ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΣΧΟΛΗ ΨΗΦΙΑΚΗΣ ΤΕΧΝΟΛΟΓΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΜΑΤΙΚΗΣ

Π.Μ.Σ. «ΠΛΗΡΟΦΟΡΙΚΗ ΚΑΙ ΤΗΛΕΜΑΤΙΚΗ»

Κατεύθυνση 1: Τεχνολογίες και Εφαρμογές Ιστού

Τίτλος Εργασίας: Εξισορρόπηση φόρτου σε ένα web service χρησιμοποιώντας εικονικές μηχανές σε περιβάλλον Docker

Διπλωματική Εργασία

Όνομα φοιτητή: Κωνσταντίνος Μπαλιώτης

Αθήνα, 2018

ΧΑΡΟΚΟΠΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΣΧΟΛΗ ΨΗΦΙΑΚΗΣ ΤΕΧΝΟΛΟΓΙΑΣ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΜΑΤΙΚΗΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ ΤΜΗΜΑΤΟΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΜΑΤΙΚΗΣ

Κατεύθυνση 1: Τεχνολογίες και Εφαρμογές Ιστού

Τριμελής Εξεταστική Επιτροπή: Κωνσταντίνος Τσερπές(Επιβλέπων),

Βασίλειος Δαλάκας,

Δημοσθένης Αναγνωστόπουλος,

Ο Κωνσταντίνος Μπαλιώτης δηλώνω υπεύθυνα ότι:

1) Είμαι ο κάτοχος των πνευματικών δικαιωμάτων της πρωτότυπης αυτής εργασίας και από όσο γνωρίζω η εργασία μου δε συκοφαντεί πρόσωπα, ούτε προσβάλει τα πνευματικά δικαιώματα τρίτων.

2) Αποδέχομαι ότι η ΒΚΠ μπορεί, χωρίς να αλλάξει το περιεχόμενο της εργασίας μου, να τη διαθέσει σε ηλεκτρονική μορφή μέσα από τη ψηφιακή Βιβλιοθήκη της, να την αντιγράψει σε οποιοδήποτε μέσο ή/και σε οποιοδήποτε μορφότυπο καθώς και να κρατά περισσότερα από ένα αντίγραφα για λόγους συντήρησης και ασφάλειας.

Στην σύζυγο μου Σοφία και τα τέκνα μου Σταύρο και Μαρίνο για την υπομονή και την υποστήριξής τους

ΕΥΧΑΡΙΣΤΙΕΣ

Με το παρόν θα ήθελα να ευχαριστήσω ιδιαίτερα τους καθηγητές μου Κωνσταντίνο Τσερπέ, Ηρακλή Βαρλάμη, Δημήτριο Μιχαήλ, Μάρα Νικολαίδου, Δημοσθένη Αναγνωστόπουλο και όσους με συνέδραμαν στο συναρπαστικό αυτό ταξίδι γνώσης στο επιστημονικό χώρο των τεχνολογιών και εφαρμογών Ιστού.

Πίνακας περιεχομένων

Πίνακας Εικόνων

Περίληψη

Η επεκτασιμότητα και υψηλή διαθεσιμότητα των συστοιχιών εξυπηρετητών ιστού προϋποθέτουν μια αποτελεσματική πολιτική κατανομής φόρτου. Αυτό θα μπορούσε να υλοποιηθεί όταν υπολογίζεται συνεχώς μια συγκεκριμένη μετρική που εκφράζει την μεταβολή του φόρτου που μπορεί να διαχειρίζεται ο εκάστοτε εξυπηρετητής. Στην εργασία αυτή, διερευνήθηκε η δυνατότητα παρακολούθησης μιας μετρικής προκειμένου όταν παρίσταται ανάγκη να γίνεται αυτοματοποιημένη κλιμάκωση της εξισορρόπησης του φόρτου (Auto-Scaling Load Balancing). Η εξέταση επικεντρώθηκε σε μια σημαντική οικογένεια μετρικών που σχετίζονται με τα HTTP αιτήματα. Ο μηχανισμός που επιλέχθηκε για την συγκέντρωση των τιμών των μετρικών αυτών είναι το Prometheus. Στο πλαίσιο αυτό πραγματοποιήθηκαν δοκιμές σε δύο διαφορετικούς τύπους σεναρίων. α) εξισορρόπηση φόρτου σε συστοιχία των nginx εξυπηρετητών με χρήση των αλγορίθμων Round Robin (RR) και Least Connections (LC) χωρίς την δυνατότητα αυτοματοποιημένης κλιμάκωσης β) εξισορρόπηση φόρτου στην συστοιχία των nginx εξυπηρετητών με χρήση των αλγορίθμων Round Robin (RR) και Least Connections (LC) με πλήρη εφαρμογή αυτοματοποιημένης κλιμάκωσης. Μια επιπλέον δυαδική διάσταση που διέπει τα εξεταζόμενα σενάρια είναι το αν η λειτουργία γίνεται σε συνθήκες ισορροπίας ή σε συνθήκες διαταραχής. Για την επίτευξη της αυτοματοποιημένης κλιμάκωσης χρησιμοποιείται το Jenkins ένα εργαλείο Continuous Integration μαζί με έναν ειδικά προσαρμοσμένο HAproxy. Τα πειραματικά αποτελέσματα δείχνουν πως βελτιώνεται η απόδοση του συστήματος με την εφαρμογή αυτού του προσαρμοζόμενου μοντέλου εξισορρόπησης φορτίου όταν ο αριθμός των ενεργών στιγμιότυπων διπλασιάζεται. Τέλος θα πρέπει να σημειωθεί πως όλη η εργασία

είναι χτισμένη στο Docker Swarm δηλαδή σε μια γηγενή εφαρμογή υπολογιστικού νέφους, που επιτρέπει την εύκολη μετάβαση του σ' ένα τέτοιο περιβάλλον.

Abstract

In order to achieve scalability and high availability of services offered by web servers, an effective load balancing policy is required. The aforementioned policy could be implemented as a result of continuous monitoring of an ad-hoc chosen metric. This metric formulated the load change that each web server can handle. In addition, the usage of the monitored metric was examined as an Auto-Scaling Load Balancing activation mechanism. Eventually, the review was focus on an important class of metrics related with http requests. To this end, the registration and processing of metric values is achieved by using an open source time series database called Prometheus. Within the above context, a set of stress test scenarios have been implemented.

a) A nginx web server cluster load balancing based on Round Robin (RR) and Least Connections (LC) algorithms without Auto-Scaling capability.
b) A nginx web server cluster load balancing based on Round Robin (RR) and Least Connections (LC) algorithms with fully Automated Scaling capability.

Furthermore, a dual, cross-cutting dimension was taken in to account based on whether the cluster operated under conditions of high imbalance. The automated instance scaling was achieved by using a Continuous Integration tool such as Jenkins along with a specially adapted HAproxy version. The experimental results showed an improved performance when the auto-scaling load balancing capability is applied. In fact, when the number of instances is doubled, the lack of performance under stress is compensated.

Finally, it should be noted that the whole system was build on one node Docker Swarm. The latter is a cloud native application and therefore it allows a smooth migration in such an environment when a business case arises.

1. Εισαγωγή

Για να επιτευχθεί επεκτασιμότητα και υψηλή διαθεσιμότητα των υπηρεσιών που προσφέρουν οι συστοιχίες εξυπηρετητών ιστού, απαιτείται μια αποτελεσματική πολιτική κατανομής φόρτου. Για να πραγματοποιηθεί αυτή η πολιτική είναι κρίσιμο να κατανέμεται ο φόρτος κατά βέλτιστο τρόπο στους διαθέσιμους εξυπηρετητές. Αυτό θα μπορούσε να υλοποιηθεί όταν υπολογίζεται συνεχώς μια κρίσιμη μετρική που εκφράζει την μεταβολή του φόρτου που είναι δυνατόν να διαχειρίζεται ένας εξυπηρετητής ιστού. Στην εργασία αυτή, εξετάστηκε η δυνατότητα παρακολούθησης μιας μετρικής κλειδιού, προκειμένου όταν παρίσταται ανάγκη, δηλαδή όταν ξεπεραστούν κάποια συγκεκριμένα όρια, να εκτελείται αυτοματοποιημένη κλιμάκωση της εξισορρόπησης του φόρτου (Auto-Scaling Load Balancing). Διαπιστώθηκε πως μια σημαντική οικογένεια παραμέτρων είναι οι μετρικές που σχετίζονται με τα http αιτήματα.

Ειδικότερα επιλέχθηκε ως εξυπηρετητής ιστού να είναι ο nginx εξυπηρετητής ιστού, που για τους σκοπούς της εργασίας διατίθεται ως ανοικτό λογισμικό. Η ευελιξία της πλατφόρμας επέτρεψε ώστε να τοποθετηθεί ένα πρόσθετο που μας έδωσε μια σειρά χρήσιμων μετρικών μεταξύ των οποίων και την nginx_http_requests_total, που είναι η βασική μετρική που ενεργοποιεί τον μηχανισμό αυτοματοποιημένης κλιμάκωσης και αποκλιμάκωσης.

Για την συγκέντρωση των τιμών των εξεταζόμενων μετρικών χρησιμοποιήθηκε μια ανοικτού λογισμικού βάση δεδομένων χρονοσειρών, το Prometheus, ώστε να καταγράφονται μια σειρά από μετρικές, συμπεριλαμβανομένου των ανωτέρων. Αυτή συνοδεύεται από έναν μηχανισμό, τον Alertmanager, που επιτρέπει την

παραμετροποίηση και ενεργοποίηση συναγερμών που είναι απαραίτητοι για την δημιουργία ενός συστήματος αυτοματοποιημένης κλιμάκωσης και από-κλιμάκωσης. Παράλληλα αξιοποιήθηκε το grafana ένα ανοικτό περιβάλλον απεικόνισης των μετρικών που συνεργάζεται απόλυτα με το Prometheus. Για την λήψη επιπλέον χρήσιμων μετρικών παρατάχθηκαν τα cAdvisor και node-exporter που επίσης συνεργάζονται απόλυτα με το Prometheus. Για την επίτευξη της αυτοματοποιημένης κλιμάκωσης χρησιμοποιήθηκε το Jenkins ένα εργαλείο Continuous Integration σε ρόλο διαχειριστή των αιτημάτων συναγερμών.

Οι δοκιμές πραγματοποιήθηκαν σε δύο διαφορετικούς τύπους σεναρίων. α) εξισορρόπηση φόρτου στην συστοιχία των nginx web servers με εφαρμογή διαδεδομένων αλγορίθμων όπως του Round Robin (RR) και του Least Connections (LC) χωρίς την δυνατότητα αυτόματου scaling β) εξισορρόπηση φόρτου στην συστοιχία των nginx web servers με εφαρμογή των ίδιων αλγορίθμων όπως του Round Robin (RR) και του Least Connections (LC) με πλήρη εφαρμογή αυτοματοποιημένης κλιμάκωσης. Σε κάθε μια από αυτές τις περιπτώσεις εξετάστηκε μια επιπλέον παράμετρος που αφορά το αν η λειτουργία γίνεται σε συνθήκες ισορροπίας ή σε συνθήκες διαταραχής. Έτσι διαμορφώθηκαν οκτώ στοχευόμενα σενάρια.

Τα πειραματικά αποτελέσματα δείχνουν πως βελτιώνεται η απόδοση του συστήματος με την εφαρμογή αυτού του προσαρμοζόμενου μοντέλου εξισορρόπησης φορτίου όταν ο αριθμός ενεργών στιγμιότυπων τουλάχιστον διπλασιάζεται.

Επίσης είναι εντυπωσιακή η προσπάθεια του Docker Swarm να διατηρήσει την διαθεσιμότητα του συστήματος. Το γεγονός αυτό δείχνει πως το Docker Swarm είναι μια γηγενής εφαρμογή σε περιβάλλον υπολογιστικού νέφους που ικανοποιεί τις σύγχρονες επιχειρησιακές απαιτήσεις.

Η σύμβαση που ακολουθήθηκε ήταν η εγκατάσταση του σε φορητό υπολογιστή ενός κόμβου για πρακτικούς λόγους. Όμως αυτό δεν απαγορεύει την εύκολη μετάπτωση του σε περιβάλλον υπολογιστικού νέφους τουλάχιστον τριών κόμβων όπως συνιστά ο δημιουργός του.

2. Το τεχνολογικό περιβάλλον

2.1. Εξισορρόπηση φόρτου ιστού: το κεντρικό ζήτημα της εργασίας

Ο κύριος λόγος της χρήσης μιας συστοιχίας εξυπηρετητών ή αλλιώς φάρμας εξυπηρετητών είναι το γεγονός ότι οι εφαρμογές ιστού πρέπει να τρέχουν σε πολλαπλούς εξυπηρετητές ταυτόχρονα για να ανταποκριθούν στην παροχή περιεχόμενου ιστού προς τους χρήστες. Ουσιαστικά θα λέγαμε πως η εξισορρόπηση φόρτου είναι η τεχνολογική δυνατότητα αρκετών εξυπηρετητών να συμμετέχουν στην παροχή της ίδιας υπηρεσίας. Δηλαδή να εκτελούν τις ίδιες εργασίες , αφού οι πόροι του καθενός είναι πεπερασμένοι. Παράλληλα θα πρέπει η υπηρεσία αυτή να προσφέρεται στον εξωτερικό χρήστη κατά τρόπο που να μην είναι αντιληπτό ότι εξυπηρετείται από διαφορετικούς εξυπηρετητές. Είναι αναγκαίο επομένως σε ένα τέτοιο σύστημα να υπάρχει η δυνατότητα για επεκτασιμότητα, διαθεσιμότητα και διαχειρισιμότητα των πόρων ενιαία.

Για την επιτέλεση της αποστολής του, ένα τέτοιο σύστημα, θα πρέπει να ικανοποιεί την κύρια επεκτασιμότητα. Με άλλα λόγια θα πρέπει να κατανέμει το φόρτο όσο το δυνατόν ισοδύναμα όσο ο αριθμός των συμμετεχόντων εξυπηρετητών αυξάνει. Μια άλλη σημαντική ιδιότητα ενός τέτοιου συστήματος είναι η ικανότητα να ανακατευθύνει τον φόρτο σε εναλλακτικούς εξυπηρετητές. Αυτό μπορεί να συμβεί όταν ένας ή οι περισσότεροι εξυπηρετητές αποτύχουν να ανταποκριθούν στην εξυπηρέτηση του φόρτου. Συνακόλουθα, παρατηρούμε την ικανότητα που έχει ένα τέτοιο σύστημα να διατηρεί σε λειτουργία μια υπηρεσία μέχρι φυσικά έναν

προκαθορισμένο αριθμό αποτυχιών. Δηλαδή, εξασφαλίζει την διαθεσιμότητα της υποστηριζόμενης υπηρεσίας. Η διαχειρισιμότητα, αντίστοιχα, είναι στην ουσία η εργαλειοθήκη που διατίθεται ώστε να είναι γίνεται η διαχείριση μιας εφαρμογής μεταξύ των εξυπηρετητών (Gilly, Juiz, & Puigjaner, 2011).

Η εξισορρόπηση φόρτου υλοποιείται με δύο κυρίως κατηγορίες αλγορίθμων. Ο ένας αναφέρεται σε στατική εξισορρόπηση φόρτου, ενώ ο άλλος αφορά σε δυναμική εξισορρόπηση φόρτου. Η διαφορά τους εντοπίζεται κυρίως στο εξής : η στατική εξισορρόπηση φόρτου δεν απαιτείται να διατηρεί καμία κατάσταση συστήματος. Ουσιαστικά προϋποθέτει μια ex-ante εκτίμηση του φόρτου κατά την δημιουργία της συστοιχίας των εξυπηρετητών. Εκ των πραγμάτων, θα λέγαμε πως ένα τέτοιο σύστημα δεν μπορεί να ανταποκριθεί αποτελεσματικά στις περιπτώσεις που ο τρέχων φόρτος απέχει από την αρχική εκτίμηση. Το κενό αυτό που δημιουργείται έρχονται να καλύψουν οι αλγόριθμοι δυναμικής εξισορρόπησης φόρτου καθώς λαμβάνουν υπόψη το εκάστοτε φόρτο του συστήματος και αφιερώνουν τους κατάλληλους πόρους του συστήματος, ώστε αυτό να ανταποκριθεί στις αυξημένες ανάγκες. Πρόκειται, δηλαδή, για αλγόριθμους που εξαρτώνται από την κατάσταση του συστήματος. (Liu, Shang, Zhu, & Feng, 2016).

Αντιπροσωπευτικός αλγόριθμος της πρώτης κατηγορίας είναι ο Round Robin (RR) αλγόριθμος, όπου οι TCP συνδέσεις ανατίθεται εκ περιτροπής στους εξυπηρετητές της συστοιχίας. Δηλαδή, το πρώτο αίτημα σύνδεσης να ανατίθεται στον πρώτο εξυπηρετητή, το δεύτερο αίτημα στον δεύτερο εξυπηρετητή κ.ο.κ. Καθώς τα αιτήματα σύνδεσης ανατίθενται σειριακά ανάμεσα στους εξυπηρετητές, κάθε εξυπηρετητής λαμβάνει το ίδιο αριθμό αιτημάτων ανεξαρτήτως πόσο γρήγορα τα εξυπηρετεί. Για τον λόγο αυτό ο αλγόριθμος αυτός είναι ίσως η καλύτερη μέθοδος για την εξισορρόπηση φόρτου για ομοιογενείς εξυπηρετητές. Όταν είναι εκ των προτέρων γνωστή μια φυσική ανομοιογένεια των εξυπηρετητών τότε μπορεί να χρησιμοποιηθεί μια παραλλαγή του που ονομάζεται ζυγισμένος(Weighted) Round

Robin (RR). Φυσικά, εναπόκειται στον διαχειριστή να καθορίσει το τι ποσοστό της κίνησης θα διοχετευτεί και σε ποιόν εξυπηρετητή.

Αντίστοιχα αντιπροσωπευτικός των δυναμικών αλγόριθμών εξισορρόπησης φόρτου (με παρακολούθηση της κατάστασης) είναι ο Least Connection (LC) δηλαδή των ελάχιστων συνδέσεων. Αυτός πρακτικά αναθέτει τις συνδέσεις στον εξυπηρετητή με τις ελάχιστες συνδέσεις. Αυτή η δυναμική προσέγγιση απαιτεί να είναι γνωστός ο αριθμός των υφιστάμενων συνδέσεων μεταξύ των πελατών και του εκάστοτε εξυπηρετητή ιστού στην εξεταζόμενη συστοιχία. Μια παραλλαγή του είναι ο Weighted Least-Connection (WLC). Στην περίπτωση αυτή εκτός από την μέτρηση των υφιστάμενων συνδέσεων λαμβάνεται υπόψη και μια ποσόστωση(βάρος) που έχει ανατεθεί στον εκάστοτε εξυπηρετητή. Τυπικά οι εξυπηρετητές με υψηλότερη ποσόστωση καλούνται να διεκπεραιώσουν ένα υψηλότερο ποσοστό από τους υπόλοιπους εξυπηρετητές σε αναλογική κλίμακα (Gilly, Juiz, & Puigjaner, 2011).

Επάνω στους προαναφερόμενους αλγορίθμους που διατίθενται εξορισμού από τους πιο δημοφιλείς εξυπηρετητές φόρτου (NGINX, 2017),(HAproxy), έχουν προταθεί αλγόριθμοι που βελτιώνουν την απόδοση σε αυξημένες απαιτήσεις φόρτου. Έτσι υπάρχουν προσεγγίσεις που μπορούν να εφαρμοστούν επάνω σε συστοιχίες με ομοιογενείς εξυπηρετητές και εξετάζεται η απόδοση τους με τους προαναφερόμενους Round Robin (RR) και Least Connection (LC) (Maluk & M.A., 2015).

Αντίστοιχα όταν υπεισέρχεται κάποιου είδους ανομοιογένεια ανάμεσα στους εξυπηρετητές τότε η ερευνητική πρόταση του βελτιωμένου αλγορίθμου δύναται να συγκριθεί με τις αντίστοιχες εκδόσεις με ποσόστωση των βασικών αλγορίθμων. Αυτή η περίπτωση έχει ιδιαίτερο νόημα στα περιβάλλοντα διαχείρισης επεξεργασίας δεδομένων μεγάλης κλίμακας(Big Data) όπου ενδέχεται οι υποκείμενες υποδομές να είναι ανομοιογενής σε επίπεδο φυσικών

χαρακτηριστικών (Liu, Shang, Zhu, & Feng, 2016). Πάντως η γενική μέθοδος που ακολουθείται είναι να εξετάζεται ποιος εξυπηρετητής έχει τον χαμηλότερο φόρτο. Δηλαδή, δημιουργείται κάποιος μηχανισμός που παρακολουθεί ανά εξυπηρετητή την κατανάλωση των πόρων και τις δυνατότητες που απομένουν και έτσι ανατίθενται οι συνδέσεις σε εκείνον τον εξυπηρετητή που έχει τους περισσότερους διαθέσιμους πόρους. Με άλλα λόγια θα λέγαμε ότι ακολουθείται μια Least Loaded (LL) αλγοριθμική προσέγγιση. Για λόγους πληρότητας αναφοράς θα πρέπει να σημειωθεί πως είναι δυνατόν να γίνεται ανάθεση φόρτου στους εξυπηρετητές και με τυχαίο, μη ντετερμινιστικό τρόπο (Gilly, Juiz, & Puigjaner, 2011).

2.2. Γενικά για τους Container

Σήμερα, στους προγραμματιστές και τους διαχειριστές κέντρων δεδομένων, η χρήση container είναι μια εξαιρετικά δημοφιλής μέθοδος διαχωρισμού μιας εφαρμογής από το λειτουργικό σύστημα και την φυσική υποδομή υλισμικού που χρησιμοποιείται για να διασυνδεθεί στο δίκτυο. Σε μια σχετικά πρόσφατη έρευνα του 2016 από την Docker Inc. έδειξε πως πάνω από το 60% των ερωτώμενων χρησιμοποιούν Docker container στις παραγωγικές τους εφαρμογές (Docker, 2016). Ενώ ήδη τον Οκτώβριο του 2016 είχε συμπληρωθεί ο αριθμός των 6 δις εξαγωγές(pulls) (Docker Pulls, 2016).

Προχωρώντας πιο πέρα θα λέγαμε πως οι container εφαρμογής είναι ένας μηχανισμός που χρησιμοποιείται στο περιβάλλον ενός λειτουργικού συστήματος για να απομονώσει τις εφαρμογές που τρέχουν σ' αυτό. Παρ' ότι σήμερα η έννοια του container γνωρίζει μια άνθηση εντούτοις πρόκειται για παλιά έννοια. Ήδη από το 1998, το FreeBSD έχει χαρακτηριστικά απομόνωσης των εφαρμογών όπως επίσης και το Solaris Zones το 2001 αλλά περιορίζονταν στο λειτουργικό σύστημα Solaris OS (Mouat, 2015).

Εννοιολογικά ένας container μοιάζει με το jail σύστημα αρχείων που μπορεί να δημιουργηθεί με ένα εργαλείο όπως το chroot σε μια μηχανή με εγκατεστημένο λειτουργικό τύπου UNIX. Το chroot μπορεί να χρησιμοποιηθεί για να δημιουργηθεί ένα εικονικό περιβάλλον με το δικό του κατάλογο root. Οι εφαρμογές στο περιβάλλον chroot δεν μπορούν να έχουν πρόσβαση πέραν από τον κατάλογο των αρχείων που τους έχει ανατεθεί. Για τον λόγο αυτό το chroot περιβάλλον είναι γνωστό και ως chroot jail ή jail σύστημα αρχείων (Hope, 2017). Ο βασικός σκοπός της χρήσης των container είναι να χωριστούν οι υπηρεσίες του λειτουργικού συστήματος (μαζί με την μνήμη, την χωρητικότητα μέχρι και το δίκτυο), έτσι ώστε η εκάστοτε εφαρμογή να μπορεί να τρέξει μέσα στο δικό της **διακριτό περιβάλλον χωρίς να επιβαρύνει τις άλλες εφαρμογές που τρέχουν στο ίδιο φυσικό μηχάνημα**. Το ενδιαφέρον είναι ότι ο container ενεργοποιείται από τον πυρήνα του λειτουργικού συστήματος και παρέχει ένα εικονικό χώρο για την εφαρμογή που καλείται να τρέξει σ' αυτό (Hogg, 2014). Αν και υπάρχουν διάφοροι τύποι container, οι Docker container είναι η κυρίαρχη και η πιο αντιπροσωπευτική πλατφόρμα και για τον λόγο αυτό εφεξής θα αναφερόμαστε ουσιαστικά στους Docker container (Mouat, 2015). Κάθε container εφαρμογής φιλοξενεί μια συγκεκριμένη εφαρμογή λογισμικού που μπορεί να εκτελείται χωρίς να επηρεάζει τις άλλες. Έτσι δημιουργείται η αίσθηση ότι μπορούν να τρέχουν παράλληλα δηλαδή η κάθε μια στον δικό της container επάνω στην ίδια μηχανή.

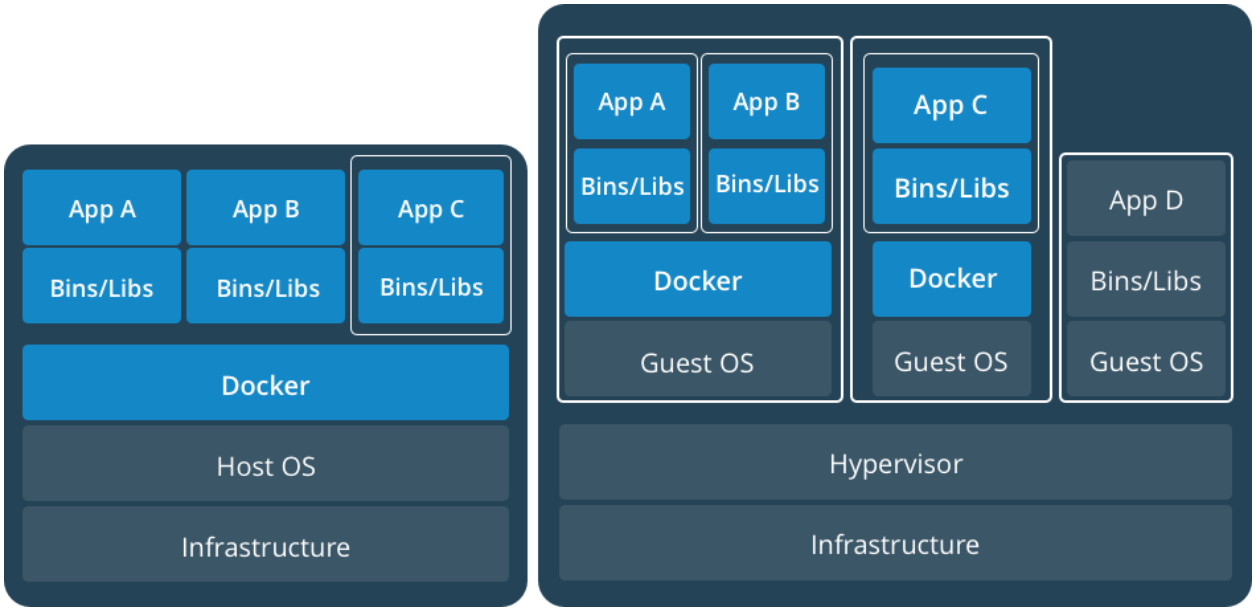
Για να επιτευχθεί αυτό ένας container εφαρμογής συνήθως:

- περιέχει ένα ξεχωριστό στιγμιότυπο της εικόνας μιας εφαρμογής λογισμικού,
- ενδέχεται να περιέχει τις δικές τις βιβλιοθήκες και ρυθμίσεις υπηρεσιών,
- μπορεί, εφόσον είναι κατάλληλα ρυθμισμένη να επαναχρησιμοποιεί βιβλιοθήκες και να διαμοιράζεται δεδομένα ανάμεσα σε container,

- διαθέτει για αποκλειστική χρήση τμήμα του δικτύου, της μνήμης και του συστήματος αρχείων,
- μοιράζεται το ίδιο λειτουργικό σύστημα με άλλους container που φιλοξενούνται στο ίδιο φυσικό μηχάνημα,
- Πολλαπλοί container βασίζονται στην ίδια εικόνα και κατά συνέπεια πολλαπλές εκδοχές της ιδίας εικόνας είναι δυνατόν να τρέχουν στο ίδιο μηχάνημα (Nickoloff, 2016).

2.3. Τα πλεονεκτήματα των Docker container

Οι χρήστες αναμένουν από τις εφαρμογές να είναι πάντοτε διαθέσιμες, επεκτάσιμες και δια-λειτουργικές (Lefler, 2014). Προκειμένου η εφαρμογή να είναι πάντα διαθέσιμη θα πρέπει το λογισμικό της να μπορεί να εκτελείται ανεξάρτητα από την υποκείμενη υποδομή. Η εφαρμογή θα πρέπει να είναι διαθέσιμη ακόμη και στην περίπτωση που μια φυσική μηχανή σταματήσει την λειτουργία της. Οι container εφαρμογών λύνουν το πρόβλημα αυτό, για τους προγραμματιστές, καθώς εύκολα ενημερώνονται και μεταφέρονται από μηχανή σε μηχανή. Για παράδειγμα μπορούν να μεταφερθούν από το στάδιο της ανάπτυξης, στο στάδιο των δοκιμών και τέλος στο στάδιο της παραγωγικής λειτουργίας. Έτσι η διάρκεια ζωής της ανάπτυξης λογισμικού σμικρύνεται σημαντικά. Παράλληλα, επειδή οι container εφαρμογών διαμοιράζονται τους πόρους στο επίπεδο εφαρμογής αντί για το επίπεδο λειτουργικού συστήματος (δηλαδή της στοίβας που περιλαμβάνει το επίπεδο του hypervisor(Εικόνα 1)) τα κέντρα δεδομένων μπορούν να υποστηρίξουν μεγαλύτερο πληθυσμό εφαρμογών. Βασικός λόγος είναι η απελευθέρωση πόρων από την διακοπή της χρήσης και συντήρησης πολλαπλών αντιγράφων εικονικών λειτουργικών συστημάτων.

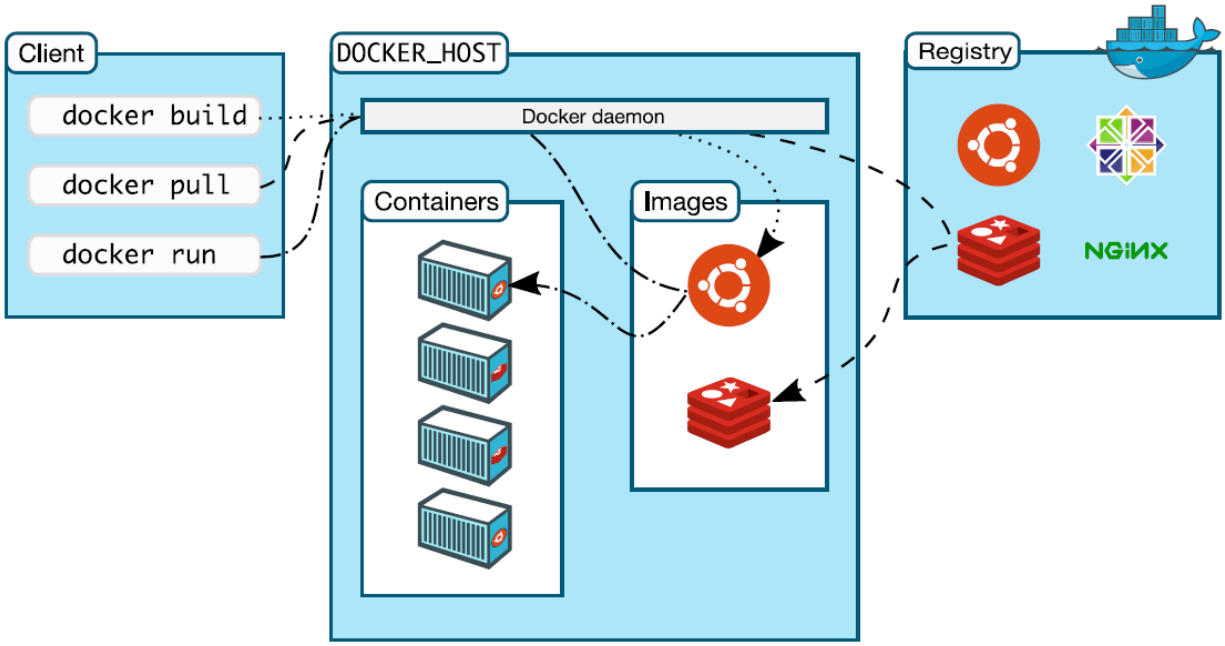


Εικόνα 1 Αρχιτεκτονικές διαφορές Container & Virtual Machine

(Πηγή: https://www.Docker.com/what-container)

2.4. Η αρχιτεκτονική λειτουργία του Docker

Το Docker είναι ένα περιβάλλον ανοικτού λογισμικού που αρχικά αναπτύχτηκε για το Linux γρήγορα όμως υποστηρίχτηκε τόσο σε windows όσο και σε OS X. Το Docker εκμεταλλεύεται την δυνατότητα του Linux να επιτυγχάνει την απομόνωση των πόρων σε ένα στεγανοποιημένο και εικονικό περιβάλλον μέσα στο οποίο είναι δυνατόν να τρέξει μια εφαρμογή. Αυτό είναι εννοιολογικά το ίδιο με το αποτέλεσμα του εργαλείου chroot δηλαδή την δημιουργία ενός απομονωμένου και προστατευμένου συστήματος αρχείων. Η διαφορά εδώ είναι ότι δεν περιορίζεται στο σύστημα αρχείων αλλά προχωρά στο να συμπεριλάβει και άλλους υπολογιστικούς πόρους όπως μνήμης και δικτύου.



Εικόνα 2: Η Αρχιτεκτονική Docker

(Πηγή https://docs.Docker.com/engine/Docker-overview/#what-can-i-use-Docker-for)

Για να κατανοηθεί καλύτερα η αρχιτεκτονική του Docker θα πρέπει αρχικά να αποσαφηνιστεί μια άλλη θεμελιώδης αρχιτεκτονική προσέγγιση. Πρόκειται για τις **Μικρο-υπηρεσίες (Microservices)**. Δηλαδή μια μέθοδος ανάπτυξης λογισμικού κατά την οποία μια εφαρμογή αναπτύσσεται σαν ένα σύνολο από μικρές υπηρεσίες. Κάθε μια από αυτές τρέχει σε μια διεργασία και επικοινωνούν με έναν ελαφρύ μηχανισμό πόρων, συχνά ένα API με πρωτόκολλο HTTP. Αυτές οι υπηρεσίες προσδιορίζονται από τις επιχειρησιακές ανάγκες της εφαρμογής και παρατάσσονται αυτόνομα από έναν αυτοματοποιημένο μηχανισμό ενορχήστρωσης. Διατηρείται μια ελάχιστη βάση κεντρικής διαχείρισης. Ενώ συχνά μπορούν να είναι γραμμένες σε διαφορετικές προγραμματιστικές γλώσσες και να χρησιμοποιούν διαφορετικές τεχνολογίες αποθήκευσης (Fowler & Levis, 2014).

Λαμβάνοντας υπόψη τα ανωτέρω, στην συνέχεια, θα αναφερθούμε στις τρείς κρισιμότερες λειτουργίες της πλατφόρμας του Docker που αποτελέσαν και το λογότυπο του Docker:

Build: Η λειτουργία αυτή του Docker επιτρέπει την σύνθεση μιας εφαρμογής από μικρο-υπηρεσίες, χωρίς την ανησυχία για την ύπαρξη ασυμβατοτήτων ανάμεσα στα περιβάλλοντα ανάπτυξης και παραγωγικής λειτουργίας. Παράλληλα δημιουργεί περιορισμούς και δεν ενθυλακώνει την εφαρμογή σε συγκεκριμένη πλατφόρμα ή γλώσσα προγραμματισμού.

Ship: Αυτή η λειτουργία επιτρέπει το σχεδιασμό του κύκλου ανάπτυξης. Καθώς επίσης των δοκιμών και διανομής μέσα από την χρήση μιας συνεπούς διεπαφής χρήστη.

Run: Πρόκειται για την κρίσιμη λειτουργία που παρέχει την δυνατότητα παράταξης των υπηρεσιών. Μάλιστα παρέχει την δυνατότητα κλιμάκωσης τους. Παράλληλα η όλη δραστηριότητα γίνεται με ασφάλεια και αξιοπιστία. (Build, Ship, & Run, 2016)

Εξάλλου θα πρέπει να αναφερθούν τα τρία δομικά στοιχεία του Docker:

Το πρώτο ονομάζεται Docker-Engine που είναι στην ουσία μια πλατφόρμα υποστήριξης container. Το δεύτερο ονομάζεται Docker Hub και αντιστοιχεί σε μια Software as a Service (SaaS) πλατφόρμα για την διαμοίραση και την διαχείριση του κύκλου ζωής των container. Τέλος το Docker Datacenter είναι μια λύση που μπορεί να χρησιμοποιηθεί στις τοπικές υποδομές για την διαμοίραση και την διαχείριση των container που έχει ένας οργανισμός. Μακροσκοπικά οι εφαρμογές του Docker βασίζονται σε αρχιτεκτονική client-server. Ο Docker client επικοινωνεί με το Docker daemon που τρέχει επάνω στο λειτουργικό της φυσικής μηχανής. Στην βάση αυτή δημιουργεί, τρέχει και διανέμει τους container. Οι χρήστες του Docker επικοινωνούν με τον Docker daemon διαμέσου του client(Εικόνα 2). Ο Docker client και ο Docker

daemon επικοινωνούν με την χρήση ενός REST API επάνω σε UNIX sockets ή διεπαφές δικτύου (Docker, 2017)

Από άποψη επαναχρησιμοποίησης το Docker περιέχει:

Εικόνες(Images) : Δηλαδή διακριτά αρχέτυπα λογισμικού που περιλαμβάνουν όλες τις αναγκαίες παραμέτρους και το απαραίτητο σύστημα αρχείων. Είναι το εκάστοτε παραγόμενο της λειτουργίας build. Οι εικόνες δημιουργούνται από άλλες στοιχειώδεις εικόνες με την χρήση ειδικού ρεπερτορίου εντολών.

Ληξιαρχεία Εικόνων(Registries): Πρόκειται για δημόσιες ή ιδιωτικές υποδομές ιστού όπου αποθηκεύονται εικόνες. Οι εικόνες φορτώνονται ή εκφορτώνονται από τα ληξιαρχεία με σχετικά αυτοματοποιημένο τρόπο. Στο σημείο αυτό θα πρέπει να αναφερθεί πως το Docker Hub παρέχει πρόσβαση στο δημόσιο ληξιαρχείο εικόνων. Τα ληξιαρχεία εικόνων είναι ουσιαστικά ένα σύστημα διαμοιρασμού των εικόνων. Στο σημείο αυτό θα μπορούσαν να ειπωθούν αρκετά για τα θέματα ασφάλειας που προκύπτουν. Όμως η ασφάλεια λογισμικού δεν αποτελεί αντικειμενικό στόχο αυτής της μελέτης. Όταν ένας χρήστης δημιουργεί μια εικόνα μπορεί να την προωθήσει σε ένα δημόσιο ληξιαρχείο όπως το Docker Hub ή στο ιδιωτικό ληξιαρχείο που μπορεί να τρέχει πίσω από τις υποδομές ασφαλείας του οργανισμού του. Με την χρήση του Docker client οι χρήστες μπορούν να μεταφορτώνουν τέτοιες δημοσιευμένες εικόνες στην δική τους εγκατάσταση Docker ώστε να δημιουργήσουν τους container που απαιτούν οι επιχειρησιακές τους ανάγκες (Docker, 2017).

Container: Ένας container είναι το τρέχων στιγμιότυπο μιας εικόνας μαζί με όλα τα απαραίτητα στοιχεία (λ.χ. πόρους σε μνήμη και μέρισμα CPU) που απαιτούνται από μια εφαρμογή(λ.χ. βάση δεδομένων ή εξυπηρετητής ιστού), ώστε αυτή να τρέχει όπως προβλέπεται.

Στο σημείο αυτό θα ήταν καλό να αναφερθεί μια λύση η οποία ονομάζεται Docker Toolbox που παρέχει έναν τρόπο να χρησιμοποιηθεί το Docker και σε προσωπικούς υπολογιστές παλαιότερων εκδόσεων Windows. Μάλιστα παρέχονται όλα εκείνα τα εργαλεία που προαναφέρθηκαν ώστε να μπορεί κανείς να αναπτύξει εφαρμογές επάνω στο Docker (Docker T. , 2017). Τέλος IDE όπως το NetBeans, το Eclipse, ή το IntelliJ IDEA παρέχουν υποστήριξη για ανάπτυξη λογισμικού με βάση Docker Container (Gupta, 2016).

2.5. Η έννοια της ενορχήστρωσης στο Docker

Μολονότι η χρήση των container είναι εξαιρετικά χρήσιμη εφ' εαυτής, δεν αρκεί από μόνη της για την δημιουργία ενός συστήματος container μεγάλης κλίμακας για να χρησιμοποιηθεί στα σημερινά παραγωγικά συστήματα υπολογιστικού νέφους. Επιπρόσθετα δεν είναι εύκολη η κλιμάκωση τέτοιων συστημάτων. Είναι επομένως αδήριτη ανάγκη η χρήση επιπρόσθετων βοηθητικών εργαλείων για να διευκολυνθούν οι εργασίες που πρέπει να γίνουν προς επίλυση των προαναφερόμενων ζητημάτων. Επομένως, η παραγωγική χρήση σε μεγάλη κλίμακα είναι καλό να εμπλέκει λογισμικό διαχείρισης συστοιχιών, διαχείρισης σεναρίων ρυθμίσεων, παραμετροποίησης και λύσεις παρακολούθησης. Θα πρέπει να παρέχονται τα ανωτέρω με τρόπο ευέλικτο, κλιμακούμενο ανάλογα με τις επιχειρησιακές ανάγκες. Αποτελεί επομένως πρόκληση η σχεδίαση και η διαχείριση της διαθεσιμότητας και της επεκτασιμότητας τέτοιων συστημάτων. Σε εντελώς αφηρημένο επίπεδο οι λειτουργίες αυτές αποδίδονται με τον όρο ενορχήστρωση. Η ενορχήστρωση ολοκληρώνει την διαχείριση των container για χρήση σε ευρεία κλίμακα. Υπάρχουν διαθέσιμα πολλά εργαλεία ενορχήστρωσης που απλοποιούν τις διαδικασίες αυτές και παρέχουν το πλαίσιο για την παράταξη ικανού αριθμού container. Είναι μάλιστα χαρακτηριστικό ότι διαχειρίζονται τους

πολλαπλούς container ως μια οντότητα ώστε να εξασφαλίζεται η διαθεσιμότητα, η κλιμάκωση και η δικτύωση (Ankerholz, 2016).

2.6. Η έννοια Ομαδοποίησης(Clustering) στο Docker

Μια ομάδα(cluster) συνδυάζει πολλαπλές μηχανές καθώς και τις δυνατότητές τους, δηλαδή τις εφαρμογές container, και τους επιτρέπει να αλληλεπιδρούν μεταξύ τους με απλό τρόπο. Η ομαδοποίηση αυτή επιτρέπει στους διαχειριστές καθώς και στους προγραμματιστές να δημιουργούν ένα αποθετήριο από έτοιμες προς χρήση υπηρεσίες που μπορούν να φιλοξενήσουν εικονικές μηχανές και να επεκταθούν εύκολα σαν να επρόκειτο για μια μηχανή (Martínez, 2017). Τα περισσότερα εργαλεία ενορχήστρωσης παρέχουν και δυνατότητες ομαδοποίησης δηλαδή υπηρεσίες. Ειδικότερα θα λέγαμε πως η ομαδοποίηση είναι ένα χαρακτηριστικό που δημιουργεί συνεργατικά σύνολα συστημάτων. Τα τελευταία παρέχουν τις αναγκαίες εφεδρείες ώστε να ανταποκριθούν στην περίπτωση αστοχίας ενός κόμβου. Επιπρόσθετα η ομαδοποίηση παρέχει στους διαχειριστές και τους προγραμματιστές την δυνατότητα να προσθέτουν ή να αφαιρούν πόρους αναλόγως των τρεχουσών υπολογιστικών αναγκών (Rouse, 2017).

2.7. Άλλες πλατφόρμες εφαρμογών container

2.7.1. Το Rkt

Το rkt (προφέρεται στα αγγλικά και ως rocket) είναι μια πλατφόρμα υποστήριξης container που έχει αναπτυχθεί από την CoreOS (νεοφυής επιχείρηση που υποστηρίζεται από την Google Ventures) και θα μπορούσε να θεωρηθεί ως ο κοντινότερος ανταγωνιστής του (rkt, 2017). Το rkt αναπτύχθηκε ως μια υλοποίηση της προδιαγραφής App Container (appc). Η τελευταία προσδιορίζει ζητήματα ασφαλείας, λειτουργικές παραμέτρους και όλες εκείνες τις υπηρεσίες που απαιτείται να υλοποιηθούν ώστε να μπορούν οι εφαρμογές να τρέχουν ανεξάρτητα

από την υποκείμενη υποδομή. Βασικό κίνητρο για την ανάπτυξη του rkt ήταν η ανάγκη για ευελιξία και για γρήγορους χρόνους στην παραγωγή αλλά με έμφαση στα ζητήματα ασφάλειας. Όπως για παράδειγμα αν επιτρέπεται να εκτελείται ένας container σε λογαριασμό που δεν έχει δικαιώματα υπερ-χρήστη. Βασικό χαρακτηριστικό είναι η σπονδυλωτή αρχιτεκτονική του (rkt, 2017) (Butle, 2016).

2.7.2. Το Kurma

Η Apcera εξέδωσε μια έκδοση ανοικτού λογισμικού με την επωνυμία Kurma. Το Kurma είναι ένας μηχανισμός εκτέλεσης container βασισμένος στην προδιαγραφή appc. Παράλληλα η βασική φιλοσοφία πίσω από το Kurma είναι οποιαδήποτε υπηρεσία συστήματος, ακόμη και ο συγχρονισμός ρολογιού ή διαχείριση αρχείων καταγραφής, θα πρέπει να τρέχουν ως container (Kurma, 2017).

2.7.3. Το Jetpack

Το Jetpack θα μπορούσε επίσης να αναφερθεί ως μια πειραματική εναλλακτική και σαφώς μη πλήρης υλοποίηση της προδιαγραφής appc για το FreeBSD. Χρησιμοποιεί το μηχανισμό απομόνωσης jail καθώς και το Z File System για αποθήκευση συστήματος σε επίπεδα (Jetpack, 2017)

2.7.4. Το OPEN CONTAINER INITIATIVE (OCI)

Το Open Container Initiative (OCI) είναι ένα έργο ανοικτής διακυβέρνησης που ξεκίνησε τον Ιούνιο 2015, κάτω από την εποπτεία του Linux Foundation. Ο Σκοπός του OCI είναι να δημιουργήσει ανοικτά πρότυπα για τους μορφότυπους container και τον τρόπο εκτέλεσής τους. Μέλη του είναι οι πρωτοπόροι εμπλεκόμενοι στην βιομηχανία των container όπως η Docker, η CoreOS, η Amazon Web Services και η Google. Το OCI προς το παρόν προσφέρει δύο προδιαγραφές (OCI, 2017)

- Την προδιαγραφή Τρεξίματος(runtime-spec)
- Και την προδιαγραφή Εικόνας(image-spec)

Με την δημιουργία του OCI στα τέλη του 2016 η appc προδιαγραφή έχει πάψει πλέον να αναπτύσσεται εκτός από μερικές μικρο-αλλαγές που απαιτούνται για να υποστηρίξουν τις υφιστάμενες υλοποιήσεις (appc, 2016).

2.8. Εργαλεία ενορχήστρωσης

Τα εργαλεία ενορχήστρωσης των container παρέχουν ένα πλαίσιο σε επίπεδο οργανισμού για την ολοκληρωμένη διαχείριση των container κυρίως στην διάσταση της κλιμάκωσης και της αποκλιμάκωσης. Αποτελούν σημαντικό εργαλείο τόσο για τους διαχειριστές όσο και για τους προγραμματιστές. Δύο κορυφαία τέτοια εργαλεία-πλατφόρμες ενορχήστρωσης είναι το Docker Swarm και το Google Kubernetes. Ενώ υπάρχουν και άλλες λύσεις ενορχήστρωσης όπως το Amazon ECS, το Heat, το Apache Mesos™ και τοOpenShift® (Wheatley, 2016). Το Google Container Engine είναι ένα σύστημα ενορχήστρωσης για την εκτέλεση Docker container στο Google Cloud Platform. Πρόκειται για ένα εργαλείο προσανατολισμένο στην λειτουργία του Kubernetes συστήματος ενορχήστρωσης (Mouat, 2015).

Κατά την άποψη της Docker, για να αποφασιστεί το κατάλληλο εργαλείο ενορχήστρωσης συνίσταται να λαμβάνονται υπόψη τα κάτωθι σημαντικά χαρακτηριστικά:

- Επίδοση κατά την κλιμάκωση: Δηλαδή το πόσο γρήγορα μπορεί να τεθούν σε λειτουργία οι container κατά την διαδικασία της κλιμάκωσης. Ταυτόχρονα το κατά πόσο ανταποκρίσιμο είναι το σύστημα όταν καταπονείται με φορτίο.
- Απλότητα: Δηλαδή ποιά είναι η ευκολία εκμάθησης και διαχείρισης του συστήματος. Παράλληλα ο αριθμός των διακριτών τμημάτων του συστήματος.

- Ευελιξία: Δηλαδή το κατά πόσο είναι εύκολη η μετάβαση από το υφιστάμενο σύστημα στις γραμμές ροής των επιχειρησιακών διαδικασιών. Σημαντική παράμετρος ευελιξίας είναι η ανώδυνη μετάβαση από το περιβάλλον ανάπτυξης στο παραγωγικό περιβάλλον. Τέλος, είναι σημαντικό η εφαρμογή να μην ενθυλακώνεται σε συγκεκριμένη πλατφόρμα. (Coleman, 2016)

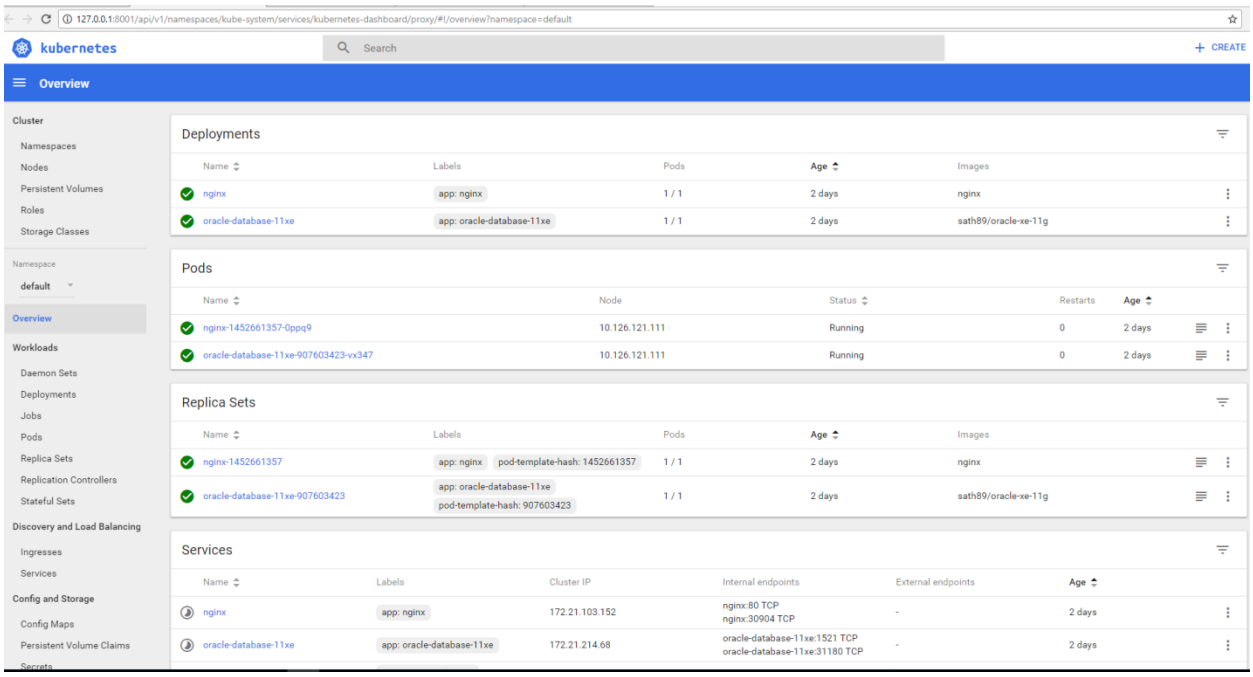
2.8.1. Το Google Kubernetes

Το Kubernetes είναι ένα σύστημα ανοικτού λογισμικού για την διαχείριση εφαρμογών container επάνω σε Linux μηχανήματα και παρέχει στοιχειώδεις μηχανισμούς για την παράταξη, συντήρηση και την κλιμάκωση των εφαρμογών. (kubernetes, 2017). Στην ουσία το Kubernetes διαχειρίζεται ομάδες container. Το πιο σημαντικό είναι πως έχει την δυνατότητα να διαχειριστεί την διασύνδεση και την κλιμάκωση της παράταξης πολλαπλών container. Τα οποία μπορούν να φιλοξενηθούν σε μια μεγάλη ομάδα Linux μηχανημάτων. Με άλλα λόγια επιτρέπει την παροχή υπηρεσιών ενορχήστρωσης τα οποία δύναται να φιλοξενηθούν επάνω σε ένα πλήθος φυσικών μηχανημάτων με ενοποιημένο τρόπο και δίνοντας λύση σε εφαρμογές που απαιτούν μεγάλη κλίμακα υπολογιστικής ισχύος. (Red-Hat, 2015)

Για να γίνει κατανοητό το Kubernetes θα πρέπει να λάβουμε υπόψη τις κάτωθι έννοιες που το συναποτελούν:

- Cluster: Ένα σύνολο από φυσικά ή εικονικά μηχανήματα μαζί με άλλους πόρους υποδομής που χρησιμοποιούνται από το Kubernetes για να τρέχουν τις εφαρμογές που απαιτούνται από τον οργανισμό. Το Kubernetes μπορεί να τρέξει οπουδήποτε.

- Node: Ένας κόμβος δηλαδή είναι στην ουσία μια εικονική ή φυσική μηχανή που τρέχει το Kubernetes. Έχει την δυνατότητα να ενεργοποιήσει επάνω του pod. Ένας κόμβος είναι ο κύριος και οι υπόλοιποι θεωρούνται οι κόμβοι εργάτες.
- Pod: Τα Pod είναι σύνολα από εφαρμογές container με διαμοιραζόμενους αποθηκευτικούς χώρους. Είναι η μικρότερη υπολογιστική μονάδα που μπορεί να δημιουργηθεί, να αξιοποιηθεί και να είναι διαχειρίσιμη από το Kubernetes. Τα pod μπορούν να δημιουργηθούν ατομικά αλλά συνήθως είναι προτιμότερο να χρησιμοποιηθεί ένας ελεγκτής δημιουργίας επαναλήψιμων pod(replication controller). Αυτό συνίσταται ακόμη και για την περίπτωση ενός μοναδικού pod.
- Replication controller: Πρόκειται επομένως για την οντότητα λογισμικού που διαχειρίζεται τον κύκλο ζωής των pod. Διασφαλίζει ότι ο καθορισμένος αριθμός των pod τρέχει σε ένα καθορισμένο χρόνο και για τον σκοπό αυτό δύναται να διακόπτει την λειτουργία ή να εκκινεί την λειτουργία pod κατά την απαίτηση αυτή.
- Service: τα Services παρέχουν ένα μοναδικό σταθερό όνομα και διεύθυνση για ένα σύνολο pod. Εκ των πραγμάτων λειτουργούν ως στοιχειώδεις μονάδες εξισορρόπησης φορτίου.
- Label: Με τα Labels παρέχεται η δυνατότητα να οργανώνονται και να επιλέγονται τα σύνολα των αντικειμένων στην βάση ζευγών κλειδιού-τιμής (kubernetes, 2017).



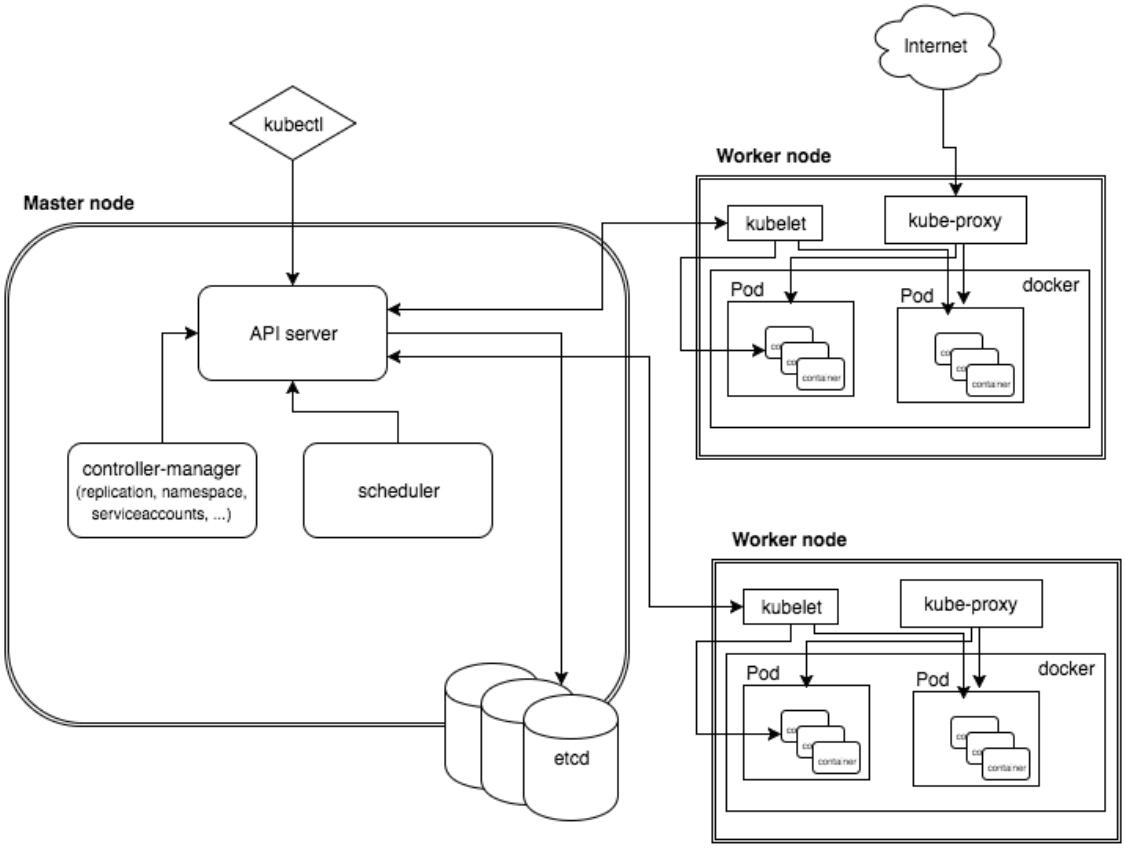
Εικόνα 3 Η κονσόλα διαχείρισης του Kubernetes με ένα Worker node

Συμπερασματικά, θα λέγαμε πως το Kubernetes είναι ένα εργαλείο ενορχήστρωσης container που έχει την δυνατότητα να χρησιμοποιηθεί είτε στις τοπικές εφαρμογές είτε επάνω σε δημόσιες υποδομές υπολογιστικού νέφους.

2.8.2. Το Google Container Engine

Το Google Container Engine (GKE) είναι ένας εργαλείο ενορχήστρωσης και διαχείρισης συστοιχιών που επιτρέπει την χρήση Docker container επάνω στην Google Cloud Platform. Οι προγραμματιστές ορίζουν τις απαιτήσεις όπως CPU, μνήμη και αριθμό αντιγράφων. Αντίστοιχα το GKE εκκινεί τους container μέσα στην συστοιχία και τους διαχειρίζεται αυτόματα. Το πιο σημαντικό που αξίζει να αναφερθεί είναι πως το GKE έχει δημιουργηθεί με βάση το εργαλείο ενορχήστρωσης ανοικτού λογισμικού Kubernetes που αναφέρθηκε νωρίτερα. Κατά συνέπεια οι οργανισμοί τυπικά χρησιμοποιούν το Google Container Engine για να δημιουργήσουν ή να επεκτείνουν συστοιχίες Docker container. Επίσης να δημιουργήσουν pod, replication controller, υπηρεσίες ή εξισορροπητές φόρτου.

Επιπρόσθετα επιτρέπει την ενημέρωση και την αναβάθμιση συστοιχιών container καθώς και την εκσφαλμάτωση τους (Rouse, 2017). Αναλυτικότερα θα λέγαμε πως το GKE περιέχει σύνολα από Google Compute Engine στιγμιότυπα να οποία τρέχουν Kubernetes. Συνήθως ένας κύριος κόμβος διαχειρίζεται μια συστοιχία Docker container. Το GKE επίσης τρέχει ένα Kubernetes API εξυπηρετητή για να συνεργάζεται με τη συστοιχία και να εκτελεί καθήκοντα όπως το να εξυπηρετεί API αιτήματα και να δρομολογεί container. Εκτός από τον κύριο κόμβο μια συστοιχία μπορεί να έχει έναν ή περισσότερους κόμβους οι οποίοι τρέχουν ένα εκτελέσιμο Docker και έναν kubelet agent που είναι αναγκαία για την διαχείριση των Docker container. Με άλλα λόγια πρόκειται για τους κόμβους εργάτες (kubernetes b.-m. , 2017) .

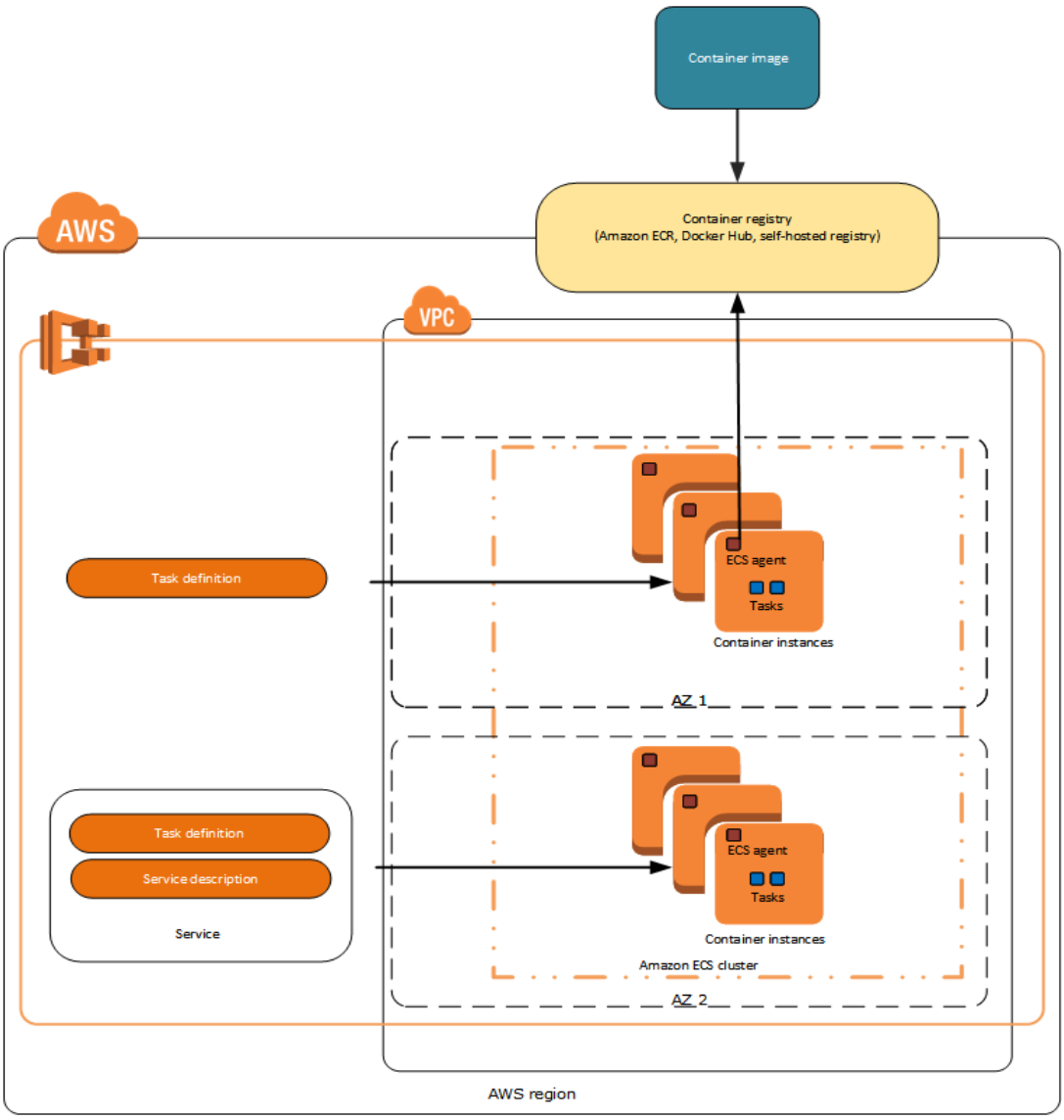


Εικόνα 4 Εποπτική Αρχιτεκτονική του Kubernetes

(Πηγή: https://x-team.com/blog/introduction-kubernetes-architecture/)

2.8.3. Το Amazon EC2 Container Service (ECS)

Το Container Service (ECS) της Amazon Web Services έχει ως βασική λειτουργία την εγκατάσταση και διαχείριση της υποδομή για τους container του Docker. Ειδικότερα, το Amazon EC2 Container Service επιτρέπει στους διαχειριστές να τρέχουν και να διαχειρίζονται τους container του Docker επάνω σε διαχειρίσιμες συστοιχίες στιγμιότυπων του Amazon Elastic Compute Cloud (EC2). Κάθε τέτοιο στιγμιότυπο της συστοιχίας, που είναι επάνω στο Amazon ECS, τρέχει μια διεργασία Docker daemon. Αυτό έχει το πλεονέκτημα της εύκολης μεταφοράς μια εφαρμογής container στον Amazon ECS, χωρίς ιδιαίτερες περαιτέρω παραμετροποιήσεις. Ο διαχειριστής μπορεί να θέσει σε λειτουργία μια συστοιχία από στιγμιότυπα container. Επίσης μπορεί να προσδιορίζει τις εργασίες τις οποίες θα πρέπει να επιτελούν. Όλα αυτά είναι εύκολα διαχειρίσιμα από το Amazon ECS. Το Amazon ECS επιτρέπει στον διαχειριστή να ορίσει εργασίες διαμέσου ενός προτύπου που λέγεται ορισμός εργασίας(Task definition). Το αρχείο ενός ορισμού εργασίας δεν έχει περιορισμούς στον αριθμό των εργασιών οι οποίες θα μπορούσαν να εκκινήσουν. Εξάλλου τα αρχεία ορισμού επιτρέπουν τον έλεγχο των εκδόσεων των προδιαγραφών των εφαρμογών. Επιπρόσθετα οι διαχειριστές μπορούν να διαχειρίζονται τις συστοιχίες και τους container Docker και να παίρνουν αναλυτικές πληροφορίες σχετικά με την κατάσταση της συστοιχίας και των στιγμιότυπων του μέσω του εκάστοτε ECS agent. Το πιο αξιοσημείωτο μάλιστα είναι ότι το Amazon ECS είναι ολοκληρωμένο με το Elastic Load Balancing (ELB) της Amazon γεγονός που επιτρέπει στον διαχειριστή να κατανέμει το φόρτο κίνησης κατά μήκος των παρατεταγμένων container (ECS, 2017).



Εικόνα 5. Εποπτική Αρχιτεκτονική του Amazon EC2 Container Service (ECS)

(Πηγή: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html)

2.8.4. Το Nomad: Εργαλείο Ενορχήστρωσης της HashiCorp.

Το Nomad είναι ένας διαχειριστής συστοιχιών που έχει σχεδιαστεί τόσο για μακρόβιες υπηρεσίες όσο για βραχύβια φορτία μαζικά επεξεργαζόμενα. Οι προγραμματιστές δηλώνουν τις προδιαγραφές που απαιτείται να καλύπτουν οι εργασίες και το Nomad φροντίζει ώστε αυτές να εκτελεστούν κατά τον βέλτιστο

δυνατό τρόπο. Βασικά χαρακτηριστικά του είναι η υποστήριξη Docker container με αυτόματη κλιμάκωση ανάλογα με τις απαιτήσεις και η αυτόματη ανάκαμψη από αποτυχία. Είναι ικανό να διαχειρίζεται πόρους σε πολλαπλά κέντρα δεδομένων και σε πολλαπλές περιοχές. Παρόλα αυτά έχει καταβληθεί προσπάθεια να είναι επιχειρησιακά απλό, κατανεμημένο και με υψηλή διαθεσιμότητα. Όμως το κύριο πλεονέκτημα του είναι η ολοκλήρωση του με τα εργαλεία του οικοσυστήματος της HashiCorp (λ.χ. Consul) (Nomad, 2017).

2.8.5. Apache Mesos

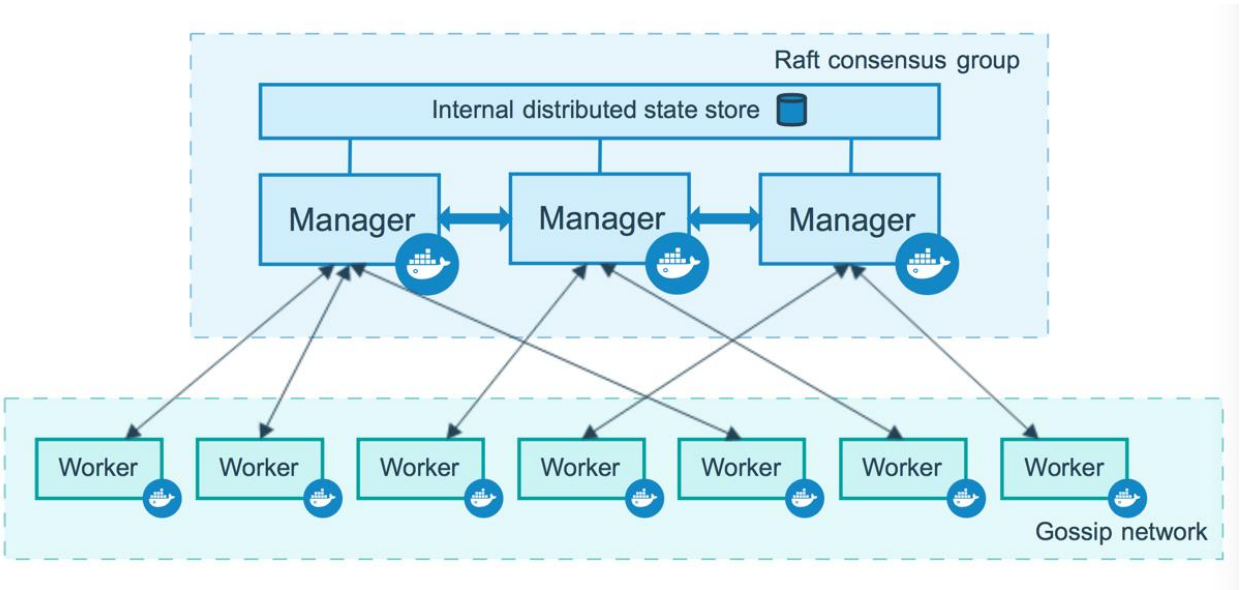
Το Mesos είναι μια πλατφόρμα για αποτελεσματική και δυναμική διαμοίραση πόρων ανάμεσα σε διαφορετικά πλαίσια. Στοχεύει δηλαδή στο να παρέχει μια κλιμακούμενη λύση για διαμοίραση πόρων σε κατανεμημένα συστήματα μεγάλης κλίμακας. Είναι ανοικτό ως προς τα πλαίσια που μπορεί να υποστηρίξει. Φυσικά μια επιλογή μπορεί να είναι οι Docker container με την χρήση του πλαισίου Marathon. Κάτω από το πρίσμα αυτό θα μπορούσαμε να πούμε πως αποτελεί ένα εργαλείο ενορχήστρωσης. Η προσέγγιση δρομολόγησης των πόρων στο Mesos γίνεται σε δύο επίπεδα. Αρχικά το εφαρμοζόμενο πλαίσιο (π.χ. Docker) ζητά να δεσμεύει τους πόρους που χρειάζεται. Ο δρομολογητής στην συνέχεια εξετάζει τους πόρους που του έχουν παραχωρηθεί και εφόσον πληρούν τις προδιαγραφές του, τους αποδέχεται. Διαφορετικά τους απορρίπτει. Ακολουθούν προσφορές μέχρι να γίνει αποδεκτό ένα επίπεδο πόρων. Αυτή η διαδικασία οδηγεί στην δυνατότητα διαχείρισης μεγάλου όγκου πόρων(μέχρι 50000 κόμβων) με αξιοσημείωτη αποτελεσματικότητα. Αν και μπορεί να χρησιμοποιηθεί σαν τυπικό εργαλείο ενορχήστρωσης Docker container η αρχική του στόχευση ήταν τα περιβάλλοντα επεξεργασίας δεδομένων μεγάλης κλίμακας(π.χ. Spark) (Hindman, et al., 2011)

2.8.6. Docker Swarm: Ενσωματωμένη Ενορχήστρωση στο Docker

Θα πρέπει να παρατηρήσουμε πως το ενδιαφέρον σημείο του Docker Swarm είναι ότι μετατρέπει ένα αριθμό διαθέσιμων Docker εξυπηρετητών σε ένα μοναδικό εικονικό Docker εξυπηρετητή. Με άλλα λόγια θα λέγαμε ότι το Docker Swarm είναι ένα εργαλείο ενορχήστρωσης που συνδέει πολλαπλές μηχανές Docker σε μια συστοιχία η οποία συμπεριφέρεται φαινομενικά ως μια ενιαία μηχανή Docker. Οι διαχειριστές και οι προγραμματιστές μπορούν να δημιουργήσουν έτσι μια υποδομή από μηχανές Docker που μπορούν να υποστηρίζουν εφαρμογές container Docker. Έχουν επίσης την δυνατότητα να δημιουργούν επιπρόσθετα αντίγραφα τους (κλιμάκωση) σαν αν επρόκειτο για μια μόνο μηχανή. (Docker-Swarm, 2017). Πρόκειται δηλαδή για ένα εργαλείο, που επικοινωνεί με μια διεργασία Docker και μπορεί να χρησιμοποιηθεί για να πραγματοποιηθεί μια κλιμάκωση. Το Docker Swarm έχει υψηλή επεκτασιμότητα και επίδοση καθώς έχει την δυνατότητα να κλιμακώνεται μέχρι 100 κόμβους και 50,000 container χωρίς να επηρεάζεται η απόδοση του συστήματος από την διαδικασία επέκτασης (Stoyanov, 2016).

Ξεκινώντας από την έκδοση 1.12 κεντρική μηχανή του Docker έχει ενσωματωμένες τις δυνατότητες ενορχήστρωσης χωρίς καμιά εξάρτηση σε εξωτερικές υποδομές. Οι διαχειριστές μπορούν να χρησιμοποιούν το Docker API με αντικείμενα όπως τις υπηρεσίες και τους κόμβους επάνω στο Swarm. Κάθε μηχανή εντός του Swarm είναι ασφαλής εξορισμού. Με την χρήση του TLS παρέχεται αξιόπιστη ταυτοποίηση, ελεγχόμενη πρόσβαση και κρυπτογράφηση ανάμεσα στις διασυνδέσεις κάθε κόμβου στο Swarm. Αυτό ο τρόπος λειτουργίας του Swarm επιτρέπει στους διαχειριστές IT υποδομών να ενορχηστρώνουν και να δρομολογούν τις εφαρμογές container. Στη γενικευμένη μορφή του το Docker Swarm αποτελείται από δύο είδη κόμβων(μηχανές Docker) τους διαχειριστές(manager) και τους εργάτες(worker). Οι κόμβοι διαχειριστές δέχονται

εντολές, δρομολογούν εργασίες και ελέγχουν την ορθή λειτουργία των κόμβων εργατών. Οι κόμβοι διαχειριστές έχουν ως βασική αποστολή τους τον συνεχή έλεγχο και την προσαρμογή του περιβάλλοντος ώστε να εξασφαλιστεί μηδενική πιθανότητα μοναδιαίου σημείου αποτυχίας(single-point-of-failure). Επίσης μεριμνούν ώστε να μην υπάρχει διακοπή της λειτουργίας της εφαρμογής. Για τον συντονισμό τους χρησιμοποιούν το RAFT αλγόριθμο ελέγχου της πλειοψηφίας. Αντίστοιχα οι κόμβοι εργάτες έχουν ως κύρια αποστολή την εκτέλεση των container που τους ανατίθενται από τους κόμβους διαχείρισης(manager). (Docker C. E., 2016). Ενώ θα πρέπει να σημειωθεί πως οι κόμβοι διαχείρισης έχουν δυνατότητα να εκτελούν εργασίες όπως και οι κόμβοι εργάτες. Αυτό μπορεί να γίνει κατανοητό αν δημιουργηθεί μια Docker Swarm συστοιχία με ένα μόνο κόμβο. Τότε εκ των πραγμάτων όλες οι εργασίες ανατίθενται στον κόμβο διαχειριστή. Είναι χαρακτηριστικό πως το Docker Swarm θυμίζει ιδιαίτερα την client server αρχιτεκτονική του Apache Hadoop. Το γεγονός αυτό έχει οδηγήσει στο να υπάρξουν ενδιαφέρουσες προτάσεις για την συνδυασμένη χρήση του Docker Swarm ως συστοιχίας για την επεξεργασία δεδομένων μεγάλης κλίμακας με λογισμικό όπως το Hadoop (Naik, 2017) .



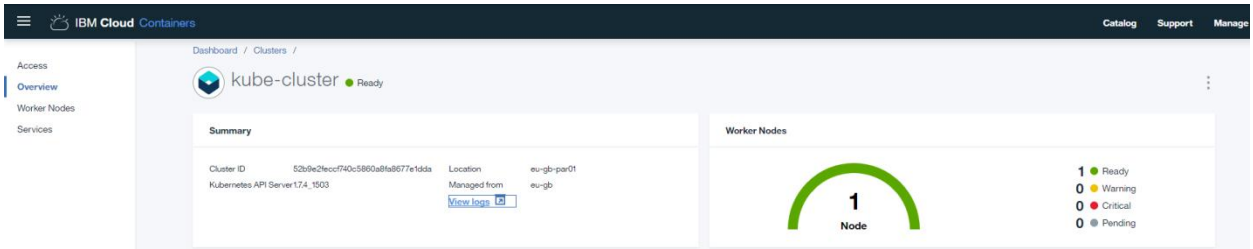
Εικόνα 6. Εποπτική Αρχιτεκτονική Docker Swarm

(Πηγή: https://docs.Docker.com/engine/swarm/how-swarm-mode-works/nodes/)

2.9. Κριτική αξιολόγηση στα μέχρι τώρα διαλαμβανόμενα

Από την ενδελεχή μελέτη όλων των ανωτέρω είναι φανερό πως η βασική και η περισσότερο δημοφιλής δομή λογισμικού εκτέλεσης μιας εφαρμογής σε container είναι το Docker. Είναι επίσης εμφανές πως στη σχετική βιομηχανία υπάρχει μια τάση ευρείας αυτοματοποίησης. Η τάση αυτή διευκολύνεται από την ύπαρξη των εργαλείων ενορχήστρωσης που προαναφέρθηκαν. Χωρίς να υποτιμάται η δυναμική των άλλων τεχνολογιών που προαναφέρθηκαν, δημιουργείται η αίσθηση πως οι κολοσσοί στον χώρο της υπολογιστικής νέφους που τις υποστηρίζουν, εκ των πραγμάτων, οδηγούν στην ενθυλάκωση της αρχιτεκτονικής μιας εφαρμογής βασισμένη σε container στις δικές τους τεχνολογικές επιλογές(λ.χ. Google Container Engine ή Amazon EC2 Container Service). Επομένως για την εξέταση μιας μελέτης

περίπτωσης όπως για παράδειγμα η αντιμετώπιση της εξισορρόπησης του φόρτου μιας εφαρμογής ιστού θα πρέπει να γίνει με την υιοθέτηση μιας τεχνολογίας όσο το δυνατόν παντοδαπής. Μια τέτοια είναι φαίνεται να είναι το Docker. Επιπρόσθετα, πρακτικά επικρατεί η τάση για χρήση συστοιχιών container, οι οποίες μάλιστα, θα πρέπει να είναι εύκολα διαχειρίσιμες. Έτσι έγινε συνειδητή επιλογή στο πλαίσιο της εργασίας αυτής η εκλογή του Docker Swarm ως εργαλείου ενορχήστρωσης για δύο κυρίως λόγους. Ο πρώτος αφορά στον όσο το δυνατόν μη εγκλωβισμό της προτεινόμενης αρχιτεκτονικής σε ad-hoc τεχνολογικά χαρακτηριστικά. Ο δεύτερος αφορά στην καλύτερη εκμετάλλευση των δυνατοτήτων των οντοτήτων λογισμικού του Docker. Κυρίως γιατί το Docker Swarm είναι προϊόν της ίδιας κοινότητας προγραμματιστών που έχει εισαγάγει το Docker. Φυσικά, με τις παραδοχές και τις προϋποθέσεις που αναλύονται σε επόμενο κεφάλαιο. Η επιλογή αυτή έγινε στο πλαίσιο της εργασίας αυτής και μόνο. Δεν αποτελεί σύσταση ούτε κανόνα για την ανάπτυξη παρόμοιων συστημάτων. Και αυτό γιατί ο τομέας παρουσιάζει συνεχή εξέλιξη. Παράλληλα κορυφαίες εταιρίες στον χώρο έχουν προχωρήσει σε εκτεταμένες επενδύσεις που ενδεχομένως να οδηγήσουν την αγορά και τους προγραμματιστές προς άλλες λύσεις. Ενδεικτική είναι η πολιτική του IBM cloud Lite account που επιτρέπει την δωρεάν δημιουργία μιας συστοιχίας ενός κόμβου εργάτη με το εργαλείο Kubernetes για 2GB μνήμη εκτέλεσης, χωρίς καμιά άλλη δέσμευση πλέον των άλλων δωρεάν παροχών (Bittner, 2017).



Εικόνα 7 Το ταμπλό παρακολούθησης του Kubernetes για ένα IBM cloud Lite account

Σε καμιά περίπτωση όμως δεν απομειώνεται η προτεινόμενη προσέγγιση καθώς η επιλεγόμενη σύνθεση δηλαδή Docker container σε τεχνολογία συστοιχίας Swarm μπορεί να υποστεί μετάπτωση σε άλλη τεχνολογία συστοιχίας όπως επί παραδείγματι στο Kubernetes (Kompose, 2017).

3. Ανασκόπηση σε σχετικές εργασίες

Οι σχετικές εργασίες κινούνται σε διαφορετικούς άξονες καθώς έχουν διαφορετικές στοχεύσεις. Αλλά εκείνο που πρέπει κανείς να παρατηρήσει είναι πως οι αναφορές είναι σχετικά πρόσφατες γεγονός που δείχνει πως το ενδιαφέρον της ακαδημαϊκής κοινότητας είναι νεοφυές. Επίσης δείχνει πως η αναζήτηση στρέφεται σε διάφορες κατευθύνσεις. Προσπαθεί μάλιστα να επανεξετάσει γνωστά τεχνικά ζητήματα μέσα στα νέα δεδομένα που επιβάλλει ο χώρος της τεχνολογίας του υπολογιστικού νέφους. Θα πρέπει να διευκρινιστεί πως η προσέγγιση παράθεσης γίνεται από το ειδικό θέμα σε πιο ευρύτερα ζητήματα ώστε σταδιακά να γίνεται κατανοητή η σειρά των επιλογών στην προσέγγιση του θέματος της εργασίας.

3.1. Μελέτη της κλιμάκωσης και εξισορρόπηση φόρτου με άλλες προσεγγίσεις

Θα πρέπει να παρατηρήσει κανείς πως επικεντρώνονται κυρίως στην κλιμάκωση και εξισορρόπηση φόρτου αλλά η τεχνολογική πλατφόρμα διαφέρει .

3.1.1. Χρήση φυσικών Μηχανημάτων

Η επίτευξη κλιμάκωσης και υψηλής διαθεσιμότητας αποτελεί βασικό στόχο των υπηρεσιών που προσφέρονται από συστοιχίες εξυπηρετητών ιστού. Για να επιτύχει ο στόχος αυτός σε φυσικά μηχανήματα απαιτείται ένας επιπλέον εξυπηρετητής με αποκλειστική λειτουργία την εξισορροπημένη κατανομή του φόρτου της κίνησης στους ανωτέρω εξυπηρετητές. Για την επιτυχή λειτουργία θα πρέπει να βρεθεί ένας αλγόριθμος που να αποδίδει την κίνηση στον διαθέσιμο εξυπηρετητή κατά βέλτιστο τρόπο. Για να προκύψει κάτι τέτοιο θα πρέπει να γίνεται υπολογισμός του φόρτου συνεχώς. Μια πιθανή τακτική θα ήταν να χρησιμοποιηθεί μια σειρά από παραμέτρους συνδυαστικά. Ειδικότερα έχει προταθεί να χρησιμοποιηθεί, ως μια

σημαντική παράμετρο φόρτου ο αριθμός των ανοικτών περιγραφέων αρχείων μαζί με συνήθεις παραμέτρους φόρτου όπως κύκλους CPU και ελεύθερης μνήμης. Στην περίπτωση μιας συστοιχίας φυσικών μηχανημάτων, για να επιτευχθεί μια τέτοια λειτουργία, όμως απαιτείται και ένας μηχανισμός μετάβασης αναφορών και μηνυμάτων. Αυτό μπόρεσε να υλοποιηθεί με την χρήση ενός SNMP agent που κοινοποιεί την κατάσταση του εκάστοτε εξυπηρετητή συμπεριλαμβανομένου του αριθμού των ανοικτών περιγραφέων αρχείων. Ενδεικτικές μετρικές απόδοσης ήταν η παροχέτευση(απλή και κανονικοποιημένη), ο χρόνος απόκρισης HTTP και τα σφάλματα. Μια ενδιαφέρουσα και συνάμα λογική προσέγγιση στις δοκιμές είναι να χωριστούν αυτές σε δύο κατηγορίες. Η πρώτη κατηγορία αφορούσε τις δοκιμές που γίνονταν σε συνθήκες ισορροπίας. Η δεύτερη κατηγορία αφορούσε σε δοκιμές όπου γινόταν σε συνθήκες ανισορροπίας δηλαδή με υπερβολικό φορτίο σε ένα ποσοστό από τους εξυπηρετητές ιστού. Κάτω από αυτές τις συνθήκες εξετάστηκε ο μηχανισμός εξισορρόπησης, που ονομάστηκε Scalable Load Balancing (SLBL) και περιγράφηκε με τις προηγούμενες μετρικές σε σχέση με τους γνωστούς αλγορίθμους εξισορρόπησης του και ειδικότερα του Round Robin (RR) που δεν διατηρεί την κατάσταση του και του Least Connections (LC) που διατηρεί γνώση της κατάστασης του. Τα αποτελέσματα έδειξαν πως η επιλογή των συγκεκριμένων μετρικών ως παραμέτρων που μεταβάλλουν την κατανομή του φόρτου έχουν καλύτερη επίδοση από τους προαναφερόμενους συνήθεις αλγορίθμους. Ενδεικτικά αναφέρεται ότι ο μέσος ρυθμός εξυπηρέτησης αιτημάτων από τον αλγόριθμο SLBL είναι 1,27 φορές περισσότερος από τον Least Connections και 1,93 φορές περισσότερος από τον Round Robin (Maluk & M.A., 2015).

3.1.2. Χρήση περιβάλλοντος προσομοίωσης Software Defined Networking

Επειδή στην πράξη η μελέτη της εξισορρόπησης του φόρτου της κίνησης απαιτεί μηχανήματα που έχουν αυξημένο κόστος σε ακαδημαϊκό επίπεδο έχουν προκύψει

προσεγγίσεις που είναι ιδιαίτερα χρήσιμες για την μελέτη ενός δυναμικού ζητήματος όπως είναι η εξισορρόπηση του φόρτου. Χαρακτηριστικό παράδειγμα είναι το mininet που μπορεί να δημιουργήσει ένα ρεαλιστικό εικονικό δίκτυο εντός μιας εικονικής μηχανής. Σε συνδυασμό με την πλατφόρμα POX. (Dordal, 2017) Έτσι δόθηκε έμφαση στην ανάπτυξη ενός εξισορροπητή φορτίου της κίνησης ο οποίος έχει γραφτεί για να τρέψει επάνω στην πλατφόρμα POX εντός του mininet. Πρόκειται δηλαδή για μια αρχιτεκτονική αμιγώς βασισμένη στο λογισμικό δικτύωσης (Software Defined Networking ή SDN). Το χαρακτηριστικό αυτής της αρχιτεκτονικής είναι πως διαχωρίζει τις εφαρμογές που τρέχουν από επίπεδο ελέγχου και το επίπεδο δεδομένων. Στο επίπεδο ελέγχου βρίσκεται ο SDN ελεγκτής που στην ουσία υλοποιεί την λογική του εξισορροπητή φορτίου. Ένα σημαντικό θέμα που τίθεται σ' αυτού το είδους την προσέγγιση είναι η χρήση ενός συστηματικού εργαλείου για την δημιουργία του απαραίτητου φόρτου κίνησης που καλείται να εξυπηρετηθεί. Θα πρέπει να έχει την δυνατότητα δημιουργίας μετρίσιμου φόρτου, χρονικά και ποσοτικά ελεγχόμενου. Στην συγκεκριμένη προσέγγιση χρησιμοποιήθηκε το εργαλείο δημιουργίας φόρτου Openload. Η SDN προσέγγιση δρομολόγησης και πάλι συγκρίθηκε με καθιερωμένους αλγορίθμους δρομολόγησης και ειδικότερα με τον Round Robin. Ενώ βασικές μετρικές ήταν οι συναλλαγές ανά δευτερόλεπτο, ο συνολικός αριθμός των αιτημάτων και ο μέσος χρόνος απόκρισης. Τα αποτελέσματα έδειξαν μια υπεροχή της SDN προσέγγισης δρομολόγησης καθώς αυξάνονταν ο αριθμός των χρηστών στο πλαίσιο του πειράματος. Δηλαδή και τις τρείς μετρικές που προαναφέρθηκαν (Kaur & Jyoti, 2017).

3.2. Χρήση Docker container για κατανομή φόρτου (load scheduling)

Εντούτοις έχει προκύψει και μια σχετικότερη προσέγγιση που βασίζεται στο Docker. Η προσέγγιση αυτή προσανατολίζεται σε ένα μοντέλο, το οποίο είναι

ευαίσθητο στην κατανάλωση ενέργειας. Είναι γεγονός ότι η βέλτιστη κατανάλωση ενέργειας σε ένα κέντρο δεδομένων αποτελεί σημαντικό ζητούμενο. Επομένως είναι ιδιαίτερα χρήσιμο να επικεντρωθεί κανείς στους μετρίσιμους παράγοντες που αποτυπώνουν την κατανάλωση ενέργειας. Στην ουσία αποτελεί μια σημαντική στρατηγική στην υλοποίηση ενός συστήματος κατανομής φόρτου και κλιμάκωσης. Η εργασία αυτή είναι ακόμη πιο συναφής με το αντικείμενο της παρούσης γιατί η τεχνολογική βάση είναι το Docker. Ιδιαίτερη σημασία έχει το να εξετάσουμε πως υλοποιείται η διαδικασία της κλιμάκωσης και αποκλιμάκωσης. Έτσι όταν ο φόρτος αυξάνει με την χρήση μιας API κλήσης ένας νέος container δημιουργείται με την χρήση RMI μεθόδων. Ο τελευταίος αναλαμβάνει να εκτελέσει το επιπρόσθετο φόρτο. Οι container δημιουργούνται αυτόματα με βάσει την παρακολούθηση των κατωφλιών CPU και RAM. Έτσι διασφαλίζεται ότι οι υπόλοιποι container δεν υπερφορτώνονται με επιπλέον εργασίες. Αντίστοιχα όταν οι απαιτήσεις φόρτου ελαττώνονται τότε ο επιπλέον container απαλοίφεται ώστε να απελευθερωθούν οι πόροι που δέσμευε και κατά συνέπεια να μειωθεί η κατανάλωση ενέργειας. Κεντρική ιδέα της προσπάθειας αυτής ήταν καταδεχτεί πως η χρήση ενός ενεργειακά προσανατολισμένου αλγορίθμου κλιμάκωσης και αποκλιμάκωσης εξασφαλίζει την μέγιστη φιλοξενία container μέσα σε ένα επιχειρησιακό χώρο. Επομένως οδηγεί στην βέλτιστη χρήση των πόρων ενός κέντρου δεδομένων. Εκείνο που παρουσιάζει ιδιαίτερο ενδιαφέρον είναι πως στην διαδικασία εμπλέκεται μια οντότητα λογισμικού που ονομάζεται Facebook feed και είναι ένα API για την επικοινωνία του συστήματος με τους container (Facebook, 2017) Είναι σαφές λοιπόν πως απαιτείται μια οντότητα διαμεσολάβησης για την εκτίμηση του όγκου του φορτίου και την εκτέλεση σχετικών εντολών κλιμάκωσης και αποκλιμάκωσης (Sureshkumar & Rajesh, 2017).

3.3. Προσέγγιση προσανατολισμένη στην αυτόματη κλιμάκωση και το Docker

Η αυτόματη κλιμάκωση παίζει σημαντικό ρόλο στα περιβάλλοντα υπολογιστικής υψηλής απόδοσης (HPC). Ειδικότερα ένα σοβαρό ζήτημα είναι η επίλυση των θεμάτων εξάρτησης λογισμικού στα περιβάλλοντα αυτά για τους διαχειριστές και τους ερευνητές. Το ζήτημα αυτό επιχειρείται να λυθεί με την εφαρμογή της τεχνολογίας container του Docker. Ειδικότερα παρουσιάζεται μια λύση για την αυτοματοποιημένη κλιμάκωση που βασίζεται στην ανακάλυψη υπηρεσίας βασισμένη στο Consul. Το οποίο είναι ένα εργαλείο ανακάλυψης υπηρεσίας δηλαδή της διεργασίας της αυτόματης παροχής πρόσβασης σε πελάτες μιας υπηρεσίας από ένα κατάλληλο στιγμιότυπο της υπηρεσίας αυτής. Παράλληλα είναι μια υψηλής διαθεσιμότητας αποθήκη ζευγών κλειδιών-τιμών, DNS εξυπηρετητή και παρέχει δυνατότητες ελέγχου της υγείας ενός συστήματος (Mouat, 2015). Το πιο ενδιαφέρον είναι πως για να εξασφαλίσει υψηλή διαθεσιμότητα υλοποιεί συστοιχία ad-hoc εικονικού τύπου επάνω σε μια συστοιχία φυσικών μηχανημάτων. Η πρόταση δείχνει να αποτελεί αξιόπιστη λύση για την λειτουργία του συστήματος σε περιβάλλον HPC. Εντούτοις οι συγγραφείς καταλήγουν θεωρώντας ότι μια λύση συστοιχίας τύπου Docker Swarm ή Kubernetes μαζί με ένα φιλικό προς τον χρήστη GUI θα βελτίωνε περισσότερο την απόδοση τους προτεινόμενου συστήματος σε σχέση με την περιγραφείσα λύση (Yu & Huang, 2015). Πέρα από το τελευταίο θα πρέπει να παρατηρήσει κανείς πως και στην περίπτωση αυτή αναδεικνύεται η ανάγκη μια οντότητα διαμεσολάβησης που θα επιτρέπει να μεταβάλλει τους διαθέσιμους πόρους ανάλογα με τις ανάγκες ώστε να διασφαλιστεί υψηλή διαθεσιμότητα και απόδοση.

3.4. Προσέγγιση προσανατολισμένη στο Docker Swarm

Ωστόσο υπάρχουν και προσεγγίσεις που στοχεύουν στην ανάδειξη της σημασίας των συστοιχιών στην υλοποίηση αποτελεσματικών λύσεων στο επίπεδο υπολογιστικού νέφους. Η επιλογή του Platform as a Service (PaaS) στο νέφος παρέχει τους απαραίτητους πόρους υποδομής για να υποστηριχθούν οι φιλοξενούμενες εφαρμογές. Τέτοια περιβάλλοντα επιβάλλεται να μπορούν να ανταποκριθούν σε αύξηση των φόρτου των κλήσεων των εφαρμογών. Το τελευταίο μπορεί να επιτευχθεί είτε με επέκταση των στιγμιότυπων της εφαρμογής είτε με αύξηση των διαθέσιμων πόρων. Εξαιτίας του γεγονότος ότι όλα τα στιγμιότυπα των εφαρμογών τρέχουν σε απομονωμένους container η γνώση που αποκτάται από την λειτουργία του πρώτου στιγμιότυπου εφαρμογής δεν μπορεί να διαμοιραστεί σε άλλα στιγμιότυπα της εφαρμογής που δημιουργούμε εκ των υστέρων λόγω κλιμάκωσης. Επομένως ένα βασικό πρόβλημα είναι αφενός ένας αυξημένος χρόνος εκκίνησης της εφαρμογής και αφετέρου σφάλματα λήξης χρόνου στην απόκριση για τα κλιμακούμενα στιγμιότυπα. Παράλληλα παρατηρήθηκε παρεμπόδιση της απόδοσης λόγω υποστήριξης στις ίδιες υποδομές πολλαπλών εφαρμογών. Για να αντιμετωπιστούν αυτά τα ζητήματα απαιτείται συντονισμός μέσω της μεταφοράς των απαραίτητων πληροφοριών. Όμως η μεταφορά των απαραίτητων πληροφοριών ανάμεσα στις εφαρμογές είναι μια διεργασία εξαιρετικά δαπανηρή σε χρόνο και υπολογιστικούς πόρους. Μια μελέτη περίπτωσης είναι η δυναμική μεταγλώττιση τεμαχίων λογισμικού. Στην περίπτωση αυτή είναι δυνατόν να υλοποιηθεί μια επεκτάσιμη και ασφαλής τεχνική όπου τα μεταγλωττισμένα τεμάχια λογισμικού που παράγονται από το πρώτο στιγμιότυπο εκτέλεσης μπορούν να διαμοιραστούν με τα επόμενα στιγμιότυπα κατά τέτοιο τρόπο ώστε να αντιμετωπίζονται τα ζητήματα που τέθηκαν. Η τεχνική αυτή ονομάστηκε Dynamically Compiled Artifact Sharing(DCAS). Πέρα από τις ιδιαίτερες λεπτομέρειες

υλοποίησης εκείνο που έχει ιδιαίτερη σημασία είναι ότι η λύση βασίζεται στο Docker Swarm το οποίο τρέχει επάνω σε έξι(6) κόμβους δηλαδή εικονικές μηχανές. Η υπηρεσία DCAS επάνω στο Docker Swarm υλοποιεί όλους τους απαραίτητους μηχανισμούς διαμεσολάβησης που διαφαίνεται και πάλι ως απαραίτητη ξεχωριστή οντότητα για την επιτέλεση μηχανισμών κλιμάκωσης και αποκλιμάκωσης εφαρμογών. Ένα άλλο αξιόλογο σημείο της προσέγγισης αυτής είναι η συστηματική δημιουργία δοκιμών καταπόνησης με την χρήση του Apache JMeter, εργαλείο ανοικτού λογισμικού, από μια μηχανή εκτός της συστοιχίας του Docker Swarm. Με την εφαρμογή αυτής της πειραματικής διάταξης οι συγγραφείς έδειξαν πως επιτυγχάνεται σημαντική βελτίωση αφενός στην μείωση του χρόνου εκκίνησης της εφαρμογής και αφετέρου στην μείωση των σφαλμάτων λήξης χρόνου της απόκριση των κλιμακούμενων στιγμιότυπων (Patros, Dilli, Kent, & Dawson, 2017).

Παράλληλα έχει υπάρξει και η προσέγγιση της υλοποίησης μιας συστοιχίας MPI προκειμένου να υλοποιηθεί ένα περιβάλλον υπολογιστικής υψηλής απόδοσης με βάση το Docker Swarm. Τυπικά η δημιουργία μιας τέτοιας συστοιχίας είναι εξαιρετικά δύσκολη και χρονοβόρα. Με άλλα λόγια αποτελεί από μόνη της μια πρόκληση. Η πρόκληση αυτή αντιμετωπίζεται με την εφαρμογή της τεχνολογίας του Docker container ώστε να υποστηριχτεί σε ανώτερο επίπεδο μια βιβλιοθήκη MPI. Η τελευταία παρέχει ένα απλοποιημένο δίκτυο API που αναπαριστά αφηρημένα το υποκείμενο API του λειτουργικού συστήματος. Η επιλογή για τον συντονισμό εδώ ήταν το Docker Swarm ως εργαλείο ενορχήστρωσης. Μια τέτοια επιλογή παρέχει το πλεονέκτημα της αυτοματοποιημένης εγκατάστασης της συστοιχίας MPI. Στην πραγματικότητα η όλη προσέγγιση υλοποιήθηκε ώστε να διευκολύνει την περαιτέρω έρευνα στον τομέα αυτό. Έτσι λειτουργεί ως προπομπός για την επίλυση προβλημάτων που μπορεί να λύσει μια συστοιχία MPI. Για τον

λόγο αυτό όλες οι εικόνες του Docker, ο πηγαίος κώδικας και η σχετική τεκμηρίωση έχουν δημόσια διατεθεί στο github (Bein & Nguyen, 2017).

Επίσης έχει υπάρξει και προσέγγιση που επικεντρώνεται στα πλεονεκτήματα του Docker Swarm. Ένα αναδυόμενο πρόβλημα στην σύγχρονη υπολογιστική νέφους είναι η δημιουργία υποδομής η οποία θα είναι συμβατή με διάφορους παρόχους υπολογιστικών πόρων στο νέφος. Αυτή η υβριδική υποδομή νέφους στην ουσία επιτρέπει την μίξη διαφορετικών πλατφορμών και παρόχων υπολογιστικών πόρων νέφους ανάλογα με τις δραστηριότητες ανάπτυξης εφαρμογών. Το πιο σημαντικό πλεονέκτημα που διαβλέπει κανείς από την προσέγγιση αυτή είναι η μη ενθυλάκωση της αναπτυσσόμενης εφαρμογής στις ειδικές απαιτήσεις και τα εργαλεία συγκεκριμένου παρόχου. Όμως δεν είναι μόνο αυτό. Πάντα ελλοχεύει ο κίνδυνος να υπάρξει εκτεταμένη απώλεια δεδομένων κυρίως ως αποτέλεσμα καταστροφής και όχι ως κακόβουλης ενέργειας. Παράλληλα πάντα είναι υπαρκτός και ο κίνδυνος ύπαρξης χρόνου πτώσης μιας εφαρμογής για τον ίδιο λόγο. Είναι σαφές πως η υλοποίηση μιας τέτοιας πλατφόρμας είναι ένα εξαιρετικά πολύπλοκο έργο λόγω των διαφορετικών τεχνολογιών, διεπαφών και υπηρεσιών. Μέρος της λύσης ήταν η δημοφιλής εισαγωγή του Docker δηλαδή για την ακρίβεια της μεθοδολογίας ανάπτυξης λογισμικού βάσει του Docker container. Όμως η πρόταση αυτή ήταν ατελής προκειμένου να δημιουργηθεί υβριδική υποδομή νέφους καθώς δεν έλυνε όλα τα προαναφερόμενα ζητήματα διαλειτουργικότητας. Η εμφάνιση όμως του Docker Swarm ως μιας νέας προσέγγισης στην βιομηχανία του υπολογιστικού νέφους αποτελεί δελεαστική πρόταση. Κυρίως γιατί έχει μεγάλη δυναμική στο να παρέχει ένα περιβάλλον ανάπτυξης σε πολλαπλές πλατφόρμες νέφους. Κατά τα άλλα η προσέγγιση αυτή θα πρέπει να θεωρηθεί ως μια προσομοίωση της δημιουργίας ενός εικονικού συστήματος από συστήματα (SoS) για τη δημιουργία λογισμικού σε πολλαπλούς παρόχους υπηρεσιών νέφους. Η προσομοίωση αυτή βασίστηκε στο Docker Swarm, VirtualBox, Mac OS X, nginx και το

redis. Όμως αυτό το περιβάλλον θα μπορούσε κανείς να το μεταφέρει εύκολα σε δημοφιλείς παρόχους υπηρεσιών νέφους όπως Amazon Web Services, Microsoft Azure, Digital Ocean, Google Compute Engine, Exoscale, Generic, OpenStack, Rackspace, IBM Softlayer, VMware vCloud Air (Naik N. , 2016). Καταδεικνύεται δηλαδή με σαφήνεια η ευκολία μετάπτωσης εφαρμογών που είναι σχεδιασμένες με την φιλοσοφία των container του Docker επάνω σε συστοιχία Docker Swarm.

3.5. Συγκριτική προσέγγιση των container και των εικονικών μηχανών

Τα τελευταία χρόνια η χρήση εικονικών μηχανών ως ένα μοντέλο αφαίρεσης από το φυσικό υλισμικό έχει γίνει μια δημοφιλής πρακτική για την απομόνωση πόρων και την συγχώνευση εξυπηρετητών. Με την γενίκευση της εφαρμογής της τεχνολογίας των εικονικών μηχανών η εγγύηση της υψηλής διαθεσιμότητας των φιλοξενούμενων εφαρμογών προβάλλει ως ένα σοβαρό ζήτημα που έχει τύχει μάλιστα ιδιαίτερης προσοχής από την ερευνητική κοινότητα. Όπως είναι φυσικό μια λογική ερευνητική προσέγγιση θα ήταν η σύγκριση των υφιστάμενων τεχνολογιών εικονικών μηχανών από άποψη υψηλής διαθεσιμότητας. Έτσι οι δύο επικρατέστερες λύσεις που είναι συγκρίσιμες συνοψίζονται στις πλατφόρμες βασιζόμενες σε hypervisor και στις πλατφόρμες βασιζόμενες σε container. Έγινε προσπάθεια στο πλαίσιο αυτό να συγκριθούν σε περιορισμούς και χαρακτηριστικά όπως την μετάπτωση εν λειτουργία, την ανίχνευση σφαλμάτων και κατοχύρωση σημείων ελέγχου-υποκατάστασης. Διαπιστώνεται ότι παρά τις προσπάθειες για ανάπτυξη συστοιχιών σε περιβάλλοντα βασισμένα σε container, οι λύσεις αυτές υστερούν των λύσεων βασιζόμενες σε hypervisor στον τομέα της υψηλής διαθεσιμότητας. Δεν υπάρχουν αξιόπιστες επιλογές για την παρακολούθηση σφαλμάτων και την διαχείριση τους προκειμένου ένα σύστημα να θεωρείται ως υλοποίηση υψηλής διαθεσιμότητας. Το ενδιαφέρον εδώ είναι ότι προτείνεται να

υπάρξουν επεκτάσεις στις τεχνολογίες container ώστε να εξυπηρετηθεί ο σκοπός αυτός (Li & Kanso, 2015).

Μέχρι στιγμής έχει καταδειχθεί η σημαντικότητα των container σε ένα περιβάλλον PaaS. Είναι συχνή η χρήση των container ώστε να γίνεται ευχερέστερη κατανομή των πόρων. Σε αντιδιαστολή, παρατηρείται μια μείωση της απόδοσης λόγω υπερφόρτωσης των εικονικών μηχανών των λύσεων με hypervisor. Από την πλευρά τους οι container έχουν τα δικά τους μειονεκτήματα. Κυρίως μειώνεται η απόδοση τους όταν καταπονούνται με υψηλό ρυθμό πακέτων. Κάτι τέτοιο δεν έχουν οι εικονικές μηχανές με hypervisor. Επομένως στο πλαίσιο της σχετικής ερευνητικής εργασίας δημιουργηθήκαν περιβάλλοντα δοκιμών και πραγματοποιήθηκαν δοκιμές ταυτόχρονα σε εικονικές μηχανές και σε container. Οι δοκιμές αυτές έγιναν για συγκεκριμένα σημεία αναφοράς π.χ. καταπόνηση εισόδου εξόδου, ταχύτητα RAM κ.α. Διαπιστώθηκε ένα ελαφρύ προβάδισμα της τεχνολογίας των container από εκείνης των εικονικών μηχανών. Ως αντιστάθμισμα αναφέρεται η περιορισμένη ασφάλεια απομόνωσης σε σχέση με τις εικονικές μηχανές (Barik, Lenka, Rao, & Ghose, 2016).

Όπως ήδη έχει διαφανεί οι εικονικές μηχανές χρησιμοποιούνται για να απεγκλωβίσουν τις εφαρμογές από τα ιδιαίτερα τεχνικά χαρακτηριστικά των φυσικών μηχανημάτων σε κέντρα δεδομένων και περιβάλλοντα υπολογιστικού νέφους, στα οποία παρατάσσονται για εκτέλεση. Οι εικονικές μηχανές και τα εικονικά λειτουργικά συστήματα έχουν και άλλες διαστάσεις που αξίζει να διερευνηθούν συγκριτικά. Ειδικότερα προκύπτει πως η παρεμβολή λόγω συγκατοίκησης μπορεί να οδηγήσει σε υποβάθμιση της επίδοσης και των δύο κατηγοριών τεχνολογίας. Μάλιστα, για ορισμένους τύπους φορτίου η υποβάθμιση της απόδοσης στην περίπτωση των container είναι πιο αξιοσημείωτη. Παρόλα αυτά τονίζεται ότι οι container δεν έχουν αυστηρά δεσμευμένα όρια πόρων όπως οι εικονικές μηχανές. Αυτό επιτρέπει να σε περιπτώσεις υπερδέσμευσης πόρων να

μπορούν να διατίθενται επιπρόσθετοι πόροι από εκείνους που υποαπασχολούν τους δικούς τους πόρους. Ενώ για το υπαρκτό θέμα της έλλειψης απομόνωσης προτείνεται η εκτέλεση των container εντός εικονικών μηχανών (Sharma, Chaufournier, Shenoy, & Tay, 2016).

4. Υπηρεσίες Container στο Docker και Εξισορρόπηση Φόρτου σε Εφαρμογές Ιστού.

Οι χρήση εφαρμογών εντός Docker container προβάλλει πλέον ως μια ιδιαιτέρα ελκυστική λύση για την δημιουργία περιβαλλόντων εικονικών μηχανών. Μπορούν να χρησιμοποιηθούν για να υποστηρίξουν ποικίλες λύσεις. Για παράδειγμα να υλοποιήσουν εξυπηρετητές ιστού, να τρέξουν ad-hoc κώδικα εφαρμογών, να υποστηρίξουν στιγμιότυπα βάσεων δεδομένων, να αναπτύξουν βασικά συστήματα ώστε να αποδειχθεί μια τεχνολογική λύση ή ιδέα (Proof Of Concept) κ.α. Όπως έχει αναφερθεί νωρίτερα καταναλώνουν λιγότερους πόρους από τις τυπικές εικονικές μηχανές(virtual machines). Για την ακρίβεια εξοικονομούνται πόροι καθώς ουσιαστικά τρέχουν πάνω στο βασικό λειτουργικό σύστημα και απαλλασσόμαστε από την επιφόρτιση της συντήρησης του hypervisor. Αυτό όμως που κάνει το Docker ακόμη πιο δημοφιλές είναι το Docker Hub Ληξιαρχείο όπου διατίθενται αρκετά προ-ρυθμισμένες εικόνες έτοιμες για χρήση (alpine, nginx, etc). Επομένως ο χρόνος παραμετροποίησης ελαττώνεται σημαντικά. Έτσι οι προγραμματιστές επικεντρώνονται στον αντικειμενικό στόχο της υλοποίησης της ολοκληρωμένης εφαρμογής χωρίς να αναλώνεται πολύτιμος χρόνος στις λεπτομέρειες της παρακολούθησης τεκμηρίωσης παραμετροποιημένου λογισμικού τρίτων.

Κατά συνέπεια, θα μπορούσε να αποτελέσει μια εξαιρετική πλατφόρμα όπου θα μπορούσαν να επιδειχθούν σημαντικές λειτουργίες ενός ολοκληρωμένου πληροφοριακού συστήματος προσανατολισμένου στις εφαρμογές ιστού. Ειδικότερα θα μπορούσε να επιδειχθεί η λειτουργία εξισορρόπησης φόρτου μιας απλής εφαρμογής Ιστού.

Φυσικά, δεν πρέπει να παραβλέπουμε πως η εξισορρόπηση του φόρτου είναι ένα σημαντικό πραγματικό πρόβλημα όταν πρέπει να εκτεθεί δημόσια στο διαδίκτυο

μια εικονική διεύθυνση ιστού και να δρομολογηθεί η κίνηση από αυτή σε μια συστοιχία εξυπηρετητών ιστού. Όπως έχει αναλυθεί πιο πάνω για τον σκοπό αυτό μπορούν να χρησιμοποιηθούν διάφοροι αλγόριθμοι δρομολόγησης π.χ. ο round robin ή ο least connections. Ενώ συχνά προτείνεται μια μεθοδολογία ώστε να προκύψει ένας πιο βελτιωμένος τρόπος εξισορρόπησης του φόρτου που δέχεται το εξεταζόμενο σύστημα (Maluk & M.A., 2015).

Όμως για να μελετηθεί το πρόβλημα της εξισορρόπησης του φόρτου μιας εφαρμογής ιστού θα πρέπει να υπάρχει η δυνατότητα να ενεργοποιηθούν αρκετοί εξυπηρετητές ιστού, καθώς και όλες εκείνες οι οντότητες λογισμικού που επιτρέπουν την ομαλή λειτουργία ενός τυπικού πληροφοριακού συστήματος ιστού.(π.χ. λογισμικό παρακολούθησης των μετρικών του φόρτου).

Ενώ για την επιβεβαίωση της λειτουργίας της υλοποίησης θα πρέπει να χρησιμοποιηθεί ένας μηχανισμός που θα μπορούσε να δημιουργήσει φορτίο. Για τον σκοπό αυτό θα μπορούσαν να χρησιμοποιηθούν εφαρμογές πελάτη(web client). Μάλιστα αυτές θα μπορούσαν να παραταχθούν μέσα στον Docker που είναι ιδανικό περιβάλλον για τον σκοπό αυτό (Colton Smith J. D., 2017). Συνήθως αυτό όμως επιβαρύνει την διαθέσιμη υποδομή. Για τον λόγο αυτό είναι δυνατόν να χρησιμοποιηθεί μια λύση ειδικά αναπτυγμένη για την εκτέλεση δοκιμών υπερφόρτωσης ενός πληροφοριακού συστήματος. Μια τέτοια επιλογή θα μπορούσε να ήταν το JMeter (Patros, Dilli, Kent, & Dawson, 2017) (Jin-Gang, Ya-Rong, Bo, & Shu, 2017).

Σκοπός της εργασίας είναι αφενός να μελετηθεί ο μηχανισμός εξισορρόπησης φόρτου, αφετέρου να σχηματοποιηθεί ένας μηχανισμός κλιμάκωσης και αποκλιμάκωσης των εξυπηρετητών ιστού ώστε να ανταποκρίνεται στις εκάστοτε ανάγκες. Με τελικό στόχο την εκμετάλλευση των πόρων κατά βέλτιστο τρόπο.

4.1. Θεμελιώδεις προδιαγραφές

Στο πλαίσιο της εργασίας χρησιμοποιηθήκαν οι τύπου Unix Docker container ως υπηρεσίες(services) για να υλοποιηθούν ο εξυπηρετητής καταμερισμού φόρτου οι πολλαπλοί εξυπηρετητές ιστού και οι βοηθητικές οντότητες για την συλλογή μετρικών και την υλοποίηση του μηχανισμού κλιμάκωσης-αποκλιμάκωσης καθώς και εφαρμογές πελάτη(web client). Εφόσον επιλέχθηκε το Docker Swarm ως εργαλείο ενορχήστρωσης, επιβάλλεται να γίνει η ανάπτυξη των container με την μορφή των υπηρεσιών. (Docker-Guides, 2017). Ενώ αξίζει να αναφέρουμε πως και στο Kubernetes καθώς και σε άλλα εργαλεία ενορχήστρωσης χρησιμοποιείται μια τέτοια προσέγγιση για την χρήση container σε συστήματα συστοιχίας.

Το στοιχειώδες web service που εξετάστηκε ήταν μια στοιχειώδεις web εφαρμογή που ήταν βασισμένη σε μια αρχική εργασία που δημιουργήθηκε για την μελέτη της εξισορρόπησης φόρτου στο ευρύτερο περιβάλλον Docker (Colton Smith J. D., 2017). Επίσης μια βασική αρχιτεκτονική επιλογή που επιτρέπει το Docker ήταν το να τεθούν οι πολλαπλοί εξυπηρετητές ιστού ώστε να μοιράζονται το κοινό περιεχόμενο HTML και να παρέχουν έτσι την ίδια απόκριση στις εξωτερικές κλήσεις. Αναφορικά με τους εξυπηρετητές υποστήριξης του φόρτου των κλήσεων επιλέχθηκαν οι nginx εξυπηρετητές. Οι οποίοι διαθέτουν βιβλιοθήκη μετρικών που επιτρέπουν την ανάλυση τους (Tolchanov, 2017).

Αρχικά για την επιβεβαίωση της λειτουργίας του συστήματος χρησιμοποιηθήκαν εφαρμογές πελάτη γραμμένες ως bash scripts (Colton Smith J. D., 2017). Τελικά όμως προτιμήθηκε η χρήση του JMeter καθώς επιτρέπει την ελεγχόμενη αποστολή φόρτου σε συγκριμένο χρόνο. Το γεγονός αυτό επιτρέπει την συστηματική παρατήρηση κατά την εκτέλεση δοκιμών καταπόνησης.

Στο πλαίσιο της εργασίας αυτής εξετάστηκαν οι διαθέσιμες μέθοδοι εξισορρόπησης φόρτου για την διαχείριση εισερχόμενων αιτημάτων και ειδικότερα οι Round-Robin και Least-Connections. Είναι χαρακτηριστικό πως το περιβάλλον ομοιομορφίας που επιβάλλει το Docker Swarm καθιστά την μελέτη των εκδόσεων με ποσοστώσεις των ανωτέρω αλγορίθμων άνευ ουσιαστικού αντικειμένου

Τέλος βασικές παράμετροι για την ανάλυση της απόδοσης αποτέλεσαν οι συναλλαγές(αιτήματα) ανά δευτερόλεπτο ο αριθμός των bytes που μεταφέρθηκαν στο δευτερόλεπτο ανά nginx εξυπηρετητή. Ενώ, βασική επιδίωξη ήταν να εξεταστεί αν οι διαφορετικοί αλγόριθμοι εξισορρόπησης φόρτου επηρεάζουν την απόδοση του εξεταζόμενου συστήματος.

4.2. Περιβάλλον του συστήματος

Σημαντικό δομικό στοιχείο του συστήματος ήταν το Alpine Linux, εξαιτίας του γεγονότος ότι διαθέτει ένα ελαφρύ αποτύπωμα (Alpine, 2017). Οι εξυπηρετητές ιστού που χρησιμοποιηθήκαν ήταν οι nginx, τοποθετήθηκαν στο Docker Swarm και αποτέλεσαν τις μονάδες υπό δοκιμή. Οι δοκιμές άλλων τύπων μονάδων ξεφεύγει από το σκοπό αυτής της εργασίας. Εντούτοις θα μπορούσε πολύ εύκολα να τεθεί ένα τέτοιο περιβάλλον δοκιμών εφόσον η στόχευση θα ήταν για παράδειγμα η απόκριση μιας βάσης δεδομένων στο φόρτο. Ενώ, όπως έγινε αντιληπτό από την σχετική βιβλιογραφία, η επιλογή του Docker ως πλατφόρμας παράταξης της εφαρμογής επιτρέπει την εύκολη μετάβαση σε διαφορετικές αρχιτεκτονικές φυσικών μηχανημάτων.

Η υλοποίηση ενός απλού web service ή αλλιώς μιας απλής web εφαρμογής η οποία θα τρέχει σε ένα container και ειδικότερα στο Docker δεν ήταν αυτοσκοπός της εργασίας. Αντίθετα ήταν ένα πρώτο βήμα ώστε να διερευνηθεί η δυνατότητα εξισορρόπησης φόρτου(Load Balancing) καθώς και η δυνατότητα κλιμάκωσης σε

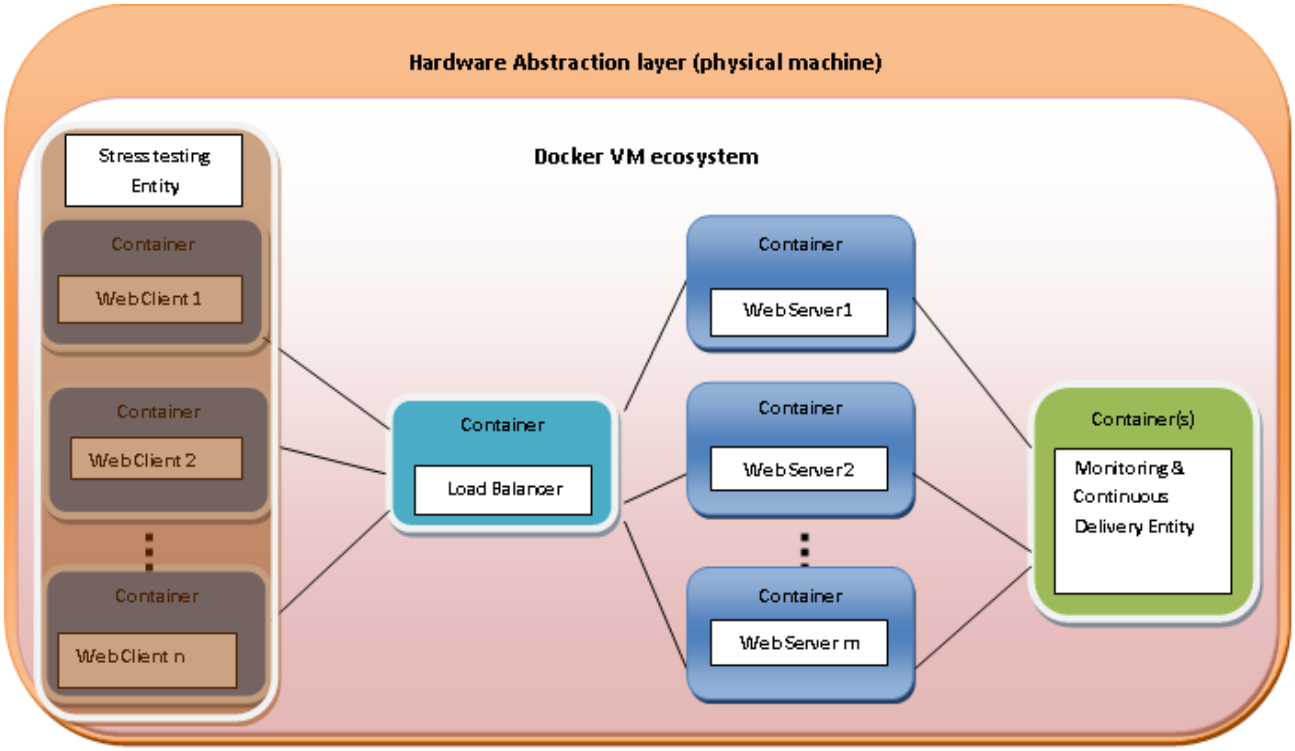
ένα τέτοιο σύστημα. Όπως θα δούμε στην συνέχεια βασική λειτουργικότητα είναι η δυνατότητα της μεταβολής του πλήθους των container με βάση την τρέχουσα ζήτηση. Τέλος διερευνήθηκαν συγκεκριμένα τεχνικά σενάρια όπου με βάση το φόρτο προς την εφαρμογή ιστού δύναται να μεταβάλλεται αντίστοιχα το πλήθος των container. Έτσι μελετήθηκε η σχέση τρέχουσας ζήτησης-φόρτου με την κλιμάκωση των container στο ευρύτερο οικοσύστημα του Docker.

Μια σημαντική σημείωση πρακτικού ενδιαφέροντος είναι το κάτωθι:

Η επιλογή της χρήσης ελαφροβαρών εικονικών μηχανών όπως αυτών του Docker επιτρέπει εξάλλου και την φορητότητα αυτής της εργασίας καθώς μπορεί να ξεκινήσει σε ένα windows laptop και εφόσον προκύψει ανάγκη για επιπλέον πόρους, μπορεί να μεταφερθεί αμεσότατα σε κατάλληλο Linux Server ακόμη και σε Cloud. (Mouat, 2015)

4.3. Αφηρημένο Αρχιτεκτονικό Μοντέλο του Συστήματος

Από την σταχυολόγηση των δομικών χαρακτηριστικών των σχετικών εργασιών που εξετάστηκαν από την βιβλιογραφία διακρίνονται οι εξής διαφορετικές οντότητες όπως φαίνεται στο παρακάτω σχήμα.



Εικόνα 8. Το μοντέλο του εξεταζόμενου συστήματος

Προκειμένου να περιγραφεί το μοντέλο σε μορφή όσο το δυνατόν πιο γενικευμένη αφαιρέθηκαν χαρακτηριστικά όπως το εργαλείο ενορχήστρωσης και συνακόλουθα η λογική των υπηρεσιών(services). Ουσιαστικά επικεντρωνόμαστε στους αναγκαίους container με σκοπό να υλοποιηθεί ένα σύστημα που δύναται να ικανοποιεί τις απαιτήσεις της εργασίας αυτής.

Έτσι ειδικότερα διακρίνονται οι εξής διαφορετικοί container μέσα στο οικοσύστημα Docker (Hassen, 2015):

- **Stress Testing Entity**: Για να επετεύχθη η ύπαρξη φορτίου αρχικά χρησιμοποιήθηκε ένας τύπος container οποίος θα πρέπει να μεταβάλλεται σε πλήθος ώστε να προσομοιώνει την μεταβολή του φορτίου (Colton Smith R. C., 2017). Η επιλογή αυτή αποτέλεσε ένα σημαντικό βήμα για την επιβεβαίωση της ορθής λειτουργίας του εξεταζόμενου συστήματος. Για την εκτέλεση όμως των σεναρίων με συστηματικότητα χρησιμοποιήθηκε ένα εργαλείο το

Apache JMeter από το φυσικό μηχάνημα εκτός του Docker Swarm κόμβου (Patros, Dilli, Kent, & Dawson, 2017). Το εργαλείο αυτό θα μπορούσε να θεωρηθεί ως μια οντότητα δημιουργίας υπερφόρτωσης συστήματος που ισοδυναμεί με πολλούς πελάτες ιστού που δημιουργούν κίνηση ταυτόχρονα.

- **Load Balancer:** Ένας ειδικός container που θα λειτουργεί ως ένας reverse proxy ώστε να προωθεί και διαμοιράζει τα αιτήματα των πελατών ιστού στους εξυπηρετητές ιστού, που αναφέρονται πιο κάτω, ανάλογα με την επιλεγόμενη μεθοδολογία εξισορρόπησης (Allspaw & Robbins, 2010).
- **Web Server**: Πρόκειται για τον εκάστοτε ελαφροβαρή εξυπηρετητή υποστήριξης(backend) ιστού που λαμβάνει και απαντά στα αιτήματα των πελατών ιστού. Στην ουσία θα είναι ένας container κάθε φορά. Στην πραγματικότητα δέχεται τα αιτήματα από τον load balancer. Επειδή δέχεται αιτήματα και σερβίρει περιεχόμενο είναι κρίσιμο στο πλαίσιο μιας μελέτης να αποθηκεύει στατιστικά για ex-post ανάλυση.
- **Monitoring & Continuous Delivery Entity**: Πρόκειται για ένα σύνολο οντοτήτων που ως στόχο έχουν την συνεχή επιτήρηση του συστήματος και την λήψη κατάλληλων μέτρων για την αντιμετώπιση των προβλεπόμενων τιμών των μετρικών. Θα μπορούσε να θεωρηθεί ως ένας διαμεσολαβητής για την ορθή λειτουργία(με βάση συγκεκριμένες παραμέτρους) του εξεταζόμενου συστήματος.

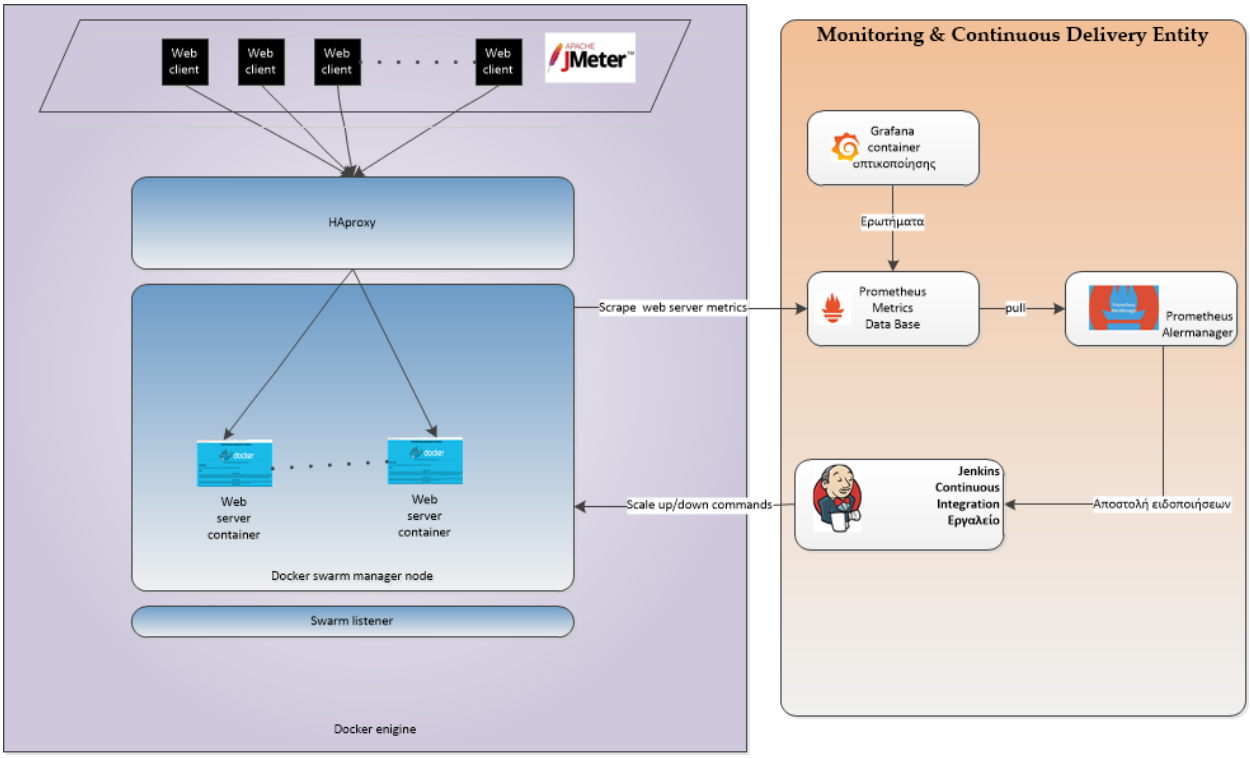
4.4. Τεχνολογική Προσέγγιση της προτεινόμενης λύσης

Προκειμένου να εξασφαλιστεί η δυνατότητα κλιμάκωσης-αποκλιμάκωσης της εξισορρόπησης φόρτου απαιτείται η χρήση ενός λογισμικού που επιτυγχάνει την ομοιόμορφη διαχείριση των container (Docker-UCP, 2016). Κατά συνέπεια απαιτείται ένας δρομολογητής container που να είναι συνάμα και ισχυρό εργαλείο ενορχήστρωσης. Μεταξύ των άλλων που αναφέρθηκαν θα μπορούσαμε να εστιάσουμε την προσοχή μας σε τρία δημοφιλή τέτοια: το Docker swarm, το Kubernetes και το Apache Mesos(με χρήση του πλαισίου Marathon). Όμως από αυτές τις τρείς επιλογές προτιμήθηκε εκείνη του Docker swarm κυρίως γιατί, όπως είναι φυσικό, μπορεί να συνεργάζεται καλύτερα με το υπόλοιπο Docker οικοσύστημα (Grillet, 2016). Το Docker Swarm θα μπορούσε να υλοποιηθεί ως ένας Docker κόμβος που δρα ταυτόχρονα ως κόμβος κεντρικής διαχείρισης μιας συστοιχίας containers (cluster) αλλά και ως ένας agent που τρέχει σε κάθε Docker host (Kane, 2015). Έτσι διευκολύνει ιδιαίτερα την επεκτασιμότητα του συστήματος. Μέσα, λοιπόν, στο περιβάλλον της συστοιχίας των container και προκειμένου να εξασφαλιστεί η λειτουργικότητα της εξισορρόπησης του φόρτου θα χρησιμοποιηθεί η τεχνική λύση του HAproxy (Gupta K. , 2017). Για λόγους ομοιομορφίας των υπηρεσιών οι εξυπηρετητές υποστήριξης θα μπορούσαν ως υπηρεσίες containers να βασιστούν στην τεχνολογία nginx (Docker-UCP, 2016). Για την συλλογή στατιστικών επιπρόσθετα θα μπορούσε να χρησιμοποιηθεί το Prometheus (Mouat, 2015). Τέλος, το περιβάλλον όπως περιγράφεται στην συνέχεια υλοποιήθηκε με την βοήθεια του Docker Toolbox σε PC Laptop.

5. Περιγραφή της Μεθοδολογίας

5.1. Προτεινόμενη αρχιτεκτονική συστήματος εξισορρόπησης συστοιχίας εξυπηρετητών ιστού

Στην παρακάτω εικόνα φαίνεται η αρχιτεκτονική δομή της λύσης η οποία επιλέχτηκε με σκοπό να επιτρέψει την παρακολούθηση της εξισορρόπησης του φόρτου και να επιλύσει το πρόβλημα του scale up και scale down. Αρχικά, θα πρέπει να σημειωθεί πως η κάθε οντότητα του σχήματος στο πλαίσιο του Docker swarm είναι μια υπηρεσία(service) ή στην ουσία ένας Docker container.



Εικόνα 9. Η Αρχιτεκτονική υλοποίηση του μοντέλου

Στην πράξη επιχειρήθηκε να επεκταθεί μια υφιστάμενη εργασία πάνω στην εξισορρόπηση φορτίου που αναπτύχθηκε με σκοπό την μελέτη της εξισορρόπησης του φόρτου nginx εξυπηρετητών ιστού σε τυπικό περιβάλλον Docker (Colton Smith J.

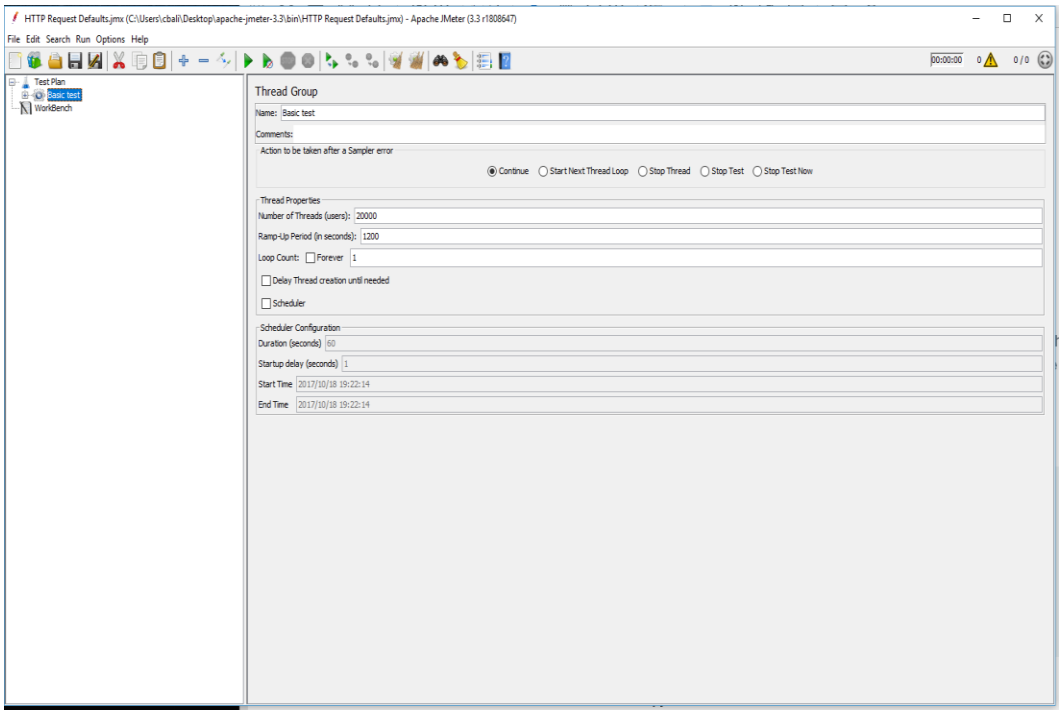
D., 2017) . Στην ίδια εργασία προτάθηκε η εφαρμογή του Docker Swarm ως μια πιθανή βελτίωση. Κατά συνέπεια επεκτάθηκε η αρχική προσέγγιση ως προς την κατεύθυνση αυτή. Ενώ θα πρέπει να επισημανθεί πως χρησιμοποιήθηκαν οι συστάσεις παρόμοιων προσεγγίσεων προκειμένου να δημιουργηθεί ένα αποτελεσματικό σύστημα για τους σκοπούς της εργασίας (Farcic, 2017).

Βασικές επισημάνσεις και παραδοχές είναι οι κάτωθι:

- Στο πλαίσιο της εργασίας αυτής το Docker swarm έχει ένα node ώστε να μην υπεισέρχεται ο παράγοντας του balancing μεταξύ των κόμβων Docker swarm. Αν και αυτό δεν συνίσταται, γιατί αποτελεί ένα βασικό σημείο αποτυχίας σε παραγωγικά συστήματα, εντούτοις οι περιορισμένοι πόροι και το γεγονός ότι το κεντρικό ζήτημα της μελέτης της εξισορρόπησης του φόρτου εξυπηρετείται, οδήγησαν σε αυτό τον συμβιβασμό. (Docker-Swarm, 2017). Άλλωστε δοθέντων των σχετικών πόρων η μετάβαση σε μια πλήρη συστοιχία τριών κόμβων είναι εξαιρετικά εύκολη.
- Οι εξυπηρετητές ιστού που χρησιμοποιήθηκαν είναι τύπου nginx. Ειδικότερα χρησιμοποιήθηκε μια εικόνα Docker με ενσωματωμένο το Lua Prometheus plugin που διαθέτει τις κατάλληλες μετρικές στο port 9145.
- Η επιλογή των υπολοίπων οντοτήτων λογισμικού και ειδικότερα του Prometheus(βάση δεδομένων μετρικών) έγινε γιατί άλλες αντίστοιχες λύσεις δεν αποτελούσαν στην πράξη μια εξολοκλήρου λύση ανοικτού λογισμικού (λ.χ. Zabbix, Sensu). Όμως το βασικότερο είναι πως το Prometheus έχει γραφτεί κατά το μεγαλύτερο μέρος του σε γλώσσα Go όπως και το Docker. Επομένως αναμένεται να έχουν καλύτερη συμβατότητα.(Prometheus-Faq, 2017).
- Για την παροχή εντολών χρησιμοποιήθηκε το Jenkins που είναι ένα εργαλείο Continuous Integration. Εκτελεί προγραμματισμένες ή triggered εργασίες. Στο

πλαίσιο της εργασίας χρησιμοποιείται ως ένας broker για να εκτελεστεί αυτοματοποιημένα κλιμάκωση και αποκλιμάκωση των εξυπηρετητών ιστού.

- Για τις δοκιμές των σεναρίων που ακολουθούν χρησιμοποιείται το εργαλείο JMeter. Το τελευταίο επιτρέπει την ελεγχόμενη αποστολή HTTP αιτημάτων. (Anicas, 2014). Αντίθετα οι Docker web clients (Colton Smith J. D., 2017), εκτός του ότι φορτώνουν το Docker σύστημα, δεν έχουν την δυνατότητα να αποστείλουν HTTP αιτήματα με ελεγχόμενο τρόπο.



Εικόνα 10. Το Γραφικό περιβάλλον διαχείρισης του JMeter

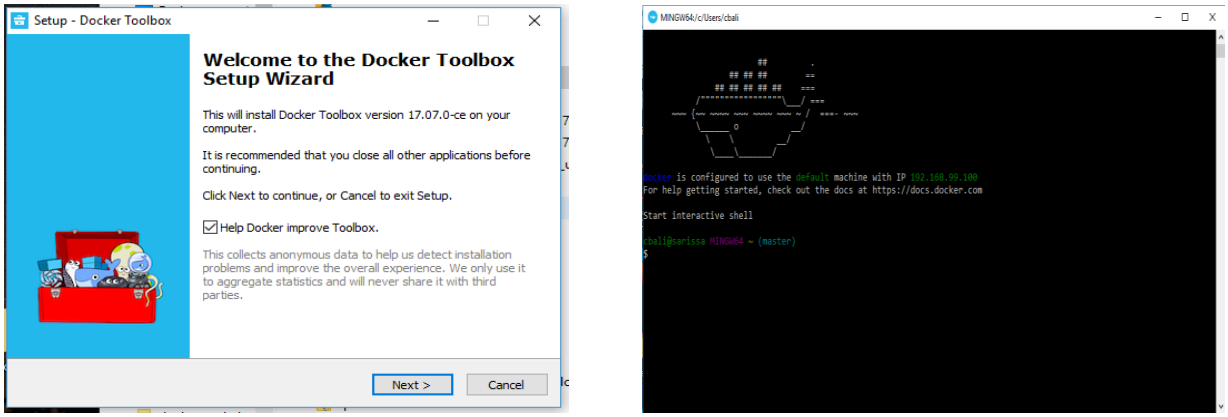
Το αποτέλεσμα του συνδυασμού αυτού είναι η εκτέλεση ενός μηχανισμού αυτοματοποιημένης κλιμάκωσης και αποκλιμάκωσης που για λόγους ευχερούς αναφοράς θα καλείται εφεξής με τον όρο Auto-Scaling Load Balancing(ASLB) .

5.2. Αναλυτική τεχνική περιγραφή των δομικών στοιχείων του Συστήματος

Στην συνέχεια περιγράφεται αναλυτικά η δημιουργία της συστοιχίας που χρησιμοποιήθηκε στα σενάρια δοκιμών που ακολουθούν στο επόμενο κεφάλαιο. Σε κάθε παράγραφο γίνεται προσπάθεια να παρουσιαστούν πληρέστερα τα πιο σημαίνοντα τεχνικά χαρακτηριστικά του εξεταζόμενου θέματος.

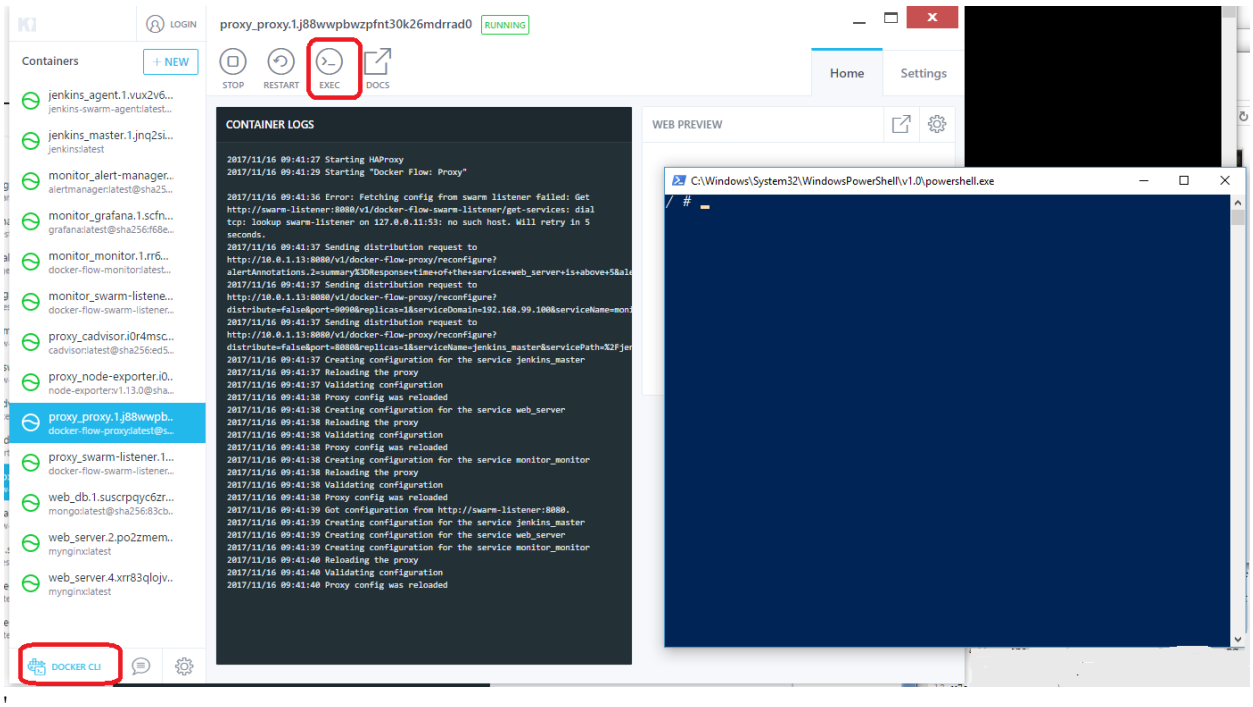
5.2.1. Δημιουργία της Υποδομής κόμβου

Με την εγκατάσταση του Docker Toolbox που εκτελείται με τον παρακάτω οδηγό δημιουργείται η υποδομή ενός κόμβου(ή μιας VirtualBox εικονικής μηχανής) που ονομάζεται default. Από την διαδικασία δοκιμής και σφάλματος διαπιστώθηκε πως απαιτούνται 8192MB RAM για την ικανοποιητική υποστήριξή των δοκιμών. Είναι σαφές για την υποστήριξη των τριών κόμβων της κανονικής λειτουργίας του Docker Swarm απαιτεί πόρους που δεν μπορούν να υπάρξουν σε συνηθισμένο μηχάνημα. Τελικά το ενδιαφέρον είναι πως με την ολοκλήρωση της εγκατάστασης δημιουργείται ένας σύνδεσμος για τερματικό γραμμής εντολών που παρέχει απευθείας πρόσβαση στο εγκατεστημένο περιβάλλον Docker το Docker Quickstart Terminal. Το τελευταίο δεν είναι παρά μια έκδοση τύπου unix τερματικού προσαρμοσμένου στο Docker, που όμως τρέχει στα Windows.

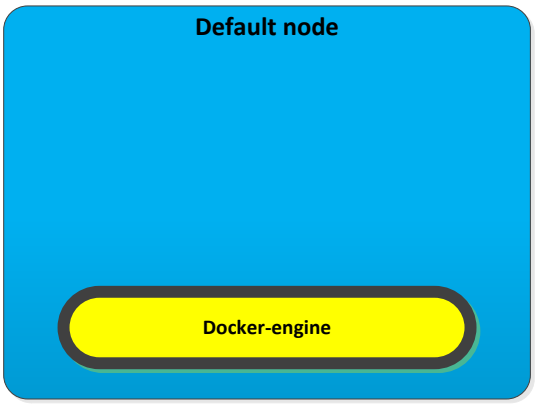


Εικόνα 11. Το Docker Toolbox

Ένα άλλο εργαλείο διαχείρισης που παρέχει το Docker Toolbox είναι η εφαρμογή πελάτη Kitematic που επιτρέπει την διαχείριση των container με ευέλικτο τρόπο. Για παράδειγμα επιτρέπει την πρόσβαση σε παράθυρο γραμμής εντολών μέσω των επιλογών που έχουν επισημανθεί με κόκκινο στην κάτωθι εικόνα. Η πρόσβαση αποδείχθηκε ιδιαίτερα χρηστική κατά τις δοκιμές που ακολουθούν. Σημειώνουμε πως το Kitematic παρακολουθεί στη παρούσα έκδοση μόνο τον default κόμβο.



Εικόνα 12 Η εφαρμογή Kitematic (Alpha)



Εικόνα 13. Η Μηχανή(κόμβος) του Docker Swarm-κενή

Για την δημιουργία του κόμβου αρχικά χρησιμοποιήθηκε η εντολή

eval $(docker-machine env default)

Η οποία επιτρέπει στην Μηχανή του Docker να έχει πρόσβαση στις μεταβλητές περιβάλλοντος του κόμβου. Στην συνέχεια χρησιμοποιείται για μια φορά μόνο η εντολή *docker swarm init* που δίδει στο Swarm την αρχική του μορφή ενώ η παράμετρος --advertise-addr είναι η διεύθυνση IP που θα εκθέσει ο κόμβος για να επιτυγχάνεται η πρόσβαση του. Στην συνέχεια, κάθε φορά που ξεκινά η μηχανή δεν απαιτείται η εκτέλεσή της.

5.2.2. Δημιουργία των απαραίτητων δικτύων.

Για την επικοινωνία μεταξύ των υπηρεσιών container απαιτείται η δημιουργία ενός δικτύου για την διαχείριση των λειτουργικών τμημάτων του δικτύου που παίρνει την γενική ονομασία proxy. Παράλληλα για την επιτήρηση του συστήματος αλλά και την αλληλεπίδραση μεταξύ των υπηρεσιών Docker κρίθηκε αναγκαία η

δημιουργία ενός δικτύου που έχει την γενική ονομασία monitor. Και τα δύο παρατάχθηκαν ως overlay δίκτυα ώστε να υπάρχει συμβατότητα ανεξάρτητα από το αν έχουμε έναν η περισσοτέρους κόμβους. Στην διαδικασία σχηματισμού του Docker Swarm δημιουργείται εξορισμού το δίκτυο **ingress** και το **Docker_gwbridge** που έχουν ρόλο αντίστοιχα εξισορροπητή φόρτου ανάμεσα σε κόμβους και διασύνδεσης γέφυρας με τον ανά κόμβο Docker δαίμονα. Λόγω όμως της ιδιαίτερης δομής του συστήματος δεν έχουν ιδιαίτερη χρησιμότητα ώστε να μας απασχολήσουν.

```
docker network create -d overlay monitor
docker network create -d overlay proxy
```

Εικόνα 14. Οι εντολές δημιουργίας των απαραιτήτων δικτύων

5.2.3. Δομικά χαρακτηριστικά των εντολών δημιουργίας του Εξισορροπητή Φόρτου(HAproxy).

Όταν μια μηχανή Docker έχει τεθεί σε λειτουργία Swarm τότε είναι δυνατόν να εκτελείται η εντολή Docker stack deploy για να παρατάξει μια σειρά από υπηρεσίες στο Swarm (Docker-Stack, 2017). Η περιγραφή των εικόνων που θα χρησιμοποιηθούν καθώς και των παραμέτρων τους δίνονται με την μορφή ενός yaml αρχείου της έκδοσης 3.0 ή υψηλότερης. Η YAML™ είναι μια γενική γλώσσα φιλική προς τον άνθρωπο που επιτρέπει την αποθήκευση των δεδομένων κατά σειριακό τρόπο. Αν και θα μπορούσε να έχει την χρήση που έχει αποκτήσει η JSON που είναι υποσύνολο της εντούτοις γνωρίζει εκτεταμένη χρήση στην δημιουργία αρχείων διαμόρφωσης και σύνθεσης συστημάτων λογισμικού. (YAML, 2009). Έτσι για παράδειγμα βλέπουμε την εντολή :

```
docker stack  deploy c proxy.yml proxy
```

Παρατηρούμε πως εντολή αναφέρεται σε ένα αρχείο .yml το οποίο θα πρέπει να είναι οπωσδήποτε τουλάχιστον έκδοσης 3.0. Θα πρέπει να έχει πρόσβαση και στο δίκτυο proxy που προαναφέρθηκαν. Επίσης θα πρέπει να λαμβάνει υπόψη του δύο υπηρεσίες container α) του proxy που είναι ένας ειδικά διαμορφωμένος HAproxy ώστε να επιτρέπει την δυναμική αναδιαμόρφωση ανάλογα με τις απαιτήσεις. β) Docker Flow Swarm Listener έχει ως ρόλο το να ακούει τα γεγονότα Docker Swarm όποτε προκύπτει μια αλλαγή. Αυτό μπορεί να οδηγήσει σε αλλαγές προς τον HAproxy όταν μια νέα υπηρεσία έχει δημιουργηθεί ή όταν μια υπηρεσία έχει αφαιρεθεί από την συστοιχία. Επομένως συνεργατικά τα δύο αυτά services επιτυγχάνουν την δυναμική διαμόρφωση του εξισορροπητή φόρτου. Περισσότερες λεπτομέρειες παρατίθενται στο αποθετήριο κώδικα της εργασίας στο αρχείο proxy.yml (9.2). Θα πρέπει να σημειωθεί πως με την εντολή :

docker stack ps -f desired-state=running proxy,

μπορούμε να δούμε τις υπηρεσίες που τρέχουν σε αυτή την στοίβα υπηρεσιών με την γενική ονομασία proxy. Τονίζεται πως για λόγους διευκόλυνσης της παρακολούθησης του συστήματος έχουν προστεθεί δύο ακόμη υπηρεσίες :

- Η πρώτη είναι ο cAdvisor (Container Advisor) της Google που παρέχει στους χρήστες των container μια σημαντική κατανόηση της χρήσης των πόρων και τα χαρακτηριστικά της απόδοσης των container που τρέχουν σε ένα κόμβο. Θα μπορούσε να γίνει κατανοητή ως υπηρεσία δαίμονα που τρέχει και συλλέγει, συγκεντρώνει, επεξεργάζεται και αποστέλλει δεδομένα σχετικά με τους container εν λειτουργία. Ειδικότερα για κάθε container κρατά τις παραμέτρους απομόνωσης των πόρων, ιστορικά στοιχεία χρήσης πόρων, ιστογράμματα χρήσης πόρων και στατιστικά δικτύων. Αυτά τα δεδομένα μπορούν να εξαχθούν τόσο ανά container όσο και επίπεδο κόμβου (cAdvisor, 2015).

- Η δεύτερη είναι η node-exporter που είναι στην ουσία μια υπηρεσία container γραμμένη σε προγραμματιστική γλώσσα Go για να εξάγει μετρικές σχετικές με το υλισμικό και το λειτουργικό σύστημα όταν είναι τύπου unix. Αποδέκτης είναι το Prometheus που θα εξετάσουμε παρακάτω.

5.2.4. Δομικά χαρακτηριστικά των εντολών δημιουργίας του Μηχανισμού Παρακολούθησης

Για τους σκοπούς της διαδικασίας παρακολούθησης τους συστήματος χρησιμοποιήθηκαν ορισμένα δομικά στοιχεία. Η ύπαρξη των οποίων επιτρέπει την εκτέλεση της εντολής *docker stack deploy -c monitor.yml monitor*. Αν κάποια ρύθμιση δεν ληφθεί όπως πρέπει τότε η εντολή :

docker stack ps -f desired-state=running monitor

Δεν δίνει την παρακάτω εικόνα που είναι απαραίτητη για την ομαλή λειτουργία της στοίβας του monitor.

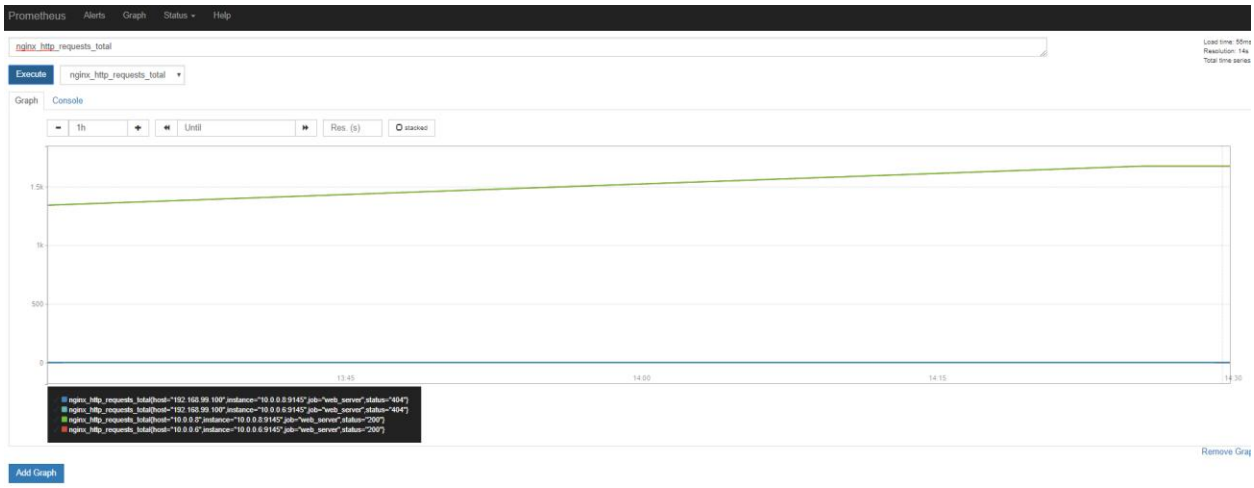
```
$ docker stack ps -f desired-state=running monitor
ID                  NAME                    IMAGE                                          NODE                DESIRED STATE       CURRENT STATE            ERROR
kt636vn1g4d7        monitor_swarm-listener.1   vfarcic/docker-flow-swarm-listener:latest   default             Running             Running 2 hours ago
xee4tcvoqvte        monitor_alert-manager.1    prom/alertmanager:latest                    default             Running             Running 2 hours ago
28wj6cgc0a7d        monitor_grafana.1          grafana/grafana:latest                      default             Running             Running 2 hours ago
wrz7f99w4hz8        monitor_monitor.1          vfarcic/docker-flow-monitor:latest          default             Running             Running 2 hours ago
```

Εικόνα 15. Η στοίβα του monitor στο Docker Swarm

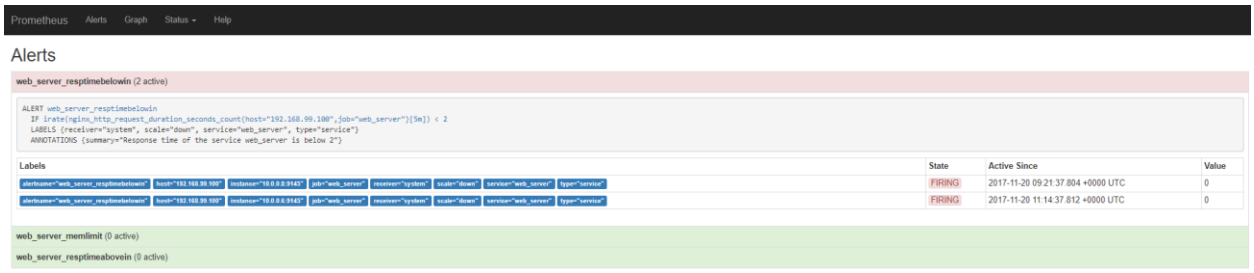
5.2.4.1. Το εργαλείο Prometheus

Το Prometheus είναι πρωταρχικά μια βάση δεδομένων χρονοσειρών. Με άλλα λόγια ροές δεδομένων με χρονοσημάνσεις που ανήκουν σε μια έννοια μετρικής και στην ίδια διάσταση. Πέρα από μια τυπική βάση αποθήκευσης δεδομένων χρονοσειρών έχει την δυνατότητα να δημιουργεί παράγωγες χρονοσειρές ως αποτέλεσμα εφαρμοζόμενων ερωτημάτων. Ένα άλλο χαρακτηριστικό του Prometheus που το

διαφοροποιεί από άλλα συστήματα παρακολούθησης είναι ότι λειτουργεί με ένα μοντέλο pull-based. Βασική προϋπόθεση λειτουργίας του είναι οι εξεταζόμενες εφαρμογές να εκθέτουν μόνες τους τις μετρικές που επιθυμούν. Οι τελευταίες τράβιουνται(pull) από τον εξυπηρετητή του Prometheus αντί να αποστέλλονται απευθείας στον Prometheus π.χ. με την βοήθεια κάποιου agent. Το Prometheus UI είναι κεντρικό μενού που επιτρέπει την υποβολή ερωτημάτων αλλά και τη σχεδίαση των αντίστοιχων σχεδιαγραμμάτων.



Εικόνα 16 Η Διεπαφή χρήστη του Prometheus



Εικόνα 17 Η Διεπαφή χρήστη του Prometheus-Alertmanager

Το πιο σημαντικό όμως κομμάτι που μας ενδιαφέρει στο πλαίσιο αυτής της εργασίας είναι το κομμάτι που ονομάζεται Alertmanager και το οποίο στην λογική

των μικρο-υπηρεσιών θα μπορούσε να αποτελεί ξεχωριστό container. Σε κάθε περίπτωση αυτό το κομμάτι λογισμικού δύναται να συγκεντρώνει και δημιουργεί συναγερμούς και να προωθεί σχετικές ειδοποιήσεις σε άλλες οντότητες λογισμικού (Mouat, 2015). Από απλό e-mail, υπηρεσία ειδοποίησης (λ.χ. slack) μέχρι εξειδικευμένες ειδοποιήσεις (λ.χ. προς το Jenkins όπως περιγράφεται στην συνέχεια). Θα πρέπει τέλος να ληφθεί υπόψη πως και στη σχετική στοίβα υπάρχει επίσης μια monitor_swarm-listener έχει ως ρόλο το να ακούει τα γεγονότα Docker Swarm όποτε απαιτείται να ληφθεί υπόψη μια αλλαγή.

5.2.4.2. **Η Γλώσσα ερωτημάτων του Prometheus**

Το πιο δυνατό σημείο του λογισμικού του Prometheus είναι η γλώσσα ερωτημάτων που έχει ενσωματωμένη. Πρόκειται στην ουσία για γλώσσα λειτουργικών εκφράσεων που επιτρέπει στον χρήστη να επιλέγει και να συσσωματώνει τα δεδομένα μιας χρονοσειράς σε πραγματικό χρόνο. Τα αποτελέσματα θα μπορούσαν να διατυπωθούν είτε αριθμητικά ή γραφικά. Βασικοί τύποι στους οποίους εφαρμόζονται οι εκφράσεις αυτές είναι:

- Ο **instant vector** , δηλαδή ένα σύνολο από δεδομένα χρονοσειρών από κάθε χρονοσειρά που όλες έχουν την ίδια χρονοσήμανση.
- Ο **range vector** , δηλαδή ένα σύνολο από χρονοσειρές που περιέχουν ένα εύρος δεδομένων κατά το πέρασμα του χρόνου για κάθε χρονοσειρά.
- Ο **scalar,** δηλαδή ένας απλός αριθμός κινητής υποδιαστολής.

Ανάλογα με την περίπτωση (δηλαδή παρατήρηση απλής τιμής ή γραφική απεικόνιση) χρησιμοποιείται ο αντίστοιχος τύπος. Έτσι αν για παράδειγμα μια έκφραση επιστρέφει ένα **instant vector** αυτό μπορεί να γίνει κατανοητό μόνο με

γραφική απεικόνιση. Για να γίνουν κατανοητές οι μετρικές που θα χρησιμοποιηθούν στην συνέχεια θα πρέπει να αναλύσουμε ιδιαίτερα το instant vector και το range vector:

Επιλογέας Instant vector

Ο επιλογέας Instant vector επιτρέπει την επιλογή ενός συνόλου από χρονοσειρές με ένα δείγμα τιμής από κάθε χρονοσήμανση (στιγμιότυπο). Στην απλούστερη μορφή του ορίζεται μόνο ένα όνομα μετρικής. Το αποτέλεσμα είναι ένα instant vector που περιέχει τα δεδομένα της χρονοσειράς που έχουν αυτό το μετρικό όνομα. Για παράδειγμα το nginx_http_requests_total επιλέγει τα δεδομένα χρονοσειράς που έχουν αυτή την ετικέτα μετρικής δηλαδή nginx_http_requests_total. Επιπρόσθετα είναι δυνατόν να φιλτραριστεί με την προσθήκη αγκύλης ({}). Επί παραδείγματι :

- nginx_http_requests_total {host="docker_host_ip", job="web_server", status="200"}

Οπότε και επιστρέφει μόνο την χρονοσειρά δεδομένων με job="web_server", status="200" και host="docker_host_ip".

Είναι επίσης πιθανόν να χρησιμοποιηθούν regular expressions:

- = : Επιλέγει ετικέτες που ταυτίζονται με το παρεχόμενο αλφαριθμητικό.
- != : Επιλέγει ετικέτες που δεν ταυτίζονται με το παρεχόμενο αλφαριθμητικό.
- =~ : Επιλέγει ετικέτες που ταυτίζονται με το παρεχόμενο αλφαριθμητικό ή μέρος αυτού.
- !~ : Επιλέγει ετικέτες που δεν ταυτίζονται με το παρεχόμενο αλφαριθμητικό ή μέρος αυτού.

Επιλογέας Range Vector

Ο επιλογέας Range vector λειτουργεί όπως ο επιλογέας instant vector με την διαφορά ότι επιλέγει ένα εύρος δειγμάτων από το τρέχων στιγμιότυπο. Συντακτικά σημειώνεται με ([ ]) εντός του οποίου τοποθετείται το χρονικό διάστημα μέσα από το οποίο αντλούνται τιμές. Τυπικά μπορεί να είναι s – δευτερόλεπτα ή m – λεπτά. Μπορεί όμως σε άλλες περιπτώσεις να τεθεί και μεγαλύτερη χρονική μονάδα. Για παράδειγμα η μετρική:

- container_network_receive_bytes_total{container_label_com_Docker_swarm_service_name="web_server"}[1m]

Συλλέγει δείγματα στο 1 m - λεπτό.

Θα πρέπει επίσης να σημειωθεί πως η εξεταζόμενη γλώσσα διαθέτει τελεστές πράξεων όπως κάθε άλλη προγραμματιστική γλώσσα. Η εξαντλητική αναφορά των οποίων δεν συνεισφέρει σημαντικά στη παρούσα ανάλυση. Ενδιαφέρον παρουσιάζουν οι τελεστές Aggregation. Δηλαδή πρόκειται για ενσωματωμένους τελεστές που μπορούν να χρησιμοποιηθούν για να συγκεντρώσουν τα στοιχεία ενός instant vector οδηγώντας σε ένα νέο vector, το αποτέλεσμα του τελεστή. Για παράδειγμα το sum υπολογίζει το άθροισμα επί των διαστάσεων της μετρικής. Αντίστοιχα το count μας επιστρέφει τον αριθμό των στοιχείων σε ένα vector. Επειδή ακριβώς μια μετρική μπορεί να μεταφέρει αρκετές διαστάσεις μπορεί να παρίστανται όλες οι διαστάσεις ή μόνο μερικές θέτοντας την all, by ή without λέξη κλειδί. Επιπλέον διαθέτει ένα ευρύ ρεπερτόριο ενσωματωμένων συναρτήσεων. Θα σταθούμε ιδιαίτερα στις :

rate() όπου το εισερχόμενο είναι range-vector και υπολογίζει το ανά δευτερόλεπτο μέσο όρο αύξησης της χρονοσειράς στο εύρος του range-vector. Επίσης εκτελεί όλες τις απαραίτητες μαθηματικές ενέργειες ώστε οι χαμένες τιμές να μην μεταβάλουν σημαντικά τον μορφότυπο της χρονοσειράς. Για παράδειγμα η πιο κάτω μετρική μας δίνει το μέσο ποσοστό χρήσης της CPU για την υπηρεσία με το όνομα web_server στο τελευταίο 1 λεπτό.

- rate(container_cpu_user_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[1m]) * 100

Είναι εύλογο πως η rate θα πρέπει να χρησιμοποιηθεί με μετρητές και είναι ιδανική για αργά μεταβαλλόμενες ποσότητες μετρητών.

irate() όπου το εισερχόμενο είναι range-vector και υπολογίζει την ανά δευτερόλεπτο στιγμιαία αύξηση των τιμών της χρονοσειράς στο εύρος του range-vector. Όπως και στην προηγούμενη περίπτωση εκτελεί όλες τις απαραίτητες μαθηματικές ενέργειες ώστε οι χαμένες τιμές να μην μεταβάλουν σημαντικά τον μορφότυπο της χρονοσειράς. Έτσι, η πιο κάτω μετρική μας επιστρέφει το τρέχων ανά δευτερόλεπτο ρυθμό των HTTP αιτημάτων στους nginx εξυπηρετητές με status="200" εξετάζοντας ένα λεπτό πίσω:

- irate(nginx_http_requests_total{host="docker_host_ip",job="web_server",status="200"}[1m])

Είναι σαφές επομένως πως μπορεί να ανιχνεύει ταχύτατες μεταβολές και είναι ιδανική για γραφικές απεικονίσεις.

5.2.4.3. Επεξήγηση των παρακολουθούμενων μετρικών

Στην συνέχεια παραθέτουμε τις μετρικές που χρησιμοποιήθηκαν για την παρακολούθηση του συστήματος.

- **irate(container_network_receive_bytes_total{container_label_com_Docker_swarm_service_name="web_server"}[1m])**

Η συγκεκριμένη μετρική υπολογίζει την ανά δευτερόλεπτο στιγμιαία αύξηση των λαμβανόμενων bytes από το δίκτυο για την υπηρεσία web_server στο περασμένο ένα λεπτό.

- irate(container_network_transmit_bytes_total{container_label_com_Docker_swarm_service_name="web_server"}[1m])

Η πιο πάνω μετρική υπολογίζει την ανά δευτερόλεπτο στιγμιαία αύξηση των αποστελλόμενων bytes προς το δίκτυο από την υπηρεσία web_server στο περασμένο ένα λεπτό.

- rate(container_cpu_user_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[1m]) * 100

Όπως αναφέρθηκε και νωρίτερα αυτή η έκφραση μας δίνει το μέσο ποσοστό χρήσης της CPU για την υπηρεσία με το όνομα web_server στο περασμένο ένα λεπτό.

- sum by (id, instance)(rate(container_cpu_usage_seconds_total[1m])) / count by (id, instance)(container_cpu_usage_seconds_total)

Με την συγκεκριμένη έκφραση παρατηρούμε το μέσο ποσοστό αύξησης της ως προς την μονάδα της χρήσης της CPU ανά κόμβο και υπηρεσία στο περασμένο ένα λεπτό.

- sum (rate (container_cpu_usage_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[1m])) / sum (machine_cpu_cores) * 100

Αντίστοιχα με την συγκεκριμένη έκφραση παρατηρούμε το μέσο ποσοστό αύξησης της ως προς το επί τοις εκατό(%) της χρήσης της CPU ανά αριθμό CPU cores για την υπηρεσία web_server στο περασμένο ένα λεπτό.

- (irate(container_cpu_usage_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[5m]))

Επειδή μας ενδιαφέρει ιδιαίτερα η συμπεριφορά των υπηρεσιών web_server με την συγκεκριμένη έκφραση παρατηρούμε το στιγμιαίο ποσοστό αύξησης της χρήσης της CPU ανά αριθμό CPU cores για την υπηρεσία web_server στο περασμένο πεντάλεπτο.

- irate(nginx_http_requests_total{host="docker_host_ip",job="web_server",status="200"}[1m])

Πρόκειται ίσως για την πιο κομβική μετρική που μας επιστρέφει το τρέχων ανά δευτερόλεπτο ρυθμό των HTTP αιτημάτων στους nginx εξυπηρετητές(web_server) με status="200" εξετάζοντας ένα λεπτό πίσω. Φυσικά όταν προέρχονται από τον host="docker_host_ip" δηλαδή είναι αιτήματα που προέρχεται από εξωτερικούς χρήστες. Αποτελεί και κριτήριο ενεργοποίησης συναγερμών.

- irate(node_cpu{mode="idle"}[5m])

Αποτελεί μια τυπική μετρική που ουσιαστικά μας δίνει το κατά πόσο είναι ήρεμος (idle) ο κόμβος και μετρά τις στιγμιαίες μεταβολές σε βάθος 5 λεπτών

- sum(up{job="web_server"})

Η έκφραση αυτή μας αποτυπώνει τον ακριβή αριθμό στιγμιότυπων που τρέχουν ανά πάσα στιγμή. Είναι ιδιαίτερα χρήσιμη για την παρακολούθηση του αριθμού των nginx εξυπηρετητών και την εκτέλεση των δοκιμών που εξετάζονται στην συνέχεια.

5.2.4.4. Τα docker secret ως μέθοδος παραμετροποίησης

Εντός των υπηρεσιών του Docker Swarm ένα secret μπορεί να είναι ένα μεγάλο δυαδικό αντικείμενο όπως ένα password, ένα SSH ιδιωτικό κλειδί, ένα SSL πιστοποιητικό, ή οποιαδήποτε άλλη πληροφορία που δεν θα έπρεπε να μεταφερθεί ή να αποθηκευτεί μη κρυπτογραφημένη. Έτσι από την έκδοση 1.13 και τις νεώτερες εκδόσεις τα docker secret μπορούν να χρησιμοποιηθούν ώστε να υπάρχει κεντρική διαχείριση και διανομή μόνο για τους container που απαιτείται να έχουν πρόσβαση στην πληροφορία αυτή. Μέσα στο Docker Swarm, τα docker secret παραμένουν κρυπτογραφημένα τόσο κατά την μεταφορά όσο και κατά την αποθήκευση. Ένα δεδομένο secret μπορεί να είναι προσβάσιμο μόνο σε εκείνες τις υπηρεσίες στις οποίες έχει παραχωρηθεί πρόσβαση ρητά. Και μόνο για το χρονικό διάστημα που αυτές οι υπηρεσίες τρέχουν. Φυσικά μπορεί να αποθηκεύσει κάθε ευαίσθητη πληροφορία που χρειάζεται ένας container για να λειτουργήσει κατά τον τρόπο αυτό. Στο πλαίσιο της εργασίας αυτή χρησιμοποιούμε τα docker secret ως ένα γενικό δυαδικό περιεχόμενο για να περάσουμε ευαίσθητη πληροφορία για την παραμετροποίηση του Prometheus Alertmanager. Στην συνέχεια θα δούμε την χρήση τους για το όνομα και τον κωδικό πρόσβασης στο Jenkins. Βέβαια η συγκεκριμένη λειτουργικότητα αποκτά ιδιαίτερο νόημα όταν υπάρχουν διαχωρισμένα περιβάλλοντα για ανάπτυξη, δοκιμή και παραγωγή μιας εφαρμογής. Κάθε ένα από αυτά τα περιβάλλοντα θα πρέπει να έχει διαφορετικά docker secret. Οι container χρειάζονται μόνο να γνωρίζουν τα συγκεκριμένα docker secret ώστε να λειτουργήσουν στο εκάστοτε παραγωγικό περιβάλλον.(Docker-secrets, 2017). Στο αποθετήριο κώδικα παρατίθεται ένα τέτοιο docker secret το alert_manager_config(πρβ. 9.2) που μας δείχνει ποια δομή θα έχει η διαμόρφωση του alertmanager. Έτσι στο monitor.yml περιγράφεται το πώς θα διαβαστεί το alert_manager_config secret από τον alertmanager container.

```
alert-manager:
  image: prom/alertmanager
  networks:
    - monitor
  secrets:
     - alert_manager_config
  command: -config.file=/run/secrets/alert_manager_config -
storage.path=/alertmanager
```

5.2.4.5. Το Περιβάλλον Granfana.

Για να μπορέσουμε να αποκτήσουμε μια εποπτική εικόνα των δεδομένων των μετρικών θα ήταν καλό να χρησιμοποιηθεί ένα εργαλείο ταμπλό(συνοπτικής γραφικής αναπαράστασης) όπως το PromDash του Prometheus. Το τελευταίο μάλιστα θεωρείται απαρχαιωμένο και οι ίδιο οι δημιουργοί του προτείνουν το Grafana(PromDash, 2017). Το Grafana είναι ένα εργαλείο ανοικτού λογισμικού που παρέχει πλούσια χαρακτηριστικά ταμπλό και διαχείρισης γραφικών για μια σειρά βάσεων χρονοσειρών όπως το Graphite, Elasticsearch, OpenTSDB,InfluxDB και φυσικά το Prometheus (Grafana, 2017).

Εικόνα 18 Το ταμπλό του Grafana στο πλαίσιο της εργασίας

Η συμβατότητα του Grafana με το Prometheus φαίνεται και από το γεγονός ότι οι μετρικές που λαμβάνονται από το Prometheus, τοποθετούνται στο Grafana και εφόσον έχουν γίνει οι ανάλογες γενικές ρυθμίσεις είναι δυνατόν να αρχίσει απευθείας η αποτύπωση των γραφικών. Θα πρέπει να τονισθεί πως οι μετρικές που αναφέρθηκαν σε προηγούμενη παράγραφο μπορούν να αποτυπωθούν μαζί με όλες τις σχετικές παραμέτρους σε ένα JSON αρχείο παραμετροποίησης.

Έτσι κάθε φορά που εκκινείται το σύστημα είναι δυνατόν να φορτώνεται το αρχείο αυτό και να μην απαιτείται η χρονοβόρα ρύθμιση των διαγραμμάτων. Φυσικά ο σχετικός ισότοπος παρέχει αρκετές πληροφορίες για επιπρόσθετα JSON αρχεία παραμετροποίησης που όμως εκφεύγουν από τους στόχους αυτής της εργασίας.

5.2.5. **Η χρήση του Jenkins.**

Η χρήση της έννοιας Continuous Integration (CI) ήταν το πρώτο βήμα που επιτάχυνε τον χρόνο ανάπτυξης ενός προϊόντος λογισμικού. Αυτό έγινε κυρίως κάτω από την πίεση των προγραμματιστών να υποβάλλουν συχνά τις αλλαγές

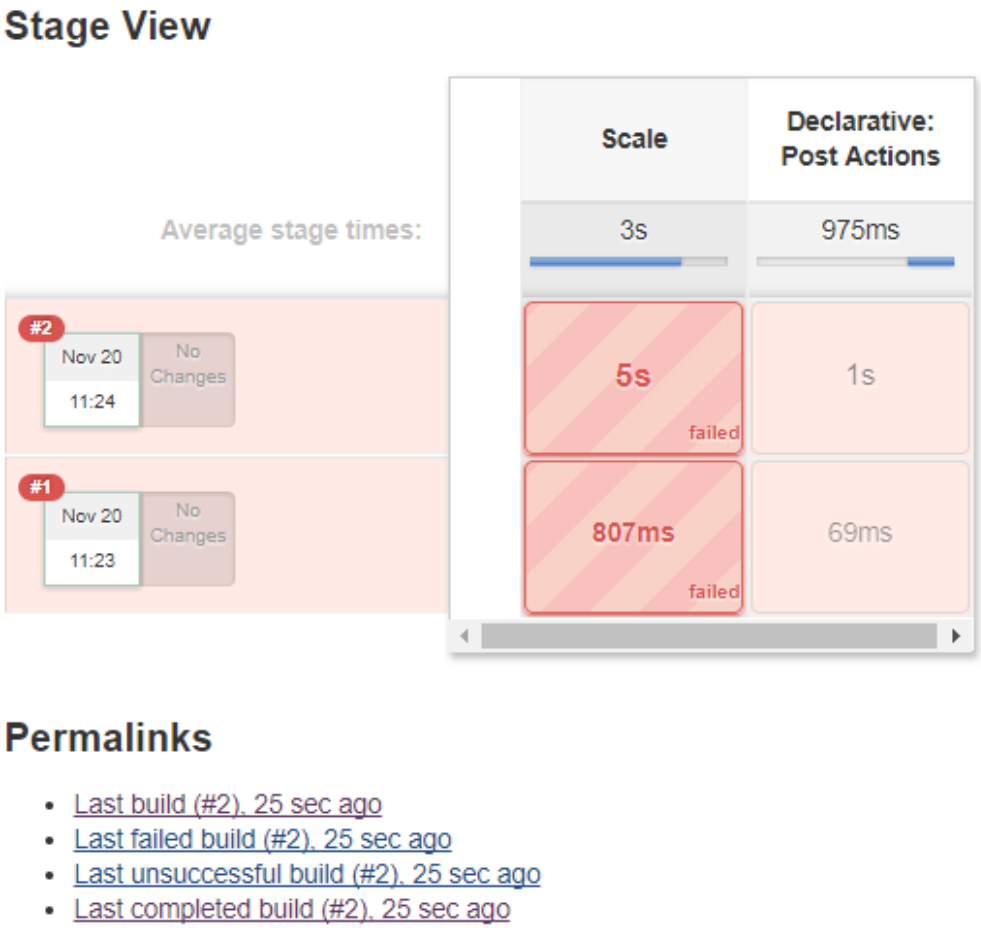
κώδικα σε ένα διαμοιραζόμενο αποθετήριο πηγαίου κώδικα. Παράλληλα ήταν απαραίτητο να ενεργοποιείται ένας μηχανισμός δοκιμών μονάδων σύμφωνα με την τάση της προσέγγισης οδηγούμενης από δοκιμές. Αυτό βοήθησε στην άμεση διαπίστωση σφαλμάτων και δυσλειτουργιών. Επομένως επιταχύνθηκε σημαντικά ο χρόνος παράδοσης παραγωγικών εφαρμογών. Παράλληλα η διαδικασία αυτή έφερε πιο κοντά τους προγραμματιστές, τους ελεγκτές ποιότητας και τους διαχειριστές των παραγωγικών συστημάτων. Καθώς είναι γεγονός πως στο παρελθόν λειτουργούσαν απομονωμένα. Αυτή σύγκλιση οδήγησε στην έννοια του Συνεχούς Παράδοσης(Continuous Delivery ή CD). Η οποία πρεσβεύει την αυτοματοποίηση όλου του κύκλου ζωής ενός λογισμικού συστήματος. Μια πλατφόρμα η οποία ακολούθησε αυτή την εξέλιξη ήταν και το Jenkins. Θα πρέπει να αναφερθεί πως πρόκειται για ένα εργαλείο γραμμένο σε Java. Η πλατφόρμα αυτή δηλαδή θα μπορούσε να θεωρηθεί ως ένα εργαλείο CI ανοικτού λογισμικού που πρωταρχικό στόχο είχε τις διαδικασίες του συνεχούς build και test. Αυτό που έδωσε ιδιαίτερη ώθηση στο Jenkins είναι η ευέλικτη αρχιτεκτονική του στην εισαγωγή πρόσθετων που εξειδικεύουν την λειτουργικότητά του. Μάλιστα τα πρόσθετα αυτά υποστηρίζονται από την κοινότητα ανοικτού λογισμικού που έχει υιοθετήσει το Jenkins. Βέβαια από την κατασκευή του πρόκειται για, κατά κάποιο τρόπο, μονολιθική εφαρμογή. Κυρίως γιατί δεν μπορεί υποστεί κλιμάκωση ανάλογα με τις απαιτήσεις ενός συστήματος. Ένας τρόπος να ξεπεραστεί η δυσκαμψία αυτή είναι να δημιουργηθεί ένας μηχανισμός master-slave και για την ακρίβεια σε ένα περιβάλλον Docker Swarm θα πρέπει να αναπτυχθεί ένα Jenkins master στιγμιότυπο που θα επικοινωνεί με ένα Jenkins agent στιγμιότυπο ανά κόμβο. Στην περίπτωση της εργασίας όπου χρησιμοποιείται ο default κόμβος θα αναπτυχθεί ένας μόνο Jenkins agent.

```
$ docker service ls
ID            NAME            MODE        REPLICAS    IMAGE                              PORTS
ok61aiil6r4q  jenkins_master  replicated  1/1         localhost:5000/jenkins:latest      *:50000->50000/tcp
wxtejlit2r9w  jenkins_agent   replicated  1/1         vfarcic/jenkins-swarm-agent:latest
```

Εικόνα 19 Η λειτουργική δομή του Jenkins

Το Jenkins μπορεί να θεωρηθεί ως ένας ενορχηστρωτής που η χρησιμότητα του ποικίλει από την αποδέσμευση κώδικα και των δοκιμών του, τον στατικό μηχανισμό ανάλυσης κώδικα μέχρι τις δοκιμές επίδοσης και αποδέσμευση παραγωγικού λογισμικού. Στο παρελθόν με βάση την χρήση πρόσθετων γινόταν η υλοποίηση ενός αγωγού παροχέτευσης εργασιών. Γρήγορα όμως δημιουργήθηκε μια μηχανή ροής εργασιών ως ένα μόνο πρόσθετο για να ανταποκριθεί με συνέπεια στην τάση του Jenkins ως εργαλείο CD. Για να επιτευχθεί αυτό στην ουσία το πρόσθετο αναδιαμορφώνει την γραφή scripts με την προώθηση της νέας Groovy Domain Specific Language (DSL) που είναι ευέλικτη και επιτρέπει την εύκολη παραμετροποίηση του συστήματος. (Armenise, 2015).

Η τελευταία εξέλιξη αποτέλεσε την μέθοδο που χρησιμοποιήθηκε το Jenkins πλαίσιο της εργασίας.



Εικόνα 20 Άποψη πρόσθετου αγωγού (pipeline)

Έτσι στην συνέχεια παραθέτουμε να στοιχεία εκείνα που απαρτίζουν το script κλιμάκωσης-αποκλιμάκωσης με την χρήση της DSL.

Αρχικά η όλη ροή ορίζεται σε ένα αγωγό (pipeline) όπου ορίζονται, το όνομα του agent δηλαδή prod και το όνομα της σχετικής δραστηριότητας δηλαδή service και scale. Το πιο σημαντικό κομμάτι που υλοποιεί την λογική της κλιμάκωσης είναι ένα stage που καλείται scale. Θα πρέπει να σημειωθεί πως το stage είναι διακριτό λογικό κομμάτι μιας διεργασίας. Έτσι για παράδειγμα στην γενική περίπτωση ενός παραγωγικού συστήματος CD θα μπορούσαμε να είχαμε εξής διακριτά stage ή στάδια:

- Τράβηγμα του κώδικα από το αποθετήριο.
- Τρέξιμο των τεστ και χτίσιμο των υπηρεσιών και των εικόνων Docker.
- Παράταξη των στιγμιότυπων των εικόνων αυτών στο περιβάλλον δοκιμών.
- Σήμανση των εικόνων Docker και αποθήκευσή τους στο σχετικό αποθετήριο.
- Χρήση των τελευταίων εικόνων Docker από το αποθετήριο για τις τελικές δοκιμές πριν την παραγωγική χρήση.
- Χρήση των τελευταίων εικόνων Docker από το αποθετήριο για παραγωγική χρήση και δοκιμή τους στη πράξη.

Από τη φύση του προβλήματος που εξετάζουμε επικεντρώνουμε το ενδιαφέρον μας στο τελευταίο στάδιο καθώς αυτό είναι που διερευνούμε στην παρούσα εργασία. Κάθε φορά που εκτελείται το stage του pipeline script σαρώνονται τα αρχεία του Docker με την εντολή *docker service inspect web_server* και το αποτέλεσμα τοποθετείται σε ένα JSON αντικείμενο. Έτσι μια σειρά από πληροφορίες είναι προσβάσιμες για την εκτέλεση αυτού του script. Στο πλαίσιο αυτό βλέπουμε πως βασικές παράμετροι που ήδη έχουν οριστεί από την αρχή είναι ο μέγιστος αριθμός των αντιγράφων(maxReplicas) και ο ελάχιστος αριθμός(minReplicas) των αντιγράφων για παράδειγμα 6 και 2 αντίστοιχα και καθώς και αριθμός των τρεχόντων αντιγράφων (currentReplicas). Όποτε ενεργοποιηθεί το script επίσης δημιουργείται το newReplicas που βασίζεται στο currentReplicas και τον αθροιστή κλιμάκωσης scale.toInteger() που είναι 1 αν υπάρχει ανάγκη να αυξηθεί ή -1 αν υπάρχει ανάγκη να μειωθεί ο αριθμός των στιγμιότυπων. Φυσικά εξετάζεται το αν ο αριθμός έχει ξεπεράσει τα όρια maxReplicas και minReplicas γεγονός που σημαίνει ότι τότε θα πρέπει να μην γίνει καμιά ενέργεια. Τελικά εφόσον επιτρέπεται, εκτελείται η εντολή sh “docker service scale $service=$newReplicas” δηλαδή η εντολή που οδηγεί την υπηρεσία web_server να αποκτήσει τόσα αντίγραφα όσα προβλέπει το newReplicas .

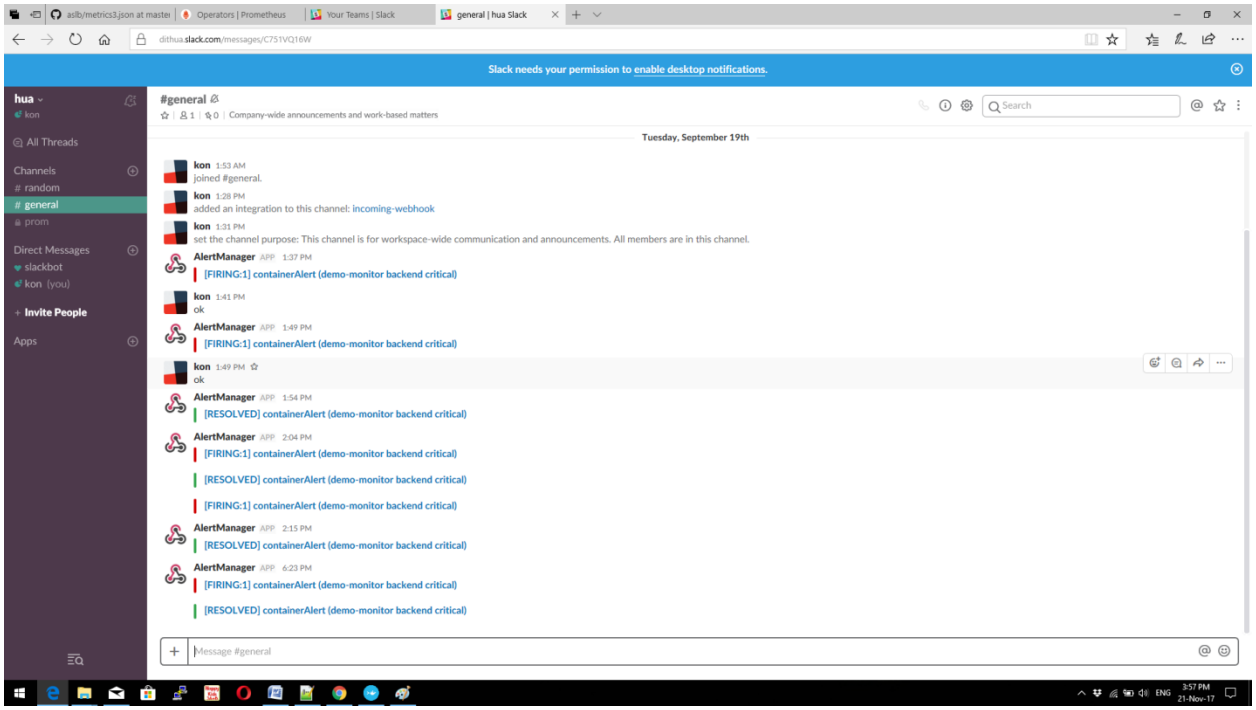
```
pipeline {
 }
 stages {
  stage("Scale") {
   steps {
    script {
     def inspectOut = sh(
      script: "Docker service inspect $service",
      returnStdout: true
     )
     def inspectJson = readJSON text: inspectOut.trim()
     def currentReplicas = inspectJson[0].Spec.Mode.Replicated.Replicas
     def newReplicas = currentReplicas + scale.toInteger()
     def minReplicas = inspectJson[0].Spec.Labels["com.df.scaleMin"].toInteger()
     def maxReplicas = inspectJson[0].Spec.Labels["com.df.scaleMax"].toInteger()
     if (newReplicas > maxReplicas) {
      error "$service is already scaled to the maximum number of $maxReplicas replicas"
     } else if (newReplicas < minReplicas) {
      error "$service is already descaled to the minimum number of $minReplicas replicas"
     } else {
      sh "Docker service scale $service=$newReplicas"
      echo "$service was scaled from $currentReplicas to $newReplicas replicas"
     }
    }
   }
  }
 }
}
```

Εικόνα 21 Το pipeline script του Jenkins

```
$ docker stack ps -f desired-state=running jenkins
ID                  NAME                IMAGE                                   NODE                DESIRED STATE       CURRENT STATE            ERROR
m3hz0396oqxm        jenkins_agent.1     vfarcic/jenkins-swarm-agent:latest      default             Running             Running 2 hours ago
izmtpz3foci0        jenkins_master.1    localhost:5000/jenkins:latest           default             Running             Running 2 hours ago
```

Εικόνα 22 Οι υπηρεσίες Jenkins master και Jenkins agent

Κατά την πλήρη παράταξη τελικά θα πρέπει να εμφανίζονται οι υπηρεσίες του Jenkins master και του agent ανά κόμβο.



Εικόνα 23 To Slack Dashboard

Μια εναλλακτική παρακολούθησης των συναγερμών του AlertManager είναι αυτή του slack, όμως δεν εξετάστηκε ενδελεχώς δεδομένης της εξάρτησης από ένα εξωτερικό μηχανισμό σε αντίθεση με την εναλλακτική του Jenkins που είναι ένας αμιγώς εσωτερικός μηχανισμός.

5.2.5.1. Η παραμετροποίηση των συναγερμών

Με την χρήση ενός YML αρχείου το web.yml το οποίο επίσης ρυθμίζει τα στοιχεία που αφορούν τους nginx εξυπηρετητές ιστού ρυθμίζονται και οι μετρικές και τα

όρια τους που δημιουργούν συναγερμούς προς το λογισμικό Jenkins. Αυτά τελικά αποτυπώνονται στον Alertmanager στην επιλογή του μενού με το όνομα Rules. Ένας πιθανός συναγερμός θα ήταν ο web_server_memlimit που ενεργοποιείται όταν σε χρήση μνήμης των υπηρεσιών web_server ξεπεράσει το όριο διαθέσιμης μνήμης κατά 0.8.

Όμως εκείνο που έχει τεθεί ως κρίσιμος παράγοντας κλιμάκωσης είναι όταν η ***irate(nginx_http_request_duration_seconds_count{host="docker_host_ip",job="web_server"}*** στο τρέχων πεντάλεπτο ξεπεράσει το 5 γεγονός που σημαίνει πως η εξυπηρέτηση των αιτημάτων αρχίζει να δυσκολεύει τότε ενεργοποιείται συναγερμός για αύξηση των στιγμιότυπων κατά ένα. Στην αντίθετη περίπτωση όπου η μετρική ***irate(nginx_http_request_duration_seconds_count{host="docker_host_ip",job="web_server"}*** βρεθεί κάτω από το 2 γεγονός που σημαίνει ότι το σύστημα υπερκαταναλώνει πόρους, στην πράξη, οπότε και δημιουργείται συναγερμός που ως στόχο έχει την μείωση των στιγμιότυπων κατά ένα. Φυσικά εννοείται πως οι συνθήκες του συναγερμού συνεχίσουν να υφίστανται τότε δημιουργούνται και επιπρόσθετοι συναγερμοί ώστε να εκτελείται περαιτέρω μεταβολή των στιγμιότυπων. Σε κάθε περίπτωση δεν μπορεί να γίνει περαιτέρω αύξηση όταν αγγίξουμε το άνω όριο των επιτρεπτών web_server ή αντίστοιχα το κάτω όριο των επιτρεπτών web_server στην μείωση. Τέλος, θα πρέπει να γίνει κατανοητό πως οι παράμετροι αυτοί των συναγερμών παγιώθηκαν με βάση το υφιστάμενο διαθέσιμο υλισμικό σύστημα. Είναι αυτονόητο σε ένα άλλο να απαιτείται αναπροσαρμογή. Όμως η γενική μεθοδολογία παραμένει η ίδια.

Rules
ALERT web_server_memlimit IF container_memory_usage_bytes{container_label_com_Docker_swarm_service_name="web_server"} / container_spec_memory_limit_bytes{container_label_com_Docker_swarm_service_name="web_server"} > 0.8

```
  FOR 5m
  LABELS {receiver="system", service="web_server"}
  ANNOTATIONS {summary="Memory of the service web_server is over 0.8"}
ALERT web_server_resptimeabovein
  IF irate(nginx_http_request_duration_seconds_count{host="docker_host_ip",job="web_server"}[5m]) > 5
  LABELS {receiver="system", scale="up", service="web_server", type="service"}
  ANNOTATIONS {summary="Response time of the service web_server is above 5"}
ALERT web_server_resptimebelowin
  IF irate(nginx_http_request_duration_seconds_count{host="docker_host_ip",job="web_server"}[5m]) < 2
  LABELS {receiver="system", scale="down", service="web_server", type="service"}
  ANNOTATIONS {summary="Response time of the service web_server is below 2"}
```

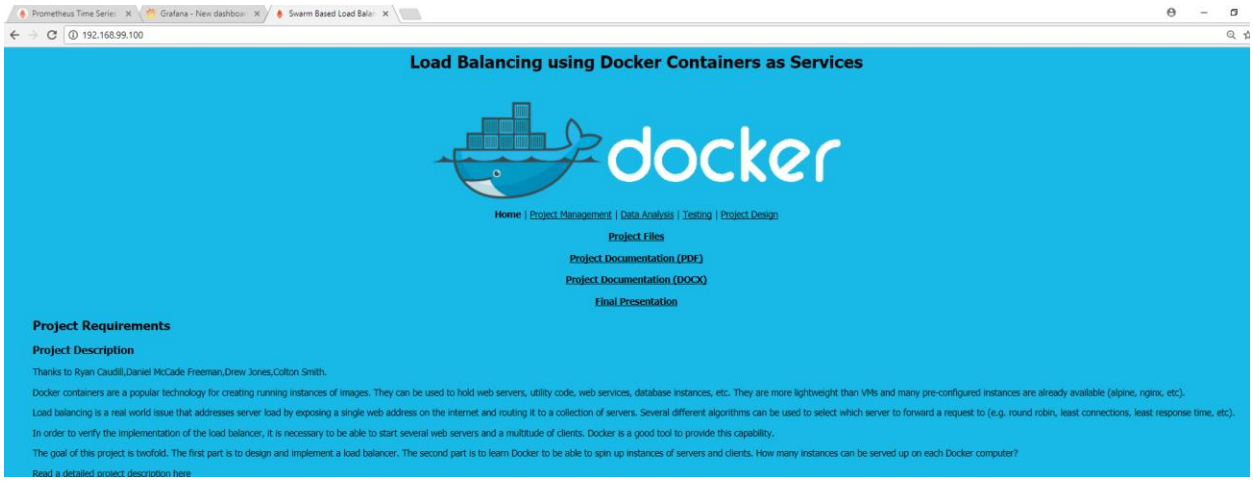
Εικόνα 24 Τα Rules των συναγερμών στον Alertmanager

5.2.6. Η εγκατάσταση της απλής εφαρμογής Web

Η εκτέλεση της εντολής:

Docker stack deploy -c web.yml web

έχει ως τελικό αποτέλεσμα την εμφάνιση ενός απλού ισοτόπου που βασίστηκε σε σχετική εργασία (Colton Smith J. D., 2017) ως μοντέλου. Μάλιστα η εμφάνιση της από την IP διεύθυνση του κόμβου δίνει μια πρώτη ένδειξη της λειτουργίας του συστήματος.



Εικόνα 25. Η εγκατάσταση της απλής εφαρμογής Web

Θα πρέπει να σημειωθεί, πως πρόκειται για μια προσαρμοσμένη έκδοση του nginx εξυπηρετητή που σκοπό έχει να παράσχει(scrape) μια σειρά μετρικών ανά στιγμιότυπο που στην κανονική έκδοση δεν υπήρχαν. Δηλαδή πρόκειται για μια εικόνα Docker με nginx και με το πρόσθετο Lua Prometheus. Οι οριζόμενες μετρικές παρέχονται στο port 9145 (Lua-Prometheus, 2017). Στο αρχείο nginx.conf περιγράφονται οι σχετικές ρυθμίσεις που δίνουν τις απαραίτητες μετρικές στο πλαίσιο της εργασίας.

```
lua_shared_dict prometheus_metrics 10M;     lua_package_path "/var/lib/lua/?.lua";

  init_by_lua '

    prometheus = require("prometheus").init("prometheus_metrics")

    metric_requests = prometheus:counter(

      "nginx_http_requests_total", "Number of HTTP requests", {"host", "status"})

       metric_request_bytes_received = prometheus:counter(

           "nginx_http_request_bytes_received", "Number of HTTP request bytes received", {"host"})
```

```
    metric_request_bytes_sent = prometheus:counter(
        "nginx_http_request_bytes_sent", "Number of HTTP request bytes sent", {"host"})
  metric_request_sizes = prometheus:counter(
    "nginx_http_request_size_bytes", "The HTTP request sizes in bytes")
    metric_latency = prometheus:histogram(
        "nginx_http_request_duration_seconds", "HTTP request latency", {"host"})
    metric_response_sizes = prometheus:histogram(
   "nginx_http_response_size_bytes", "The HTTP response sizes in bytes", {"handler"},
     {10, 100, 1000, 10000, 100000, 1000000, 10000000})
  metric_connections = prometheus:gauge(
    "nginx_http_connections", "Number of HTTP connections", {"state"})
';   log_by_lua '
  local host = ngx.var.host:gsub("^nginx.", "")
  metric_requests:inc(1, {host, ngx.var.status})
  metric_request_sizes:inc(tonumber(ngx.var.request_length))
  metric_latency:observe(ngx.now() - ngx.req.start_time(), {host})
  metric_request_bytes_sent:inc(tonumber(ngx.var.bytes_sent), {host})
  metric_request_bytes_received:inc(tonumber(ngx.var.bytes_received), {host})
  metric_response_sizes:observe(tonumber(ngx.var.bytes_sent), {host})
';
```

Εικόνα 26 Η διαμόρφωση του nginx.conf

Ο ανωτέρω πίνακας περιέχει τις Lua εντολές παραμετροποίησης και αξίζει να τονίσουμε τα εξής:

- Διαμορφώνεται ένα διαμοιραζόμενο λεξικό για τις μετρικές μας που καλείται prometheus_metrics με ένα όριο μεγέθους 10MB;
- Καταχωρεί έναν μετρητή που ονομάζεται nginx_http_requests_total με δύο ετικέτες: host και status.
- Καταχωρεί έναν μετρητή που ονομάζεται nginx_http_request_bytes_received με ετικέτα την host.
- Καταχωρεί έναν μετρητή που ονομάζεται nginx_http_request_bytes_sent με ετικέτα την host.
- Καταχωρεί έναν ιστόγραμμα που ονομάζεται nginx_http_request_duration_seconds με μια ετικέτα host.
- Καταχωρεί έναν ιστόγραμμα που ονομάζεται nginx_http_response_size_bytes με μια ετικέτα host.
- Καταχωρεί δείκτη(gauge) που ονομάζεται nginx_http_connections με μια ετικέτα την host και τον HTTP κωδικό κατάστασης ως status.

Φυσικά απαιτείται η διαμόρφωση του εκάστοτε nginx εξυπηρετητή που θα εκθέτει τις μετρικές στο Prometheus.

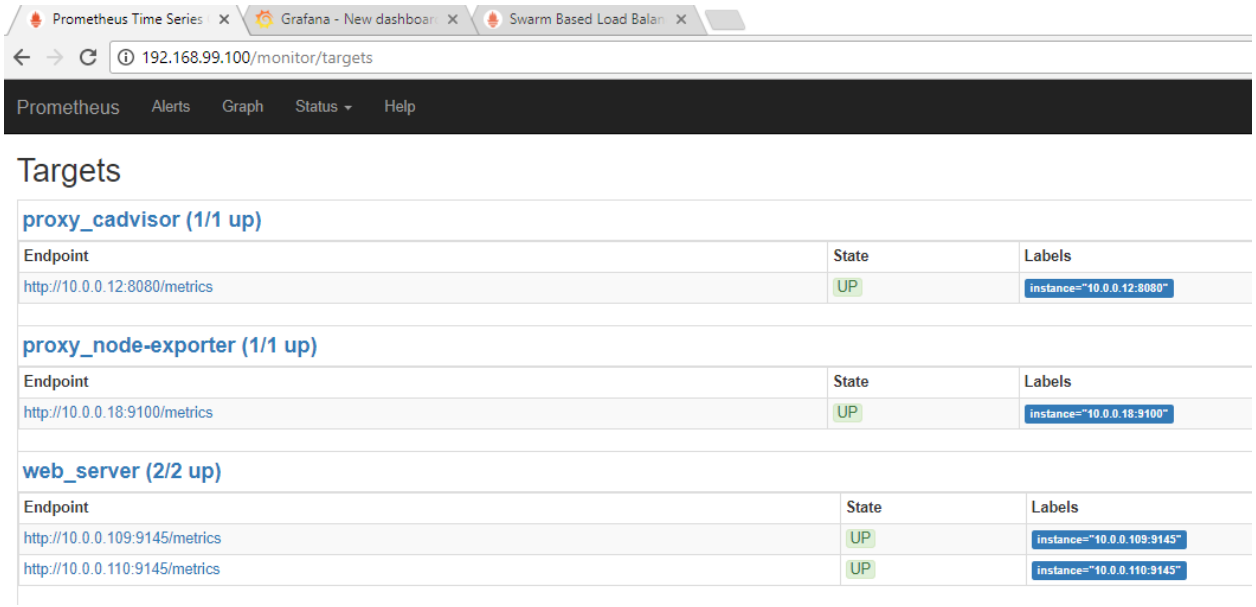
Τελικά, η πλήρης παρακολούθηση του συστήματος γίνεται από την εξέταση των κάτωθι συνδέσμων:

- *http://docker_host_ip/monitor*

για την πρόσβαση στο Prometheus γενικά

- http://docker_host_ip/monitor/targets

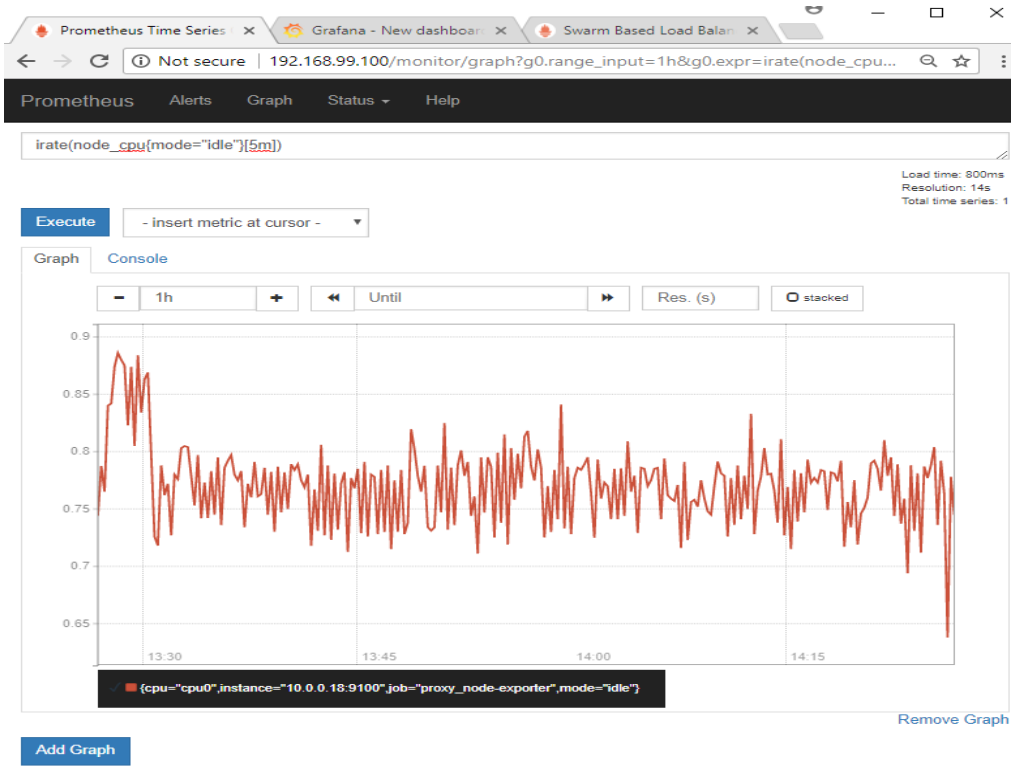
για την πρόσβαση στις παρακολουθούμενες υπηρεσίες container(targets) μπορούμε να εξετάσουμε τον ως άνω σύνδεσμο που φαίνεται και στην πιο κάτω εικόνα.



Εικόνα 27 Τα targets της εφαρμογής

- http://docker_host_ip/monitor/graph

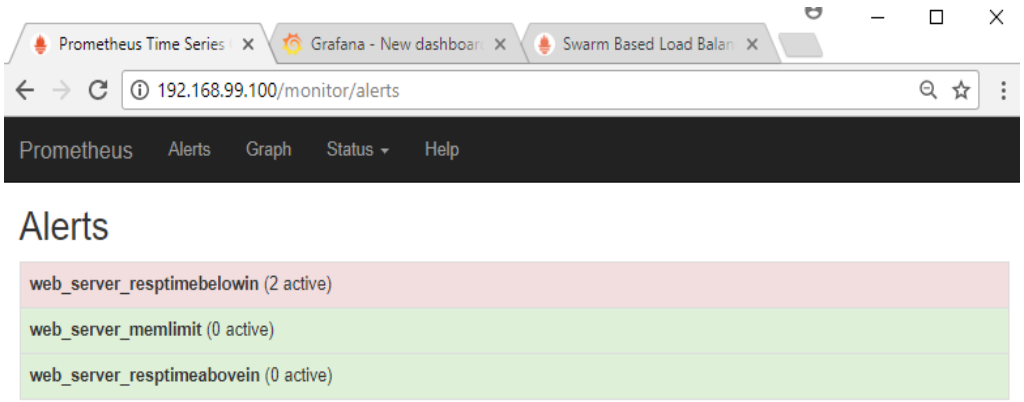
Για την πρόσβαση στο περιβάλλον εκτέλεσης ερωτημάτων είτε αριθμητικά είτε γραφικά χρησιμοποιείται ο ανωτέρω σύνδεσμος.



Εικόνα 28 Το περιβάλλον γραφικών του Prometheus

- http://docker_host_ip/monitor/alerts

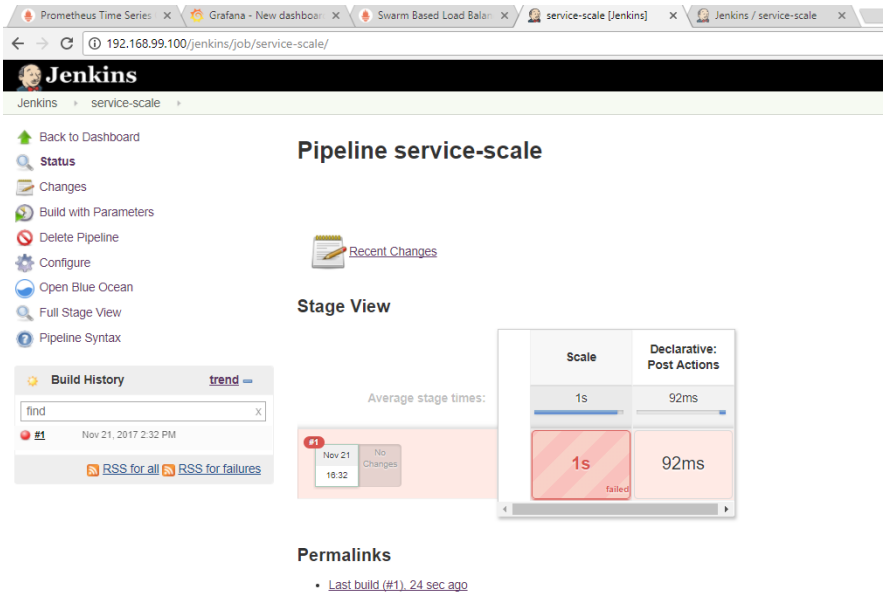
Ο σύνδεσμος αυτός παρέχει άμεση εποπτεία για την περίπτωση ενεργοποίησης συναγερμών.



Εικόνα 29 Το περιβάλλον συναγερμών του Prometheus

- http://docker_host_ip/jenkins/job/service-scale/configure

Ο πιο πάνω σύνδεσμος χρησιμοποιείται για την παρακολούθηση του αγωγού εργασιών του Jenkins. Ενώ θα πρέπει να γίνει αντιληπτό πως για να εκκινήσει ο μηχανισμός στο Jenkins θα πρέπει να ενεργοποιηθεί την πρώτη από τον διαχειριστή μέσα από την επιλογή Open Blue Ocean.

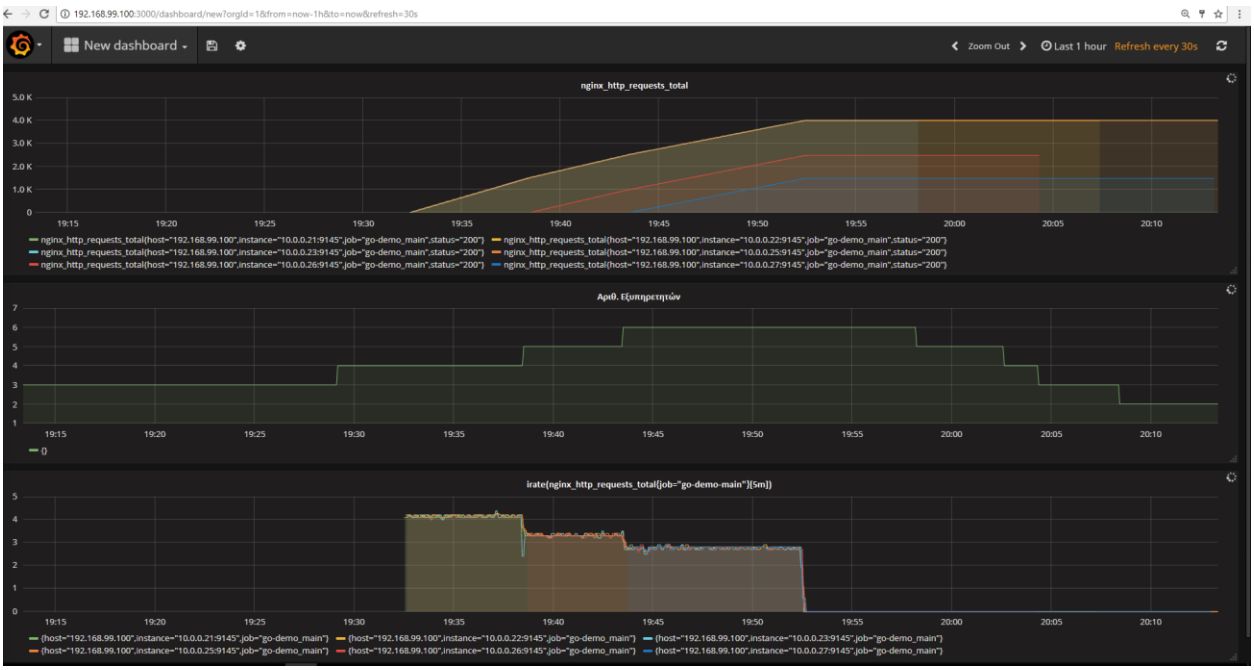


Εικόνα 30 Το περιβάλλον Jenkins

6. Επεξήγηση και Εξέταση Σεναρίων φόρτου

6.1. Αποτύπωση της Κλιμάκωσης-Αποκλιμάκωσης

Στην συνέχεια παρατίθεται ad-hoc παράδειγμα με σκοπό να επιδειχθεί αποκλειστικά η λειτουργικότητα του μηχανισμού κλιμάκωσης-αποκλιμάκσης των nginx εξυπηρετητών. Δηλαδή πραγματοποιείται μια αυτόνομη εκτέλεση του Auto-Scaling Load Balancing(ASLB) μηχανισμού. Στην περίπτωση αυτή ο βασικός αλγόριθμος είναι ο Round Robin. Θα πρέπει να σημειωθεί πως στο συγκεκριμένο παράδειγμα έχει οριστεί ως αρχική τιμή των nginx εξυπηρετητών να είναι τρεις (3). Επιτρέπεται να φτάσουν τους έξι (6) κατά μέγιστο και κατά δύο (2) το ελάχιστο.



Εικόνα 31 Παράδειγμα κλιμάκωσης και αποκλιμάκωσης

Παρατηρούμε πως το σύστημα ξεκινά με τρείς(3) nginx εξυπηρετητές και δέχεται 20000 κλείσεις σε 1200 δευτερόλεπτα. Για την εκτέλεση της κλιμάκωσης και αποκλιμάκωσης του ASLB συστήματος χρησιμοποιήθηκε η υπολογιζόμενη μετρική:

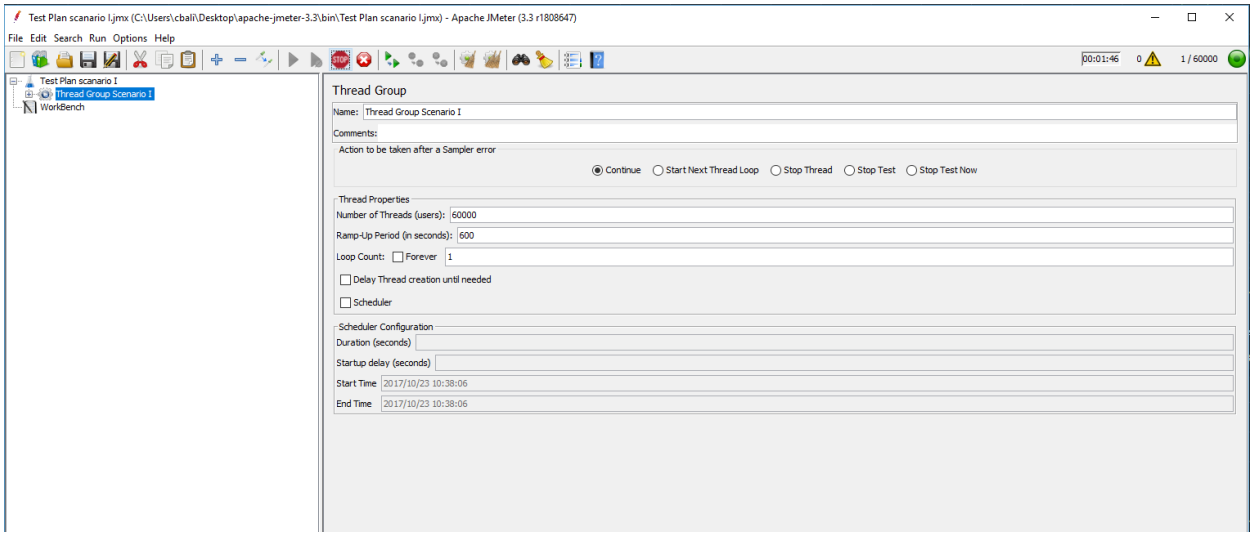
irate(nginx_http_requests_total{host="docker_host_ip",job="web-server",status="200"})

Βασίζεται στην nginx_http_requests_total που μας δίνει το πρόσθετο prometheus lua, όπως αναφέρθηκε νωρίτερα, και την irate(σε range-vector) συνάρτηση ερωτήματος του Prometheus που μας δίνει το ανά δευτερόλεπτο στιγμιαίο ρυθμό αύξησης της χρονοσειράς αιτημάτων HTTP σε ένα διάστημα εύρους πέντε λεπτών. Δηλαδή επιστρέφει **τον τρέχων ρυθμό ανά δευτερόλεπτο των αιτημάτων HTTP** για τις δύο πιο πρόσφατες καταγραφές μέσα σε ένα εύρος 5 λεπτών. Το εύρος θα μπορούσε να είναι μικρότερο έως του σημείο του ρυθμού δειγματοληψίας(14s). Έτσι όταν ο nginx εξυπηρετητής δέχεται ρυθμό αύξησης πάνω από 5 αιτήματα ανά δευτερόλεπτο ενεργοποιείται ο μηχανισμός κλιμάκωσης αυξάνοντας κατά ένα τους nginx εξυπηρετητές. Αντίστοιχα όταν ένας nginx εξυπηρετητής δέχεται ρυθμό κάτω από 2 αιτήματα ανά δευτερόλεπτο ενεργοποιείται ο μηχανισμός αποκλιμάκωσης κατά ένα εξυπηρετητή. Στο παραπάνω στιγμιότυπο παρατηρούμε πως όσο δέχεται κίνηση το σύστημα έχουμε συνεχή κλιμάκωση κατά ένα εξυπηρετητή. Ενώ μόλις σταματήσει η κίνηση και αποσταλούν τα σχετικά alerts αρχίζει η σταδιακή αποκλιμάκωση κατά ένα εξυπηρετητή. Φυσικά η λειτουργία της κλιμάκωσης και αποκλιμάκωσης σταματά μόλις αγγίξουμε το μέγιστο και το ελάχιστο όριο αντίστοιχα. Για την καλύτερη αποτίμηση του συστήματος θα πρέπει να εξεταστούν τα σενάρια που ακολουθούν.

6.2. Σενάριο Ι: Συνθήκες ισορροπίας

Σ' αυτό το σενάριο οι κλήσεις των HTTP πελατών ανακτούν αντικείμενο 14KB από την συστοιχία των εξυπηρετητών ιστού. Αυτή η λειτουργία πραγματοποιείται στα 60000 HTTP requests ανά 600 δευτερόλεπτα δηλαδή για 10 min. Η δημιουργία

φόρτου γίνεται με το εργαλείο JMeter. Βασική παράμετρος εδώ είναι το γεγονός ότι οι εξυπηρετητές δεν επηρεάζονται από αστάθμητους παράγοντες και παραμένουν αφοσιωμένοι στην εξυπηρέτηση της κίνησης.



Εικόνα 32 Οι ρυθμίσεις του JMeter για την διεξαγωγή των σεναρίων

Σ.Ι.α.: Κεντρικό σημείο είναι ότι εδώ ο εξυπηρετητής φορτίου HAproxy έχει τεθεί να λειτουργήσει σε RR(Round-Robin).

i) Αρχικά εκτελείται το πείραμα χωρίς να παρέχεται στο σύστημα η δυνατότητα να κλιμακώσει την nginx υπηρεσία ή να την αποκλιμακώσει αντίστοιχα.

ii) Στην συνέχεια επαναλαμβάνεται το ίδιο πείραμα με ενεργοποίηση της δυνατότητας κλιμάκωσης και αποκλιμάκωσης.

Έτσι στην υποκατηγορία αυτή θα εξεταστεί η συμπεριφορά των αλγόριθμων δηλαδή ο RR(Round-Robin), και η προσαρμογή του ASLB-RR.

Σ.Ι.β.: Κεντρικό σημείο είναι ότι εδώ ο εξυπηρετητής φορτίου HAproxy έχει τεθεί να λειτουργήσει σε LC(least connections). Έτσι οι αλγόριθμοι που εξετάζονται είναι ο LC(least connections), και η προσαρμογή του ASLB-LC

i) Αρχικά, στο πλαίσιο αυτό εκτελείται το πείραμα χωρίς να παρέχεται στο σύστημα η δυνατότητα της κλιμάκωσης της nginx υπηρεσίας ή της αποκλιμάκωσης αντίστοιχα.

ii) Στην συνέχεια πραγματοποιείται το ίδιο πείραμα με ενεργοποίηση της δυνατότητας κλιμάκωσης και αποκλιμάκωσης.

Όπως είναι εύλογο ο σκοπός αυτού του σεναρίου είναι να εξεταστεί ο στοιχειώδης RR αλγόριθμος χωρίς κατάσταση, ο αλγόριθμος LC με καταστάσεις και η προσαρμογή ASLB στις αντίστοιχες περιπτώσεις.

6.3. Σενάριο ΙΙ: Συνθήκες ανισορροπίας

Σ' αυτό το σενάριο οι πόροι ενός εξυπηρετητή θα εξαντληθούν ή θα υπερφορτωθούν ενώ οι HTTP clients θα ανακτούν αντικείμενα των 14 KB από την συστοιχία των web server. Αυτή η λειτουργία πραγματοποιείται στα 60000 HTTP requests ανά 600 δευτερόλεπτα δηλαδή για 10 min.

Σ.ΙΙ.α.: οι αλγόριθμοι που εξετάζονται είναι ο RR(Round-Robin), και η προσαρμογή του ASLB-RR

i) Αρχικά εκτελείται το πείραμα με τον HAproxy σε RR(Round-Robin) και σε συνθήκες ανισορροπίας χωρίς να παρέχεται στο σύστημα η δυνατότητα να κλιμακώσει την nginx υπηρεσία ή να αποκλιμακώσει αντίστοιχα.

ii) Στην συνέχεια επαναλαμβάνεται το ίδιο πείραμα με ενεργοποίηση της δυνατότητας κλιμάκωσης και αποκλιμάκωσης.

Σ.ΙΙ.β.:οι αλγόριθμοι που εξετάζονται είναι ο LC(least connections), και η προσαρμογή του ASLB-LC

i) Σε πρώτη φάση εκτελείται το πείραμα με τον HAproxy σε LC(least connections) και σε συνθήκες ανισορροπίας χωρίς να παρέχεται στο

σύστημα η δυνατότητα να κλιμακώσει ή να αποκλιμακώσει αντίστοιχα την nginx υπηρεσία.

ii) Στην συνέχεια επαναλαμβάνεται το ίδιο πείραμα με ενεργοποίηση της δυνατότητας κλιμάκωσης και αποκλιμάκωσης.

Ο σκοπός αυτού του σεναρίου είναι να εξεταστεί συγκριτικά η απόδοση του στοιχειώδους RR (αλγόριθμος χωρίς κατάσταση), του αλγόριθμος LC(με καταστάσεις) και της αντίστοιχης προσαρμογής ASBL κάτω από συνθήκες ανισορροπίας εξαιτίας της υπερφόρτωσης ενός μέρους των αρχικών web servers π.χ. έναν από τους δύο. Για την επίτευξη αυτού του σεναρίου θα πρέπει να εκτελεστεί ένα, κοστοβόρο από άποψη πόρων, script στους προαναφερόμενους web server(πχ 5 φορές η εντολή if=/dev/zero of=/dev/null ανά server).

Εικόνα 33 Παράδειγμα λειτουργίας του συστήματος σε συνθήκες ανισορροπίας

Για παράδειγμα, στο πιο πάνω διάγραμμα, βλέπουμε πως ο ένας από τους δύο εξυπηρετητές (με πράσινο χρώμα) καταπονείται από ένα κοστοβόρο scirpt. Εντούτοις το Docker Swarm φροντίζει ο Αλγόριθμος Round-Robin να εκτελείται

κανονικά και να εξισορροπείται το φορτίο.

7. Παρουσίαση-Ανάλυση των Αποτελεσμάτων

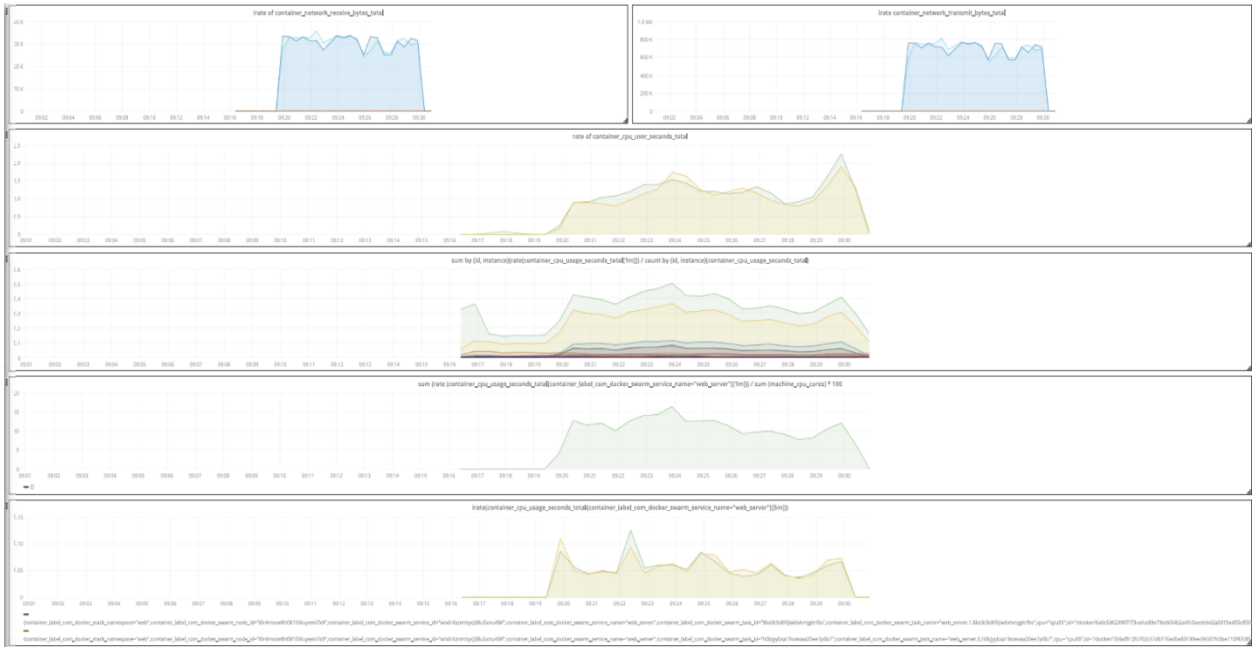
Στην συνέχεια παρουσιάζονται τα διαγράμματα που πρόεκυψαν από το Grafana με βάση τα σενάρια που αναφέρθηκαν και τις τεχνικές λεπτομέρειες που διερευνήθηκαν στο κεφάλαιο περιγραφής της μεθοδολογίας της εργασίας.

7.1. Σ.Ι.α: HAproxy σε RR(Round-Robin)

Σε αυτή την κατηγορία σεναρίων ο εξυπηρετητής φορτίου HAproxy έχει τεθεί να λειτουργήσει σε RR(Round-Robin).

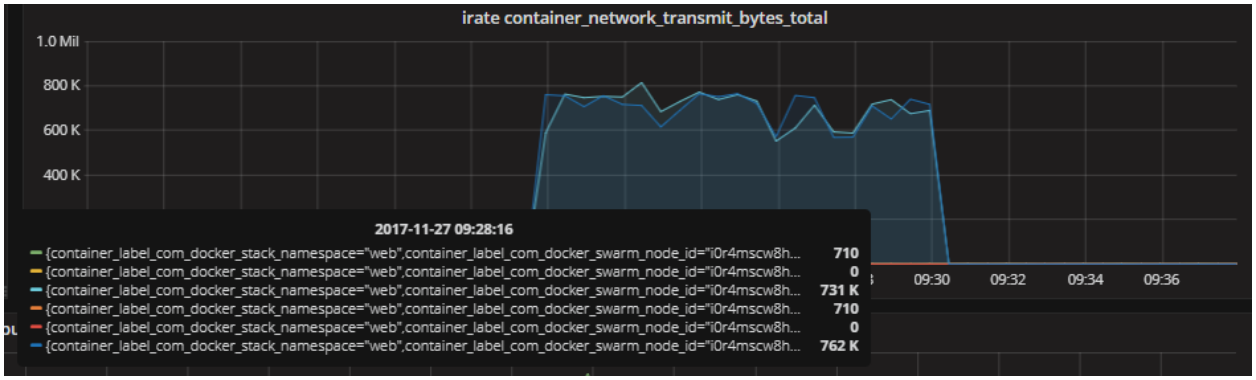
7.1.1. Σ.Ι.α.i Χωρίς την εφαρμογή ASBL

Στα παρακάτω διαγράμματα παρατηρούμε την συμπεριφορά του συστήματος όταν έχει ρυθμιστεί να λειτουργεί χωρίς την δυνατότητα ASBL.

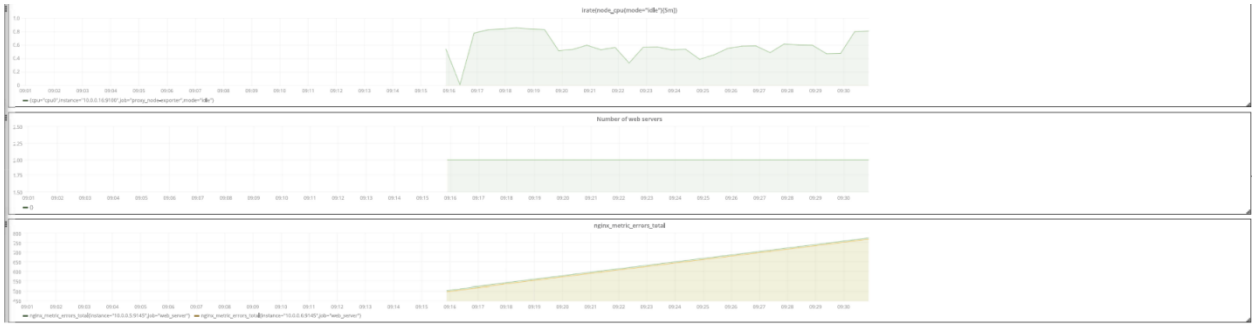


Εικόνα 34 Σ.Ι.α.i Χωρίς την εφαρμογή ASBL 1ο μέρος

Αυτό που μπορεί να παρατηρήσει κανείς στο ανωτέρω διάγραμμα είναι πως οι δύο εξυπηρετητές λειτουργούν (λαμβάνουν και αποστέλλουν) αντίστοιχο αριθμό bytes. Επίσης στη πιο κάτω εικόνα βλέπουμε ποσό ισορροπημένη είναι η αναλογία της μετρικής αποστολής bytes ανάμεσα στους δύο εξυπηρετητές. Για παράδειγμα και οι δύο έχουν εξυπηρετήσει της ίδια τάξης μεγέθους ποσότητα bytes δηλαδή 731 K έναντι 762K όπως φαίνεται στην συνέχεια.



Εικόνα 35 Σ.Ι.α.i Χωρίς την εφαρμογή ASBL 2ο μέρος



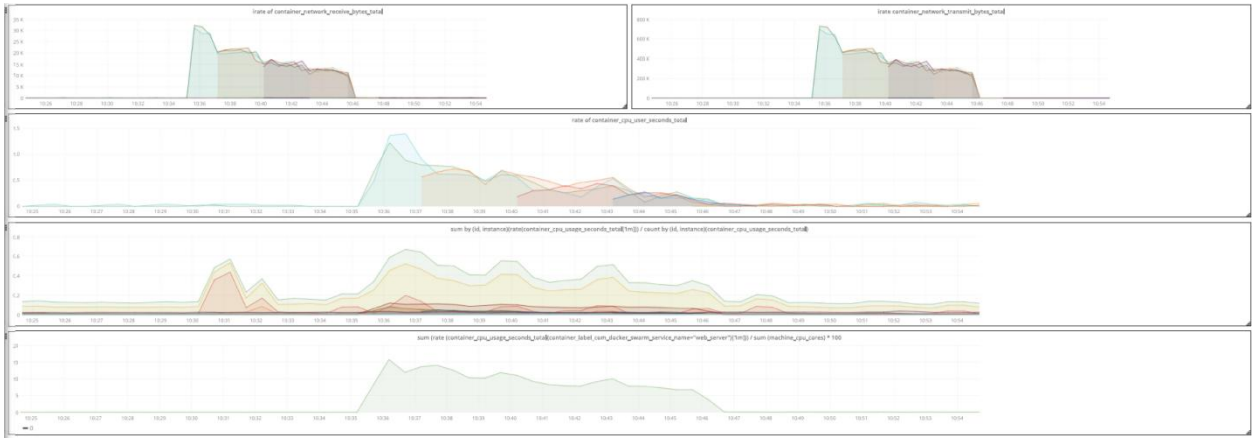
Εικόνα 36 Σ.Ι.α.i Χωρίς την εφαρμογή ASBL 3ο μέρος

Ενώ φαίνεται πως, καθ' όλη την διάρκεια της δοκιμής, υπήρχαν σταθερά δύο διαθέσιμες nginx υπηρεσίες για την εξυπηρέτηση του συστήματος, ταυτόχρονα η κατανάλωση της CPU δείχνει πως υπάρχει ένα διαθέσιμο επιπλέον περιθώριο χρήσης. Το γεγονός αυτό καταδεικνύει ότι για τον υφιστάμενο φόρτο HTTP αιτημάτων της δοκιμής, η χρήση δύο εξυπηρετητών ιστού επιτρέπει την ανεκτή

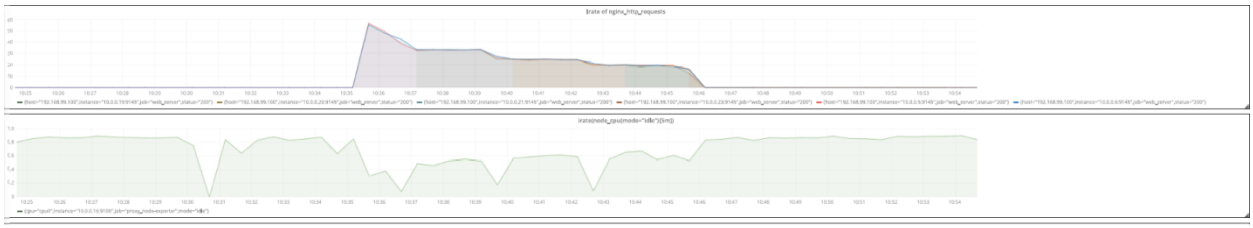
λειτουργία του συστήματος. Και αυτό γιατί η διαθέσιμη CPU κινείται οριακά κάτω από το 60%. Ειδικά κατά το μέσο της δοκιμής για ένα μικρό διάστημα βρίσκεται και κάτω από το 40%. Από την εξέταση συνδυαστικά των πιο πάνω διαγραμμάτων φαίνεται πως υπάρχει επιπλέον περιθώριο για CPU ώστε να εξυπηρετηθεί επιπλέον φόρτος. Το πιο αξιοπρόσεκτο, όμως, είναι πως η αναλογία της http_nginx_requests μετρικής παραμένει ουσιαστικά ανεπηρέαστη. Διαπιστώνεται επίσης πως ο αριθμός των εξυπηρετητών παραμένει σταθερός(δύο).

7.1.2. Σ.Ι.α.ii Με την εφαρμογή ASBL

Στην συνέχεια ενεργοποιήθηκε το Jenkins ώστε να αντιδρά στους προκαθορισμένους συναγερμούς και να εκτελεί build εργασίες. Θα μπορούσε κανείς να παρατηρήσει(στο διάγραμμα της 3η γραμμής, την κυματομορφή με το έντονο πορτοκαλί χρώμα) το πόσο κοστοβόρο είναι το πρώτο build που εκτελείται από τον διαχειριστή και ουσιαστικά θέτει σε λειτουργία την pipeline. Επίσης στην συνέχεια όποτε έχουμε αλλαγή κατάστασης(build) εμφανίζονται μικρότερες πορτοκαλί ακίδες. Επομένως φαίνεται πως κάθε φορά που εμφανίζεται μια τέτοια ακίδα έχουμε μεταβολή(κλιμάκωση) στον αριθμό των nginx εξυπηρετητών(αν παρατηρήσουμε το τελευταίο διάγραμμα της παραγράφου αυτής). Αυτό συνάμα δείχνει και την επιβάρυνση του Jenkins στο συνολικό σύστημα.



Εικόνα 37 Σ.Ι.α.ii Με την εφαρμογή ASBL 1ο μέρος



Εικόνα 38 Σ.Ι.α.ii Με την εφαρμογή ASBL 2ο μέρος

Εξάλλου από τη συγκριτική παρατήρηση των ανωτέρω βλέπουμε πως οι μετρικές που σχετίζονται με την εξυπηρέτηση των HTTP αιτημάτων ακολουθούν μια μορφή αποκλιμάκωσης γεγονός που έχει λογική, καθώς περισσότερες συνδέσεις μπορούν να εξυπηρετούνται ταυτόχρονα όταν αυξάνονται οι nginx υπηρεσίες εξυπηρετητή.

Εικόνα 39 Σ.Ι.α.ii Με την εφαρμογή ASBL 3ο μέρος

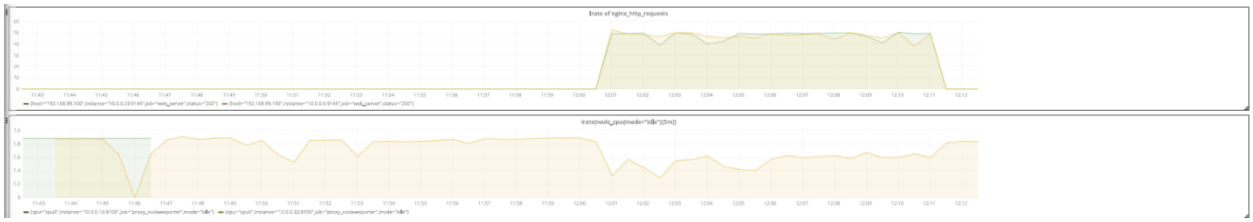
Στο διάγραμμα αυτό φαίνεται το ιστορικό μεταβολής των στιγμιότυπων nginx υπηρεσιών εξυπηρετητή. Δηλαδή, ενώ ξεκινά από τις ελάχιστες δύο, αυξάνει διαδοχικά μέχρι τον μέγιστο των έξι. Στο μεταξύ όμως, έχει τελειώσει το δεκάλεπτο του φόρτου από το JMeter οπότε και σταδιακά μειώνονται τα στιγμιότυπα nginx υπηρεσίας εξυπηρετητή καθώς δεν υπάρχει λόγος να υφίσταται αυξημένος αριθμός τέτοιων εξυπηρετητών. Στο υπόβαθρο βέβαια, αυτό γίνεται γιατί το Jenkins λαμβάνει συναγερμούς που το εντέλουν να εκτελέσει την εντολή αποκλιμάκωσης.

7.2. Σ.Ι.β. HAproxy σε LC(Least Connections)

7.2.1. Σ.Ι.β.i Χωρίς την εφαρμογή ASBL

Εικόνα 40 Σ.Ι.β.i Χωρίς την εφαρμογή ASBL 1ο μέρος

Στην περίπτωση αυτή, παρατηρούμε πως ως προς την αποστολή και λήψη bytes κατά τα αρχικά στάδια είναι οριακά διαφοροποιημένα σε σχέση με την αντίστοιχη περίπτωση σε Round Robin. Ενώ κατά τα άλλα δεν παρουσιάζει διαφορά από αυτή.

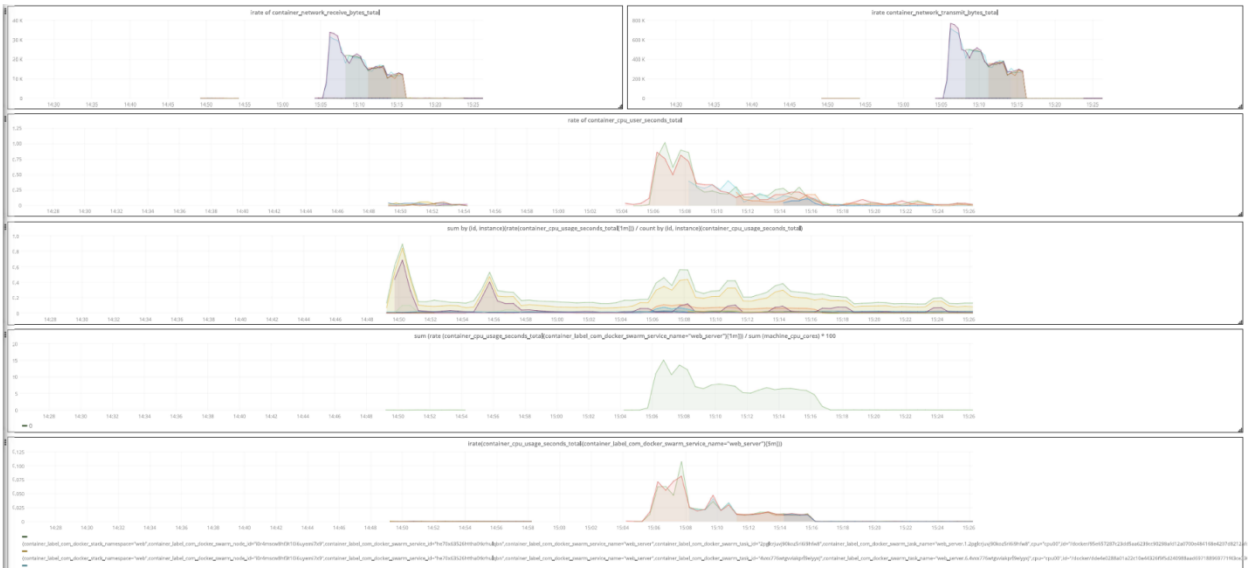


Εικόνα 41 Σ.Ι.β.i Χωρίς την εφαρμογή ASBL 2ο μέρος

7.2.2. Σ.Ι.β.ii Με την εφαρμογή ASBL

Στην συνέχεια παρουσιάζονται τα διαγράμματα που προκύπτουν από την εφαρμογή του ASBL όταν ο HAproxy έχει ρυθμιστεί σε Least connections. Είναι επίσης σαφές πως οι μετρικές που σχετίζονται με τα HTTP αιτήματα

αποκλιμακώνονται όπως ακριβώς είδαμε και στην περίπτωση του Round Robin. Όμως παρατηρούμε και μια ελαφρά ανισοκατανομή στην αποστολή και λήψη bytes.



Εικόνα 42 Σ.Ι.β.ii Με την εφαρμογή ASBL 1ο μέρος

Στο ανωτέρω διάγραμμα δεν θα μας απασχολήσουν οι δύο πρώτες ακίδες του διαγράμματος της 3ης σειράς καθώς πρόεκυψαν από τις αναγκαίες ρυθμίσεις του συστήματος ώστε να ανταποκριθεί στις ανάγκες αυτού του σεναρίου.

Εικόνα 43 Σ.Ι.β.ii Με την εφαρμογή ASBL 2ο μέρος

Παράλληλα παρατηρείται στο κάτωθι διάγραμμα(1η γραμμή) μια αφαίμαξη πόρων κατά την στιγμή που εκτελείται η διαδικασία κλιμάκωσης κατά έναν επιπλέον nginx εξυπηρετητή.

Εικόνα 44 Σ.Ι.β.ii Με την εφαρμογή ASBL 3ο μέρος

Εξάλλου παρατηρούμε πως οι nginx εξυπηρετητές φτάνουν μέχρι το μέγιστο(6) όμως επειδή ολοκληρώθηκε στο μεταξύ το χρονικό όριο του σεναρίου, αρχίζει η σταδιακή αποκλιμάκωση.

7.3. Σ.ΙΙ.α. HAproxy σε RR(Round-Robin) με διαταραχή

Εδώ η συνθήκη ανισορροπίας εφαρμόζεται 50% των εξυπηρετητών π.χ. 1 στους δύο αρχικούς.

7.3.1. Σ.ΙΙ.α.i: Χωρίς την εφαρμογή ASBL

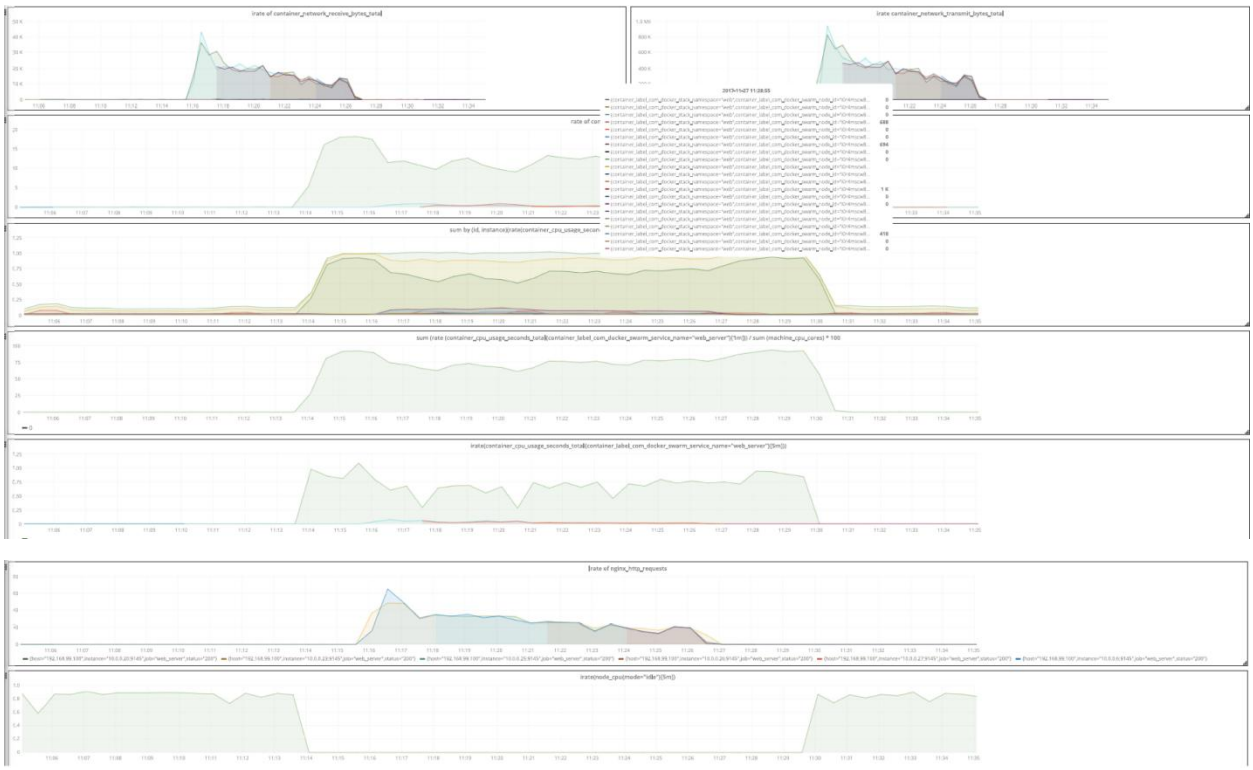
Ειδικότερα, εφαρμόστηκε η ανωτέρω διαταραχή στον nginx εξυπηρετητή με διεύθυνση IP 10.0.0.6 ενεργοποιώντας το προαναφερόμενο script κατανάλωσης πόρων στο αντίστοιχο στιγμιότυπο.

Εικόνα 45 Σ.Ι.α.ii Χωρίς την εφαρμογή ASBL

Αν και δεν αποτυπώνεται σ' αυτή την συγκριτική εξέταση των μετρικών αυτών, ο αριθμός των εξυπηρετητών ιστού παραμένει σταθερός, καθ' όλη την διάρκεια του σεναρίου. Επίσης είναι αξιοσημείωτο πως όσο διαρκεί το κοστοβόρο από άποψη πόρων script (5 φορές η εντολή if=/dev/zero of=/dev/null) η CPU του κόμβου εξαντλεί όλα τα περιθώρια διαθεσιμότητας. Επίσης στο διάγραμμα της έκτης σειράς φαίνεται πόσο καταπονείται ο container που υφίσταται την εκτέλεση του script. Το αξιοσημείωτο είναι πως όταν αρχίζει η αποστολή φόρτου από το JMeter, το σύστημα αναπροσαρμόζει την χρήση του χρόνου CPU ώστε να μπορέσει να ανταποκριθεί και στην κύρια λειτουργία του που είναι η εξυπηρέτηση του φόρτου. Φυσικά τα καταφέρνει με μικρή σχετικά επιτυχία καθώς υπάρχουν στιγμές που τα αιτήματα προς τον ομαλό εξυπηρετητή ξεπερνούν τον καταπονημένο κατά 40% περίπου.

7.3.2. Σ.ΙΙ.α.ii: Με την εφαρμογή ASBL

Είναι εύλογο επομένως να εξετάσει κανείς την συμπεριφορά του συστήματος με την εφαρμογή της αυτοματοποιημένης κλιμάκωσης μέσω του Jenkins. Και στην περίπτωση αυτή εφαρμόστηκε η ανωτέρω διαταραχή στον nginx εξυπηρετητή με διεύθυνση IP 10.0.0.6

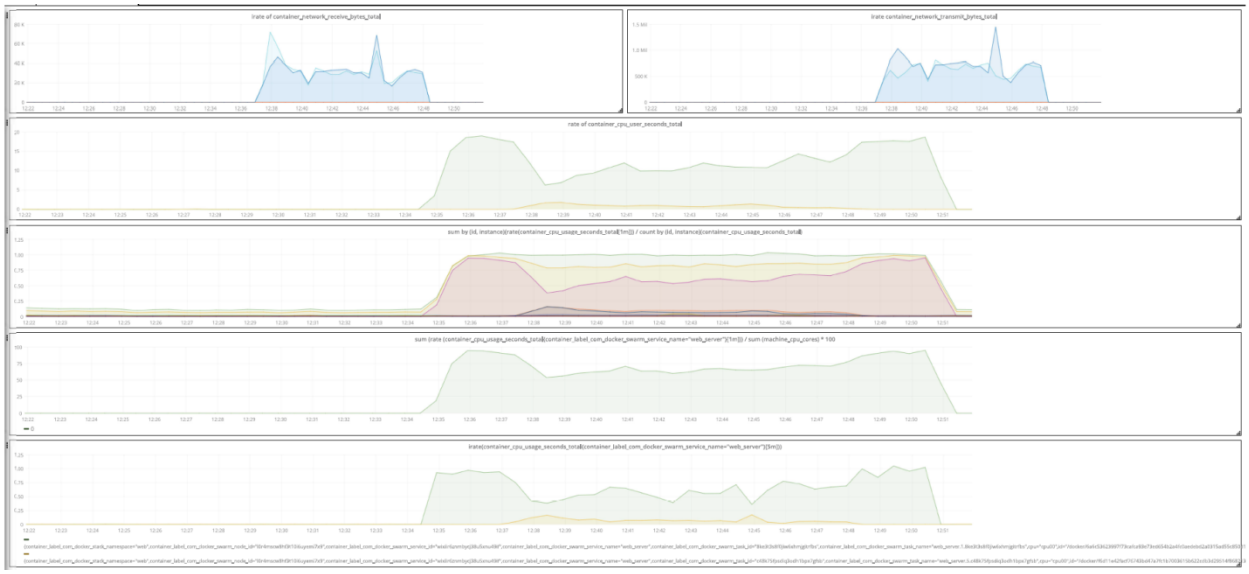


Εικόνα 46 Σ.Ι.α.ii Με εφαρμογή ASBL

Αναφορικά, με τις παρατήσεις ισχύει ότι ειπώθηκαν στην προηγούμενη παράγραφο όσο δεν έχει ενεργοποιηθεί η κλιμάκωση. Αμέσως όμως μόλις ενεργοποιηθεί ένας εξυπηρετητής αρχίζει ο καταπονημένος εξυπηρετητής να υστερεί λιγότερο σε σχέση με το στατικό ισοδύναμο σενάριο. Μάλιστα μόλις ενεργοποιηθεί ακόμα ένας nginx εξυπηρετητής, όποτε και ο φόρτος που καλείται να εξυπηρετήσει ο καταπονημένος εξυπηρετητής μειώνεται σημαντικά, παρατηρούμε πως η κατάσταση ομαλοποιείται. Επομένως η λειτουργία αυτοματοποιημένης κλιμάκωσης μέσω του Jenkins(ASLB) οδηγεί και στην εξομάλυνση των εσωτερικών ανισορροπιών. Το τελευταίο δεν θα μπορούσε να θεωρηθεί εύλογο στην πράξη, προ της αποτύπωσης του στο εξεταζόμενο σενάριο.

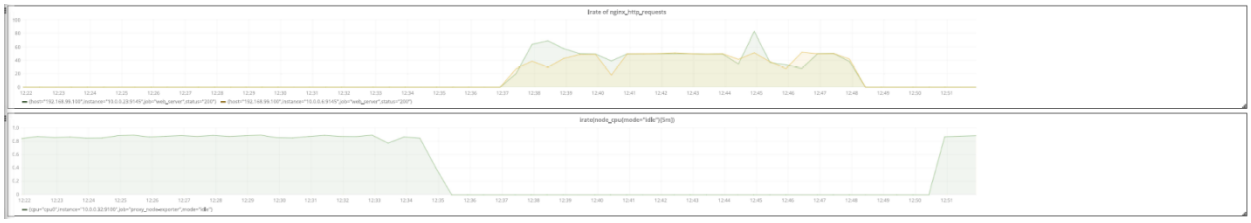
7.4. Σ.ΙΙ.β. HAproxy σε LC(Least Connections) με διαταραχή

7.4.1. Σ.ΙΙ.β.i: Χωρίς την εφαρμογή ASBL



Εικόνα 47 Χωρίς την εφαρμογή ASBL1ο μέρος

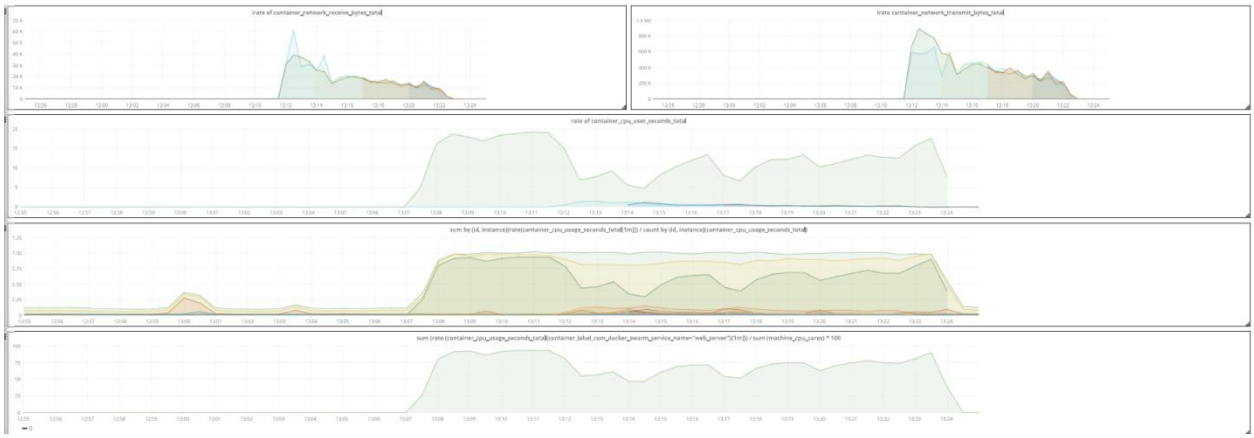
Στην περίπτωση του HAproxy σε LC(Least Connections) με διαταραχή χωρίς ASBL παρατηρούμε πως ο εξυπηρετητής που υφίσταται καταπόνηση δεν μπορεί κυρίως να ακολουθήσει τις απότομες μεταβολές στην ροή των δεδομένων αυτό. Το γεγονός που ταυτίζεται με την λογική της εξυπηρέτησης αυτού του αλγορίθμου που βασίζεται σε καταστάσεις.



Εικόνα 48 Χωρίς την εφαρμογή ASBL 2ο μέρος

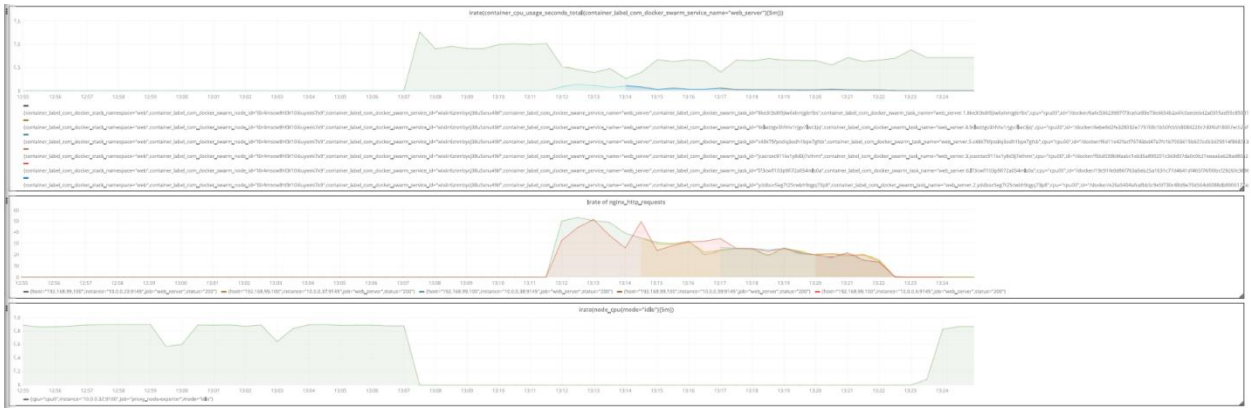
Πάντως θα λέγαμε πως στο μέσο του σεναρίου αποκαθίσταται η ισορροπία. Ενώ και πάλι έχουμε πλήρη κατανάλωση της διαθέσιμης CPU.

7.4.2. Σ.ΙΙ.β.ii: Με την εφαρμογή ASBL



Εικόνα 49 Με την εφαρμογή ASBL 1ο μέρος

Στην περίπτωση του HAproxy σε LC(Least Connections) με διαταραχή και εφαρμογή ASBL παρατηρούμε πως ο εξυπηρετητής που υφίσταται καταπόνηση αρχικά δεν μπορεί να ανταποκριθεί στην εξυπηρέτηση του φόρτου ενώ η σταδιακή κλιμάκωση των στιγμιότυπων επιτρέπει στον καταπονημένο εξυπηρετητή να παρακολουθήσει το σύστημα, κυρίως μετά από το δεύτερο επιπλέον nginx εξυπηρετητή.



Εικόνα 50 Με την εφαρμογή ASBL 2ο μέρος

Και πάλι έχουμε πλήρη απορρόφηση των διαθέσιμων πόρων.

Εικόνα 51 Με την εφαρμογή ASBL 3ο μέρος

Το αξιοσημείωτο εδώ είναι πως το ASBL δρα ευεργετικά σε τέτοιο βαθμό που δεν απαιτείται περαιτέρω κλιμάκωση πάνω από το πέμπτο nginx εξυπηρετητή. Δηλαδή βλέπουμε πως το σενάριο ολοκληρώνεται πριν προλάβει το σύστημα να κλιμακωθεί στο μέγιστο δυνατό αριθμό στιγμιότυπων. Ενώ η διακοπή της διαταραχής επαναφέρει την χρήση της CPU στα επίπεδα της αδράνειας(idle mode) μετά το τέλος του σεναρίου.

8. Συμπεράσματα και μελλοντικές επεκτάσεις

8.1. Συμπεράσματα

Από την συνδυαζόμενη εξέταση των διαγραμμάτων των μετρικών προέκυψαν μερικά χρήσιμα συμπεράσματα από την χρήση του ASBL σε εφαρμογές υπολογιστικού νέφους. Πρωταρχικά, έχει προκύψει πως *το Docker Swarm ακόμη και σε συνθήκες έντονης διαταραχής και ανισορροπίας των στιγμιότυπων μιας υπηρεσίας καταβάλει κάθε δυνατή προσπάθεια να κατανείμει τον φόρτο των HTTP αιτημάτων όσο το δυνατόν ομοιόμορφα.*

Αναλυτικότερα από την εξέταση των :

- sum (rate (container_cpu_usage_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[1m])) / sum (machine_cpu_cores) * 100
- (irate(container_cpu_usage_seconds_total{container_label_com_Docker_swarm_service_name="web_server"}[5m]))

Δηλαδή των παράγωγων μετρικών της container_cpu_usage_seconds_total στο εργαλείο Prometheteus που αντιστοιχούν στο 3ο και το 4ο κατά σειρά διάγραμμα των σεναρίων κάτω από συνθήκες διαταραχής, παρατηρούμε πόσος χρόνος διατίθεται από τον συνολικό διαθέσιμο σε κάθε στιγμιότυπο. Με άλλα λόγια, ο χρόνος της CPU διατίθεται στην συντριπτική του πλειοψηφία στο στιγμιότυπο της υπηρεσίας που υφίσταται την καταπόνηση. Αυτό έχει σαν αποτέλεσμα την απορρόφηση της διαθέσιμης CPU του κόμβου στο 100%. Εντούτοις παρατηρούμε μια ικανοποιητική υποδοχή HTTP αιτημάτων. Φυσικά είναι ομαλότερη στην περίπτωση του Round-Robin. Στην περίπτωση του Least Connection όπως αναμενόταν βλέπουμε πως αν και δεν κατανέμεται ο ίδιος φόρτος εντούτοις το στιγμιότυπο συμπεριφέρεται καλύτερα από ότι θα αναμένονταν ex-ante διαισθητικά.

Βέβαια, όταν αρχίζουν να προστίθενται στιγμιότυπα υπηρεσίας η κατάσταση βελτιώνεται. Ειδικότερα όταν ο αριθμός των στιγμιότυπων διπλασιαστεί τότε πλέον η κατάσταση έχει ομαλοποιηθεί. Επίσης, παρατηρείται πως μετά από τον διπλασιασμό αυτών, ας τον ονομάσουμε παράγοντα 2, έτι περαιτέρω αύξηση των στιγμιότυπων δεν συνεισφέρει ουσιαστικά στην βελτίωση της κατάστασης. Δηλαδή ως παράγοντα 2 θα ονομάζαμε την πειραματική παρατήρηση πως όταν ένας εκ των δύο στιγμιότυπων βρίσκεται σε κατάσταση καταπόνησης(συνθήκη ανισορροπίας), εφόσον εφαρμοστεί ASLB, από την στιγμή που θα διπλασιαστεί ο αριθμός των στιγμιότυπων τότε ομαλοποιείται η λειτουργία του. Πρακτικά αυτό συμβαίνει γιατί πλέον ο φόρτος που καλείται να διαχειριστεί πέφτει σε επίπεδα που του το επιτρέπουν οι διαθέσιμοι πόροι του.

Φυσικά θα πρέπει να αναφερθεί πως η πίεση που υφίσταται ο κόμβος είναι καταφανής τόσο από τις μετρικές όσο και από το γεγονός πως η διαθέσιμη CPU του κόμβου απορροφάται στο 100%. Εντούτοις το σύστημα εξυπηρετεί το φόρτο. Η κατάσταση αυτή σε ένα φυσικό σύστημα θα οδηγούσε σε μη διαθεσιμότητα. Αντίθετα ένα περιβάλλον Docker Swarm σε Cloud δύναται να αναπροσαρμόζεται πολύ εύκολα εξασφαλίζοντας στην πράξη την επεκτασιμότητα μιας εφαρμογής.

8.2. Μελλοντικές επεκτάσεις

Με βάση τα ανωτέρω συμπεράσματα, θα ήταν εξαιρετικά ενδιαφέρον να διαπιστωθεί αν ο παράγοντας 2 αποτελεί ιδιότητα της συγκεκριμένης διαμόρφωσης η αποτελεί ένα γενικό κανόνα που θα μπορούσε να μοντελοποιηθεί. Με άλλα λόγια θα μπορούσε να γίνει διερεύνηση αν θα υπήρχαν αντίστοιχα αποτελέσματα και για ευρύτερες συστοιχίες. Φυσικά, κάτι τέτοιο θα απαιτούσε πολύ περισσότερους πόρους υλισμικού ώστε να ανταποκρίνεται στο γεγονός ότι το σύστημα είναι πρωταρχικά γηγενές στην υπολογιστική νέφους.

Όπως έχει αναφερθεί η συγκεκριμένη εργασία επικεντρώθηκε στην μελέτη ενός κόμβου στο Docker Swarm. Επειδή όμως για να γίνει, όπως προβλέπεται, η πρόσφορη εκμετάλλευση του απαιτούνται τουλάχιστον τρείς κόμβοι, θα ήταν ενδιαφέρον να εξεταστεί η συμπεριφορά του συστήματος σε ένα τέτοιο οικοσύστημα. Ακόμη πιο ενδιαφέρον θα ήταν η διερεύνηση να γίνει με τους κόμβους εγκατεστημένους σε ανομοιογενή φυσικά μηχανήματα.

Επιπρόσθετα και επειδή έχει αναφερθεί πως μια εφαρμογή container μπορεί να εγκατασταθεί και σε συστοιχία Kubernetes για παράδειγμα, θα ήταν ελκυστική πρόκληση να εξεταστεί η συμπεριφορά του συστήματος και στο τελευταίο περιβάλλον ενορχήστρωσης.

9. Παραρτήματα

9.1. Επεξήγηση Συντομογραφιών

Ακρωνύμιο	Επεξήγηση
API	Διεπαφή προγραμματισμού εφαρμογής(Application Programming Interface)
ASLB	Αυτοματοποιημένη κλιμάκωση και εξισορρόπηση Φόρτου(Auto-Scaling Load Balancing)
CPU	Κεντρική Μονάδα επεξεργασίας(Central Processing Unit)
DNS	Σύστημα Ονομάτων Τομέων (Domain Name System)
GUI	Γραφικό Περιβάλλον Διασύνδεσης(Graphical User Interface**)**
HPC	Υπολογιστική Υψηλής Απόδοσης(High Performance Computing)
HTML	Γλώσσα Μορφοποίησης Υπερκειμένου (HyperText Markup Language)
MPI	Διεπαφή Μεταβίβασης Μηνύματος(Message Passing Interface)
PaaS	Πλατφόρμα ως υπηρεσία(Platform as a Service)
SoS	Συστήματος Από Συστήματα (System of Systems)
TLS	Ασφάλεια Επιπέδου Μεταφοράς(Transport Layer Security)
URL	Ενιαίος Εντοπιστής Πόρων (Uniform Resource Locator)

IDE	Ολοκληρωμένο Περιβάλλον Ανάπτυξης(Intergated Development Environment)

9.2. URL σχετικών scripts στο Github

https://github.com/wis-02/aslb.git

10. Βιβλιογραφία

Allspaw, J., & Robbins, J. (2010). *Web Operations: Keeping the Data On Time.* Sebastopol, CA: O'Reilly Media.

Alpine. (2017). *Alpine Linux*. Ανάκτηση από https://alpinelinux.org/about/.

Anicas, M. (2014). *How To Use Apache JMeter To Perform Load Testing on a Web Server*. Ανάκτηση από https://www.digitalocean.com/community/tutorials/how-to-use-apache-jmeter-to-perform-load-testing-on-a-web-server.

appc. (2016). *appc spec*. Ανάκτηση 2017, από https://github.com/appc/spec

Armenise, V. (2015). Continuous Delivery with Jenkins. *IEEE/ACM 3rd International Workshop on Release Engineering.* IEEE.

Barik, R. K., Lenka, R. K., Rao, K. R., & Ghose, D. (2016). Performance analysis of virtual machines and containers in cloud computing. *International Conference on Computing, Communication and Automation (ICCCA).* Noida, India: IEEE.

Bein, D., & Nguyen, N. (2017). Distributed MPI Cluster with Docker Swarm Mode. *Computing and Communication Workshop and Conference (CCWC)l.* Las Vegas, NV, USA: IEEE.

Bittner, C. (2017, Nov 6). *Introducing the IBM Cloud Lite account*. Ανάκτηση από https://www.ibm.com/blogs/bluemix/2017/11/introducing-ibm-cloud-lite-account-2/

Build, Ship, & Run. (2016). *Build, Ship, Run*. Ανάκτηση από https://www.docker.com/sites/default/files/Infographic_OneDocker_09.20.2016.pdf

Butle, B. (2016). *CLOUD CHRONICLES*. Ανάκτηση 2017, από https://www.networkworld.com/article/3030597/cloud-computing/coreos-launches-rkt-the-container-that-s-not-docker.html

cAdvisor. (2015). *cAdvisor (Container Advisor).* Ανάκτηση από https://github.com/google/cadvisor

Coleman, M. (2016). *Docker Swarm Exceeds Kubernetes Performance at Scale*. Ανάκτηση 2017, από https://blog.docker.com/2016/03/swarmweek-docker-swarm-exceeds-kubernetes-scale/

Colton Smith, J. D. (2017). *Load Balancing using Docker Containers*. Ανάκτηση από http://cs.uky.edu/~cwsm229/CS499/.

Colton Smith, R. C. (2017). *University of Kentucky-Load Balancing using Docker Containers*. Ανάκτηση από http://cs.uky.edu/~cwsm229/CS499

Congdon, B. (2017, 04 25). *Deploying Microservices with Diocker*. Ανάκτηση από benjamincongdon.me/blog/: http://benjamincongdon.me/blog/

Docker. (2017). Ανάκτηση από https://docs.docker.com/engine/docker-overview/#docker-objects

Docker Pulls. (2016). *Docker pulls*. Ανάκτηση από https://blog.docker.com/2016/10/introducing-infrakit-an-open-source-toolkit-for-declarative-infrastructure/.

Docker. (2016). *The Docker Survey, 2016*. Ανάκτηση από https://www.docker.com/survey-2016.
Docker, C. E. (2016). *Docker 1.12: Now with Built-in Orchestration! - Docker Blog*. Ανάκτηση από https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/
Docker, T. (2017). *Docker Toolbox*. Ανάκτηση από Docker : https://docs.docker.com/toolbox/toolbox_install_windows/#step-3-verify-your-installation
Docker-Guides. (2017). *How services work*. Ανάκτηση από Docker: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/
Docker-secrets. (2017). *Manage sensitive data with Docker secrets.* Ανάκτηση από https://docs.docker.com/engine/swarm/secrets/
Docker-Stack. (2017). *Deploy a stack to a swarm.* Ανάκτηση από Deploy a stack to a swarm: https://docs.docker.com/engine/swarm/stack-deploy/
Docker-Swarm. (2017). Ανάκτηση από https://docs.docker.com/engine/swarm/
Docker-Swarm. (2017). Ανάκτηση από https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes
Docker-UCP. (2016, February). Ανάκτηση από Docker Reference Architecture: Universal Control Plane 2.0 Service Discovery and Load Balancing: https://success.docker.com/article/Docker_Reference_Architecture-_Universal_Control_Plane_2.0_Service_Discovery_and_Load_Balancing
Dordal, P. L. (2017). Mininet and Pox. http://pld.cs.luc.edu/courses/351/sum16/notes/mininet_and_pox.html.
ECS, A. (2017). *Amazon ECS*. Ανάκτηση από https://aws.amazon.com/ecs/details/
Facebook. (2017). Facebook feed API. https://developers.facebook.com/docs/graph-api/reference/v2.10/page/feed.
Farcic, V. (2017). *Docker Flow Monitor*. Ανάκτηση από http://monitor.dockerflow.com/tutorial/.
Fowler, M., & Levis, J. (2014). *Microservices*. Ανάκτηση από https://martinfowler.com: https://martinfowler.com/articles/microservices.html
Gilly, K., Juiz, C., & Puigjaner, R. (2011). An up-to-date survey in web load balancing. *Springer Science+Business Media* .
Grafana. (2017). *Grafana.* Ανάκτηση από https://github.com/grafana/grafana
Grillet, A. (2016). *Comparison of Container Schedulers*. Ανάκτηση από https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421
Gupta, A. (2016). *Docker for Java Developers.* O'Reilly.
Gupta, K. (2017, April 26). *Haproxy vs Nginx: Which Software Load Balancer Is Better?* Ανάκτηση 2017, από https://www.freelancinggig.com/blog/2017/04/26/haproxy-vs-nginx-software-load-balancer-better/
HAproxy. (n.d.). *HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer*. Ανάκτηση 2017, από http://www.haproxy.org/
Hassen, A. (2015). A survey of Docker Swarm scheduling strategies.

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., και συν. (2011). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *8th USENIX Symposium on Networked Systems Design.*

Hogg, S. (2014, May 26). *Network World*. Ανάκτηση 2017, από Software Containers: Used More Frequently than Most Realize: https://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html

Hope, C. (2017). *Chroot*. Ανάκτηση από https://www.computerhope.com/jargon/c/chroot.htm.

Jetpack. (2017). *github*. Ανάκτηση από https://github.com/3ofcoins/jetpack

Jin-Gang, Y., Ya-Rong, Z., Bo, Y., & Shu, L. (2017). Research and Application of Auto-Scaling Unified Communication Server Based on Docker. Changsha, China: IEEE.

Kane, K. M. (2015). *Docker Up & Running.* Boston: O'Reilly.

Kaur, H., & Jyoti, N. (2017, Jun). Traffic Based Load Balancing in Software Defined Networking. *International Journal on Computer Science and Engineering (IJCSE)* , σσ. 379-384.

Kompose. (2017). *Translate a Docker Compose File to Kubernetes Resources*. Ανάκτηση από kubernetes.io: https://kubernetes.io/docs/tools/kompose/user-guide/

kubernetes. (2017). *github kubernetes*. Ανάκτηση από https://github.com/kubernetes/kubernetes

kubernetes, b.-m. (2017). *kubernetes bare metal*. Ανάκτηση από https://kubernetes.io/docs/setup/#bare-metal

Kurma. (2017). *github*. Ανάκτηση από https://github.com/apcera/kurma

Lefler, L. (2014). *The New Stack Makers: Docker Creator Solomon Hykes*. Ανάκτηση 2017, από the new stack: https://thenewstack.io/the-new-stack-makers-docker-creator-solomon-hykes/

Li, W., & Kanso, A. (2015). Comparing Containers versus Virtual Machines for Achieving High Availability. *International Conference on Cloud Engineering (IC2E.* IEEE .

Liu, D., Shang, W., Zhu, L., & Feng, D. (2016). An Improved Dynamic Load-balancing Model. *4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence* (σσ. 337 - 341). Las Vegas, NV, USA: IEEE.

Lua-Prometheus. (2017). *Prometheus metric library for Nginx.* Ανάκτηση από Prometheus metric library for Nginx: https://github.com/knyar/nginx-lua-prometheus

Maluk, M. M., & M.A., S. (2015, JUNE). Efficient server load balancing through improved server health report. *ARPN Journal of Engineering and Applied Sciences* .

Martínez, Q. (2017). *Working with Docker*. Ανάκτηση από Microsoft/HealthClinic.biz: https://github.com/Microsoft/HealthClinic.biz/wiki/Working-with-Docker

Mouat, A. (2015). *Using Docker.* Beiing-Boston-Tokyo: O'Reilly.

Naik. (2017). Docker Container-Based Big Data Processing in multiple clouds for everyone. *IEEE International Systems Engineering Symposium (ISSE).* Vienna: IEEE.

Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds. *International Symposium on Systems Engineering (ISSE).* Edinburgh, UK: IEEE .

NGINX. (2017). *WHAT IS LOAD BALANCING?* Ανάκτηση 2017, από https://www.nginx.com/resources/glossary/load-balancing/

Nickoloff, J. (2016). *Docker in Action.* Greenwich, CT, USA: Manning Publications Co.

Nomad. (2017). *Nomad* . Ανάκτηση από https://www.nomadproject.io/: https://github.com/hashicorp/nomad

OCI. (2017). *Open Container Initiative*. Ανάκτηση από https://www.opencontainers.org

Patros, P., Dilli, D., Kent, K. B., & Dawson, M. (2017). Dynamically Compiled Artifact Sharing for Clouds. *IEEE International Conference on Cluster Computing*, (σσ. 290-300).

PromDash. (2017). *PromDash is deprecated.* Ανάκτηση από https://github.com/prometheus-junkyard/promdash

Prometheus-Faq. (2017). Ανάκτηση από Prometheus-FAQ: https://prometheus.io/docs/introduction/faq/

Red-Hat. (2015). *RED HAT ENTERPRISE LINUX ATOMIC HOST A platform optimized for Linux containers*. Ανάκτηση από http://fiercesw.com/wp-content/uploads/2016/01/Atomic-Host-Datasheet

rkt. (2017). Ανάκτηση από https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html

Rouse, M. (2017). *Google Container Engine GKE definition*. Ανάκτηση 2017, από http://searchitoperations.techtarget.com/definition/Google-Container-Engine-GKE

Sharma, P., Chaufournier, L., Shenoy, P., & Tay, Y. (2016). Containers and Virtual Machines at Scale: A Comparative Study. *Proceedings of the 17th International Middleware Conference.* Trento, Italy: ACM.

Stoyanov, I. (2016). *Container Management Comparison*. Ανάκτηση από https://github.com/vmware/admiral/wiki/Container-Management-Comparison

Sureshkumar, M., & Rajesh, P. (2017). Optimizing the docker container usage based on load scheduling. *Computing and Communications Technologies (ICCCT), 2017 2nd International Conference on* (σσ. 165-168). Chennai, India: IEEE.

Tolchanov, A. (2017, 06 10). *knyar/nginx-lua-prometheus*. Ανάκτηση από https://github.com/knyar/nginx-lua-prometheus.

Wheatley, M. (2016). *Docker's Swarm smashes Kubernetes in Docker-sponsored benchmark tests*. Ανάκτηση από https://siliconangle.com/blog/2016/03/14/dockers-swarm-smashes-kubernetes-in-docker-sponsored-benchmark-tests/

YAML. (2009). *YAML Ain't Markup Language (YAML™) Version 1.2.* Ανάκτηση από http://www.yaml.org/spec/1.2/spec.html

Yu, H., & Huang, W. (2015). Building a Virtual HPC Cluster with Auto Scaling by the Docke.

Yu, H.-E., & Huang, W. (2015). Building a Virtual HPC Cluster with Auto Scaling by the Docker.

Ankerholz, A. (2016). *https://www.linux.com*. Ανάκτηση 2017, από https://www.linux.com/news/8-open-source-container-orchestration-tools-know.