud-controller-manager**

  Στοιχείο του control plane του Kubernetes που ενσωματώνει τη λογική ελέγχου που σχετίζεται με το cloud. Επιτρέπει στον cluster να συνδέεται με το API του παρόχου του νέφους (cloud provider) και διαχωρίζει τα στοιχεία που αλληλεπιδρούν με την εν λόγω πλατφόρμα νέφους από τα στοιχεία που αλληλεπιδρούν μόνο με τον cluster.

  Ο cloud-controller-manager εκτελεί μόνο ελεγκτές που σχετίζονται με τον πάροχο του cloud στον οποίο φιλοξενείται ο cluster. Εάν ο Kubernetes εκτελείται σε τοπικές

(on premise) υποδομές ή σε προσωπικό υπολογιστή (π.χ. για λόγους εκμάθησης/ δοκιμής), ο cluster δεν έχει cloud controller manager .

Όπως και με τον kube-controller-manager, ο cloud-controller-manager συνδυάζει διάφορους λογικά ανεξάρτητους ελεγκτές σε ένα ενιαίο εκτελέσιμο που εκτελείται ως μια ενιαία διεργασία.

Μέρη ενός node

- **kubelet**

  Ένας πράκτορας (agent) που εκτελείται σε κάθε node του custer. Διασφαλίζει ότι τα containers εκτελούνται σε ένα Pod.

  Το kubelet λαμβάνει ένα σύνολο PodSpecs που παρέχονται μέσω διαφόρων μηχανισμών και διασφαλίζει ότι τα containers που περιγράφονται σε αυτά τα PodSpecs λειτουργούν και είναι υγιή. Το kubelet δε διαχειρίζεται containers που δε δημιουργήθηκαν από το Kubernetes.

- **kube-proxy**

  ο kube-proxy είναι ένας διακομιστής μεσολάβησης δικτύου (network proxy) που εκτελείται σε κάθε κόμβο του cluster.

  Ο kube-proxy διατηρεί κανόνες δικτύου στους κόμβους. Αυτοί οι κανόνες δικτύου επιτρέπουν τη δικτυακή επικοινωνία με τα Pods από συνεδρίες δικτύου εντός ή εκτός του cluster.

  Το kube-proxy χρησιμοποιεί το επίπεδο φιλτραρίσματος πακέτων (packet filtering layer) του λειτουργικού συστήματος, εάν αυτό υπάρχει και είναι διαθέσιμο. Διαφορετικά, ο kube-proxy προωθεί την κυκλοφορία μόνος του.

- **Container runtime**

  Ένα θεμελιώδες συστατικό που επιτρέπει στο Kubernetes να διαχειρίζεται τα containers. Είναι υπεύθυνο για τη διαχείριση της εκτέλεσης και του κύκλου ζωής των containers εντός του περιβάλλοντος Kubernetes.

  Το Kubernetes υποστηρίζει container runtimes όπως το Docker, το containerd, το CRI-O και οποιαδήποτε άλλη υλοποίηση του Kubernetes CRI (Container Runtime Interface).

Τα παραπάνω μέρη καθώς και οι σχέσεις μεταξύ τους φαίνονται αναλυτικά στην εικόνα 1.2

1.2.3 Kubernetes API

Ο πυρήνας του control plane του Kubernetes είναι ο API server. Ο API server εκθέτει ένα HTTP API που επιτρέπει στους τελικούς χρήστες, τα διάφορα τμήματα του cluster και εξωτερικά στοιχεία να επικοινωνούν μεταξύ τους.

Το API του Kubernetes επιτρέπει την παρακολούθηση και τη διαχείριση της κατάστασης των API objects στο Kubernetes (για παράδειγμα: Pods, Deployments, Namespaces, ConfigMaps και Events).

Οι περισσότερες λειτουργίες μπορούν να εκτελεστούν μέσω της διεπαφής γραμμής εντολών kubectl ή άλλων εργαλείων γραμμής εντολών, όπως το kubeadm, τα οποία με τη σειρά τους χρησιμοποιούν το API. Ωστόσο, είναι δυνατή η πρόσβαση στο API απευθείας χρησιμοποιώντας REST calls. Το Kubernetes παρέχει ένα σύνολο client libraries για την ανάπτυξη εφαρμογών που χρησιμοποιούν το API του Kubernetes.

1.2.4 Kubernetes Objects

Τα αντικείμενα (objects) είναι μόνιμες οντότητες στο περιβάλλον του Kubernetes. Το Kubernetes χρησιμοποιεί αυτές τις οντότητες για να αναπαριστά την κατάσταση του cluster. Συγκεκριμένα, μπορούν να περιγράψουν ποιες εφαρμογές εκτελούνται (και σε ποια nodes), τους πόρους που είναι διαθέσιμοι σε αυτές τις εφαρμογές, τις πολιτικές γύρω από τον τρόπο συμπεριφοράς αυτών των εφαρμογών, όπως πολιτικές επανεκκίνησης, αναβάθμισης και ανοχής σε σφάλματα. Τα objects ανήκουν σε πολλά είδη (kinds), όπως Pods, Deployments, Services μεταξύ άλλων. Το kubernetes API ομαδοποιεί objects ίδιου είδους σε API endpoinds τα οποία ονομάζονται πόροι (resources). Παραδείγματος χάριν, για τη διαχείριση των Pods το API έχει το resource `/pods`. Υπάρχει επίσης η δυνατότητα για δημιουργία προσαρμοσμένων πόρων (custom resources).

Ο τρόπος που διαχειρίζεται το Kubernetes τα αντικείμενα είναι δηλωτικός. Με τη δημιουργία ενός αντικειμένου, ορίζεται στο σύστημα Kubernetes πώς πρέπει να μοιάζει το συγκεκριμένο αντικείμενο στον cluster - αυτή είναι η επιθυμητή κατάσταση του. Από τη στιγμή που θα δημιουργηθεί, το Kubernetes εργάζεται συνεχώς για τη διασφάλιση της ύπαρξης αυτού του αντικειμένου και τη διατήρηση της κατάστασής του.

Για τη διαχείριση των αντικειμένων του Kubernetes, δηλαδή για τη δημιουργία, την τροποποίηση ή τη διαγραφή τους) θα πρέπει να χρησιμοποιηθεί το API του Kubernetes (βλ 1.2.3).

Σχεδόν κάθε αντικείμενο του Kubernetes περιλαμβάνει δύο ένθετα πεδία που διέπουν τη διαμόρφωση του αντικειμένου: το object spec και το object status.

Για τα αντικείμενα που έχουν spec, πρέπει να οριστεί κατά τη δημιουργία του αντικειμένου, η επιθυμητή κατάστασή του, μέσω μιας περιγραφής των χαρακτηριστικών που χρειάζεται να έχει ο πόρος.

Το status περιγράφει την τρέχουσα κατάσταση του αντικειμένου, η οποία παρέχεται και ενημερώνεται από το σύστημα Kubernetes και τα μέρη του. Το control plane του Kubernetes διαχειρίζεται την πραγματική κατάσταση κάθε αντικειμένου, ώστε να ταιριάζει με την επιθυμητή κατάσταση του.

Για παράδειγμα, στο Kubernetes, ένα Deployment είναι ένα αντικείμενο που μπορεί να αντιπροσωπεύει μια εφαρμογή που εκτελείται στον cluster. Όταν δημιουργείται το Deployment, μπορεί να έχει οριστεί στο Deployment spec ότι πρέπει να εκτελούνται τρία αντίγραφα της εφαρμογής. Το σύστημα Kubernetes διαβάζει το Deployment spec και εκκινεί τρία instances της επιθυμητής εφαρμογής, επικαιροποιώντας το status της ώστε να ταιριάζει με το spec

ΚΩΔΙΚΑΣ 1.1: *Παράδειγμα manifest για Kubernetes Deployment.*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

της. Εάν κάποιο από αυτά τα instances αποτύχει, το σύστημα Kubernetes ανταποκρίνεται στη διαφορά μεταξύ spec και status κάνοντας μια διόρθωση - ξεκινώντας στην προκειμένη περίπτωση ένα νέο instance για να το αντικαταστήσει.

Για τη δημιουργία ενός αντικειμένου στο Kubernetes πρέπει να παρέχεται το object spec που περιγράφει την επιθυμητή κατάστασή του, καθώς και κάποιες βασικές πληροφορίες για το αντικείμενο (όπως ένα όνομα). Όταν χρησιμοποιείται το API του Kubernetes για να δημιουργηθεί το αντικείμενο (είτε απευθείας είτε μέσω kubectl), το εν λόγω αίτημα API πρέπει να περιλαμβάνει αυτές τις πληροφορίες ως JSON στο σώμα του αιτήματος. Τις περισσότερες φορές, οι πληροφορίες παρέχονται στο kubectl σε αρχείο γνωστό ως manifest. Κατά σύμβαση, τα manifests είναι σε μορφή YAML (ή σπανιότερα σε μορφή JSON). Εν συνεχεία το kubectl μετατρέπει τις πληροφορίες από ένα manifest σε JSON κατά την υποβολή του αιτήματος API μέσω HTTP. Παράδειγμα manifest για ένα Deployment φαίνεται στον Κώδικα 1.1

1.2.5 Διαχείριση πρόσβασης σε ένα Kubernetes Cluster

Σε περίπτωση όπου δε χρησιμοποιείται άμεσα το API, αλλά το εργαλείο kubectl, δε χρειάζεται να οριστεί η διεύθυνση στην οποία λαμβάνει αιτήματα το API. Το kubectl χρησιμοποιεί τα αρχεία διαμόρφωσης kubeconfig για να βρει τις απαραίτητες πληροφορίες που χρειάζεται για να συνδεθεί με τον cluster, όπως η διεύθυνση, τα πιστοποιητικά ασφαλείας κ.λ.π. Προεπιλεγμένα, το kubectl αναζητά αρχείο με όνομα config στο directory `home/chris/.kube`. Εναλλακτικά αρχεία διαμόρφωσης kubeconfig μπορούν να οριστούν είτε μέσω της μεταβλητής περιβάλλοντος `KUBECONFIG` είτε μέσω της παραμέτρου γραμμής εντολών

`-kubeconfig`.

1.3 Multi-Cluster Υποδομές

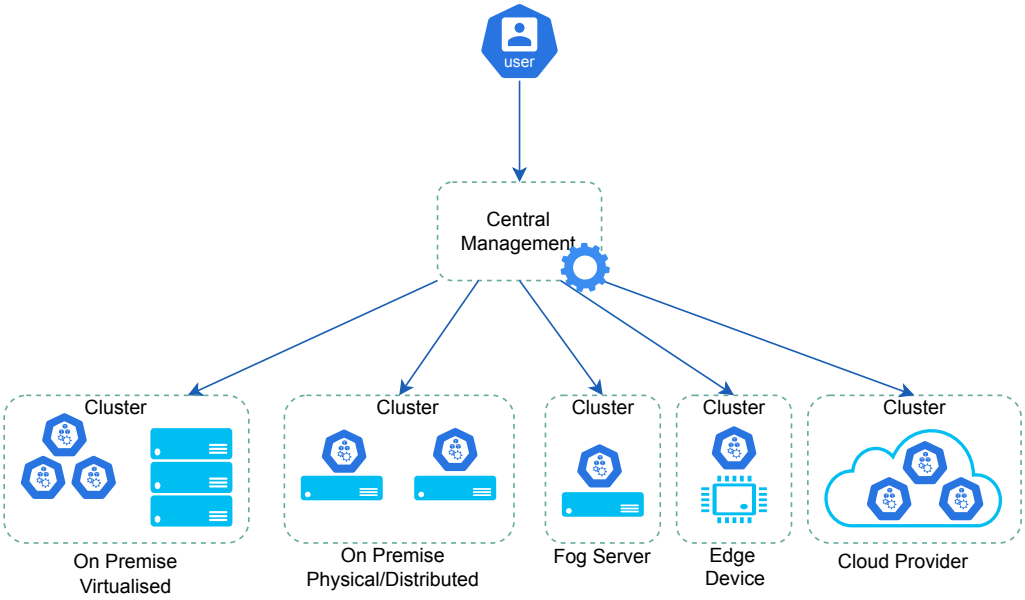
Όπως έχει ήδη αναφερθεί, το Kubernetes αποτελεί απαραίτητο εργαλείο για την ενορχήστρωση συστημάτων βασισμένων σε containerised εφαρμογές. Για αυτό τον λόγο χρησιμοποιείται από τη συντριπτική πλειοψηφία τόσο των εταιρειών όσο και των παρόχων cloud. Αξίζει να σημειωθεί πως, ένας Kubernetes cluster επαρκεί για την υλοποίηση απλών Edge και Fog υποδομών. Παρόλα αυτά δεν παύει να διέπεται από ορισμένους περιορισμούς, που κάνουν τη χρήση ενός Kubernetes cluster για την υλοποίηση πολύπλοκων συστημάτων υπολογιστικού νέφους δύσκολη.

Αρχικά, ένα Kubernetes node δεν μπορεί να διαχειριστεί πάνω από 110 pods. Αυτό δεν αποτελεί απαραίτητα πρόβλημα για μία cloud υποδομή, καθώς λύνεται με την αντικατάσταση μεγάλων nodes από μικρότερους. Ένας φυσικός (bare metal) node μπορεί να χωριστεί σε μικρότερους λογικούς Nodes με τη χρήση VMs, ενώ ένα μεγάλο εικονικό node μπορεί άμεσα να αντικατασταθεί από μικρότερα. Επίσης, ένας kubernetes cluster δεν μπορεί να έχει πάνω από 5000 nodes. Ταυτόχρονα, δεν μπορεί να διατηρεί συνολικά (ανάμεσα σε πολλά nodes) πάνω από 150000 pods, ούτε πάνω από 300000 containers [9]. Οι αριθμοί αυτοί είναι αρκετά μεγάλοι για ένα απλό cloud σύστημα. Σε ένα περιβάλλον edge όμως, όπου ο αριθμός των συσκευών που απαρτίζουν τον cluster ανέρχονται σε εκατομμύρια ή δισεκατομμύρια συσκευές, είναι προφανές ότι η χρήση ενός μόνο cluster για την ενορχήστρωση εφαρμογών σε ένα τέτοιο περιβάλλον είναι αδύνατη [10].

Όμως, ακόμα και για ένα μικρότερο σύστημα, που δεν εξαντλεί τους περιορισμούς του Kubernetes, η χρήση περισσότερων από ένα cluster μπορεί να έχει αρκετά πλεονεκτήματα. Αρχικά, σε περιβάλλοντα όπου το Kubernetes χρησιμοποιείται ως υπηρεσία μέσω cloud provider, είναι πολλές φορές χρήσιμο να μην εξαρτάται η υποδομή από έναν μόνο πάροχο, ούτως ώστε να μην επηρεαστεί η υποδομή σε περίπτωση που υπάρξει πρόβλημα στον συγκεκριμένο πάροχο, είτε τεχνικό (π.χ. σφάλμα) είτε μη-τεχνικό (π.χ. ανατίμηση υπηρεσίας). Λύση στο παραπάνω δίνει η διατήρηση πολλών clusters σε διαφορετικούς παρόχους (multi provider). Επίσης, λόγω της χρήσης REST api για την επικοινωνία μεταξύ των μερών του cluster, η αποδοτική λειτουργία ενός cluster απαιτεί μικρές καθυστερήσεις δικτύου ανάμεσα στα nodes. Για τον λόγο αυτό, σε γεωγραφικά κατανεμημένα συστήματα, όπως συστήματα edge ή fog, είναι πολλές φορές προτιμότερο η υποδομή να αποτελείται από γεωγραφικά κατανεμημένους clusters που επικοινωνούν μεταξύ τους παρά από έναν ενιαίο cluster με γεωγραφικά κατανεμημένα nodes. Τέλος, για λόγους ασφάλειας, ιδιωτικότητας ή αποφυγής σφαλμάτων, είναι πολλές φορές χρήσιμο μέρη μίας υποδομής να μην επικοινωνούν μεταξύ τους. Για παράδειγμα, είναι χρήσιμο ένα περιβάλλον ανάπτυξης ή ένα περιβάλλον δοκιμών να μην επικοινωνούν με ένα περιβάλλον παραγωγής. Επίσης, για εφαρμογές υψηλής ασφαλείας, είναι προτιμότερο τα δεδομένα να μη μεταφέρονται εκτός μίας γεωγραφικά περιορισμένης υποδομής, όπως π.χ. έναν server, ένα δίκτυο ή μία χώρα.

Παρότι οι υποδομές Multi-Cluster προσφέρουν σημαντικά οφέλη, η υλοποίησή τους φέρει και ορισμένες προκλήσεις. Η διαχείριση πολλαπλών clusters εισάγει πρόσθετη πολυπλοκότητα στο σύστημα. Η διαμόρφωση, η ανάπτυξη και η παρακολούθηση εφαρμογών

σε πολλούς clusters απαιτεί πρόσθετα εργαλεία και διαδικασίες. Επιπλέον, οι clusters μπορεί να διαφέρουν ως προς το υλικό, το λογισμικό και τη διαμόρφωση, απαιτώντας έτσι διαφορετικές ροές εργασίας ανά cluster. Για αυτούς τους λόγους, ειδικά σε περιβάλλοντα όπου οι επιμέρους clusters παρουσιάζουν διαφορές, χρειάζεται η κεντρική διαχείριση των clusters. Αυτό σημαίνει ότι μέσα από κατάλληλα εργαλεία και υποδομές, ένα σύστημα που απαρτίζεται από διαφορετικούς clusters είναι διαχειρίσιμο μέσω μίας ενιαίας διεπαφής, ανεξάρτητα από τη δομή και τις απαιτήσεις των επιμέρους clusters. Μάλιστα πέρα από την απλούστευση της υποδομής, η κεντρική διαχείριση ενός Multi Cluster συστήματος φαίνεται να παρουσιάζει αυξημένη απόδοση αναφορικά με την ταχύτητα εκτέλεσης, τη χρήση εύρους ζώνης και την καθυστέρηση, σε σχέση με συστήματα Multi Cluster όπου αυτή απουσιάζει. [15]



Σχήμα 1.3: *Παράδειγμα Multi-Cluster Υποδομής με κεντρική διαχείριση*

Συνοψίζοντας, είναι πολλοί οι λόγοι για τους οποίους η χρήση περισσότερων από έναν Kubernetes clusters είναι χρήσιμη για τη λειτουργία μίας υποδομής. Αυτό φαίνεται εξάλλου και από το γεγονός ότι, σύμφωνα με στατιστικά του CNCF για το 2022, η συντριπτική πλειοψηφία των οργανισμών που χρησιμοποιούν το Kubernetes για την ενορχήστρωση της υποδομής τους, χρησιμοποιεί πάνω από έναν cluster.[16]

Κεφάλαιο 2

Το λογισμικό Karmada

Στο παρακάτω κεφάλαιο αναλύεται το λογισμικό Karmada. Αρχικά, γίνεται αναφορά στην αρχιτεκτονική του Karmada και τις ομοιότητες που αυτή έχει με το Kubernetes. Εν συνεχεία παρουσιάζονται οι βασικές έννοιες και οι τρόποι χρήσης του.

2.1 Εισαγωγή

Το Karmada (Kubernetes - Armada) είναι ένα σύστημα δημιουργίας και διαχείρισης Multi Cluster υποδομών. Αποτελεί ανοιχτού κώδικα λογισμικό υπό την αιγίδα του Cloud Native Computing Foundation (CNCF). Στόχος του είναι η δυνατότητα για Multi Cluster ενορχήστρωση υποδομών χωρίς αλλαγή των εφαρμογών σε σχέση με ένα single cluster σύστημα. Είναι πλήρως συμβατό με το περιβάλλον του Kubernetes. Επίσης, είναι ανεξάρτητο από την υποδομή που φιλοξενεί τους Clusters. Όλα αυτά και πολλά ακόμη βοηθούν στην αποδοχή του από την κοινότητα της υπολογιστικής νέφους για Multi Cluster λύσεις [17].

2.2 Αρχιτεκτονική

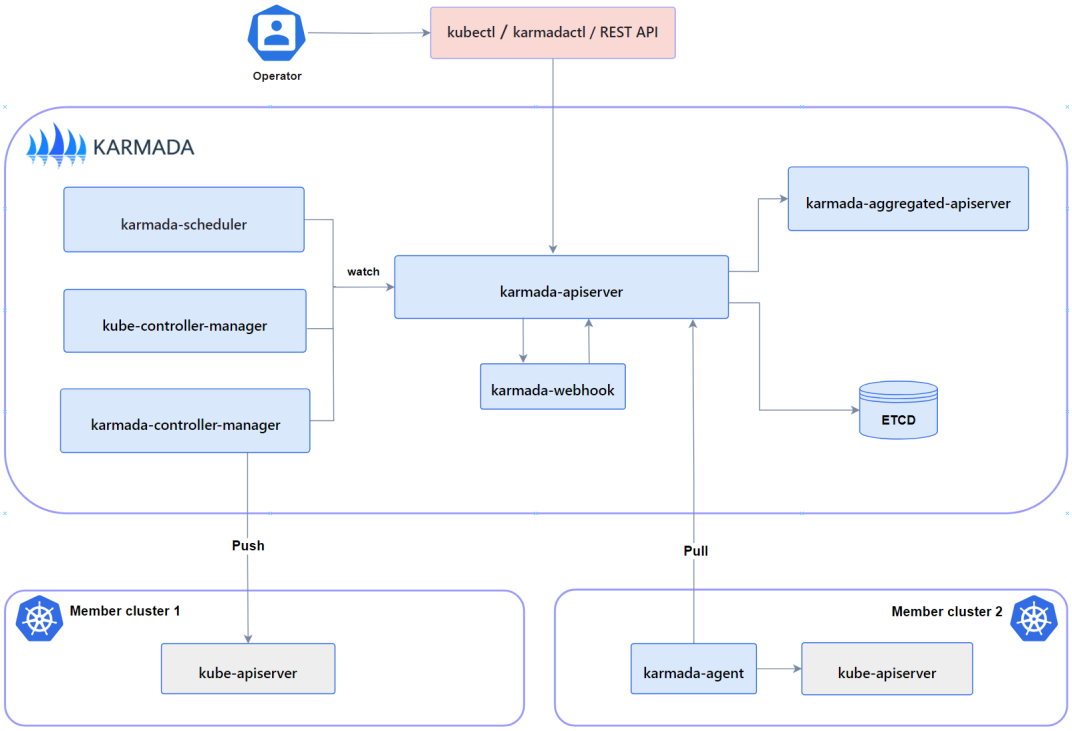
Η αρχιτεκτονική του Karmada ακολουθεί σε πολύ μεγάλο βαθμό την αρχιτεκτονική του Kubernetes. Όπως σε ένα Kubernetes cluster το control plane διαχειρίζεται πολλά nodes, στο Karmada υπάρχει το karmada control plane το οποίο διαχειρίζεται πολλούς member clusters. Αναλογία παρουσιάζουν και τα επιμέρους μέρη του karmada control plane που φαίνεται και στην εικόνα 2.1.

Παρακάτω παρουσιάζονται αναλυτικά τα επιμέρους μέρη του karmada control plane και των member clusters, στην περίπτωση που αυτοί συνδέονται μέσω pull mode (βλ. 2.3.1).

- **karmada-apiserver**

  Ο API server είναι ένα στοιχείο του karmada control plane που εκθέτει το API του Karmada εκτός από το API του Kubernetes. Ο API server είναι το front end του control plane του Karmada.

  Ο Karmada API server χρησιμοποιεί απευθείας την υλοποίηση του kube-apiserver από το Kubernetes, και αυτός είναι ο λόγος για τον οποίο το Karmada είναι φυσικά συμβατό με το Kubernetes API. Αυτό καθιστά την ενσωμάτωση με το οικοσύστημα Kubernetes πολύ απλή για το Karmada, όπως για παράδειγμα το να επιτρέπεται στους χρήστες να χρησιμοποιούν το kubectl για να χειρίζονται το Karmada



Σχήμα 2.1: *Αρχιτεκτονική του Karmada*

- **karmada-aggregated-apiserver**

  Ο aggregated API server είναι ένας εκτεταμένος API server που υλοποιείται χρησιμοποιώντας την τεχνολογία Kubernetes API Aggregation Layer. Προσφέρει το Cluster API και συναφή επιμέρους resources, όπως τα cluster/status και cluster/proxy και υλοποιεί προηγμένες δυνατότητες όπως το Aggregated Kubernetes API που μπορεί να χρησιμοποιηθεί για την πρόσβαση σε member clusters μέσω του karmada-apiserver.

- **kube-controller-manager**

  Το Karmada κληρονομεί μερικούς ελεγκτές (controllers) από την επίσημη εικόνα του Kubernetes για να διατηρήσει μια συνεπή εμπειρία και συμπεριφορά του χρήστη.

  Σημείωση: Όταν οι χρήστες υποβάλλουν Deployment ή άλλους τυπικούς πόρους (resources) του Kubernetes στον karmada-apiserver, καταγράφονται αποκλειστικά στο etcd του Karmada control plane. Στη συνέχεια, οι πόροι αυτοί συγχρονίζονται με τον member cluster. Ωστόσο, αυτά τα Deployments δεν υποβάλλονται σε διαδικασίες συγχρονισμού (όπως η δημιουργία pod) στον cluster που φιλοξενεί το control plane του Karmada.

- **karmada-controller-manager**

  Ο karmada-controller-manager εκτελεί διάφορους προσαρμοσμένους controllers που δεν περιέχονται στον kube controller manager. Οι controllers παρακολουθούν τα αντικείμενα του Karmada και στη συνέχεια επικοινωνούν με τους API servers των member clusters για να δημιουργήσουν Kubernetes objects.

- **karmada-scheduler**

  Ο karmada-scheduler είναι υπεύθυνος για τη χρονοδρομολόγηση (scheduling) των kubernetes objects στους member clusters.

  Ο scheduler καθορίζει ποιοι clusters είναι έγκυροι για τοποθέτηση (placement) για κάθε πόρο στην ουρά χρονοδρομολόγησης σύμφωνα με τους περιορισμούς και τους διαθέσιμους πόρους. Στη συνέχεια, ο scheduler κατατάσσει κάθε έγκυρο cluster και δεσμεύει τον πόρο στον καταλληλότερο cluster.

- **karmada-webhook**

  Τα karmada-webhooks είναι callbacks HTTP που λαμβάνουν αιτήματα API του Karmada/Kubernetes και εκτελούν μία λειτουργία με βάση τα αιτήματα αυτά. Μπορούν να οριστούν δύο τύποι karmada-webhook, το validating webhook και το mutating webhook.

  Τα mutating webhooks καλούνται πρώτα και μπορούν να τροποποιήσουν τα αντικείμενα που αποστέλλονται στον karmada-apiserver για να επιβάλλουν προσαρμοσμένες προεπιλογές. Αφού ολοκληρωθούν όλες οι τροποποιήσεις αντικειμένων και αφού το εισερχόμενο αντικείμενο επικυρωθεί από τον karmada-apiserver, καλούνται τα validating webhooks και μπορούν να απορρίψουν αιτήσεις για την επιβολή προσαρμοσμένων πολιτικών.

- **etcd**

  Κατά αναλογία με το etcd του kubernetes, το etcd είναι συνεπής και υψηλής διαθεσιμότητας αποθηκευτικός χώρος τύπου key - value που χρησιμοποιείται ως αποθηκευτικός χώρος υποστήριξης του Karmada για όλα τα objects API του Karmada/Kubernetes.

- **karmada-agent** Δεν αποτελεί μέρος του ίδιου του control plane αλλά των member clusters που είναι εγγεγραμμένοι μέσω pull mode (βλ 2.3.1).

  Μπορεί να καταχωρίσει έναν cluster στο Karmada control plane και να συγχρονίσει τα manifests από το Karmada control plane στον member cluster. Επιπλέον, συγχρονίζει την κατάσταση του member cluster και τα manifests με το Karmada control plane.

Όπως φαίνεται από τα παραπάνω η αρχιτεκτονική του karmada είναι ανάλογη του kubernetes. Επίσης, χάρη στο karmada-apiserver είναι συμβατό με το API του kubernetes, κάνοντας τη χρήση του εύκολη για χρήστες εξοικειωμένους με το περιβάλλον του kubernetes. Για τη διαχείριση του karmada μπορούν να οριστούν objects (βλ 2.4) χρησιμοποιώντας είτε απευθείας το API είτε μέσα από CLI όπως το kubectl. Όπως και για το kubernetes τα objects αυτά ορίζονται με χρήση manisfests τα οποία είναι σε μορφή YAML ή JSON.

2.3 Περιβάλλον του Karmada

2.3.1 Εγκατάσταση και Συμμετοχή Clusters

Τόσο για την εγκατάσταση του karmada στον cluster, όσο και για τη διαχείριση των member clusters είναι απαραίτητη η εγκατάσταση ενός εκ των δύο CLI εργαλείων του karmada: το karmadactl, το οποίο αποτελεί αυτόνομο εκτελέσιμο, ή το kubectl-karmada το οποίο αποτελεί πρόσθετο (plugin) για το kubectl. Η χρήση και των δύο είναι η ίδια.

Για παράδειγμα, για την εγκατάσταση του karmada control plane στον cluster που θα το φιλοξενεί, αρκεί να εκτελεσθεί η εντολή `karmadactl init` (ή `kubectl karmada init` για το kubectl-karmada), θεωρώντας ότι έχουμε ορίσει κατάλληλα το κατάλληλο αρχείο kubeconfig (βλ. 1.2.5).

Για την προσθήκη ενός member cluster στο σύστημα υπάρχουν δύο μέθοδοι, οι pull και push. Στη μέθοδο push, η οποία είναι και η απλούστερη, το karmada control plane επικοινωνεί άμεσα με το kubernetes API του member cluster. Αντίθετα, στη μέθοδο pull εγκαθίσταται ο karmada-agent στο control plane του member cluster, ο οποίος λειτουργεί σαν μεσάζοντας ανάμεσα στο control plane του karmada και του member cluster. Ο karmada-agent καταχωρεί τον cluster στο Karmada control plane και συγχρονίζει τα object manifests από το Karmada control plane στον member clustetr και το αντίστροφο.

2.3.2 Μοντελοποίηση Πόρων (Resource Modelling)

Για την επιτυχή τοποθέτηση objects στους member clusters είναι απαραίτητο το Karmada να γνωρίζει τους πόρους που διαθέτει κάθε member cluster. Αυτό γίνεται μέσω της μοντελοποίησης Πόρων (Resource Modelling). Το Karmada API εισάγει στο cluster resource το πεδίο ResourceSummary (βλ. κώδικας 2.1). Το πεδίο αυτό περιέχει πληροφορίες για τους συνολικούς πόρους (`resourceSummary.allocatable`) και τους κατειλημμένους πόρους (`resourceSummary.allocated`) τη στιγμή του request. Οι μετρικές που παρακολουθεί είναι η (δυνητική) χρήση του επεξεργαστή σε πυρήνες (`cpu`), η χρήση της μνήμης RAM σε bytes (`memory`) και ο αριθμός των pods (`pods`).

Τα παραπάνω μεγέθη χρησιμοποιούνται για τον υπολογισμό της βασικής μετρικής που χρησιμοποιεί το Karmada για την τοποθέτηση, τα AvailableReplicas. Η μετρική αυτή δηλώνει τον μέγιστο αριθμό από replicas ενός συγκεκριμένου object που μπορούν να δρομολογηθούν στον member cluster. Αποτελεί συνάρτηση των μεγεθών αυτών (cpu,memory,pods) και των απαιτήσεων του object στις αντίστοιχες μετρικές. Πιο συγκεκριμένα δίνεται από τη σχέση 2.1

$$AR = min(\frac{c_{total} - c_{allocated}}{c_{required}}, \frac{m_{total} - m_{allocated}}{m_{required}}, \frac{p_{total} - p_{allocated}}{p_{required}}) \tag{2.1}$$

όπου c cpu, m μνήμη ram και p pods, αφαιρώντας τους όρους που δεν υπάρχει απαίτηση, για αποφυγή της διαίρεσης με το 0. Για παράδειγμα, για ένα Deployment με ένα Pod και απαίτηση για 0.5 cpu, τα AvailableReplicas για τον παραπάνω member cluster είναι $AR = min(\frac{4-0.95}{0.5}, \frac{110-11}{1}) = min(6.1, 99) = 6.1$

ΚΩΔΙΚΑΣ 2.1: *Παράδειγμα Resource Summary*

```
resourceSummary:
    allocatable:
      cpu: "4"
      ephemeral-storage: 206291924Ki
      hugepages-1Gi: "0"
      hugepages-2Mi: "0"
      memory: 16265856Ki
      pods: "110"
    allocated:
      cpu: 950m
      memory: 290Mi
      pods: "11"
  #...
```

2.4 Objects του Karmada

2.4.1 Πολιτικές Διάδοσης (Propagation Policies)

Εφαρμόζοντας ένα object σε ένα multi cluster σύστημα βασισμένο στο karmada, αυτό δε δρομολογείται αυτόματα στους clusters, καθώς το karmada δεν έχει τις απαραίτητες πληροφορίες για τον τρόπο με τον οποίο θα γίνει ο επιθυμητός διαμοιρασμός. Οι πολιτικές διάδοσης (Propagation Policies ή PP) ή πολιτικές τοποθέτησης (Placement Policies) χρησιμοποιούνται για τη δρομολόγηση των objects στους member clusters. Έχουν τη δυνατότητα να αντιστοιχίζουν μία πολιτική με πάνω από ένα αντικείμενα, με αποτέλεσμα να μπορεί η ίδια πολιτική να χρησιμοποιηθεί για πολλά μέρη της εφαρμογής. Το spec μίας PP περιλαμβάνει δύο κύρια πεδία, τα `.spec.resourceSelectors` και `.spec.placement`. Το πεδίο `resourceSelectors` καθορίζει τα objects (ή resources) στα οποία απευθύνεται η συγκεκριμένη πολιτική. Το πεδίο `placement` καθορίζει τον τρόπο με τον οποίο γίνεται η δρομολόγηση. Τα πεδία του `placement` που μας αφορούν για την παρούσα εργασία είναι τα `.spec.placement.clusterAffinity`,`placement.clusterAffinities` και `placement.replicaScheduling`. Τα δύο πρώτα καθορίζουν τους clusters τους οποίους αφορά η συγκεκριμένη πολιτική. Το πεδίο `.spec.placement.clusterAffinity` ορίζει ένα υποσύνολο των member clusters χρησιμοποιώντας ένα ή περισσότερα από τα παρακάτω πεδία: LabelSelector, το οποίο επιλέγει clusters με βάση τα labels που ορίζονται στα metadata του cluster object, FieldSelector, το οποίο επιλέγει clusters με βάση τα fields που ορίζονται στο spec του cluster object, ClusterNames, το οποίο χρησιμοποιεί μόνο τους clusters που δίνονται και ExcludeClusters το οποίο χρησιμοποιεί όλους τους clusters εκτός από αυτούς που δίνονται.

Το πεδίο `.spec.placement.clusterAffinities` δίνει τη δυνατότητα για ομαδοποίηση πάνω από ένα Cluster Affinity σε ένα Afinity Group.

Αν λείπουν και τα δύο πεδία, η πολιτική αφορά όλους τους member clusters. Για λόγους πληρότητας αναφέρονται ονομαστικά και τα πεδία `.spec.placement.clusterTolerations` και

`.spec.placement.spreadConstraints`, με τα οποία δεν ασχολείται η παρούσα εργασία και λόγω της πολυπλοκότητας τους δεν αναλύονται περαιτέρω.

Το πεδίο `placement.replicaScheduling` καθορίζει τον τρόπο με τον οποίο τα αντίγραφα (replicas) του object κατανέμονται στους member clusters. Αυτός ορίζεται από το πεδίο `replocaScheduling.replicaSchedulingType`. Το συγκεκριμένο πεδίο έχει εξέχουσα σημασία για την παρούσα εργασία, καθώς κύριος σκοπός της είναι η μοντελοποίηση αυτών των τρόπων. Οι τιμές που μπορεί να έχει, και κατά αντιστοιχία οι μέθοδοι που μπορούν να εφαρμοστούν, είναι οι `Duplicated` και `Divided`.

Σημείωση: Στη συνέχεια της εργασίας όταν αναφερόμαστε σε πολιτικές μετάδοσης (PP) θα αναφερόμαστε συνήθως σε εναλλακτικές μεθόδους replica Scheduling και όχι σε μεμονωμένα objects.

Με τη σειρά της, μία Divided PP δέχεται το πεδίο `ReplicaDivisionPreference`, το οποίο δέχεται τις τιμές `Aggregated` και `Weighted`. Μία `Weighted` πολιτική δέχεται το πεδίο `WeightPreference`, το οποίο με τη σειρά του δέχεται τις τιμές `StaticWeightList` και `DynamicWeight`.

Duplicated PP

Η Duplicated , ή πολιτική αντιγραφής, είναι η απλούστερη από τις πολιτικές διάδοσης. Στην πολιτική αυτή, τα objects στα οποία εφαρμόζεται αντιγράφονται σε όλους τους member clusters. Για παράδειγμα, αν εφαρμοστεί σε ένα Deployment με n replicas, μετά την εφαρμογή θα δημιουργηθούν n replicas σε κάθε member cluster. Αξίζει να σημειωθεί ότι είναι η μόνη πολιτική διάδοσης που δημιουργεί συνολικά περισσότερα αντίγραφα από όσα ορίζονται από το manifest του αρχικού object. Έτσι, σε σύστημα με n replicas και C clusters ο συνολικός αριθμός των replicas σε όλο το multi cluster σύστημα είναι $C \cdot n$.

Divided Aggregated PP

Η Aggregated , ή πολιτική συγκέντρωσης, είναι η πολιτική σύμφωνα με την οποία τα replicas μοιράζονται σε όσο το δυνατόν λιγότερους member clusters, λαμβάνοντας υπόψη τη διαθεσιμότητα τους σε πόρους. Για να το επιτύχει αυτό εφαρμόζει greedy αλγόριθμο, σύμφωνα με τον οποίο ταξινομεί τους member clusters σε φθίνουσα σειρά με βάση τα Available Replicas (βλ.2.3.2) και δρομολογεί σε αυτόν με τα περισσότερα όσα replicas αυτός μπορεί να χωρέσει. Στη συνέχεια επαναλαμβάνει το βήμα με τον επόμενο στη σειρά cluster έως ότου δρομολογηθούν όλα τα replicas ή δεν υπάρχουν άλλοι clusters. Για παράδειγμα, αν εφαρμοστεί σε ένα Deployment με n replicas, σε σύστημα με member clustersc_1, c_2, c_3 με `AvailableReplicas` $Ar_{(C_1)} = n-i, Ar_{(C_2)} = i+c_1, Ar_{(C_3)} = c_2$ όπου $i+c_1 < n-i$, $c_2 < i$, $i \geq 0$ μετά την εφαρμογή θα δημιουργηθούν $R_{(C_1)} = n-i, R_{(C_2)} = i, R_{(C_3)} = 0$. Παρατηρούμε ότι ο αριθμός των συνολικών replicas είναι ίσος με $n-i+i = n$, δηλαδή ίσος με τον αριθμό των replicas στο αρχικό object. Σε ένα πιο απλό αριθμητικό παράδειγμα, για $n = 10, i = 3, c_1 = 3, c_2 = 2$ έχουμε $Ar_{(C_1)} = 7, Ar_{(C_2)} = 6, Ar_{(C_3)} = 2$ και μετά την εφαρμογή της πολιτικής θα δρομολογηθούν $R_{(C_1)} = 7, R_{(C_2)} = 3, R_{(C_3)} = 0$.

Σημείωση: Η συμπεριφορά αυτή, εξαιρώντας τη χρησιμοποίηση όσο το δυνατόν λιγότερων member clusters, δεν καθορίζεται ρητά από την τεκμηρίωση του Karmada, αλλά αποτελεί υπόθεση με βάση αρχικούς πειραματισμούς με το εργαλείο. Στο κεφάλαιο 6, όπου γίνεται εκτενέστερος έλεγχος, παρατηρούνται και επεξηγούνται διαφορές στη συμπεριφορά του karmada. Παρόλα αυτά, επειδή η υλοποίηση χρησιμοποιεί αυτή την παραδοχή, δεν αναλύεται περαιτέρω σε αυτό το σημείο.

Divided Weighted PP με StaticWeightList

Στη Weighted με StaticWeightList, ή διαιρεμένη με στατικά βάρη πολιτική, τα replicas μοιράζονται στους member clusters με βάση προκαθορισμένα σταθερά βάρη. Κατά τον ορισμό της αντιστοιχίζεται ένα βάρος ανά member cluster, με βάση τα οποία υπολογίζεται ο αριθμός των replicas που θα δρομολογηθούν σε κάθε cluster. Ο τρόπος που υπολογίζεται ο αριθμός των replicas που θα δρομολογηθούν δίνεται από τη σχέση 2.2

$$R_{(C_n)} = \frac{w_n}{\sum_{i=1}^{C} w_{C_i}} \cdot r \tag{2.2}$$

όπου w_i το βάρος του i-οστού member cluster, r ο αριθμός των επιθυμητών replicas και C ο συνολικός αριθμός των member clusters. Για παράδειγμα, για $r = 12$ και $w_1 = 1$, $w_2 = 2$ $w_2 = 3$ δρομολογούνται $R_{(C_1)} = 12 \cdot \frac{1}{6} = 2$, $R_{(C_2)} = 12 \cdot \frac{2}{6} = 4$ και $R_{(C_2)} = 12 \cdot \frac{3}{6} = 6$. Είναι προφανές ότι η σχέση 2.2 δεν έχει πάντα ακέραια αποτελέσματα. Επειδή δεν υφίσταται μη-ακέραιος αριθμός replicas, το παραπάνω αποτέλεσμα πρέπει να στρογγυλοποιηθεί. Ο τρόπος που υλοποιείται αυτό δεν είναι ο προφανής, δηλαδή είτε μαθηματική στρογγυλοποίηση, είτε διατήρηση του ακέραιου μέρους. Ειδική αναφορά στον τρόπο στρογγυλοποίησης γίνεται στο πρακτικό μέρος αυτής της εργασίας (βλ. 4.3.2, 6.3.1).

Divided Weighted PP με DynamicWeights

Η Weighted με DynamicWeights, ή διαιρεμένη με δυναμικά βάρη πολιτική, παρουσιάζει σημαντικά πολλές ομοιότητες με τη Weighted με StaticWeightList. Η μόνη διαφορά που έχουν είναι ότι η λίστα των βαρών του κάθε member cluster δεν ορίζεται από τον χρήστη στο manifest αλλά παράγεται αυτόματα από το karmada χρησιμοποιώντας τη μοντελοποίηση πόρων των clusters (βλ. 2.3.2). Αυτό γίνεται θέτοντας ως βάρη τα `AvailableReplicas` και εφαρμόζοντας τη σχέση 2.2 σαν να επρόκειτο για στατικά βάρη. Έτσι για παράδειγμα, $r = 12$ και $AR_1 = 6$, $AR_12 = 2$ $AR_2 = 18$, όπου AR τα `AvailableReplicas`, το πρόβλημα πρόκειται για πολιτική Weighted με StaticWeightList με $w_1 = 6$, $w_12 = 2$ $w_2 = 18$, η οποία μάλιστα είναι ισοδύναμη με αυτή του προηγούμενου παραδείγματος, αφού $AR_1 = 6 \cdot w_1$, $AR_2 = 6 \cdot w_2$ και $AR_3 = 6 \cdot w_3$. Αξίζει να σημειωθεί ότι στην περίπτωση όπου δεν ορίζεται `WeightPreference` το karmada μοιράζει τα αντίγραφα ίσα στους clusters.

Στον κώδικα 2.2 φαίνεται μία ολοκληρωμένη Propagation Policy.

ΚΩΔΙΚΑΣ 2.2: *Παράδειγμα Propagation Policy*

```
apiVersion: policy.karmada.io/v1alpha1
kind: PropagationPolicy
metadata:
  name: nginx-propagation
spec:
  resourceSelectors:
    - apiVersion: apps/v1
      kind: Deployment
      name: nginx
  placement:
    clusterAffinity:
      clusterNames:
        - member1
        - member2
    replicaScheduling:
      replicaDivisionPreference: Weighted
      replicaSchedulingType: Divided
      weightPreference:
        staticWeightList:
          - targetCluster:
              clusterNames:
                - member1
            weight: 1
          - targetCluster:
              clusterNames:
                - member2
            weight: 1
```

2.4.2 Πολιτικές Παράκαμψης (Override Policies)

Πολλές φορές, οι εφαρμογές που τρέχουν σε ένα multi cluster σύστημα δεν είναι ίδιες για κάθε cluster. Για παράδειγμα, μπορεί να χρειάζεται να τρέχει διαφορετική έκδοση της containerised εφαρμογής σε cluster διαφορετικής περιοχής, ή να χρησιμοποιούνται διαφορετικές παράμετροι. Επίσης, μπορεί να χρειάζεται να αλλάζουν διαφόρων ειδών metadata, για την καλύτερη οργάνωση του συστήματος. Λύση στα παραπάνω δίνουν οι Πολιτικές Παράκαμψης (Override Policies). Αποτελούν objects που επιλέγουν ένα σύνολο από objects και εφαρμόζουν ένα σύνολο από κανόνες (overriders) με βάση τους clusters στους οποίους βρίσκονται. Για την επιλογή των objects, το `.spec` μίας Override Policy περιέχει το πεδίο `.spec.resourceSelectors`, όπως ακριβώς σε μία PropagationPolicy (βλ. 2.4.1). Το πεδίο του `.spec` που είναι υπεύθυνο για τον ορισμό των κανόνων είναι το `.spec.overrideRules`. Περιέχει μία λίστα από στοιχεία, τα οποία χαρακτηρίζονται από τα πεδία `targetCluster` και `overriders`. Το πεδίο `targetCluster` ορίζεται κατά αντιστοιχία με το clusterAffinity ενός Propagation Policy. Η τιμή του πεδίου overriders είναι μία λίστα με έναν ή περισσότερους από τους παρακάτω τύπους overrider:

- `ImageOverrider` : αλλάζει την εικόνα (image) κάποιου container. Συγκεκριμένα μπορεί να τροποποιήσει το repository, το registry ή το tag της εικόνας. Υποστηρίζει τους τελεστές `add`,`replace` και `remove`.
- `CommandOverrider` : αλλάζει την εντολή (command) κάποιου container. Μπορεί να προσθέσει ή να αφαιρέσει παραμέτρους γραμμής εντολών στην εφαρμογή. Υποστηρίζει τους τελεστές `add` και `remove`.
- `ArgsOverrider` : αντίστοιχο του `CommandOverrider` για τα argumetns της εφαρμογής. Υποστηρίζει τους τελεστές `add` και `remove`.
- `LabelsOverrider` : αλλάζει τα labels στα metadata του object. Υποστηρίζει τους τελεστές `add`,`replace` και `remove`.
- `AnnotationsOverrider` : αλλάζει τα annotations στα metadata του object. Αντίστοιχο με το `LabelsOverrider`.
- `PlaintextOverrider` : ένας απλός overrider που αλλάζει οποιοδήποτε πεδίο με βάση το path του στο API. Λειτουργεί κατά αντιστοιχία με την εντολή `kubectl patch`.

Στον κώδικα 2.3 φαίνεται μία ολοκληρωμένη Override Policy.

2.4.3 Άλλα objects

Το Karmada, πέρα από τη δρομολόγηση εφαρμογών υποστηρίζει μία ευρεία γκάμα λειτουργιών, όπως ανακατεύθυνση σε περίπτωση βλάβης (Multi-cluster Failover), Multi-cluster Δικτύωση, διαχείριση συμμόρφωσης ασφαλείας (Security Compliance Governance), Ομόσπονδη Αυτόματη Οριζόντια Κλιμάκωση (Federated Horizontal Pod Autoscaling) και πολλά άλλα. Για την υλοποίηση πολλών από αυτές τις λειτουργίες ορίζει και άλλα objects. Όμως οι λειτουργίες αυτές ξεφεύγουν από το εύρος της παρούσας διπλωματικής και για τον λόγο αυτό δεν αναλύονται περαιτέρω.

ΚΩΔΙΚΑΣ 2.3: *Παράδειγμα Override Policy*

```
apiVersion: policy.karmada.io/v1alpha1
kind: OverridePolicy
metadata:
  name: nginx-op
spec:
  resourceSelectors:
    - apiVersion: apps/v1
      kind: Deployment
      name: nginx
  overrideRules:
    - targetCluster:
        clusterNames:
          - member2
      overriders:
        labelsOverrider:
          - operator: add
            value:
              env: skoala-dev
          - operator: add
            value:
              env-stat: skoala-stage
          - operator: remove
            value:
              for: for
          - operator: replace
            value:
              bar: test
    - targetCluster:
        clusterNames:
          - member1
      overriders:
        annotationsOverrider:
          - operator: add
            value:
              env: skoala-stage
          - operator: remove
            value:
              bom: bom
          - operator: replace
            value:
              emma: sophia
```

Κεφάλαιο 3

Δίκτυα Petri

Στο κεφάλαιο αυτό γίνεται μία εισαγωγή στην έννοια των δικτύων Petri. Αρχικά, αναλύονται η δομή και οι βασικές έννοιες των δικτύων Petri. Έπειτα αναλύονται παραδείγματα χρήσης τους για τη μοντελοποίηση απλών συστημάτων. Στη συνέχεια παρουσιάζονται διάφορες υποκατηγορίες των δικτύων Petri καθώς και προεκτάσεις που επεκτείνουν τη χρήση τους. Τέλος, αναφέρονται τυπικές μέθοδοι ανάλυσης τους.

3.1 Εισαγωγή

Τα Δίκτυα Petri (Petri Nets ή PNs) αποτελούν υπολογιστικές δομές ικανές να μοντελοποιήσουν Συστήματα Διακριτών Συμβάντων (Discreat Event Systems ή DES) [18]. Αναπτύχθηκαν από τον C.A. Petri στις αρχές τις δεκαετίας του 1960. Χρησιμοποιούνται κυρίως από την επιστήμη του αυτόματου ελέγχου, όμως είναι επαρκώς γενικά ώστε να μπορούν να μοντελοποιήσουν ποικιλία συστημάτων και διαδικασιών [19]. Παρουσιάζουν παρόμοια δομή με τα πεπερασμένα αυτόματα (finite-state automata ή FA), αφού και τα δύο αποτελούν συσκευές που χειρίζονται συμβάντα με βάση ορισμένους κανόνες και αποτελούν αναπαράσταση της συνάρτησης μετάβασης ενός DES. Σε αντίθεση με τα αυτόματα, οι μεταβάσεις γίνονται υπό ρητά ορισμένες συνθήκες, με αποτέλεσμα να μπορούν να μοντελοποιήσουν δυνητικά πολυπλοκότερα συστήματα από τα αυτόματα. Αξίζει να σημειωθεί ότι ένα πεπερασμένο αυτόματο μπορεί πάντα να αναπαρασταθεί σαν ένα δίκτυο Petri, ενώ το αντίθετο δεν ισχύει. Συνεπώς, η κλάση των συστημάτων που περιγράφουν τα δίκτυα Petri είναι γνήσιο υπερσύνολο της κλάσης των πεπερασμένων αυτομάτων. Τα δίκτυα Petri μπορούν επίσης να αναπαρασταθούν γραφικά με τη μορφή γράφου δικτύου Petri (petri net graph), διατηρώντας στη γραφική τους αναπαράσταση πολλές δομικές πληροφορίες του συστήματος με έναν εποπτικό και διαισθητικό τρόπο [18].

3.2 Δομή και Ιδιότητες

Ένα Petri Net αποτελεί ένα γράφο με δύο είδη κόμβων, τις Θέσεις (Places) και τις Μεταβάσεις (Transitions). Οι κόμβοι του γράφου συνδέονται με ακμές που ονομάζονται Arcs. Αποτελεί διμερή (bipartite) γράφο: τα places και transitions εναλλάσσονται σε μονοπάτια συνδεδεμένα από arcs. Αυτό σημαίνει ότι ένα arc δεν μπορεί να συνδέει δύο κόμβους ίδιου τύπου, αλλά πάντοτε είτε ξεκινάει από place και καταλήγει σε transition ή το αντίστροφο [18]. Για την αναπαράσταση των ενεργών και των μη ενεργών καταστάσεων χρησιμοποιούνται

τα tokens. Τα χαρακτηριστικά και οι ιδιότητες των διαφόρων μερών του δικτύου αναλύονται παρακάτω.

3.2.1 Places

Τα Places αναπαριστούν τις καταστάσεις του συστήματος, ή τις συνθήκες που μπορούν επικρατούν στο σύστημα, ανάλογα με το σύστημα και την προσέγγιση ως προς τη μοντελοποίηση. Στη γραφική αναπαράσταση του petri net graph συμβολίζονται με ένα κύκλο. Ο αριθμός των places σε ένα μη εκφυλισμένο petri net είναι πεπερασμένος και μη μηδενικός.

3.2.2 Transitions

Τα transistions, όπως δηλώνει και η ονομασία τους, αναπαριστούν τις μεταβάσεις, ή τα συμβάντα που μεταβάλλουν την κατάσταση του συστήματος. Στη γραφική αναπαράσταση του petri net graph συμβολίζονται με ένα παραλληλόγραμμο ή μία μπάρα. Ο αριθμός των transitions σε ένα μη εκφυλισμένο Petri Net είναι πεπερασμένος και μη μηδενικός. Κάθε transition παρεμβάλλεται ανάμεσα σε δύο places, μεταφέροντας Tokens από το ένα στο άλλο, την είσοδο (input) και την έξοδο (output) αντίστοιχα. Εξαίρεση αποτελούν οι μεταβάσεις πηγής (source transitions), που αποτελούν μεταβάσεις χωρίς place εισόδου, και οι μεταβάσεις καταβόθρας (sink transitions), που αποτελούν μεταβάσεις χωρίς έξοδο.

3.2.3 Arcs

Τα τόξα (arcs) αποτελούν συνδέσεις ανάμεσα στα places και τα transitions. Όπως αναφέρθηκε και παραπάνω, ένα arc μπορεί να συνδέει είτε place με transition είτε το αντίστροφο. Στη γενική περίπτωση μπορούν πολλά Arcs να καταλήγουν σε ένα κοινό place ή transition. Στη γραφική αναπαράσταση του petri net graph συμβολίζονται με βέλη, όπως οι ακμές σε έναν κατευθυνόμενο γράφο. Ο αριθμός των arcs σε ένα μη εκφυλισμένο Petri Net είναι πεπερασμένος και μη μηδενικός.

3.2.4 Tokens και Marking

Όπως αναφέρθηκε προηγουμένως, τα places συμβολίζουν τις καταστάσεις/συνθήκες του συστήματος, τις οποίες μεταβάλλουν τα transitions. Για να οριστεί ποιες συνθήκες ικανοποιούνται (ή ποιες καταστάσεις είναι πραγματικές και όχι δυνητικές) χρησιμοποιούνται τα Σημεία (Tokens). Σε ένα απλό Petri Net, κάθε place μπορεί να περιέχει ένα ακέραιο αριθμό από tokens, τα οποία δηλώνουν την ικανοποίηση της συνθήκης. Στη γραφική αναπαράσταση του petri net graph αναπαρίστανται συνήθως ως κουκκίδες. Η αντιστοιχιση των places με τα tokens τους ονομάζεται Σήμανση (Marking), και ένα Petri Net που περιέχει tokens ονομάζεται Marked Petri Net.

3.2.5 Firing Transitions

Για να εφαρμοστεί μία μετάβαση σε ένα DES πρέπει να ικανοποιούνται οι συνθήκες που απαιτούνται για τη μετάβαση αυτή. Ο τρόπος που αναπαρίσταται η διαδικασία αυτή στο Petri Net είναι μέσω της πυροδότησης (firing) των transitions. Προϋπόθεση για να πυροδοτηθεί

ένα transition σε ένα απλό Petri Net είναι κάθε source place του, δηλαδή κάθε place από το οποίο ξεκινά arc που καταλήγει στο συγκεκριμένο transition, να έχει τουλάχιστον ένα token. Στην περίπτωση αυτή λέμε ότι το transition είναι ενεργοποιημένο (enabled) ή πυροδοτήσιμο (firable). Εάν ισχύει το παραπάνω, το transition μπορεί να πυροδοτηθεί, δηλαδή να καταναλώσει ένα token από κάθε place εισόδου και να δημιουργήσει ένα token σε κάθε place εξόδου.

Παρατήρηση: Ένα source place είναι πάντα enabled.

3.2.6 Τυπική Αναπαράσταση δικτύου Petri

Ένα Petri Net ορίζεται ως ένα διμερής γράφος

$$(P, T, A)$$

όπου:
P το σύνολο των places,
T το σύνολο των transitions,
$A \subseteq (P \times T) \cup (T \times P)$ το σύνολο των arcs

Στο δίκτυο αντιστοιχεί μία συνάρτηση $M_{(p_i)} : P- > N^*$ που δηλώνει το Marking του δικτύου. Η συνάρτηση αυτή δεν αποτελεί χαρακτηριστικό του δικτύου, καθώς αλλάζει μορφή μόλις ένα transition πυροδοτηθεί. Το Marking μπορεί να αναπαρασταθεί και ως ένα μονοδιάστατο διάνυσμα M με μέγεθος όσο το πλήθος των states, και τιμές τον αριθμό των tokens σε κάθε place. Για παράδειγμα, για Petri Net με $P = p_1, p_2, p_3$ και $M_{(p_1)} = 1, M_{(p_2)} = 2, M_{(p_1)} = 0$ τότε $M = [1, 2, 0]$

3.2.7 Παραδείγματα απλών δικτύων Petri

Παρακάτω φαίνονται δύο απλά παραδείγματα Petri Net.

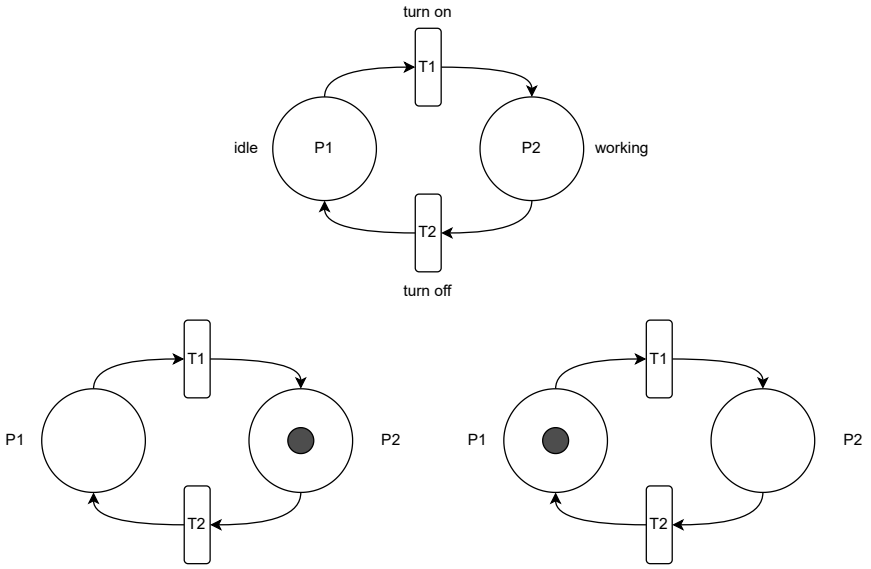
Παράδειγμα 1: Απλοϊκή συσκευή

Έστω μια απλοϊκή συσκευή, η οποία μπορεί είτε να είναι σε λειτουργία, είτε να είναι σε κατάσταση αδράνειας. Η απεικόνιση του petri net graph που την αναπαριστά καθώς και οι πιθανές του καταστάσεις φαίνονται στην σχήμα 3.1,

Τυπικά το Petri Net ορίζεται ως εξής:

$$(P, T, A)$$

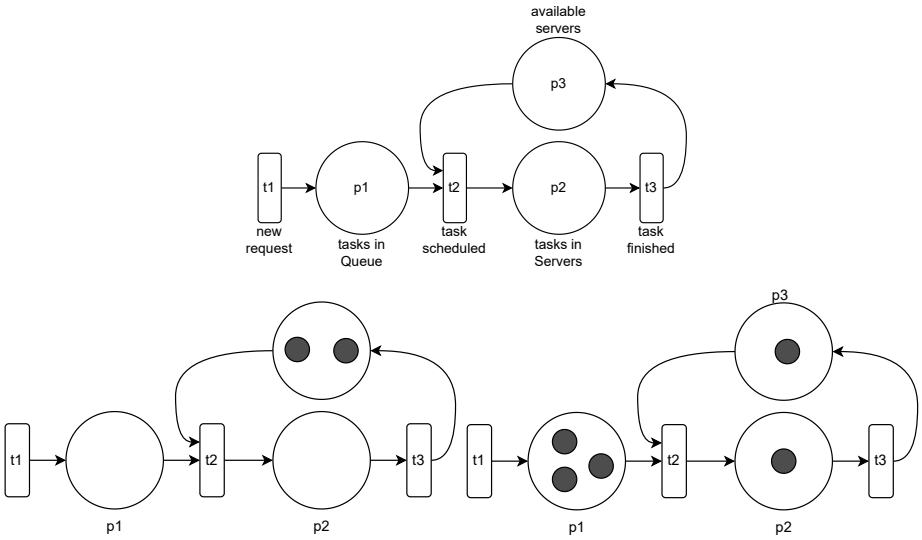
όπου:
$P = P1, P2$
$T = T1, T2$
$A = (P1, T1), (T1, P2), (P2, T2), (T2, P1)$



Σχήμα 3.1: *Παράδειγμα petri net απλοϊκής συσκευής*

Παράδειγμα 2 : Σύστημα αναμονής

Έστω άπειρης χωρητικότητας ουρά αναμονής, η οποία εξυπηρετείται από N servers. Η απεικόνιση του petri net graph που αναπαριστά το σύστημα αυτό καθώς και η αρχικοποίηση του δικτύου και μία τυχαία στιγμή λειτουργίας του για $N = 2$ φαίνεται στο σχήμα 3.2.



Σχήμα 3.2: *Παράδειγμα Petri Net συστήματος αναμονής*

Τυπικά το Petri Net ορίζεται ως εξής:

$$(P, T, A)$$

όπου:

$P = p1, p2, p3$

$T = t1, t2, t3$

$A = (t1, p1), (p1, t2), (t2, p2), (p2, t3), (t3, p3), (p3, t2)$

3.3 Ειδικές Περιπτώσεις, Παραλλαγές και επεκτάσεις

3.3.1 Autonomous και Non Autonomous PNs

Στην ενότητα 3.2.5 αναλύονται οι συνθήκες οι οποίες κάνουν ένα transition firable. Όμως, δε γίνεται σαφές πότε ένα firable transition πυροδοτείται. Ο λόγος είναι γιατί, ανάλογα με το σύστημα και τον λόγο που το μελετάμε, τα Petri Nets διακρίνονται ανάλογα με το πότε συμβαίνουν τα firings των transitions σε αυτόνομα (autonomous) και μη-αυτόνομα (non-autonomous).

Σε ένα αυτόνομο (autonomous Petri Net), η σειρά και ο τρόπος που πυροδοτούνται τα transitions είτε είναι άγνωστος, είτε δεν καθορίζεται ρητά, είτε δεν έχει σημασία για τη μελέτη του προβλήματος. Αντίθετα, σε ένα μη-αυτόνομο (non-autonomous) Petri Net, τα transitions πυροδοτούνται είτε όταν συμβαίνουν ορισμένα εξωτερικά, ανεξάρτητα από το σύστημα συμβάντα είτε σε καθορισμένες χρονικές στιγμές. Κατά σύμβαση όταν ένα Petri Net είναι autonomous δε χρειάζεται αυτό να προσδιοριστεί, μπορεί δηλαδή να καλείται απλώς Petri Net.

3.3.2 Bounded PNs

Ένα place p ονομάζεται φραγμένο (bounded) για ένα αρχικό Marking M_0 εάν υπάρχει αριθμός $k \in \mathbb{N}$, τέτοιος ώστε, για κάθε πιθανό Marking, $M_{(p)} \leq k$. Συνεπώς, ένα Petri Net ονομάζεται φραγμένο (bounded) για ένα αρχικό Marking M_0 εάν κάθε place είναι bounded για το M_0. Ένα Petri Net ονομάζεται δομικά φραγμένο (stucturally bounded) εάν είναι bounded για κάθε αρχικό marking.

Παράδειγμα: Στα παραδείγματα της υπο-ενότητας 3.2.7, το Παράδειγμα 1 είναι bounded, και μάλιστα δομικά bounded (χωρίς να διατηρεί όμως τη πρακτική σημασία του για $k \geq 1$). Αντίθετα, το Παράδειγμα 2 δεν είναι bounded. Αυτό επιβεβαιώνεται και στην υπο-ενότητα 3.4.3.

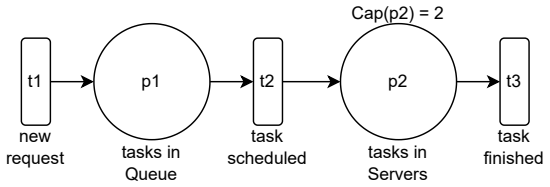
3.3.3 Generalised PNs

Σε ένα Γενικευμένο (Generalised) Petri Net, βάρη σχετίζονται με τα Arcs του δικτύου. Τα βάρη αυτά είναι φυσικοί αριθμοί, διάφοροι του 0. Κάθε arc μεταφέρει τόσα tokens όσα το βάρος που του αναλογεί. *Παρατήρηση: Κάθε Generalised Petri Net μπορεί να μετατραπεί σε ένα απλό Petri Net και το αντίστροφο. Για τη μετατροπή του Generalised Petri Net σε απλό, αρκεί η αντικατάσταση κάθε arc βάρους μεγαλύτερο του 1 με τόσα απλά arcs, όσα το βάρος. Για το αντίστροφο, αρκεί η ανάθεση βάρους 1 σε κάθε arc, και στην περίπτωση όπου δύο ή περισσότερα arcs ξεκινούν και τελειώνουν στον ίδιο κόμβο, η ένωση τους σε arc με βάρος όσο το πλήθος τους.*

3.3.4 Finite Capacity PNs

Σε ένα Πεπερασμένης Χωρητικότητας (Finite Capacity) Petri Net, χωρητικότητες σχετίζονται με τα places του δικτύου. Οι χωρητικότητες των places είναι αυστηρά θετικοί αριθμοί, που δηλώνουν τον μέγιστο αριθμό από tokens που μπορεί να λάβει το συγκεκριμένο place. Σε ένα Finite Capacity Petri Net, για να είναι ένα transition firable, εκτός από το να υπάρχει επαρκής αριθμός από tokens σε κάθε input place, πρέπει ο αριθμός των tokens που θα υπάρχουν σε κάθε output place να μην υπερβαίνει τη χωρητικότητα του. Να σημειωθεί ότι δεν είναι απαραίτητο κάθε place του δικτύου να έχει πεπερασμένη χωρητικότητα. *Παρατήρηση: Κάθε Finite Capacity Petri net μπορεί να μετατραπεί σε ένα απλό Petri Net. Για τη μετατροπή του Finite Capacity Petri net σε απλό χρειάζεται η αντικατάσταση κάθε finite capacity place P με ένα απλό, η προσθήκη ενός βοηθητικού place p_1, με τόσα tokens όσο η χωρητικότητα του αρχικού finite capacity place, καθώς και arcs από το p_1 προς κάθε transition το οποίο έχει έξοδο το p και από κάθε transition που έχει είσοδο το p προς το p_1.*

Παράδειγμα: Στο παράδειγμα 2 της ενότητας 3.2.7, το σύστημα αναμονής μπορεί να αναπαρασταθεί με το finite capacity petri net του σχήματος 3.3. Το Petri Net που παρουσιάζεται στο σχήμα 3.2 είναι το ισοδύναμο απλό Petri Net που προκύπτει ακολουθώντας την παραπάνω διαδικασία.



Σχήμα 3.3: *Παράδειγμα Finite Capacity Petri Net*

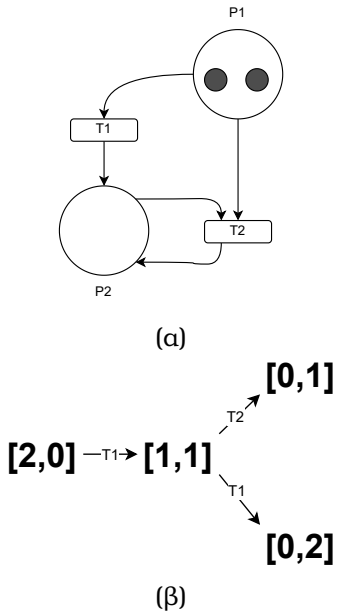
3.4 Ανάλυση δικτύων Petri

Τα Petri Nets αποτελούν χρήσιμα εργαλεία για τη μελέτη συστημάτων. Όμως, όταν αυτά αναπαριστούν πολύπλοκα συστήματα, η διαισθητική ανάλυση των δικτύων δεν αρκεί. Για τον λόγο αυτό είναι αναγκαία η χρήση τυπικών μεθόδων και δομών για την ανάλυση των δικτύων. Στα πλαίσια της εργασίας αυτής θα αναλύσουμε δομές που βρίσκουν υπολογιστικά όλα τα πιθανά Markings ενός συστήματος. Μπορούν με τη σειρά τους να χρησιμοποιηθούν για την εξαγωγή χρήσιμων συμπερασμάτων όπως την ύπαρξη επαναλαμβανόμενων firings, την ύπαρξη σταθερών καταστάσεων, τον εντοπισμό ανεπιθύμητων καταστάσεων κλπ. Τέτοιες δομές είναι ο Graph of Markings και τα Coverability Tree και Coverability Graph.

3.4.1 Graph of Markings

Ο Γράφος Σημάνσεων (Graph of Markings), γνωστός και ως γράφος καταστάσεων (State Graph) αποτελεί βασική δομή για τη μελέτη πιθανών καταστάσεων σε ένα PN. Αποτελείται από κόμβους, οι οποίοι αντιστοιχούν στα διανύσματα των Markings που μπορεί να έχει το δίκτυο, και ακμές, οι οποίες αντιστοιχούν στα transitions που πρέπει να πυροδοτηθούν για να μεταβεί το δίκτυο από το ένα Marking στο άλλο. Ο Graph of Markings υπολογίζεται

με εξαντλητική αναζήτηση (exhaustive search) όπως φαίνεται και στον αλγόριθμο 3.1. Στο σχήμα 3.4 φαίνεται ένα απλό Petri Net και ο αντίστοιχος Graph of Markings.



Σχήμα 3.4: *Παράδειγμα Petri Net (α) και αντίστοιχου Graph of Markings (β)*

ΚΩΔΙΚΑΣ 3.1: *Αλγόριθμος εύρεσης Graph of Markings με εξαντλητική Αναζήτηση*

```
result = empty_graph (V,E)
function GOM(P,T,A,M):
      for t in T:
            if t is firable:
               M_new = fire_transition(t,P,T,A,M)
               if M_new not in result.V:
                        result.add_vertex(M_new)
                result.add_edge(M , M_new , t)
               GOM(P,T,A,M_new)

result.add_vertex( initial_Marking)
GOM(initial_Marking)
```

3.4.2 Coverability Graph και Coverability Tree

Ο Graph of Markings είναι αρκετά βοηθητικός για bounded petri nets. Όμως για unbounded petri nets, όπου ο αριθμός των πιθανών Markings είναι άπειρος, είναι προφανές ότι ο Graph of Markings δεν μπορεί να βοηθήσει στη μελέτη τους, καθώς δεν μπορεί καν να κατασκευαστεί, αφού ο αλγόριθμος παραγωγής του δε θα ολοκληρωθεί ποτέ. Για παράδειγμα, στο πολύ απλό Petri Net του σχήματος 3.4 αποτελεί άπειρη μαρκοβιανή αλυσίδα. Το πρόβλημα αυτό λύνεται με χρήση δύο παρόμοιων δομών, του Δέντρου και του Γράφου Καλυψιμότητας (Coverability Tree και Graph). Αποτελούν προεκτάσεις του Graph

of Markings, με πεπερασμένο όμως αριθμό κόμβων.

Ορισμός: Ένα Marking M_1 ¨καλύπτει¨ (covers) ένα Marking M_2 αν ισχύει :

$$M_1(p) \geq M_2(p) \forall p \in P$$

Στην περίπτωση αυτή συμβολίζουμε $M_1 \geq M_2$

Εάν ισχύει

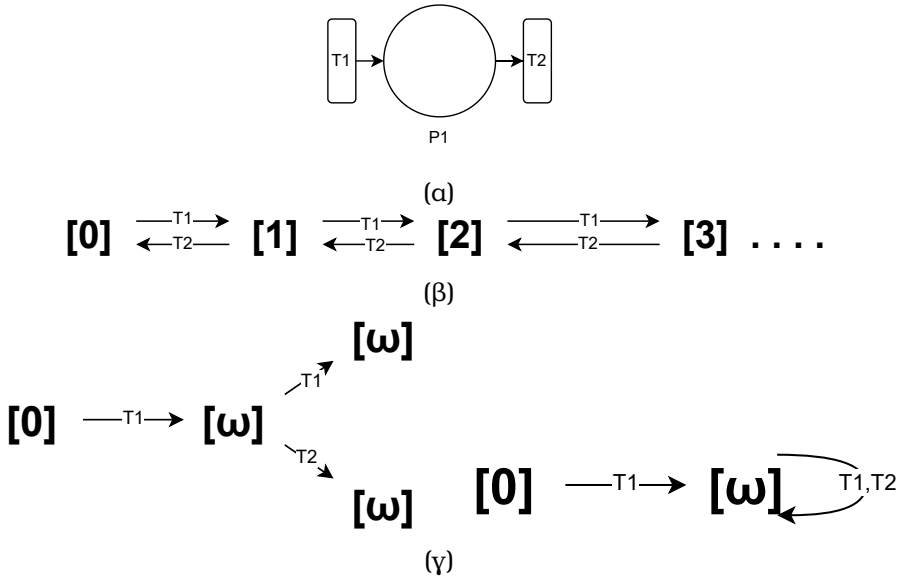
$$M_1(p) \geq M_2(p) \forall p \in P$$

και υπάρχει έστω ένα $p1 \in P$ τέτοιο ώστε

$$M_1(p1) > M_2(p1)$$

λέμε ότι το M_1 ¨καλύπτει αυστηρά¨ (strictly covers) το M_2 και το συμβολίζουμε με $M_1 > M_2$

Μπορούμε να παρατηρήσουμε ότι σε ένα unbounded petri net, για να γίνει ο αριθμός των tokens σε ένα place ∞, πρέπει μετά από ακολουθία πυροδοτήσεων (firing sequence) που του αυξάνει τον αριθμό των tokens, ο αριθμός των tokens στα υπόλοιπα places να παραμένει ίδιος (ή να αυξάνεται), διαφορετικά μετά από αρκετές πυροδοτήσεις θα τελείωναν. Για τον λόγο αυτό μπορούμε να θεωρήσουμε ότι τα Markings που απειρίζουν ένα unbounded petri net καλύπτουν το ένα το άλλο. Έτσι, τα καλυπτόμενα Markings μπορούν να αντικατασταθούν με ένα κοινό Marking, κάνοντας τα ισοδύναμα. Το νέο αυτό Marking αποτελείται από ένα διάνυσμα τα στοιχεία του οποίου είναι τα κοινά σημεία των δύο επικαλυπτόμενων Markings, και ένα σύμβολο (χρησιμοποιούμε τον χαρακτήρα ω) στα στοιχεία που αυξάνονται, το οποίο αναπαριστά τις άπειρες τιμές που παίρνουν τα στοιχεία αυτά.



Σχήμα 3.5: *Παράδειγμα Petri Net (α) και των αντίστοιχων Graph of Markings (β) και Coverability Tree και Graph (γ)*

Για την κατασκευή του Coverability Tree ακολουθείται ο αλγόριθμος 3.2. Για την κατασκευή του Coverability Graph, αρκεί η ένωση όμοιων κόμβων στο Coverability Tree.

Σημείωση: Για ένα bounded petri net ο Coverability Graph είναι ισοδύναμος με τον Graph

ΚΩΔΙΚΑΣ 3.2: *Αλγόριθμος εύρεσης Coverability Tree*

```
Tree = empty_tree()
Tree.add_root(M0)

Markings = []
for t in T:
        if t is firable:
                M_new = t.fire(M_0)
                if M_new > M0:
                        for p in P :
                                if M_new(p) > M0(p) :
                                        M_new(p) =
                Markings.add(M_new)
def CT(P,T,A,M):
        if Mj = M  in Tree.path(M0 —> M): stop
        else:
                for t in T:
                        if t is firable:
                                M_new = t.fire(M)
                                Tree.add(M—>M_new)

                                if Mk < M_new in Tree.path(M0 —> M_new):
                                        for p in P :
                                                if M_new(p) > Mk(p) :
                                                        M_new(p) =
                                CTP(P,T,A,M_new)
```

of Markings.

3.4.3 Ανάλυση παραδειγμάτων ενότητας 3.2.7

Παρακάτω αναλύονται ως προς τα πιθανά Markings τα παραδείγματα της ενότητας 3.2.7.

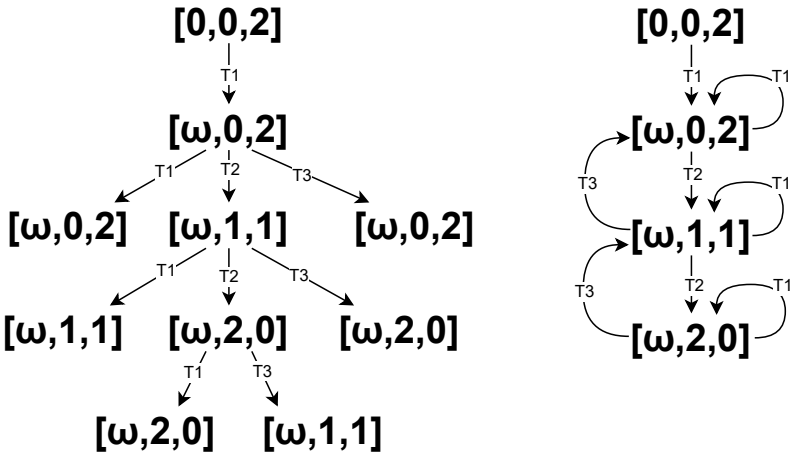
Παράδειγμα 1 : Απλοϊκή Συσκευή

Το Petri Net που περιγράφει τη συσκευή αυτή είναι bounded. Συνεπώς, κατασκευάζουμε τον Graph of Markings (βλ Σχήμα 3.6).

Σχήμα 3.6: *Graph of Markings απλοϊκής συσκευής*

Παράδειγμα 2 : Σύστημα Αναμονής

Το Petri Net που περιγράφει τη συσκευή αυτή δεν είναι bounded. Κατασκευάζουμε λοιπόν Coverability Tree και στη συνέχεια Coverability Graph (βλ Σχήμα 3.7).



Σχήμα 3.7: *Graph of Markings απλοϊκής συσκευής*

3.5 Colored Petri Nets

Τα Έγχρωμα (Colored) Petri Net (CPN) αποτελούν επέκταση των απλών Petri Net. Χρησιμοποιούνται για να απλουστεύσουν πολύπλοκα Petri Nets σε πιο κατανοητές αναπαραστάσεις, καθώς και για τη μοντελοποίηση συστημάτων που είναι δύσκολο να αναπαρασταθούν με απλά Petri Nets. Η κύρια διαφορά τους είναι ότι σε Coloured Petri Net, τα tokens περιέχουν πληροφορία, η οποία ονομάζεται χρώμα (color). Το χρώμα μπορεί να χρησιμοποιηθεί για να περιγράψει πολυπλοκότερες έννοιες από την ύπαρξη ή όχι μίας συνθήκης, όπως συμβαίνει με τα tokens ενός απλού petri net. [20]

3.5.1 Χρώματα και Color Sets

Όπως αναφέρθηκε και προηγουμένως, σε ένα Coloured Petri net , σε κάθε token ανατίθεται ένα χρώμα. Τα χρώματα αυτά είναι τιμές δεδομένων, αυθαίρετης πολυπλοκότητας. Για παράδειγμα, μπορούν να είναι ακέραιοι αριθμοί, συμβολοσειρές, αλλά και πολυπλοκότερες δομές όπως λίστες, πλειάδες (tuples) και άλλες αφηρημένες δομές. Κάθε place μπορεί να περιέχει συγκεκριμένου τύπου tokens. Ο τύπος των tokens που μπορεί να περιέχει ένα place καλείται σύνολο χρωμάτων (color set). Η χρήση των color sets για ένα CPN είναι ανάλογη με τη χρήση τύπων σε μία γλώσσα προγραμματισμού, περιορίζοντας τις πιθανές τιμές που μπορεί να έχουν τα tokens όπως γίνεται στις γλώσσες προγραμματισμού με τις μεταβλητές. Τα color sets συμβολίζονται στη γραφική αναπαράσταση του petri net graph με πλάγια γραφή (italics) δίπλα στο εκάστοτε place.

3.5.2 Marking σε CPN :MultiSets

Σε ένα απλό Petri Net, το Marking αποτελεί ένα διάνυσμα, το οποίο έχει σαν τιμές τον αριθμό των Tokens του κάθε place. Όπως είναι αναμενόμενο η αναπαράσταση αυτή δεν αρκεί για την απεικόνιση του Marking σε ένα Colored Petri Net, καθώς πλέον τα tokens περιέχουν πληροφορίες. Σε ένα CPN, το Marking δεν είναι ένας ακέραιος αριθμός αλλά ένα πολυσύνολο (multi-set ή multiset). Τα multisets είναι επέκταση των απλών συνόλων, με την ιδιότητα ότι μπορούν να περιέχουν αντίγραφα του ίδιου στοιχείου. Για παράδειγμα,

ενώ το σύνολο $\{2, 3, 7, 1, 3\}$ δεν αποτελεί καλώς ορισμένο σύνολο, λόγω του διπλότυπου 3, αποτελεί καλώς ορισμένο multiset. Για χάρη συντομίας μπορούμε να αντιπροσωπεύσουμε τα στοιχεία ενός Multiset με ένα ακέραιο αριθμό που δηλώνει τον αριθμό των εμφανίσεων του στοιχείου στο σύνολο, ακολουθούμενο από το χαρακτήρα $'$‘$'$. Για παράδειγμα, το multiset $\{a, \beta, a, \gamma, a, \gamma\}$ μπορεί να αναπαρασταθεί ως $\{3‘a, 1‘\beta, 2‘\gamma, 1‘\delta\}$ ή απλούστερα $\{3‘a, \beta, 2‘\gamma, \delta\}$. παραλείποντας τα στοιχεία που εμφανίζονται μια φορά. Έτσι το Marking ενός place είναι ένα πολυσύνολο με στοιχεία που ανήκουν στο color set του place.

3.5.3 Arcs σε CPN : Expressions και Variables

Όπως και για το Marking ενός Petri Net, τα arcs των CPN παρουσιάζουν διαφορές σε σχέση με τα απλά Petri Net. Για να μπορούν να μοντελοποιούν ικανοποιητικά τα πολύπλοκα συστήματα που καλούνται να μοντελοποιήσουν τα CPN, τα arcs συσχετίζονται με μία έκφραση (expression), όπως συσχετίζονται με ένα ακέραιο αριθμό στα Generalised petri nets. Τα expressions αποτελούν συναρτήσεις που αποτιμώνται σε Multisets. Για να μπορούν να εφαρμοστούν σε διάφορα χρώματα tokens, οι εκφράσεις μπορούν να λαμβάνουν έναν αριθμό από μεταβλητές (variables). Οι εκφράσεις συμβολίζονται στη γραφική αναπαράσταση του petri net graph δίπλα ή πάνω στο εκάστοτε arc. Σε περίπτωση που σε ένα arc η έκφραση είναι η συνάρτηση ταυτότητα, δηλαδή η συνάρτηση που δίνει ως έξοδο τη μεταβλητή εισόδου, αρκεί να αναφέρουμε τη μεταβλητή ως έκφραση.

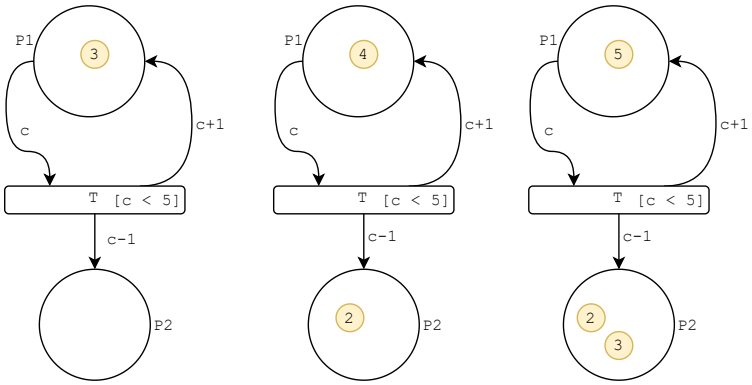
3.5.4 Transitions σε CPN : bindings και guards

Η αντιστοίχιση ενός token σε μία μεταβλητή ονομάζεται δέσμευση (binding), και όπως είναι αναμενόμενο μπορούν ανάλογα με το Marking του δικτύου να αντιστοιχούν διάφορα bindings. Κατά την πυροδότηση ενός Transition, ελέγχονται όλα τα πιθανά bindings και επιλέγεται ένα από αυτά που ικανοποιούν τις απαραίτητες συνθήκες, δηλαδή να μπορούν να αποτιμηθούν από τις εκφράσεις των arcs.

Παράλληλα με τις εκφράσεις των arcs, για λόγους απλοποίησης του δικτύου, μπορούν να οριστούν επιπρόσθετες συνθήκες που πρέπει να πληρούν τα bindings ώστε να θεωρηθεί firable το transition. Οι συνθήκες αυτές είναι λογικές συναρτήσεις πάνω στο σύνολο των color sets που αποτιμώνται σε ¨αληθής¨ ή ¨ψευδής¨. Το σύνολο των λογικών αυτών εκφράσεων αν συζευχθεί μέσω του λογικού τελεστή ¨και¨ ($\wedge$) δημιουργούν μία λογική έκφραση που ονομάζεται φύλακας (guard) του Transition. Στη γραφική αναπαράσταση του petri net graph αναγράφονται μέσα σε αγκύλες ([]) δίπλα ή μέσα στο εκάστοτε transition

3.5.5 Παράδειγμα CPN

Ένα απλό παράδειγμα Colored Petri Net φαίνεται στο σχήμα 3.8 Στο CPN αυτό, το αρχικό Marking είναι το διάνυσμα πολυσυνόλων $[\{3\}, \{\}]$ ενώ μετά από δύο επαναλαμβανόμενες πυροδοτήσεις της T έχουμε αντίστοιχα τα Markings $[\{4\}, \{2\}]$ και $[\{5\}, \{2, 3\}]$. Παρατηρούμε επίσης ότι μετά από τις δύο αυτές πυροδοτήσεις, η συνθήκη $c < 5$ του guard της T δεν ικανοποιείται από κανένα binding (το μόνο πιθανό binding είναι $c = 5$). Έτσι δεν είναι πλέον firable.



$color\ C = int$

$color\_set\ P1 : C$

$color\_set\ P2 : C$

$var\ c : C$

Σχήμα 3.8: *Παράδειγμα Colored Petri Net*

Μέρος II

Πρακτικό Μέρος

Κεφάλαιο 4

Μοντελοποίηση

Στο κεφάλαιο αυτό γίνεται μελέτη της μοντελοποίησης ενός Multi Cluster συστήματος με χρήση δικτύων Petri. Συγκεκριμένα αναλύονται τα διάφορα Petri Net που σχεδιάστηκαν στα πλαίσια της εργασίας για τη μοντελοποίηση διαφόρων δομικών στοιχείων του συστήματος, όπως οι member clusters και οι διάφορες πολιτικές διάδοσης. Τέλος, περιγράφονται οι διάφοροι τρόποι επέκτασης και συνδυασμού των δικτύων που σχεδιάστηκαν.

4.1 Γενικά

Όπως αναφέρθηκε και στην εισαγωγή, σκοπός της παρούσας εργασίας είναι η μοντελοποίηση Multi Cluster συστημάτων που χρησιμοποιούν το Karmada, με χρήση Petri Net. Για την κατανόηση των παρακάτω ενοτήτων που αναλύουν τα δίκτυα που σχεδιάστηκαν στα πλαίσια της εργασίας, πρέπει να γίνουν πρώτα ορισμένες επισημάνσεις. Λόγω της πολυπλοκότητας των συστημάτων αυτών, χρησιμοποιούμε Colored Petri Nets, τα οποία μας επιτρέπουν να ενσωματώνουμε στη μοντελοποίηση αφηρημένες σχέσεις με έναν πιο συνεκτικό και κατανοητό τρόπο. Στα CPN αυτά, τα objects του Karmada τα οποία χρίζουν δρομολόγησης (τα οποία στο εξής καλούμε services, καθώς αποτελούν τις υπηρεσίες που φιλοξενούνται από το σύστημα, σε αντίθεση με τις διάφορες πολιτικές και τα υπόλοιπα αντικείμενα), τα nodes των member clusters αλλά και τα resource modellings των member clusters μοντελοποιούνται ως colored tokens.

Συγκεκριμένα, μπορούμε να απεικονίσουμε ένα service ως μία πλειάδα (tuple) με τύπους (*str, float, float, float, float, int, int*), για την οποία το πρώτο στοιχείο αντιστοιχεί στο όνομα του service, το δεύτερο και το τρίτο στην ελάχιστη και τη μέγιστη απαίτηση σε επεξεργαστή (cpu) αντίστοιχα, το τέταρτο και το πέμπτο στην ελάχιστη και τη μέγιστη απαίτηση σε μνήμη (ram) αντίστοιχα, ενώ τα τελευταία δύο στοιχεία στον ελάχιστο και μέγιστο αριθμό από pods.Αν ένα από τα μέγιστα είναι ορισμένο σε 0 σημαίνει ότι μπορεί να έχει απεριόριστα μεγάλη τιμή, δηλαδή δεν ορίζεται άνω όριο. Να σημειωθεί ότι στα πλαίσια της εργασίας, καθώς η τοποθέτηση από το Karmada δε λαμβάνει υπόψη τα μέγιστα αλλά μόνο τα ελάχιστα όρια, τα μέγιστα όρια θα μπορούσαν να αφαιρεθούν από την απεικόνιση. Όμως, για λόγους τόσο πληρότητας όσο και επεκτασιμότητας, επιλέγουμε να τα διατηρούμε στην απεικόνιση των services.

Κατά αντίστοιχο τρόπο μπορούμε να απεικονίσουμε τα nodes ως πλειάδες με τύπους (*str, float, float, float, float, int, int*), για τους οποίους το πρώτο στοιχείο αντιστοιχεί στο όνομα του node, το δεύτερο και το τρίτο στην allocated και την allocatable χρήση επεξεργαστή

(cpu) αντίστοιχα, το τέταρτο και το πέμπτο στην allocated και την allocatable μνήμη (ram) αντίστοιχα, ενώ τα τελευταία δύο στοιχεία στα running και total available pods. Τέλος, τα resource modells των member clusters απεικονίζονται ως πλειάδες με τύπους (*float, float, float, float, int, int*), οι οποίοι συμπεριφέρονται αντίστοιχα με τα nodes, χωρίς χρήση του ονόματος. Η μη συμπερίληψη του ονόματος γίνεται σαφής στη συνέχεια του κεφαλαίου.

Έτσι μπορούμε να ορίσουμε τρία color sets: services(S), nodes (N) και resource models (C).

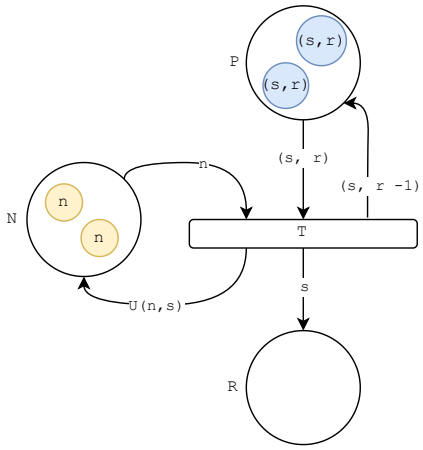
4.2 Μοντελοποίηση Member Cluster

Ένα Multi Cluster σύστημα αποτελείται από πολλούς Kubernetes Clusters. Για το λόγο αυτό, για την υλοποίηση ενός Multi Cluster συστήματος χρειάζεται αρχικά η μοντελοποίηση των επιμέρους member clusters. Αναλυτικές μοντελοποιήσεις Kubernetes Clusters με χρήση Petri Net υπάρχουν στη βιβλιογραφία όπως στα [21], [22]. Παρατηρούμε όμως ότι οι συγκεκριμένες υλοποιήσεις είναι αρκετά πολύπλοκες και εστιάζουν κυρίως στον κύκλο ζωής της εφαρμογής μέσα στον Cluster. Για παράδειγμα, μπορούν να αναπαράγουν με ακρίβεια τη διαδικασία δημιουργίας των pods, καθώς και να εξετάζουν τι συμβαίνει σε περίπτωση σφάλματος. Ενώ αυτό αποτελεί όντως μέρος της λειτουργίας ενός Multi Cluster συστήματος, όσον αφορά το πρόβλημα της τοποθέτησης εφαρμογών (placement), μπορούμε για λόγους απλότητας να χρησιμοποιήσουμε μία πιο αφαιρετική μοντελοποίηση του Cluster. Η απλούστευση αυτή εξετάζει μόνο τη δυνατότητα του Cluster να εκτελέσει επαρκή αριθμό replicas. Δε λαμβάνει δηλαδή υπόψη τυχόν σφάλματα, ούτε εξετάζει τι θα γίνει σε περίπτωση που τα Pods σταματήσουν. Αξίζει να σημειωθεί ότι η συγκεκριμένη υλοποίηση μπορεί να αντικατασταθεί από πολυπλοκότερες υλοποιήσεις, ή να επεκταθεί ώστε να περιλαμβάνει περισσότερες λειτουργίες. Στο σχήμα 4.1 φαίνεται η υλοποίηση του Petri Net που χρησιμοποιείται στα πλαίσια της εργασίας για τη μοντελοποίηση των clusters. Η συνάρτηση $U(s,n)$ που χρησιμοποιείται στην έκφραση του arc (T,N) χρησιμοποιείται για την ενημέρωση της χρήσης πόρων του cluster μετά τη δρομολόγηση του service, και ορίζεται ως:

$$\begin{aligned} U(s,n) = (&name(n), c_{allocated}(n) - c_{required}(s), c_{total}(n), \\ &m_{allocated}(n) - m_{required}(s), m_{allocatable}(n), \\ &p_{allocated}(n) - p_{required}(s), pods_{total}(n)) \end{aligned} \quad (4.1)$$

όπου c cpu, m μνήμη ram και p pods, Παρατηρούμε ότι η συνάρτηση αυτή επιστρέφει ένα node σε μορφή πλειάδας, όπως αυτή ορίστηκε παραπάνω. Θεωρώντας τα s και n ως πλειάδες μπορούμε να ορίσουμε την $U(s,n)$ και ως:

$$U(s,n) = (n[0], n[1] - s[1], n[2], n[3] - s[3], n[4], n[5] - s[5], n[6]) \quad (4.2)$$



$$Color\ S\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ N\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ R\ =\ int$$

$$var\ s\ :\ S$$

$$var\ r\ :\ R$$

$$var\ n\ :\ N$$

Σχήμα 4.1: *Petri Net για Kubernetes Cluster*

4.3 Μοντελοποίηση Πολιτικών Διάδοσης

Όλες οι πολιτικές διάδοσης ακολουθούν αντίστοιχη δομή. Σε κάθε μία από αυτές ορίζεται ένα place *Svc*, το οποίο αρχικοποιείται με τα service tokens των εφαρμογών στις οποίες εφαρμόζεται η πολιτική. Το *Svc* συνδέεται με ένα transition *Pr*, το οποίο συνδέεται με τα *P* places των member clusters. Έτσι, ανάλογα με το guard και τα expressions των arcs που συνδέουν το transition *Pr* με τους clusters δρομολογείται ο κατάλληλος αριθμός από replicas στους member clusters. Παράδειγμα μίας γενικής πολιτικής διάδοσης φαίνεται στο σχήμα 4.2. Όπως αναφέρεται και στο κεφάλαιο 2, ορισμένες πολιτικές διάδοσης χρειάζεται να λαμβάνουν υπόψη τους το resource modelling των member clusters. Όμως, επειδή το karmada δεν έχει πρόσβαση απευθείας στις μετρικές των nodes, αλλά χρησιμοποιεί ένα συνολικό modelling για τον cluster, ορίζουμε για τις πολιτικές αυτές ένα place ανά member cluster, το οποίο περιέχει το resource modelling του, όπως φαίνεται στο σχήμα 4.3.

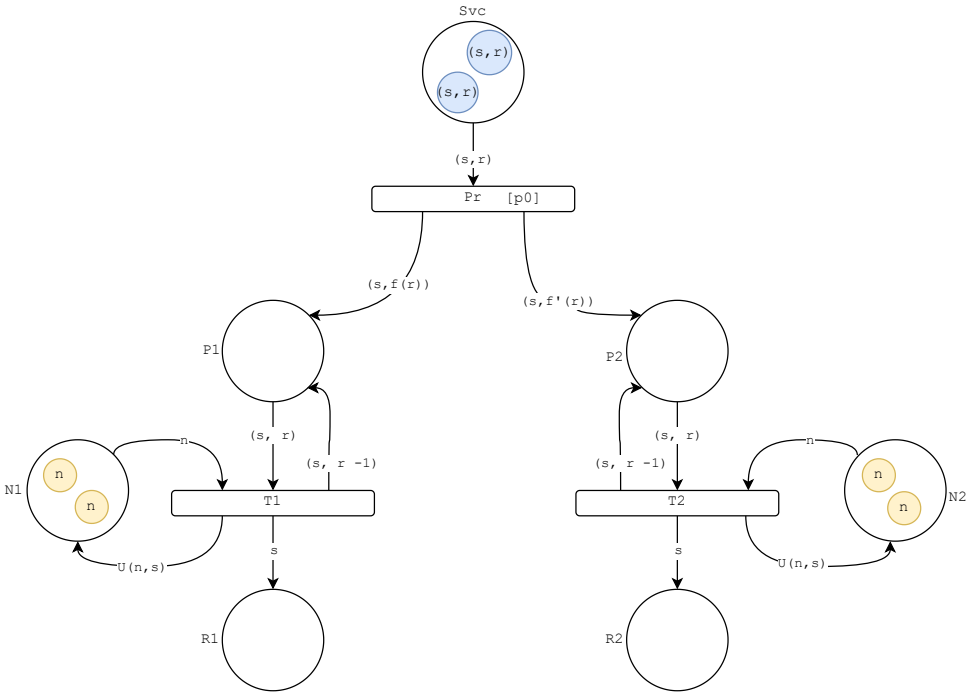
Στις υπόλοιπες υποενότητες αναλύονται οι διάφορες πολιτικές διάδοσης και οι αλλαγές ή προσθήκες που πρέπει να γίνουν στα δυο αυτά δίκτυα για να τις υλοποιήσουν. Συγκεκριμένα αρκεί να οριστεί ένα σύνολο συναρτήσεων $f_i(r)$ που επιστρέφουν τον αριθμό των replicas που δρομολογούνται στον i-οστό member cluster. Για τις πολιτικές που χρησιμοποιούν resource modelling ορίζουμε συνάρτηση $U_i'(c, s, r)$ η οποία χρησιμοποιεί την f_i για να υπολογίσει το

νέο resource modelling μετά τη δρομολόγηση και ορίζεται ως

$$U_i'(s,c,r) = (c_{allocated}(c) - c_{required}(s) \cdot f_i(r), c_{total}(c), \\ m_{allocated}(c) - m_{required}(s) \cdot f_i(r), m_{total}(c), \\ p_{running}(c) - p_{required}(s) \cdot f_i(r), p_{total}(c)) \quad (4.3)$$

ή χρησιμοποιώντας πλειάδες

$$U_i'(s,c,r) = (c[0] - s[1] \cdot f_i(r), c[1], n[2] - s[3] \cdot f_i(r), c[3], n[4] - s[5] \cdot f_i(r), c[5]) \quad (4.4)$$



$$Color\ S\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ N\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ R\ =\ int$$

$$Color\ C\ =\ (float, float, float, flaot, int, int)$$

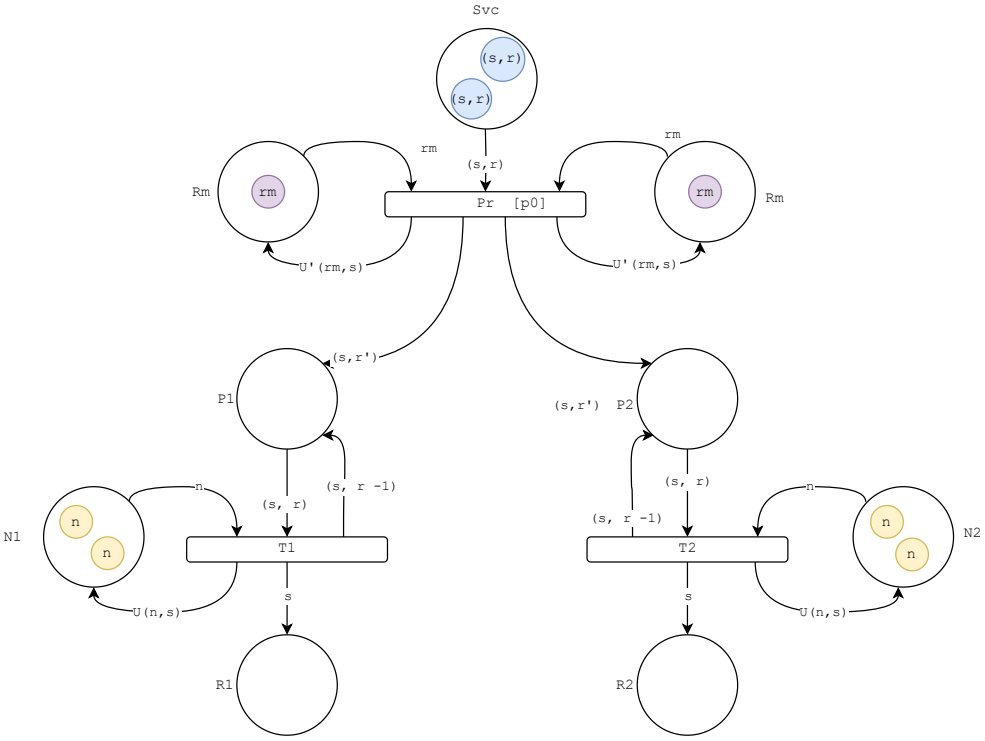
$$var\ s\ :\ S$$

$$var\ r\ :\ R$$

$$var\ n\ :\ N$$

$$var\ c\ :\ C$$

Σχήμα 4.2: *Petri Net για Propagation Policy*



$$Color\ S\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ N\ =\ (str, float, float, float, flaot, int, int)$$

$$Color\ C\ =\ (float, float, float, flaot, int, int)$$

$$Color\ R\ =\ int$$

$$var\ s\ :\ S$$

$$var\ r\ :\ R$$

$$var\ n\ :\ N$$

$$var\ c\ :\ C$$

Σχήμα 4.3: *Petri Net για Propagation Policy με resouce modelling.*

4.3.1 Μοντελοποίηση Duplicated Propagatioin Policy

Η Duplicated Propagation policy είναι η απλούστερη από τις πολιτικές διάδοσης. Στην πολιτική αυτή, πρέπει να δρομολογηθούν σε κάθε member cluster τόσοι πόροι όσοι ορίζονται στο manifest του object, δηλαδή στη μεταβλητή r των tokens του place *Svc*. Για τον λόγο αυτό χρησιμοποιούμε το σύνολο συναρτήσεων που δίνεται από τη σχέση:

$$f_i(r) = r \tag{4.5}$$

4.3.2 Μοντελοποίηση Divided - Weighted Propagatioin Policyμε Static Weights

Στη συγκεκριμένη πολιτική, σταθερά βάρη ορίζονται για τα services ανάλογα με τον cluster. Για την υλοποίηση της πολιτικής, κάθε service token χρειάζεται να περιέχει πέρα από το r και μία πλειάδα με βάρη $w = (w_1, w_2...., w_n)$, με το βάρος που αντιστοιχεί στον κάθε server. Έτσι το νέο color set των tokens είναι (s, w, r). Με βάση αυτή την αλλαγή, και σύμφωνα με την τεκμηρίωση του karmada ,το σύνολο συναρτήσεων δίνεται από τη σχέση:

$$f_i(r) = \frac{w_i}{\sum_{j=1}^{|clusters|} w_j} \cdot r \tag{4.6}$$

Όπως αναφέρεται και στο κεφάλαιο 2, η υλοποίηση του karmada παρουσιάζει μια ιδιαιτερότητα ως προς τη στρογγυλοποίηση των μη ακέραιων αριθμών replicas. Αντί να χρησιμοποιεί μαθηματική στρογγυλοποίηση (στον πλησιέστερο ακέραιο), διατήρηση ακέραιου μέρους ή στρογγυλοποίηση στον επόμενο ακέραιο, φαίνεται να έχει μία διαφορετική συμπεριφορά, η οποία δεν αναφέρεται στην τεκμηρίωση. Παράδειγμα της συμπεριφοράς φαίνεται στους πίνακες 4.1 και 4.2

replicas	round	floor/int	ceil	Karmada
1	1,0	0,0	1,1	1,0
2	1,1	1,0	2,1	2,0
3	2,1	2,1	2,1	2,1
4	3,1	2,1	3,2	3,1
5	3,2	3,1	4,2	4,1
6	4,2	4,2	4,2	4,2
7	5,2	4,2	5,3	5,2
8	5,3	5,2	6,3	6,2
9	6,3	6,3	6,3	6,3
10	7,3	6,3	7,4	7,3
11	7,4	7,3	8,4	8,3
12	8,4	8,4	8,4	8,4

Πίνακας 4.1: *Σύγκριση αλγορίθμων στρογγυλοποίησης και πραγματικής μέτρησης για διαφόρους αριθμούς replicas και βάρη (2,1)*

Παρατηρούμε ότι η συνάρτηση για τον cluster 1 το μεγαλύτερο βάρος ακολουθεί μέθοδο στρογγυλοποίησης προς τον επόμενο ακέραιο (ceiling), Όμως αυτό δεν ισχύει για κάθε cluster, κάτι που εκτός από το ότι δε συμφωνεί με τη μέτρηση, δε θα έδινε και επιθυμητό αριθμό από replicas. Το αντίστροφο συμβαίνει με τη στρογγυλοποίηση ακεραίου μέρους (floor), η οποία φαίνεται να συμφωνεί μόνο με τον τελευταίο cluster. Έτσι παρατηρούμε ότι κανείς από τους παραπάνω αλγορίθμους δεν αρκεί για να μοντελοποιήσει αυτή τη συμπεριφορά. Για τον λόγο αυτό κατασκευάσαμε τον αλγόριθμο που φαίνεται υλοποιημένος σε ψευδογλώσσα στον κώδικα 4.1, που πειραματικά έχει την ίδια συμπεριφορά με αυτόν που μετρήθηκε από το karmada σε όσα δοκιμαστικά σενάρια και αν εξετάστηκαν. Ο αλγόριθμος αυτός πρακτικά υπολογίζει το βάρος του κόμβου με το μεγαλύτερο βάρος χρησιμοποιώντας μέθοδο στρογγυλοποίησης επόμενου ακέραιου, αναθέτει την τιμή στον συγκεκριμένο cluster και στη συνέχεια υπολογίζει εξαρχής το πρόβλημα με τα υπόλοιπα replicas, και αφαιρώντας

replicas	round	floor/int	ceil	Karmada
1	0,0,1	0,0,0	1,1,1	0,0,1
2	0,1,1	0,0,1	1,1,1	0,1,1
3	0,1,2	0,1,1	1,1,2	0,1,2
4	1,1,2	0,1,2	1,2,2	0,2,2
5	1,2,2	0,1,2	1,2,3	0,2,3
6	1,2,3	1,2,3	1,2,3	1,2,3
7	1,2,4	1,2,3	2,3,4	1,2,4
8	1,3,4	1,2,4	2,3,4	1,3,4
9	2,3,4	1,3,4	2,3,5	1,3,5
10	2,3,5	1,3,5	2,4,5	1,4,5
11	2,4,6	1,3,5	2,4,6	1,4,6
12	2,4,6	2,4,6	2,4,6	2,4,6

Πίνακας 4.2: *Σύγκριση αλγορίθμων στρογγυλοποίησης και πραγματικής μέτρησης για διαφόρους αριθμούς replicas και βάρη (1,2,3)*

τον συγκεκριμένο cluster.

ΚΩΔΙΚΑΣ 4.1: *Αλγόριθμος Στρογγυλοποίησης*

```
function F(replicas,weights):
  sweights = sort_descending (weights)
  indexes = [index of w in weights for w in sweights]
  result_replicas = [0]*weights
  remainder = 0
  for w in sweights:
      W = sum( sweights[w ... end])
      x = ceil((replicas-remainder)*w/W)
      rest += x
      res[indexes[w]] = x
  return (res)
```

Χρησιμοποιώντας τη συνάρτηση F όπως ορίστηκε στον κώδικα 4.1, το σύνολο συναρτήσεων δίνεται από τη σχέση:

$$f_i(r) = F(r, \bigcup_{j=1}^{|clusters|} w_j)[i] \tag{4.7}$$

4.3.3 Μοντελοποίηση Divided - Weighted Propagatioin Policyμε Dynamic Weights

Η πολιτική αυτή είναι αντίστοιχη με την προηγούμενη, με τη μόνη διαφορά ότι αντί τα βάρη να δίνονται από το manifest, και κατά συνέπεια να πρέπει να ορίζονται στα tokens, υπολογίζονται με βάση το cluster resource modelling. Συγκεκριμένα το σύνολο συναρτήσεων, σύμφωνα με την τεκμηρίωση του karmada, δίνεται από τη σχέση:

$$f_i(r) = \frac{AR(i,s)}{\sum_{j=1}^{|clusters|} AR(j,s)} \cdot r \tag{4.8}$$

όπου το $AR(i,s)$ είναι τα Available Replicas του i-οστού member cluster για το object s όπως αυτά ορίζονται στην εξίσωση 2.1.

Προφανώς, η συγκεκριμένη πολιτική ακολουθεί το σχήμα 4.3. Το πρόβλημα της στρογγυλοποίησης που περιγράφεται στην υποενότητα 4.3.2 εμφανίζεται και εδώ, οπότε λαμβάνοντας υπόψη τη στρογγυλοποίηση ο τύπος που υπολογίζει το σύνολο συναρτήσεων δίνεται από τη σχέση:

$$f_i(r) = F(r, \bigcup_{j=1}^{|clusters|} AR(j,s))[i] \tag{4.9}$$

4.3.4 Μοντελοποίηση Divided, Aggregated Propagation Policy

Στη συγκεκριμένη πολιτική, τα αντίγραφα μοιράζονται σε όσο δυνατόν λιγότερους clusters, σεβόμενοι τους πόρους των member clusters. Για να υλοποιηθεί αυτό, πρέπει να κατασκευαστεί αλγόριθμος που αφού ταξινομήσει τους member clusters με βάση τα available replicas δρομολογεί στον server με τη μεγαλύτερη χωρητικότητα όσα replicas μπορεί να χωρέσει (φυσικά εφόσον επαρκούν) και επαναλαμβάνει τη διαδικασία με τους επόμενους clusters έως ότου δεν υπάρχουν άλλα replicas για δρομολόγηση. Έτσι κατασκευάζουμε τον αλγόριθμο που φαίνεται στον κώδικα 4.2

ΚΩΔΙΚΑΣ 4.2: *Αλγόριθμος Aggregation*

```
function A(replicas,clusters,current_cluster):
  sorted_AR = sort_descending ([AR(c,s) for c in clusters])
  sorted_clusters = [current_cluster of w in weights for w in sorted_AR]

  current_scheduled = sum([AR(c,s) for c in sorted_clusters[ 0 to current_cluster])
  result = min(replicas − current_scheduled, AR(current_cluster,s))

  return max(result,0)
```

Χρησιμοποιώντας αυτόν τον αλγόριθμο το σύνολο των συναρτήσεων ορίζεται ως:

$$f_i(r) = A(r, (\bigcup_{j=0}^{|clusters|} rm_j), i) \tag{4.10}$$

4.3.5 Μοντελοποίηση Override Policy

Οι πολιτικές παράκαμψης αλλάζουν τα χαρακτηριστικά των objects, αφού αυτά δρομολογηθούν, με βάση τον member cluster. Για την αναπαράστασή τους, αρκεί να τροποποιηθεί η έκφραση κάθε arc από το transition *Pr* στους member clusters, εφαρμόζοντας κατάλληλη

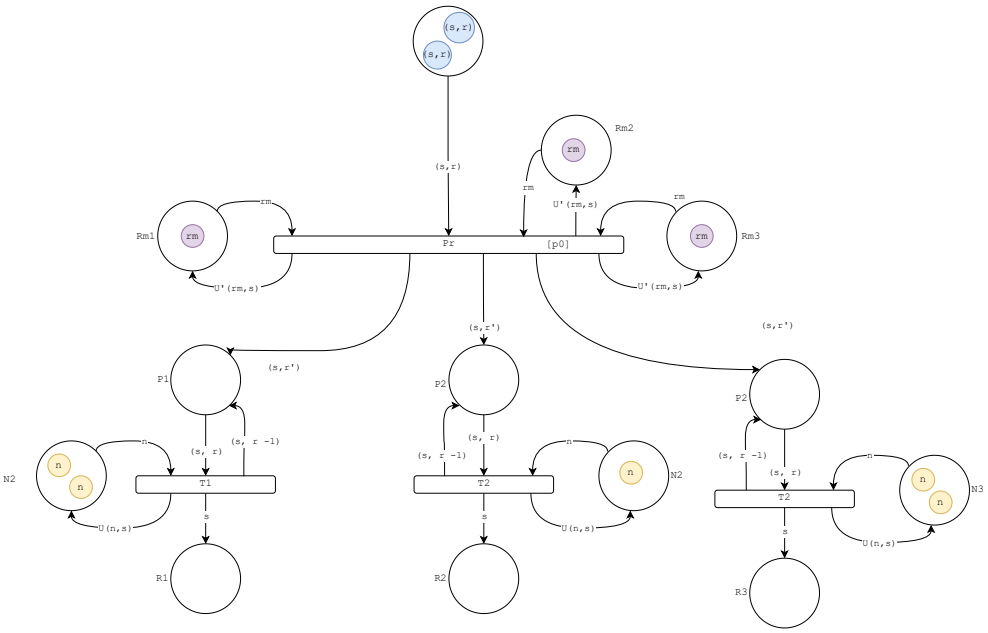
συνάρτηση O_i πάνω στο χρώμα s. Έτσι το συνολικό χρώμα του token θα είναι:

$$(O_i(s), f_i(s))$$

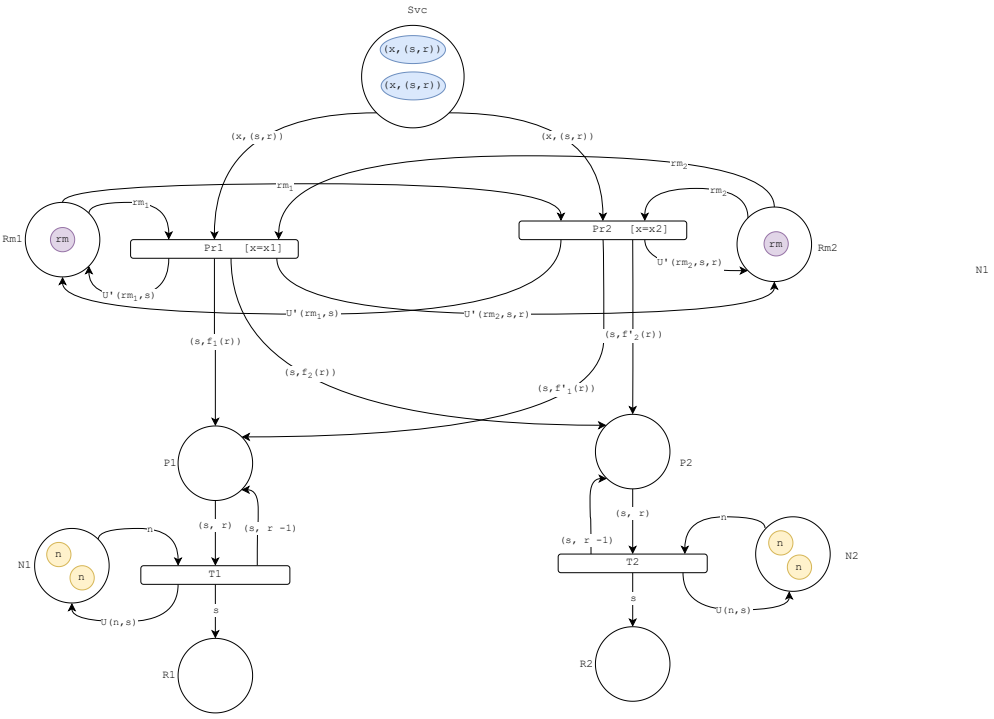
Στη συγκεκριμένη υλοποίηση των tokens το μόνο στοιχείο που μπορεί να αλλάζει η O_i είναι το όνομα. Όμως, για τροποποιημένα tokens της μορφής $(str, float, float, float, float, int, int, metadata)$, όπου τα *metadata* είναι αφηρημένου τύπου δεδομένα που αναπαριστούν τα στοιχεία του object manifest, μπορεί να εκτελέσει οποιαδήποτε Override Policy. Στα πλαίσια της εργασίας δεν υλοποιούμε πρακτικά Override Policies, καθότι δεν επηρεάζουν πρακτικά την τοποθέτηση των εφαρμογών.

4.4 Συνδυασμός και Επέκταση Δικτύων

Όπως προκύπτει από τα παραπάνω, τα δίκτυα που περιγράφηκαν επαρκούν για να καλύψουν βασικές περιπτώσεις συστημάτων. Όμως ένα πραγματικό σύστημα είναι αρκετά πιο πολύπλοκο. Για τον λόγο αυτό πρέπει να υπάρχει η δυνατότητα για τροποποίηση των δικτύων που περιγράφονται παραπάνω, είτε μέσω του συνδυασμού τους, είτε επεκτείνοντας τις δυνατότητες τους. Απλούστερο παράδειγμα αυτού, είναι η προσθήκη περισσότερων member clusters. Σε όλα τα παραδείγματα που εμφανίζονται μέχρι στιγμής, περιγράφονται συστήματα με 2 member clusters. Αυτό δεν αποτελεί περιορισμό της υλοποίησης. Στο σχήμα 4.4 παρουσιάζεται παράδειγμα συστήματος με 3 member clusters. Όπως φαίνεται στο σχήμα αρκεί η προσθήκη ενός ακόμα υποδικτύου μοντελοποίησης cluster και ενός resource modelling place στην περίπτωση που αυτό απαιτείται από την πολιτική. Πέρα όμως από τον συνδυασμό δικτύων clusters και κάποιου propagation policy, μπορούμε να συνδυάσουμε πολλές πολιτικές μεταξύ τους. Αν και θα μπορούσαμε να εκτελέσουμε δύο διαφορετικά μοντέλα, αυτό δε θα απεικόνιζε εξίσου αποτελεσματικά το πραγματικό σύστημα. Επίσης, μπορεί να είναι επιθυμητό να κατασκευαστεί ένα ενιαίο Petri Net για ένα σύστημα, το οποίο να περιλαμβάνει όλες τις ισχύουσες πολιτικές και να αλλάζει μόνο το αρχικό marking ανάλογα με τα services. Το πρόβλημα αυτό λύνεται με τον συνδυασμό των Petri Net πολλών πολιτικών μεταξύ τους, όπως φαίνεται στο σχήμα 4.5. Στην περίπτωση αυτή, ο συνδυασμός γίνεται με το να συνδεθούν αρχικά τα transitions των πολιτικών με κοινά places για όλες τις πολιτικές. Επίσης, χρειάζεται να δοθεί μία νέα τιμή στο χρώμα των tokens του *Svc*, η οποία διατηρεί την τιμή του policy που εφαρμόζεται στο συγκεκριμένο token, και η οποία ελέγχεται στο guard των transition των πολιτικών. Έτσι αποφεύγεται η πυροδότηση λάθος πολιτικής. Τέλος, ανάλογα με τις δυνατότητες και τους περιορισμούς του συστήματος μπορούν τα δίκτυα που περιγράφονται να επεκταθούν, προσθέτοντας κατάλληλα guards, αλλάζοντας τα expressions ή συνδυάζοντας τα με άλλα Petri Nets.



Σχήμα 4.4: *Petri Net για Propagation Policy με 3 member clusters*



$$Color\ X\ =\ str$$

$$var\ x\ :\ X$$

$$var\ x\ :\ X$$

Σχήμα 4.5: *Petri Net για συνδυασμό δύο Propagation Policies*

Κεφάλαιο 5

Υλοποίηση

Στο κεφάλαιο αυτό περιγράφεται η υλοποίηση, με βάση τα δίκτυα που σχεδιάστηκαν στο προηγούμενο κεφάλαιο. Αρχικά παρουσιάζονται τα προγραμματιστικά εργαλεία που χρησιμοποιήθηκαν. Στη συνέχεια δίνονται οι λεπτομέρειες υλοποίησης για τα διάφορα μέρη της βιβλιοθήκης που υλοποιήθηκε. Τέλος, δίνονται παραδείγματα χρήσης της βιβλιοθήκης για διάφορα δίκτυα.

5.1 Εισαγωγή

Όπως αναφέρθηκε και στην εισαγωγή, πέρα από τον σχεδιασμό κατάλληλων Petri Net για τη μοντελοποίηση Multi Cluster συστημάτων, σκοπός της εργασίας είναι και η ανάπτυξη ενός προγραμματιστικού περιβάλλοντος πρακτικής υλοποίησης των δικτύων αυτών. Για τον λόγο αυτό σχεδιάστηκε και υλοποιήθηκε η βιβλιοθήκη KarmadaPN. Η βιβλιοθήκη περιέχει έτοιμες υλοποιήσεις των παραπάνω μοντέλων, και δίνει τη δυνατότητα για συνδυασμό και επέκταση τους, όπως περιγράφεται στην ενότητα 4.4. Επίσης, περιέχει βοηθητικές συναρτήσεις για την απεικόνιση, την ανάλυση και τη δυναμική κατασκευή τους. Αξίζει να σημειωθεί ότι στα πλαίσια της εργασίας η βιβλιοθήκη λειτουργεί περισσότερο ως ένα εργαλείο απόδειξης της ιδέας (proof of concept) και όχι σαν ένα έτοιμο εργαλείο για χρήση από πραγματικά συστήματα.

5.2 Εργαλεία και βιβλιοθήκες

5.2.1 Python

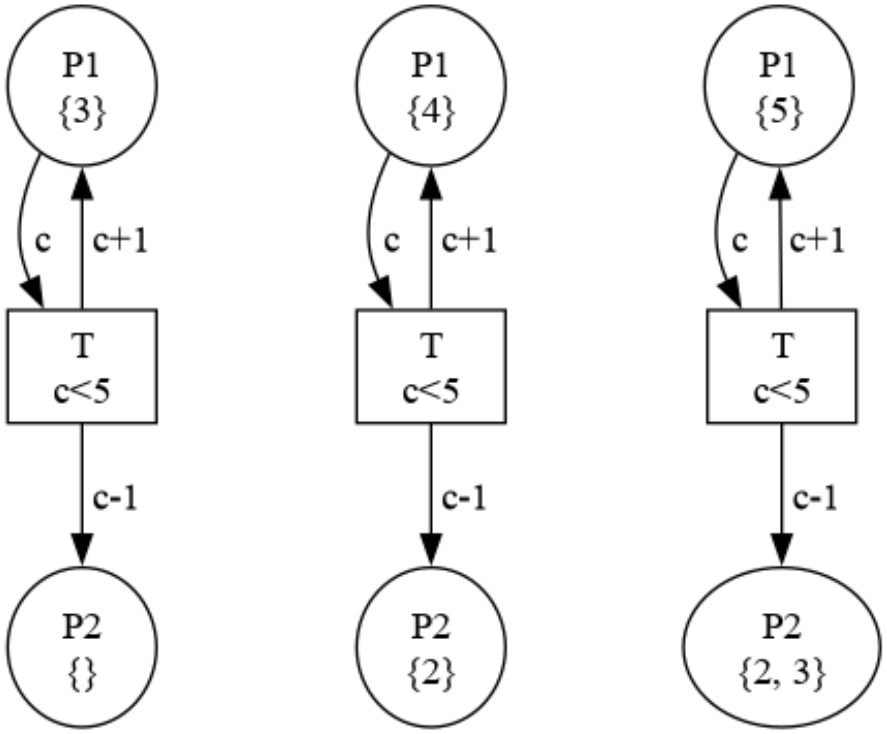
Για την υλοποίηση της βιβλιοθήκης χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python. Η γλώσσα Python είναι μία διερμηνευμένη (interpreted), αντικειμενοστραφής γλώσσα προγραμματισμού, που ενσωματώνει modules, εξαιρέσεις, δυναμική τυποθέτηση και υψηλού επιπέδου δομές δεδομένων και κλάσεις. Υποστηρίζει πολλαπλά παραδείγματα προγραμματισμού πέρα από τον αντικειμενοστρεφή, όπως διαδικαστικό και συναρτησιακό προγραμματισμό. Συνδυάζει υπολογιστική ισχύ με πολύ απλό συντακτικό. Είναι ανοιχτού κώδικα και μπορεί να εκτελεσθεί σε διάφορα λειτουργικά συστήματα [23]. Επίσης, η γλώσσα προγραμματισμού python περιλαμβάνει μια πληθώρα από βοηθητικές βιβλιοθήκες [24], και χρησιμοποιείται από μια μεγάλη κοινότητα προγραμματιστών και οργανισμών [25][23].

5.2.2 SNAKES

Για την υλοποίηση των Petri Nets αρχικά δοκιμάστηκε η ανάπτυξη κώδικα που υλοποιεί Petri Nets. Όμως αποδείχθηκε δύσκολο έργο που ξέφευγε από τους σκοπούς της παρούσας εργασίας. Για τον λόγο αυτό χρησιμοποιήθηκε η βιβλιοθήκη SNAKES [26]. Η βιβλιοθήκη SNAKES (**S**nakes is the **N**et **A**lgebra **K**it for **E**ditors and **S**imulators) είναι μία γενικού σκοπού βιβλιοθήκη για Petri Nets. Έχει σχεδιαστεί για τη γλώσσα Python αλλά είναι επεκτάσιμη και σε άλλες γλώσσες προγραμματισμού. Μπορεί να υλοποιήσει μία μεγάλη ποικιλία colored petri nets, χρησιμοποιώντας την python ως γλώσσα για τον ορισμό χρωμάτων, variabes, expressions και guards. Δίνει επίσης τη δυνατότητα για ανάλυση των καταστάσεων των δικτύων μέσω Graph of Markings. Περιέχει επίσης μία σειρά από πρόσθετα (plugins) που επεκτείνουν τις δυνατότητες της βιβλιοθήκης [27]. Στον κώδικα 5.1 φαίνεται παράδειγμα υλοποίησης του Colored Petri Net του σχήματος 3.8 χρησιμοποιώντας τη βιβλιοθήκη SNAKES. Το παραγόμενο Petri Net φαίνεται στην εικόνα 5.1, όπως αυτό οπτικοποιείται με χρήση ενός plugin του SNAKES, που ονομάζεται graphviz

ΚΩΔΙΚΑΣ 5.1: *Παράδειγμα υλοποίησης Petri Net με χρήση της βιβλιοθληκης SNAKES*

```
# simple import
#import snakes.nets

# import using plugins
import snakes.plugins
snakes.plugins.load("gv","snakes.nets","nets")
from nets import *

pn = PetriNet("Example_PetriNet")
pn.add_place(Place("P1"))
pn.add_place(Place("P2"))
pn.add_transition(Transition("T",Expression("c<5")))
pn.add_input("P1","T",Variable("c"))
pn.add_output("P1","T",Expression("c+1"))
pn.add_output("P2","T",Expression("c-1"))
pn.set_marking(Marking(P1=MultiSet([3])))
pn.draw("/tmp/out1.png")

t = pn.transition("T")
t.fire(t.modes().pop())

pn.draw("/tmp/out2.png")

t.fire(t.modes().pop())

pn.draw("/tmp/out3.png")
```



Σχήμα 5.1: *Petri Net για Propagation Policy με resouce modelling.*

5.2.3 kubectl

Μέρος της υλοποίησης αποτελεί η δυναμική ενημέρωση του Marking ενός Petri Net που περιγράφει ένα σύστημα από μετρικές πραγματικού χρόνου (real time metrics). Για την υλοποίηση αυτού όπως είναι αναμενόμενο χρειάζεται επικοινωνία με το Kubernetes API. Στα πλαίσια της παρούσας εργασίας χρησιμοποιήσαμε το εργαλείο γραμμής εντολών kubectl, κυρίως λόγω εξοικείωσης και καθότι η συγκεκριμένη δυνατότητα δεν απαιτεί λειτουργική απαίτηση της υλοποίησης, ούτως ώστε να αναζητηθεί πιο σταθερή εναλλακτική. Διαφορετικά θα μπορούσε να χρησιμοποιηθεί η βιβλιοθήκη kubernetes της python [9], ή ακόμα και εργαλεία παρακολούθησης (monitoring) ανεξάρτητα του kubernetes API, όπως το λογισμικό Prometheus [28].

5.2.4 networkx

Αν και η βιβλιοθήκη SNAKES δίνει τη δυνατότητα για ανάλυση, κυρίως μέσω παραγωγής του Graph of Markings, η συγκεκριμένη βιβλιοθήκη δεν προορίζεται για ανάλυση των δικτύων [27]. Ο συγκεκριμένος γράφος για παράδειγμα δεν είναι εύκολο να οπτικοποιηθεί, ειδικά αν πρόκειται για σχετικά πολύπλοκα δίκτυα, ούτε μπορεί να αναλυθεί επαρκώς. Για τον λόγω αυτό υλοποιούμε κατάλληλες συναρτήσεις για να τον εξάγουν σε μορφή συμβατή με τη βιβλιοθήκη networkx. Η βιβλιοθήκη networkx είναι μία βιβλιοθήκη της γλώσσας python που υποστηρίζει τη δημιουργία, τροποποίηση, και ανάλυση σύνθετων δικτύων και γράφων. Υποστηρίζει διάφορους τύπους γράφων και περιέχει έτοιμες υλοποιήσεις για αλγορίθμους

ανάλυσης τους [29].

5.3 Υλοποίηση Petri Nets που σχεδιάστηκαν

Στην ενότητα αυτή αναλύεται η υλοποίηση που πραγματοποιήθηκε στα πλαίσια της παρούσας διπλωματικής. Αποτελείται από ένα σύνολο κλάσεων και συναρτήσεων, οι οποίες υλοποιούν τα διάφορα μέρη και λειτουργίες ενός μοντέλου και επιτρέπουν με κατάλληλο συνδυασμό την περιγραφή διαφορετικών συστημάτων. Πιο συγκεκριμένα μπορούμε να τις διαχωρίσουμε σε συναρτήσεις και κλάσεις που ορίζουν την υλοποίηση όσων περιγράφηκαν στα προηγούμενα κεφάλαια, και σε βοηθητικές κλάσεις και μεθόδους, οι οποίες χρησιμοποιούνται αφενός για την ευκολία του χρήστη της βιβλιοθήκης προσθέτοντας ένα επίπεδο αφαίρεσης, και αφετέρου για την καλύτερα δομημένη οργάνωση του κώδικα.

Σημείωση: Στις επόμενες υποενότητες δεν περιγράφεται η πλήρης υλοποίηση κάθε συνάρτησης και κλάσης. Ο πηγαίος κώδικας όλων των υλοποιήσεων είναι διαθέσιμος σε δημόσιο αποθετήριο στο github το οποίο βρίσκεται στη διεύθυνση `https://github.com/SeekerRook/KarmadaPN`.

5.3.1 Βοηθητικές Κλάσεις και μέθοδοι

Η Κλάση PNComponent

Η κλάση `PNComponent` αποτελεί τη βασικότερη κλάση της υλοποίησης. Αποτελεί γενικευμένη κλάση, η οποία περιγράφει ένα υποδίκτυο ενός Petri Net. Χρησιμοποιείται για τη δημιουργία, επεξεργασία και συνδυασμό περιγραφών Petri Nets, οι οποίες μπορούν να μετατραπούν σε Petri Net της βιβλιοθήκης SNAKES. Οι μέθοδοι της κλάσης `PNComponent` φαίνονται παρακάτω:

- `__init__` : Μέθοδος κατασκευής της κλάσης. Αρχικοποιεί ένα κενό δίκτυο.
- `add_place` : Προσθέτει ένα νέο place στο δίκτυο. Αντίστοιχη της ομώνυμης συνάρτησης του `SNAKES.PetriNet`
- `add_transition` : Όμοια με `add_place` για transitions
- `add_input` : Όμοια με `add_place` για input arcs
- `add_output` : Όμοια με `add_place` για output arcs
- `add_component` : Προσθέτει ένα νέο υποδίκτυο στο δίκτυο.
- `merge` : Ενώνει δύο places του δικτύου σε ένα. Χρησιμοποιείται για την ένωση δύο υποδικτύων μεταξύ τους. Αξίζει να σημειωθεί ότι η ένωση υποδικτύων γίνεται μόνο μέσω κοινών places. Σε περίπτωση που θέλουμε να ενωθεί ένα υποδίκτυο με transition ενός άλλου, πρέπει να συνδέεται με την ένωση βοηθητικό place το οποίο θα ενωθεί με το place του αρχικού.
- `build` : Επιστρέφει το `SNAKES.PetriNet` που αντιστοιχεί στο `PNComponent`.

Οι κλάσεις Service, Node και ResoureModelling

Όπως αναφέρθηκε στο κεφάλαιο 4, τα tokens ενός δικτύου ενός Multi Cluster συστήματος αποτελούν πλειάδες που αναπαριστούν τα Services, τα Nodes και τα Resource Modellings του συστήματος. Θα αρκούσε λοιπόν για τον ορισμό τους να δίνεται από τον χρήστη της βιβλιοθήκης μία κατάλληλη πλειάδα. Έτσι όμως γίνεται πολυπλοκότερη η ανάπτυξη κώδικα και υπάρχει μεγαλύτερος κίνδυνος για λάθη, αφού πρέπει να αναγράφεται πάντα ολόκληρη η πλειάδα, ανεξαρτήτως του αν τα στοιχεία παίρνουν προεπιλεγμένες τιμές η όχι, και φυσικά με τη σωστή σειρά. Για τον λόγο αυτό ορίζονται οι κλάσεις `Service`, `Node` και `ResourceModelling`, οι οποίες αποτελούν μία αφαίρεση της εκάστοτε πλειάδας. Για παράδειγμα, έστω ένα Service για το οποίο ο μοναδικός περιορισμός ως προς τους πόρους είναι η ελάχιστη χρήση επεξεργαστή σε 0.2. Αντί για

```
token = ["Service_1", 0.2, 0, 0, 0, 1, 110]
```

μπορεί να οριστεί ως:

```
token = Service("Service_1", minCPU=0.2)
```

Το σύνολο συναρτήσεων Functions

Οι συναρτήσεις αυτές αποτελούν βοηθητικές συναρτήσεις που είναι απαραίτητες για την υλοποίηση των δικτύων. Χρησιμοποιούνται κυρίως για να απλοποιούν τα Expressions των δικτύων.

- `Update` : Συνάρτηση που δεδομένου ενός node n και ενός service s σε μορφή πλειάδας, επιστρέφει την πλειάδα που περιγράφει το n μετά την τοποθέτηση του s σε αυτό.
- `Add` : Συνάρτηση που δεδομένου ενός node n και ενός service s σε μορφή πλειάδας, επιστρέφει Αληθής (True) εάν οι πόροι του n επαρκούν για να τοποθετηθεί σε αυτόν το s, διαφορετικά επιστρέφει Ψευδής (False).
- `Update_rm` : Συνάρτηση που δεδομένου ενός Resource Modelling rm, ενός service r και του επιθυμητού αριθμού replicas r σε μορφή πλειάδας, επιστρέφει την πλειάδα που περιγράφει το r μετά την τοποθέτηση r αντιγράφων του s στον αντίστοιχο cluster.
- `Available_replicas` : Συνάρτηση που δεδομένου του resource modelling ενός cluster και ενός service επιστρέφει το μέγεθος Available Replicas όπως αυτό ορίζεται από τη σχέση 2.1.

Το σύνολο συναρτήσεων analysis

Οι συναρτήσεις αυτές αποτελούν βοηθητικές συναρτήσεις που δεν είναι απαραίτητες για την υλοποίηση των δικτύων. Χρησιμοποιούνται για την εξαγωγή του Graph of Markings σε διάφορες επεξεργάσιμες μορφές, με σκοπό την ανάλυση του δικτύου.

- `explain` : Συνάρτηση που δεδομένου του Graph of Markings ενός δικτύου όπως αυτός υπολογίζεται από τη βιβλιοθήκη SNAKES, επιστρέφει τον πίνακα γειτνίασης του

γράφου και την αντιστοίχιση (mapping) των markings που αντιστοιχούν στους κόμβους του.

- `nparray2networkx` : Συνάρτηση που δεδομένου του πίνακα γειτνίασης ενός Graph of Markings ενός δικτύου σε μορφή numpy array επιστρέφει τον γράφο σε μορφή κατευθυνόμενου γράφου της βιβλιοθήκης networkx.

- `SNAKES2networkx` : Συνάρτηση που δεδομένου του Graph of Markings ενός δικτύου όπως αυτός υπολογίζεται από τη βιβλιοθήκη SNAKES, επιστρέφει τον γράφο σε μορφή κατευθυνόμενου γράφου της βιβλιοθήκης networkx. Αποτελεί συνδυασμό των `explain` και `nparray2networkx`.

- `txt2nparray` : συνάρτηση που ανακατασκευάζει τον πίνακα γειτνίασης σε μορφή numpy array από αρχείο txt.

- `txt2networkx` : συνάρτηση που ανακατασκευάζει τον γράφο σε μορφή κατευθυνόμενου γράφου της βιβλιοθήκης networkx από αρχείο txt.

- `recover` : Συνάρτηση που δεδομένου του mapping και του αριθμού που αντιστοιχεί σε ένα κόμβο του γράφου, επιστρέφει το αντίστοιχο marking του δικτύου.

- `final_states` : Συνάρτηση που χρησιμοποιεί την `recover` και μεθόδους της βιβλιοθήκης networkx για να εντοπίσει και να επιστρέψει τα markings των τελικών καταστάσεων του δικτύου, δηλαδή των markings από τα οποία κανένα transition δεν είναι πυροδοτήσιμο. Αυτές υπολογίζονται από τον graph of markings ως οι κόμβοι με μηδενικό αριθμό εξερχόμενων ακμών.

Το σύνολο συναρτήσεων metrics

Οι συναρτήσεις αυτές αποτελούν βοηθητικές συναρτήσεις που δεν είναι απαραίτητες για την υλοποίηση των δικτύων. Χρησιμοποιούνται για την ανάκτηση μετρικών από ένα πραγματικό σύστημα και την δυναμική κατασκευή Petri Nets.

- `transform` : Συνάρτηση που εκτελεί μετατροπές των μεγεθών cpu και ram μεταξύ διαφορετικών υποδιαιρέσεων.

- `get_node_resources` : Συνάρτηση που χρησιμοποιώντας το εργαλείο kubectl [1], επιστρέφει σε μορφή python dictionary τις μετρικές που περιγράφουν την κατάσταση ενός node.

- `node_tokenize` : Συνάρτηση που κατασκευάζει το token ενός node σε μορφή πλειάδας από ένα dictionary, όπως αυτό δίνεται από την `get_node_resources`.

- `get_cluster_resources` : Συνάρτηση που χρησιμοποιώντας το εργαλείο kubectl, επιστρέφει σε μορφή python dictionary τις μετρικές που περιγράφουν την κατάσταση ενός member cluster.

[1] χρησιμοποιείται το kubectl αντί για το kubernetes API για λόγους που αναφέρθηκαν στην υποενότητα 5.2.3

- `cluster_tokenize` : Συνάρτηση που κατασκευάζει το token του resource modelling ενός cluster σε μορφή πλειάδας από ένα dictionary, όπως αυτό δίνεται από την `get_node_resources`.

Το σύνολο συναρτήσεων util

Οι συναρτήσεις αυτές αποτελούν βοηθητικές συναρτήσεις που δεν είναι απαραίτητες για την υλοποίηση των δικτύων. Περιέχει όλες τις βοηθητικές συναρτήσεις που αναπτύχθηκαν και δεν υπάγονται σε καμία από τις παραπάνω κατηγορίες. Οι συναρτήσεις αυτές αποτελούν κατά κύριο λόγο σενάρια (scripts) που βασίζονται σε άλλες βοηθητικές συναρτήσεις και χρησιμοποιήθηκαν κυρίως για τη διευκόλυνση της δημιουργίας δοκιμαστικών σεναρίων κατά την ανάπτυξη της βιβλιοθήκης. Δεν αποτελούν απαραίτητες συναρτήσεις για τη χρήση της βιβλιοθήκης, και για τον λόγο αυτό δεν αναλύονται περαιτέρω. Παρόλα αυτά αναφέρονται τόσο για λόγους πληρότητας όσο και επειδή αποτελούν παραδείγματα χρήσης της βιβλιοθήκης.

5.3.2 Υλοποίηση Petri Nets

Για την υλοποίηση των Petri Nets που περιγράφηκαν σχεδιάστηκε μια συνάρτηση η οποία δεδομένων ορισμένων παραμέτρων επιστρέφει ένα `PnComponent` το οποίο υλοποιεί το αντίστοιχο Petri Net. Οι συναρτήσεις που υλοποιούν τα διάφορα Petri Nets περιγράφονται αναλυτικά παρακάτω. Ιδιαίτερη σημασία έχουν τα ονόματα των Places που ορίζονται από κάθε component, αφού έχουν πολύ σημαντικό ρόλο στον συνδυασμό των δικτύων, όπως περιγράφεται στην ενότητα 5.4.

Υλοποίηση Petri Net για Member Cluster

Για την υλοποίηση του Petri Net που περιγράφει τους Member Clusters υλοποιήθηκε η συνάρτηση `MultiNodeClusterPN` (κώδικας 5.2). Κατασκευάζει το δίκτυο που περιγράφεται στην ενότητα 4.2, και επιστρέφει το αντίστοιχο `PnComponent`. Το συγκεκριμένο component αποτελείται από τα places Pending, Running και Nodes και από ένα Transition, το In-Cluster_Placement.

Υλοποίηση Petri Net για Duplicated Propagatioin Policy

Για την υλοποίηση του Petri Net που περιγράφει ένα Duplicated Propagatioin Policy υλοποιήθηκε η συνάρτηση `PP_DuplicatedPN` (κώδικας 5.3). Όπως και η MultiNodeClusterPN, κατασκευάζει το δίκτυο που περιγράφεται στην ενότητα 4.3.1, και επιστρέφει το αντίστοιχο `PnComponent`. Το συγκεκριμένο component αποτελείται από το Transition Propagate, το place Services, και ένα Place με όνομα Ci $\forall i \in (1, N)$ όπου N ο αριθμός των Member Clusters. Για παράδειγμα, για ένα σύστημα με 2 Member Clusters θα έχει τα places C1 και C2. Τα Places αυτά συγχωνεύονται με τα places Pending των PnComponents των Member Clusters, δημιουργώντας ένα συνολικό δίκτυο.

ΚΩΔΙΚΑΣ 5.2: *Η Συνάρτηση* `MultiNodeClusterPN`

```
def MultiNodeClusterPN( name, pending=[], allocated=[], available=[],
    running=[] ):
      pn = PNComponent(name)
      pn.globals.append("from KarmadaPN.Functions import Update")
      pn.globals.append("from KarmadaPN.Functions import Add")
      #Places
      pn.add_place(Place("Pending"))
      pn.add_place(Place("Running"))
      pn.add_place(Place("Nodes"))
      #Transitions
      pn.add_transition( Transition( "In-Cluster_Placement",
          Expression("svc[1] > 0 and Add(node,svc[0])") ))
      #Arcs
      pn.add_input("Pending", "In-Cluster_Placement", Variable("svc"))
      pn.add_output("Running", "In-Cluster_Placement",
          Expression("svc[0]"))
      pn.add_output("Pending", "In-Cluster_Placement",
          Expression("(svc[0],svc[1]-1)"))
      pn.add_input("Nodes", "In-Cluster_Placement", Variable("node"))
      pn.add_output("Nodes", "In-Cluster_Placement",
          Expression("Update(node,svc[0])"))

 return pn
```

ΚΩΔΙΚΑΣ 5.3: *Η Συνάρτηση* `PP_DuplicatedPN`

```
def PP_DuplicatedPN (name,cluster_number:int=2):
    pn = PNComponent(name)
    pn.add_place(Place("Services"))
    pn.add_transition(Transition( "Propagate", Expression("policy ==
        'Duplicated'") ))
    pn.add_input( "Services", "Propagate",
        Tuple([Variable("policy"),Variable("svc")] ))

    for i in range(cluster_number):
        pn.add_place( Place(f"C{i+1}") )
        pn.add_output( f"C{i+1}", "Propagate", Expression("svc") )

    return pn
```

Υλοποίηση Petri Net για Divided - Weighted Propagatioin Policyμε Static Weights

Για την υλοποίηση του Petri Net που περιγράφει ένα Divided - Weighted Propagatioin Policy με Static Weights υλοποιήθηκε η συνάρτηση `PP_StaticWeightsPN` (κώδικας 5.4). Όπως και η PP_DuplicatedPN, κατασκευάζει το δίκτυο που περιγράφεται στην ενότητα 4.3.2, και επιστρέφει το αντίστοιχο `PnComponent`. Το συγκεκριμένο component αποτελείται από το Transition Propagate, το place Services, και ένα Place με όνομα Ci $\forall i \in (1, N)$ όπου N ο αριθμός των Member Clusters, όπως ακριβώς και στην PP_DuplicatedPN. Για την απλοποίηση των Expressions των Arcs, χρησιμοποιήθηκε η βοηθητική συνάρτηση `fi_static`, η οποία υπολογίζει τα βάρη του κάθε Member Cluster χρησιμοποιώντας τον αλγόριθμο 4.1.

ΚΩΔΙΚΑΣ 5.4: *Η Συνάρτηση `PP_StaticWeightsPN`*

```
def  PP_StaticWeightsPN(name, cluster_number:int=2):
    pn = PNComponent(name)
    pn.globals.append("from KarmadaPN.PNS.Propagation import fi_static
        as fs")
    pn.add_place(Place("Services"))
    pn.add_transition(Transition("Propagate",Expression("policy ==
        'Weighted_Static'")))
    pn.add_input("Services","Propagate", Tuple([Variable("policy"),
        Variable("svc")]))

    for i in range(cluster_number):
        pn.add_place(Place(f"C{i+1}"))
        pn.add_output(f"C{i+1}", "Propagate", Expression(f"(svc[0],
            fs(svc[2], svc[1], {i+1}))" ))

    return pn
```

όπου

```
def fi_static(replicas, weights, idx):
    from math import ceil
    import numpy as np
    suma = sum(weights)
    sweights = sorted(weights,reverse=True)
    res = [0 for _ in weights]
    rest = 0
    indexes = [i for i, x in sorted(enumerate(weights), key=lambda
        x:x[1], reverse=True)]
    for i,w in enumerate(sweights):
        a = ceil((replicas-rest)*w/sum(sweights[i:]))
        rest += a
        res[indexes[i]]= a
    return res[idx-1]
```

Σημείωση: Κατά τη διάρκεια των πειραμάτων τροποποιήθηκε η συνάρτηση `fi_static` *με σκοπό την αύξηση της ακρίβειας της μοντελοποίησης (βλ. 6.3.1). Η τροποποιημένη εκδοχή της φαίνεται στον κώδικα 6.1.*

Υλοποίηση Petri Net για Divided - Weighted Propagatioin Policyμε Dynamic Weights

Για την υλοποίηση του Petri Net που περιγράφει ένα Divided - Weighted Propagatioin Policyμε Static Weights υλοποιήθηκε η συνάρτηση `PP_DynamicWeightsPN` (κώδικας 5.5). Όπως και όλες οι προηγούμενες συναρτήσεις, κατασκευάζει το δίκτυο που περιγράφεται στην ενότητα 4.3.3, και επιστρέφει το αντίστοιχο `PnComponent`. Το συγκεκριμένο component αποτελείται από το Transition Propagate, το place Services,και ένα ζεύγος places με ονόματα Ci και Ci_Resource_Modeling $\forall i \in (1, N)$ όπου N ο αριθμός των Member Clusters. Όπως ακριβώς και στην PP_StaticWeightsPN. Για την απλοποίηση των Expressions των Arcs, χρησιμοποιήθηκε η βοηθητική συνάρτηση `fi_dynamic`, η οποία υπολογίζει τα βάρη του κάθε Member Cluster χρησιμοποιώντας τη συνάρτηση `fi_dynamic`, αλλά αντλώντας τα βάρη από το resource modelling αντί για τα ορίσματα της συνάρτησης, όπως περιγράφεται στην ενότητα 4.3.3.

Υλοποίηση Petri Net για Divided - Aggregated Propagation Policy

Για την υλοποίηση του Petri Net που περιγράφει ένα Aggregated Propagatioin Policy υλοποιήθηκε η συνάρτηση `PP_AggregatedPN` (κώδικας 5.6). Όπως και όλες οι προηγούμενες συναρτήσεις, κατασκευάζει το δίκτυο που περιγράφεται στην ενότητα 4.3.4, και επιστρέφει το αντίστοιχο `PnComponent`. Το συγκεκριμένο component αποτελείται από το Transition Propagate, το place Services, και ένα ζεύγος places με ονόματα Ci και Ci_Resource_Modeling $\forall i \in (1, N)$ όπου N ο αριθμός των Member Clusters. Για την απλοποίηση των Expressions των Arcs, χρησιμοποιήθηκε η βοηθητική συνάρτηση `fi_aggregated`, η οποία υπολογίζει τα βάρη του κάθε Member Cluster χρησιμοποιώντας τον αλγόριθμο 4.2.

5.4 Χρήση της βιβλιοθήκης

Για την κατασκευή του Petri Net ενός οποιουδήποτε Multi Cluster συστήματος, χρειάζεται ο συνδυασμός κατάλληλων PnComponents. Αρχικά, πρέπει να κατασκευαστεί ένα PnComponent για κάθε Member Cluster καθώς και ένα PnComponent για κάθε πολιτική. Εν συνεχεία χρειάζεται να γίνει ένωση των επιμέρους PnComponents μεταξύ τους. Αυτό πρακτικά γίνεται προσθέτοντας όλα τα επιμέρους PnComponents σε ένα γενικό PnComponent με την μέθοδο `add_component`, και έπειτα συγχωνεύοντας τα Pending places των Member Clusters με τα Ci των πολιτικών. Για να μπορέσει να προσπελάσει ο χρήστης το κατάλληλο Place η transition, χρησιμοποιείται μία προκαθορισμένη διαδικασία ονοματοδοσίας. Σε κάθε PnComponent από αυτά που κατασκευάζουν οι έτοιμες συναρτήσεις, τα places και τα transitions έχουν ένα όνομα, το οποίο προέρχεται από το όνομα του PnComponent συνοδευόμενο από τον χαρακτήρα '_' και έπειτα το όνομα του όπως αυτό ορίζεται στην ενότητα 5.3.2. Για παράδειγμα, έστω PnComponent ενός Member Cluster με όνομα Cluster4. Το Pending

ΚΩΔΙΚΑΣ 5.5: *Η Συνάρτηση* `PP_DynamicWeightsPN`

```
def  PP_DynamicWeightsPN(name, cluster_number:int=2):
    pn = PNComponent(name)
    pn.globals.append("from KarmadaPN.PNS.Propagation import fi_dynamic
        as fd")
    pn.globals.append("from KarmadaPN.Functions import Update_rm")
    pn.add_place(Place("Services"))
    pn.add_transition(Transition("Propagate", Expression("policy ==
        'Weighted_Dynamic'")))
    pn.add_input("Services", "Propagate", Tuple([Variable("policy"),
        Variable("svc")]))

    clusters = "[" + ','.join([f'c{i+1}' for i in
        range(cluster_number)]) + "]"

    for i in range(cluster_number):
        pn.add_place(Place(f"C{i+1}_Resource_Modeling"))
        pn.add_input(f"C{i+1}_Resource_Modeling", "Propagate",
            Variable(f"c{i+1}"))
        pn.add_output(f"C{i+1}_Resource_Modeling", "Propagate",
            Expression(f"Update_rm(c{i+1},svc[0],fd(svc,{clusters},{i+1}))"))
        pn.add_place(Place(f"C{i+1}"))
        pn.add_output(f"C{i+1}", "Propagate",
            Expression(f"(svc[0],fd(svc,{clusters},{i+1}))"))

    return pn
```

όπου

```
def fi_dynamic(svc, c, idx):
    from ..Functions import Available_replicas as AR
    cluster_number = len(c)
    weights = [AR(c[i],svc[0]) for i in range(cluster_number)]
    return fi_static(svc[1],weights,idx-1)
```

ΚΩΔΙΚΑΣ 5.6: *Η Συνάρτηση* `PP_AggregatedPN`

```
def  PP_AggregatedPN(name, cluster_number:int=2):
    pn = PNComponent(name)
    pn.globals.append( "from KarmadaPN.PNS.Propagation import
        fi_aggregated as fa" )
    pn.globals.append("from KarmadaPN.Functions import Update_rm")
    pn.add_place(Place("Services"))
    pn.add_transition(Transition( "Propagate", Expression("policy ==
        'Aggregated'")))
    pn.add_input("Services", "Propagate",
        Tuple([Variable("policy"),Variable("svc")]))

    clusters = "[" + ','.join([f'c{i+1}' for i in
        range(cluster_number)]) + "]"

    for i in range(cluster_number):
        pn.add_place(Place(f"C{i+1}_Resource_Modeling"))
        pn.add_input(f"C{i+1}_Resource_Modeling", "Propagate",
            Variable(f"c{i+1}"))
        pn.add_output(f"C{i+1}_Resource_Modeling", "Propagate",
            Expression(f"Update_rm(c{i+1},svc[0],fa(svc,{clusters},{i+1}))"))
        pn.add_place(Place(f"C{i+1}"))
        pn.add_output(f"C{i+1}", "Propagate", Expression(
            f"(svc[0],fa(svc,{clusters},{i+1}))"))
    return pn
```

όπου

```
def fi_aggregated(svc, c, idx):
    from ..Functions import Available_replicas as AR
    cluster_number = len(c)
    w = [ AR(c[i],svc[0]) for i in range(cluster_number)]
    clusters = sorted([(i+1,(w[i])) for i in range(cluster_number)],
        key=lambda x:x[1], reverse=True)
    x = max(min(svc[1]-sum([x[1] for x in clusters][:[x[0] for x in
        clusters].index(idx)]),w[idx-1]),0)
    return x
```

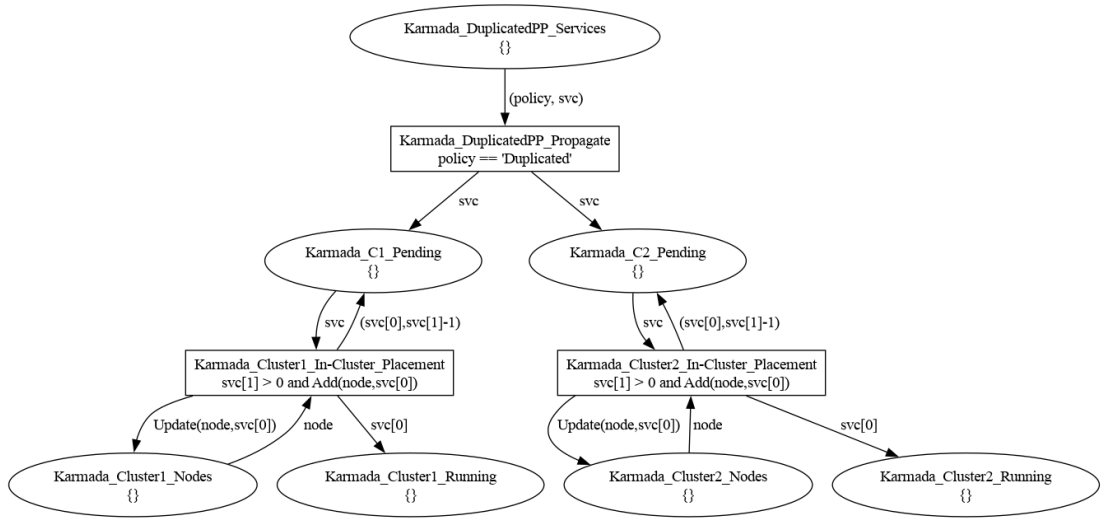
place του θα ονομάζεται `Cluster4_Pending`. Ακολούθως όταν το PnComponent προστεθεί σε ένα άλλο με τη μέθοδο `add_component`, κάθε place και transition του θα έχει όνομα το όνομα του δεύτερου PnComponent, ακολουθούμενο από τον χαρακτήρα '_' και έπειτα το όνομα που είχε προηγουμένως. Για παράδειγμα, εάν το PnComponent του Cluster4 του προηγούμενου παραδείγματος προστεθεί σε ένα PnComponent με όνομα Karmada το αντίστοιχο place θα μετονομαστεί σε `Karmada_Cluster4_Pending`. Αφού ολοκληρωθούν τα παραπάνω μπορεί το PnComponent να μετατραπεί σε `SNAKES.PetriNet` με τη μέθοδο `build`, και εν συνεχεία να αναλυθεί ορίζοντας Markings, κατασκευάζοντας Graph of Markins, γραφική αναπαράσταση ή οτιδήποτε άλλο χρειάζεται με βάση το σενάριο. Παράδειγμα της διαδικασίας αυτής φαίνεται και στον κώδικα 5.7.

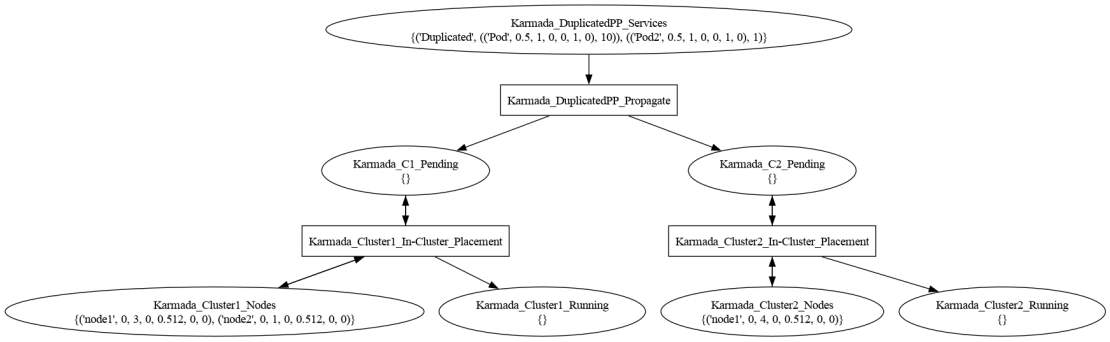
ΚΩΔΙΚΑΣ 5.7: *Παράδειγμα κατασκευής Petri Net για εφαρμογή με Duplicated Propagation Policy σε υποδομή με 2 Member Clusters*

```
from KarmadaPN.PNS import ClusterPN as CPN
from KarmadaPN.PNS import Propagation as P
from KarmadaPN import PN as PN
from KarmadaPN.Tokens import Service, Node
from KarmadaPN import SNAKES as nets

# ~~~~~~~~~~ PN Generation ~~~~~~~~~~

c1 = CPN.MultiNodeClusterPN("Cluster1")
# pn = karmada.build()
c2 = CPN.MultiNodeClusterPN("Cluster2")
# c3 = CPN.MultiNodeClusterPN("Cluster3")

p = P.PP_DuplicatedPN("DuplicatedPP",2)
# p = P.PP_DuplicatedPN("DuplicatedPP",3)

karmada = PN.PNComponent("Karmada")
karmada.add_component(p)
karmada.add_component(c1)
karmada.add_component(c2)
karmada.merge( "DuplicatedPP_C1", "Cluster1_Pending", "C1_Pending")
karmada.merge( "DuplicatedPP_C2", "Cluster2_Pending", "C2_Pending")

karmadapn = karmada.build()


print("Places")
for i in pn.place() : print(i)
print("Transitions")
for i in pn.transition() : print(i)

# Generate Empty Representation
karmadapn.draw("test_empty.png",)

karmadapn.set_marking( nets.Marking( Karmada_DuplicatedPP_Services=
    nets.MultiSet([("Duplicated",
    (Service("Pod",minCPU=0.5,maxCPU=1)(),10)),
    (Service("Pod2",minCPU=0.5,maxCPU=1)(), 1)]),
                          Karmada_Cluster1_Nodes= nets.MultiSet([
                              Node("node1",3,0.512)(),
                              Node("node2",1,0.512)()]),
                          Karmada_Cluster2_Nodes=
                              nets.MultiSet([Node("node1",4,0.512)()]),
)


# Generate Marked Representation (without expressions for simplicity)
karmadapn.draw("test_init.png",trans_attr=trmt,arc_attr=amt)
```

ΚΩΔΙΚΑΣ 5.8: *Έξοδος του κώδικα 5.7*

```
Places
Karmada_DuplicatedPP_Services
Karmada_Cluster1_Running
Karmada_Cluster1_Nodes
Karmada_Cluster2_Running
Karmada_Cluster2_Nodes
Karmada_C1_Pending
Karmada_C2_Pending

Transitions
Karmada_DuplicatedPP_Propagate
Karmada_Cluster1_In—Cluster_Placement
Karmada_Cluster2_In—Cluster_Placement
```





Κεφάλαιο 6

Έλεγχος σε πραγματικό Σύστημα (Proof of Concept)

Στο κεφάλαιο αυτό γίνεται ο έλεγχος καλής λειτουργίας του συστήματος. Όπως έχει εξηγηθεί και στα προηγούμενα κεφάλαια, στόχος της εργασίας είναι το σύστημα που υλοποιήθηκε να προσομοιώνει όσο πιο πιστά γίνεται την λειτουργία ενός πραγματικού Multi Cluster συστήματος. Για τον λόγο αυτό ο έλεγχος της υλοποίησης έγινε με χρήση πειραματικών σεναρίων. Τα σενάρια αυτά αποτελούν διάφορους συνδυασμούς πολιτικών διάδοσης και services που δρομολογούνται σε ένα Multi Cluster σύστημα. Συγκρίνοντας την συμπεριφορά του πραγματικού μοντέλου με τα τελικά Markings του Petri Net που περιγράφει το σύστημα, μπορούμε να ελέγξουμε εάν η πρόβλεψη για την μελλοντική κατάσταση του συστήματος συμπίπτει με την πραγματική απόκριση του.

6.1 Περιγραφή Υποδομής

Για την εκτέλεση των πειραμάτων, δημιουργήθηκε μία πραγματική Multi Cluster υποδομή. Η υποδομή αυτή αποτελείται από 4 εικονικές μηχανές, στις οποίες είναι εγκαταστημένο το λειτουργικό σύστημα Ubuntu. Σε μία από αυτές έχει εγκατασταθεί ένας Kubernetes Cluster με χρήση του Microk8s, μια ελαφριά διανομή που έρχεται προεγκατεστημένη με το λειτουργικό σύστημα Ubuntu Server. Ο Cluster αυτός φιλοξενεί το control-plane του Karmada. Στα υπόλοιπα 3 VMs έχει εγκατασταθεί η διανομή K3S του kubernetes [30], μία διανομή με πολύ μικρή απαίτηση σε υπολογιστικούς πόρους, η οποία χρησιμοποιείται κυρίως σε συστήματα Edge Computing και IoT.[31]

Αναφορικά με τα services που εφαρμόστηκαν στο σύστημα, δημιουργήθηκαν 3 deployments, με διαφορετικές απαιτήσεις σε πόρους. Κάθε ένα από αυτά τα deployments δημιουργεί 1 pod βασισμένο στην εικόνα `vish/stress` με απαιτήσεις πόρων που φαίνονται στον πίνακα 6.1.

Όνομα	minimum cpu	maximum cpu	minimum memory	maximum memory
svc1	0.2	1	1	512Mi
svc1	0.1	1	512Mi	1024Mi

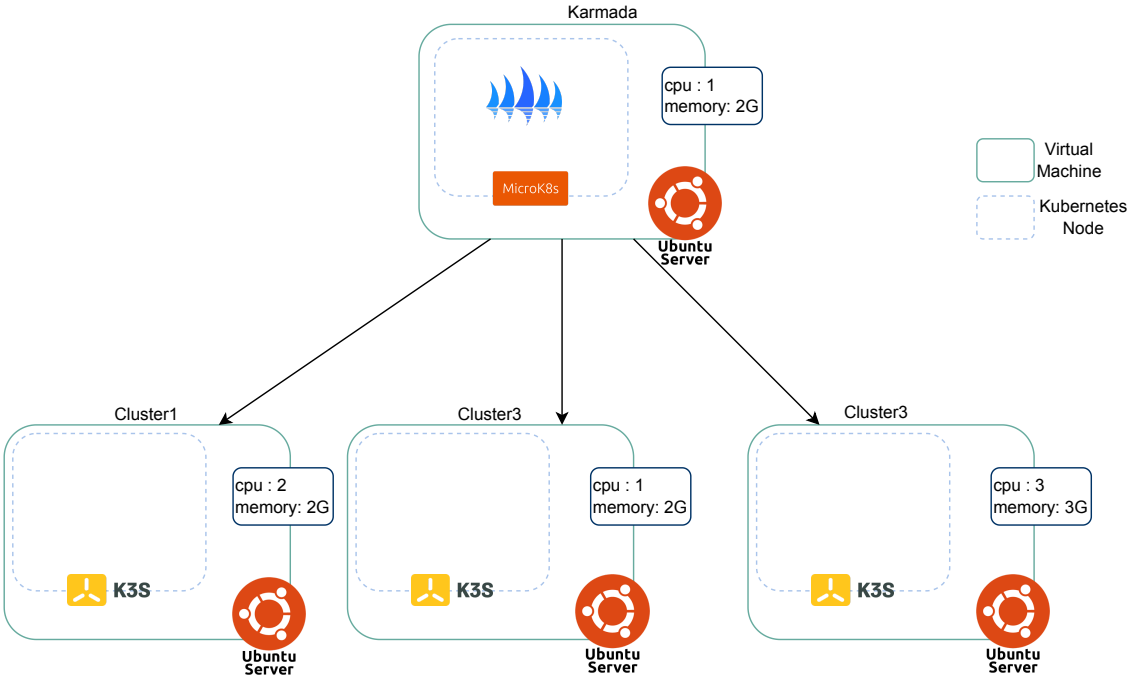
Πίνακας 6.1: *Services πειραμάτων*

Όνομα	Token
svc1	$('svc1', 20, 100, 1, 536870912, 1, 1)$
svc3	$('svc1', 10, 100, 536870912, 536870912, 1, 1)$

Πίνακας 6.2: *Tokens για τα διάφορα services*

Στο παράρτημα Α΄ φαίνεται το manifest για το πρώτο service. Τα υπόλοιπα προκύπτουν από το ίδιο manifest αντικαθιστώντας τα χωρία `limits.cpu` , `limits.memory`, `requests.cpu`,`requests.memory` στο `spec.template.spec.cointers[0]`

Σημείωση: Στη συνέχεια του κεφαλαίου μπορούμε αντικαθιστούμε τα tokens με το όνομα του κάθε service. Για παράδειγμα, το token $(svc1, 20, 100, 1, 512000000, 1, 1)$ *μπορεί να αντικατασταθεί με* $svc1$



Σχήμα 6.1: *Υποδομή*

6.2 Μεθοδολογία

Για τον έλεγχο του συστήματος ακολουθήθηκε η παρακάτω διαδικασία. Για κάθε ένα από τα deployments εξετάστηκε κάθε πολιτική δρομολογώντας στο σύστημα αριθμό replicas από 1 έως 28 για κάθε πολιτική, εκτός από την Duplicated, για την οποία δρομολογήθηκαν από 1 έως 16 replicas. Ο λόγος που επιλέχθηκαν οι αριθμοί αυτοί είναι γιατί αποτελούν τον μέγιστο αριθμό replicas του deployment "svc1" που μπορούν να δρομολογηθούν στο σύστημα χωρίς κάθε νέο replica να είναι Pending. Για λόγους ομοιογένειας τα ίδια εύρη χρησιμοποιήθηκαν και για το deployment "svc2". Για την αξιολόγηση της πολιτικής Divided, Weighted με Static Weights, δοκιμάστηκε μία σειρά από βάρη τα οποία στοχεύουν στην εξέταση διαφορετικών edge cases. Συγκεκριμένα εξετάστηκαν τα βάρη τα οποία :

- Μοιράζουν τα replicas ομοιόμορφα σε όλους ή μερικούς από τους clusters ((1, 1, 1), (1, 2, 2))
- Μοιράζουν τα replicas με βάση την ακριβή χωρητικότητα του κάθε cluster ((9, 4, 14) , (9, 8, 14))
- Μοιράζουν τα replicas με βάση την προσεγγιστική χωρητικότητα του κάθε cluster ((2, 1, 3))
- Μοιράζουν τα replicas με βάση αυθαίρετα βάρη ((13, 7, 28), (1, 2, 3))

Για την εξασφάλιση της ακρίβειας των αποτελεσμάτων και λόγω της μη ντετερμινιστικής λειτουργίας των συστημάτων, κάθε ένα από τα 18 σενάρια που προέκυψαν επανελήφθη 5 φορές, και συγκρίθηκε με τα αποτελέσματα τόσο η συχνότερα εμφανιζόμενη τελική κατάσταση, όσο και το αν η πρόβλεψη του μοντέλου συμπίπτει με κάποια από τις τελικές καταστάσεις του συστήματος, ακόμα και αν αυτή δεν είναι η συχνότερη.

6.3 Αποτελέσματα

Οι καταστάσεις που προκύπτουν από ένα συγκεκριμένο πείραμα ορίζουν ένα σύνολο S. Η συχνότερα εμφανιζόμενη από τις καταστάσεις αυτές ορίζεται ως $s^* \in S$. Επίσης, ορίζουμε ως p την κατάσταση που προκύπτει από τη μοντελοποίηση του αντίστοιχου πειράματος. Με βάση τα παραπάνω εξετάστηκαν 3 διαφορετικές μετρικές για τον έλεγχο της προσομοίωσης.

Η πρώτη μετρική, μετρά το εάν η πρόβλεψη του μοντέλου συμπίπτει με την πιο συχνά εμφανιζόμενη κατάσταση του πραγματικού μοντέλου, δηλαδή $p = s^*$. Η μετρική αυτή αν και αποτελεσματική, παρουσιάζει ορισμένους περιορισμούς. Αρχικά, λόγω του περιορισμένου αριθμού των επαναλήψεων που εκτελέστηκαν τα πειράματα, καταστάσεις με ίση ή περίπου ίση πιθανότητα εμφάνισης με την συχνότερα εμφανιζόμενη κατάσταση μπορεί να φαίνονται ως λιγότερο συχνές. Έτσι, ενώ μπορεί να προβλέπονται από το μοντέλο, η μετρική επιστρέφει αποτυχία δημιουργώντας false negatives, δηλαδή καταστάσεις όπου ενώ το μοντέλο προβλέπει μία πιθανή κατάσταση, αυτή δεν προσμετράται θετικά για την αξιολόγηση του.

Το πρόβλημα αυτό λύνει μερικώς η δεύτερη μετρική, η οποία μετρά εάν η πρόβλεψη του μοντέλου εμφανίστηκε σε κάποιο από τα πειράματα, δηλαδή εάν $p \in S$. Έτσι η μετρική αυτή δε δημιουργεί false negatives. Και οι δύο παραπάνω μετρικές παρουσιάζουν τον εξής περιορισμό: δεν είναι ικανές να αξιολογήσουν εάν η πρόβλεψη του μοντέλου αποκλίνει πολύ ή λίγο από τις πραγματικές καταστάσεις. Αυτό σημαίνει ότι, στις περιπτώσεις όπου το μοντέλο προσεγγίζει αρκετά την πραγματική κατάσταση (π.χ. υπολογίζει επιτυχώς τη χρησιμοποίηση (utilisation) των member clusters, αλλά αποτυγχάνει στον ακριβή υπολογισμό των replicas ανά cluster, οι παραπάνω μετρικές τις χαρακτηρίζουν εξίσου λάθος με καταστάσεις που δεν προσεγγίζουν καθόλου το επιθυμητό αποτέλεσμα. Για παράδειγμα, εάν σε ένα σύστημα η πραγματική απόκριση είναι ($cluster1 : 10, cluster2 : 0, cluster3 : 5$) η πρόβλεψη ($cluster1 : 9, cluster2 : 0, cluster3 : 6$)) είναι σαφέστατα καλύτερη από τις προβλέψεις ($cluster1 : 15, cluster2 : 0, cluster3 : 0$) ή ($cluster1 : 0, cluster2 : 0, cluster3 : 15$). Όμως για τις παραπάνω μετρικές όλες οι προβλέψεις είναι εξίσου λανθασμένες.

Για τον λόγο αυτό εξετάστηκε και η ομοιότητα (similarity). Συγκεκριμένα υπολογίστηκαν η απόσταση συνημίτονου (cosine distance) και η ευκλείδεια απόσταση (euclidean distance), πάνω στα διανύσματα των τελικών Markings του πραγματικού συστήματος και του μοντέλου. Η κύρια διαφορά τους έγκειται στη συμμετοχή του μήκους των διανυσμάτων στον υπολογισμό της απόστασης. Η απόσταση συνημίτονου λαμβάνει υπόψιν μόνο τη γωνία των διανυσμάτων ενώ η ευκλείδεια απόσταση τόσο το μήκος όσο και τη γωνία. Στην περίπτωση της αξιολόγησης μας η ευκλείδεια απόσταση δίνει περισσότερη έμφαση στον αριθμό των replicas, ενώ η Απόσταση συνημιτόνου στον διαμοιρασμό τους στους member clusters.

Πέρα από την αξιολόγηση του μοντέλου, μελετήθηκε και η μεταβολή του χρόνου και των καταστάσεων του Graph of Markings, σε σχέση με την κλιμάκωση των δρομολογούμενων replicas.

6.3.1 Παραμετροποίηση με βάση τα αποτελέσματα

Κατά τη διάρκεια των πειραμάτων, παρατηρήθηκαν διάφορες συμπεριφορές του πραγματικού μοντέλου που απέκλιναν τόσο από την αναμενόμενη λειτουργία του, όπως αυτή προκύπτει από την τεκμηρίωση [17], όσο και από τη συνήθη συμπεριφορά του. Ένα παράδειγμα αυτής της συμπεριφοράς ήταν αποκλίσεις στη λειτουργία της Divided, Weighted Propagation Policy με Static weights. Η πολιτική αυτή παρουσίαζε μια διαφοροποίηση σχετικά με τη στρογγυλοποίηση των δρομολογούμενων replicas, η οποία περιγράφεται αναλυτικότερα και στην υπο-ενότητα 4.3.2. Κατά τις μετρήσεις φάνηκε πως ο αλγόριθμος αυτός δεν κάλυπτε δύο πολύ συγκεκριμένες ακραίες περιπτώσεις edge-cases.Στην πρώτη από αυτές, όταν το αποτέλεσμα που προκύπτει από τη στρογγυλοποίηση σε έναν cluster είναι μικρότερο από το ακέραιο μέρος της τιμής πριν τη στρογγυλοποίηση, τότε το karmada δρομολογεί ένα παραπάνω replica στον clusterαυτό, και ένα λιγότερο στον αμέσως προηγούμενό του. Στη δεύτερη, παρατηρήθηκε ότι σε περιπτώσεις όπου η τιμή πριν τη στρογγυλοποίηση ήταν ακέραια, αλλά οι υπόλοιπες τιμές έχρηζαν στρογγυλοποίησης, τότε το karmada στρογγυλοποιούσε και την ακέραια τιμή προς τον επόμενο ακέραιο, αυξάνοντας την πρακτικά κατά ένα. Οι περιπτώσεις αυτές αν και παρατηρήθηκαν εποπτικά, φάνηκε να είναι συνεπείς στα πειράματα που είχαν εκτελεστεί μέχρι εκείνη τη στιγμή. Δοκιμάστηκε λοιπόν η εφαρμογή τους στον κώδικα, μέσω της αλλαγής στη συνάρτηση `fi_static` που φαίνεται στον κώδικα 6.1. Εκτελέστηκαν επιπρόσθετα πειράματα χρησιμοποιώντας τη νέα αλλαγή και σε αυτά το μοντέλο παρουσίασε σημαντικά καλύτερη ικανότητα πρόβλεψης. Το γεγονός αυτό αναδεικνύει την ευελιξία και ικανότητα παραμετροποίησης τόσο της υλοποίησης της παρούσας εργασίας όσο και γενικότερα των μοντελοποιήσεων βασισμένων σε Petri Nets.

ΚΩΔΙΚΑΣ 6.1: *Η Συνάρτηση `PP_StaticWeightsPN` μετά την παραμετροποίηση*

```
def fi_static(replicas, weights, idx):
    from math import ceil
    import numpy as np
    suma = sum(weights)
    sweights = sorted(weights,reverse=True)
    res = [0 for _ in weights]
    rest = 0
    indexes = [i for i, x in sorted(enumerate(weights), key=lambda x:
        x[1],reverse=True)]
    natural = [replicas*w/sum(weights) for w in weights]

    for i,w in enumerate(sweights):
        a = ceil((replicas-rest)*w/sum(sweights[i:]))# if
            sum(sweights[i:])>0 else 0
        # changes start here:
        if i ==len(sweights)-1 and a < int(natural[indexes[i]]):
            a += 1
            res[indexes[i-1]] -=1
        elif i == 0 and a == natural[indexes[i]] and not all([x%1 == 0
            for x in natural]):
            a += 1
            res[indexes[i+1]] -=1
        # changes end here.
        rest += a
        res[indexes[i]]= a

    return res[idx-1]
```

6.3.2 Αποτελέσματα μετρικών

Στην ενότητα αυτή παρουσιάζονται αναλυτικά τα αποτελέσματα των πειραμάτων ανά πολιτική και συγκεντρωτικά για όλες τις πολιτικές.

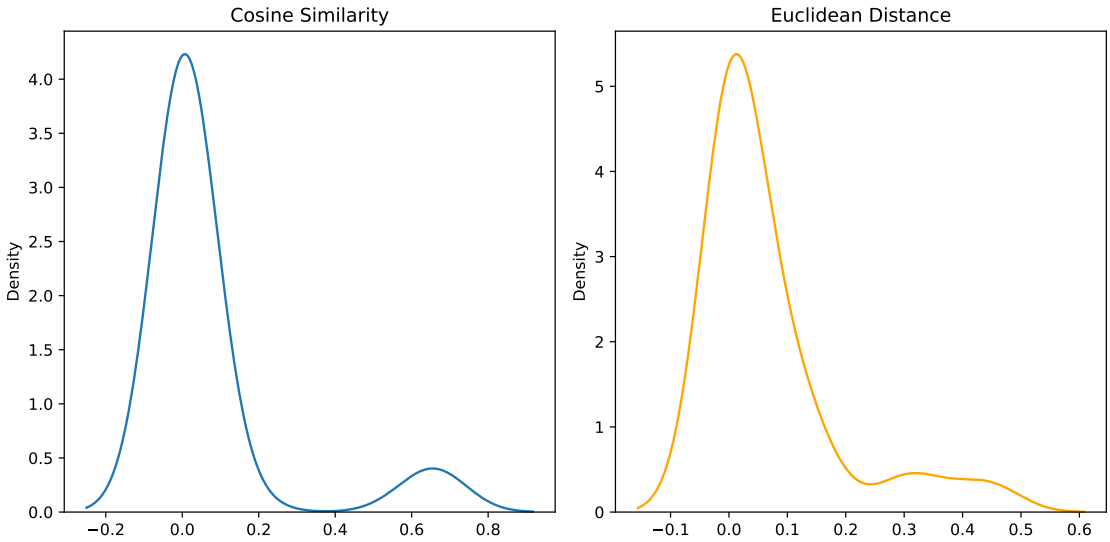
Divided, Aggregated Propagation Policy

Παρακάτω φαίνονται τα αποτελέσματα των μετρικών για την Divided, Aggregated Propagation Policy. Η πολιτική αυτή, αν και παρουσιάζει ικανότητα πρόβλεψης, εμφανίζει χαμηλά αποτελέσματα για ένα σημαντικό ποσοστό των πειραμάτων. Αυτό οφείλεται σε μία λειτουργία του Karmada, την οποία επειδή δεν εμφανίζεται ξεκάθαρα στην τεκμηρίωση αλλά παρατηρήθηκε πειραματικά, δε λαμβάνει υπόψιν η μοντελοποίηση της παρούσας εργασίας. Στην περίπτωση όπου ο ελάχιστος αριθμός member clusters παραμένει ίδιος και ένας εκ των χρησιμοποιούμενων member clusters τείνει να φτάσει το ανώτατο όριο replicas, το Karmada προτιμά να μοιράζει τα replicas στους χρησιμοποιημένους clusters. Έτσι παρατηρούνται σημαντικές διαφορές. Επίσης, παρατηρήθηκαν αστοχίες σε περιπτώσεις όπου, λόγω σφαλμάτων, το Karmada οδηγούνταν σε καταστάσεις που δε συμφωνούσαν με

την τεκμηρίωση.

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	31	25	55.36%	44.64%
$p \in S$	38	18	67.86%	32.14%

Πίνακας 6.3: *Αποτελέσματα πειραμάτων για Divided, Aggregated PP για $p = s^*$ και $p \in S$.*



Σχήμα 6.2: *Κατανομές Ομοιότητας για Divided, Aggregated PP για Απόσταση Συνημιτόνου (αριστερά) και Ευκλείδεια Απόσταση (δεξιά)*

Μετρική	Μέσος	Διακύμανση
Συνημιτόνου	0.06704619798920523	0.18491242882778747
Ευκλείδεια	0.06741323579775894	0.11538023442695293

Πίνακας 6.4: *Αποτελέσματα για Divided, Aggregated PP για τις μετρικές Απόσταση Συνημιτόνου και Ευκλείδεια Απόσταση.*

Duplicated Propagation Policy

Παρακάτω φαίνονται τα αποτελέσματα των μετρικών για την Duplicated Propagation Policy. Η πολιτική αυτή, παρουσίασε πλήρη ικανότητα πρόβλεψης σε οποιαδήποτε μετρική και ανεξαρτήτως αριθμού replicas. Η μοντελοποίηση κατάφερε να προβλέψει ακριβώς οποιαδήποτε κατάσταση του πραγματικού συστήματος. Αυτό οφείλεται στην απλότητα της πολιτικής, και την ντετερμινιστική φύση των βαρών της, σε αντίθεση με άλλες πολιτικές, για τις οποίες λόγω ελλιπούς τεκμηρίωσης έπρεπε να γίνουν παραδοχές για τη λειτουργία τους, ή παρουσίαζαν μη ντετερμινιστική συμπεριφορά ως προς τη δρομολόγηση, την οποία δεν μπορούν να προβλέψουν τα δίκτυα που υλοποιήθηκαν στα πλαίσια της εργασίας.

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	32	0	100.00%	0.00%
$p \in S$	32	0	100.00%	0.00%

Πίνακας 6.5: *Αποτελέσματα πειραμάτων για Duplicated PP για $p = s^*$ και $p \in S$.*

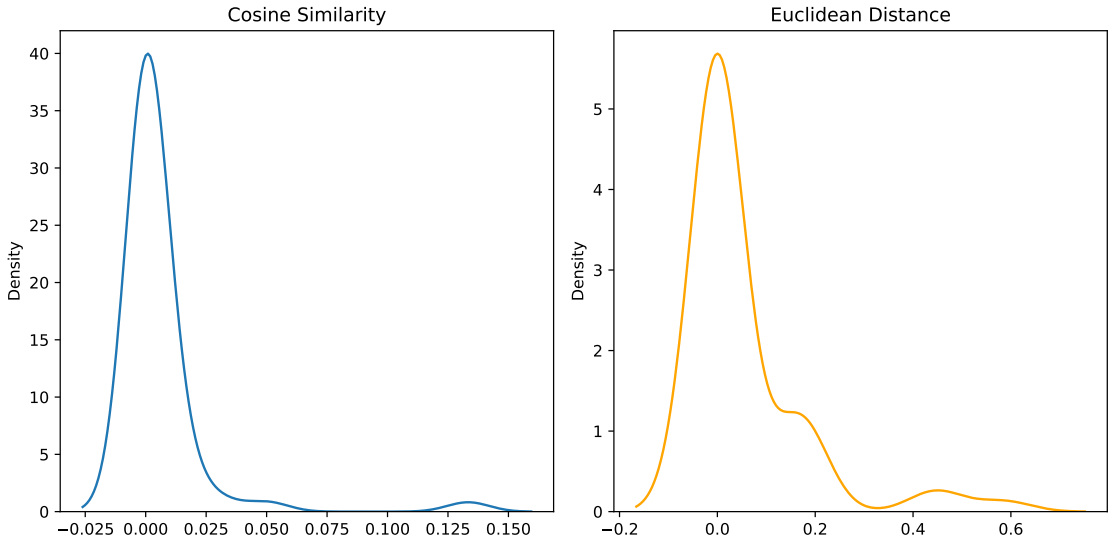
Για τη συγκεκριμένη πολιτική δεν υπολογίστηκαν οι μετρικές ομοιότητας, καθώς και στις δύο περιπτώσεις τα διανύσματα είναι πάντα όμοια.

Divided, Weighted Propagation Policy με Dynamic Weights

Παρακάτω φαίνονται τα αποτελέσματα των μετρικών για τη Divided, Weighted Propagation Policy με Dynamic Weights. Η πολιτική αυτή, παρουσιάζει ικανοποιητική ικανότητα πρόβλεψης. Οι αστοχίες που σημειώνονται οφείλονται κυρίως στη στοχαστική φύση ορισμένων καταστάσεων, στις οποίες οι τελικές καταστάσεις ήταν εξίσου πιθανές, με αποτέλεσμα το μοντέλο να προβλέπει πραγματικές καταστάσεις, αλλά όχι απαραίτητα τη συχνότερη. Επίσης, παρατηρήθηκαν αστοχίες σε περιπτώσεις όπου, λόγω σφαλμάτων, το Karmada οδηγούνταν σε καταστάσεις που δε συμφωνούσαν με την τεκμηρίωση. Αυτό επιβεβαιώνεται και από τη δεύτερη μετρική, όπου σημειώνει επιτυχία σε σχεδόν κάθε πείραμα.

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	44	12	78.57%	21.43%
$p \in S$	55	1	98.21%	1.79%

Πίνακας 6.6: *Αποτελέσματα πειραμάτων για Divided, Weighted PP με Dynamic Weights για $p = s^*$ και $p \in S$.*



Σχήμα 6.3: *Κατανομές Ομοιότητας για Divided, Weighted PP με Dynamic Weights για Απόσταση Συνημιτόνου (αριστερά) και Ευκλείδεια Απόσταση (δεξιά)*

Μετρική	Μέσος	Διακύμανση
Συνημιτόνου	0.006784284644144842	0.053266195542008364
Ευκλείδεια	0.017036181807513175	0.051826238389563184

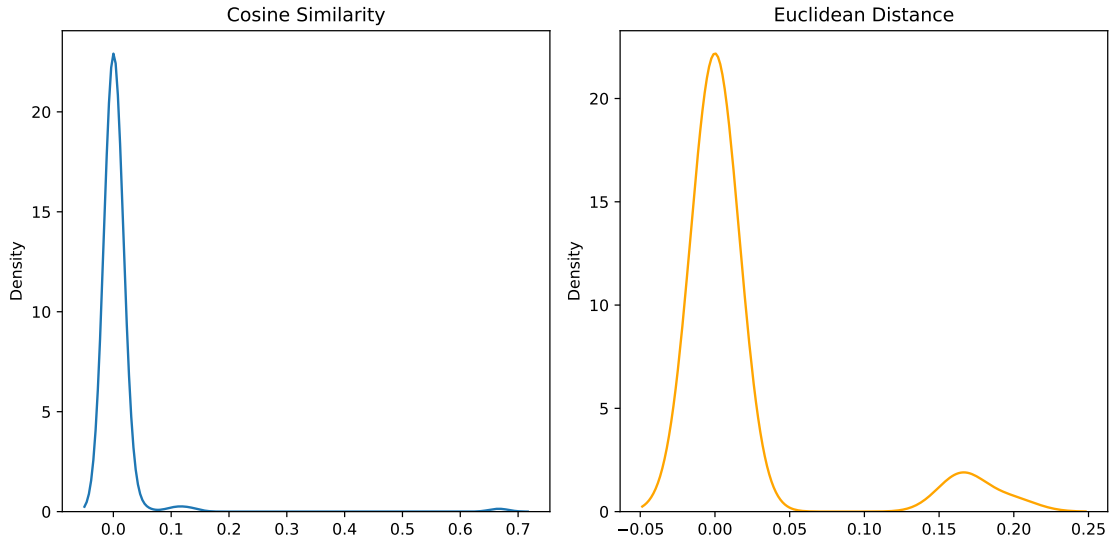
Πίνακας 6.7: *Αποτελέσματα για Divided, Weighted PP με Dynamic Weights για τις μετρικές Απόσταση Συνημιτόνου και Ευκλείδεια Απόσταση.*

Divided, Weighted Propagation Policy με Static Weights

Παρακάτω φαίνονται τα αποτελέσματα των μετρικών για την Divided, Weighted Propagation Policy με Static Weights. Η πολιτική αυτή παρουσιάζει εξαιρετικά ικανοποιητική ικανότητα πρόβλεψης. Όπως και στην Divided, Weighted Propagation Policy με Dynamic Weights, οι αστοχίες που σημειώνονται οφείλονται κυρίως στη στοχαστική φύση ορισμένων καταστάσεων, στις οποίες οι τελικές καταστάσεις ήταν εξίσου πιθανές, με αποτέλεσμα το μοντέλο να προβλέπει πραγματικές καταστάσεις, αλλά όχι απαραίτητα τη συχνότερη. Η αυξημένη ακρίβεια σε σχέση με την προηγούμενη οφείλεται στο γεγονός ότι εάν τα βάρη που ορίζει ο χρήστης διαφέρουν μεταξύ τους, η συμπεριφορά του Karmada είναι ντετερμινιστική. Έτσι τα μόνα σενάρια στα οποία εμφανίζονται αστοχίες είναι αυτά στα οποία τα replicas μοιράζονται ομοιόμορφα σε μερικούς ή σε όλους τους member clusters. Αυτό επιβεβαιώνεται και από τη δεύτερη μετρική, όπου σημειώνει επιτυχία σε κάθε πείραμα ανεξαρτήτως βάρους.

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	247	33	88.21%	11.79%
$p \in S$	280	0	100.00%	0.00%

Πίνακας 6.8: *Συγκεντρωτικά αποτελέσματα πειραμάτων για Divided, Weighted PP με Static Weights για $p = s^*$ και $p \in S$.*



Σχήμα 6.4: *Κατανομές Ομοιότητας για Divided, Weighted PP με Static Weights για Απόσταση Συνημιτόνου (αριστερά) και Ευκλείδεια Απόσταση (δεξιά)*

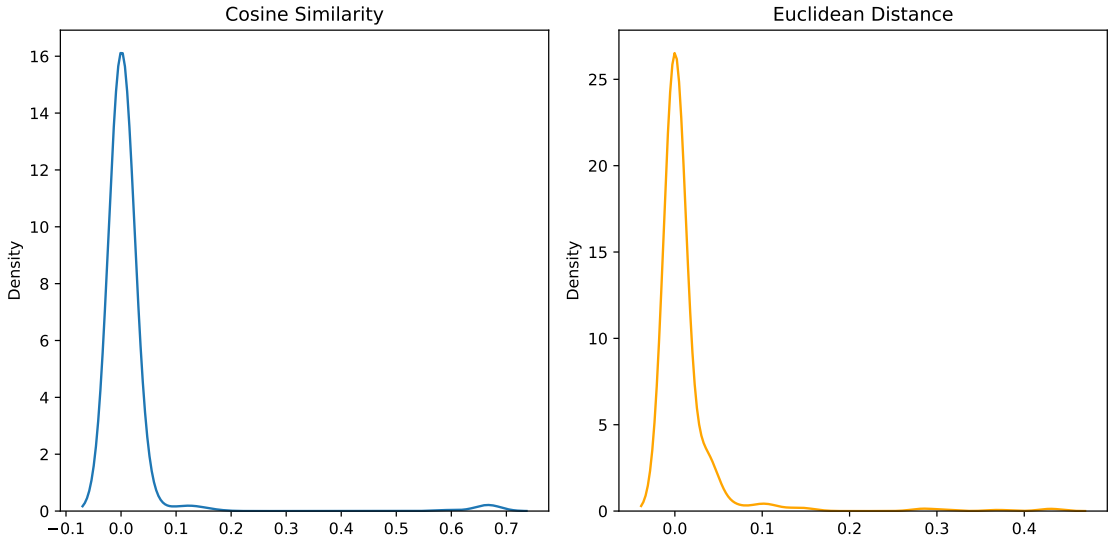
Μετρική	Μέσος	Διακύμανση
Συνημιτόνου	0.013218349274655117	0.08002540787152781
Ευκλείδεια	0.011371467666693933	0.04420433300518274

Πίνακας 6.9: *Αποτελέσματα για Divided, Weighted PP με Static Weights για τις μετρικές Απόσταση Συνημιτόνου και Ευκλείδεια Απόσταση.*

Συγκεντρωτικά

Παρακάτω φαίνονται τα συγκεντρωτικά αποτελέσματα για όλες τις πολιτικές. Λόγω του μεγαλύτερου αριθμού πειραμάτων που εκτελέστηκαν για την Divided, Weighted PP με Static Weights, παρουσιάζονται σε δύο μορφές: Αφενός λαμβάνοντας κάθε πείραμα ξεχωριστά 6.10 και κανονικοποιώντας τα αποτελέσματα της συγκεκριμένης πολιτικής δια το πλήθος των διαφορετικών βαρών που δοκιμάστηκαν. Έτσι η συμμετοχή της στο τελικό αποτέλεσμα εξισώνεται με αυτή των υπόλοιπων πολιτικών.

Συνολικά η υλοποίηση παρουσιάζει σημαντική ικανότητα πρόβλεψης. Δεδομένου ότι οι αστοχίες στην πλειονότητα των περιπτώσεων προκύπτουν από την ίδια τη στοχαστικότητα του συστήματος, αλλά ακόμα και σε αυτές τις περιπτώσεις η μοντελοποίηση προσεγγίζει με αξιοσημείωτη ομοιότητα την τελική κατάσταση, το ποσοστό των επιτυχιών καθιστά τη μοντελοποίηση μας ικανή να περιγράψει επαρκώς τη συντριπτική πλειοψηφία των πιθανών καταστάσεων που μπορεί να φτάσει μία Multi Cluster υποδομή βασισμένη στο Karmada. Λαμβάνοντας μάλιστα υπόψη τη δυνατότητα βελτίωσης της υπάρχουσας υλοποίησης, όπως έγινε για παράδειγμα στην υποενότητα 6.3.1, τα ήδη υψηλά ποσοστά που παρουσιάζονται δύνανται να αυξηθούν περαιτέρω.



Σχήμα 6.5: *Συνολικές Κατανομές Ομοιότητας για Απόσταση Συνημιτόνου (αριστερά) και Ευκλείδεια Απόσταση (δεξιά)*

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	354	70	83.49%	16.51%
$p \in S$	405	19	95.52%	4.48%

Πίνακας 6.10: *Συνολικά Αποτελέσματα για $p = s^*$ και $p \in S$.*

Μετρική	Επιτυχή πειράματα	Αποτυχημένα Πειράματα	Ποσοστό Επιτυχίας	Ποσοστό Αποτυχίας
$p = s^*$	156.4	43.6	78.20%	21.80%
$p \in S$	181	19	90.50%	9.50%

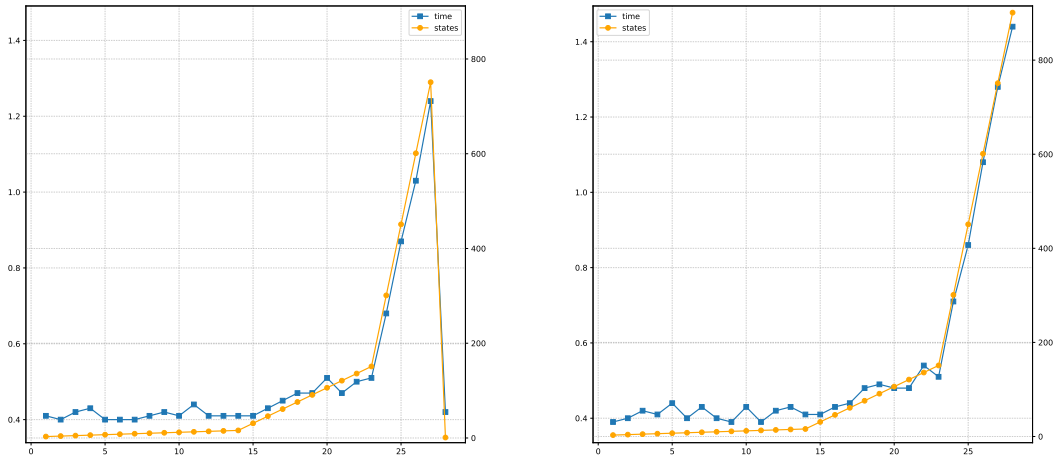
Πίνακας 6.11: *Κανονικοποιημένα Συνολικά Αποτελέσματα για $p = s^*$ και $p \in S$.*

Μετρική	Μέσος	Διακύμανση
Συνημιτόνου	0.013218349274655117	0.08002540787152781
Ευκλείδεια	0.01137146766693933	0.04420433300518274

Πίνακας 6.12: *Συνολικά Αποτελέσματα για τις μετρικές Απόσταση Συνημιτόνου και Ευκλείδεια Απόσταση.*

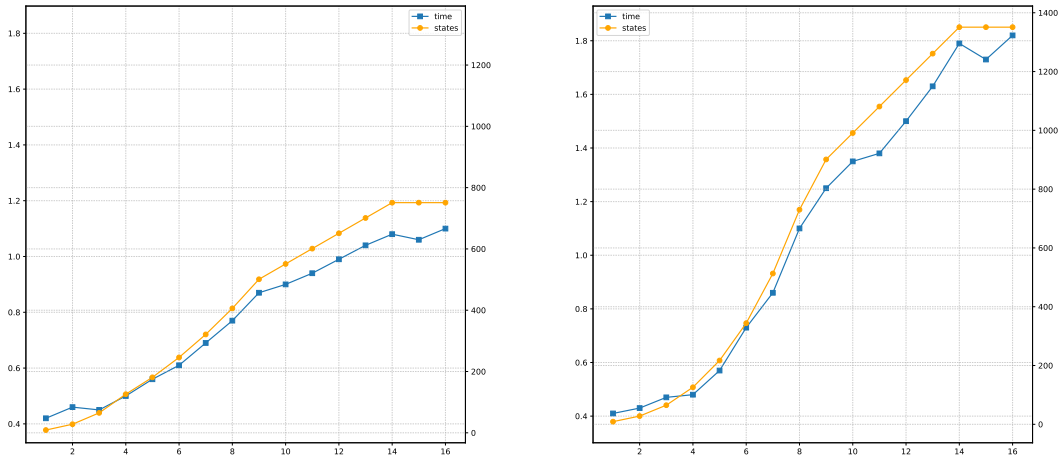
6.3.3 Ανάλυση χρόνου

Παρακάτω φαίνονται διαγράμματα που απεικονίζουν τη μεταβολή του χρόνου εκτέλεσης της μοντελοποίησης και των παραγόμενων καταστάσεων του graph of Markings, συνάρτησει του αριθμού των δρομολογούμενων replicas. Αξίζει να παρατηρηθεί ότι ο χρόνος εκτέλεσης είναι ανάλογος των παραγόμενων states, γεγονός αναμενόμενο καθώς οι υπόλοιπες ενέργειες του κώδικα πλην της κατασκευής και αναζήτησης καταστάσεων στο γράφο είναι ανεξάρτητες από τον αριθμό των replicas. Επίσης, παρατηρώντας τα διαγράμματα, η πολυπλοκότητα της διαδικασίας φαίνεται να είναι χρονικά πολυωνυμική. Αυτό επιβεβαιώνεται και από την τυπική μελέτη πολυπλοκότητας, που λαμβάνει χώρα στο παράρτημα Β΄.



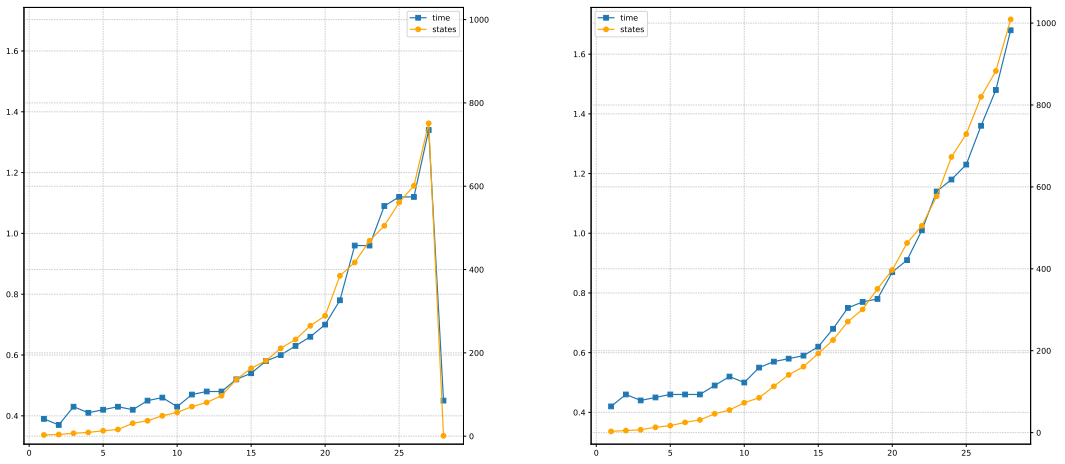
(α') *svc1* (β') *svc2*

Σχήμα 6.6: *Διαγράμματα χρόνου και καταστάσεων για Divided, Aggregated Propagation Policy*



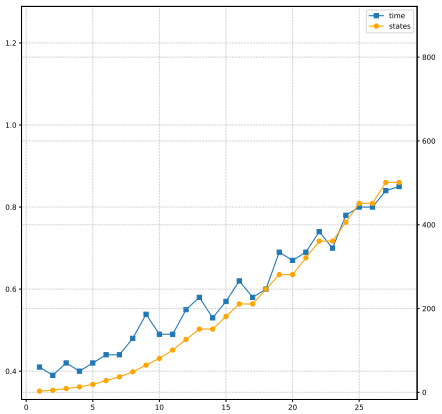
(α') *svc1* (β') *svc2*

Σχήμα 6.7: *Διαγράμματα χρόνου και καταστάσεων για Duplicated Propagation Policy*

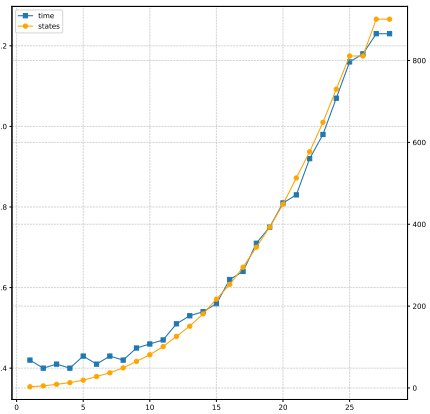


(α') *svc1* (β') *svc2*

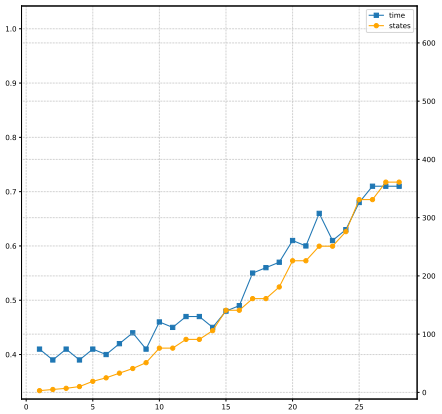
Σχήμα 6.8: *Διαγράμματα χρόνου και καταστάσεων για Divided, Weighted Propagation Policy με Dynamic weights*



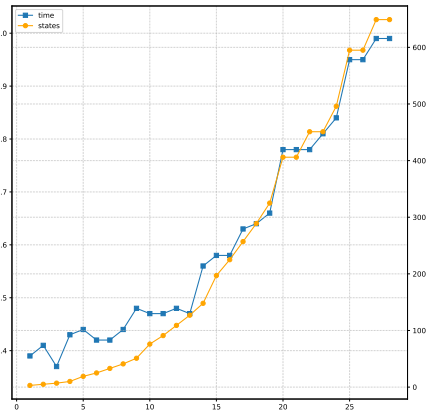
(α') *svc1*



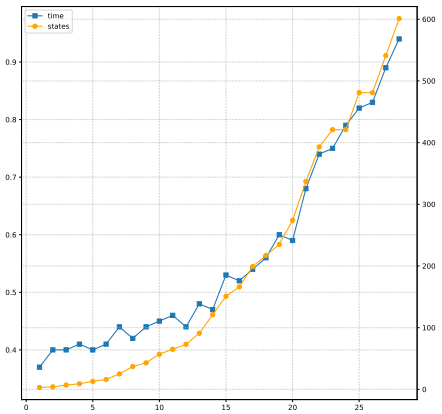
(β') *svc2*



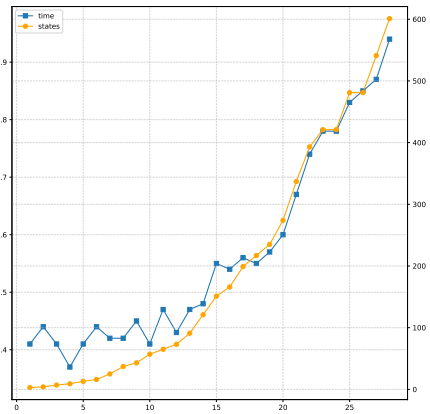
(γ') *svc1*



(δ') *svc2*

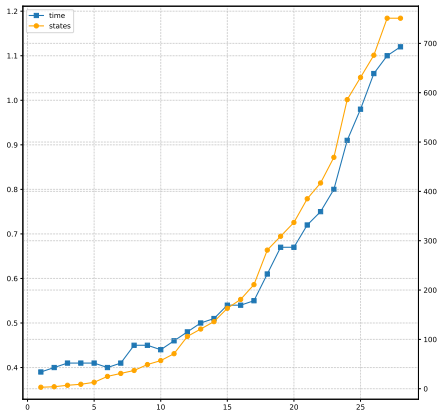


(ε') *svc1*

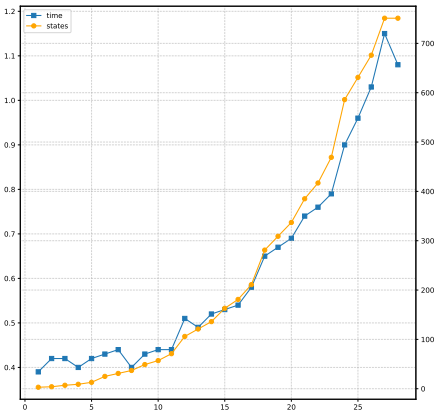


(ϛ') *svc2*

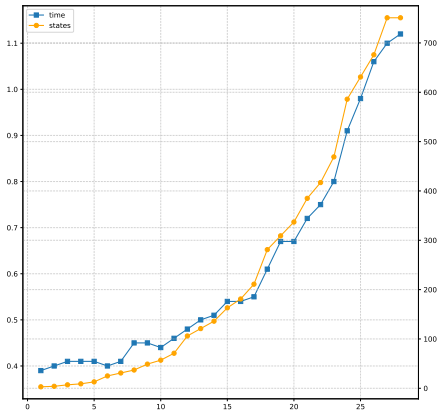
Σχήμα 6.9: *Διαγράμματα χρόνου και καταστάσεων για Divided, Weighted Propagation Policy με Static weights (I)*



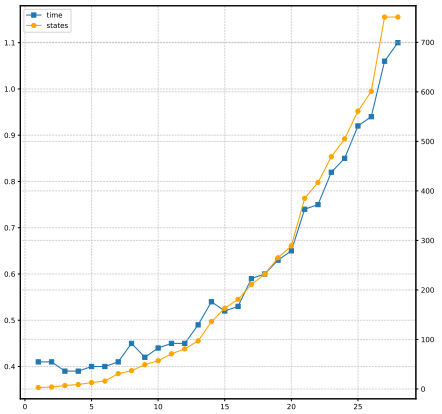
(α') *svc1*



(β') *svc2*



(γ') *svc1*



(δ') *svc2*

Σχήμα 6.10: *Διαγράμματα χρόνου και καταστάσεων για Divided, Weighted Propagation Policy με Static weights (II)*

Επίλογος

Συμπεράσματα

Εξαιτίας της ραγδαίας ανάπτυξης τεχνολογιών, όπως τα δίκτυα 5ης γενιάς και το Cloud Continuum, ανάλογη πρωτοπορία και πρόοδο παρουσιάζει και η ανάπτυξη λογισμικών διαχείρισης virtualised και containerised εφαρμογών, παραδείγματα των οποίων αποτελούν οι ενορχηστρωτές, όπως το Kubernetes, και τα λογισμικά ενορχήστρωσης σε Multi Cluster συστήματα, όπως το Karmada. Παρά την πρωτοπορία αυτή, η δυναμικότητα των συνθηκών στις απαιτήσεις των εφαρμογών (ανάπτυξη replicas, συχνές αλλαγές κλπ), καθώς και αυξημένη πολυπλοκότητα και κλιμάκωσή τους, καθιστούν αναγκαία την ανάπτυξη λύσεων και μηχανισμών που αφορούν την προληπτική συντήρηση (predictive maintenance) των συστημάτων. Τέτοιου είδους μηχανισμοί μπορούν να κάνουν τα συστήματα αυτά περισσότερο ανθεκτικά (resilient) και εύρωστα (robust).

Για τον λόγο αυτό προτείναμε τη μοντελοποίηση ειδικότερα Multi Cluster υποδομών με χρήση δικτύων Petri και παρατηρήσαμε ότι:

Η χρήση PNs για modelling υπολογιστικών συστημάτων στο φάσμα του υπολογιστικού νέφους, αποτελεί μια αποτελεσματική μέθοδο τόσο για τη μοντελοποίηση όσο και για τον έλεγχο των συστημάτων αυτών. Όπως φαίνεται και από τη βιβλιογραφία, τα δίκτυα Petri μπορούν να απεικονίσουν πληθώρα συστημάτων, στα οποία περιλαμβάνονται και συστήματα ενορχήστρωσης Containerised εφαρμογών με χρήση του Kubernetes [21][22]. Για τον λόγο αυτό εξετάστηκε η χρήση των δικτύων Petri για τη μοντελοποίηση της ενορχήστρωσης εφαρμογών σε Multi-Cluster συστήματα, τα οποία χαρακτηρίζονται από αυξημένη πολυπλοκότητα και ετερογένεια.

Τα αποτελέσματα της εργασίας παρουσιάζουν ποιοτικά χαρακτηριστικά με ικανότητα πρόβλεψης ακολουθώντας διαφορετικές πολιτικές. Η υλοποίηση παρουσιάζει ικανοποιητικά αποτελέσματα στη συντριπτική πλειοψηφία των περιπτώσεων, ποσοστό της τάξης του 90%, φτάνοντας ανάλογα με την πολιτική έως και 100% ακρίβεια πρόβλεψης.

Σε περιπτώσεις όπου παρουσιάστηκαν αστοχίες το μοντέλο παρουσίασε δυνατότητες προσαρμογής οι οποίες επιτρέπουν την καλύτερη ικανότητα πρόβλεψης. Λαμβάνοντας υπόψη τις αιτίες που προκάλεσαν τις αστοχίες αυτές, είναι βέβαιο πως με κατάλληλη τροποποίηση αντίστοιχες μοντελοποιήσεις μπορούν να επιτύχουν πολύ καλύτερες επιδόσεις, προσεγγίζοντας κατά πολύ τα μηδενικά ποσοστά αστοχίας.

Μελλοντικές Επεκτάσεις

Η υλοποίηση που αναπτύχθηκε στα πλαίσια αυτής της διπλωματικής εργασίας θα μπορούσε να βελτιωθεί και να επεκταθεί περαιτέρω ως προς διάφορες κατευθύνσεις. Συγκεκριμένα, αναφέρονται οι ακόλουθες:

- **Βελτίωση Ακρίβειας.** Όπως παρατηρούμε στο κεφάλαιο 6, ενώ η προτεινόμενη μοντελοποίηση μπορεί να προβλέψει τη συμπεριφορά ενός Multi Cluster συστήματος στην πλειοψηφία των περιπτώσεων, υπάρχουν περιπτώσεις στις οποίες αποτυγχάνει να προβλέψει τη συμπεριφορά. Παρόλα αυτά, ένα από τα προσόντα των CPNs είναι η ικανότητα παραμετροποίησής τους. Μπορούν λοιπόν τα μοντέλα που παρουσιάζονται στην εργασία να τροποποιηθούν κατάλληλα, ώστε να επιτύχουν μεγαλύτερη ακρίβεια. Μάλιστα, στα πλαίσια της εργασίας εντοπίστηκαν ήδη πιθανές βελτιώσεις που μπορούν να εφαρμοστούν στην υπάρχουσα υλοποίηση για τη βελτίωση της ακρίβειας πρόβλεψης.

- **Σχεδιασμός και υλοποίηση δικτύων για περισσότερες διαδικασίες.** Η παρούσα εργασία εστιάζει στο πρόβλημα της τοποθέτησης (placement) των εφαρμογών. Όμως υπάρχει μία πληθώρα διαδικασιών που λαμβάνουν χώρα σε ένα Multi CLuster σύστημα και θα μπορούσαν να μοντελοποιηθούν χρησιμοποιώντας δίκτυα Petri. Ενδεικτικά αναφέρονται προβλήματα όπως η διαδικασία ολοκλήρωσης των Pods (Pod completion), η αυτόματη διαχείριση σφαλμάτων (failover), καθώς και η δυνατότητα ομόσπονδης αυτοματοποιημένης κλιμάκωσης (fedrated autoscaling). Όλα αυτά αποτελούν διαδικασίες που ενώ υλοποιούνται από το Karmada [17], δεν καλύπτονται από τα μοντέλα τις εργασίας. Η δυνατότητα μοντελοποίησης τους θα παρείχε μία πληρέστερη μοντελοποίηση των διαδικασιών που συμβαίνουν στο σύστημα και θα μπορούσε να χρησιμοποιηθεί για την επίλυση περισσότερων προβλημάτων.

- **Ανάπτυξη συστήματος αξιολόγησης καταστάσεων.** Παρόλο που τα δίκτυα petri αποτελούν χρήσιμο εργαλείο για τη μοντελοποίηση ενός συστήματος, εξίσου χρήσιμη είναι και η αξιολόγηση των αποτελεσμάτων της μοντελοποίησης αυτής. Για τον λόγο αυτό, ενδιαφέρουσα επέκταση θα ήταν η ανάπτυξη ενός συστήματος, το οποίο χρησιμοποιώντας προκαθορισμένες συνθήκες, κλασσικούς αλγορίθμους, ή ακόμα και τεχνητή νοημοσύνη, θα έχει τη δυνατότητα να αξιολογήσει τα αποτελέσματα της προσομοίωσης, χαρακτηρίζοντας τις τελικές καταστάσεις ως επιθυμητές ή ανεπιθύμητες, ή οποιασδήποτε άλλης μορφής αξιολόγηση.

- **Ανάπτυξη συστήματος ενημερωμένης τοποθέτησης με χρήση της προσομοίωσης.** Ως συνέπεια της προηγούμενης επέκτασης, τα αποτελέσματα τις αξιολόγησης των καταστάσεων μπορούν να χρησιμοποιηθούν για την αξιολόγηση των συνεπειών αρχικών επιλογών για το σύστημα, όπως πολιτικών ή κατακόρυφης κλιμάκωσης. Προτείνεται λοιπόν ο σχεδιασμός και η ανάπτυξη συστήματος που θα χρησιμοποιεί τη γνώση αυτή για να εφαρμόσει αυτόματα βέλτιστες ρυθμίσεις για την τοποθέτηση εφαρμογών στο αρχικό σύστημα, χρησιμοποιώντας αντί για τις δυνατότητες αυτόματης δυναμικής τοποθέτησης που παρέχει το karmada ή και το kubernetes, τις βέλτιστες αρχικές συνθήκες που προκύπτουν από τη μοντελοποίηση. Το σύστημα αυτό θα μπορούσε να χρησιμοποιεί εξαντλητική αναζήτηση στον χώρο των αρχικών Markings, αλγόριθμους αναζήτησης βέλτιστης λύσης, αλγορίθμους οι οποίοι βελτιστοποιούν πολλαπλές αντικειμενικές συναρτήσεις (Multi Objective) ακόμα και μεθόδους μηχανικής μάθησης, που χρησιμοποιούν τα αποτελέσματα των προσομοιώσεων μέσω δικτύων petri για την εκπαίδευση μοντέλων βέλτιστης τοποθέτησης.

- **Αυτοματοποιημένη δημιουργία δικτύων από αρχεία manifest**. Ενώ τα δίκτυα Petri που σχεδιάστηκαν και υλοποιήθηκαν δίνουν τη δυνατότητα για επαρκή προσομοίωση των διαδικασιών που μοντελοποιούν, παρουσιάζουν ένα σημαντικό μειονέκτημα. Η χρήση τους για τη μοντελοποίηση ακόμα και του πιο απλού συστήματος, χρειάζεται μελέτη της δομής και των ιδιοτήτων των petri nets, καθώς και εκτενή μελέτη του κώδικα. Λύση στα παραπάνω θα ήταν η δημιουργία κατάλληλων προγραμματιστικών εργαλείων, είτε σε μορφή συναρτήσεων είτε σε μορφή προγράμματος, τα οποία θα μπορούν να παράγουν αυτόματα το δίκτυο petri ενός συστήματος, χρησιμοποιώντας πληροφορίες από τα αρχεία manifest των αντικειμένων, με ελάχιστη ή καθόλου παραμετροποίηση από τον χρήστη. Έτσι, η μοντελοποίηση θα ήταν λιγότερο επιρρεπής σε σφάλματα λόγω κακής διαμόρφωσης, οδηγώντας και σε ορθότερα αποτελέσματα. Ακόμη η επέκταση αυτή σε συνδυασμό με τις προηγούμενες επεκτάσεις θα μπορούσε να ανάγει τα δίκτυα petri, από ένα μαθηματικό μοντέλο αναπαράστασης, σε ένα ανεξάρτητο, αυτοματοποιημένο και εύχρηστο εργαλείο παρακολούθησης και προσομοίωσης διεργασιών για Multi cluster συστήματα.

Παραρτήματα

Παράρτημα Α΄

Manifest για το Deployment svc1

ΚΩΔΙΚΑΣ Α΄.1: *Manifest για το Deployment svc1*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: svc1
spec:
  replicas: 0
  selector:
    matchLabels:
      app: stress
  template:
    metadata:
      labels:
        app: stress
    spec:
      containers:
      - name: stress
        image: vish/stress
        resources:
          limits:
            cpu: "1"
            memory: "512Mi"

          requests:
            cpu: "0.2"
            memory: "1"

        ports:
        - containerPort: 80
```

Παράρτημα Β΄

Υπολογισμός Χρονικής Πολυπλοκότητας

Για την εύρεση της τελικής κατάστασης, αλλά και για οποιαδήποτε άλλη ανάλυση των πιθανών καταστάσεων του συστήματος, χρειάζεται να γίνει προσπέλαση του Graph of Markings. Η προσπέλαση αυτή έχει πολυπλοκότητα τουλάχιστον $O(V)$ όπου V το πλήθος των κόμβων του γράφου, δηλαδή των πιθανών καταστάσεων του συστήματος. Άρα για τον υπολογισμό της χρονικής πολυπλοκότητας αρκεί ο υπολογισμός των πιθανών καταστάσεων του συστήματος.

Έστω r ο αριθμός των δρομολογούμενων replicas και c ο αριθμός των clusters που χρησιμοποιούνται για δρομολόγηση. Από τη δομή ενός απλού Petri Net (αγνοώντας σενάρια multi policy και θεωρώντας single node clusters) μπορούμε να συμπεράνουμε τα εξής :

- Η πολυπλοκότητα ενός Propagation Policy είναι $\Pi_{pol} = O(1)$.
- Κάθε member cluster δρομολογεί κάθε replica ένα προς ένα, δημιουργώντας τόσα states όσα τα replicas που δρομολογήθηκαν σε αυτόν και έφτασαν σε running state, τα οποία στη χειρότερη περίπτωση είναι όσα τα συνολικά replicas ($\Pi_{cl} = O(r)$)
- Οι συνολικές καταστάσεις υπολογίζονται σαν τον συνδυασμό όλων των καταστάσεων κάθε member cluster με αυτές κάθε άλλου member cluster : $\Pi_{cl} \cdot \Pi_{cl} \cdot \Pi_{cl} \cdot \ldots = \Pi_{cl}^{c}$

Από τα παραπάνω συμπεραίνουμε ότι:

$$\Pi_{all} = \Pi_{pol} + \Pi_{cl}{}^{c} =$$

$$= O(1) + O^{c}(r) = O(1) + O(r^{c}) = O(r^{c})$$

Ισχύει ότι $c \leq C$ όπου C ο συνολικός αριθμός clusters του συστήματος. Άρα

$$\Pi_{all} = O(r^{C})$$

Όπως προκύπτει από τα χαρακτηριστικά των συστημάτων, ισχύει ότι $lim_{r \to \infty} c = C$. Επίσης, ισχύει ότι $c \leq r$, αφού πρόκειται για τον αριθμό των χρησιμοποιούμενων clusters, τους clusters δηλαδή στους οποίους δρομολογούνται replicas. Επειδή στην πλειοψηφία των συστημάτων το πλήθος των clusters είναι σταθερό, και για μεγάλο αριθμό replicas ισχύει ότι $C << r$.

Από τα παραπάνω προκύπτει ότι η πολυπλοκότητα είναι αυστηρά πολυωνυμική ως προς τον αριθμό των replicas.

Βιβλιογραφία

[1] Dinh C. Nguyen, Ming Ding, Pubudu N. Pathirana, Aruna Seneviratne, Jun Li, Dusit Niyato, Octavia Dobre και H. Vincent Poor. *6G Internet of Things: A Comprehensive Survey*. *IEEE Internet of Things Journal*, 9(1):359–383, 2022.

[2] Bäckströ H. Bravo C. *Edge computing and deployment strategies for communication service providers*. Τεχνική Αναφορά με αριθμό, Ericsson, East Lansing, Michigan, 2020.

[3] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob και Arif Ahmed. *Edge computing: A survey*. *Future Generation Computer Systems*, 97:219–235, 2019.

[4] R. David και H. Alla. *The Cloud-to-Thing Continuum*. Palgrave Macmillan Cham, 2020.

[5] Wei Bao, Dong Yuan, Zhengjie Yang, Shen Wang, Wei Li, Bing Zhou και Albert Zomaya. *Follow Me Fog: Toward Seamless Handover Timing Schemes in a Fog Computing Environment*. *IEEE Communications Magazine*, 55:72–78, 2017.

[6] Yun Chao Huet al. *Mobile edge computing—A key technology towards 5G*. Τεχνική Αναφορά με αριθμό 11, ETSI, 2015.

[7] Assad Abbas και Konal Khan. *Edge Computing: Extending the Cloud to the Edge of the Network*. 2023.

[8] Aleksandr Ometov, Oliver Molua, Mikhail Komarov και Jari Nurmi. *A Survey of Security in Cloud, Edge, and Fog Computing*. *Sensors*, 22:927, 2022.

[9] *Kubernetes*. `https://kubernetes.io/`. Ημερομηνία πρόσβασης: 23-03-2024.

[10] *Cisco Annual Internet Report (2018-2023) White Paper*. Τεχνική Αναφορά με αριθμό c11 741490, CISCO, 2020.

[11] Saravanan J, Saravanan P., J Saravanan και Saravanan Pichaimani. *An overview on role of Server Process in Virtualization -A case study of Cloud Computing*. *The International journal of analytical and experimental modal analysis*, 12:182–190, 2020.

[12] Zeyi Tao, Qi Xia, Zijiang Hao, Cheng Li, Lele Ma, Shanhe Yi και Qun Li. *A Survey of Virtual Machine Management in Edge Computing*. *Proceedings of the IEEE*, 107(8):1482–1499, 2019.

[13] Claus Pahl και Brian Lee. *Containers and Clusters for Edge Cloud Architectures - A Technology Review*. *2015 3rd International Conference on Future Internet of Things and Cloud*, σελίδες 379–386, 2015.

[14] Satya Bhushan Verma, Brijesh Pandey και Bineet Kumar Gupta. *Containerization and its Architectures: A Study*. *ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal*, 11(4):395–409, 2023.

[15] Sara Alraddady, Ben Soh, Mohammed A. AlZain και Alice S. Li. *Fog Computing: Strategies for Optimal Performance and Cost Effectiveness*. *Electronics*, 11(21), 2022.

[16] *CNCF Annual Survey 2022*. https://www.cncf.io/reports/cncf-annual-survey-2022/. Ημερομηνία πρόσβασης: 27-03-2024.

[17] *Open, Multi-Cloud, Multi-Cluster Kubernetes Orchestration | karmada*. https://karmada.io/. Ημερομηνία πρόσβασης: 02-04-2024.

[18] Cassandras C. και Lafortune S. *Introduction to Discrete Event Systems*. Springer Cham, 3η έκδοση, 2021.

[19] Lynn T. Mooney J. G. Lee B. Takako Endo P. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice Hall, 1992.

[20] Jensen K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1*. Springer Berlin, Heidelberg, 2η έκδοση, 1996.

[21] Víctor Medel, Omer Rana, José Ángel Bañares και Unai Arronategui. *Adaptive application scheduling under interference in Kubernetes*. *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, σελίδα 426–427, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Víctor Medel, Rafael Tolosana-Calasanz, JoséÁngel Bañares, Unai Arronategui και Omer F. Rana. *Characterising resource management performance in Kubernetes*. *Computers & Electrical Engineering*, 68:286–297, 2018.

[23] *Welcome to Python.org*. https://www.python.org/. Ημερομηνία πρόσβασης: 17-04-2024.

[24] *Python developers survey 2022*. https://lp.jetbrains.com/python-developers-survey-2022/. Ημερομηνία πρόσβασης: 17-04-2024.

[25] *Python developers survey 2022*. https://lp.jetbrains.com/python-developers-survey-2022/. Ημερομηνία πρόσβασης: 17-04-2024.

[26] *SNAKES is the net algebra kit for editors and simulators - SNAKES*. https://snakes.ibisc.univ-evry.fr/. Ημερομηνία πρόσβασης: 17-04-2024.

[27] Franck Pommereau. *SNAKES: A Flexible High-Level Petri Nets Library (Tool Paper)*. *Application and Theory of Petri Nets and Concurrency*Raymond Devillers και Antti Valmari, επιμελητές, σελίδες 254–265, Cham, 2015. Springer International Publishing.

[28] Nitin Sukhija και Elizabeth Bautista. *Towards a Framework for Monitoring and Analyzing High Performance Computing Environments Using Kubernetes and Prometheus. 2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, σελίδες 257–262, 2019.

[29] *NetworkX - NetworkX documentation*. `https://networkx.org/`. Ημερομηνία πρόσβασης: 17-04-2024.

[30] *K3S*. `https://k3s.io/`. Ημερομηνία πρόσβασης: 05-06-2024.

[31] Sebastian Böhm και Guido Wirtz. *Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes*. 2021.

Συντομογραφίες - Αρκτικόλεξα - Ακρωνύμια

βλ.	βλέπε
κ.λπ.	και λοιπά
δηλ.	δηλαδή
PN	Petri Net
CPN	Coloured Petri Net
VM	Virtual Machine
PP	Propagation Policy

Απόδοση ξενόγλωσσων όρων

Απόδοση	Ξενόγλωσσος όρος
κόμβος	node
νέφος	cloud
ομίχλη	fog
άκρα	edge
Διαδίκτυο των Αντικειμένων	Internet of Things
Δίκτυο Petri	Petri Net
Έγχρωμο Δίκτυο Petri	Coloured Petri Net
ενορχήστρωση	Orchestration
τοποθέτηση	placement
πολιτική διάδοσης	propagation policy
εικονική Μηχανή	Virtual Machine
δίκτυο Petri	Petri Net