Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Μελέτη Αποτίμησης Ελαστικής Διαχείρισης Δεδομένων σε Περιβάλλοντα Υπολογιστικών Νεφών

Διπλωματικη Εργασια

του

ΓΕΩΡΓΙΟΥ Φ. ΑΓΓΕΛΟΠΟΥΛΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2025

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Μελέτη Αποτίμησης Ελαστικής Διαχείρισης Δεδομένων σε Περιβάλλοντα Υπολογιστικών Νεφών

Διπλωματικη Εργασια

του

ΓΕΩΡΓΙΟΥ Φ. ΑΓΓΕΛΟΠΟΥΛΟΥ

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27η Φεβρουαρίου 2025.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Ιωάννης Κωνσταντίνου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2025

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

ΔΗΛΩΣΗ ΜΗ ΛΟΓΟΚΛΟΠΗΣ ΚΑΙ ΑΝΑΛΗΨΗΣ ΠΡΟΣΩΠΙΚΗΣ ΕΥΘΥΝΗΣ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ενυπογράφως ότι είμαι αποκλειστικός συγγραφέας της παρούσας Πτυχιακής Εργασίας, για την ολοκλήρωση της οποίας κάθε βοήθεια είναι πλήρως αναγνωρισμένη και αναφέρεται λεπτομερώς στην εργασία αυτή. Έχω αναφέρει πλήρως και με σαφείς αναφορές, όλες τις πηγές χρήσης δεδομένων, απόψεων, θέσεων και προτάσεων, ιδεών και λεκτικών αναφορών, είτε κατά κυριολεξία είτε βάσει επιστημονικής παράφρασης. Αναλαμβάνω την προσωπική και ατομική ευθύνη ότι σε περίπτωση αποτυχίας στην υλοποίηση των ανωτέρω δηλωθέντων στοιχείων, είμαι υπόλογος έναντι λογοκλοπής, γεγονός που σημαίνει αποτυχία στην Πτυχιακή μου Εργασία και κατά συνέπεια αποτυχία απόκτησης του Τίτλου Σπουδών, πέραν των λοιπών συνεπειών του νόμου περί πνευματικών δικαιωμάτων. Δηλώνω, συνεπώς, ότι αυτή η Πτυχιακή Εργασία προετοιμάστηκε και ολοκληρώθηκε από εμένα προσωπικά και αποκλειστικά και ότι, αναλαμβάνω πλήρως όλες τις συνέπειες του νόμου στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής άλλης πνευματικής ιδιοκτησίας.

(Υπογραφή)

.....
Γεώργιος Φ. Αγγελόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και
Μηχανικός Υπολογιστών Ε.Μ.Π
22 Φεβρουάριος 2025

Περίληψη

Η δυναμική φύση των Big Data απαιτεί την χρήση περιβαλλόντων υπολογιστικών νεφών και την υιοθέτηση μη παραδοσιακών τρόπων διαχείρισης τους. Υλοποιούμε σε περιβάλλον Kubernetes την Apache Cassandra με σκοπό την μελέτη της ελαστικότητας της, δηλαδή της ικανότητας της να ανταποκρίνεται δυναμικά σε ποικίλους φόρτους εργασίας χωρίς χειροκίνητη παρέμβαση. Παράλληλα θα εξερευνήσουμε τρόπους δυναμικής παροχής τόμων χρησιμοποιώντας εγγενή στοιχεία του Kubernetes. Μέσω των πειραμάτων παρατηρήσαμε και κατανοήσαμε πως σχεδιάζεται ένα κατανεμημένο υπολογιστικό σύστημα με γνώμονα την υψηλή διαθεσιμότητα, αντοχή σε κατατμήσεις δικτύου και ελαστικότητα. Μέσω της πειραματικής αποτίμησης παρατηρήσαμε ότι, παρά τους περιορισμούς σε πόρους πετύχαμε να διαχειριστούμε ελαστικά τα δεδομένα μέσω της Apache Cassandra.

Λέξεις Κλειδιά

Kubernetes, Apache Cassandra, Prometheus, YCSB, Δυναμική Παροχή Τόμων, Amazon Dynamo, Ελαστικότητα, Θεώρημα CAP, Κατανεμημένα Συστήματα, Υπολογιστικό Νέφος

Abstract

The dynamic nature of Big Data necessitates the use of cloud computing environments and the adoption of non traditional ways of managing them. Within a Kuberneters environment we deployed Apache Cassandra with the aim to study its elasticity, in other words its ability to respond dynamically to various workloads without manual intervention. At the same time, we examine ways of dynamically provisioning volumes utilizing native elements of Kubernetes. Through the experiments we observed and concluded the principles and methodologies by which a distributed computing system is designed with the aim of achieving high availability and network partition tolerance. The study concludes and highlights that despite the limited computing resources we successfully achieved elastic data management with the utilization of Apache Cassandra.

Keywords

Kubernetes, Apache Cassandra, Prometheus, YCSB, Dynamic Volume Provisioning, Amazon Dynamo, Elasticity, CAP Theorem, Distributed Systems, Cloud Computing

στη Μαρία-Ζωή

Ευχαριστίες

Θα ήθελα να ευχαριστήσω ιδιαιτέρως τον καθηγητή Δρ. Ιωάννη Κωνσταντίνου για την καθοδήγηση, καθώς και για την έμπνευση που μου έδωσε με το μάθημά του Προηγμένα Θέματα Βάσεων Δεδομένων για να προβώ στην επιλογή της συγκεκριμένης θεματολογίας για την εκπόνηση της παρούσας διπλωματικής εργασίας. Επίσης, θα ήθελα να ευχαριστήσω την οικογένεια μου για την στήριξη τους καθ'όλη την διάρκεια των σπουδών μου.

Αθήνα, Φεβρουάριος 2025

Γεώργιος Φ. Αγγελόπουλος

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Μέρος I

Εισαγωγή

Κεφάλαιο 1

Εισαγωγή

Η ραγδαία εξέλιξη του Διαδικτύου έχει οδηγήσει σε συνεχόμενα μεγαλύτερο αριθμό δεδομένων. Αυτά τα δεδομένα συνήθως αποκαλούνται Big Data, με κύρια χαρακτηριστικά τον μεγάλο όγκο τους, την ταχύτητα που δημιουργούνται και την ποικιλία τους ως προς το πόσο δομημένα είναι και ποια είναι η πηγή τους [6]. Αυτή η δυναμική συμπεριφορά τους έχει οδηγήσει στην ανάγκη της δυναμικής διαχείρισής τους. Αυτή η δυναμική διαχείριση απαιτεί διαφορετικές τεχνολογίες και υποδομές, μία από τις οποίες είναι τα υπολογιστικά νέφη. Τα υπολογιστικά νέφη είναι υπηρεσίες που προσφέρουν υπολογιστική και αποθηκευτική ισχύ ανάλογα με την επιθυμητή ζήτηση. Αυτή η δυναμική διάθεση πόρων επιτρέπει την διαχείριση των Big Data με τρόπο που δύσκολα τα παραδοσιακά τοπικά κέντρα μπορούν. Νέα συστήματα αποθήκευσης και διαχείρισης των δεδομένων εμφανίστηκαν που μπορούν να διαχειριστούν αυτά τα δεδομένα με αποδοτικό τρόπο, όπως το Apache Hadoop από τους Dean, και Ghemawat [7].

1.1 Αντικείμενο της διπλωματικής

Ορμώμενοι από το συγκεκριμένη στροφή στον τρόπο διαχείρισης του υπολογισμού θα εξετάσουμε, σε αυτήν την διπλωματική εργασία, την ελαστική διαχείριση δεδομένων σε περιβάλλοντα υπολογιστικών νεφών. Αυτά τα νέφη έχουν συνήθως δικές τους αφαιρέσεις πάνω από υπολογιστικούς πόρους που δεν επιτρέπουν την εύκολη αλλαγή παρόχου, με αποτέλεσμα τον εγκλωβισμό προμηθευτή [8]. Αυτό άλλαξε με την κυκλοφορία του Kubernetes [9] όπου αποτελεί μία διαπλατφορμική τεχνολογία [10]. Θα μελετήσουμε πως μπορούμε να χρησιμοποιήσουμε τις αφαιρέσεις και τα αντικείμενα του ούτως ώστε να υλοποιήσουμε ένα σύστημα που να λειτουργεί σε ένα υπολογιστικό νέφος και να μπορεί να διαχειριστεί αποδοτικά όγκο δεδομένων ελαστικά. Θα εξετάσουμε το κατανεμημένο, μη σχεσιακό σύστημα διαχείρισης βάσεων δεδομένων Apache Cassandra των Lakshman και Malik [11], καθώς και μία σειρά από εργαλεία και προγράμματα που θα μας βοηθήσουν στην εργασία αυτή.

Αρχικά θα εξετάσουμε σε θεωρητικό πλαίσιο διάφορα πεδία και τεχνολογίες που μετέπειτα θα χρησιμοποιήσουμε, υλοποιώντας διάφορα πειράματα ελαστικής διαχείρησης δεδομένων.

Μέρος II

Θεωρητικό Μέρος

Κεφάλαιο 2

Θεωρητικό υπόβαθρο

Στο συγκεκριμένο κεφάλαιο παρουσιάζονται αναλυτικά οι τέσσερις βασικές τεχνολογίες που χρησιμοποιηθήκαν στην εργασία, το σύστημα ενορχήστρωσης Kubernetes, το σύστημα διαχείρισης βάσεων δεδομένων Apache Cassandra, το πρόγραμμα αξιολόγησης επιδόσεων YCSB και το εργαλείο παρακολούθησης Prometheus.

2.1 Kubernetes

2.1.1 Τι είναι ο Kubernetes

Ο Kubernetes είναι μία πλατφόρμα ενορχήστρωσης Containers ανοιχτού κώδικα που σχεδιάστηκε με σκοπό την αυτοματοποίηση στη διάθεση και διαχείριση υπολογιστικών συστημάτων. Την ανάπτυξή της πραγματοποίησε αρχικά η Google το 2014, και μετέπειτα συντηρείται από το 2015 από το Cloud Native Computing Foundation (CNCF) [12]. Έρχεται σαν διάδοχος και εξέλιξη των συστημάτων διάθεσης εφαρμογών Container απλοποιώντας προκλήσεις των προηγούμενων συστημάτων, όπως είναι ο συγχρονισμός, η κλιμάκωση, εξισορρόπηση φόρτου, και η επανεκκίνηση Container [9].

Ο Kubernetes επιτρέπει στους χρήστες του να διαχειρίζονται τις εφαρμογές και τα συστήματα τους ανεξάρτητα από το αν φιλοξενούνται σε τοπικά κέντρα δεδομένων (on-premises) ή σε περιβάλλοντα υπολογιστικών νεφών (cloud) [10] χρησιμοποιώντας ένα δηλωτικό API [12]. Οι διαχειριστές και χρήστες του cluster περιγράφουν ποια είναι η επιθυμητή κατάσταση και ο Kubernetes πραγματοποιεί όλες τις ενέργειες για την επίτευξης της.

Ο Kubernetes αποτελείται από 2 ομάδες συστημάτων, τα στοιχεία του Επιπέδου Ελέγχου (Control Plane) και τα στοιχεία του Κόμβου (Node) [1].

Στοιχεία Control Plane

Τα στοιχεία Control plane λαμβάνουν αποφάσεις για όλο το σύστημα καθώς και διαχειρίζονται τα γεγονότα του cluster. Συνήθως τα στοιχεία αυτά φιλοξενούνται στον ίδιο κόμβο και συνίσταται να μην εκτελούνται άλλοι φόρτοι εργασίας στον συγκεκριμένο κόμβο για την πιο αποδοτική λειτουργία του Kubernetes [1].

Τα στοιχεία αυτά είναι:

- API Server: εκθέτει το Kubernetes API και διαχειρίζεται όλες τις αλληλεπιδράσεις μεταξύ του cluster, του συστήματος και των χρηστών [13].

- etcd: μία κατανεμημένη βάση κλειδιού-τιμής που αποθηκεύει την κατάσταση και τα δεδομένα διαμόρφωσης του cluster, εξασφαλίζοντας συνοχή στο σύστημα [14].

- Scheduler: υπεύθυνος για την δρομολόγηση και εκχώρηση των νέων Pods και φόρτων εργασίας στους κατάλληλους κόμβους [15].

- Controller Manager: εκτελεί διαχειριστικές εργασίες που παρακολουθούν την κατάσταση του cluster, των Nodes, και του φόρτου εργασιών, πραγματοποιώντας όλες τις απαραίτητες ενέργειες για να διατηρηθεί η επιθυμητή λειτουργία του, όπως κλιμάκωση, αναπαραγωγή και επανεκκίνηση μονάδων του συστήματος [16].

Σε εμπορικά περιβάλλοντα υπολογιστικών νεφών υπάρχει και ο Cloud Controller manager που ενσωματώνει τις εγγενείς υπηρεσίας του υπολογιστικού νέφους με τον Kubernetes, όπως Frontend UIs, πίνακες ελέγχου και δυναμική παροχή τόμων[17].

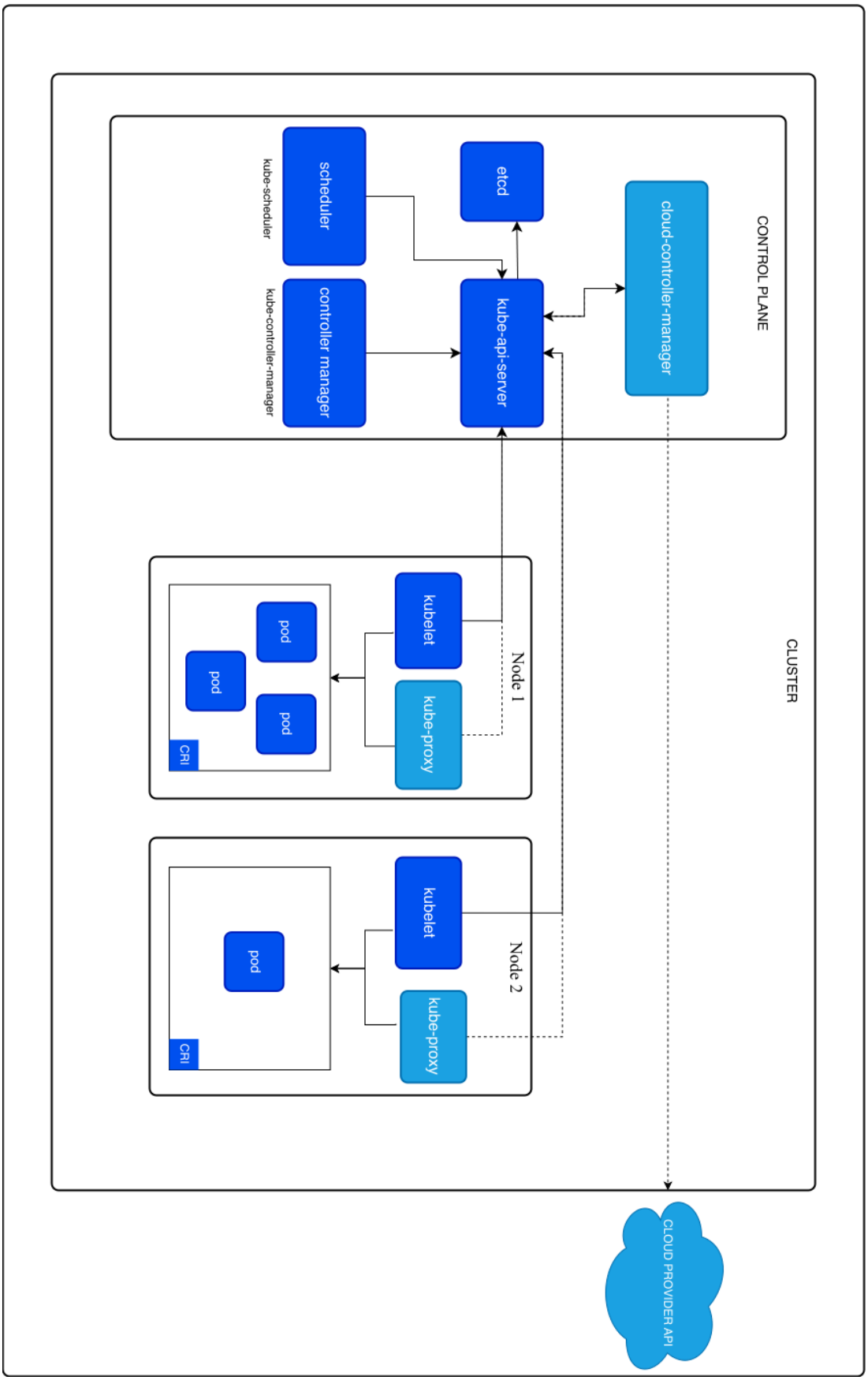
Στοιχεία Node

Τα στοιχεία Node εκτελούνται σε κάθε έναν κόμβο και εκτελούν όλες τις ενέργειες που χρειάζονται για την κανονική λειτουργία του φόρτου [1]. Τα στοιχεία αυτά είναι:

- Kubelet: εκτελείται σε κάθε κόμβο και διασφαλίζει ότι οι φόρτοι εκτελούνται σύμφωνα με την επιθυμητή κατάσταση που ορίζεται από το control plane [18].

- Kube-proxy: διαχειριστής των δικτυακών κανόνων στους κόμβους του cluster, εξασφαλίζοντας την επικοινωνία μεταξύ των Pods και των υπηρεσιών [19].

- Container Runtime: εκτελεί τους Containers και διαχειρίζεται τον κύκλο ζωής τους, όπως το Docker, containerd ή CRI-O [20].



Σχήμα 2.1: *Kubernetes Architecture [1]*

2.1.2 Pods

Το Pod είναι το σύνολο των υπηρεσιών Linux (namespaces, cgroups) με σκοπό την απομόνωση ενός υπολογισμού [21]. Συνιστά τη μικρότερη μονάδα υπολογισμού που διαθέτει ο Kubernetes [22]. Αποτελείται από έναν ή περισσότερους Containters (συνήθως έναν) που διαχειρίζεται κοινούς πόρους (υπολογιστικούς και δικτύου), τόμους και προσφέρει απομόνωση[23] . Τα Pods χρησιμοποιούνται με 2 τρόπους, είτε με έναν Container ή με παραπάνω από έναν [24]. Με έναν Container, που αποτελεί την πιο συνηθισμένη διαμόρφωση, το Pod δρα ως κέλυφος γύρω του προσφέροντας απομόνωση. Η διαμόρφωση με παραπάνω από έναν Container είναι πιο ενδιαφέρουσα καθώς εγγυάται ότι οι Containers θα φιλοξενούνται στον ίδιο κόμβο, μοιραζόμενοι πόρους μεταξύ τους, προσποιώντας την λειτουργία τους στο ίδιο μηχάνημα. Ενδεικτικά, κάποια από τα πρότυπα σχεδιασμού με παραπάνω από έναν Container είναι:

- Στενή συσχέτιση Container μέσω της κοινής φιλοξενίας στον ίδιο κόμβο, όπου μπορούν να μοιράζονται πόρους για την πιο αποδοτική επικοινωνία, λειτουργία και συγχρονισμό τους [24].
- Πρότυπο sidecar, με τον τρόπο αυτό μπορεί να επεκταθεί η λειτουργικότητα ενός Container με την προσθήκη ενός άλλου. Κάποια από τα σημεία επέκτασης είναι η καταγραφή συμβάντων, η παρακολούθηση της λειτουργίας και η διαμεσολάβηση δικτύου [24].
- Βοηθητική λειτουργία, όπου τροποποιούν την διαμόρφωση του κυρίου Container σε διάφορες στιγμές της διάρκειας ζωής του (Container εκκίνησης [25], λήξης)[24].

Είναι σημαντικό να παρατηρηθεί ότι η λειτουργία ενός Pod είναι εφήμερη [22], [26], και με την παύση λειτουργίας, προγραμματισμένης ή αναπάντεχης τότε όλα τα δεδομένα και οι τόμοι του διαγράφονται. Αυτό το χαρακτηριστικό το καθιστά ιδανικό για εφαρμογές REST, αλλά ακατάλληλο για εφαρμογές όπου η διατήρηση δεδομένων και της κατάστασης είναι απαραίτητη (π.χ. βάσεις δεδομένων).

Τα Pods σπάνια εμφανίζονται μόνα τους σε περιβάλλοντα παραγωγής, είναι συνθετικά στοιχεία άλλων πόρων φόρτου εργασίας [27]:

- Το Deployment επιτρέπει την κλιμάκωση και ενημέρωση Pods, διασφαλίζοντας διαθέσιμες και επαναλαμβανόμενες αναπτύξεις και προωθήσεις εφαρμογών [28].
- Το DaemonSet εξασφαλίζει ότι ένα Pod εκτελείται σε κάθε ή σε επιλεγμένους κόμβους του cluster, χρήσιμο για δαίμονες συστήματος και πράκτορες παρακολούθησης [29].
- Το StatefulSet διαχειρίζεται εφαρμογές με διατηρήσιμη ταυτότητα και σταθερούς πόρους, ιδανικό για βάσεις δεδομένων και κατανεμημένα συστήματα. Θα το εξετάσουμε σε επόμενη υποενότητα [30].

2.1.3 Persistent Volumes

Όπως προαναφερθήκαμε, Ενότητα 2.1.2, οι τόμοι, και η κατάσταση, των Pods είναι εφήμερη [22], [26]. Για την αντιμετώπιση αυτής της πρόκλησης, σχεδιάστηκαν οι Μόνιμοι

Τόμοι (Persistent Volumes) [31]. Τα Persistent Volumes αποτελούν βασικό στοιχείο για την διαχείριση αποθηκευτικού χώρου σε περιβάλλοντα του Kubernetes. Αποτελεί πόρο του cluster με τον ίδιο τρόπο που κάθε κόμβος είναι πόρος του cluster. Υπάρχει ανεξάρτητα από τον κύκλο ζωής των Pods ή των κόμβων, διασφαλίζοντας ότι τα δεδομένα παραμένουν ακόμα κι αν η εφαρμογή ή το σύστημα που τα χρησιμοποιούν τερματιστούν ή επανεκκινήσουν [31].

Δεσμεύονται με 2 τρόπους, στατικό ή δυναμικό [32]. Για τη στατική δέσμευση ο διαχειριστής του Kubernetes δημιουργεί και διαχειρίζεται τα PV χειροκίνητα και ο Kubernetes μπορεί να τα χρησιμοποιήσει. Η δυναμική δέσμευση πραγματοποιείται μέσω Storage Classes [33] και PVCs [34] . Ο Kubernetes ανάλογα με την επιθυμητή κατάστασή μπορεί να τα δεσμεύσει, αποδεσμεύσει και να επαναδιαθέσει τους τόμους στους κόμβους και στα Pods ανάλογα με τις προδιαγραφές του συστήματος χωρίς χειροκίνητες ενέργιες του διαχειριστή.

Τα PVs έχουν μεγάλο βαθμό διαμόρφωσης. Πέραν της χωρητικότητας κάποιες από τις κατηγορίες διαμόρφωσης τους είναι:

Τρόπος λειτουργίας

Ο διαχειριστής επιλέγει μεταξύ τρόπο λειτουργίας συστήματος αρχείων ή block [35].

Τρόπος πρόσβασης

- ReadWriteOnce, ο τόμος τοποθετείτε σε έναν μόνο κόμβο και Pods μόνο από αυτό τον κόμβο μπορούν να γράψουν, αναγνώσουν δεδομένα.
- ReadOnlyMany, ο τόμος μπορεί να αναγνωστεί από πολλούς κόμβους.
- ReadWriteMany, ο τόμος μπορεί να γραφτεί και να αναγνωστεί από πολλούς κόμβους.
- ReadWriteOncePod, ο τόμος τοποθετείτε σε έναν μόνο κόμβο και μόνο ένα Pod μπορεί να γράψει και να αναγνώσει δεδομένα. Αυτή η ρύθμιση είναι σε δοκιμαστικό στάδιο στην έκδοση 1.27.

[36]

Storage Class

Η Storage Class προσφέρει στους διαχειριστές την δυνατότητα να περιγράφουν τις ομάδες αποθήκευσης τόμων στο σύστημα [37], [33]. Ο διαχειριστής διαμορφώνει ποιος θα είναι ο διαθέτης των PV, μπορεί να επιλεγεί AzureFile, CephFS, τοπικός, NFS, RDB, κ.α. [38]. Παράλληλα ρυθμίζει την πολιτική ανάκτησης τόμων μετά το τέλος χρήσης τους, διαγραφή ή διατήρηση [39]. Μία πολύ σημαντική διαμόρφωση είναι αυτή του τρόπου σύνδεσης του τόμου που μπορεί να είναι είτε άμεση ή αναμονή για την πρώτη κατανάλωση [40]. Η άμεση συνδέει τον τόμο κατά την δημιουργία του PVC, σε περιβάλλοντα με τοπολογικούς περιορισμούς όπου αυτό οδηγεί στην σύνδεση PV πριν την γνώση σε ποιον κόμβο δρομολογείται το Pod - γεγονός που μπορεί να οδηγήσει σε αδρομολόγητο Pod που δεν έχει πρόσβαση στο PV. Αυτό το πρόβλημα αντιμετωπίζεται με την δεύτερη διαμόρφωση που περιμένει την δημιουργία του Pod για την σύνδεση του με PV. Τέλος, μπορούν να οριστούν επιτρεπτές διαμορφώσεις τοπολογίας [41].

Πολιτική Ανάκτησης PV

Όταν το Pod δεν χρειάζεται το PV η πολιτική ανάκτησης καθορίζει τι θα συμβεί με το τόμο, αν δηλαδή θα διατηρηθεί (Retain) ή θα διαγραφεί (Delete). Παλαιότερα υπήρχε η επιλογή να ανακυκλωθεί (Recycle) ο τόμος άλλα έχει αφαιρεθεί αυτή η δυνατότητα [42].

Συγγένεια κόμβων

Το PV μπορεί να θέσει περιορισμούς σε ποιον κόμβο τοποθετηθείτε. Τα Pods που χρησιμοποιούν τον συγκεκριμένο PV μπορούν να δρομολογηθούν μόνο στον συγκεκριμένο κόμβο [43].

2.1.4 Deployments

Τα Deployments διαχειρίζονται ένα σύνολο Pods εκτελώντας έναν φόρτο εργασίας χωρίς κατάσταση. Χρησιμοποιούνται συνήθως για την κυλιόμενη ενημέρωση και διάθεση νέων εκδόσεων των Pods. Το Deployment δημιουργεί ένα σύνολο αντιγράφων και αυτό στη συνέχεια έναν αριθμό από Pod τα οποία είναι πιστά αντίγραφα μεταξύ τους. Κάθε ένα αντίγραφο επιτελεί την ίδια ακριβώς λειτουργία στον cluster και δεν υπάρχει καμία σημειολογική διαφορά μεταξύ τους [28].

2.1.5 StatefulSets

Τα StatefulSets είναι τα στοιχεία που δίνουν την δυνατότητα στον χρήστη να διατηρήσει την κατάσταση Pods στον Kubernetes [30]. Διαχειρίζονται την προώθηση και την κλιμάκωση για έναν σύνολο Pods. Αυτά τα Pods έχουν τις ίδιες προδιαγραφές αλλά το σημείο που διαφοροποιεί το Statefulset από ένα Deployment είναι ότι τα Pods φέρουν μια διατηρούμενη ταυτότητα, δηλαδή τα Pods μπορεί να είναι ανάλογα μεταξύ τους [44]. Ωστόσο, έχει μοναδικό όνομα με δείκτη και αναγνωριστικό δικτύου πχ για το Statefulset my-app το πρώτο αντίγραφο είναι το my-app-0 [44], [45]. Κάθε αντίγραφο δημιουργείται με αύξουσα σειρά από το χαμηλότερο στο μεγαλύτερο και αντιστρόφως κατά την αποκλιμάκωση με φθίνουσα σειρά.

Τα Statefulset έρχονται όμως και με περιορισμούς. Για την διατήρηση της κατάστασης και των δεδομένων χρειάζεται να γίνει PVC. Η κλιμάκωσή δεν διαγράφει αυτόματα τα PVC. Χρειάζεται η χειροκίνητη δημιουργία ακέφαλης υπηρεσίας για τις ταυτότητες δικτύου των Pods. Τέλος η διαγραφή του δεν εγγυάται την διαγραφή όλων των Pods, προτείνεται λοιπόν η αποκλιμάκωση σε μηδέν αντίγραφα και μετά η διαγραφή του [46].

Για την διαμόρφωση του Statefulset πέραν από την ρύθμιση του προτύπου του Pod, πρέπει να οριστεί ο αριθμός των αντιγράφων, η ακέφαλη υπηρεσία δικτύου, το πρότυπο που ορίζει ποια Storage Class θα χρησιμοποιηθεί για την PVC καθώς και η πολιτική διατήρησης PVC [30].

Αυτή η πολιτική ορίζει τι γίνεται με τις PVC όταν διαγράφονται και αποκλιμακώνονται τα Pods του Statefulset. Μπορούν είτε να διαγραφούν ή να διατηρηθούν [47].

2.1.6 Κλιμάκωση

Κλιμάκωση είναι η διαδικασία κατά την οποία αλλάζουν οι υπολογιστικές ικανότητες ενός συστήματος. Αυτό μπορεί να επιτευχθεί είτε με την οριζόντια ή την κατακόρυφη κλιμάκωση του συστήματος [48]. Οριζόντια κλιμάκωση είναι όταν προστίθεται υπολογιστική δύναμη σε ένα σύστημα με την προσθήκη υπολογιστών. Αυτή είναι διαφορετική από την κατακόρυφη κλιμάκωση που αυξάνονται οι υπολογιστικές ικανότητες του συστήματος με την αύξηση της μνήμης, επεξεργαστικής ισχύος και αποθηκευτικής ικανότητας χωρίς την προσθήκη νέων υπολογιστών [49]. Αυτές οι δύο διαφορετικές στρατηγικές έχουν διαφορετικά πλεονεκτήματα και μειονεκτήματα και ο συμψηφισμός τους έρχεται στις ανάγκες του συστήματος [50].

Η κατακόρυφη κλιμάκωση επιλέγεται για:

- Λιγότερο σημαντικά συστήματα και φόρτους εργασίας.
- Συστήματα που δεν αναμένεται να αυξηθεί ο φόρτος τους στο μέλλον.
- Μείωση του αρχικού κόστους.
- Συστήματα που δεν χρειάζονται υψηλή διαθεσιμότητα, απόδοση και διάθεση σε διαφορετικές περιοχές.

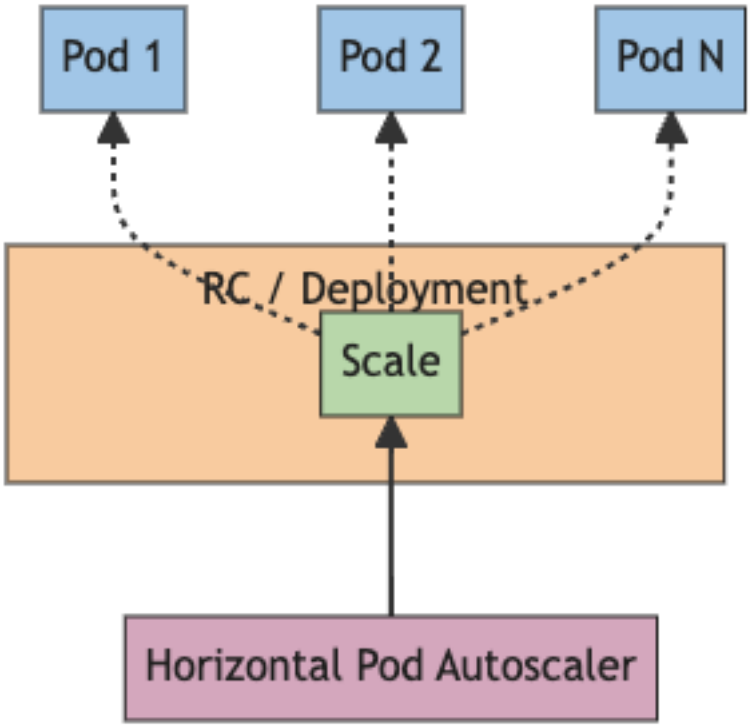
Η οριζόντια κλιμάκωση επιλέγεται για:

- Σημαντικά συστήματα και φόρτους εργασίας.
- Συστήματα που αναμένεται να αυξηθεί ο φόρτος τους στο μέλλον.
- Συστήματα που χρειάζονται υψηλή διαθεσιμότητα, απόδοση και διάθεση σε διαφορετικές περιοχές.

[51]

2.1.7 Horizontal Pod Autoscaler

Ο Kubernetes παρέχει αντικείμενα για οριζόντια [52] και κατακόρυφη αυτόματη κλιμάκωση[53]. Παρόλα αυτά θα εστιάσουμε στον Αυτόματο Οριζόντιο Κλιμακωτή (Horizontal Pod Autoscaler). Το HPA είναι το στοιχείο του Kubernetes που επιτρέπει σε έναν φόρτο εργασίας να κλιμακώνεται αυτόματα ανάλογα με την ζήτηση. Μπορεί να συνδεθεί μόνο με αντικείμενα που κλιμακώνονται (Deployments και Statefulsets) [52].



Σχήμα 2.2: *HPA controlling Deployment [2]*

Κατά την διαμόρφωση του HPA επιλέγεται το αντικείμενο που συνδέεται μαζί του, ορίζεται ο μεγαλύτερος και μικρότερος επιτρεπτός αριθμός αντιγράφων, ποια μετρική παρακολουθείται και ποια είναι η επιθυμητή συμπεριφορά κλιμάκωσης και αποκλιμάκωσης. Δεν επιτρέπεται το ίδιο αντικείμενο να συνδεθεί με παραπάνω από ένα HPA [52].

Ο τρόπος που δουλεύει το HPA είναι να παρακολουθεί τις ορισμένες μετρικές τακτικά (προεπιλεγμένη τιμή δεκαπέντε δευτερόλεπτα) και ανάλογα με την χρήση της μετρικής προβαίνει σε προκαθορισμένες ενέργιες [2]. Ο αλγοριθμικός τύπος που ακολουθεί είναι $desiredReplicas = (currentReplicas * currentMetricValue/desiredMetricValue)$ [2].

Οι μετρικές που μπορούν να χρησιμοποιηθούν είναι οι μετρικές χρήσης του Pod (χρήση επεξεργαστή, μνήμης) [54] ή προσαρμοσμένες μετρικές (αιτήματα ανά δευτερόλεπτο, ρυθμός σφαλμάτων, ενεργές συνδέσεις, κ.ο.κ.) [55]. Επίσης μπορεί να γίνει συνδυασμός μετρικών μεταξύ τους στο ίδιο HPA [56]. Είναι σημαντικό να παρατηρηθεί ότι το HPA παίρνει τον μέσο όρο χρήσης από όλα τα αντίγραφα, οπότε υπάρχει η περίπτωση για ένα Pod να έχουμε υπερβεί το όριο χρήσης αλλά να μην κλιμακώνεται το αντικείμενο καθότι η χρήση σε άλλα Pods να είναι χαμηλότερη [54].

Η συμπεριφορά ορίζει τον τρόπο κλιμάκωσης και αποκλιμάκωσης του συστήματος, ορίζοντας πολιτικές με επιτρεπτές ενέργειες και την περίοδο που κάθε πολιτική ισχύει [57]. Όταν υπάρχουν παραπάνω από μία πολιτικές επιλέγεται αυτή με τον μεγαλύτερο αριθμό αλλαγής [57]. Η λογική του τύπου 2.1.7 σε συνδυασμό με τις ενέργειες [57] επιτρέπει την δυαδική λειτουργία του συστήματος όπου το αριθμός αντιγράφων μπορεί να ταλαντώνεται ανάμεσα σε κάποιες τιμές. Αυτό μπορεί να περιοριστεί με την χρήση του παραθύρου σταθεροποίησης [58] που δεν επιτρέπει στο σύστημα να πράξει κάποια ενέργεια αντίθετη με την

επιλογή που έγινε στην προκαθορισμένη περίοδο.

Το HPA έχει περιορισμούς, σύμφωνα με τους Jiang και Wu [59]:

- Ο αλγόριθμός κλιμάκωσης και αποκλιμάκωσης είναι πολύ απλός και ανελαστικός.
- Το παράθυρο σταθεροποίησης [58] δεν επιτρέπει την περαιτέρω κλιμάκωση του συστήματος.
- Η περίοδος συλλογής δεδομένων δεν μπορεί να αλλάξει δυναμικά ανάμεσα σε περισσότερο ή λιγοτερο απαιτητικές περιόδους φόρτου.

Ακόμα δεν μπορεί να οριστεί περιοχή υστέρησης, όπου το σύστημα δεν κλιμακώνει ή αποκλιμακώνει τον αριθμό Pods.

2.2 Apache Cassandra

Η Apache Cassandra αποτελεί ένα κατανεμημένο, μη σχεσιακό σύστημα διαχείρισης βάσεων δεδομένων. Αναπτύχθηκε ως σύστημα ανοιχτού κώδικα από τους Lakshman και Malik [11] [60] για την Αναζήτηση Μηνυμάτων του Facebook το 2008 και το 2010 προωθήθηκε από το Ίδρυμα Λογισμικού Apache. Σχεδιάστηκε με σκοπό την διαχείριση μεγάλου όγκου δεδομένων με γνώμονα την διαθεσιμότητα και επεκτασιμότητα του συστήματος έναντι της συνέπειας των δεδομένων. Τα κύρια χαρακτηριστικά του είναι τα παρακάτω ([60] και Χρυσόπουλος [61]):

- Πολλαπλή μίσθωση κόμβων σε διαφορετικά κέντρα δεδομένων.
- Αποκεντρωμένη λειτουργία: όλοι οι κόμβοι είναι όμοιοι, χωρίς να υπάρχει κάποιος κύριος κόμβος, εκτελούν όλοι τις ίδιες λειτουργίες επικοινωνώντας μέσω Peer-to-Peer πρωτοκόλλου.
- Ανεκτικότητα σε σφάλματα: τα δεδομένα αναπαράγονται σε πολλούς κόμβους και διαφορετικά κέντρα δεδομένων, εξαλείφοντας τα μεμονωμένα σημεία αστοχίας.
- Ελαστικότητα και γραμμική επεκτασιμότητα: η προσθήκη νέων κόμβων δεν διακόπτει την λειτουργία του συστήματος και αυξαίνει γραμμικά την απόδοση του.
- Διαμορφώσιμη συνέπεια: το σύστημα εγγυάται τελική συνέπεια και κάθε συναλλαγή μπορεί να έχει διαφορετική συνέπεια. Αυτή η ρύθμιση επηρεάζει πόσοι κόμβοι πρέπει να αναγνωρίσουν επιτυχώς μία συναλλαγή.

2.2.1 Μοντέλο Δεδομένων

Το Μοντέλο Δεδομένων της Apache Cassandra είναι μη σχεσιακό, απαρτίζεται από όρους και έννοιες που θυμίζουν παραδοσιακά συστήματα διαχείρισης βάσεων δεδομένων (RDBMS) [62].

Keyspaces

Το Keyspace δηλώνει τον τρόπο με τον οποίο δεδομένα αναπαράγονται, σε πόσα αντίγραφα. Είναι ουσιαστικά ένα σύνολο από πίνακες.

Πίνακες

Ο Πίνακας αποτελείται από στήλες και σειρές. Οι στήλες περιγράφουν το σχήμα των δεδομένων και τι τύπους έχουν. Οι στήλες μπορούν να επεκταθούν με μηδενική διακοπή λειτουργίας [62].

Κατάτμηση

Η Κατάτμηση ορίζει το διάστημα των σειρών του κλειδιού κατάτμησης που ανήκουν σε κάθε κόμβο [62].

Σειρά

Η Σειρά αποτελεί μια πλειάδα από δεδομένα που αναγνωρίζεται από ένα πρωτεύον κλειδί. Το πρωτεύον κλειδί συνθέτεται από το κλειδί κατάτμησης και προαιρετικά άλλες στήλες [62].

Στήλη

Η στήλη είναι ένα κομμάτι δεδομένων με ένα τύπο, η μικρότερη έννοια δεδομένων [62].

CQL

Η Apache Cassandra χρησιμοποιεί την γλώσσα ερωτημάτων Cassandra (CQL), εμπνευσμένη από την γλώσσα δομημένων ερωτημάτων (SQL). Παρατηρούμε ότι κάποιες από τις δομές της Apache Cassandra αντιστοιχούν σε αυτές ενός παραδοσιακού RDBMS οπότε δεν είναι ξένη η CQL [62], [63].

Αποκανονικοποίηση

Στην Apache Cassandra τα δεδομένα είναι αποκανονικοποιημένα, το οποίο έρχεται σε αντίθεση με τα RDBMS και δεν επιτρέπει ενέργειες Join καθώς τα συσχετιζόμενα δεδομένα βρίσκονται σε έναν πίνακα [62].

2.2.2 Amazon Dynamo

Η λειτουργία της Apache Cassandra βασίζεται πάνω στο κατανεμημένο σύστημα κλειδιού-τιμής Amazon Dynamo των DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall και Vogels [64], [65]. Τα κύρια στοιχεία του είναι:

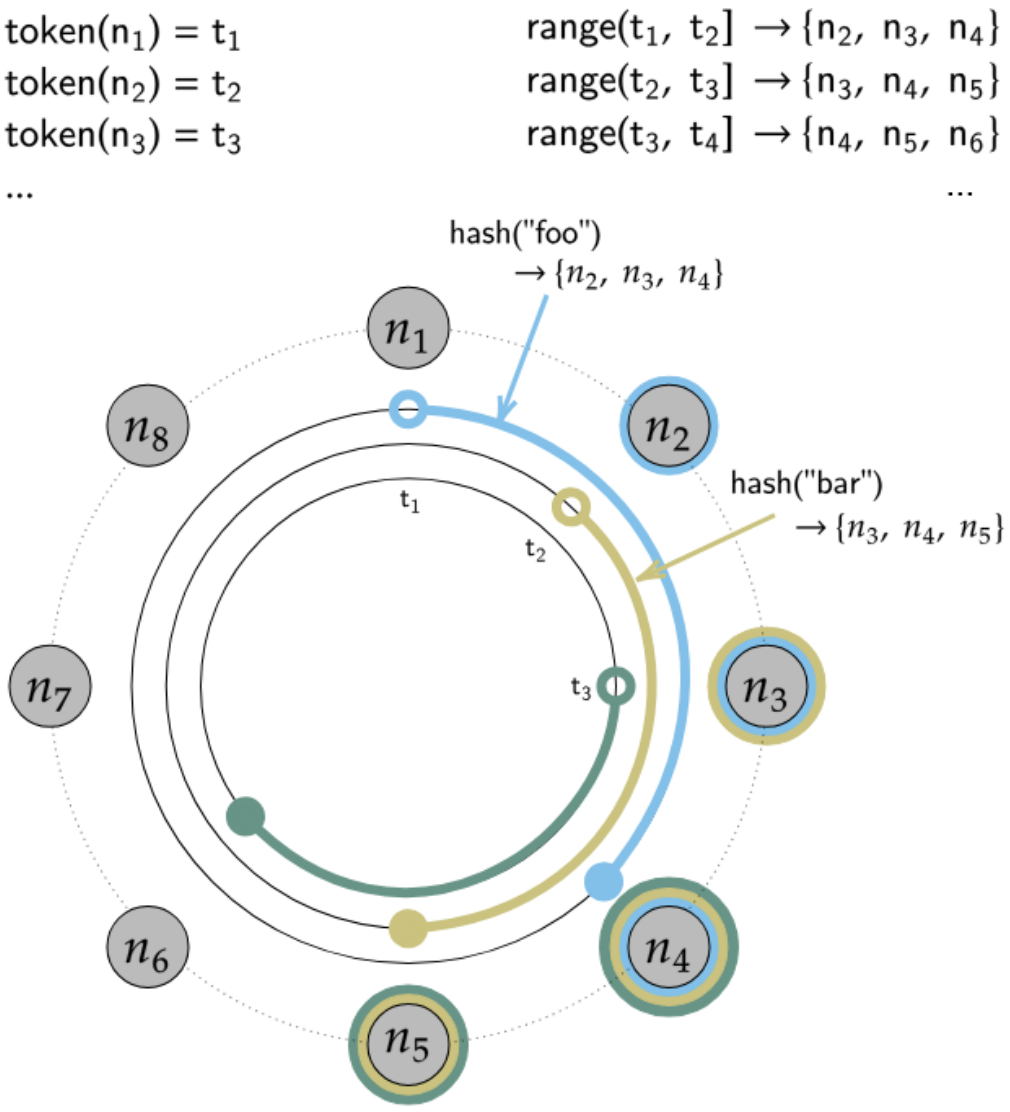
- Συντονισμός αιτημάτων σε ένα κατανεμημένο σύνολο δεδομένων
- Πολιτική Αντιγράφων
- Ρυθμιζόμενη Συνέπεια

- Συμμετοχή σε δακτύλιο και ανίχνευση σφαλμάτων

[64], [65]

Συντονισμός αιτημάτων σε ένα κατανεμημένο σύνολο δεδομένων

Η Apache Cassandra για την επίτευξη οριζόντιας κλιμάκωσης κατανέμει τα δεδομένα σε κόμβους βάσει συνάρτησης συνεπούς κατακερματισμού [66]. Δεδομένα κατανέμονται σε έναν συνεχή δακτύλιο βάσει του κλειδιού κατάτμησης τους. Οι κόμβοι, μάρκες, ανήκουν στον δακτύλιο και αποθηκεύονται σε αυτούς ένα εύρος κλειδιών μετά από την μάρκα τους [3]. Ανάλογα με τον αριθμό αντιγράφων των δεδομένων τότε αυτά αναπαράγονται και σε επόμενους κόμβους για να εξασφαλιστεί λειτουργία ανθεκτική σε σφάλματα. Στο Σχήμα 2.3, βλέπουμε τον node4 να περιέχει τα εύρος κλειδιών, t1, t2, t3 και t4. Αν τερματιστεί αναπάντεχα βλέπουμε ότι τα δεδομένα του υπάρχουν σε άλλους κόμβους οπότε το σύστημα μπορεί να λειτουργήσει χωρίς απώλια δεδομένων.



Σχήμα 2.3: *Cassandra Token Ring [3]*

Η εξισορρόπηση κατανομής των δεδομένων δουλεύει επιτυχώς όταν έχουμε αρκετούς κόμβους καθώς αυτοί κατανέμονται ομοιόμορφα στον δακτύλιο [3]. Αυτό όμως μπορεί να μην συμβαίνει για λιγότερους κόμβους. Το Amazon Dynamo αντιμετωπίζει αυτό το πρόβλημα με την δημιουργία εικονικών κόμβων με μάρκες στον δακτύλιο που κατανέμονται πιο ομοιόμορφα σε αυτόν [67].

Πολιτική Αντιγράφων

Κάθε Keyspace ορίζει την πολιτική αντιγράφων του, καθορίζοντας έτσι τον τρόπο με τον οποίο ορίζονται αντίγραφα των δεδομένων. Δυνητικά μπορεί να είναι η NetworkTopologyStrategy, που καθορίζει πως αντιγράφονται τα δεδομένα σε διαφορετικά υπολογιστικά κέντρα, ή η SimpleStrategy που απλά περιγράφει πόσοι κόμβοι έχουν αντίγραφο των δεδομένων. Συνίσταται σε περιβάλλοντα παραγωγής να χρησιμοποιείται η NetworkTopologyStrategy και σε περιβάλλοντα ανάπτυξης η SimpleStrategy [68].

Ρυθμιζόμενη Συνέπεια

Το θεώρημα CAP του Brewer [69], είναι ένα από σημαντικά θεωρήματα στον κόσμο των κατανεμημένων συστημάτων, σύμφωνα με τους Gilbert και Lynch [70]. Αναγνωρίζει τρία χαρακτηριστικά που διέπουν κάθε τέτοιο σύστημα [69], [70], [71]:

- Συνέπεια: κάθε αίτημα ανάγνωσης σε κάθε κόμβο επιστρέφει την πιο πρόσφατη εγγραφή ή αποτυγχάνει.
- Διαθεσιμότητα: κάθε αίτημα λαμβάνει μια επιτυχή απάντηση, χωρίς αυτή να είναι αναγκαστικά η πιο πρόσφατη εγγραφή στο σύστημα.
- Ανοχή Διαχωρισμού: η ικανότητα του συστήματος να λειτουργεί ακόμα και αν υπάρχει διακοπή στην επικοινωνία των κόμβων.

Το Θεώρημα [69], [70], [71] μας λέει ότι ένα κατανεμημένο σύστημα δεν μπορεί να έχει και τα τρία χαρακτηριστικά, αλλά μόνο δύο. Αυτό σημαίνει ότι ένα σύστημα με Ανοχή Διαχωρισμού κατά την διάρκεια ενός διαχωρισμού δικτύου δεν μπορεί να εγγυηθεί ταυτόχρονα Συνέπεια και Διαθεσιμότητα καθώς μια συναλλαγή θα επιστρέψει την πιο πρόσφατη εγγραφή ή θα αποτύχει και ταυτόχρονα θα πρέπει να λάβει απάντηση χωρίς αυτή να είναι αποτυχία.

Η Apache Cassandra προσφέρει τις ακόλουθες εγγυήσεις [72]:

- Υψηλή Κλιμακωσιμότητα [73]: το σύστημα μπορεί να προσθέτει ή αφαιρεί κόμβος χρησιμοποιώντας το πρωτόκολλο Διασποράς για την αποδοτική αναγνώριση των μελών του.
- Υψηλή Διαθεσιμότητα: [74] χρησιμοποιεί ένα ανθεκτικό σε σφάλματα σύστημα αποθήκευσης.
- Ανθεκτικότητα [75]: αντίγραφα των δεδομένων υπάρχουν σε διαφορετικούς κόμβους ή και υπολογιστικά κέντρα που εγγυώνται ότι αν χαθεί ένας κόμβος για οποιονδήποτε λόγο τότε θα υπάρχουν αντίγραφα των δεδομένων.

- Τελική Συνέπεια [76]: οι εγγραφές θα υπάρχουν σε όλα τα αντίγραφα τελικά. Διαφορετικές εκδόσεις των δεδομένων θα υπάρχουν σε διαφορετικούς κόμβος μέχρι τελικά να έχουμε συνεπή κατάσταση.

- Ελαφριές συναλλαγές [77]: εκτελούνται σειριακά χρησιμοποιώντας το πρωτόκολλο ομοφωνίας Paxos, με αυτόν τον τρόπο εξασφαλίζεται συνέπεια στα αντίγραφα λόγω της συναλλαγής Compare and Set επιτρέποντας τις αναγνώσεις να βλέπουν τιμές δεδομένων χωρίς να πραγματοποιούνται αλλαγές.

- Ομαδικές εγγραφές σε πολλούς πίνακες [78]: είτε θα επιτύχουν όλες πλήρως ή καθόλου.

- Δευτερεύοντα ευρετήρια που επιτρέπουν την αναζήτηση στηλών είναι εγγυημένο να είναι συνεπή στα τοπικά αντίγραφα [79].

Η Apache Cassandra διαθέτει ρυθμιζόμενη συνέπεια για κάθε συναλλαγή [76]. Ο χρήστης μπορεί να επιλέξει τον βαθμό συνέπειας. Μεγαλύτερος βαθμός συνέπειας επηρεάζει αρνητικά την Διαθεσιμότητα καθώς το σύστημα περιμένει περισσότερα αντίγραφα να επιβεβαιώσουν μια συναλλαγή [80]. Παρατίθενται οι ρυθμίσεις συνέπειας [80], [81], [82]:

- Ένα μόνο αντίγραφο

- Δύο αντίγραφα

- Τρία αντίγραφα

- Απαρτία, η πλειοψηφία των αντιγράφων

- Όλα τα αντίγραφα

- Τοπική απαρτία, η πλειοψηφία των αντιγράφων μόνο ενός κέντρου υπολογιστών (στο κέντρο υπολογιστών που έγινε το αίτημα)

- Κάθε απαρτία, η πλειοψηφία για κάθε υπολογιστικό κέντρο

- Τοπικό ένα αντίγραφο, αυτή η διαμόρφωση εγγυάται ότι το αίτημα ανάγνωσης δεν αποστέλλεται σε άλλο υπολογιστικό κέντρο

- Οποιαδήποτε, ένα μόνο αντίγραφο απαντά ή ο κόμβος διαμεσολάβησης αποθηκεύει μία υπόδειξη. Ο διαμεσολαβητής θα προσπαθήσει να μεταφέρει την μετάλλαξη στα αντίγραφα αργότερα. Η διαμόρφωση αυτού του βαθμού συνέπειας είναι διαθέσιμο μόνο για εγγραφές.

Συμμετοχή σε δακτύλιο και ανίχνευση σφαλμάτων

Για την Κατάτμηση Δεδομένων και τη Πολιτική αντιγράφων το σύστημα χρειάζεται να γνωρίζει την υγεία των κόμβων ώστε να δρομολογούνται τα αιτήματα γραφής και ανάγνωσης αποδοτικά [83]. Αυτές οι πληροφορίες μοιράζονται με κατανεμημένο τρόπο στην Apache Cassandra μέσω των επόμενων δυνατοτήτων· Πρωτόκολλο Διασποράς και Αναγνώρισης Σφαλμάτων.

Πρωτόκολλο Διασποράς [84] είναι ένα σύστημα όπου οι κόμβοι μοιράζονται πληροφορίες σχετικά με την δικιά τους υγεία αλλά και όσων κόμβων γνωρίζουν. Χρησιμοποιούν πλειάδες που φέρουν την μονοτονική ώρα και την έκδοση της πληροφορίας επιτρέποντας στους κόμβους να αξιοποιούν την πιο πρόσφατη έκδοση της διασποράς πληροφοριών και αναγνώριση σφαλμάτων. Κάθε κόμβος περιοδικά:

1. ενημερώνει την κατάσταση υγείας του και δημιουργεί την τοπική όψη της διασποράς του cluster που γνωρίζει.

2. Επιλέγει έναν τυχαίο κόμβο και ανταλλάσει πληροφορίες διασποράς.

3. Προσπαθεί πιθανοκρατικά να ανταλλάξει πληροφορίες με έναν κόμβο που δεν έχει την κατάσταση σου (αν υπάρχει).

4. Διασπείρει πληροφορίες με τον κόμβο εκκίνησης αν δεν επικοινώνησε μαζί του στο δεύτερο βήμα.

[84]

Ο πρώτος κόμβος αποτελεί τον κόμβο εκκίνησης και η μόνη διαφορά του από τους άλλους κόμβους είναι ότι δεν χρειάζονται άλλους κόμβους για να συνδεθούν στον δακτύλιο. Λόγω του βήματος 4 αυτοί οι κόμβοι έχουν υψηλή χρήση στην διαδικασία της διασποράς.

Η Apache Cassandra για την αναγνώριση σφαλμάτων [83] χρησιμοποιεί μία παραλλαγή του αλγορίθμου Phi Accrual Failure Detector των D?fago, Hayashibara, Yared, Rami και Katayama [85], όπου κάθε κόμβος αποφασίζει ότι κάποιος άλλος κόμβος είναι κάτω αν δεν λάβει υγιή κατάσταση από αυτόν για συγκεκριμένο χρονικό διάστημα. Τότε δεν θα κατευθύνει τις συναλλαγές ανάγνωσης σε αυτόν τον κόμβο και θα περιμένει να αποστείλει τις εγγραφές όταν λάβει υγιή κατάσταση από αυτό.

2.2.3 Μηχανή αποθήκευσης

Η Apache Cassandra χρησιμοποιεί μία αποκεντρωμένη μηχανή αποθήκευσης η οποία της δίνει την δυνατότητα για γρήγορες εγγραφές και αναγνώσεις [86]. Υπάρχουν δύο κύρια μονοπάτια συναλλαγών· εγγραφής και ανάγνωσης.

Μονοπάτι Εγγραφής

Κατά την εγγραφή [87], κάθε εισερχόμενη συναλλαγή σε κάθε κόμβο εγγράφεται πρώτα σε ένα Commitlog, ένα αρχείο καταγραφής δεσμεύσεων που επιτρέπει μόνο την προσθήκη νέων εγγραφών. Οι εγγραφές αποθηκεύονται σε κομμάτια γεγονός που επιτρέπει την αποδοτική εγγραφή στον δίσκο. Η εγγραφή επιτρέπει στον κόμβο να κρατάει τα δεδομένα μετά από αναπάντεχες απώλειες λειτουργίας [88]. Στη συνέχεια, τα δεδομένα τοποθετούνται στο Memtable, πίνακα που υπάρχει στην μνήμη (σωρό) υλοποιημένο με ταξινομημένη δομή δεδομένων (ισορροπημένο δένδρο ή λίστα παράκαμψης) [89]. Ο κόμβος στέλνει επιβεβαίωση της συναλλαγής όταν αυτή εγγραφεί στο Commitlog και στο Memtable [87]. Όταν το Memtable ξεπεράσει το προκαθορισμένο μέγεθος αποθηκεύεται στον δίσκο σε αρχεία SSTable, αμετάβλητα, αποθηκευμένα ταξινομημένα βάση του κλειδιού κατάτμησης και του

κλειδιού ομαδοποίησης. Υπάρχει η δυνατότητα μεταφοράς SSTable από άλλον κόμβο κατά την εξισορρόπηση των δεδομένων [90]. Η Apache Cassandra συμπιέζει SSTables για την πιο αποδοτική λειτουργία του συστήματος [87].

Μονοπάτι Ανάγνωσης

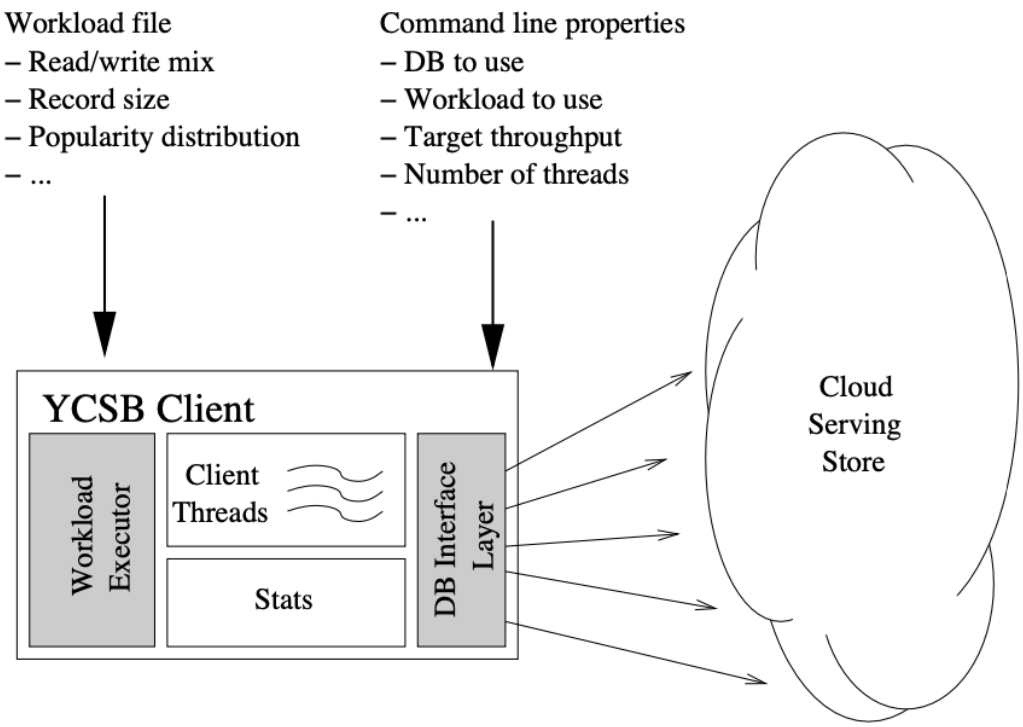
Κατά την ανάγνωση [91], κάθε εισερχόμενη συναλλαγή σε κάθε κόμβο, αναγιγνώσκεται το Memtable και αν αυτή η ανάγνωση δεν επιφέρει αποτέλεσμα τότε διαβάζονται τα SSTables από τον δίσκο. Επειδή τα αρχεία είναι ταξινομημένα, η Apache Cassandra μπορεί αποδοτικά να εντοπίσει τα δεδομένα χρησιμοποιώντας τα κλειδιά κατάτμησης, ομαδοποίησης και φίλτρα Bloom [91], [90]. Τέλος, συγχωνεύει τα δεδομένα και επιστρέφει την νεότερη τιμή.

Βλέπουμε ότι η μηχανή αποθήκευσης δεδομένων της Apache Cassandra προσφέρει υψηλή απόδοση εγγραφής και ανάγνωσης, ανθεκτικότητα σε σφάλματα και μεγάλο βαθμό ρυθμισιμότητας, καθώς κάθε προαναφερθέν στοιχείο μπορεί να ρυθμιστεί [11], [86].

2.3 YCSB

Το YCSB είναι ένα πρόγραμμα αξιολόγησης επιδόσεων βάσεων δεδομένων ανοιχτού κώδικα. Αναπτύχθηκε από τους Cooper, Silberstein, Tam, Ramakrishnan και Sears στην Yahoo [4] το 2010 με σκοπό την αξιολόγηση νέων διαφορετικών βάσεων δεδομένων που αναπτύχθηκαν εκείνα τα χρόνια, συνήθως όχι RDBMSs. Τα τρία στοιχεία που εξετάζει το YCSB είναι η Ελαστικότητα (η δυνατότητα του συστήματος να μεταφέρει το φόρτο σε νέους πόρους κατά την λειτουργία του), η Υψηλή Διαθεσιμότητα (η ικανότητα του να μπορεί να λειτουργεί ακόμα και διάμεσο σφαλμάτων), και η Οριζόντια Κλιμάκωση (η ικανότητα του να διαχειρίζεται μεγάλο όγκο δεδομένων με την προσθήκη τυποποιημένων server και όχι μέσω της κατακόρυφης κλιμάκωσης)[4].

Το YCSB είναι ένα πρόγραμμα Java [92] που δέχεται δύο ειδών διαμορφώσεις, του φόρτου εργασίας και της διεπαφής βάσης δεδομένων. Ο Client του δημιουργεί ClientThreads, βλ. Σχήμα 2.4, τα οποία ορίζουν το φόρτο εργασίας του προγράμματος αξιολόγησης και στην συνέχεια η διεπαφή ορίζει ποιες συναλλαγές θα πραγματοποιηθούν πάνω στην βάση καθώς υπάρχουν διαφορετικές συναλλαγές μεταξύ τόσων διαφορετικών βάσεων. Τέλος εκτελούνται οι συναλλαγές με τον καθορισμένο τρόπο πάνω στην βάση δεδομένων [93].



Σχήμα 2.4: *YCSB Architecture [4]*

Οι διαμορφώσεις φόρτου εργασίας αφορούν [93]: .

- Αριθμός νημάτων επεξεργαστή
- Στόχο ρυθμού μετάδοσης των συναλλαγών
- Η αναλογία του είδους συναλλαγών που εκτελούνται (εγγραφής, ανάγνωσης, προσάρμοσης, σάρωσης)
- Επιθυμητή συνέπεια ανά είδος συναλλαγών
- Αριθμός σειρών στην βάση δεδομένων
- Αριθμός συναλλαγών
- Κατανομή επιλογής σειρών για τις συναλλαγές

Οι διαμορφώσεις διεπαφής βάσης δεδομένων αφορούν τον τρόπο που συνδέεται το YCSB με την βάση καθώς και εξειδικευμένη διαμόρφωση βάσης δεδομένων [93].

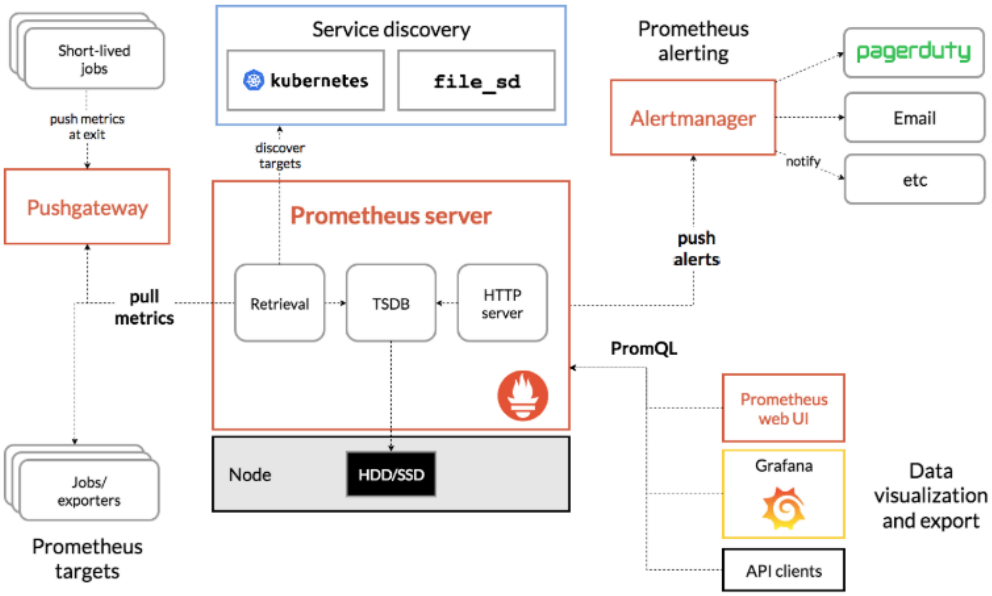
2.4 Prometheus

Ο Prometheus είναι ένα σύστημα παρακολούθησης και ειδοποίησης ανοιχτού κώδικα που αναπτύχθηκε από την Soundcloud το 2012 [5]. Το 2016 αποτέλεσε το δεύτερο πρόγραμμα που φιλοξενείτε από το CNCF, μετά τον Kubernetes καθώς υιοθετήθηκε από

μεγάλο αριθμό εταιριών και οργανισμών. Συλλέγει και αποθηκεύει μετρικές ως δεδομένων χρονοσειράς, δηλαδή δεδομένα μαζί με την χρονική στιγμή καταγραφής τους. Τα κύρια χαρακτηριστικά του είναι:

- Πολυδιάστατο μοντέλο δεδομένων με δεδομένα χρονοσειράς
- Μια εύχρηστη γλώσσα ερωτημάτων (PromQL)
- Η συλλογή των δεδομένων ακολουθεί το μοντέλο έλξης δεδομένων μέσω του πρωτοκόλλου HTTP, επιτρέποντας παρακολούθηση χωρίς την τροποποίηση των εφαρμογών.
- Τα δεδομένα κάθε Prometheus server αποθηκεύονται τοπικά σε αυτό και δεν κατανέμονται σε άλλους κόμβους. Αυτό επιτρέπει στον Prometheus να μην χρειάζεται να συντονίζει συναλλαγές, απλοποιώντας την αρχιτεκτονική.

[5]



Σχήμα 2.5: *Prometheus Architecture [5]*

Τα στοιχεία που συνθέτουν τον Prometheus είναι, βλ Σχήμα 8.2:

- Ο κύριος Prometheus server που συλλέγει δεδομένα και τα αποθηκεύει στη βάση δεδομένων χρονοσειράς (TSDB) [5].
- Η πύλη προώθησης βραχύβιων εργασιών [5].
- Ειδικούς εξαγωγείς, σχεδιασμένους αρκετά σύστημα και βάσεις δεδομένων όπως [5], JMX Exporter, Elsticsearch, PostgreSQL, Redis [94].
- Το σύστημα διαχείρισης ειδοποιήσεων [5].
- Βοηθητικά Frontend UI όπως Graphana, Prometheus Web UI [5].

Το Grafana αποτελεί ένα πρόγραμμα ανοιχτού κώδικα που επιτρέπει την αναζήτηση, οπτικοποίηση, ειδοποίηση και εξερεύνηση μετρικών από διάφορες πηγές, μεταξύ των οποίων και ο Prometheus [95].

Μέρος III

Πρακτικό Μέρος

Κεφάλαιο 3

Περιγραφή της Υλοποίησης

Στο κεφάλαιο αυτό γίνεται η περιγραφή της υλοποίησης. Παραθέτονται όλες οι αποφάσεις που συνείσφεραν στην επιλογή συγκεκριμένων τεχνολογιών και οι βασικοί περιορισμοί της υλοποίησης.

Στόχος μας είναι να εξετάσουμε την ελαστικότητα ενός κατανεμημένου συστήματος δεδομένων, σε περιβάλλον υπολογιστικού νέφους. Για να το επιτύχουμε αυτό πρέπει να εγκαταστήσουμε ένα περιβάλλον Kubernetes, να καταστήσουμε ικανό το Kubernetes να παρέχει δυναμικά τόμους, να εκκινήσουμε την Apache Cassandra, να σχεδιάσουμε πολιτική οριζόντιας κλιμάκωσης, να διαμορφώσουμε την παρακολούθηση του συστήματος, και να χρησιμοποιήσουμε το πρόγραμμα αξιολόγησης επιδόσεων YCSB.

3.1 Εγκατάσταση και ρύθμιση του Kubernetes

Τα μηχανήματα που θα συνθέσουν τον Kubernetes cluster είναι εικονικές μηχανές που παραχωρήθηκαν στο υπολογιστικό νέφος του Εργαστήριο Υπολογιστικών Συστημάτων της Σχολής Ηλεκτρολόγων Μηχανικών και Μηχανικών Η/Υ του ΕΜΠ. Προμηθεύτηκαν 6 εικονικές μηχανές (node1 - node6) συνδεδεμένες στο ίδιο δίκτυο και όλες οι μηχανές φιλοξενούνται σε διαφορετικό κόμβο του υπολογιστικού νέφους. Αυτό επιτρέπει στο σύστημα μας να είναι πιο ασφαλές από βλάβες και πιθανές εξαντλήσεις πόρων καθώς η λειτουργία κάθε εικονικής μηχανής είναι ανεξάρτητη από τις άλλες.

Host	Όνομα	IP
cloud1	node3	10.0.1.122
cloud3	node2	10.0.1.121
cloud4	node4	10.0.1.128
cloud6	node1	10.0.1.102
cloud5	node6	10.0.1.136
cloud7	node5	10.0.1.125

Πίνακας 3.1: *Τοπολογία κόμβων*

Υπάρχουν αρκετές επιλογές για την εγκατάσταση και ρύθμιση του Kubernetes:

1. Minikube [96]
2. MicroK8s [97]

3. Kubeadm [98]

4. Kubespray [99]

Επιλέξαμε το Kubespray καθώς οι πρώτες 2 επιλογές είναι ιδανικές για εκπαίδευση και τοπικές εγκαταστάσεις [96], [97], η τρίτη επιλογή είναι για περιβάλλοντα παραγωγής [98], ενώ το Kubespray είναι αρκετά ευέλικτο ώστε να επιτρέπει την εύκολη εγκατάσταση του cluster ανεξαρτήτως περιβάλλοντος και παρέχει αξιόπιστο περιβάλλον παραγωγής [99]. Από την έκδοση 2.3 και μετά το Kubespray άρχισε να χρησιμοποιεί το Kubeadm στο εσωτερικό του συστήματος για την δημιουργία του cluster [100].

Χρησιμοποιούμε την έκδοση 2.23.3 του Kubespray και εγκαθιστούμε την έκδοση 1.27 του Kubernetes, επιλογή που παρείχε την πιο σταθερή εγκατάσταση [101].

Αλγόριθμος Α΄.1

1. Δημιουργούμε αντίγραφο του δείγματος cluster.

2. Δηλώνουμε τα IP των κόμβων.

3. Δηλώνουμε το αρχείο διαμόρφωσης και δημιουργούμε τον κατάλογο υποδομής Ansible.

4. Εκτελούμε το βιβλίο εργασιών Ansible που εγκαθιστά τον cluster.

Μετά την εγκατάσταση μπορούμε να δούμε την κατάσταση του cluster εκτελώντας την εντολή *kubectl get nodes*. Το αποτέλεσμα της μας δίνει την κατάσταση των κόμβων του cluster.

Name	Status	Roles	Age	Version
node1	Ready	control-plane	127d	v1.27.10
node2	Ready	control-plane	127d	v1.27.10
node3	Ready	<none>	127d	v1.27.10
node4	Ready	<none>	127d	v1.27.10
node5	Ready	<none>	127d	v1.27.10
node6	Ready	<none>	127d	v1.27.10

Πίνακας 3.2: *Κατάσταση cluster*

Παρατηρούμε ότι οι δύο πρώτοι κόμβοι είναι Control planes. Λόγω το μικρού μεγέθους του cluster, θα χρησιμοποιήσουμε μόνο τον πρώτο σαν κύριο κόμβο που δεν θα εκτελούνται πάνω του φόρτοι εργασίας. Στους άλλους πέντε θα προωθηθεί το σύστημα το πειράματος.

3.2 Δυναμική Παροχή Τόμων

Όπως είδαμε στο προηγούμενο κεφάλαιο υπάρχουν αρκετές επιλογές για την παροχή τόμων (AzureFile, CephFS, τοπικός, NFS, RDB [32]). Επιλέξαμε τον CephFS, έναν αρκετά διαδεδομένο κατανεμημένο σύστημα αρχείων[102]. Επιτρέπει σε κόμβους να διαχειρίζονται ένα κοινό σύστημα αρχείων προσφέροντας υψηλή διαθεσιμότητα και κλιμακωσιμότητα. Είναι ιδανικό για την αποθήκευση δεδομένων που χρειάζονται κοινή πρόσβαση σε επίπεδο

αρχείων[102]. Το Rook είναι ένας χειριστής του Kubernetes που διευκολύνει την διάθεση, διαχείριση και ενορχήστρωση του συστήματος Ceph, ενσωματώνοντάς το στον Kubernetes με μικρή χειροκίνητη παρέμβαση [103].

Μία από τος προϋπόθεσης για την εγκατάσταση του Rook είναι η ύπαρξη [104]:

- Ακατέργαστων συσκευών δίσκου
- Ακατέργαστων κατατμήσεων
- Λογικών τόμων LVM
- Διαθέσιμων PV από κάποιο Storage Class σε λειτουργία block.

```
→ lsblk -f
NAME    FSTYPE   LABEL           UUID                                   FSAVAIL FSUSE% MOUNTPOINT
loop0   squashfs                                                             0   100% /snap/core18/2829
loop1   squashfs                                                             0   100% /snap/core18/2846
loop2   squashfs                                                             0   100% /snap/lxd/24061
loop3   squashfs                                                             0   100% /snap/core20/2379
loop4   squashfs                                                             0   100% /snap/core20/2434
loop5   squashfs                                                             0   100% /snap/core22/1722
loop6   squashfs                                                             0   100% /snap/core22/1748
loop7   squashfs                                                             0   100% /snap/lxd/29619
loop8   squashfs                                                             0   100% /snap/snapd/23545
loop9   squashfs                                                             0   100% /snap/snapd/23258
vda
├─vda1  ext4     cloudimg-rootfs 09603282-eb3e-4e2f-bb5a-fd09629981b7   30.1G    48% /
├─vda14
└─vda15 vfat     UEFI            0845-F46B                              98.3M     6% /boot/efi
```

Σχήμα 3.1: *lsblk command output*

Εκτελώντας την εντολή *lsblk -f* (απαρίθμηση συσκευών block), βλ. Σχήμα 3.1, δεν υπάρχει καμία ακατέργαστη συσκευή δίσκου, κατάτμηση ή λογικός τόμος. Μπορούμε να εγκαταστήσουμε το Rook χρησιμοποιώντας PV από κάποιο Storage Class. Ο τρόπος που θα χρησιμοποιήσουμε τον CephFS είναι να δημιουργήσουμε μία Storage Class που θα διαθέσει PV για να τους διαχειριστούν τα Pods μας. Οπότε:

1. θα πρέπει να δημιουργήσουμε PVs από κάποια Storage Class
2. που θα χρησιμοποιήσει ο Rook για να δημιουργήσει ένα Ceph cluster
3. που θα χρησιμοποιείται από μία Storage Class που θα δημιουργεί PVs για τα Pods μας.

Βλέπουμε ότι το δεύτερο βήμα είναι περιττό καθώς αν καταφέρουμε να δημιουργήσουμε αυτή την Storage Class και PVs τότε θα έχουμε δημιουργήσει τον πάροχο τόμων που θέλουμε.

1. Δημιουργούμε μία Storage Class και ρυθμίζουμε μόνο την λειτουργία δέσμευσης τόμων να περιμένει την κατανάλωση Α΄.2.
2. Δημιουργούμε έναν αριθμό από PV, ρυθμίζοντάς τα με τρόπο πρόσβασης ReadWriteOnce, χωρητικότητας πέντε Gigabyte, πολιτική ανάκτησης Retain, τρόπο λειτουργίας σύστημα αρχείων, συγγένεια κόμβων σε μόνο έναν κόμβο (κάθε κόμβος έχει αρκετά PVs) και τέλος το τοπικό μονοπάτι που αποθηκεύονται τα δεδομένα Α΄.3.

Σημειώνεται ότι οι δηλώσεις και οι διαμορφώσεις αυτού το συστήματος βρίσκονται αποθετήριο ανοιχτού κώδικα [105].

Με αυτόν τον τρόπο ορίζουμε σε ποιους κόμβους μπορούν να δρομολογηθούν Pods, τα Pods δρομολογούνται μόνο σε κόμβους με PVs, έτσι αποκλείουμε τον πρώτο κόμβο αν δεν δημιουργήσουμε PVs σε αυτόν. Παράλληλα εξασφαλίζουμε ότι οι τόμοι παραμένουν μετά το τέλος της λειτουργίας τους διαθέσιμοι.

Μπορούμε να ορίσουμε σε επίπεδο Statefulset ποια είναι η πολιτική ανάκτησης PVC, ορίζοντας την να διαγράφεται σε περίπτωση αποκλιμάκωσης και να παραμένει σε περίπτωση διαγραφής, μπορούμε να ελέγξουμε την ζωή του PV που συνδέεται στο PVC. Κατά την αποκλιμάκωση, διαγράφεται το PVC πράγμα που απελευθερώνει το PV. Έχουμε ένα σενάριο Python Α΄.9 που κατά τακτά χρονικά διαστήματα περιμένει να βρει απελευθερωμένα PVs, διαγράφει τα δεδομένα του (από το τοπικό μονοπάτι) και τέλος ανακυκλώνει το PV. Έχουμε έτσι δυναμική παροχή τόμων χρησιμοποιώντας πρωτογενή στοιχεία του Kubernetes, του λειτουργικού συστήματος και ένα σενάριο Python.

3.3 Apache Cassandra

Για την υλοποίηση της Apache Cassandra χρησιμοποιούμε Statefulset. Το συγκεκριμένο αντικείμενο μας εγγυάται την ταυτότητα δικτύου και σημειακή ταυτότητα που είναι απαραίτητη για την λειτουργία του κατανεμημένου συστήματος.

3.3.1 Εικόνα Container

Επιλέγουμε την εικόνα gcr.io/google-samples/cassandra:v14 που συντηρείται από την Google για περιβάλλοντα ανάπτυξης και έρευνας [106], σε περίπτωση που θέλουμε να εξετάσουμε περιβάλλοντα παραγωγής θα χρησιμοποιούσαμε είτε cassandra:4.1 ή datastax/dse-server .

3.3.2 Μέγεθος και Υπολογιστικοί Πόροι cluster

Για την εκτέλεση του φόρτου εργασίας πρέπει να ορίσουμε το μέγεθος του cluster μας. Έχουμε διαθέσιμους πέντε κόμβους Kubernetes, κάθε ένας από αυτούς έχει τέσσερις πυρήνες και οχτώ GB μνήμης. Αποφασίσαμε ότι για την εξέταση της γραμμικής κλιμάκωσης σε ένα επαρκές επίπεδο θα πρέπει να μπορούμε να πενταπλασιάσουμε τον φόρτο εργασίας. Αυτό μας οδηγεί στην επιλογή να κινηθούμε μεταξύ ενός και πέντε κόμβων Apache Cassandra, πράγμα που μας αφαιρεί την δυνατότητα να έχουμε αντίγραφα των δεδομένων σε άλλους κόμβους στην περίπτωση εκκίνησης του συστήματος. Αποφασίζουμε λοιπόν να κινηθούμε μεταξύ τριών και δεκαπέντε κόμβων Cassandra με πολιτική αντιγράφων δύο ή τριών ανάλογα το πείραμα. Θα πρέπει κάθε κόμβος Kubernetes να μπορεί να διαθέσει τρεις κόμβους Cassandra (Pods). Ορίζουμε τα όρια υπολογιστικών πόρων κάθε Container σε έναν πυρήνα και δύο GiB μνήμης. Σε αυτό το σημείο να σημειώσουμε ότι οι συγκεκριμένοι πόροι είναι λιγότεροι από όσους ορίζονται από την τεχνική τεκμηρίωση (για περιβάλλοντα ανάπτυξης 2 πυρήνες και 8 GB [107] μνήμης για συνθήκες μη φόρτωσης, ένα εικονικό μηχάνημα t2.large του Amazon Web Services[108]) και είναι σαν ένα εικονικό μηχάνημα t2.small της Amazon

Web Services [108]. Αυτό θα μας επιτρέψει να εξετάσουμε πως μπορεί να κλιμακωθεί και να ανταποκριθεί ελαστικά το σύστημα μας Α΄.6, Α΄.7, Α΄.8.

Πρέπει να διαμορφώσουμε τις διαστάσεις της μνήμης σωρού της εικονικής μηχανής Java (JVM). Η τεχνική τεκμηρίωση [109] συνιστά να ακολουθήσουμε τον τύπο $heap = max(min(1/2 * ram, 1024MB), min(1/4 * ram, 8GB)$. Αυτό μας δίνει μέγεθος σωρού 1024 MB. Πρέπει να ορίσουμε και το μέγεθος του τμήματος της μνήμης για τα νέα αντικείμενα όπου συνιστάται [109] να ακολουθήσουμε τον τύπο $heapnew = min(100 * cpu, 1/4 * ram)$. Αυτό μας δίνει 100 MB, ωστόσο μετά από πειραματισμό είδαμε ότι 256 MB εξασφαλίζουν καλύτερη απόδοση στο σύστημα μας.

3.3.3 Δίκτυο

Για την λειτουργία του Statefulset χρειάζεται μία ακέφαλη υπηρεσία η οποία λειτουργεί στην πόρτα 9042. Παράλληλα διαθέτουμε τις πόρτες 7000 και 7001 για επικοινωνία των κόμβων. Τέλος πρέπει να ορίσουμε τους κόμβους εκκίνησης του συστήματος, ορίζουμε τα πρώτα 2Pods Α΄.4, Α΄.6, Α΄.7, Α΄.8.

3.3.4 Δυναμική Παροχή Τόμων

Χρησιμοποιούμε την Storage Class και την πολιτική ανάκτησης PVC που ορίσαμε στην προηγούμενη ενότητα.

3.3.5 Ελαστικότητα

Για την επίτευξη ελαστικότητας πρέπει η Apache Cassandra να μπορεί να κλιμακώνεται και να αποκλιμακώνεται. Κατά την προσθήκη νέου Pod ο νέος Cassandra Container επικοινωνεί με τους κόμβους εκκίνησης, συνδέεται στον δακτύλιο και μεταφέρονται τα δεδομένα που του ανήκουν. Για την αντίθετη διαδικασία, μεταφορά αυτών των δεδομένων από τον κόμβο και την απεγγραφή του από τον δακτύλιο διαμορφώνουμε τον κύκλο ζωής του Container προσθέτωντας 2 εντολές του κόμβου Cassandra:

- nodetool decommission [110]
- nodetool drain [111]

Σημειώνεται ότι οι δηλώσεις και οι διαμορφώσεις αυτού το συστήματος βρίσκονται αποθετήριο ανοιχτού κώδικα [105].

3.4 Οριζόντια Κλιμάκωση

Για την οριζόντια κλιμάκωση θα χρησιμοποιήσουμε έναν Server Μετρικών [112], αναπτύχθηκε και παρέχεται από τον Kubernetes, και επιτρέπει να ορίσουμε HPA που μπορεί να ελέγχει την χρήση του επεξεργαστή ή και της μνήμης. Δεν μπορεί να ελέγξει άλλες μετρικές, αλλά επιλέχθηκε καθώς αυτές οι μετρικές μπορούν να δώσουν την σωστή εικόνα του cluster κάτω από φόρτο.

Παρατηρούμε ότι λόγω της περιορισμένης μνήμης (ένα τέταρτο της προτεινόμενης) το Pod δεσμεύει όση περισσότερη μπορεί (ακόμα και πάνω από το όριο) και αυτή μένει αμετάβλητη ανεξαρτήτως του φόρτου. Οπότε η μετρική που θα ελέγχει το HPA είναι η χρήση του επεξεργαστή.

Μετά από διάφορα πειράματα είδαμε ότι καλά όρια είναι το ογδόντα και ενενήντα τις εκατό του επεξεργαστή.

Δημιουργούμε 3 HPAs, ένα για κλιμάκωση σταθερού φόρτου Α΄.10, ένα για κλιμάκωση ημιτονοειδούς σήματος Α΄.11 και έναν για αποκλιμάκωση ημιτονοειδούς σήματος Α΄.12. Η διαφορά μεταξύ των 2 τελευταίων HPA είναι ότι το Α΄.12 είναι πιο ευαίσθητο στην αποκλιμάκωση καθώς έχει μικρότερη περίοδο, παράθυρο σταθεροποίησης και μέση χρήση.

3.5 Αξιολόγηση Επιδόσεων

Για την αξιολόγηση των επιδόσεων του συστήματος χρησιμοποιούμε το YCSB. Μπορούμε να τρέξουμε φόρτο εργασιών ρυθμίζοντας διάφορες παραμέτρους.

- Αριθμός νημάτων επεξεργαστή
- Στόχο ρυθμού μετάδοσης των συναλλαγών
- Η αναλογία του τι είδος συναλλαγών εκτελούνται (εγγραφής, ανάγνωσης, προσάρμοσης, σάρωσης)
- Επιθυμητή συνέπεια ανά είδος συναλλαγών
- Αριθμός σειρών στην βάση δεδομένων
- Αριθμός συναλλαγών
- Κατανομή επιλογής σειρών για τις συναλλαγές

Θα προσαρμόζονται αυτά τα στοιχεία ανάλογα με τον στόχο κάθε πειράματος.

Σημαντική σημείωση είναι ότι το YCSB δεν υποστηρίζει φόρτο μεταβαλλόμενου πλάτους. Για να εκτελέσουμε ημιτονοειδές σήμα πρέπει να το επεκτείνουμε.

Ο τρόπος που ελέγχεται ο ρυθμός μετάδοσης των συναλλαγών είναι κατά την διάρκεια εκτέλεσης κάθε συναλλαγής να υπάρχει μία καθυστέρηση που να ορίζει αυτόν τον ρυθμό. Όμως αυτή η καθυστέρηση παραμένει σταθερή. Ο τρόπος που μπορούμε να επηρεάσουμε την καθυστέρηση είναι να αλλάζουμε την τιμή της ακολουθώντας ένα ημιτονοειδές σήμα. Επεκτείνουμε αυτό το κομμάτι [113] του κώδικα δημιουργώντας μία Διεπαφή Α΄.13 και δύο Στρατηγικές μία για την σταθερή αναμονή Α΄.14 και μία για την ημιτονοειδή Α΄.15. Κατά την δημιουργία κάθε ClientThread διαμορφώνεται η στρατηγική και κατά την εκτέλεση των συναλλαγών του, η αναμονή ακολουθεί την Στρατηγική αυτή Α΄.16.

3.6 Παρακολούθηση

Για την παρακολούθηση της κατάστασης του cluster χρησιμοποιούμε τον Prometheus. Τον εγκαθιστούμε χρησιμοποιώντας το εργαλείο Helm, το οποίο ομαδοποιεί σε Χάρτες όλες

τις δηλώσεις αντικειμένων Kubernetes, τις διαμορφώσεις τους και πιθανών άλλους Χάρτες, που χρειάζονται για την εγκατάσταση ενός συστήματος [114]. Χρησιμοποιούμε τον Χάρτη που έχει αναπτυχθεί από την κοινότητα ανοιχτού κώδικα του Prometheus [115]. Παράλληλα διαμορφώνουμε αντικείμενα παρακολούθησης υπηρεσιών του Kubernetes που ορίζουν τις υπηρεσίες και τις πόρτες που θα χρησιμοποιήσει ο Prometheus [105]. Με αυτόν τον τρόπο έχουμε μετρικές από τον Kubernetes.

Αυτό όμως δεν είναι ικανό να παρακολουθήσει εξειδικευμένες μετρικές Apache Cassandra. Για αυτό το λόγο, χρησιμοποιούμε έναν εξαγωγέα JMX [116]. Το JMX αποτελεί μία Java API που προσφέρει πρότυπο τρόπο για να παρακαλοθούνται εφαρμογές Java. Ο τρόπος που τον χρησιμοποιούμε είναι με το να διαθέσουμε σε κάθε Pod άλλον έναν Container με συνδεσμολογία sidecar. Προσθέτουμε τον ορισμό του Container και ορίζουμε 2 νέες πόρτες την 7199 για τον εξαγωγέα και την 8080 για διάθεση των μετρικών του κάθε Pod που τρέχει κόμβο Cassandra Α΄.6, Α΄.7, Α΄.8, Α΄.4. Προσαρμόζουμε και τα αντικείμενα παρακολούθησης υπηρεσιών να χρησιμοποιούν την πόρτα μετρικών.

Χρησιμοποιούμε το πρόγραμμα Grafana για την παρουσίαση των μετρικών που μας ενδιαφέρουν για την λειτουργία του πειράματος. Αυτό θα οπτικοποιήσει τα δεδομένα από αναζητήσεις στην γλώσσα ερωτημάτων του Prometheus, PromQL [117].

Αυτές είναι οι:

- Α΄.17: τα αιτήματα στο σύστημα
- Α΄.18: ο αριθμός των αντιγράφων
- Α΄.19: το εύρος ζώνης εισερχόμενης κίνησης δικτύου
- Α΄.20: ο ρυθμός εισερχόμενων πακέτων
- Α΄.21: το εύρος ζώνης εξερχόμενης κίνησης δικτύου
- Α΄.22: ο ρυθμός εξερχόμενων πακέτων
- Α΄.23: τα αιτήματα ανάγνωσης στο σύστημα
- Α΄.24: τα αιτήματα εγγραφής στο σύστημα
- Α΄.25: οι λειτουργίες IO ανάγνωσης
- Α΄.26: ο ρυθμός IO ανάγνωσης
- Α΄.27: οι λειτουργίες IO εγγραφής
- Α΄.28: ο ρυθμός IO εγγραφής
- Α΄.29: η χρήση της μνήμης
- Α΄.30: το όριο χρήσης της μνήμης
- Α΄.31: η χρήση επεξεργαστή
- Α΄.32: το όριο χρήσης του επεξεργαστή
- Α΄.33: ο βαθμός χρήσης του HPA

3.7 Περιορισμοί

Ο βασικός περιορισμός που διέπει την υλοποίηση αυτής της εργασίας είναι οι διαθέσιμοι υπολογιστικοί πόροι. Κάθε Pod της Apache Cassandra έχει διαθέσιμους πόρους ενός t2.small και όχι t2.large [108], αυτό οδηγεί την εξάντληση όλων των πόρων σε κατάσταση ηρεμίας και δεν επιτρέπει την παρατήρηση μεγαλύτερου εύρους λειτουργίας του συστήματος. Κατ' επέκταση περιορίζει το εύρος των πειραμάτων που γίνεται να υλοποιηθούν - ενδεικτικά Οριζόντιες Κλιμακώσεις σε διαφορετικά όρια. Ακόμα, ο τρόπος λειτουργίας του HPA δεν επιτρέπει τον ορισμό μίας περιοχής υστέρησης, όπου δεν κλιμακώνεται ούτε αποκλιμακώνεται το σύστημα.

Κεφάλαιο 4

Περιγραφή του Πειράματος

Στο κεφάλαιο αυτό γίνεται η περιγραφή του πειράματος.

4.1 Πείραμα Ρυθμιζόμενης Συνέπειας

Στο συγκεκριμένο πείραμα θα εξετάσουμε πως ανταποκρίνεται το σύστημά μας σε διαφορετικούς βαθμούς συνέπειας.

Επιλογή	Τιμή
Αριθμός Pods	10
Αντίγραφα	3
Νήματα επεξεργαστή	20
Σειρές	2000000
Συναλλαγές	2000000
Κατανομή επιλογής σειρών	Zipfian
Βαθμοί συνέπειας	Ένα, Απαρτία και Όλοι
Αναλογίες συναλλαγών	Φόρτος εργασίας B, Φόρτος εργασίας A

Πίνακας 4.1: *Πείραμα Συνέπειας*

Ο Φόρτος εργασίας A αποτελείται από 50% αναγνώσεις και 50% τροποποιήσεις, παρομοιάζοντας φόρτο αποθήκες καταγραφής συνεδριών. Ο Φόρτος εργασίας B αποτελείται από 95% αναγνώσεις και 5% τροποποιήσεις, παρομοιάζοντας φόρτο εργασίας συστήματος επισήμανσης αρχείων, τροποποίηση είναι η δημιουργία επισήμανσης και οι περισσότερες συναλλαγές είναι αναγνώσεις.

Περιμένουμε την απόδοση του συστήματος να μειώνεται όσο ανεβάζουμε τον βαθμό συνέπειας, καθώς κάθε συναλλαγή θα πρέπει να φτάσει σε περισσότερα αντίγραφα πριν επιβεβαιωθεί.

4.2 Πείραμα Κλιμάκωσης - Αποκλιμάκωσης χωρίς Φόρτο εργασίας

Στο συγκεκριμένο πείραμα θα εξετάσουμε πως ανταποκρίνεται το σύστημά μας σε εντολή κλιμάκωσης. Εκκινούμε με 3 Pods που περιέχουν 2000000 σειρές και δίνουμε εντολή

κλιμάκωσης του Statefulset σε 15 Pods. Τέλος δίνουμε εντολή αποκλιμάκωσης σε 3 Pods ξανά.

Επιλογή	Τιμή
Αριθμός Pods	3-15
Αντίγραφα	3
Σειρές	2000000

Πίνακας 4.2: *Πείραμα Κλιμάκωσης - Αποκλιμάκωσης*

Θα παρατηρήσουμε πως το σύστημα αντιδρά σε κλιμάκωση, αποκλιμάκωση, πως κατανέμονται τα δεδομένα, και ποιες ενέργειες εκτελούνται.

4.3 Πείραμα Ελαστικότητας - Κλιμάκωση

Στο συγκεκριμένο πείραμα θα εξετάσουμε πως ανταποκρίνεται το σύστημά μας σε σταθερό φόρτο εργασίας με διαφορετικούς αριθμούς σειρών στη βάση.

Επιλογή	Τιμή
Αριθμός Pods	3-15
Αντίγραφα	2
Νήματα επεξεργαστή	500
Σειρές	100000, 2000000
Συναλλαγές	55000000
Κατανομή επιλογής σειρών	Zipfian
Βαθμοί συνέπειας	Ένα
Αναλογίες συναλλαγών	Φόρτος εργασίας A
HPA	Α΄.10

Πίνακας 4.3: *Πείραμα Κλιμάκωσης με φόρτο*

Αναμένουμε ότι ο μικρός αριθμός σειρών δεν θα γεμίζει τα Memtables οπότε ούτε θα χρειαστεί να εγγραφούν σε SSTables. Αναμένουμε ότι ο μεγαλύτερος αριθμός σειρών θα γεμίζει τα Memtables οπότε θα έχουν δημιουργηθεί SSTables, η προσπέλαση των δεδομένων από τα SSTables θα αυξήσει τον αριθμό των αναγνώσεων από τον δίσκο. Ακόμα θέλουμε να δούμε ότι το σύστημα μας είναι ελαστικό και είναι ικανό να διαχειριστεί τον όγκο συναλλαγών προσθέτοντας κόμβους χωρίς να μειωθεί η απόδοσή και η σταθερότητά του. Τέλος θέλουμε να δούμε την γραμμική κλιμάκωση του φόρτου εργασίας και αν θα είναι ανάλογη της κλιμάκωσης.

4.4 Πείραμα Ελαστικότητας - Κλιμάκωση με Ημιτονοειδές Σήμα

Στο συγκεκριμένο πείραμα θα εξετάσουμε πως ανταποκρίνεται το σύστημά μας σε ημιτονοειδές σήμα εκκινώντας από 3 Pods.

Επιλογή	Τιμή
Αριθμός Pods	3-15
Αντίγραφα	2
Νήματα επεξεργαστή	15
Σειρές	100000
Συναλλαγές	5000000
Κατανομή επιλογής σειρών	Zipfian
Βαθμοί συνέπειας	Ένα
Αναλογίες συναλλαγών	Φόρτος εργασίας A
Περίοδος	1200
Πλάτος	100
Κατακόρυφη Μετατόπιση	150
HPA	Α΄.11

Πίνακας 4.4: *Πείραμα Κλιμάκωσης με ημιτονοειδές φόρτο*

Αρχικά θέλουμε να δούμε ένα ημιτονοειδές φόρτο εργασίας. Αναμένουμε να δούμε οι σειρές να χωράνε στα Memtables, οπότε δεν θα υπάρχουν αναγνώσεις από τον δίσκο. Παράλληλα περιμένουμε κλιμάκωση και αποκλιμάκωση του συστήματος μας ανάλογα με τον φόρτο εργασίας χωρίς σφάλματα.

4.5 Πείραμα Ελαστικότητας - Αποκλιμάκωση με Ημιτονοειδές Σήμα

Στο συγκεκριμένο πείραμα κάνουμε την αντίθετη εργασία από το προηγούμενο πείραμα, θα εξετάσουμε πως ανταποκρίνεται το σύστημά μας σε ημιτονοειδές σήμα εκκινώντας από 10 Pods.

Επιλογή	Τιμή
Αριθμός Pods	3-15
Αντίγραφα	3
Νήματα επεξεργαστή	5
Σειρές	2000000
Συναλλαγές	5000000
Κατανομή επιλογής σειρών	Zipfian
Βαθμοί συνέπειας	Ένα
Αναλογίες συναλλαγών	Φόρτος εργασίας A
Περίοδος	1200
Πλάτος	100
Κατακόρυφη Μετατόπιση	150
HPA	Α΄.12

Πίνακας 4.5: *Πείραμα Αποκλιμάκωσης με ημιτονοειδές φόρτο*

Θέλουμε να δούμε πως θα ανταποκριθεί το σύστημά μας στην αποκλιμάκωση. Το σήμα που χρησιμοποιούμε θα είναι αρκετό για να γίνονται αποκλιμακώσεις. Περιμένουμε ο αριθμός των αντιγράφων να κυμαίνεται μεταξύ κάποιων τιμών, μεγαλύτερων κάτω από την μέγιστη τιμή του ημιτονοειδούς σήματος και μικρότερων κατά την ελάχιστη τιμή του.

Κεφάλαιο 5

Σχολιασμός Αποτελεσμάτων Πειραμάτων

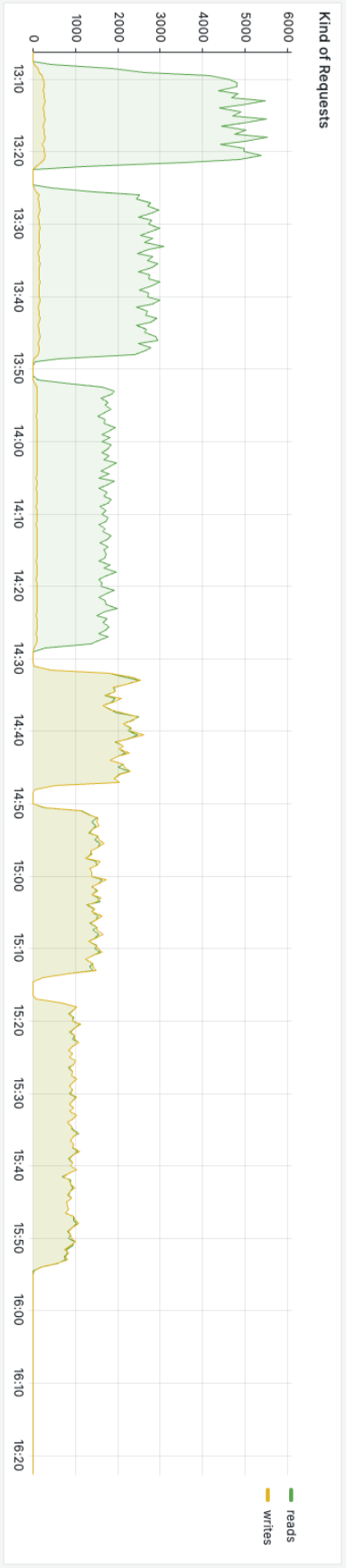
Στο κεφάλαιο αυτό παρουσιάζονται τα αποτελέσματα των πειραμάτων, όπως αυτά έχουν περιγραφεί στο προηγούμενο κεφάλαιο.

5.1 Ρυθμιζόμενη Συνέπεια

Στην ενότητα παραθέτουμε τα αποτελέσματά του Πειράματος Ρυθμιζόμενης Συνέπειας 4.1. Τρέξαμε τον φόρτο εργασίας που περιγράφεται στον Πίνακα 4.1. Τρεις εκτελέσεις με αυξανόμενη συνέπεια για τον Φόρτο Εργασίας Β και μετά τις ίδιες συνέπειες για τον Φόρτο Εργασίας Α.

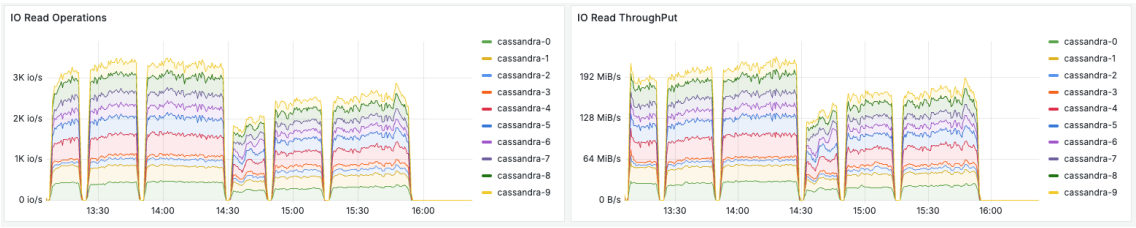
Σχήμα 5.1: *Ρυθμός Αιτημάτων Α'.17*

Παρατηρούμε ότι, βλ. Σχήμα 5.1, όσο αυξάνουμε την Συνέπεια (Ένα αντίγραφο, Απαρτία, Όλα) τόσο μειώνεται η απόδοση του συστήματος και ο χρόνος εκτέλεσης του αυξάνεται. Αυτό είναι αναμενόμενο καθώς πρέπει να λάβουμε επιβεβαιώσεις από παραπάνω από ένα αντίγραφο. Στην προκειμένη περίπτωση με 3 αντίγραφα λαμβάνουμε 1, 2, και 3 επιβεβαιώσεις αντίστοιχα.

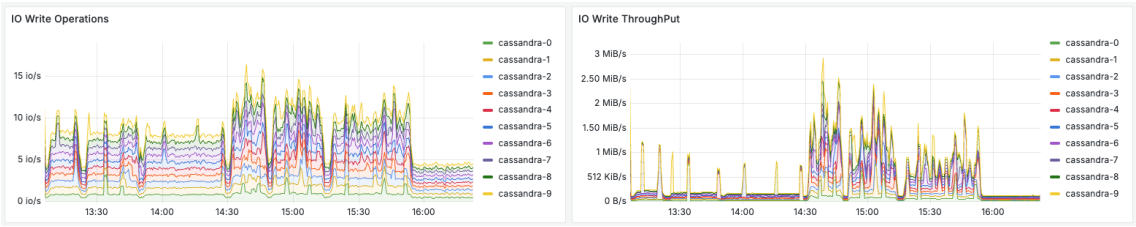


Σχήμα 5.2: *Χωρισμός Αιτημάτων Α΄.23, Α΄.24*

Παράλληλα παρατηρούμε ότι, βλ. Σχήμα 5.2, οι διαφορετικοί φόροι εργασίας εκτελούν την αναμενόμενη αναλογία εντολών ανάγνωσης και μορφοποίησης/εγγραφής.



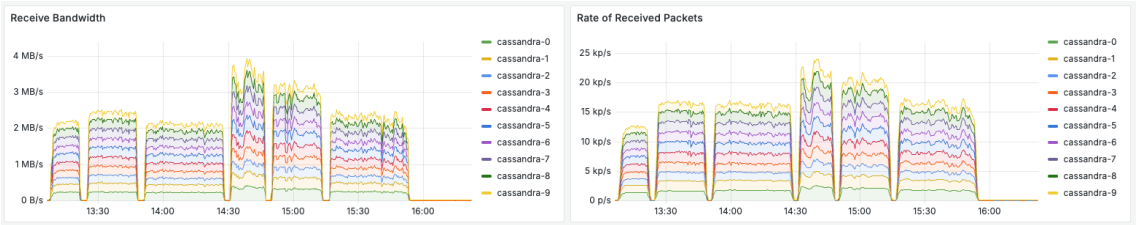
Σχήμα 5.3: *Αναγνώσεις Α'.25, Α'.26*



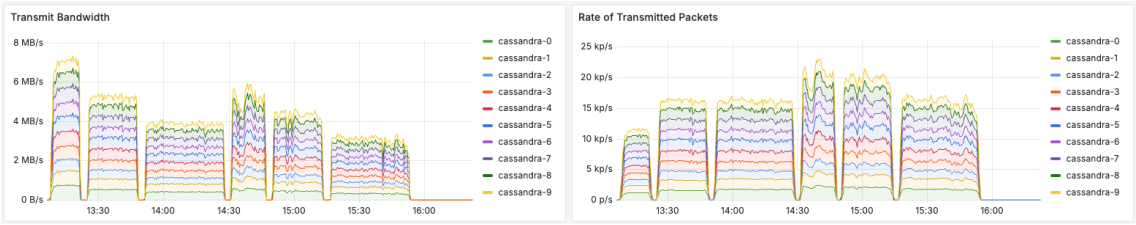
Σχήμα 5.4: *Εγγραφές Α'.27, Α'.28*

Παρατηρούμε ότι, βλ. Σχήμα 5.3, ο αριθμός των εντελών ανά φόρτο εργασίας είναι παρόμοιος, γεγονός που σημαίνει ότι η Apache Cassandra εκτελεί όσες περισσότερες εργασίες μπορεί καθότι για τον ίδιο αριθμό ανάγνωσης από τον δίσκο επιβεβαιώνεται μικρότερη απόδοση. Παράλληλα επιβεβαιώνει οτι χρειάζονται περισσότερες διεργασίες από το σύστημα για κάθε αίτημα.

Παρατηρούμε ότι, βλ. Σχήμα 5.4, οι εγγραφές δεν είναι το ίδιο σταθερές, καθώς κάθε συναλλαγή εγγράφεται στο Commitlog στον δίσκο και εισέρχεται σε Memtable στην μνήμη αλλά μόλις γεμίσει γράφεται στον δίσκο σε SSTable.



Σχήμα 5.5: *Εισερχόμενη κίνηση δικτύου Α'.19, Α'.20*



Σχήμα 5.6: *Εξερχόμενη κίνηση δικτύου Α'.21, Α'.22*

Παρατηρούμαι ότι, βλ. Σχήμα 5.1, 5.5, 5.6, ενώ η απόδοση του συστήματος είναι ίδια μεταξύ φόρτων εργασίας, ενώ η χρήση του δικτύου είναι διαφορετική. Ο Φόρτος Εργασίας Α εκτελεί μεγαλύτερη αναλογία αιτημάτων μορφοποίησης (εγγραφής) οπότε περισσότερα δεδομένα μεταφέρονται για κάθε αίτημα.

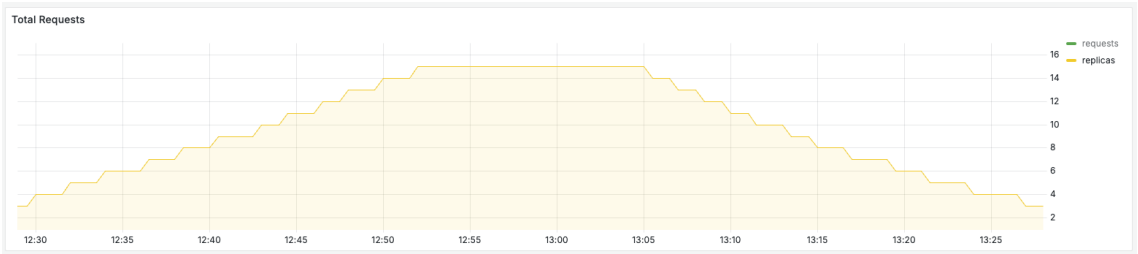
Αυτό το πείραμα μας δείχνει ότι η απόδοση (διαθεσιμότητα) και η συνέπεια ενός κατανεμημένου συστήματος είναι αντιστρόφως ανάλογες. Αυτή η συμπεριφορά είναι μία από τις εγγυήσεις της Apache Cassandra.

Τέλος, είναι εντυπωσιακό ότι ο φόρτος εργασίας αναγνώσεων έχει την ίδια απόδοση με τον φόρτο εργασίας εγγραφών και αναγνώσεων.

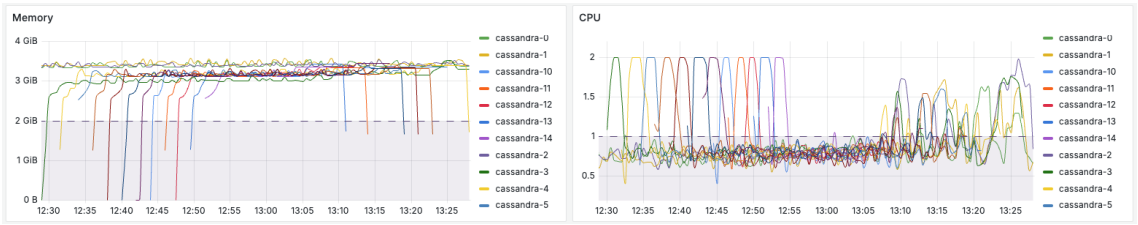
5.2 Κλιμάκωση - Αποκλιμάκωση χωρίς Φόρτο εργασίας

Στην ενότητα παραθέτουμε τα αποτελέσματά του Πειράματος Κλιμάκωσης - Αποκλιμάκωσης χωρίς Φόρτο εργασίας 4.2.

Τρέξαμε τον φόρτο εργασίας που περιγράφεται στον Πίνακα 4.2. Εκκινούμε με 3 κόμβους με 2000000 σειρές εγγεγραμμένες. Κλιμακώνουμε και αποκλιμακώνουμε την Apache Cassandra από 3 σε 15 και μετά ξανά 3 κόμβους.

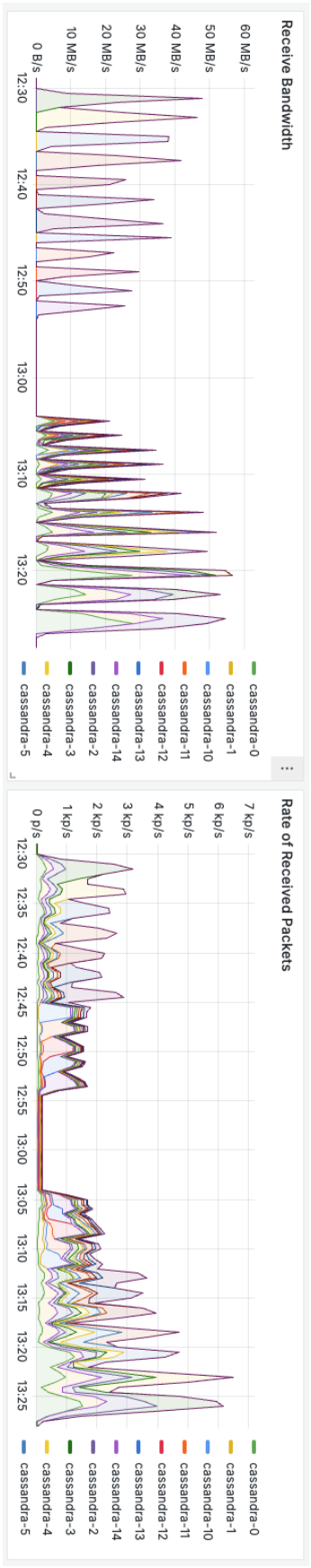


Σχήμα 5.7: *Αριθμός Αντιγράφων Α΄.18*

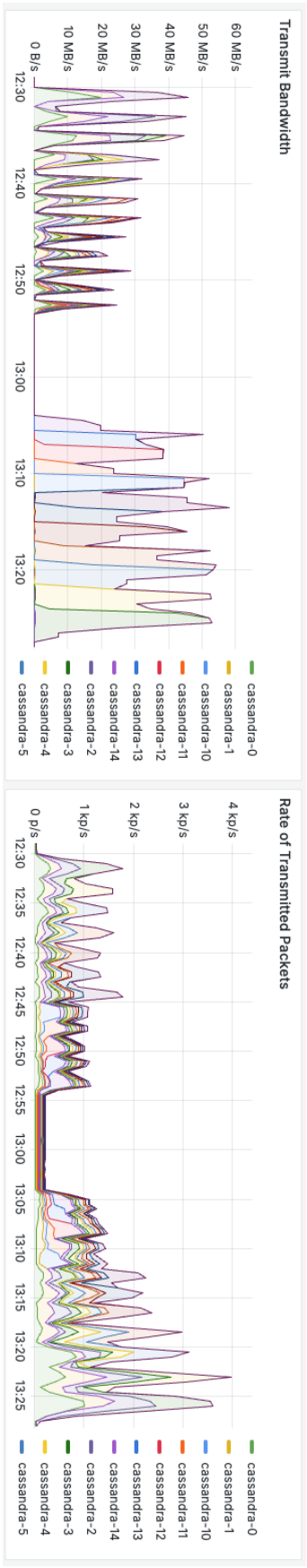


Σχήμα 5.8: *Χρήση Υπολογιστικών Πόρων Α΄.29, Α΄.30, Α΄.31, Α΄.32*

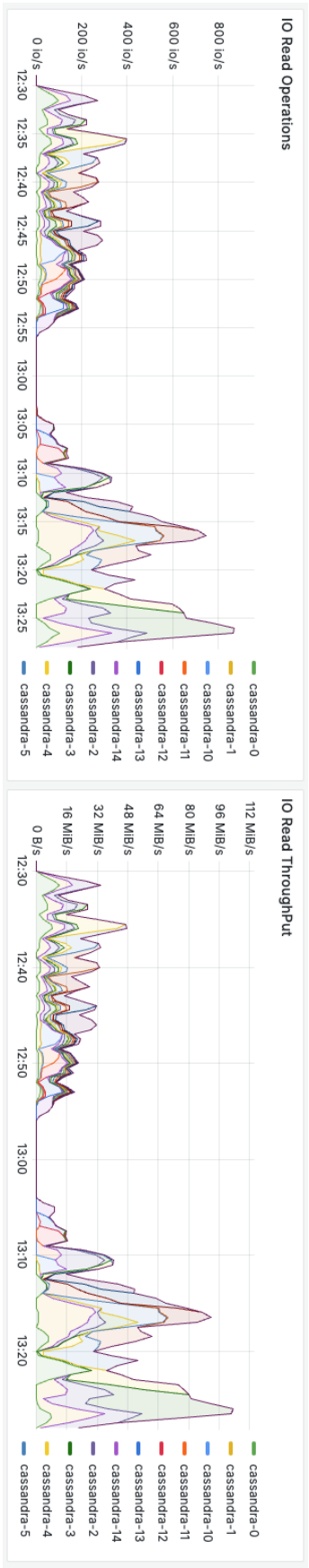
Παρατηρούμε ότι, βλ. Σχήματα 5.7, 5.8, η χρήση μνήμης καθ' όλη την διάρκεια του κύκλου ζωής ενός Container Apache Cassandra είναι αρκετά πάνω από το όριο. Αυτό δεν μας επιτρέπει να χρησιμοποιήσουμε την συγκεκριμένη μετρική για κλιμάκωση του συστήματος. Παρατηρούμε ακόμα ότι η χρήση του επεξεργαστή κυμαίνεται από 70% μέχρι 200%, η συγκεκριμένη μετρική μπορεί να χρησιμοποιηθεί για την Οριζόντια κλιμάκωση του συστήματος. Ο περιορισμός ότι κάθε Pod έχει αρκετά λιγότερους υπολογιστικούς πόρους από το προτεινόμενο περιορίζει τον τρόπο με τον οποίο μπορούμε να εξετάσουμε την ελαστικότητα του συστήματος μας. Παρόλα αυτά θα την εξετάσουμε με βάση την χρήση του επεξεργαστή.



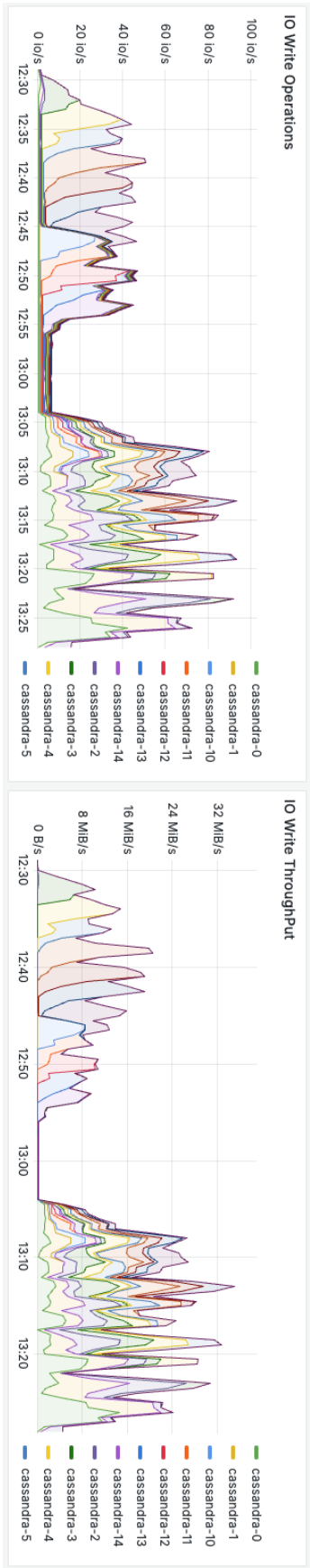
Σχήμα 5.9: *Εισερχόμενη κίνηση δικτύου Α'.19, Α'.20*



Σχήμα 5.10: *Εξερχόμενη κίνηση δικτύου Α'.21, Α'.22*



Σχήμα 5.11: *Αναγνώσεις Α'.25, Α'.26*



Σχήμα 5.12: *Εγγραφές Α'.27, Α'.28*

Παρατηρούμε ότι, βλ. Σχήματα 5.9, 5.10, 5.11, 5.12, καθώς προστίθενται κόμβοι λαμβάνουν (αποκλειστικά τα αντίγραφα των δεδομένων τους) και όλοι οι προϋπάρχοντες κόμβοι τα αποστέλλουν. Αυτό είναι αναμενόμενο και μας αποδεικνύει ότι το σύστημα χρησιμοποιεί εικονικούς κόμβους, μάρκες, στον χώρο δακτυλίων, αν δεν υπήρχαν αυτοί τότε οι αναπροσαρμογές θα ήταν πιο τοπικές. Παράλληλα, από τις αναγνώσεις και εγγραφές βλέπουμε πιο ασυνεπή συμπεριφορά, αυτή οφείλεται στην αναπροσαρμογή των δεδομένων τους που χρειάζεται αναγνώσεις και εγγραφές SSTables.

```
Datacenter: ntua
================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  10.233.74.77    1.48 GiB   32           17.7%             72999a77-d2f8-4788-be0e-8f8ae5112e84  rack1
UN  10.233.75.13    751.47 MiB  32           18.5%             1588345e-f1fc-40b4-aa4e-df2764241750  rack1
UN  10.233.71.13    532.27 MiB  32           20.3%             260f6237-e6ae-421b-ae0c-232e0e67e243  rack1
UN  10.233.74.79    471.68 MiB  32           21.1%             1bc21661-a2a1-42da-b3d7-03e03c32938f  rack1
UN  10.233.97.174   1.09 GiB   32           22.0%             55c95d60-9cce-4256-b469-87f30f72139c  rack1
UN  10.233.75.1     2.04 GiB   32           16.6%             f7c1fa0b-ea19-4e8b-8e98-7d7e1ba17b00  rack1
UN  10.233.97.129   567.72 MiB  32           20.2%             b9f8913a-9b69-4057-aec5-e88d81f342af  rack1
UN  10.233.97.160   2.53 GiB   32           19.1%             9f95425c-e48d-4f48-b902-98f2a5d9f96b  rack1
UN  10.233.74.69    763.4 MiB  32           21.0%             ba9de8f3-6dd5-4aab-81e8-1370039aba47  rack1
UN  10.233.75.88    449.16 MiB  32           21.5%             2962d76f-329b-4fe3-a1eb-8af84dced15f  rack1
UN  10.233.71.24    2.71 GiB   32           20.5%             621b56e5-c788-4958-9ec1-642e80d99544  rack1
UN  10.233.75.124   718.13 MiB  32           21.3%             205a5344-c498-41b3-8b2f-fa4296585b2c  rack1
UN  10.233.71.63    1.22 GiB   32           17.5%             0d26ec8f-2169-4c78-a689-e9b9b1bf4134  rack1
UN  10.233.75.20    523.06 MiB  32           19.9%             0621ec03-dfc4-4849-ad96-bf0d9fab33d6  rack1
UN  10.233.75.118   1.19 GiB   32           22.9%             0121ae5f-42cc-4e8b-b3a0-da988d1534ad  rack1
```

Σχήμα 5.13: *Κατάσταση 15 Κόμβων*

```
Datacenter: ntua
================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  10.233.71.24    4.36 GiB   32           100.0%            621b56e5-c788-4958-9ec1-642e80d99544  rack1
UN  10.233.75.1     4.48 GiB   32           100.0%            f7c1fa0b-ea19-4e8b-8e98-7d7e1ba17b00  rack1
UN  10.233.97.160   4.47 GiB   32           100.0%            9f95425c-e48d-4f48-b902-98f2a5d9f96b  rack1
```

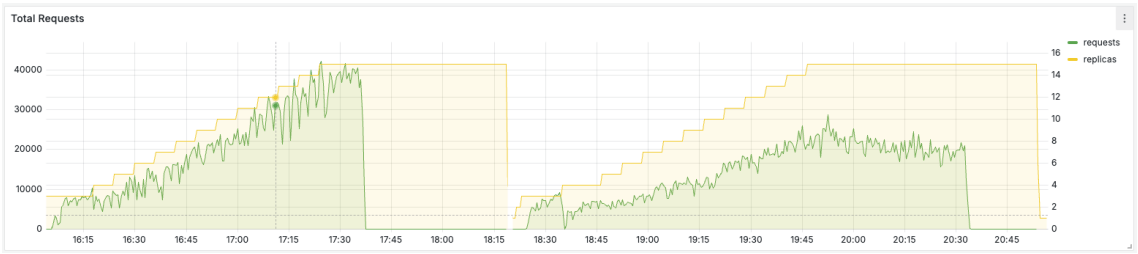
Σχήμα 5.14: *Κατάσταση 3 Κόμβων*

Τέλος, παρατηρούμε, βλ. Σχήματα 5.13, 5.14, την κατανομή των δεδομένων στην μεγαλύτερη και μικρότερη έκταση του συστήματος. Τα δεδομένα είναι κατανεμημένα ομοιόμορφα (μεταξύ 16.6% και 22.9%). Υπάρχει αναντιστοιχία μεταξύ φόρτου και κυριότητας των δεδομένων μεταξύ κόμβων, αυτό οφείλεται στο γεγονός ότι κάποια σημεία του δακτυλίου είναι πιο "ζεστά" (προσπελάσσονται πιο συχνά), οπότε υπάρχουν περισσότερα SSTables, Commitlogs, διαμορφώθηκε από την zipfian κατανομή.

5.3 Ελαστικότητα - Κλιμάκωση

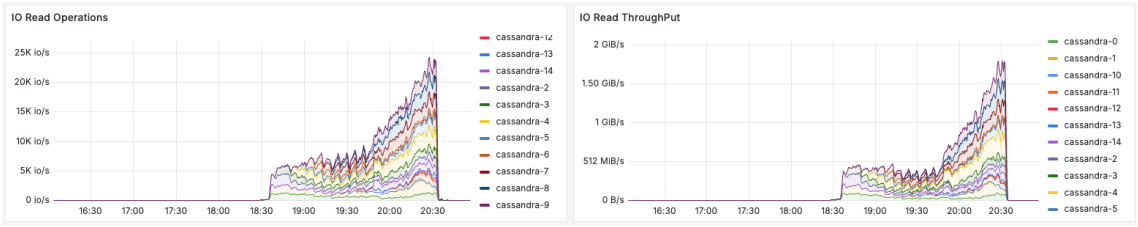
Στην ενότητα παραθέτουμε τα αποτελέσματά του Πειράματος Ελαστικότητας - Κλιμάκωση 4.3.

Τρέξαμε τον φόρτο εργασίας που περιγράφεται στον Πίνακα 4.3. Δύο εκτελέσεις με διαφορετικό αριθμό σειρών.

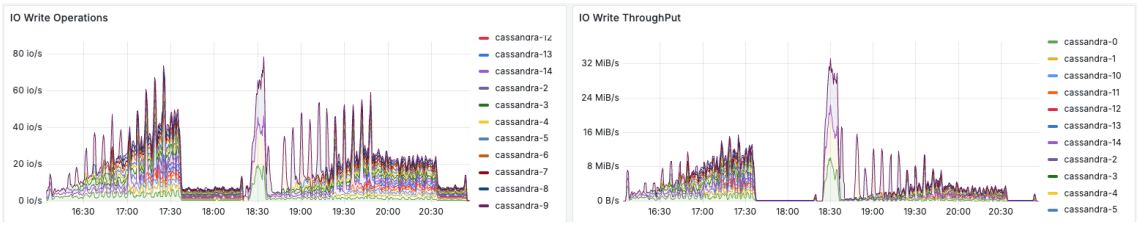


Σχήμα 5.15: *Ρυθμός Αιτημάτων Α΄.17 - Αριθμός Αντιγράφων Α΄.18*

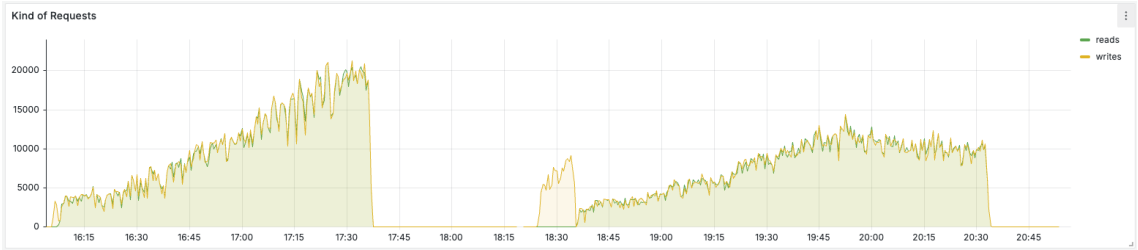
Παρατηρούμε ότι, βλ. Σχήμα 5.15, και στους δύο φόρτους η απόδοση του συστήματος αυξάνεται γραμμικά, πενταπλασιάζοντας την απόδοση του καθώς πενταπλασιάζεται το μέγεθός του ακόμα και αν το σύστημα έχει λιγότερους από το προτεινόμενο πόρους. Αυτή η γραμμική αύξηση της απόδοσης υπάρχει ανεξαρτήτως του αριθμού των σειρών.



Σχήμα 5.16: *Αναγνώσεις Α΄.25, Α΄.26*

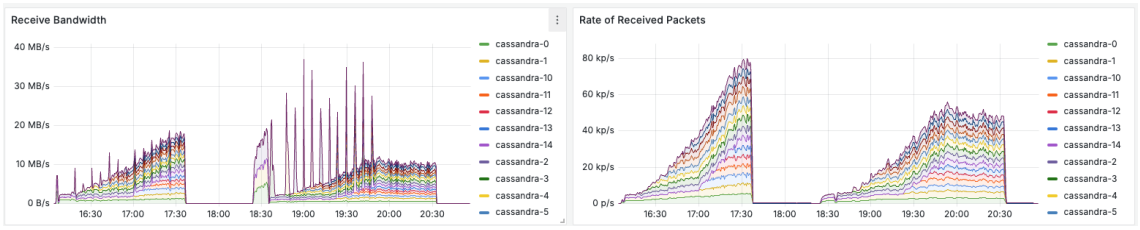


Σχήμα 5.17: *Εγγραφές Α΄.27, Α΄.28*

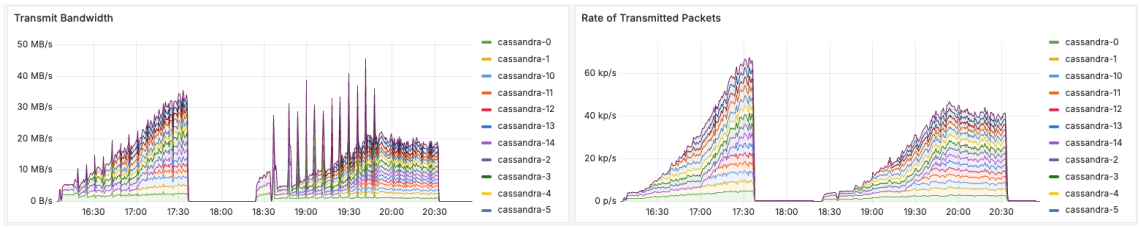


Σχήμα 5.18: *Χωρισμός Αιτημάτων Α΄.23, Α΄.24*

Ακόμα, ο μεγαλύτερος αριθμός σειρών γεμίζει τα Memtables και ο κάθε κόμβος τα αποθηκεύει στον δίσκο σε SSTables, αυτό φαίνεται από τις αναγνώσεις που γίνονται στον δεύτερο φόρτο εργασίας, βλ. Σχήμα 5.16. Οι εγγραφές είναι παρόμοιες για τους 2 φόρτους εκτός από την φόρτωση των δεδομένων όπου στον δεύτερο φόρτο εργασίας έχουμε τάξη μεγέθους περισσότερες σειρές, βλ. Σχήματα 5.17, 5.18.



Σχήμα 5.19: *Εισερχόμενη κίνηση δικτύου Α΄.19, Α΄.20*



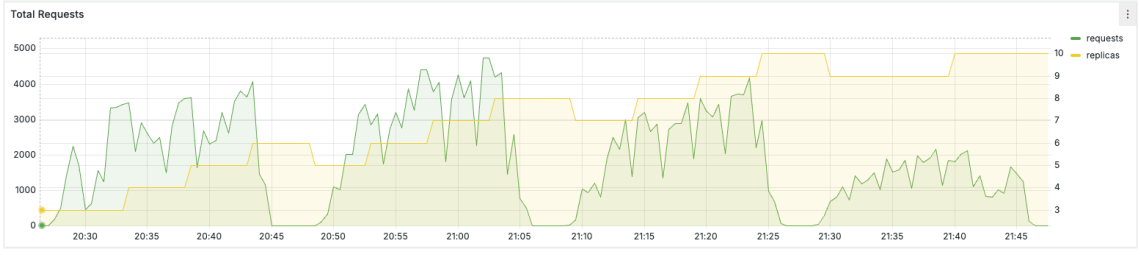
Σχήμα 5.20: *Εξερχόμενη κίνηση δικτύου Α΄.21, Α΄.22*

Παρατηρούμε ότι, βλ. Σχήματα 5.19, 5.20, 5.17, κατά την κλιμάκωση του συστήματος υπάρχουν άλματα στην κίνηση του δικτύου και τις εγγραφές, όπως παρατηρήσαμε κατά το προηγούμενο πείραμα, καθώς μεταφέρονται δεδομένα στους νέους κόμβους και αναπροσαρμόζονται στους προϋπάρχοντες κόμβους. Η κίνηση αυτή είναι μικρότερή στον φόρτο εργασίας λιγότερων σειρών καθώς μιλάμε για τάξη μεγέθους λιγότερα στοιχεία.

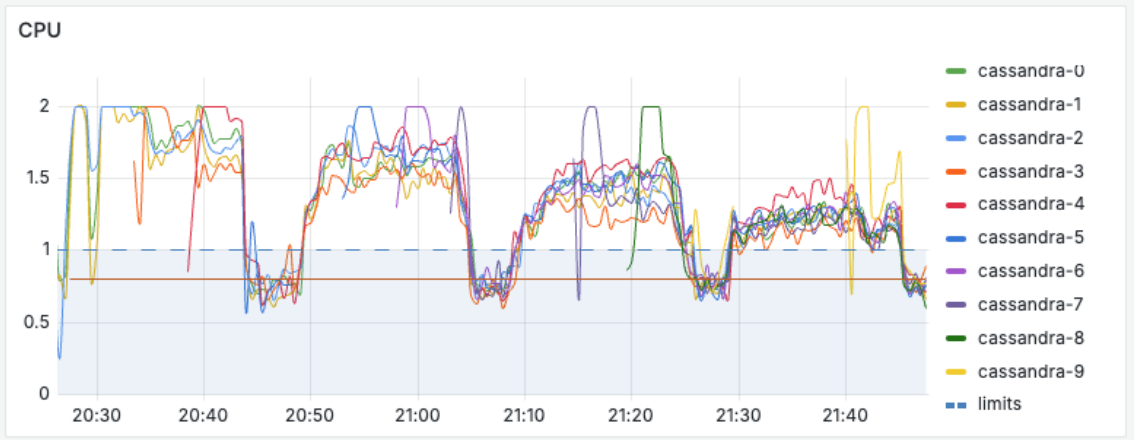
Τέλος, παρατηρούμε ότι, βλ. Σχήμα 5.15, κατά την κλιμάκωση η απόδοση του συστήματος πέφτει για περιορισμένο χρόνο. Αυτό οφείλεται στο ότι υπολογιστικοί πόροι του συστήματος δαπανώνται για την μεταφορά δεδομένων, βλ. Σχήματα 5.16, 5.17, 5.19, 5.20. Η απόδοση στον φόρτο εργασίας πολλών σειρών φθίνει στο τέλος καθώς πραγματοποιούνται πολλές αναγνώσεις από τον δίσκο, βλ. Σχήμα 5.16.

5.4 Ελαστικότητα - Κλιμάκωση με Ημιτονοειδές Σήμα

Στην ενότητα παραθέτουμε τα αποτελέσματά του Πειράματος Ελαστικότητας - Κλιμάκωση με Ημιτονοειδές Σήμα 4.4.



Σχήμα 5.21: *Ρυθμός Αιτημάτων Α΄.17 - Αριθμός Αντιγράφων Α΄.18*



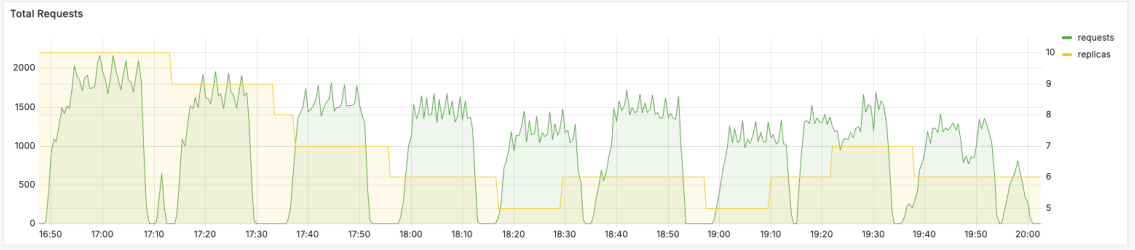
Σχήμα 5.22: *Χρήση Υπολογιστικών Πόρων Α΄.29, Α΄.30, Α΄.31, Α΄.32, Α΄.33*

Παρατηρούμε ότι, βλ. Σχήμα 5.21, η τροποποίηση μας στο YCSB παράγει ημιτονοειδή σήματα. Ακόμα, βλέπουμε, βλ. Σχήμα 5.22, ότι το αρχικό μέγεθος του συστήματος δεν είναι αρκετό να ανταπεξέλθει στο σήμα και κλιμακώνεται όπως και στα προηγούμενα πειράματα. Κατά τις ελάχιστες τιμές του σήματος το σύστημα αποκλιμακώνεται. Παράλληλα βλέπουμε στην τρίτη περίοδο μειωμένη απόδοση που οφείλεται στην κλιμάκωση και αποκλιμάκωση του συστήματος, κάθε τέτοια εργασία μειώνει στιγμιαία την απόδοση του συστήματος καθώς πόροι διατίθενται για την μεταφορά και αναπροσαρμογή των δεδομένων. Το πλάτος της τελευταίας περιόδου μπορεί να είναι μικρότερο από το ονομαστικό καθώς το YCSB δεν έχει στόχο να εκτελέσει όσες συναλλαγές εκτελούσε τις προηγούμενες περιόδους.

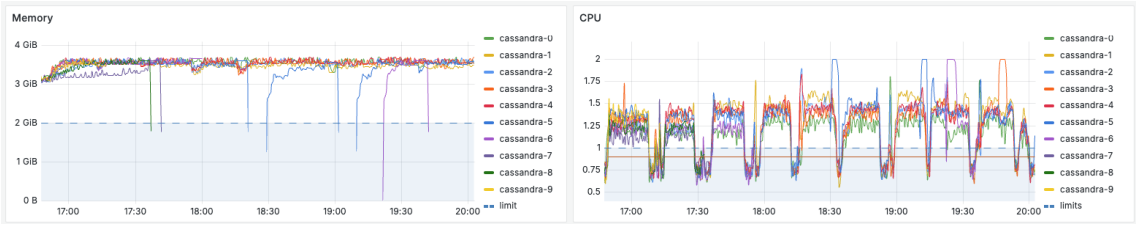
Τέλος, βλέπουμε ότι το σύστημα καταναλώνει σχεδόν πάντα πάνω από την τιμή που ορίζει ο HPA, παρόλα αυτά υπάρχουνε μεμονωμένες αποκλιμακώσεις.

5.5 Ελαστικότητα - Αποκλιμάκωση με Ημιτονοειδές Σήμα

Στην ενότητα παραθέτουμε τα αποτελέσματά του Πειράματος Ελαστικότητας - Αποκλιμάκωση με Ημιτονοειδές Σήμα 4.5.



Σχήμα 5.23: *Ρυθμός Αιτημάτων Α΄.17 - Αριθμός Αντιγράφων Α΄.18*



Σχήμα 5.24: *Χρήση Υπολογιστικών Πόρων Α΄.29, Α΄.30, Α΄.31, Α΄.32, Α΄.33*

Παρατηρούμε ότι, βλ. Σχήμα 5.23, το σύστημα αποκλιμακώνεται κάθε φορά που το σήμα μηδενίζεται, αλλά κλιμακώνεται όταν το σύστημα έχει αρκετό φόρτο. Παρατηρούμε ταλάντωση μεταξύ των τιμών 5 και 7 αντιγράφων (δεν είναι συγχρονισμένη με την περίοδο του σήματος) κρατώντας παρόμοια απόδοση. Αυτή η ταλάντωση οφείλεται στον τρόπο λειτουργίας του HPA που αποφασίζει ποια πράξη θα ακολουθήσει, κλιμάκωση ή αποκλιμάκωση, αν η χρήση είναι πάνω ή κάτω από το όριο αντίστοιχα.

Δεν μπορεί να δημιουργηθεί ένα εύρος υστέρησης όπου καμία πράξη (κλιμάκωση, αποκλιμάκωση) πραγματοποιείται. Δοκιμάσαμε να δημιουργήσουμε δύο HPAs που να συνδέονται με το σύστημα μας, ένα για κλιμάκωση και ένα για αποκλιμάκωση αλλά αυτό δημιουργεί σφάλμα καθώς μόνο ένα HPA μπορεί να συνδεθεί με έναν επίλεκτη εφαρμογής, βλ. Σχήμα 5.25.

```
Name:                                                  cassandra-hpa-cpu-sine-up
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Fri, 14 Feb 2025 15:48:24 +0000
Reference:                                             StatefulSet/cassandra
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  <unknown> / 100%
Min replicas:                                          3
Max replicas:                                          15
Behavior:
  Scale Up:
    Stabilization Window: 360 seconds
    Select Policy: Max
    Policies:
      - Type: Pods  Value: 1  Period: 360 seconds
  Scale Down:
    Select Policy: Max
    Policies:
      - Type: Percent  Value: 100  Period: 15 seconds
StatefulSet pods:    10 current / 0 desired
Conditions:
  Type         Status  Reason              Message
  ----         ------  ------              -------
  AbleToScale  True    SucceededGetScale   the HPA controller was able to get the target's current scale
  ScalingActive  False   AmbiguousSelector  pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-up default/cassandra-hpa-cpu-sine-down]
Events:
  Type     Reason                        Age                From                       Message
  ----     ------                        ----               ----                       -------
  Warning  FailedComputeMetricsReplicas  10m                horizontal-pod-autoscaler  pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-down default/cassandra-hpa-cpu-sine-up]
  Warning  FailedComputeMetricsReplicas  9m6s (x11 over 11m)  horizontal-pod-autoscaler  pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-down default/cassandra-hpa-cpu-sine-up]
```

Σχήμα 5.25: *Μήνυμα Σφάλματος για πολλούς HPA*

Να σημειωθεί ότι κατά την διάρκεια ανάπτυξης των πειραμάτων πραγματοποιήθηκαν εκκινήσεις στις εικονικές μηχανές, και μετά από κάθε τέτοια διαδικασία το σύστημα διατηρούσε την αναμενόμενη κατάσταση λόγω των δυνατοτήτων αυτοΐασης του Kubernetes.

5.6 Σύνοψη Αποτελεσμάτων

Με τα παραπάνω πειράματα παρατηρήσαμε πως ότι η συμπεριφορά του συστήματος ανταποκρίθηκε κατά βάση στα αναμενόμενα αποτελέσματα όπως αυτά περιγράφηκαν στο θεωρητικό μέρος. Ακόμα, παρατηρήσαμε την ελαστική λειτουργιά της Apache Cassandra χρησιμοποιώντας εγγενή στοιχεία του Kubernetes με ελάχιστες τροποποιήσεις. Παρά τον περιορισμό στους πόρους για κάθε Pod, έχουμε την αναμενόμενη γραμμική απόδοση. Να επισημανθεί οτι χρησιμοποιήσαμε το αντίστοιχο t2.small έναντι του t2.large που προτείνεται για περιβάλλοντα μη φόρτου [107], [108].

Μέρος IV

Επίλογος

Κεφάλαιο 6

Επίλογος

6.1 Συμπεράσματα

Σε αυτό το σημείο θα συνοψίσουμε τα βασικά συμπεράσματα της παρούσας διπλωματικής εργασίας καθώς προκύπτει από την θεωρία και τα πειράματα.

Ο Kubernetes έχει την ικανότητα να αυτοματοποιεί την διάθεση και διαχείριση του κατανεμημένου συστήματος Apache Cassandra χωρίς την χειροκίνητη διαμόρφωση που θα απαιτούσε ένα τέτοιο σύστημα. Ουσιαστικά δηλώνοντας μία Storage Class Α΄.2, ένα Statefulset Α΄.6, Α΄.7, Α΄.8, έναν αριθμό από PVs Α΄.3, και μία υπηρεσία Α΄.4 καταφέραμε να στήσουμε την Apache Cassandra. Πρακτικά με τον ορισμό της υπηρεσίας και μερικών πορτών επιτύχαμε την ομαλή δικτυακή λειτουργία και τη σύνδεση δεκαπέντε κόμβων. Παρατηρήσαμε και την δυνατότητα αυτοϊασης του Kubernetes στις επανακκινήσεις των εικονικών μηχανών.

Ο Kubernetes, όντας αγνωστική πλατφόρμα, μας επιτρέπει να μεταφέρουμε τις δηλώσεις σε οποιαδήποτε πλατφόρμα υπολογιστικού νέφους εργαζόμαστε χωρίς να χρειάζεται προσαρμοσμένες διαμορφώσεις ειδικές για κάθε πλατφόρμα. Αυτό είναι ένα ιδιαίτερα σημαντικό χαρακτηριστικό καθώς προστατεύει τους χρήστες από δυνητική δέσμευση προμηθευτή. Χρησιμοποιήσαμε εργαλεία και επεκτάσεις ανοιχτού κώδικα (Prometheus, Metrics-server) που μας επέτρεψαν να κινηθούμε ευέλικτα και εύκολα, γεγονός που συνδράμει στην ευκολία χρήσης. Με ελάχιστο κώδικα Α΄.9 καταφέραμε να υλοποιήσουμε δυναμική παροχή πόρων, γεγονός που αποδεικνύει την επεκτασιμότητα του Kubernetes ως τεχνολογία ανοιχτού κώδικα. Διαπιστώσαμε τους περιορισμούς στη λειτουργία του HPA καθώς αποτελεί μη ευέλικτο αντικείμενο για πολύ δυναμικά δεδομένα.

Παρατηρήσαμε ότι παρά τους περιορισμένους υπολογιστικούς πόρους, υλοποιήσαμε ένα σύστημα που αντιδρά ελαστικά στον φόρτο εργασίας. Κλιμακώνεται οριζόντια με γραμμικό τρόπο για να ανταπεξέλθει στην ζήτηση και κατάφερε να αποδώσει, όπως αναμενόταν, χωρίς σφάλματα. Αυτή η μεθοδολογία χρησιμοποιείται ευρέα με αρχιτεκτονικές Microservices, όπου κλιμακώνονται οριζόντια, απομακρύνοντας μας από μονολιθικά συστήματα.

Μέσω των πειραμάτων παρατηρήσαμε και κατανοήσαμε πως σχεδιάζεται ένα κατανεμημένο υπολογιστικό σύστημα με γνώμονα την υψηλή διαθεσιμότητα, αντοχή σε κατατμήσεις δικτύου και ελαστικότητα.

Είναι δε εντυπωσιακό το γεγονός ότι τεχνολογίες ανοιχτού κώδικα δίνουν σε κάθε χρήστη την δυνατότητα σχεδιασμού και υλοποίησης ισχυρών συστημάτων.

6.2 Μελλοντικές Επεκτάσεις

- Η επανεκτέλεση των πειραμάτων με περισσότερους υπολογιστικούς πόρους ώστε να προβούμε σε συγκριτικά συμπεράσματα.
- Η εξέταση διαφορετικών βάσεων δεδομένων, με διαφορετικές εγγυήσεις με τη παρούσα διαμόρφωση του συστήματος ώστε να συγκρίνουμε την απόδοση και λειτουργία μεταξύ τους.
- Η υλοποίηση ενός κλιμακωτή με περισσότερα χαρακτηριστικά, περιοχή υστέρησης, συγκερασμός πολλαπλών μετρικών, δυναμική συμπεριφορά βάση φόρτου.
- Η παρούσα εργασία μπορεί να λειτουργήσει ως βάση για αναζήτηση θεμάτων ελαστικής διαχείρισης υπολογιστικών νεφών (πχ. ελαστικότητα με χρήση δυναμικού προγραμματισμού, τεχνολογιών μηχανικής μάθησης, κ.ο.κ.).

Μέρος V

Introduction

Κεφάλαιο 7

Introduction

The rapid development of the internet has led to a continuous increase in the number of data generated. This kind of data is usually described by the term Big Data, with their large volume, their speed of production, and their high variability in structure and source being some of the core characteristics associated with them [6]. This dynamic behavior necessitates the utilization of alternative technologies and infrastructure requirements, one example of which are the cloud computing environments. Cloud computing environments are services that offer computing and storage capabilities dependent on the demand. This dynamic supply of resources allows for the management of Big Data in a different way to what traditional on-premises centers facilitate. We have the emergence of new systems of data storage and management allowing improved efficiency, like Apache Hadoop from Dean, and Ghemawat [7].

7.1 Subject of the Study

Inspired by these developments in computing, we will examine within this paper the elastic data management in cloud computing environments. Each cloud environment usually offers its own abstractions over compute and does not allow cloud migration, otherwise referred to as vendor lock in [8]. This has changed with the release of Kubernetes[9], which is platform agnostic [10]. We will examine how we can use Kubernetes abstractions and objects so that we can implement a cloud agnostic system that can manage large volumes of data efficiently and in an elastic way. We will also examine the distributed, non relational database management system Apache Cassandra from Lakshman and Malik [11], as well as a series of tools and programs that will aid in the completion of this study and paper.

This paper will start with examining the theoretical foundation of various fields and technologies, which we will then utilize in order to perform experiments of data management to assess elasticity.

Μέρος VI

Theory

Κεφάλαιο 8

Theory

In this chapter we present in detail the four basic technologies that were utilized for the purpose of this paper. Those are: the cloud orchestration system Kubernetes, the data management system Apache Cassandra, the benchmark tool YCSB, and the monitoring tool Prometheus.

8.1 Kubernetes

8.1.1 Defining Kubernetes

Kubernetes is an open source container orchestration platform that was developed and designed to automate the deployment and management of computing systems. Google initially developed and released it in 2014, and afterwards it is maintained by the Cloud Native Computing Foundation (CNCF) [12]. It is the successor of container deployment applications, simplifying challenges such as synchronization, release synchronization, scaling, load balancing and container restart [9].

Kubernetes allows its users to manage its applications and systems regardless of whether those are hosted on on premises centers or cloud environments [10] using a declarative API [12]. The administrators and users declare the desired state and Kubernetes performs all the actions to achieve it.

Kubernetes consists of two types of components, the Control Plane components and the Node components [1].

Control Plane Components

The Control plane components make decisions for the whole system as they manage the cluster events. Usually, those components are hosted on the same node and it is advised to not run other workloads in order to ensure the efficient operation of Kubernetes [1].

Those components are:

- API Server: exposes the Kubernetes API and manages the iterations between the cluster, the system and the users [13].

- etcd: a distributed key-value storage that stores the state and the configuration of the cluster, ensuring system consistency [14].

- Scheduler: is responsible for the scheduling and assignment of new Pods and workloads to the appropriate nodes[15].

- Controller Manager: runs management jobs that monitor the state of the cluster, the nodes, and the workload, taking all the necessary actions to maintain the desired state such as scaling, replication, and restarts[16].

In commercial cloud computing environments there is also a Cloud Controller manager that consolidates the cloud native services with Kubernetes, such as Frontend UIs, dashboards, and dynamic volume provisioning systems [17].

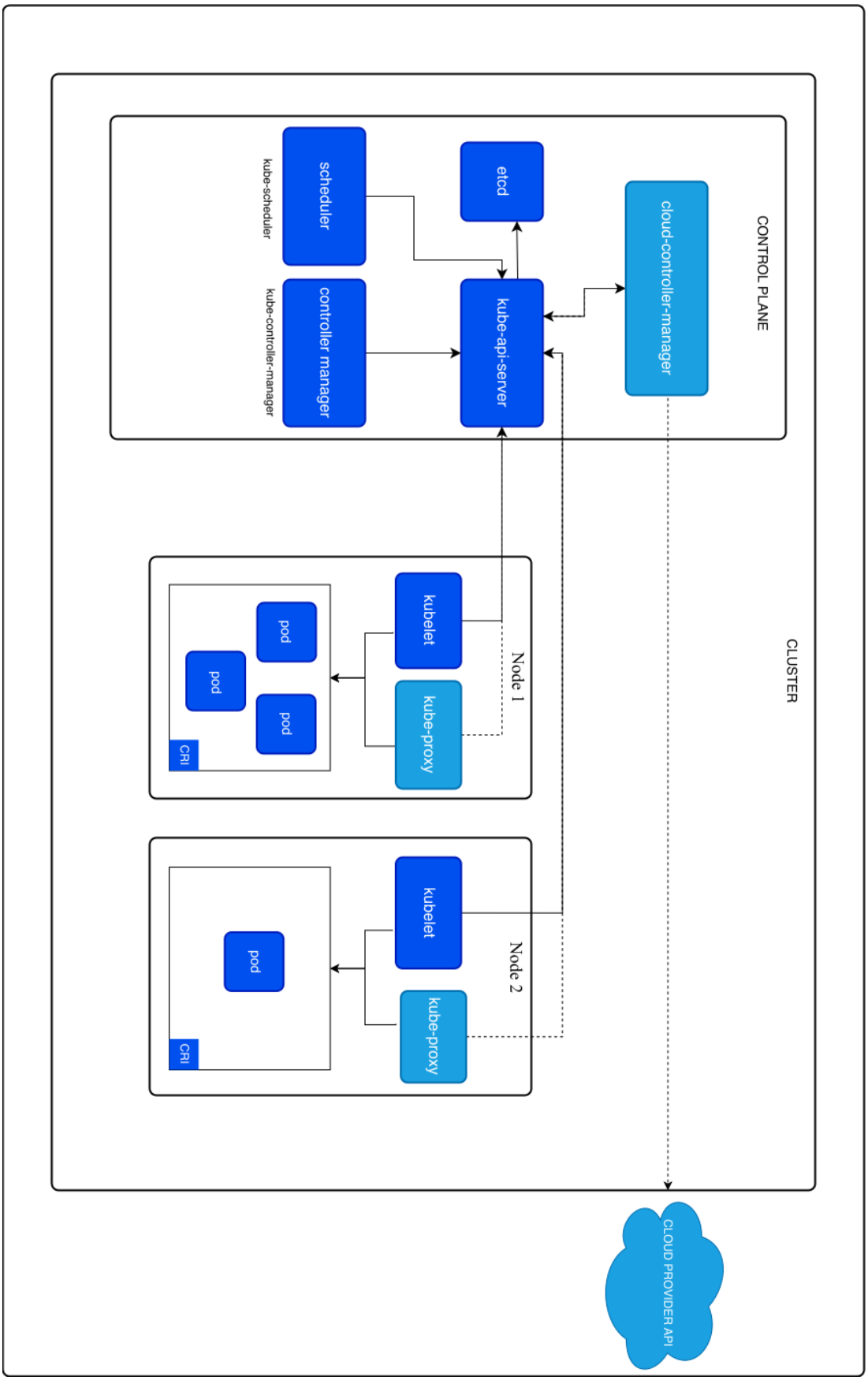
Node Components

Node components run in each node and also run all the necessary actions for the smooth workload execution [1]. Those components are:

- Kubelet: is executed on each node and ensures that the workloads are executed in allignment with the desired state that is defined by the control plane [18].

- Kube-proxy: network rules administrator, ensuring the communication between the Pods and Services [19].

- Container Runtime: executes the Containers and manages their lifecycle, some examples are Docker, containerd, or CRI-O [20].



Σχήμα 8.1: *Kubernetes Architecture [1]*

8.1.2 Pods

Pods are a collection of Linux services (namespaces, cgroups) that aim in the isolation of compute [21]. It is the smallest unit of compute that Kubernetes offers [22]. It consists of one or more Containers (unusually one), that manages common computing, network resources, volumes, and offers isolation [23]. Pods are used in two ways, with one Container or with more than one [24]. In the case of one Container, the most common configuration, is that the Pod acts as a shell around it, offering isolation. More interesting is the configuration with more than one Containers as it ensures that the Containers will be hosted on the same Node sharing resources, and emulating the operation within the same host. To demonstrate, some of the design patterns with more than one Container are the following:

- Tight Container coupling through some node collocation, where they can share resources for a more efficient communication, operation and synchronization [24].
- Sidecar prototype, by which the container's functionality can be extended by the addition of another. Some of the extension points are the logging, monitoring and proxying [24].
- Auxilary function, that changes the configuration of the main Container during its lifecycle (init container [25], termination) [24].

It is important to note that the operation of a Pod is ephemeral [22], [26], and upon termination, scheduled or unexpected, then all data and volumes are deleted. This characteristic makes them ideal for REST applications, but not suitable for applications where state and data maintenance are required (i.e. databases).

Pods rarely exist by themselves in production environments, they are composite elements of other workload resources [27]:

- Deployment allows the Pods to scale and update, guaranteeing available and rolling updates and deployments [28].
- DaemonSet ensures that a Pod runs on either all or specific cluster nodes, useful for system daemons and monitoring agents [29].
- StatefulSet manages applications with sticky identity and persisting resources, ideal for databases and distributed systems [30]. We will elaborate further in a consequent chapter.

8.1.3 Persistent Volumes

As previously mentioned, Chapter 8.1.2, the Pods' volumes and states are ephemeral [22], [26]. To overcome this limitation, we have Persistent Volumes [31]. Persistent Volumes are a core component for storage management in Kubernetes. It is a cluster resource, the same way that a Node is. It exists regardless of the Pod's life cycle or the Nodes, ensuring the data persist even in the instances of the system or application crashing or restarting [31].

There are two ways to be bound, static or dynamic [32]. For the static binding, the Kubernetes administrator creates and manages the PVs manually, which then is used by Kubernetes. The dynamic binding is facilitated by storage classes [33] and PVCs [34]. Depending on the desired behavior, Kubernetes can bind, unbind, and repurpose volumes on the Nodes and Pods according to the system specification without manual intervention from the administrator.

PVs have a wide configuration range. Besides the storage size, some of the configuration categories are the following:

Volume Mode

The administrator can select the volume mode between the options of file system or block storage [35].

Access Modes

- ReadWriteOnce, the volume is located in a single node and only Pods from this node can write and read data.
- ReadOnlyMany, the volume can be read by many nodes.
- ReadWriteMany, the volume can be written by and read by many nodes.
- ReadWriteOncePod, the volume is located on a single node and only one Pod can write and read it. This configuration is Beta in version 1.27.

[36]

Storage Class

Storage Class allows administrators to describe volume storage groups in the system [37], [33]. The administrator selects the PV provisioner (AzureFile, CephFS, local, NFS, RDB etc.) [38]. At the same time can configure the reclaim policy after the end of its use, its deletion or retention [39]. One important configuration is the Volume binding mode, which is either immediate or WaitForFirstConsumer [40]. The immediate binds the volume during the creation of the PVC, in environments with topological restrictions this leads to the PV binding before prior knowledge of which Node the Pod will be scheduled on, an event that can result in an unscheduled Pod that has no access to the PV. This limitation is overcome by the second configuration that awaits the Pod creation prior to the PV binding. Last but not least, there is the possibility of configuring allowed topologies [41].

PV Reclaim Policy

When the Pod does not use the PV, the reclaim policy defines what will happen to the Volume, in other words, whether it is going to be retained or deleted. In the past, there was the option to recycle it but this functionality has been deprecated [42].

Node Affinity

The PV can impose limitations on which Node it will be mounted on. The Pods that use the specific PV can only be scheduled at the specified Node [43].

8.1.4 Deployments

Deployments manage a set of Pods that run a stateless workload. They are usually used for rolling updates and deployment of new editions of the Pods. Deployment uses a replicaset, and consequently the replicaset a number of Pods that are exact copies. Each copy performs the same functionality in the cluster without any semantic difference between them [28].

8.1.5 StatefulSets

Statefulsets are objects that enable the user to persist Pods' state in Kubernetes [30]. They manage deployment and scaling for a set of Pods that have the same specifications. The differentiating factor between a Statefulset and a deployment is that Pods maintain a sticky identity, in other words the Pods are analogous but not the same [44]. Nevertheless, they have a unique indexed name, and network identifier. For instance for the Statefulset my-app the first replica is my-app-0 [44], [45]. Every replica is created in an ascending order starting from the lowest and thereafter increasing, and vice versa when replicas are deleted.

Statefulsets also come with limitations. For the data and state persistence, PVCs need to occur. Scaling down does not automatically delete PVCs. We need to create a headless service for the Pods' network identity. To conclude, the deletion of the statefulset does not guarantee the deletion of all the Pods, therefore it is advised to scale down to zero the replicas and then delete them [46].

For the configuration of the Statefulset, besides the Pod template we have to define the number of replicas, the headless service, the storage class template that will be used by the PVCs, as well as the PVC reclaim policy [30]. This policy defines what happens to the PVCs after Pods are scaled down and deleted, in particular whether they will be stored or deleted [47].

8.1.6 Scaling

Scaling is the process during which the computing capabilities of a system change. This can happen by either horizontal or vertical scaling [48]. Horizontal scaling is when we increase the compute power by increasing the number of computers. This differs from the vertical scaling where computing capabilities are increased by increasing memory, CPU power, storage capacity without the addition of computers [49]. These two different strategies each have their own advantages and disadvantages, and the decision is made depending on the system's needs [50].

Vertical scaling is selected in the case of:

- Less critical systems and workloads.

- Systems where workload is not expected to increase in the future.
- Attempt to decrease initial costs.
- Systems that do not require high availability, throughput and multi-region deployment.

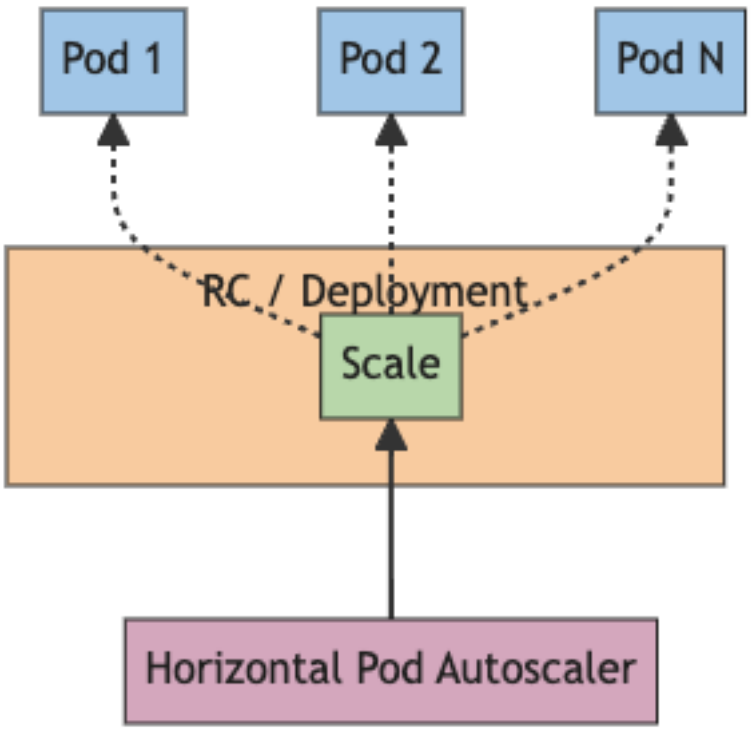
Horizontal scaling is selected in the case of:

- Critical systems and workloads.
- Systems where workload is expected to increase in the future.
- Systems that require high availability, throughput and multi-region deployment.

[51]

8.1.7 Horizontal Pod Autoscaler

Kubernetes offers objects for horizontal [52] and vertical automated scaling [53]. Nonetheless, we will focus on the Horizontal Pod Autoscaler. HPA is a Kubernetes component that allows workloads to scale based on demand. It can only be attached to objects that scale, Deployments and Statefulsets [52].



Σχήμα 8.2: *HPA controlling Deployment [2]*

During the configuration of HPA we select the attached object, the maximum and minimum number of replicas, the monitored metrics, and the desired scaling up and

down behavior. It is not allowed for more than one HPAs to attached to the same object [52].

The way HPA works is to regularly monitor specific metrics (the default value is 15 seconds) and depending on the use of this resource HPA follows predefined actions [2]. The algorithm that is followed is $desiredReplicas = (currentReplicas * currentMetricValue / desiredMetricValue)$ [2].

The metrics that can be selected are the Pod's usage metrics (memory, CPU) [54] or custom metrics (requests per second, failure rate, active connections, etc.) [55]. Additionally different metrics can be combined under the same HPA [56]. It is important to note that HPA averages the usage for all the replicas, there is the possibility that a Pod is over the limit but no scaling action is taken because the average usage is within limits [54].

The behavior defines the way of scaling up and down by defining policies with allowed actions and the period in which every policy has effect [57]. When there are more than one policies, the one with the larger rate of change is selected [57]. The algorithm logic 8.1.7 combined with the actions [57] allows for the binary operation of the system where the number of the replicas can oscillate between a range of values. This can be restricted by the selection of a stabilization window [58] that does not allow the system to perform an opposite action in the specified period.

HPA has limitations according to Jiang and Wu [59]:

- The scaling algorithm is simplistic and not flexible.
- Stabilization windows [58] do not allow for the further scaling of the system.
- The collection period of the metrics cannot be altered dynamically between higher and lower load periods.

Additionally, there can not be defined a hysteresis region, where the system does not scale up of down the number of Pods.

8.2 Apache Cassandra

Apache Cassandra is a distributed, non relational database management system. It was developed in 2008 as an open source system from Lakshman and Malik [11], [60] for searching Facebook messages, and was released by the Apache Software Foundation in 2010. It was designed with the goal of managing large volumes of data, while offering scalability and availability, rather than data consistency. The main characteristics are listed below [60], also in accordance to Chrysopoulos [61]:

- Multiple node selection in different data centers.
- Distributed operation: all the nodes are similar, without having a master node, executing the same functionality through Peer-to-Peer protocol communication.
- Fault tolerance: data are replicated in various nodes and data centers, guaranteeing against single point failures.

- Elasticity and linear expansion: the addition of new nodes does not stop the system's operation and increases linearly its throughput.
- Tunable Consistency: the system ensures eventual consistency and each transaction can have different consistency level. This configuration influences how many nodes need to acknowledge the transaction successfully.

8.2.1 Data Model

Apache Cassandra's Data Model is non relational and consists of terms that resemble traditional Database Management Systems [62].

Keyspaces

Keyspace defines the way data are replicated, that is, in how many replicas. It is essentially a container of tables.

Tables

Tables consist of columns and lines. Columns describe the shape of data and their types. Columns can be extended with no down-time [62].

Partition

Partition defines the range of lines of the partition key that belong to each node [62].

Lines

Line is a data tuple that is recognized by a primary key. The primary key is composed by the partition key and optionally other columns [62].

Column

Column is the chunk of typed data, the smallest unit [62].

CQL

Apache Cassandra uses the Cassandra Query Language (CQL), inspired by the Structured Query Lanaguage (SQL). We observe that some of the Apache Cassandra components correspond to those of traditional RDMSs, therefore CQL is not unfamiliar [62], [63].

Denormalization

In Apache Cassandra the data are denormalized, which contrasts with RDBMSs as it does not allow Joins, because the related data are located in the same table [62].

8.2.2 Amazon Dynamo

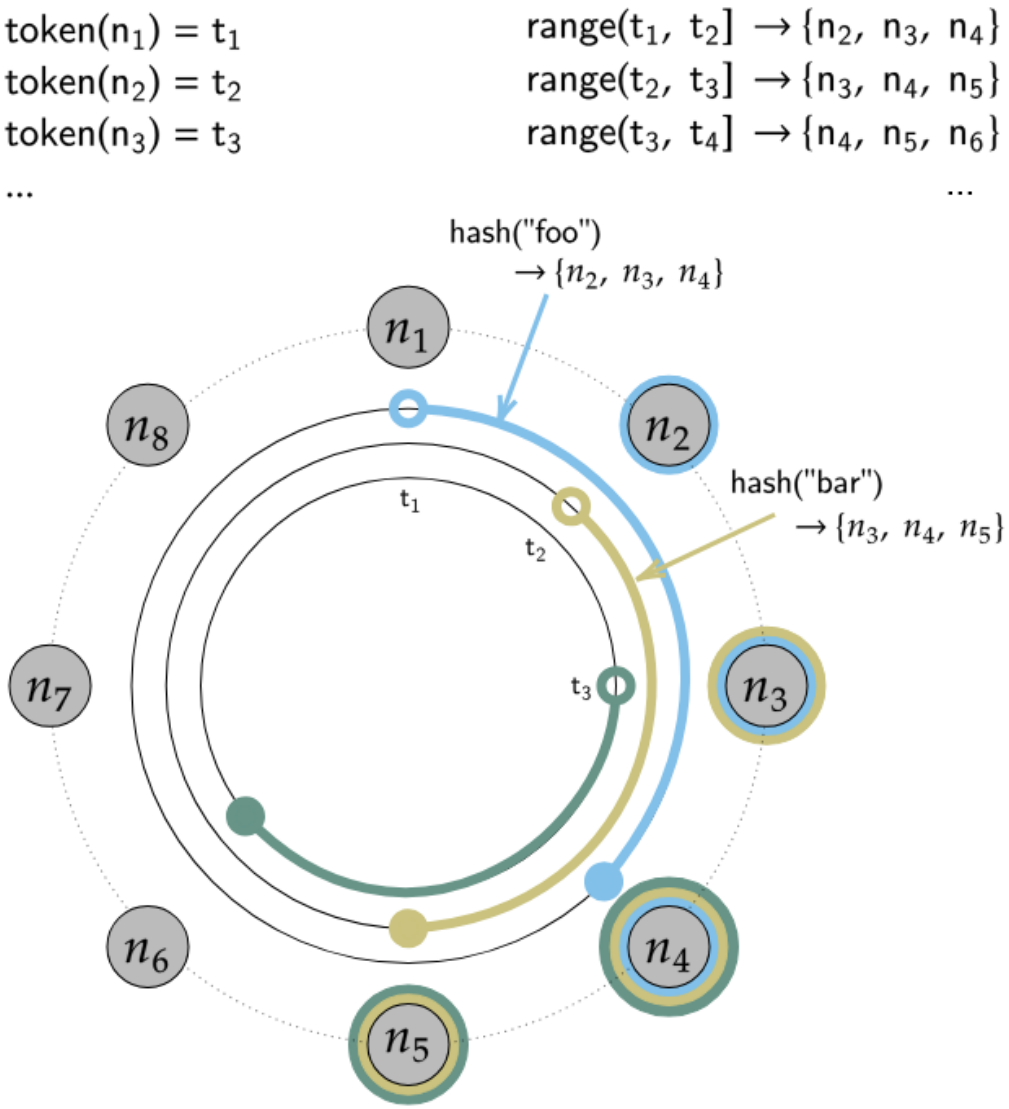
Appache Cassandra's functionality is based on the distributed key-value storage system from DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall and Vogels [64], [65]. Its main elements are:

- Request coordination over a partitioned dataset
- Replication Policy
- Tunable Consistency
- Cluster Membership and Failure Detection

[64], [65]

Request coordination over a partitioned dataset

For achieving horizontal scaling, Apache Cassandra distributes the data on the Nodes based on a consistent hashing function [66]. Data is distributed in a continuous ring based on their partition key. The Nodes, represented as tokens, belong to the ring and store key ranges after their tokens [3]. Depending on the replicas number, data are replicated to following Nodes in order to achieve fault tolerant functionality. In figure 8.3 we observe node 4 owning key ranges t1, t2, t3 and t4. If node 4 crashes we observe that data exists in following nodes so the system can continue working without any data loss.



Σχήμα 8.3: *Cassandra Token Ring [3]*

The data distribution balancing works optimally when we have enough Nodes, as these are distributed uniformly through out the ring [3]. This does not necessarily happen for a lower number of nodes. Amazon Dynamo overcomes this challenge with the creation of virtual node tokens that are distributed more evenly on the ring [67].

Replication Strategy

Every Keyspace defines its replication strategy, specifying the way by which data are replicated, It can either be NetworkTopologyStrategy , that determines how data are copied between different data centers or SimpleStrategy that describes how many nodes have the replicas of the data. It is advised that within production environments NetworkTopologyStrategy is used and within development environments we utilize SimpleStrategy [68].

Tunable Consistency

The CAP Theorem of Brewer[69] is one of the most important theorems in the distributed systems world, according to Gilbert and Lynch [70]. It recognizes three characteristics that govern every such system [69], [70], [71]:

- Consistency: every read request in each node returns the most recent write or fails.
- Availability: every request receives a successful response, without necessarily being the most recent write in the system.
- Partition Tolerance: the ability of the system to operate even in the event of a disruption in the communication of the nodes.

The theorem [69], [70], [71] states that a distributed system cannot have all three characteristics at the same time, but only two. This means that a Partition Tolerant system cannot ensure Consistency and Availability at the same, as a request will receive the latest write or fail and at the same time cannot receive a response that is not a failure.

Apache Cassandra offers the following guarantees[72]:

- High Scalability [73]: the system can add or remove nodes using the Gossip protocol for the efficient node membership.
- High Availability [74]: using a fault tolerant storage engine.
- Durability [75]: data are replicated at different nodes and data centers that ensure that with the loss of one node there are enough copies of the data.
- Eventual Consistency [76]: writes will eventually exist in every replica. Different versions of the data will exist in different nodes until there is a final consistency.
- Lightweight Transaction [77]: are executed sequentially using the Paxos consensus protocol, ensuring consistency among replicas by using the Compare and Set operation, allowing read transactions to only read values without changing them.
- Batched writes in multiple tables [78]: either all of them succeed or all of them fail.
- Secondary indexes that allow efficient column searches is guaranteed to be consistent among local replicas [79].

Apache Cassandra uses tunable consistency for every transaction. The user selects the desired degree of consistency [76]. Higher degree of consistency affects negatively the Availability as the system waits for more replicas to acknowledge the transaction [80]. We list the consistency configurations [80], [81], [82]:

- One replica
- Two replicas
- Three replicas

- Quorum, the majority of the replicas
- All the replicas
- Local quorum, the majority of replicas from a specific data center (the data center where the request was made)
- Each quorum, the majority of every data center
- Local only, this configuration ensures that the read request is not transfered to a different data center
- Every, only one replica responds or the proxy node stores a hint. The proxy will eventually try to transfer the change to the replicas. This configuration is only supported for read requests.

Ring Membership and Failure Detection

For the Data Partition and Replication Strategy the system needs to know the nodes' health so that it routes the read and write requests efficiently [83]. This information is transferred in a distributed way using the Gossip protocol and the Failure Detection.

The Gossip protocol [84] is a system where the nodes share information regarding their health but also of the health of the nodes they know. They use tuples that have the monotonic time and version of the information, allowing nodes to use the most recent version of the Gossip and fault detection. Every node periodically:

1. updates its health status and creates a local view of the cluster Gossip that it knows.
2. selects a random node and exchanges Gossip information.
3. tries to exchange Gossip information with an unreachable node (if it exists) in a probabilistic way.
4. Gossips with a seed node if it did not communicate with it during step two.

[84]

The seed node is the initial node and the only difference between it and the other nodes is that it does not need other nodes in order to connect to the ring. Because of step 4 these nodes have higher usage during Gossip.

For Failure detection [83] Apache Cassandra uses a modified version of Phi Accrual Failure Detector of D?fago, Hayashibara, Yared, Rami and Katayama [85], where each node decides that some other node is down if it does not receive a healthy state for a specific time period. It will then not forward the read transactions to the down node and wait for a healthy state in order to send the write transactions.

8.2.3 Storage Engine

Apache Cassandra uses a distributed storage engine that allows for fast reads and writes [86]. There are two main request flows; write and read flows.

Write Request Flow

During the write request flow [87], each incoming transaction is written in a Commitlog in every node, an append only log file. The writes are written in chuncks allowing for efficient disk write. This write allows the data persistence during crashes [88]. Following, the data are stored in a Memtable, a table stored on the heap, implemented as a balanced tree or a skip list [89]. The node acknowledges the request only when it is written in the Commitlog and the Memtable [87]. When the Memtable grows larger than a specified size, it is stored in immutable files called SSTables that are sorted based on the partition key. There is the possibility of transferring SSTables to other nodes for data balancing [90]. Apache Cassandra compacts the SSTables for a more efficient system operation [87].

Read Request Flow

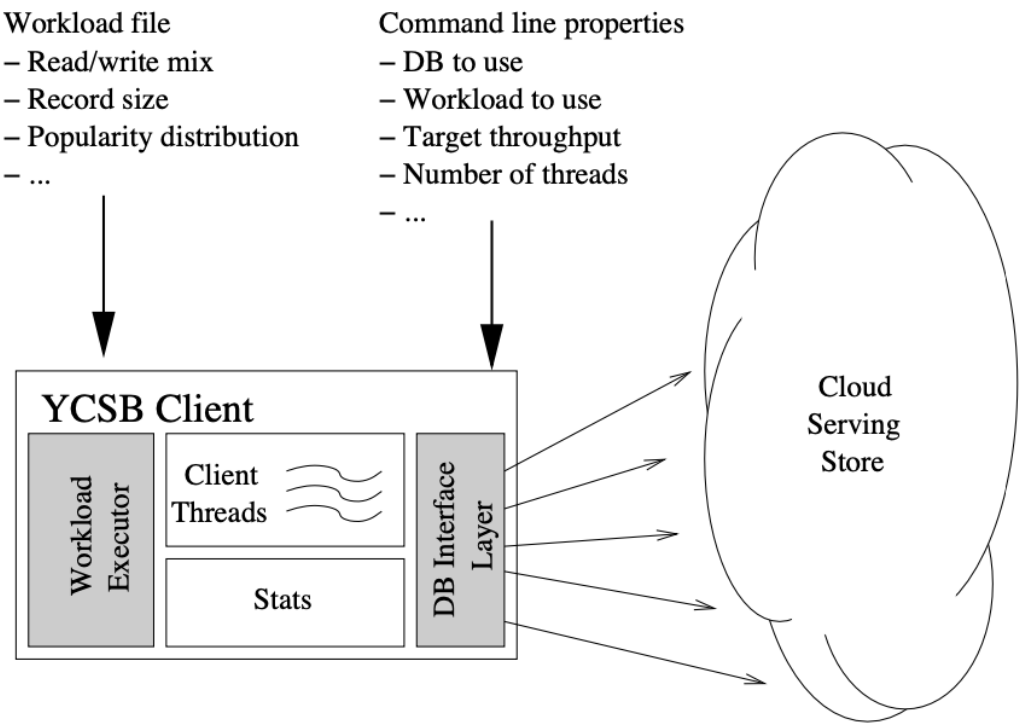
During the read request flow [91], each incoming transaction reads the Memtable and if this request does not return a value, then it reads the SSTables from the disk. Due to the sorted nature of these files, Apache Cassandra can retrieve the data efficiently using partition keys, groupings and Bloom filters [91], [90]. Finally, it merges all the data and returns the newer version of the value.

We see that the Apache Cassandra storage engine offers high read and write throughput, crash durability and a high degree of configuration and modifiability as every mentioned element can be tuned [11], [86].

8.3 YCSB

YCSB is an open source database benchmark program. In 2010 it was developed by Cooper, Silberstein, Tam, Ramakrishnan and Sears at Yahoo [4] for evaluating the performance of novel database systems, usually not RDBMSs. The three elements that YCSB monitors is Elasticity (ability of the system to load new nodes during operation), High Availability (ability of the system to operate through crashes), and Horizontal Scaling (ability of the system to handle large volume of data with the addition of commodity servers and not vertical scaling) [4].

YCSB is a Java program [92] that uses two types of configuration, workload and database interface. The Client spawns ClientThreads, figure 8.4, defines the workload of the benchmark and additionally the database interfaces define which transactions will be executed on the database as there are different transactions among different databases. Lastly, transactions are executed in a database specified way [93].



Σχήμα 8.4: *YCSB Architecture [4]*

The workload configuration concerns the [93]:

- Number of Threads
- Target
- Ratio of the executed transactions (read, write, update, scan)
- Desired Transaction Consistency
- Record count
- Operation count
- Record distribution

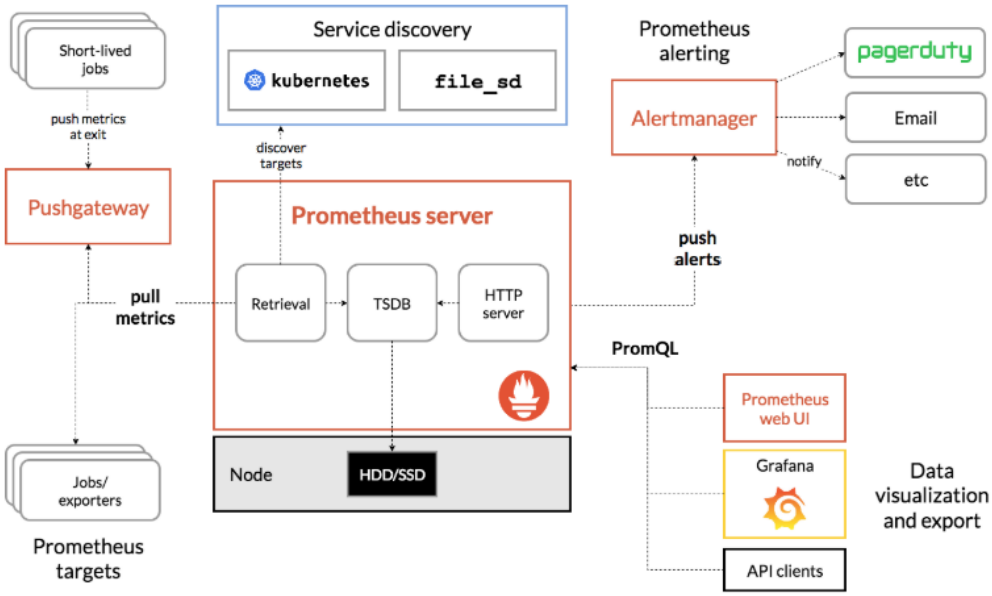
The database interface configuration concerns the way in which YCSB connects to the database, as well as specific database configuration [93].

8.4 Prometheus

Prometheus is a monitoring and notification open source system that was developed by Soundcloud in 2012 [5]. In 2016 it was the second hosted project of CNCF, following Kubernetes as it was adopted by a large number of companies and organizations. It collects and stores metrics as time series data, in other words data accompanied by a timestamp. The main characteristics of Prometheus are the following:

- Time series multi-dimensional data model
- A flexible query language (PromQL)
- Pull based collection model, using the HTTP protocol, allowing for monitoring without the need to modify the monitored application.
- Each Prometheus server's data is stored locally without being distributed in other nodes. This frees it from needing to synchronize transactions, simplifying its architecture.

[5]



Σχήμα 8.5: *Prometheus Architecture [5]*

The constituent components of Prometheus are, figure 8.5:

- The main Prometheus server that collects the data and stores them in a time series database (TSDB) [5].
- The Push gateway for short lived jobs [5].
- The specific exporters, developed for different systems and databases like [5], JMX Exporter, Elsticsearch, PostgreSQL, Redis [94].
- The Alert Manager [5].
- The auxiliary UI like Graphana, Prometheus Web UI [5].

Grafana is an open source program that allows metrics querying, visualization, alerting and investigation from different sources, one of them is Prometheus [95].

Μέρος VII

Practical Study

Κεφάλαιο 9

Description of the Implementation

This chapter covers the implementation details of the study. It describes the decisions that supported the selection of the respective technologies and the main limitations governing the implementation.

The goal is to examine the elasticity of a distributed database system, in a cloud computing Kubernetes environment. To achieve this, we have to install a Kubernetes environment, enable Kubernetes to dynamically provision volumes, start Apache Cassandra, design the horizontal auto scaling policy, configure the monitoring metrics of the system, and utilize the YCSB benchmark.

9.1 Kubernetes Installation and Configuration

The computers composing the Kubernetes cluster are virtual machines that were provisioned by the computing cloud of the Computer Systems Laboratory of the Electrical and Computer Engineering School of the National Technical University of Athens. Six virtual machines were provisioned (node1-node6), which are connected to the same network, and all machines are hosted on different nodes of the cloud. This allows for the system to be less susceptible to crashes and resource starvation as each virtual machine's operation is independent from the others.

Host	Όνομα	IP
cloud1	node3	10.0.1.122
cloud3	node2	10.0.1.121
cloud4	node4	10.0.1.128
cloud6	node1	10.0.1.102
cloud5	node6	10.0.1.136
cloud7	node5	10.0.1.125

Πίνακας 9.1: *Node Topology*

There are many options to install and configure Kubernetes:

1. Minikube [96]
2. MicroK8s [97]
3. Kubeadm [98]

4. Kubespray [99]

We selected Kybespray as the first two options are ideal for educational and local installations [96], [97], the third option is for production environments [98], whereas Kubespray is flexible enough to support ease of cluster installation regardless of the environment and offers reliability in terms of its production capabilities [99]. Since the 2.3 version Kubespray started using Kubeadm for the internal system operations for the cluster creation [100].

We are using version 2.23.2 of Kubespray, and installing version 1.27 of Kubernetes, selections that offer a stable installation [101].

Algorithm A′.1:

1. We copy the cluster template.

2. We define the node IP Addresses .

3. We specify the configuration file and create the Ansible inventory.

4. We execute the Ansible playbook that installs the cluster.

After the installation we can inspect the state of the cluster by executing the command *kubectl get nodes*. The result will give us the state of the cluster.

Name	Status	Roles	Age	Version
node1	Ready	control-plane	127d	v1.27.10
node2	Ready	control-plane	127d	v1.27.10
node3	Ready	<none>	127d	v1.27.10
node4	Ready	<none>	127d	v1.27.10
node5	Ready	<none>	127d	v1.27.10
node6	Ready	<none>	127d	v1.27.10

Πίνακας 9.2: *Cluster state*

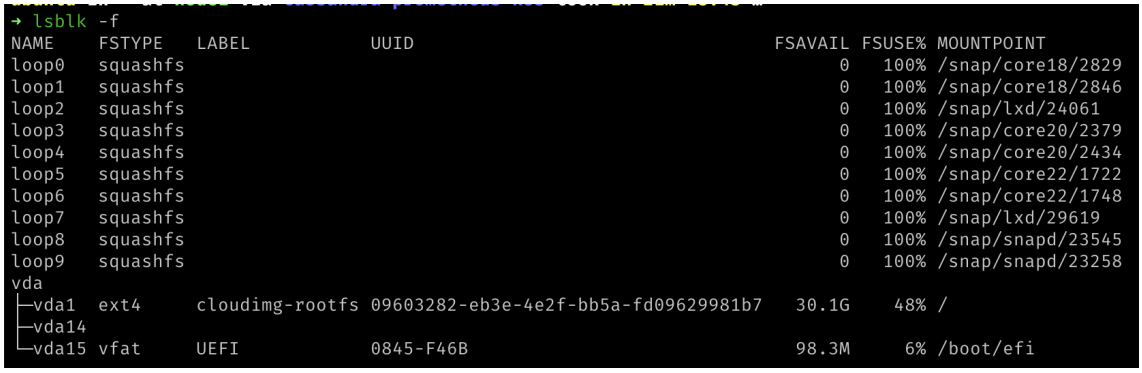
We observe that the first two clusters are Control planes. Because of the small size of the cluster, we will use only the first one as the primary cluster, on which we will not run any workloads. On the remaining five we will run the experiments.

9.2 Dynamic Volume Provisioning

As we observed in the previous chapter, there are several options for volume provisioning (AzureFile, CephFS, local, NFS, RDB [32]). We selected CephFS, a quite popular distributed file system [102]. It allows the Nodes to manage a shared file system, offering high availability and scalability. It is ideal for storing data that need to be shared in the file level [102]. Rook is a Kubernetes controller that facilitates the deployment, management, and orchestration of a Ceph system, integrating it to Kubernetes with minimum manual effort [103].

One of the prerequisites of installing Rook is the existance of [104]:

- Raw disk devices
- Raw partitions
- LVM logical volumes
- Available PVs from a Storage Class in block mode.



```
→ lsblk -f
NAME    FSTYPE   LABEL           UUID                                   FSAVAIL FSUSE% MOUNTPOINT
loop0   squashfs                                                              0   100% /snap/core18/2829
loop1   squashfs                                                              0   100% /snap/core18/2846
loop2   squashfs                                                              0   100% /snap/lxd/24061
loop3   squashfs                                                              0   100% /snap/core20/2379
loop4   squashfs                                                              0   100% /snap/core20/2434
loop5   squashfs                                                              0   100% /snap/core22/1722
loop6   squashfs                                                              0   100% /snap/core22/1748
loop7   squashfs                                                              0   100% /snap/lxd/29619
loop8   squashfs                                                              0   100% /snap/snapd/23545
loop9   squashfs                                                              0   100% /snap/snapd/23258
vda
├─vda1  ext4     cloudimg-rootfs 09603282-eb3e-4e2f-bb5a-fd09629981b7   30.1G    48% /
├─vda14
└─vda15 vfat     UEFI            0845-F46B                              98.3M     6% /boot/efi
```

Σχήμα 9.1: *lsblk command output*

While executing the command *lsblk -f* (list block devices), figure 9.1, there is no raw disk device, partition, or logical volume. We can install Rook using a PV for a storage class. The way we will use CephFS is to create a storage class that will provision PVs that will be mounted by our Pods. Therefore:

1. we need to create PVs from a Storage Class
2. it is going to be used by Rook to create the Ceph cluster
3. which will then be used by a Storage Class that will provision PVs for our Pods.

We see that the second step could be omitted because if we can achieve the creation of the Storage class and the PVs, then we can provision the volumes we need.

1. We create a Storage Class and configure it in WaitForFirstConsumer mode Α΄.2.
2. We create a number of PVs, configuring them with the ReadWriteOnce access mode, capacity of 5 GB, the Retain reclaim policy ,the file system mode, only one node the node affinity (every node has enough PVs) and finally the local file path Α΄.3.

It is to be noted that the manifests and configurations of this system are stored in the open source repository [105].

In this way, we define on which Nodes the Pods can be scheduled, Pods can be scheduled only on Nodes with PVs. By not creating PVs on Node1 we exclude the Pods scheduling. Simultaneously, we ensure that the Volumes remain available after their usage.

We can define in the Statefulset level what is the reclaim policy of the PVC, defining it to be deleted in the case of scaling down and to remain in the case of deletion, this

way we can control the life cycle of the PV related to this PVC. During scaling down the PVC is deleted releasing the PV. We have a Python script A′.9 that regularly checks for released PVs, deletes their data (local file path), and recycles the PV. In this way we achieve dynamic volume provisioning, utilizing primary objects from Kubernetes, the operating system, and a python script.

9.3 Apache Cassandra

For the implementation of Apache Cassandra we are utilizing Statefulset. This object ensures the network and sticky identity, which is necessary for the operation of a distributed system.

9.3.1 Container Image

We select the container image gcr.io/google-samples/cassandra:v14 that is maintained by Google for development and research environments [106], in case of evaluation production environments we would use either cassandra:4.1 or datastax/dse-server.

9.3.2 Cluster size and resources

For the execution of the workload we need to specify the cluster size. We have five Kubernetes nodes available, each of them with four CPU cores and eight GB of memory. We decided that for the observation of the linear scaling to a reasonable extent, we need to be able to scale five fold. This leads to the possibility to have one to five Apache Cassandra nodes, something that restricts the replication capability of the system. We therefore decide to have between three and fifteen Cassandra nodes, with two to three replication strategies depending on the experiment. This would mean that each Kubernetes node needs to schedule three Cassandra nodes (Pods). We define the computer resources limits for each cluster to one CPU and two GiB of memory. At this point, it is noted that this resources are far lower than those specified in the technical documentation (for non-load development environments, 2 CPUs and 8 GB of memory [107], a t2.large AWS virtual machine [108]), in our case they are similar to a t2.small AWS virtual machine [108]. This will allow as to examine how the scaling will happen and if it will respond in an elastic way A′.6, A′.7, A′.8.

We have to configure the heap memory size of the Java Virtual Machine. Technical documentation [109] suggests that we follow the formula $heap = max(min(1/2 * ram, 1024MB), min(1/4 * ram, 8GB)$. This gives us 1024 MB. We have to define the part of the memory size for new objects [109], following the formula $heapnew = min(100 * cpu, 1/4 * ram)$. This returns 100 MB, but after experimenting we observed that 256 MB returned improved performance.

9.3.3 Networking

For the operation of the Statefulset we need a headless service that works on port 9042.

At the same time we have ports 7000 and 7001 available for the node communication. Finally, we define the system seed nodes, essentially the first two Pods A′.4, A′.6, A′.7, A′.8.

9.3.4 Dynamic Volume Provisioning

We utilize the Storage Class and PVC that we defined in a previous section.

9.3.5 Elasticity

In order to achieve elasticity Apache Cassandra has to be able to scale up and scale down. While adding new Pods, the new Cassandra Container communicates with the seed Nodes, connects to the token ring, and receives the data that it owns. For the opposite activity we configure the life cycle of the Container with two Cassandra node commands that transfers the data away from the Node and unsubscribes it from the ring.

- nodetool decommission [110]
- nodetool drain [111]

It is to be noted that the manifests and configurations of this system are stored in the open source repository [105].

9.4 Horizontal Scaling

For the horizontal scaling we will utilize a Metrics Server [112], which is developed and offered by Kubernetes, which allows us to use an HPA that monitors CPU and memory usage. It can not monitor other metrics, but was selected because these metrics are enough to assess the cluster load.

We observe that because of the limited memory (one fourth of the proposed), the Pod restricts as much resources as it can get (even over the limit) and this metric remains unchanged regardless of the load.

Following several experiments we observed that a good limit is eighty and ninety percent of the CPU.

We create HPAs, one for constant load scaling A′.10, one for sine load scaling up A′.11, and one for sine load scaling down A′.12. Comparing the last two HPAs, HPA A′.12 is more sensitive to scaling down as it has a shorter period, stabilization window, and average utilization.

9.5 Benchmark

For evalutation of the system's performance we use the YCSB bemchmark. We can run loads by configuring several parametes.

- Threads
- Target

- Transaction ratio (read, write, update, scan)
- Desired Transaction Consistency
- Record count
- Operation count
- Record selection distribution

We will modify them depending on the goal of each experiment.

It is important to note that YCSB does not support alternating target load. In order to run a sine wave we need to extend it.

The way that the target rate is set is that during the execution of each transaction we delay the next transaction by a specified value. But this delay remains constant. The way that we can influence the delay is to alter it following a sine wave. We extend this code part [113] by creating an Interface Α΄.13 and 2 Strategy Design Patterns, one for constant delay Α΄.14 and one for sine delay Α΄.15. During the creation of each ClientThread the Strategy is configured and during the execution of the transactions, the delay follows the specified Strategy Α΄.16

9.6 Monitoring

To monitor the cluster state we use Prometheus. We install it by using the Helm tool, which groups into charts all the Kubernetes object manifests, configurations, and potentially other charts that are needed for the installation of a system [114]. We use the Chart that has been developed by the Open source community of Prometheus [115]. At the same time, we configure Kubernetes service monitors that specify the services and ports that Prometheus will use [105]. This way we have metrics from Kubernetes.

This though is not sufficient to monitor the custom and internal Apache Cassandra metrics. For this reason we use a JMX exporter [116]. JMX is a Java API that offers a unified way to monitor Java applications. The way to use it is to deploy in each Pod a container with a sidecar configuration. We add the container template and specify two new ports 7199 for the exporter and 8080 for the metrics of each Pod that run a Cassandra node Α΄.6, Α΄.7, Α΄.8, Α΄.4. We configure the service monitor objects to use these ports.

We use Grafana for the presentation of the insightful metrics during the experiments. This will aid in visualizing data from PromQL queries [117].

These queries are:

- Α΄.17: requests
- Α΄.18: number of replicas
- Α΄.19: receive bandwidth
- Α΄.20: rate of received packets

- Α΄.21: transmit bandwidth
- Α΄.22: rate of transmitted packets
- Α΄.23: read requests
- Α΄.24: write requests
- Α΄.25: IO read operations
- Α΄.26: IO read throughput
- Α΄.27: IO write operations
- Α΄.28: IO write throughput
- Α΄.29: memory usage
- Α΄.30: memory limit
- Α΄.31: cpu usage
- Α΄.32: cpu limit
- Α΄.33: hpa average utilization

9.7 Limitations

The main limitation that governs the implementation of this study are the available computing resources. Every Apache Cassandra Pod has available resources of a t2.small and not a t2.large [108], which leads to the resource exhaustion when idle and doesn't allow the monitoring of a larger span of operation. Subsequently, it limits the range of experiments we can run, for example horizontal scaling with different limits. Additionally, HPAs mode of operation doesn't allow the creation of a hysteresis area where the system doesn't scale up or scale down.

Κεφάλαιο 10

Experiment Descriptions

This chapter concerns the description of the experiments.

10.1 Tunable Consistency Experiment

In this specific experiment we will examine how the system reacts to the different levels of consistency.

Option	Value
Number of Pods	10
Replicas	3
Threads	20
Record count	2000000
Operation count	2000000
Distribution	Zipfian
Consistency level	One, Quorum, All
Transation Ratio	Workload B, Workload A

Πίνακας 10.1: *Consistency Expirement*

Workload A consists of 50% reads and 50% updates, simulating a session storage. Workload B consists of 95% reads and 5% updates, simulating a file tagging system, where updates are the creation of tags and the majority of the transactions are reads.

We expect the system's performance to decrease as we increase the consistency level, as each transaction needs to reach more replicas before it is acknowledged.

10.2 Experiment of Scaling up and down with no workload

In this specific experiment we will examine how the system reacts to a scaling command. We start with 3 Pods that contain 2000000 records and command it to scale to 15 Pods. Finally, we command it to scale down to 3 Pods.

Option	Value
Number of Pods	3-15
Replicas	3
Record count	2000000

Πίνακας 10.2: *Experiment of Scaling up and down with no workload*

We will observe how the system reacts to a scale up, scale down, how the data are distributed, and which operations are executed.

10.3 Experiment of Elasticity - Scaling up

In this experiment we will examine how the system reacts to a constant workload with different record counts.

Option	Value
Number of Pods	3-15
Replicas	2
Threads	500
Record count	100000, 2000000
Operation count	55000000
Distribution	Zipfian
Consistency level	One
Transaction Ratio	Workload A
HPA	Α΄.10

Πίνακας 10.3: *Experiment of Elasticity - Scaling up*

We expect that the fewer record counts will not fill the Memtables so there will be no need to be written in SSTables. We also expect that the larger record count will fill the Memtables so that we will have the SSTable creation. Reading data from SSTables will increase the number of IO disk reads. Additionally, we want to examine that the system is elastic and is able to manage the load by adding nodes without decreasing the performance and stability of the system. Lastly, we want to observe the linear scaling of the workload, and whether it will be following the escalation in an analogous way.

10.4 Experiment of Elasticity - Scaling up with sine wave

In this experiment we examine how the system reacts to a sine wave, starting from 3 Pods.

Option	Value
Number of Pods	3-15
Replicas	2
Threads	15
Record count	100000
Operation count	5000000
Distribution	Zipfian
Consistency level	One
Transaction Ratio	Workload A
Period	1200
Amplitude	100
Base target	150
HPA	Α΄.11

Πίνακας 10.4: *Experiment of Elasticity - Scaling up with sine wave*

Essentially we want to observe a sinusoidal workload. We expect that we will view that the records will fit in the Memtables, and therefore there will be no need for disk reads. At the same time, we expect that the scaling up and down of the system is dependent on the load, without any errors.

10.5 Experiment of Elasticity - Scaling down with sine wave

In this experiment we do the opposite task from the previous one, we will examine how the system reacts to a sine wave starting from 10 Pods.

Option	Value
Number of Pods	3-15
Replicas	3
Threads	5
Record count	2000000
Operation count	5000000
Distribution	Zipfian
Consistency level	One
Transaction Ratio	Workload A
Period	1200
Amplitude	100
Base target	150
HPA	Α΄.12

Πίνακας 10.5: *Experiment of Elasticity - Scaling down with sine wave*

We want to observe how the system will react to the scaling down. The wave that we use will be sufficient to trigger scaling down. We expect that the replica number will oscillate between some values, higher during the peak of the sine wave and lower during the minimum value.

Κεφάλαιο 11

Presentation and Discussion of Experiment Results

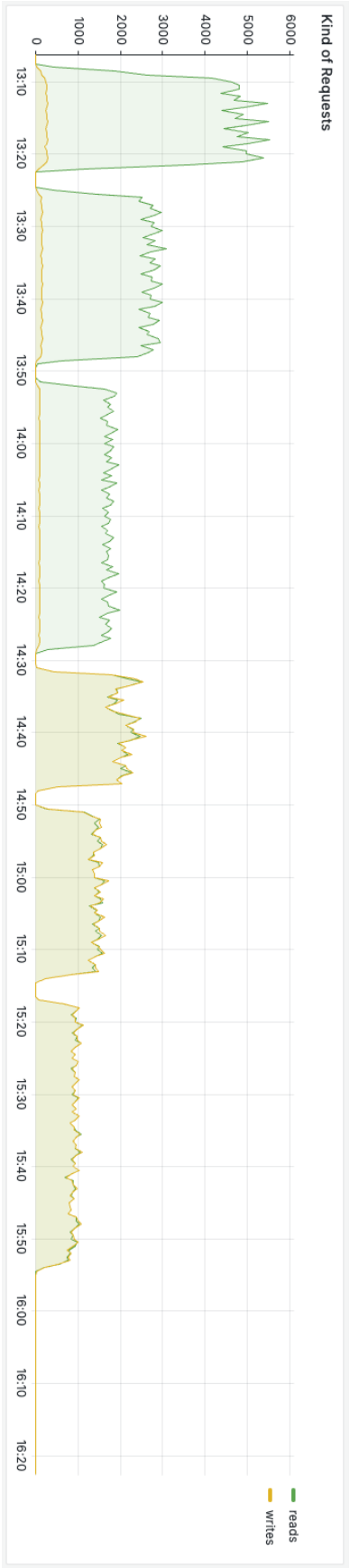
In this chapter we present the results of the experiments, as those are described in the previous chapter.

11.1 Tunable Consistency

In this section we present the results of the Tunable Consistency Experiment 10.1. We executed the workloads described in table 10.1. Three executions with increasing consistency level for Workload B and the same consistencies for Workload A.

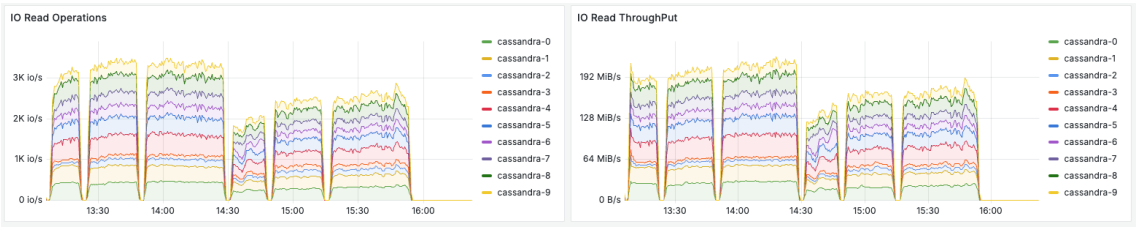
Σχήμα 11.1: *Requests Α'.17*

We observe, figure 11.1, that as we increase the consistency (One, Quorum, All) the throughput of the system decreases and the execution time of each run increases. This is expected as we wait for more acknowledgments from more than one replicas. In the specific case with 3 replicas, we receive 1, 2, 3 acknowledgments respectively.

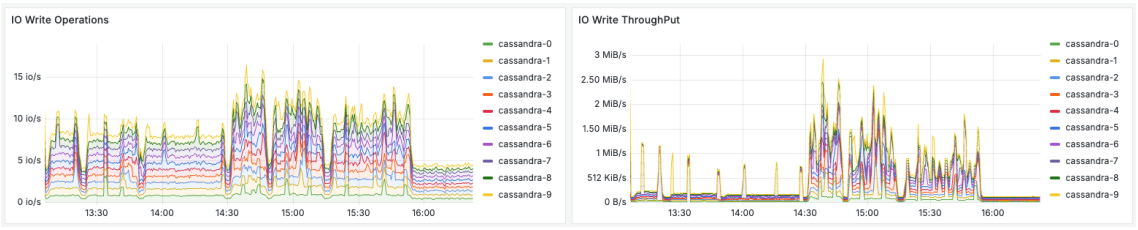


Σχήμα 11.2: *Request Split Α΄.23, Α΄.24*

At the same time we observe, figure 11.2, that different workloads execute the expected ratio of read and update/write requests.



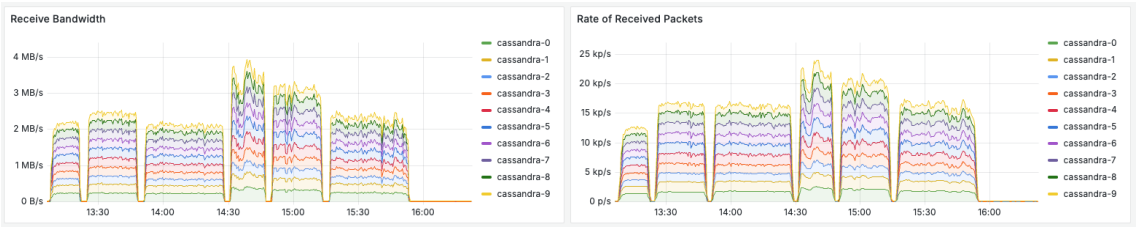
Σχήμα 11.3: *IO Reads Α΄.25, Α΄.26*



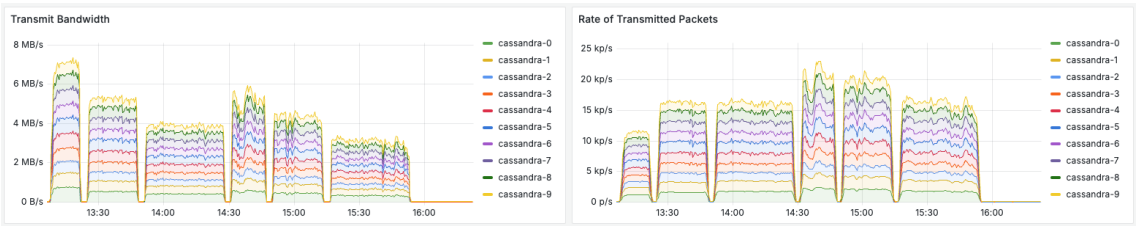
Σχήμα 11.4: *IO Writes Α΄.27, Α΄.28*

We observe ,in figure 11.3, that the number of transactions per workload are similar, a fact that shows that Apache Cassandra runs as many operations as it can, because for the same number of IO disk reads we have smaller throughput. At the same time it confirms that it needs more transactions from the system for each request.

We observe, in figure 11.4, that the reads are not consistent, as each transaction is written in the Commitlog in disk and stored on the Memtable on memory, but once the Memetable is full it is written in the disk in a SSTable.



Σχήμα 11.5: *Incoming Network Activity Α΄.19, Α΄.20*



Σχήμα 11.6: *Outgoing Network Activity Α΄.21, Α΄.22*

We observe, in figures 11.1, 11.5, 11.6, that while the throughput of the system is the same across workloads, the network usage is different. Workload A executes more

updates (writes) and as such more data are being transferred for each request.

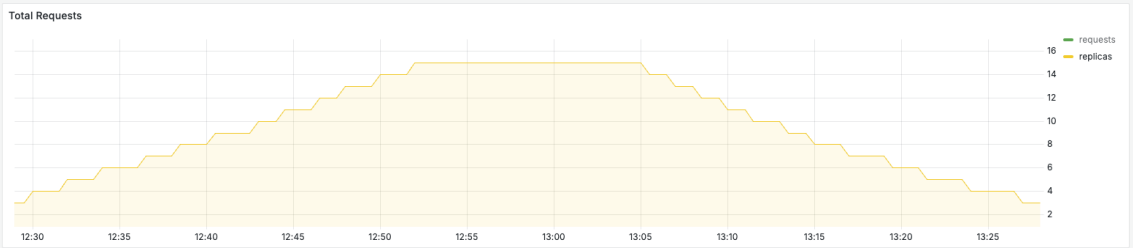
This experiment shows us that throughput (availability) and consistency in a distributed system are reversely correlated. This behaviour is one of the guarantees of Apache Cassandra.

Last but not least, it is fascinating that the workload of reads has the same throughput as the read and write workload.

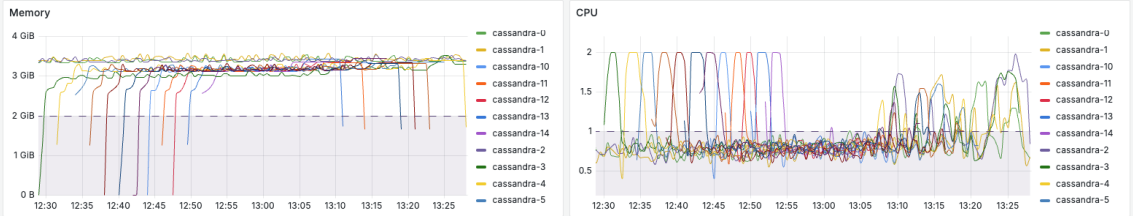
11.2 Scaling up and down with no workload

In this section we present the results of the Scaling up and down with no workload Experiment 10.2.

We run the workload described in table 10.2. We started with 3 nodes and 2000000 records written in the table. We scaled up to 15 Apache Cassandra nodes and then scaled down back to 3.

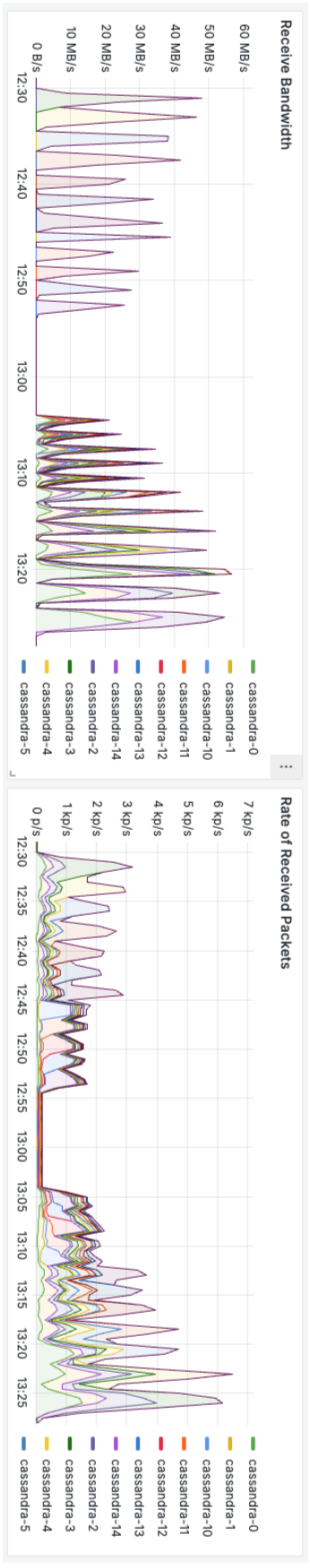


Σχήμα 11.7: *Replica Count A′.18*

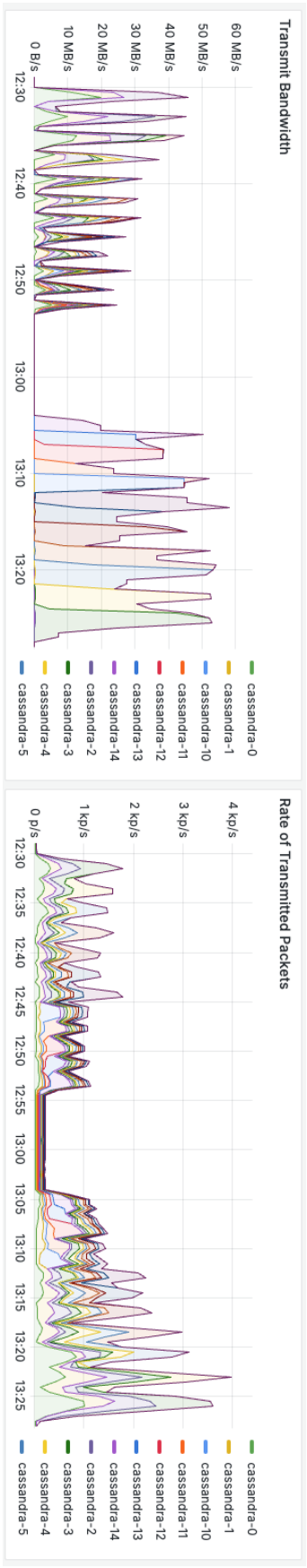


Σχήμα 11.8: *Resource usage A′.29, A′.30, A′.31, A′.32*

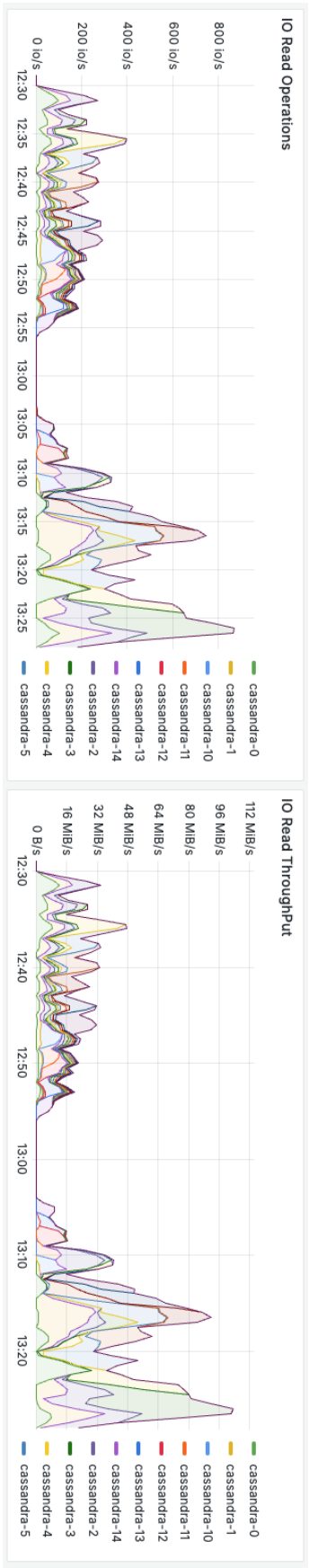
We observed, in figures 11.7, 11.8, that the use of the memory throughout the duration of the life cycle of an Apache Cassandra Container is substantially over the limit. This prohibited us from using the specific metric for the scaling of our system. We also observed that the usage of the CPU ranged from 70% to 200%, this metric can be used for the horizontal scaling of the system. The limitation that each Pod had substantially less computing resources from the proposed level, constrains the way in which we could assess the elasticity of our system. Despite this, we assessed it through the usage of the CPU.



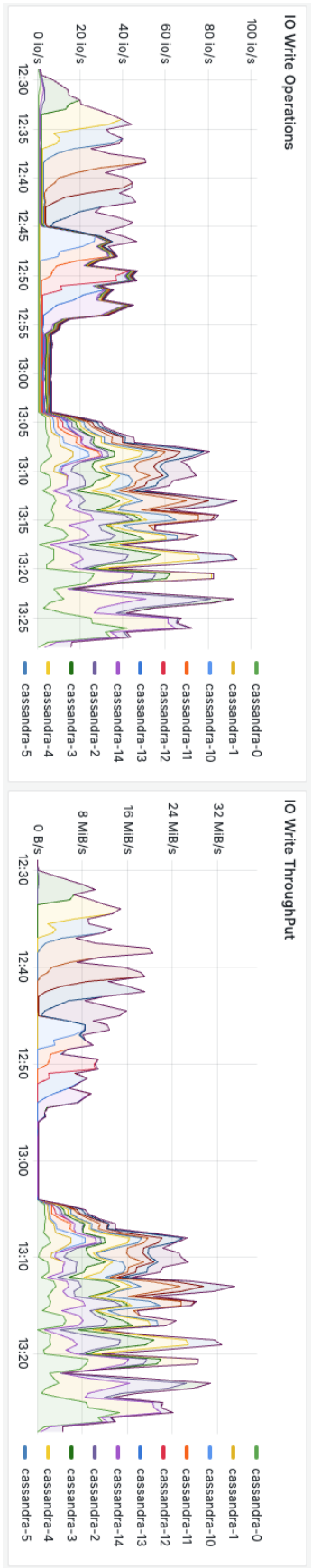
Σχήμα 11.9: *Incoming Network Activity A'.19, A'.20*



Σχήμα 11.10: *Outgoing Network Activity Α'.21, Α'.22*



Σχήμα 11.11: *IO Reads Α'.25, Α'.26*



Σχήμα 11.12: *IO Writes Α'.27, Α'.28*

We observed, in figures **??**, 11.10, 11.11, 11.12, that as we added nodes they received (exclusively the replica of the data they own) and the preexisting nodes sent them. This was expected and confirms that the system uses virtual nodes, tokens, in the ring. If these did not exist then the data rebalancing would be more local. At the same time, from the reads and writes we observed a more inconsistent behavior which was caused by the data rebalancing that performs read and write operations on SSTables.

```
Datacenter: ntua
================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  10.233.74.77    1.48 GiB   32           17.7%             72999a77-d2f8-4788-be0e-8f8ae5112e84  rack1
UN  10.233.75.13    751.47 MiB  32           18.5%             1588345e-f1fc-40b4-aa4e-df2764241750  rack1
UN  10.233.71.13    532.27 MiB  32           20.3%             260f6237-e6ae-421b-ae0c-232e0e67e243  rack1
UN  10.233.74.79    471.68 MiB  32           21.1%             1bc21661-a2a1-42da-b3d7-03e03c32938f  rack1
UN  10.233.97.174   1.09 GiB   32           22.0%             55c95d60-9cce-4256-b469-87f30f72139c  rack1
UN  10.233.75.1     2.04 GiB   32           16.6%             f7c1fa0b-ea19-4e8b-8e98-7d7e1ba17b00  rack1
UN  10.233.97.129   567.72 MiB  32           20.2%             b9f8913a-9b69-4057-aec5-e88d81f342af  rack1
UN  10.233.97.160   2.53 GiB   32           19.1%             9f95425c-e48d-4f48-b902-98f2a5d9f96b  rack1
UN  10.233.74.69    763.4 MiB  32           21.0%             ba9de8f3-6dd5-4aab-81e8-1370039aba47  rack1
UN  10.233.75.88    449.16 MiB  32           21.5%             2962d76f-329b-4fe3-a1eb-8af84dced15f  rack1
UN  10.233.71.24    2.71 GiB   32           20.5%             621b56e5-c788-4958-9ec1-642e80d99544  rack1
UN  10.233.75.124   718.13 MiB  32           21.3%             205a5344-c498-41b3-8b2f-fa4296585b2c  rack1
UN  10.233.71.63    1.22 GiB   32           17.5%             0d26ec8f-2169-4c78-a689-e9b9b1bf4134  rack1
UN  10.233.75.20    523.06 MiB  32           19.9%             0621ec03-dfc4-4849-ad96-bf0d9fab33d6  rack1
UN  10.233.75.118   1.19 GiB   32           22.9%             0121ae5f-42cc-4e8b-b3a0-da988d1534ad  rack1
```

Σχήμα 11.13: *Status for 15 nodes*

```
Datacenter: ntua
================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  10.233.71.24    4.36 GiB   32           100.0%            621b56e5-c788-4958-9ec1-642e80d99544  rack1
UN  10.233.75.1     4.48 GiB   32           100.0%            f7c1fa0b-ea19-4e8b-8e98-7d7e1ba17b00  rack1
UN  10.233.97.160   4.47 GiB   32           100.0%            9f95425c-e48d-4f48-b902-98f2a5d9f96b  rack1
```

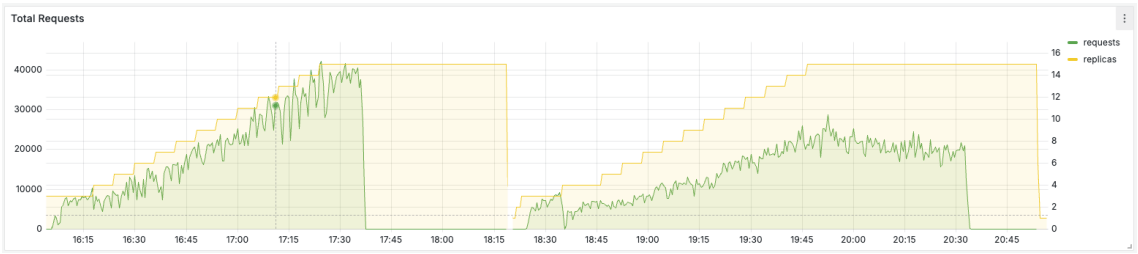
Σχήμα 11.14: *Status for 3 nodes*

Lastly we observed, in figures 11.13, 11.14, the distribution of data in the largest and smallest cluster size. The data were distributed evenly (between 16.6% and 22.9%). Load and data owned are disproportional because some regions of the ring are “hotter” (read more often), and therefore we had the existence of more SSTables and Commitlogs, which were caused by the zipfian distribution.

11.3 Elasticity - Scaling up

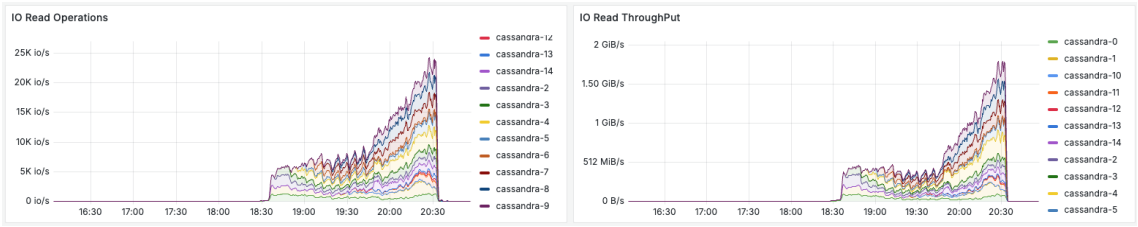
In this section we present the results of the Elasticity - Scaling up Experiment 10.3.

We run the workload specified in table 10.3. Two execution with different record count.

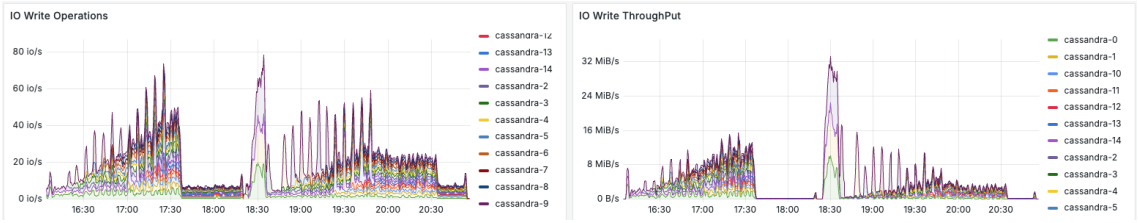


Σχήμα 11.15: *Requests Α'.17 - Replica Count Α'.18*

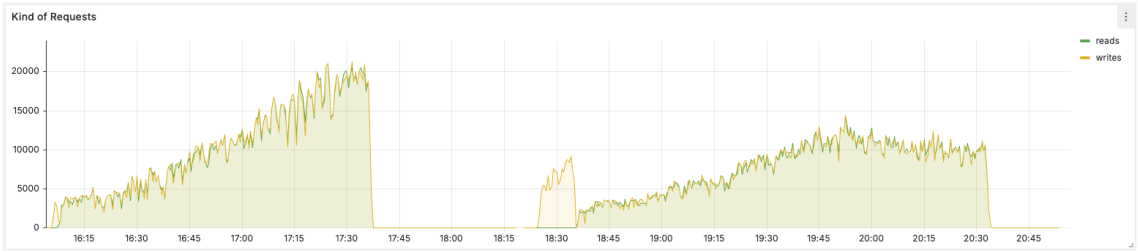
We observed that, in figure 11.15, in both loads the throughput of the system increased linearly, increasing five fold its throughput as the size of the system five folds, even if the system had less than the proposed resource levels. This linear increase of throughput existed regardless of the record count.



Σχήμα 11.16: *IO Reads Α'.25, Α'.26*

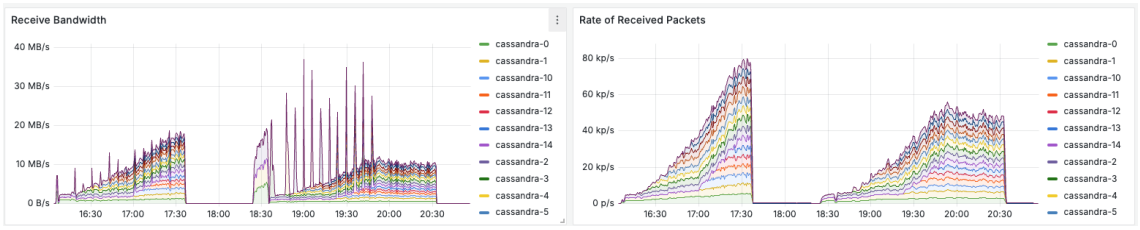


Σχήμα 11.17: *IO Writes Α'.27, Α'.28*

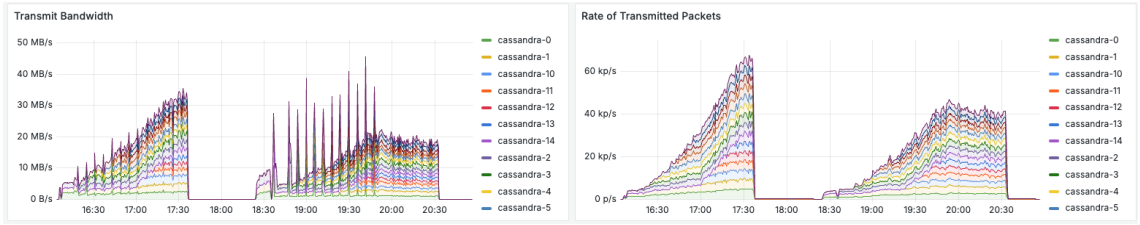


Σχήμα 11.18: *Request Split Α'.23, Α'.24*

Additionally, the larger record count filled the Memtables and each node stored them in the SSTables disk, this was shown by the reads that took place in the second workload, figure 11.16. The writes were similar for the two workloads, besides the loading of the data in the second load where we had significantly more records, in figures 11.17, 11.18.



Σχήμα 11.19: *Incoming Network Activity A'.19, A'.20*



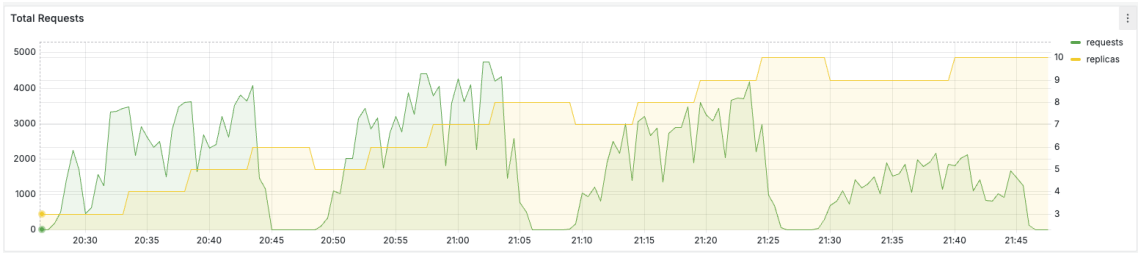
Σχήμα 11.20: *Outgoing Network Activity A'.21, A'.22*

We observed, in figures 11.19, 11.20, 11.17, that during the scaling up of the system we had leaps in the movement of the network and the writes, as we also observed in the previous experiment as data were transferred to new nodes they are rebalanced. This observation was smaller in the fewer record count workload since we were dealing with substantially fewer elements.

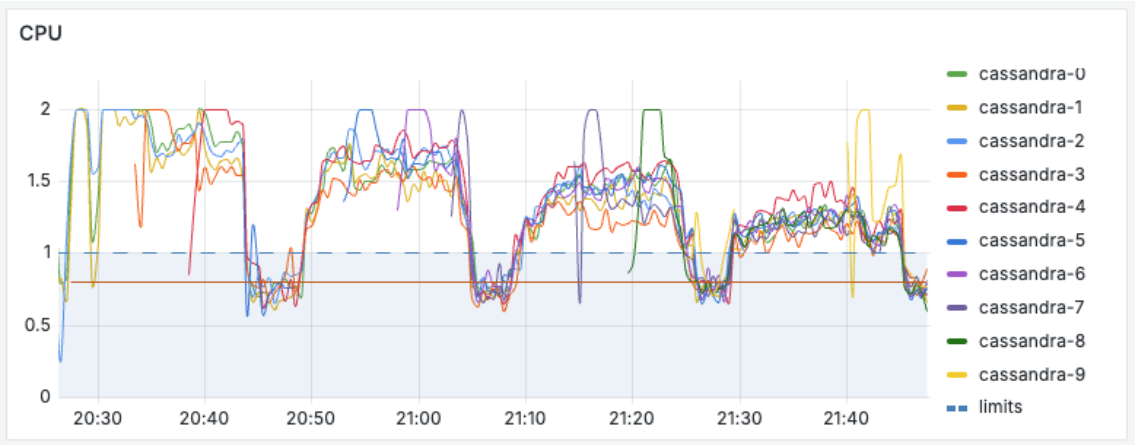
Lastly, we observed, in figure 11.15, that during the scaling up, the throughput of the system decreased for a limited time. This is a result of the computing resources of the system which were consumed for the transfer of data, in figures 11.16, 11.17, 11.19, 11.20. The throughput for the large record count workload slightly decreased in the end of the workload, as there were many reads from disks, in figure 11.16.

11.4 Elasticity - Scaling up with sine wave

In this section we present the results coming from the Elasticity - Scaling up with sine wave experiment 10.4.



Σχήμα 11.21: *Requests A'.17 - Replica count A'.18*



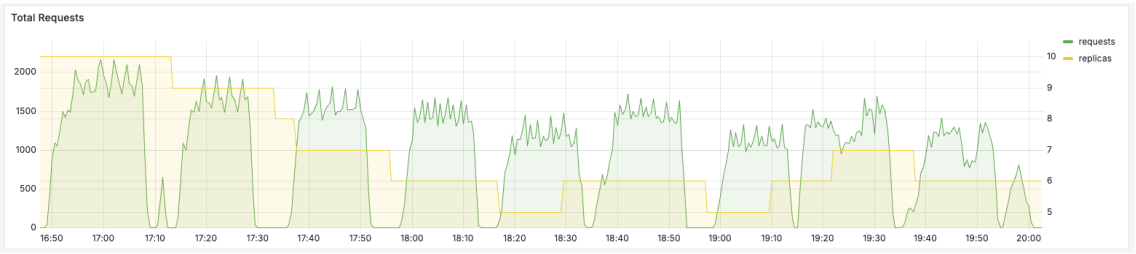
Σχήμα 11.22: *Resource usage A'.29, A'.30, A'.31, A'.32, A'.33*

We observed, in figure 11.21, that our YCSB extension produced a sine wave. Additionally, in figure 11.22, that the initial size of the system was not sufficient to handle the load and was scaling similarly to the previous experiments. During minimum load, the system scaled down. At the same time, we saw during the third period a decrease for a moment of the system as the available resources were focused on the transfer and balancing of the data. The amplitude of the last period can be lower than the nominal, as YCSB does not have the target to execute as many transactions as in the previous periods.

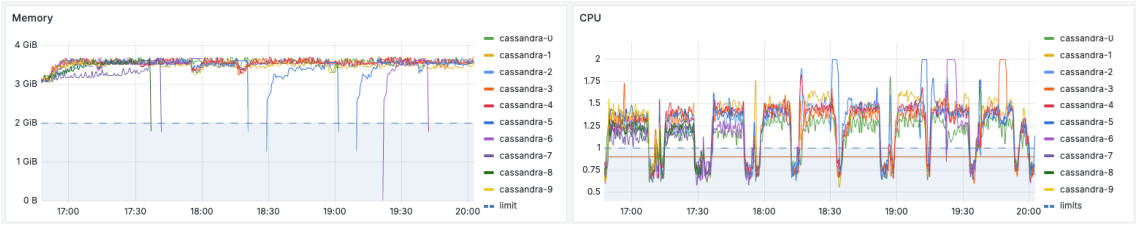
Lastly we understood that the system consumed almost always more resources than the average utilization from the HPA. Nonetheless, there were isolated scaling downs.

11.5 Elasticity - Scaling down with sine wave

In this section we present the results coming from the Elasticity - Scaling down with sine wave experiment 10.5.



Σχήμα 11.23: *Requests A'.17 - Replica count A'.18*



Σχήμα 11.24: *Resource usage A'.29, A'.30, A'.31, A'.32, A'.33*

We observed, in figure 11.23, that the system scaled down with zero load, but scaled up when the system had high load. We witnessed an oscillation between 5 and 7 replicas (not synchronized with the sine wave period), maintaining similar performance. This oscillation is caused by the way HPA functions, needing to scale up or scale down depending on the usage.

It can't create a hysteresis region where no action is taken (scaling up, scaling down). We attempted to create 2 HPAs that are related to our system, one for scaling up and one for scaling down, but this created error, as only one HPA can use an application selector, in figure 11.25.

```
Name:                                                  cassandra-hpa-cpu-sine-up
Namespace:                                             default
Labels:                                                <none>
Annotations:                                           <none>
CreationTimestamp:                                     Fri, 14 Feb 2025 15:48:24 +0000
Reference:                                             StatefulSet/cassandra
Metrics:                                               ( current / target )
  resource cpu on pods  (as a percentage of request):  <unknown> / 100%
Min replicas:                                          3
Max replicas:                                          15
Behavior:
  Scale Up:
    Stabilization Window: 360 seconds
    Select Policy: Max
    Policies:
      - Type: Pods  Value: 1  Period: 360 seconds
  Scale Down:
    Select Policy: Max
    Policies:
      - Type: Percent  Value: 100  Period: 15 seconds
StatefulSet pods:    10 current / 0 desired
Conditions:
  Type            Status  Reason              Message
  ----            ------  ------              -------
  AbleToScale     True    SucceededGetScale   the HPA controller was able to get the target's current scale
  ScalingActive   False   AmbiguousSelector   pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-up default/cassandra-hpa-cpu-sine-down]
Events:
  Type     Reason                   Age                From                         Message
  ----     ------                   ----               ----                         -------
  Warning  FailedComputeMetricsReplicas  10m                horizontal-pod-autoscaler  pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-down default/cassandra-hpa-cpu-sine-up]
  Warning  FailedComputeMetricsReplicas  9m6s (x11 over 11m)  horizontal-pod-autoscaler  pods by selector app=cassandra are controlled by multiple HPAs: [default/cassandra-hpa-cpu-sine-down default/cassandra-hpa-cpu-sine-up]
```

Σχήμα 11.25: *Multiple HPA Error Message*

It is to be noted that during the experiments, we reset the virtual machines and after every such action the system maintained its state, demonstrating the self-healing capabilities of Kubernetes.

11.6 Summary of Results

With the aforementioned experiments we observed that the behavior of the system corresponded to the expected results, as those were described in the theoretical part of this paper. Additionally, we observed the elastic behavior of Apache Cassandra, using Kubernetes native elements with minor modifications. Despite the limitation of resources for each Pod, we achieved the expected linear performance. It is to be highlighted that we used t2.small instead of t2.large, the lateral being the proposed for non load environments [107], [108].

Μέρος VIII

Epilogue

Κεφάλαιο 12

Epilogue

12.1 Conclusions

At this stage we will summarize the main conclusions we reached with this paper, as it has been derived from the theory and the experiments.

Kubernates has the ability to automate the deployment and management of the distributed system Apache Cassandra, without the manual efforts a system like this would require. Essentially, just by declaring a Storage Class A′.2, a Statefulset A′.6, A′.7, A′.8, a number of PVs A′.3, and a service A′.4 we achieved to set up Apache Cassandra. In practice, with the definition of the services and some ports we succeeded in setting up a stable network for the communication of fifteen nodes. We observed the self-healing capability of Kubernetes during virtual machine resets.

Kubernetes, being a platform agnostic system, allows us to transfer the manifests in any cloud computing environment, without needing platform specific configurations. We use tools and extensions that are open source (Prometheus, Metrics-server), which allowed us to navigate this research process effectively and efficiently, aspects that add to the ease of use. With minimal code A′.9 we implemented dynamic volume provisioning, which adds to the extendability capabilities of Kubernetes. We realized the limitations in the operation of HPAs, as it is an inflexible option for very dynamic loads.

A further observation is that despite the limited resources, we set up a system that can react elastically to load. It scaled horizontally linearly to match the load and was able to offer the expected performance without faults. This methodology is widely adopted with microservices architectures, that scale horizontally, moving away from monolithic systems.

Through the experiments, we observed and comprehended how a distributed computing system is designed, governed by high availability, partition tolerance, and elasticity.

The fascinating conclusion is that open source technologies allow for each user to design and implement powerful systems without major modifications.

12.2 Future Considerations

- The repetition of the experiments with more computing resources so that we can perform a comparative analysis.

- The examination of different databases, with different guarantees but maintaining the same infrastructure so that we can compare the behavior and performance between them.
- The implementation of an autoscaler with more features, hysteresis region, combination of multiple metrics, and dynamic behavior based on load.
- This study can act as a basis of exploration for further concepts of elastic data management (elasticity based on dynamic programming, machine learning, etc.) .

Παραρτήματα

Παράρτημα Α΄

Παραμετροποίηση Συστημάτων

Στο συγκεκριμένο παράρτημα παρατίθενται: οι εντολές, τα κομμάτια κώδικά, οι αναζητήσεις και οι διαμορφώσεις που χρησιμοποιήθηκαν για την εκτέλεση του πειραματικής λειτουργίας. Κάθε στοιχείο εξηγείται στο σημείο που εξηγείται στο Πρακτικό μέρος.

Αλγοριθμος Α΄.1: *Kubespray Installation Commands*

```
cp -rfp inventory/sample inventory/mycluster
declare -a IPS=(10.0.1.102 10.0.1.121 10.0.1.122 10.0.1.128 10.0.1.135 \
    10.0.1.136)
CONFIG_FILE=inventory/mycluster/hosts.yaml \
    python3 contrib/inventory_builder/inventory.py ${IPS[@]}
sudo ansible-playbook -i inventory/mycluster/hosts.yaml \
    --private-key=/home/ubuntu/.ssh/id_rsa -u ubuntu \
    --become --become-user=root cluster.yml
```

Αλγοριθμος Α΄.2: *Storage Class*

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

Αλγοριθμος Α΄.3: *Persistant Volume*

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv-2-a
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 5Gi
  local:
    path: /mnt/data/a
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - node2
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
```

Αλγοριθμος Α΄.4: *Cassandra Headless Service*

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

Αλγοριθμος Α΄.5: *Cassandra Metrics Service*

```
apiVersion: v1
kind: Service
metadata:
  name: cassandra-metrics
  labels:
    app: cassandra
    service: metrics
spec:
  ports:
  - port: 8080
    targetPort: metrics
    name: metrics
  selector:
    app: cassandra
```

Αλγοριθμος Α΄.6: *Cassandra Statefulset part 1*

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  persistentVolumeClaimRetentionPolicy:
    whenDeleted: "Retain"
    whenScaled: "Delete"
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      terminationGracePeriodSeconds: 1800
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v14
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "1000m"
            memory: 2Gi
          requests:
            cpu: "500m"
            memory: 2Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
```

Αλγοριθμος Α'.7: *Cassandra Statefulset part 2*

```
        lifecycle:
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - "nodetool decommission || nodetool drain"
        env:
          - name: MAX_HEAP_SIZE
            value: 1024M
          - name: HEAP_NEWSIZE
            value: 256M
          - name: CASSANDRA_SEEDS
            value: -| "cassandra-0.cassandra.default.svc.cluster.local,
                cassandra-1.cassandra.default.svc.cluster.local"
          - name: CASSANDRA_CLUSTER_NAME
            value: "thesis"
          - name: CASSANDRA_DC
            value: "ntua"
          - name: CASSANDRA_RACK
            value: "rack1"
          - name: POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
        readinessProbe:
          exec:
            command:
            - /bin/bash
            - -c
            - /ready-probe.sh
          initialDelaySeconds: 60
          timeoutSeconds: 60
        volumeMounts:
        - name: cassandra-data
          mountPath: /cassandra_data
```

Αλγοριθμος Α΄.8: *Cassandra Statefulset part 3*

```
      - name: cassandra-exporter
        image: bitnami/cassandra-exporter:latest
        ports:
        - containerPort: 8080
          name: metrics
        resources:
          limits:
            cpu: "200m"
            memory: 256Mi
          requests:
            cpu: "200m"
            memory: 256Mi
        env:
          - name: JVM_OPTS
            value: "-Xms128m -Xmx128m"
          - name: CONFIG
            value: |
              hostPort: localhost:7199
              ssl: False
              lowercaseOutputName: False
              lowercaseOutputLabelNames: False
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: local-storage
      resources:
        requests:
          storage: 5Gi
```

Αλγοριθμος Α΄.9: *Python Recycle Script*

```
def monitor_released_pvs(self):
    while True:
        pvs = self.clients.v1.list_persistent_volume()
        released_pvs = [
            pv for pv in pvs.items if pv.status.phase == 'Released']
        self.recycle_pvs(released_pvs)
        time.sleep(30)

def recycle_pvs(self, pvs: List[Any]):
    for pv in pvs:
        name = pv.metadata.name
        path = pv.spec.local.path
        node = pv.spec.node_affinity.required.node_selector_terms[
            0].match_expressions[0].values[0]
        self.delete_pv(name)
        self.clean_mnt_directory(node, path)
        logger.info(f"Persistent Volume {name} cleaned successfully.")
    if pvs:
        self.reset_manager.apply_manifests(
            "manifests/local-pv-massive.yaml")

def clean_mnt_directory(self, node: str, path: str):
    command = f"rm -rf {path}/*"
    ssh_command = ["ssh", f"ubuntu@{node}", f"{command}"]
    logger.info(
        f'Command for clearing the directory: {" ".join(ssh_command)}')
    self.reset_manager.try_subprocess(ssh_command)

def delete_pv(self, pv_name: str):
    try:
        self.clients.v1.delete_persistent_volume(name=pv_name)
        logger.info(f"Persistent Volume {pv_name} deleted successfully.")
    except client.exceptions.ApiException as e:
        logger.error(
            f"Exception when deleting Persistent Volume {pv_name}: {e}")
```

Αλγοριθμος Α'.10: *HPA Constant Scaling*

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cassandra-hpa-cpu
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: cassandra
  minReplicas: 3
  maxReplicas: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 600
      policies:
      - type: Pods
        value: 1
        periodSeconds: 360
    scaleDown:
      stabilizationWindowSeconds: 600
      policies:
      - type: Pods
        value: 1
        periodSeconds: 360
```

Αλγοριθμος Α΄.11: *HPA Sine Scaling*

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cassandra-hpa-cpu-sine
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: cassandra
  minReplicas: 3
  maxReplicas: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 360
      policies:
      - type: Pods
        value: 1
        periodSeconds: 360
    scaleDown:
      stabilizationWindowSeconds: 360
      policies:
      - type: Pods
        value: 1
        periodSeconds: 360
```

Αλγοριθμος Α΄.12: *HPA Sine Scaling sensitive to scaling down*

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: cassandra-hpa-cpu-sine
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: cassandra
  minReplicas: 3
  maxReplicas: 15
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 90
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 360
      policies:
      - type: Pods
        value: 1
        periodSeconds: 360
    scaleDown:
      stabilizationWindowSeconds: 180
      policies:
      - type: Pods
        value: 1
        periodSeconds: 180
```

Αλγοριθμος Α΄.13: *Strategy Interface*

```
public interface TargetStrategy {
  long calculate();
}
```

Αλγοριθμος Α΄.14: *Constant Strategy Class*

```
public class ConstantTargetStrategy implements TargetStrategy {
  private final ClientThread clientThread;

  public ConstantTargetStrategy(ClientThread clientThread) {
    this.clientThread = clientThread;
  }

  @Override
  public long calculate() {
    return clientThread.getTargetOpsTickNs();
  }
}
```

Αλγοριθμος Α΄.15: *Sine Strategy Class*

```
public class SineTargetStrategy implements TargetStrategy {
  private final int period;
  private final int baseTarget;
  private final int amplitude;
  private final long startTime;

  public SineTargetStrategy(Properties properties, long startTime) {
    this.startTime = startTime;
    period = Integer.parseInt(properties.getProperty("period", "60"));
    baseTarget = Integer.parseInt(properties.getProperty("baseTarget", "50"));
    amplitude = Integer.parseInt(properties.getProperty("amplitude", "25"));
    if (baseTarget <= 0 || baseTarget - amplitude <= 0) {
      throw new IllegalArgumentException(
          String.format("baseTarget %d and amplitude %d will lead to stuck strategy."
            baseTarget, amplitude));
    }
  }

  @Override
  public long calculate() {
    long now = System.currentTimeMillis();
    double radians = (2 * Math.PI * now - startTime) / (period * 1_000);
    double newTarget = baseTarget + amplitude * Math.sin(radians);
    double newTargetPerThreadPerMs = newTarget / 1_000.0;
    return (long) (1_000_000 / newTargetPerThreadPerMs);
  }
}
```

Αλγοριθμος Α΄.16: *Strategy Usage*

```
private void throttleNanos(long startTimeNanos) {
  //throttle the operations
  if (targetOpsPerMs > 0) {
    long newTargetOpsTickNs = targetStrategy.calculate();
    // delay until next tick
    long deadline = startTimeNanos + opsdone * newTargetOpsTickNs;
    sleepUntil(deadline);
    measurements.setIntendedStartTimeNs(deadline);
  }
}
```

Αλγοριθμος Α΄.17: *Requests*

```
sum(rate(cassandra_stats{
    name=~
        "org:apache:cassandra:metrics:clientrequest:write.*:latency:count"
}[1m]))
+
sum(rate(cassandra_stats{
    name=~
        "org:apache:cassandra:metrics:clientrequest:read.*:latency:count"
}[1m]))
```

Αλγοριθμος Α΄.18: *Number of Replicas*

```
sum(kube_pod_status_phase{
    phase="Running", pod=~"cassandra-.*", namespace="default"
})
```

Αλγοριθμος Α΄.19: *Receive Bandwidth*

```
sum(irate(container_network_receive_bytes_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    pod=~"cassandra-.*"
}[1m])) by (pod)
```

Αλγοριθμος Α΄.20: *Rate of Received Packets*

```
sum(rate(container_network_receive_packets_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    pod=~"cassandra-.*"
}[2m])) by (pod)
```

Αλγοριθμος Α΄.21: *Transmit Bandwidth*

```
sum(irate(container_network_transmit_bytes_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    pod=~"cassandra-.*"
}[1m])) by (pod)
```

Αλγοριθμος Α΄.22: *Rate of Trasmitted Packets*

```
sum(rate(container_network_transmit_packets_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    pod=~"cassandra-.*"
}[2m])) by (pod)
```

Αλγοριθμος Α΄.23: *Read Requests*

```
sum(rate(cassandra_stats{
    name=~
        "org:apache:cassandra:metrics:clientrequest:read.*:latency:count"
}[1m]))
```

Αλγοριθμος Α΄.24: *Write Requests*

```
sum(rate(cassandra_stats{
    name=~
        "org:apache:cassandra:metrics:clientrequest:write.*:latency:count"
}[1m]))
```

Αλγοριθμος Α'.25: *IO Read Operations*

```
(sum by(pod) (rate(container_fs_reads_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    device=~"(/dev/)?
        (mmcblk.p.+|nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|md.+|dasd.+)",
    container!="",
    pod=~"cassandra-.*"
}[1m])))
```

Αλγοριθμος Α'.26: *IO Read ThroughPut*

```
(sum by(pod) (rate(container_fs_reads_bytes_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    device=~"(/dev/)?
        (mmcblk.p.+|nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|md.+|dasd.+)",
    container!="",
    pod=~"cassandra-.*"
}[1m])))
```

Αλγοριθμος Α'.27: *IO Write Operations*

```
(sum by(pod) (rate(container_fs_writes_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    device=~"(/dev/)?
        (mmcblk.p.+|nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|md.+|dasd.+)",
    container!="",
    pod=~"cassandra-.*"
}[1m])))
```

Αλγοριθμος Α'.28: *IO Write ThroughPut*

```
(sum by(pod) (rate(container_fs_writes_bytes_total{
    job="kubelet", metrics_path="/metrics/cadvisor",
    device=~"(/dev/)?
        (mmcblk.p.+|nvme.+|rbd.+|sd.+|vd.+|xvd.+|dm-.+|md.+|dasd.+)",
    container!="",
    pod=~"cassandra-.*"
}[1m])))
```

Αλγοριθμος Α΄.29: *Memory Usage*

```
sum(container_memory_working_set_bytes{
    job="kubelet",
    metrics_path="/metrics/cadvisor",
    namespace="default",
    pod=~"cassandra-.*",
    container=~"cassandra"
}) by (pod)
```

Αλγοριθμος Α΄.30: *Memory Limit*

```
sum(kube_pod_container_resource_limits{
    job="kube-state-metrics",
    pod="cassandra-0",
    resource="memory",
    container="cassandra"
})
```

Αλγοριθμος Α΄.31: *CPU Usage*

```
sum(node_namespace_pod_container:
    container_cpu_usage_seconds_total:sum_irate{
        pod=~"cassandra.*",
        container="cassandra"
}) by (pod)
```

Αλγοριθμος Α΄.32: *CPU Limit*

```
sum(kube_pod_container_resource_limits{
    job="kube-state-metrics",
    pod="cassandra-0",
    resource="cpu",
    container="cassandra"
})
```

Αλγοριθμος Α΄.33: *HPA Utilization*

```
sum(kube_horizontalpodautoscaler_spec_target_metric{}/100)
```

Βιβλιογραφία

[1] *Cluster Architecture*. https://kubernetes.io/docs/concepts/architecture/. Ημερομηνία πρόσβασης: 10-02-2025.

[2] *How does a HorizontalPodAutoscaler work?* https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#how-does-a-horizontalpodautoscaler-work. Ημερομηνία πρόσβασης: 10-02-2025.

[3] *Dynamo Consistent Hashing using a Token Ring*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#consistent-hashing-using-a-token-ring. Ημερομηνία πρόσβασης: 10-02-2025.

[4] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan και Russell Sears. *Benchmarking cloud serving systems with YCSB*. σελίδες 143–154, 2010.

[5] *Prometheus Overview*. https://prometheus.io/docs/introduction/overview/. Ημερομηνία πρόσβασης: 10-02-2025.

[6] *What is Big Data?* https://cloud.google.com/learn/what-is-big-data?hl=en. Ημερομηνία πρόσβασης: 10-02-2025.

[7] Jeffrey Dean και Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. *Commun. ACM*, 51(1):107–113, 2008.

[8] *What is vendor lock-in? | Vendor lock-in and cloud computing*. https://www.cloudflare.com/learning/cloud/what-is-vendor-lock-in/. Ημερομηνία πρόσβασης: 10-02-2025.

[9] *Historical context for Kubernetes*. https://kubernetes.io/docs/concepts/overview/#going-back-in-time. Ημερομηνία πρόσβασης: 10-02-2025.

[10] *Turnkey Cloud Solutions | Kubernetes*. https://kubernetes.io/docs/setup/production-environment/turnkey-solutions/. Ημερομηνία πρόσβασης: 10-02-2025.

[11] Avinash Lakshman και Prashant Malik. *Cassandra — A Decentralized Structured Storage System*. *Operating Systems Review*, 44:35–40, 2010.

[12] *Overview | Kubernetes*. https://kubernetes.io/docs/concepts/overview/. Ημερομηνία πρόσβασης: 10-02-2025.

[13] *kube-apiserver*. https://kubernetes.io/docs/concepts/architecture/#kube-apiserver. Ημερομηνία πρόσβασης: 10-02-2025.

[14] *etcd*. https://kubernetes.io/docs/concepts/architecture/#etcd. Ημερομηνία πρόσβασης: 10-02-2025.

[15] *kube-scheduler*. https://kubernetes.io/docs/concepts/architecture/#kube-scheduler. Ημερομηνία πρόσβασης: 10-02-2025.

[16] *kube-controller-manager*. https://kubernetes.io/docs/concepts/architecture/#kube-controller-manager. Ημερομηνία πρόσβασης: 10-02-2025.

[17] *cloud-controller-manager*. https://kubernetes.io/docs/concepts/architecture/#cloud-controller-manager. Ημερομηνία πρόσβασης: 10-02-2025.

[18] *kubelet*. https://kubernetes.io/docs/concepts/architecture/#kubelet. Ημερομηνία πρόσβασης: 10-02-2025.

[19] *kube-proxy*. https://kubernetes.io/docs/concepts/architecture/#kube-proxy. Ημερομηνία πρόσβασης: 10-02-2025.

[20] *Container Runtimes*. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. Ημερομηνία πρόσβασης: 10-02-2025.

[21] *What is a Pod*. https://kubernetes.io/docs/concepts/workloads/pods/#what-is-a-pod. Ημερομηνία πρόσβασης: 10-02-2025.

[22] *Pods*. https://kubernetes.io/docs/concepts/workloads/pods/. Ημερομηνία πρόσβασης: 10-02-2025.

[23] *Resource sharing and communication*. https://kubernetes.io/docs/concepts/workloads/pods/#resource-sharing-and-communication. Ημερομηνία πρόσβασης: 10-02-2025.

[24] *Pods with multiple containers*. https://kubernetes.io/docs/concepts/workloads/pods/#how-pods-manage-multiple-containers. Ημερομηνία πρόσβασης: 10-02-2025.

[25] *Init Containers*. https://kubernetes.io/docs/concepts/workloads/pods/init-containers/. Ημερομηνία πρόσβασης: 10-02-2025.

[26] *Why volumes are important*. https://kubernetes.io/docs/concepts/storage/volumes/#why-volumes-are-important. Ημερομηνία πρόσβασης: 10-02-2025.

[27] *Pods and Controllers*. https://kubernetes.io/docs/concepts/workloads/pods/#pods-and-controllers. Ημερομηνία πρόσβασης: 10-02-2025.

[28] *Deployment*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Ημερομηνία πρόσβασης: 10-02-2025.

[29] *Daemonset*. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. Ημερομηνία πρόσβασης: 10-02-2025.

[30] *Statefulset*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. Ημερομηνία πρόσβασης: 10-02-2025.

[31] *Persistant Volumes*. https://docs.kubernetes.io/docs/concepts/storage/persistent-volumes/. Ημερομηνία πρόσβασης: 10-02-2025.

[32] *Persistant Volumes Provisioning*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#provisioning. Ημερομηνία πρόσβασης: 10-02-2025.

[33] *Storage Classes*. https://docs.kubernetes.io/docs/concepts/storage/storage-classes/. Ημερομηνία πρόσβασης: 10-02-2025.

[34] *Persistant Volume Claims*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims. Ημερομηνία πρόσβασης: 10-02-2025.

[35] *Persistant Volumes Volume Mode*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#volume-mode. Ημερομηνία πρόσβασης: 10-02-2025.

[36] *Persistant Volumes Access Mode*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#access-modes. Ημερομηνία πρόσβασης: 10-02-2025.

[37] *Persistant Volume Claims*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims. Ημερομηνία πρόσβασης: 10-02-2025.

[38] *Storage Classes Provisioner*. https://kubernetes.io/docs/concepts/storage/storage-classes/#provisioner. Ημερομηνία πρόσβασης: 10-02-2025.

[39] *Storage Classes Reclaim Policy*. https://kubernetes.io/docs/concepts/storage/storage-classes/#reclaim-policy. Ημερομηνία πρόσβασης: 10-02-2025.

[40] *Storage Classes Volume binding mode*. https://kubernetes.io/docs/concepts/storage/storage-classes/#volume-binding-mode. Ημερομηνία πρόσβασης: 10-02-2025.

[41] *Storage Classes Allowed Topologies*. https://kubernetes.io/docs/concepts/storage/storage-classes/#allowed-topologies. Ημερομηνία πρόσβασης: 10-02-2025.

[42] *Persistant Volumes Reclaiming*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#reclaiming. Ημερομηνία πρόσβασης: 10-02-2025.

[43] *Persistant Volumes Node Affinity*. https://kubernetes.io/docs/concepts/storage/persistent-volumes/#node-affinity. Ημερομηνία πρόσβασης: 10-02-2025.

[44] *Statefulset Pod Identity*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#pod-identity. Ημερομηνία πρόσβασης: 10-02-2025.

[45] *Using StatefulSets*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#using-statefulsets. Ημερομηνία πρόσβασης: 10-02-2025.

[46] *Statefulset Limitation*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#limitations. Ημερομηνία πρόσβασης: 10-02-2025.

[47] *Statefulset PersistentVolumeClaim retention*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#persistentvolumeclaim-retention. Ημερομηνία πρόσβασης: 10-02-2025.

[48] *A Guide to Horizontal vs Vertical Scaling*. https://www.mongodb.com/resources/basics/horizontal-vs-vertical-scaling. Ημερομηνία πρόσβασης: 10-02-2025.

[49] *Horizontal scaling*. https://wa.aws.amazon.com/wat.concept.horizontal-scaling.en.html. Ημερομηνία πρόσβασης: 10-02-2025.

[50] *Vertical vs. horizontal scaling: What's the difference and which is better?* https://www.cockroachlabs.com/blog/vertical-scaling-vs-horizontal-scaling/. Ημερομηνία πρόσβασης: 10-02-2025.

[51] *Vertical vs. horizontal scaling: What's the difference and which is better? TL;DR*. https://www.cockroachlabs.com/blog/vertical-scaling-vs-horizontal-scaling/#TL;DR. Ημερομηνία πρόσβασης: 10-02-2025.

[52] *Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Ημερομηνία πρόσβασης: 10-02-2025.

[53] *Resize CPU and Memory Resources assigned to Containers*. https://kubernetes.io/docs/tasks/configure-pod-container/resize-container-resources/. Ημερομηνία πρόσβασης: 10-02-2025.

[54] *Horizontal Pod Autoscaler Support for resource metrics*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-resource-metrics. Ημερομηνία πρόσβασης: 10-02-2025.

[55] *Horizontal Pod Autoscaler Scaling on custom metrics*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-on-custom-metrics. Ημερομηνία πρόσβασης: 10-02-2025.

[56] *Horizontal Pod Autoscaler Scaling on custom multiple*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#scaling-on-multiple-metrics. Ημερομηνία πρόσβασης: 10-02-2025.

[57] *Horizontal Pod Autoscaler Configurable scaling behavior*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#configurable-scaling-behavior. Ημερομηνία πρόσβασης: 10-02-2025.

[58] *Horizontal Pod Autoscaler Stabilization window*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#stabilization-window. Ημερομηνία πρόσβασης: 10-02-2025.

[59] Chunmao Jiang και Peng Wu. *A Fine-Grained Horizontal Scaling Method for Container-Based Cloud*. *Scientific Programming*, 2021(1):6397786, 2021.

[60] *Cassandra Overview*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/overview.html. Ημερομηνία πρόσβασης: 10-02-2025.

[61] Χρυσόπουλος, Μιλτιάδης Β. *Προσαρμοστικός διαμοιρασμός πόρων σε υπολογιστικά νέφη με χρήση περιεκτών και βαθιάς ενισχυτικής μάθησης*. Διπλωματική εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 2022.

[62] *Cassandra Overview Featured*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/overview.html#features. Ημερομηνία πρόσβασης: 10-02-2025.

[63] *Cassandra Query Language*. https://cassandra.apache.org/doc/3.11/cassandra/cql/index.html. Ημερομηνία πρόσβασης: 10-02-2025.

[64] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall και Werner Vogels. *Dynamo: amazon's highly available key-value store*. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[65] *Dynamo*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html. Ημερομηνία πρόσβασης: 10-02-2025.

[66] *Dynamo Dataset Partitioning: Consistent Hashing*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#dataset-partitioning-consistent-hashingl. Ημερομηνία πρόσβασης: 10-02-2025.

[67] *Dynamo Multiple Tokens per Physical Node (vnodes)*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#multiple-tokens-per-physical-node-vnodes. Ημερομηνία πρόσβασης: 10-02-2025.

[68] *Dynamo Replication Strategy*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#replication-strategy. Ημερομηνία πρόσβασης: 10-02-2025.

[69] Eric Brewer. *Towards robust distributed systems*. σελίδα 7, 2000.

[70] Seth Gilbert και Nancy Lynch. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. *SIGACT News*, 33(2):51–59, 2002.

[71] *What is CAP?* https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#what-is-cap. Ημερομηνία πρόσβασης: 10-02-2025.

[72] *Cassandra Guarantees*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html. Ημερομηνία πρόσβασης: 10-02-2025.

[73] *Cassandra High Scalability*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#high-scalability. Ημερομηνία πρόσβασης: 10-02-2025.

[74] *Cassandra High Availability*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#high-availability. Ημερομηνία πρόσβασης: 10-02-2025.

[75] *Cassandra Durability*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#durability. Ημερομηνία πρόσβασης: 10-02-2025.

[76] *Cassandra Eventual Consistency*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#eventual-consistency. Ημερομηνία πρόσβασης: 10-02-2025.

[77] *Cassandra Lightweight transactions with linearizable consistency*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#lightweight-transactions-with-linearizable-consistency. Ημερομηνία πρόσβασης: 10-02-2025.

[78] *Cassandra Batched Writes*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#batched-writes. Ημερομηνία πρόσβασης: 10-02-2025.

[79] *Cassandra Secondary Indexes*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/guarantees.html#secondary-indexes. Ημερομηνία πρόσβασης: 10-02-2025.

[80] *Dynamo Tunable Consistency*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#tunable-consistency. Ημερομηνία πρόσβασης: 10-02-2025.

[81] *How is the consistency level configured?* https://docs.datastax.com/en/dse/6.9/architecture/database-internals/configure-consistency.html. Ημερομηνία πρόσβασης: 10-02-2025.

[82] *Configuring data consistency*. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/dml_config_consistency_c.html. Ημερομηνία πρόσβασης: 10-02-2025.

[83] *Dynamo Ring Membership and Failure Detection*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#ring-membership-and-failure-detection. Ημερομηνία πρόσβασης: 10-02-2025.

[84] *Dynamo Gossip*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/dynamo.html#gossip. Ημερομηνία πρόσβασης: 10-02-2025.

[85] Xavier D?fago, Naohiro Hayashibara, Rami Yared και Takuya Katayama. *The Accrual Failure Detector* . *Reliable Distributed Systems, IEEE Symposium on*, σελίδες 66–78, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[86] *Cassandra Storage Engine*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/storage_engine.html. Ημερομηνία πρόσβασης: 10-02-2025.

[87] *How is data written?* https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlHowDataWritten.html. Ημερομηνία πρόσβασης: 10-02-2025.

[88] *CommitLog*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/storage_engine.html#commit-log. Ημερομηνία πρόσβασης: 10-02-2025.

[89] *Memtables*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/storage_engine.html#memtables. Ημερομηνία πρόσβασης: 10-02-2025.

[90] *SSTables*. https://cassandra.apache.org/doc/3.11/cassandra/architecture/storage_engine.html#sstables. Ημερομηνία πρόσβασης: 10-02-2025.

[91] *How is data read?* https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlAboutReads.html. Ημερομηνία πρόσβασης: 10-02-2025.

[92] *YCSB*. https://github.com/brianfrankcooper/YCSB. Ημερομηνία πρόσβασης: 10-02-2025.

[93] *YCSB Running a Workload*. https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload. Ημερομηνία πρόσβασης: 10-02-2025.

[94] *Prometheus Exporters and integrations*. https://prometheus.io/docs/instrumenting/exporters/. Ημερομηνία πρόσβασης: 10-02-2025.

[95] *Grafana Documentation*. https://grafana.com/docs/grafana/latest/. Ημερομηνία πρόσβασης: 10-02-2025.

[96] *minikube*. https://minikube.sigs.k8s.io/docs/. Ημερομηνία πρόσβασης: 10-02-2025.

[97] *MicroK8s*. https://microk8s.io/. Ημερομηνία πρόσβασης: 10-02-2025.

[98] *Kubeadm*. https://kubernetes.io/docs/reference/setup-tools/kubeadm/. Ημερομηνία πρόσβασης: 10-02-2025.

[99] *Kubespray*. https://kubespray.io/#/. Ημερομηνία πρόσβασης: 10-02-2025.

[100] *Kubespray vs Kubeadm*. https://kubespray.io/#/docs/getting_started/comparisons?id=kubespray-vs-kubeadm. Ημερομηνία πρόσβασης: 10-02-2025.

[101] *Kubernetes Installation Tutorial: Kubespray*. https://dev.to/admantium/kubernetes-installation-tutorial-kubespray-46ek. Ημερομηνία πρόσβασης: 10-02-2025.

[102] *Rook Prerequisites*. https://docs.ceph.com/en/quincy/cephfs/index.html. Ημερομηνία πρόσβασης: 10-02-2025.

[103] *Rook*. https://rook.io/docs/rook/v1.9/Getting-Started/intro/. Ημερομηνία πρόσβασης: 10-02-2025.

[104] *Rook Prerequisites*. https://rook.io/docs/rook/v1.9/Getting-Started/Prerequisites/prerequisites/. Ημερομηνία πρόσβασης: 10-02-2025.

[105] *cassandra-prometheus-k8s Repository*. https://github.com/geoangelotti/cassandra-prometheus-k8s. Ημερομηνία πρόσβασης: 10-02-2025.

[106] *Example: Deploying Cassandra with a StatefulSet*. https://kubernetes.io/docs/tutorials/stateful-application/cassandra/. Ημερομηνία πρόσβασης: 10-02-2025.

[107] *Capacity planning and hardware selection for Apache Cassandra implementations*. https://docs.datastax.com/en/cassandra-oss/planning/planning/ossCapacityPlanning.html. Ημερομηνία πρόσβασης: 10-02-2025.

[108] *Amazon EC2 T2 Instances*. https://aws.amazon.com/ec2/instance-types/t2/. Ημερομηνία πρόσβασης: 10-02-2025.

[109] *Tuning Java resources*. https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/operations/opsTuneJVM.html#Determiningtheheapsize. Ημερομηνία πρόσβασης: 10-02-2025.

[110] *nodetools decommission*. https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/tools/toolsDecommission.html. Ημερομηνία πρόσβασης: 10-02-2025.

[111] *nodetools drain*. https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/tools/toolsDrain.html. Ημερομηνία πρόσβασης: 10-02-2025.

[112] *Metrics Server*. https://github.com/kubernetes-sigs/metrics-server. Ημερομηνία πρόσβασης: 10-02-2025.

[113] *YCSB Extention*. https://github.com/geoangelotti/YCSB. Ημερομηνία πρόσβασης: 10-02-2025.

[114] *Helm Architecture*. https://helm.sh/docs/topics/architecture/. Ημερομηνία πρόσβασης: 10-02-2025.

[115] *kube-prometheus-stack*. https://github.com/prometheus-community/helm-charts/blob/main/charts/kube-prometheus-stack/README.md. Ημερομηνία πρόσβασης: 10-02-2025.

[116] *Cassandra Exporter*. https://bitnami.com/stack/cassandra-exporter. Ημερομηνία πρόσβασης: 10-02-2025.

[117] *Querying Prometheus*. https://prometheus.io/docs/prometheus/latest/querying/basics/. Ημερομηνία πρόσβασης: 10-02-2025.

Συντομογραφίες - Αρκτικόλεξα - Ακρωνύμια

βλ.	βλέπε
ΕΜΠ	Εθνικό Μετσόβιο Πολυτεχνείο
κ.λπ.	και λοιπά
κ.ο.κ	και ούτω καθεξής
π.χ.	παραδείγματος χάριν
API	Application Programming Interface
AWS	Amazon Web Services
CAP	Consistency, Availability, Partition Tolerance
CNCF	Cloud Native Computing Foundation
CPU	Central Proccessing Unit
CQL	Cassandra Query Language
etc.	et cetera
GB	GigaBytes
GiB	Gibibytes
HPA	Horizontal Pod Autoscaler
HTTP	Hypertext Transfer Protocol
i.e.	that is
IO	Input Output
IP	Internet Protocol
JMX	Java Management Extension
LVM	Logical Volume Manager
MB	MegaBytes
PromQL	Prometheus Query Language
PV	Persistant Volume
PVC	Persistant Volume Claim
RDBMS	Relational Database Management System
SQL	Structured Query Language
UI	User Interface
YCSB	Yahoo Cloud Serving Benchmark

Απόδοση ξενόγλωσσων όρων

Απόδοση	Ξενόγλωσσος όρος
δέσμευση μόνιμου τόμου	persistant volume claim
διάθεση	deployment
διακομιστής	server
διεπαφή προγραμματισμού εφαρμογών	application programming interface
διεπαφή χρήστη εμπρόσθιου άκρου	frontend ui
κάψουλα	pod
κλάση αποθήκευσης	storage class
κόμβος	node
μεγάλα δεδομένα	big data
μόνιμος τόμος	persistant volume
οριζόντιος αυτόματης κλιμακωτής κάψουλας	horizontal pod autoscaler
περιέκτης	container
σύζευξη	join
σύμπλεγμα κόμβων	cluster
σύνολο με κατάσταση	statefulset
χρήστης	client